# Design und Implementierung eines Ethernet-Treibers und TCP/IP-Protokolls für das Java-Betriebssystem JX

Marcus Meyerhöfer

Oktober 2000 SA-I4-2000-16

Studienarbeit

Institut für
Mathematische Maschinen
und Datenverarbeitung
der
Friedrich-Alexander-Universität
Erlangen-Nürnberg

Lehrstuhl für Informatik IV
(Betriebssysteme)

# Design und Implementierung eines Ethernet-Treibers und TCP/IP-Protokolls für das Java-Betriebssystem JX

**Studienarbeit im Fach Informatik**

vorgelegt von

***Marcus Bernd Meyerhöfer***

geb. am 29.06.1976 in Treuchtlingen

Angefertigt am

Institut für Mathematische Maschinen und Datenverarbeitung (IV)
Friedrich-Alexander-Universität Erlangen-Nürnberg

| | |
|---|---|
| Betreuer: | ***Prof. Dr. F. Hofmann*** |
| | ***Dipl.-Inf. Michael Golm*** |

| | |
|---|---|
| Beginn der Arbeit: | 01. Juli 2000 |
| Abgabe der Arbeit: | 27. Oktober 2000 |

Ich versichere, daß ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und daß die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, den 26.10.2000 ______________________________

# Kurzfassung

Heutige Betriebssysteme bieten eine Unterteilung der Ausführungsumgebung in einen nichtprivilegierten Teil für Anwendungen sowie einen privilegierten Teil, in dem der Betriebssystemkern ausgeführt wird. Diese Zweiteilung wird durch spezielle Hardwarekonzepte u.a. des Prozessors unterstützt. Dabei macht diese Aufteilung eine Erweiterung des Kerns durch viele implizite Abhängigkeiten und auch durch die Komplexität der Komponenten sehr schwierig. Das JX-System bietet einen Ansatz, diese Problematik zu lösen, wobei hier die Sicherheit der Ausführungsumgebung durch die Verwendung einer sicheren Sprache garantiert wird - Java.

In dieser Studienarbeit wurde für das JX-System ein Treiber in Java implementiert, um Netzwerkkarten der Serie 3C90x sowie 3C90xB des Herstellers 3COM betreiben zu können. Darauf aufbauend sollte ein einfacher Netzwerkstack implementiert werden, der die Möglichkeit bietet, Daten über das "User Datagram Protocol - UDP" bzw. über das "Transmission Control Protocol - TCP" mit einem anderen Rechner auszutauschen.

Diese Studienarbeit gibt einen Überblick über die Möglichkeiten der verwendeten Netzwerkkarte sowie ihrer Programmierung und beschreibt ausführlich die theoretischen Grundlagen der implementierten Protokolle. Im zweiten Teil wird der interne Aufbau sowie die Verwendung der Implementierung beschrieben. Eine Bewertung der erzielten Ergebnisse schließt diese Arbeit ab.

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# 1 Einführung

## 1.1 Zielsetzung und Rahmenbedingungen

Diese Studienarbeit wurde im Rahmen eines größeren Projekts erstellt und behandelt den Netzwerkteil innerhalb des Projekts JX. Die Aufgabe dieser Arbeit war die Implementierung eines Treibers für die Netzwerkkarte 3C905B-TX der Firma 3COM. Darauf aufbauend sollte ein Netzwerkstack entwickelt werden, der die grundlegenden Möglichkeiten anbietet, um in einem Netzwerk mit anderen Rechnern über die Protokolle UDP und TCP kommunizieren zu können. Dabei konnte auf einige Klassen aufgebaut werden, die Methoden zum Empfang und zum Versand von IP-Paketen (Internet Protocol) zur Verfügung stellen.

Das JX-System ist der Nachfolger des metaXa-Systems, unter welchem die Studienarbeit begonnen wurde. JX ist eine adaptierbare Java-Betriebssystemarchitektur, welche zwei Probleme heutiger Betriebssystemarchitekturen zu lösen versucht. Einerseits die schlechte Adaptierbarkeit des privilegierten Bereichs und andererseits das Problem der Komplexität von Betriebssystemkomponenten. Der Großteil heutiger Systeme bietet eine privilegierte sowie eine nichtprivilegierte Ausführungsumgebung an, wobei die privilegierte Ausführungsumgebung, in der üblicherweise der Betriebssystemkern selbst sowie beispielsweise Hardwaretreiber ablaufen, durch entsprechende Hardwareunterstützung realisiert wird (beispielsweise Stufen von Privilegien bei der Protected Mode Adressierung des Pentium Prozessors [Hau97]). Solch eine Zweiteilung der Ausführungsumgebung erschwert eine Erweiterung des Kerns, zumal oft Abhängigkeiten zwischen verschiedenen Teilen des Kerns bestehen. Das JX-System verwendet zur Lösung dieses Problem den Ansatz der software-basierten Sicherheit, d.h. es wird keine Sicherheit durch den Ablauf von Programmen in bestimmten Adreßbereichen implementiert, sondern durch die Verwendung einer sicheren Sprache. Dabei erlaubt dieses System, Komponenten in das System zu integrieren, welchen man nicht aufgrund ihrer Herkunft oder einer digitalen Signatur trauen kann. Dabei muss das System derart aufgebaut sein, dass eine möglicherweise fehlerhafte oder sogar bösartige Komponente nicht die Möglichkeit hat, die Systemstabilität negativ zu beeinflussen. Die einzige Auswirkung darf das Verhalten der Dienste sein, die diese Komponente anbietet, welche sich dann nicht wie erwartet verhalten bzw. fehlerhaft sind [Gol00]. Es wird hierbei die Sprache Java verwendet und alle Teile der Studienarbeit, auch die des Netzwerkkartentreibers, wurden somit ebenfalls in dieser Sprache programmiert. Ein Überblick über die Sprache Java wird in dieser Arbeit nicht gegeben, da die Konzepte dieser Sprache meist bekannt sind und es ferner viel Literatur zu dieser Programmiersprache gibt, siehe beispielsweise [Sun] oder [Eck98], die eine umfassende Einführung bieten. In JX liegen somit die verschiedenen Betriebssystemkomponenten in Java-Bytecode vor und der statische Betriebssystemkern kann sehr klein gehalten werden. Er enthält eine Laufzeitumgebung für Java, wobei eine Übersetzung des Bytecode in Maschinencode durchgeführt wird [Kopp98]. In JX werden die Ausführungsumgebungen durch sogenannte *Domains* getrennt, wobei jede dieser Domains einen eigenen Heap sowie eigene Threads besitzt. Die Kommunikation zwischen den Domains erfolgt über vom System speziell behandelte Objektreferenzen, sogenannte *Portale*. Die verschiedenen Domains können jeweils einen eigenen Garbage Collector für ihren Speicher besitzen, d.h. jeweils verschiedene Strategien oder Implementierungen verwenden.

Jedem Portal einer Domain ist ein Thread zugeordnet, auf den die Umschaltung bei Aufruf einer Domain erfolgt. Somit ergibt sich auch eine Unabhängigkeit der Threads verschiedener Domains, wodurch beispielsweise verschiedene Domains verschiedene Scheduler verwenden können. Das Scheduling zwischen den Domains erfolgt dabei vollständig preemptiv. Der Zugriff auf die Hardware des Rechners erfolgt durch eine spezielle Domain, die sogenannte *DomainZero*. Dies ist auch die einzige Domain, die in C bzw. Assembler implementiert ist. Durch ihre exportierten Portale können andere Domains dann die Dienste dieser Domain benutzen.

Das JX-System unterscheidet vier Arten von Betriebssystemkomponenten: Interfacekomponenten, Dienstkomponenten, Domainkomponenten sowie Bibliotheken. Dabei besteht jede dieser Komponenten aus einer Menge von Klassen oder Interfaces. Dienstkomponenten stellen einen Dienst zur Verfügung, wie beispielsweise der Netzwerktreiber einen Dienst zur Ansteuerung der Netzwerkkarte zur Verfügung stellt. Dienste werden über die Portale angesprochen. Deren Interfaces sind wiederrum in einer Interfacekomponente definiert. Eine Domain beinhaltet normalerweise mehrere Dienstkomponenten, welche durch eine Domainkomponente miteinander “verdrahtet” sind. In Bibliotheken werden schließlich wiederverwendbare Klassen zusammengefaßt [GK00].

## 1.2 Kapitelübersicht

Die Arbeit gliedert sich in zwei Hauptteile. Die nächsten zwei Kapitel stellen einen Grundlagenteil dar, während die beiden darauffolgenden Kapitel die Implementierung beschreiben. Im nächsten Kapitel werden die Eigenschaften der verwendeten Netzwerkkarte ausführlich behandelt und einige benötigte Datenstrukturen näher erläutert. Es wird die Registerbelegung der Karte beschrieben und ausführlich in die Programmierung dieser Karte eingeführt. Darüberhinaus wird auf die Serie von Netzwerkkarten, zu der die verwendete Karte gehört, eingegangen und die Unterschiede bezüglich Funktionalität und Programmierung werden angesprochen.

Das dritte Kapitel behandelt die verwendeten Netzwerkprotokolle. Beginnend mit einem kurzen Abriß der beiden bekannten Referenzmodelle, dem ISO/OSI-Modell sowie dem TCP/IP-Modell, werden die verschiedenen in der Studienarbeit implementierten bzw. benötigten Protokolle behandelt. Über IP (Internet Protocol), ARP (Address Resolution Protocol) und UDP (User Datagram Protocol) wird nur ein kurzer Überblick gegeben und jeweils auf das Paketformat eingegangen, da diese Protokolle in der Funktionsweise gegenüber TCP einfacher ausfallen und auch schon teilweise als Java-Klasse vorlagen (z.B. IP). Der Großteil des dritten Kapitels behandelt das TCP Protokoll. Dabei wird dieses Protokoll sehr ausführlich behandelt, wobei neben den Grundlagen auch erst in späteren Ergänzungen der Protokollspezifikation hinzugekommende Möglichkeiten wie z.B. SACK (Selective Acknowledgment) oder RTTM (Round trip time measurement) behandelt werden.

Das vierte Kapitel beschreibt die Implementierung des Treibers für die Netzwerkkarte. Dabei wird sowohl ein Überblick über die erstellten Klassen mit den jeweils angebotenen Methoden als auch eine Darstellung bestimmter Abläufe im Treiber, wie die Initialisierung der Karte, der Ablauf beim Versand von Daten oder die Interruptbehandlung, gegeben.

Das fünfte Kapitel erläutert den darauf aufbauenden Netzwerkstack. Beginnend mit einem Überblick über die wichtigsten Klassen wird auf das Konzept der Format-Klassen eingegangen. Es werden die Implementierungen aller Protokollschichten näher betrachtet, wobei auch die internen Abläufe bei z.B. Datenempfang behandelt werden. Eine Beschreibung der Benutzerschnittstelle sowie einige Übersichtsgrafiken runden das fünfte Kapitel ab.

Die Arbeit schließt mit einer Bewertung des Treibers und des Netzwerkstacks und gibt Anregungen für Erweiterungen bzw. Ergänzungen.

## 1.3 Anmerkung zur Textformatierung

In dieser Arbeit werden zur besseren Hervorhebung gegenüber dem Text die Bezeichnungen von Registern bzw. spezieller Bits in Registern durch die Formatierung in `Courier` hervorgehoben. Ebenso werden Klassen, die in dieser Arbeit implementiert wurden bzw. auf welche die Arbeit aufbauen konnte, in `Courier` formatiert. Methodennamen, die Namen der Befehle zur Steuerung der Netzwerkkarte, Einträge in Protokollköpfen (“Header”) und die Bezeichnungen der Interrupts werden in *kursiv* gesetzt.

## 1.4 Zusammenfassung

Es wurden die Rahmenbedingungen und Zielsetzungen dieser Arbeit erläutert. Dabei wurde ein kurzer Überblick über das JX System gegeben, welches das Rahmenprojekt für diese Arbeit darstellt. Schließlich wurde in der Kapitelübersicht eine Übersicht über den Aufbau dieser Arbeit gegeben.

# 2 Die 3COM 3C90x- und 3C90xB-Netzwerkkarten

## 2.1 Überblick

In diesem Kapitel wird ein Überblick über die in der Studienarbeit verwendete Netzwerkkarte gegeben. Das verwendete Modell, eine 3C90xB-TX Karte, gehört zur Serie der 3C90xB-Netzwerkkarten von 3COM. Diese Serie von Karten hat viele Ähnlichkeiten mit den Vorgängermodellen der 3C90x-Serie, was sich auch in der Programmierung auswirkt. Im Prinzip kann ein Treiber eine Vielzahl der verschiedenen Modelle beider Serien unterstützen. Auf dem Markt werden diese Karten unter dem Namen EtherLink XL und Fast EtherLink XL angeboten, die Bezeichnungen 3C90x und 3C90xB sind Abkürzungen für die verschiedenen Karten einer Modellreihe, die sich jeweils noch durch ein spezielles Kürzel unterscheiden (z.B. 3C900B-TPO oder 3C905B-TX). Die verschiedenen Karten unterscheiden sich jeweils in spezielleren Punkten wie z.B. den angebotenen Anschlußmöglichkeiten (normaler RJ-45 Anschluß oder aber BNC, Karten für Glasfaseranschluß, usw.) und dadurch natürlich auch in ihren Leistungsmerkmalen (z.B. kann nur bei bestimmten Medien eine Auto-Negotiation durchgeführt werden). Ferner ist der jeweilige strukturelle Aufbau der Karten voneinander verschieden, jedoch ist das Konzept für den Programmierer immer gleich und die grundsätzlichen Möglichkeiten der Karten sind ebenso gleich. Deswegen wird in der Beschreibung immer allgemein über Karten der beiden Serien gesprochen, wobei auch auf die Unterschiede zwischen den älteren Karten der Serie 3C90x sowie den neueren Karten der Serie 3C90xB eingegangen wird. Nur bei sehr speziellen Dingen wie z.B. der Architektur der Netzwerkkarte wird exemplarisch das verwendete Modell 3C90xB-TX besprochen.

Die Reihe der 3C90x- und 3C90xB-Netzwerkkarten ist eine moderne Serie von NICs (Network Interface Cards), die die Entwicklung eines Treibers dadurch erleichtern, dass sie eine Menge von Konfigurationsmöglichkeiten bieten und somit dem Entwickler nicht vorschreiben, wie bestimmte Dinge gehandhabt werden müssen (siehe z.B. Verwaltung des Speichers für Empfangen und Senden oder die Möglichkeiten der Interruptbehandlung bzw. Interrupterzeugung in Kap. 2.5 und Kap. 2.6). Außerdem können z.B. die Karten der Serie 3C90xB das TCP-Checksumming übernehmen und bei allen Karten, auch den Modellen der 3C90x-Serie, ist die Verwaltung des Speichers sehr bequem, da nur bestimmte Verwaltungsstrukturen vom Treiber übernommen werden müssen, aber die eigentlichen Speichertransfers komplett von der Karte selbst abgewickelt werden. Hierbei müssen die zu übertragenden Daten nicht in einem zusammenhängenden Speicherbereich liegen, sondern die Karte “sammelt” quasi selbst die entsprechenden Fragmente zusammen (sogenanntes “multipacket, multifragment gather/scatter”).

Die Karten der 3C90xB-Serie besitzen darüber hinaus noch einige weitere Möglichkeiten. So gibt es getrennte FIFO-Puffer sowohl für den Empfang als auch für das Senden (je 2KB groß), wobei das Verhalten der Karten im Bezug auf die Pufferspeicher genau konfiguriert werden kann. Die Karten bieten außerdem die Möglichkeit, das Versenden von Paketen zeitlich zu steuern (sogenanntes “scheduling”), das heißt, dem Paket kann innerhalb gewisser Spannen ein Zeitstempel mitgegeben werden, der bestimmt, wann das Paket versendet wird. Der Umgang mit Multicast-Paketen wird durch einen Hash-Filter besser unterstützt, indem genauer bestimmt werden kann, welche Pakete empfangen werden sollen und welche nicht. Außerdem sind Powermanagementfunktionen integriert und es besteht die Möglichkeit, entsprechend vorbereitete Rechner durch spezielle Netzwerkpakete aus einem "Schlafzustand aufzuwecken" (sogenanntes "remote wakeup"). Diese Funktionen (Powermanagement, "remote wakeup") fanden in dieser Studienarbeit jedoch keine weitere Beachtung.

## 2.2 Architektur der 3C905B-TX Karte

Wie schon oben erwähnt, werden spezielle Eigenschaften, die nicht alle Karten besitzen, an der in der Studienarbeit verwendeten 3C905B-TX Karte erläutert. Ein Punkt, in dem sich die Karten sehr stark unterscheiden, ist die Architektur der Karte. Hierbei sind sogar Unterschiede bei dem selben Kartenmodell möglich, wie z.B. bei den 3C905B-TX Karten, da diese Karte mit drei verschiedene ASICs (Application Specific Integrated Circuit - ein anwendungsspezifischer Schaltkreis, der für eine spezielle Anwendung entworfen wird; hier ein Netzwerkchip) im Handel ist. Diese unterschiedlichen ASICs sind bei diesem Modell jedoch funktional identisch bis auf denjenigen Teil, welcher den Zugriff auf den PHY (Physical Connector) ermöglicht: dieser Funktionsteil ist bei jedem ASIC anders und somit unterscheidet sich auch die Registerbelegung des jeweiligen MII-Teils (Medium Independent Interface - siehe Kap. 2.9).

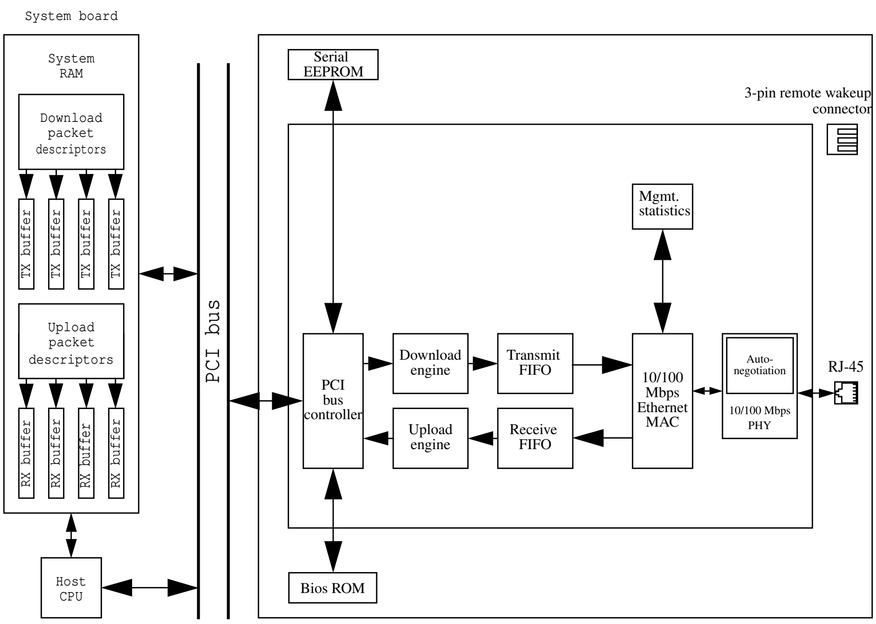


**Abb. 2.1 Architektur der 3C905B-TX Karte (aus [3CO98])**

In Abb. 2.1 ist die Struktur der Netzwerkkarte im Rechnerumfeld grafisch dargestellt. Der linke Teil der Grafik (mit dem PCI-Bus als Trennelement) wird in Kap. 2.5 behandelt und wird hier nicht weiter betrachtet. Rechts in der Grafik ist die Karte dargestellt mit dem ASIC als zentralem

Element. Der ASIC kann in verschiedene Funktionseinheiten unterteilt werden, die als kleine Rechtecke innerhalb des großen Rechtecks dargestellt sind. Pfeile geben die Richtung von Datenströmen an: so kann beispielsweise der PCI-Bus Controller lesend und schreibend auf den Bus zugreifen und über ihn werden Daten an die “Download engine” übergeben, welche dann in dem Transmit FIFO gespeichert werden. Die einzelnen Funktionseinheiten werden im folgenden näher erläutert:

- PCI-Bus Controller: hier sind die PCI Interface Funktionen implementiert, wie das Erzeugen von PCI-Master-Zyklen und das Antworten auf PCI-Target-Zyklen. Für die Sende- und Empfangseinheiten werden Busmaster-Dienste angeboten und auch die Kontrolle des BIOS-ROM wird von diesem Teil übernommen.

- Speichertransfereinheiten (“download engine and upload engine”): hier werden die Deskriptorpakete verarbeitet und die Daten mittels PCI Busmaster-Zyklen, die über den PCI-Bus Controller zur Verfügung gestellt werden, übertragen. Die Datenübertragung findet hierbei zwischen Hauptspeicher und Karte statt.

- Sende- und Empfangs FIFOs (“transmit FIFO and receive FIFO”): dies sind sehr schnelle Pufferspeicher. Auf Karten der 3C90x-Serie ist 8KB Speicher vorhanden, der für Senden und Empfangen aufgeteilt werden kann. Auf den neueren 3C90xB Karten sind jeweils 2KB für das Senden und 2KB für das Empfangen getrennt vorhanden.

- 10/100 Mbps Ethernet MAC: dieser Teil implementiert die Funktionen für den Zugriff auf das physikalische Medium nach der Norm IEEE 802.3. Das Zugriffsprotokoll beinhaltet unter anderem die Kollisionserkennung auf dem Netz, das Warten bis zum erneuten Senden, das Filtern empfangener Pakete sowie Fehlererkennung. Außerdem wird Information an die Management Statistics Einheit (siehe hierzu Kap. 2.7) weitergeleitet.

- Management Statistics: hier werden verschiedene Statistiken gesammelt, wie die Zahl versandter Pakete, Kollisionen usw. Es ist Aufgabe des Treibers diese Daten auszulesen und zu verwenden.

- Auto-Negotiation: hier werden Funktionen zur Verfügung gestellt, um anderen Einheiten im selben Netzwerksegment die Signalfähigkeiten mitzuteilen und eine gemeinsame Übertragungsgeschwindigkeit automatisch auszuhandeln (siehe hierzu Kap. 2.9). Die Auto-Negotiation funktioniert hierbei nur für Twisted-Pair Verkabelung, auf anderen Medien kann keine automatische Einstellung der Geschwindigkeit erfolgen. Darüber hinaus ist dieser Block vom verwendeten ASIC abhängig. Diese Funktionen sind entweder als eigenständiger Teil ausgelagert oder schon im 10/100 Mbps PHY integriert.

- 10/100 Mbps PHY: dieser Teil ist nur auf 3C90xB Karten enthalten und hier nur bei 2 bestimmten ASICs. Dieser Teil integriert den 10Base-T/AUI, die 100 Mbps Signalisierung und die Autonegotiation-Einheit, die auf älteren Modellen einzelne Einheiten waren.

- MII-Control Logik: hier sind Funktionen enthalten, um die Media Independent Interface Spezifikation zu erfüllen (siehe hierzu Kap. 2.9). Durch dieses Interface kann der Netzwerkchip mit jedem Chip für die Erzeugung der physikalischen Signale zusammenarbeiten, der dieses Interface ebenfalls unterstützt. Je nach ASIC kann dieser Teil schon wiederum im 10/100 Mbps PHY integriert sein [3CO98].

## 2.3 Das EEPROM

Die Netzwerkkarten besitzen ein serielles EEPROM, dass je nach Kartenserie eine unterschiedliche Anzahl von Einträgen beinhaltet. Im EEPROM sind einige für die Konfiguration der Karte notwendige Daten wie die Ethernet-Adresse oder die PCI-DeviceID der Karte abgespeichert. Zusätzlich sind Informationen wie das Herstellungsdatum der Karte, Daten über unterstützte Betriebsmodi wie z.B. BusMaster-Zugriffe oder Powermanagementfunktionen und Informationen über behobene Hardware-Bugs enthalten.

Der Zugriff auf das EEPROM erfolgt über zwei Register: Mittels des `EepromCommand` Registers wird das gewünschte Register adressiert und die Art des Zugriffs festgelegt (Schreiben/Lesen/Löschen) und mittels `EepromData` können dann die Daten gelesen bzw. geschrieben werden.

## 2.4 Die Register

Die Steuerung der Netzwerkkarte wird über die Register der Karte abgewickelt. Hierbei unterscheidet sich das Registerlayout zwischen Karten der Serie 3C90x und Karten der Serie 3C90xB. Bei der 3C90x Serie werden die Register auf 64 Byte des I/O-Adreßraums der CPU des Rechners abgebildet, während bei der 3C90xB Serie die Register auf 128 Byte des I/O-Adreßraums oder auf den normalen Speicheradreßbereich abgebildet werden. Es ist auch möglich, es auf beide Adreßräume gleichzeitig abzubilden bzw. über beide Verfahren auf die Karte zuzugreifen.

Die Register sind in acht verschiedene Registerbänke aufgeteilt. Hierbei bilden die ersten 16 Byte des “Registermappings” ein Fenster in eines der acht Fenster, zwischen denen mittels eines speziellen Kommandos umgeschaltet wird. Die restlichen Byte sind dagegen immer sichtbar.

Der Zugriff auf die Register erfolgt als Offset vom `IoBaseAdress` Register oder, falls das Register in einem Registerfenster liegt, durch die Angabe der Fensternummer und einem Offset innerhalb dieses Fensters.

Beim Zugriff auf die Register ist auf die Breite des jeweiligen Registers zu achten. Ein Zugriff darf im allgemeinen nur in der Breite des Registers erfolgen, d.h., falls beispielsweise zwei 16 Bit breite Register nebeneinander liegen, darf dennoch nicht mit einem einzigen 32-Bit Zugriff darauf zugegriffen werden.

Ein spezielles Paar von Registern sind das `Command` Register und das `IntStatus` Register. Beide Register liegen an derselben Position (“gleicher Offset”), wobei die Register in jedem Register-Fenster eingeblendet sind. Während in das `Command` Register nur geschrieben werden kann, kann aus dem `IntStatus` Register nur gelesen werden. Im Prinzip ist dieses Registerpaar ein einzelnes Register, welches durch die Art des Zugriffs unterschieden wird.

Über das `Command` Register werden die entsprechenden Befehle an die Netzwerkkarte abgesetzt, wie z.B. das Ein- oder Ausschalten der Empfangs- und Sendeeinheiten. Befehle sind dabei normale Codes (also z.B. 32 Bit), die unter Umständen noch Parameter enthalten können (siehe Kap. 2.8).

Das `IntStatus` Register wird dagegen vom Treiber verwendet, um feststellen zu können, was der Grund für einen von der Karte ausgelösten Interrupt ist. Darüber hinaus kann über ein Bit auch festgestellt werden, ob ein an die Karte abgesetzter Befehl sich noch in Bearbeitung befindet (siehe Kap. 2.6).

| Byte 3 | Byte 2 | Byte 1 | Byte 0 | Offset |
|---|---|---|---|---|
| | | PowerMgmtCtrl (write only) | | 7c |
| UpMaxBurst | | DnMaxBurst | | 78 |
| | | DebugControl | | 74 |
| DebugData | | | | 70 |
| | | | | 6c - 44 |
| RealTimeCnt | | | | 40 |
| | UpBurstThresh | UpPoll | UpPriorityThresh | 3c |
| UpListPtr | | | | 38 |
| Countdown | | FreeTimer | | 34 |
| UpPktStatus | | | | 2c |
| | | DnPoll | DnPriorityThresh | 30 |
| | DnBurstThresh | | | 28 |
| DnListPtr | | | | 24 |
| DmaCtrl | | | | 20 |
| IntStatusAuto | | | | 1c |
| TxStatus | Timer | | TxPktId | 18 |
| | | | | 14 |
| | | | | 10 |
| Register windows 0 through 7 | | | | c |
| | | | | 8 |
| | | | | 4 |
| | | | | 0 |

**Abb. 2.2 Registerlayout der Serie 3C90xB (aus [3CO98])**

In Abb. 2.2 ist das Registerlayout der Karten der Serie 3C90xB dargestellt. In den unteren Bereich von 0x0 bis 0xc wird eines von acht Registerfenstern eingeblendet. Das bedeutet, um auf ein spezielles Register in einem Fenster zugreifen zu können, muss erst auf das entsprechende Registerfenster umgeschaltet werden, dann kann auf es ganz normal im Bereich 0x0 bis 0xc zugegriffen werden. Die grauen Bereiche sind “ungültige” Bereiche, in denen derzeit noch keine

Register liegen. Im Vergleich zur Serie 3C90x ist das Registerlayout wesentlich umfangreicher geworden, da bei der alten Serie außer dem Registerfenster nur noch einige wenige andere Register vorhanden sind.

In Abb. 2.3 sind alle Register aus den Registerfenstern dargestellt. Diese Grafik soll als Übersicht dienen, um einen Überblick über die in den weiteren Kapiteln angesprochenen Register zu geben.

| Byte 3 | Byte 2 | Byte 1 | Byte 0 | Offset | Window |
|---|---|---|---|---|---|
| IntStatus/Command | | PowerMgmtEvent | | c | 7 |
| | | | | 8 | |
| | | VlanEtherType | | 4 | |
| | | VlanMask | | 0 | |
| IntStatus/Command | | BytesXmittedOk | | c | 6 |
| BytesRcvdOk | | UpperFramesOK | FramesDeferred | 8 | |
| FramesRcvdOk | FramesXmittedOk | RxOverruns | LateCollisions | 4 | |
| SingleCollisions | MultipleCollisions | SqeErrors | CarrierLost | 0 | |
| IntStatus/Command | | IndicationEnable | | c | 5 |
| InterruptEnable | | TxReclaimThresh | RxFilter | 8 | |
| RxEarlyThresh | | | | 4 | |
| | | TxStartThresh | | 0 | |
| IntStatus/Command | | UpperBytesOk | BadSSD | c | 4 |
| MediaStatus | | PhysicalMgmt | | 8 | |
| Network Diagnostic | | FifoDiagnostic | | 4 | |
| VcoDiagnostic (not supported) | | | | 0 | |
| IntStatus/Command | | TxFree | | c | 3 |
| RxFree | | MediaOptions | | 8 | |
| MacControl | | MaxPktSize | | 4 | |
| InternalConfig | | | | 0 | |
| IntStatus/Command | | ResetOptions | | c | 2 |
| StationMask (Hi) | | StationMask (Mid) | | 8 | |
| StationMask (Lo) | | StationAddress (Hi) | | 4 | |
| StationAddress (Mid) | | StationAddress (Lo) | | 0 | |
| IntStatus/Command | | | | c | 1 |
| | | | | 8 | |
| | | | | 4 | |
| | | | | 0 | |
| IntStatus/Command | | EepromData | | c | 0 |
| EepromCommand | | | BiosRomData | 8 | |
| BiosRomAddr | | | | 4 | |
| | | | | 0 | |

**Abb. 2.3 Die Registerfenster der Serie 3C90xB (aus [3CO98])**

## 2.5 Die Funktionsweise der Netzwerkkarte

Um zu versendende Daten an die Karte zu übergeben als auch beim Empfang von Daten aus dem Netzwerk wird der Transfer zwischen der Netzwerkkarte und dem Hauptspeicher des Computers über spezielle Datenstrukturen, sogenannte Deskriptoren, gesteuert. Diese Deskriptoren werden im Hauptspeicher des Rechners erzeugt und ihre Startadresse wird an die Karte übergeben, welche dann diese verarbeitet.

Der Datentransfer selbst wird von der Karte mit PCI Busmaster Operationen durchgeführt. Auch zum Lesen der Deskriptoren und zum Schreiben von Informationen in die Deskriptoren selbst werden Busmaster Operationen verwendet.

Im RAM des Rechners werden vom Treiber zwei Listen dieser Deskriptoren verwaltet. Eine sogenannte “Downlist”, die für das Versenden von Paketen zuständig ist und eine “Uplist”, die für vom Netzwerk empfangene Pakete benötigt wird. Die Bezeichnungen rühren daher, dass der Transfer von zu sendenden Daten an die Karte als “Download” bezeichnet wird. Dies führt die Karte mit Hilfe der Deskriptoren selbst durch. Äquivalent werden empfangene Daten von der Karte in den Hauptspeicher transferiert, was als “Upload” bezeichnet wird.

Diese beiden Listen werden vom Treiber erzeugt und verwaltet. Verarbeitet werden sie von der Netzwerkkarte. Dabei werden das Empfangen und das Senden von Daten durch das Schreiben von zwei Registern gestartet. In das `DnListPtr` Register wird die Adresse des ersten “down packet descriptors” (DPD) geschrieben, in das `UpListPtr` Register die Adresse des ersten “upload packet descriptors” (UPD).

### 2.5.1 Der Download

Wie schon erwähnt, wird der Transfer der zu versendenden Daten vom Hauptspeicher des Computers zur Netzwerkkarte über eine Liste von Deskriptoren gesteuert. Einen schematischen Aufbau dieser Liste, der sogenannten “Downlist”, zeigt Abb. 2.4.

Diese Liste besteht aus Download Deskriptoren, die miteinander verkettet sind. Um ein Paket zu versenden, wird dieses im Hauptspeicher erzeugt. Die Einträge des DPD zeigen nun auf alle Speicherblöcke, die zu übertragen sind. Zum exakten Aufbau dieser Deskriptoren siehe Kap. 2.5.2.

Um Daten vom Hauptspeicher zur Karte zu übertragen, geht man wie folgt vor: es werden DPDs erzeugt, die auf die zu übertragenden Speicherfragmente zeigen und aus diesen DPDs wird wiederum eine verkettete Liste erzeugt (falls mehrere DPDs benötigt werden, um die Daten zu beschreiben). Dies geschieht alles im Hauptspeicher des Computer. Die Adresse des ersten DPDs wird im `DnListPtr` Register eingetragen. Bei der folgenden Verarbeitung ist der Kopf der Liste immer derjenige DPD, der sich gerade in Verarbeitung findet und auf diesen zeigt das `DnListPtr` Register. Das heißt, bei der Verarbeitung wird dieses Register ständig geändert, indem nach Verarbeitung eines Deskriptors der Eintrag im `DnNextPtr` in das Register übernommen wird. Ist dieser Wert null, so geht die entsprechende Funktionseinheit in einen Idle-Zustand über. In Karten der Serie 3C90xB existiert noch eine zweite Methode: hier kann ein sogenanntes “Download-Polling” aktiviert werden. Bei dieser Methode wird, nachdem ein `DnNextPtr` Eintrag aus dem aktuellen DPD gelesen wurde, der den Wert null hat, zwar ebenfalls wie beim normalen Vorgehen in einen Idle-Zustand übergegegangen, aber es wird in regelmäßigen Abständen der aktuelle DPD untersucht. Ändert sich dessen `DnNextPtr` Eintrag, wird die Verarbeitung der Liste wieder angestoßen. Dabei wird der alte Wert im `DnListPtr` Register vorerst nicht überschrieben, da dieses Register auf den aktuellen Deskriptor zeigt, der dann “gepollt” wird. Der Vorteil dieser Methode ist, dass der Download einfacher wieder gestartet werden kann, wenn neue Daten bereit stehen. Es muss nur im Speicher des Rechners ein Eintrag im Deskriptor geändert werden und die Karte bemerkt dies durch das Polling. Beim herkömmlichen Verfahren muss dagegen ein Registerzugriff erfolgen, um einen neuen Wert in das `DnListPtr` Register zu schreiben (für den genauen Aufbau der Deskriptoren und der hier beschriebenen Felder siehe Kap. 2.5.2).

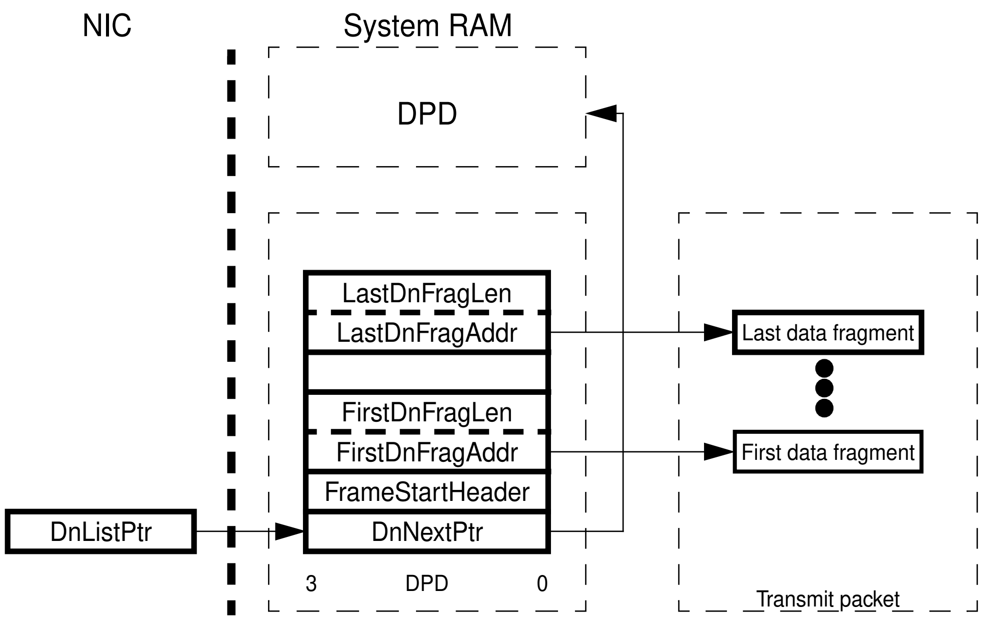


**Abb. 2.4 Aufbau der Download-Liste (aus [3CO98])**

Der Download-Vorgang selbst wird begonnen, sobald ein Wert ungleich null in das `DnListPtr` Register eingetragen wird. Weitere schreibende Zugriffe auf dieses Register werden dann ignoriert, solange ein Wert ungleich null enthalten ist.

### 2.5.2 Das Format der DPDs

Es gibt zwei verschiedene Formate, welche mit Type 0 und Type 1 bezeichnet werden. Das Type 0 Format ist abwärtskompatibel zu Karten der 3C90x Serie. Das Type 1 Format hat gegenüber Type 0 zusätzlich ein Feld für das sogenannte “Scheduling” der Downloads. Unterschieden werden die beiden Formate durch die Bits [31:30] an Position 4. Ferner müssen Type 0 DPDs auf acht Byte Adressen ausgerichtet sein, Type 1 DPDs dagegen auf 16 Byte (d.h. sie müssen ohne Rest durch acht bzw. 16 teilbar sein).

Der grundsätzliche strukturelle Aufbau der beiden Formate ist bis auf das zusätzliche Feld für Download-Scheduling bei Type 1 und der daraus resultierenden Verschiebung der folgenden Einträge jedoch gleich. Allerdings sind je nach Kartenserie einige Unterschiede im Inhalt der Felder vorhanden.

Der erste Eintrag, der `DnNextPtr`, zeigt auf den nächsten Deskriptor oder ist null, falls der Deskriptor selbst der letzte Eintrag in der Liste ist.

Der nächste Eintrag bei Type 0 ist der `FrameStartHeader`, mit dem das Verhalten der Karte für die folgenden Fragmente gesteuert werden kann. So kann beispielsweise das Anhängen einer CRC-Prüfsumme unterbunden werden, oder es kann veranlaßt werden, dass ein Interrupt nach dem Download erzeugt wird. Jedoch haben die Einträge in dieses Feld für Karten der Serie 3C90x eine andere Bedeutung als für die Serie 3C90xB. Der Treiber für die Netzwerkkarte kann jedoch dieses Feld in einer abwärtskompatiblen Weise benutzen. Bei Type 1 DPDs befindet sich an dieser Stelle zuerst der Eintrag für die `ScheduleTime`, mit dem gesteuert werden kann, wann ein Paket zur Karte übertragen wird. Der `FrameStartHeader` folgt darauf.

Nach diesen Einträgen folgen bei beiden Formaten, nur an unterschiedlicher Startposition, bis zu 63*2 Einträge, die Speicherblöcke beschreiben, in welchen sich die zu übertragenden Daten befinden. Hierbei zeigt der erste Eintrag immer auf den Speicherblock, der übertragen werden soll (der Eintrag ist also die physische, nicht die virtuelle Adresse des Speicherblocks), der zweite Eintrag gibt die Länge dieses Speicherblocks an. Im Gegensatz zu den Deskriptoren selbst, bei denen auf ein eine Ausrichtung auf bestimmte Adressen ("Alignment") geachtet werden muss, können die zu übertragenden Daten an einer beliebigen Stelle liegen.

In Abb. 2.5 sind nochmals beide Formate grafisch dargestellt.

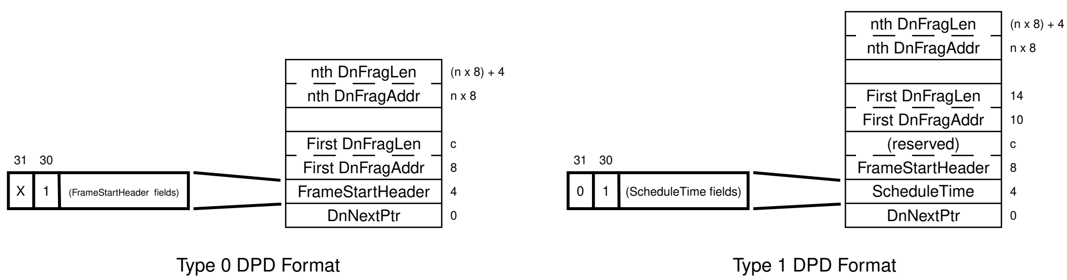


**Abb. 2.5 Die beiden DPD-Formate (aus [3CO98])**

### 2.5.3 Der Download-Prozess

Mit den vorgestellten Mitteln sind viele verschiedene Arten der Übertragung der Daten denkbar. Die einfachste ist sicherlich, dass der Treiber jedesmal, wenn Daten zu übertragen sind, einen neuen DPD (oder fall ein Deskriptor nicht ausreicht, mehrere DPDs) erzeugt und dessen Adresse in das `DnListPtr` Register einträgt. Jedoch ist es sicherlich sinnvoller, dass der Treiber mehrere DPDs verwaltet und diese auch wiederverwendet. Wie der Treiber dies macht, bleibt auch wiederum dem Entwickler überlassen. Eine Möglichkeit ist es, mehrere DPDs zu erzeugen und diese als Liste zu verwalten. In den letzten DPD wird an Stelle des `DnNextPtr` eine null eingetragen. Bei diesem Vorgehen ist natürlich mehr Aufwand nötig als bei der oben beschriebenen Vorgehensweise, weil nun der Treiber und die Karte womöglich gleichzeitig auf dieser Liste arbeiten. Deshalb muss der Treiber die Karte in den sogenannten “Stalled-Modus” versetzen (d.h. die Listenverarbeitung vorübergehend abschalten), bevor er die Liste verändern oder einen neuen Eintrag in das `DnListPtr` Register eintragen will. Um die Karte wieder zu aktivieren, muss danach wieder ein Unstall-Kommando abgesetzt werden (siehe Kap. 2.8).

Bei den Karten der neueren 3C90xB-Serie gibt es jedoch auch die schon in Kap. 2.5.1 dargestellte Möglichkeit, den aktuellen DPD zu “pollen”, das heißt, in regelmäßigen Abständen zu lesen und nach einem neuen Wert zu überprüfen. Der Download startet dann, sobald ein Wert ungleich null in das `DnNextPtr` Feld des DPD eingetragen wird. Dieser Polling-Mechanismus wird auch bei DPDs verwendet, die eine `ScheduleTime` beinhalten. Hier wird dieses Feld “gepollt”, um festzustellen, ob die Übertragung gestartet werden kann oder noch weiter gewartet werden muss.

Ist die Übertragung der Daten eines Deskriptors zur Karte fertig, so schreibt die Karte das `dnComplete` Bit im DPD. Dadurch muss der Treiber nicht das `DnListPtr` Register auslesen, um festzustellen, welche DPDs schon verarbeitet wurden, sondern kann einfach durch die Liste im Hauptspeicher gehen und dieses Bit in jedem Deskriptor auslesen. Dieses Vorgehen ist auch notwendig, wenn man die zweite Möglichkeit verwendet, das Ende eines Transfers zu bestimmen. Die Netzwerkkarte kann so konfiguriert werden, dass sie nach Verarbeitung eines Deskriptors einen Interrupt auslöst. Dies kann pro Deskriptor frei gewählt werden, da dies über einen Eintrag im `FrameStartHeader` des Deskriptors aktiviert wird. Bei der darauf folgenden Interrupt-Behandlung muss der Treiber feststellen können, welche DPDs verarbeitet worden sind, da es möglich ist, dass bis zu dem Zeitpunkt, zu dem der Treiber in den Interrupt-Handler eintritt, schon mehrere Deskriptoren verarbeitet worden sind oder nur bei bestimmten Deskriptoren eine Erzeugung eines Interrupts eingestellt wurde. Der Treiber muss nun nur die Liste durchgehen und kann alle DPDs, deren `dnComplete` Bit gesetzt ist, entsprechend behandeln und kann sie beispielsweise wiederverwenden. Es ist sogar möglich, dass im `FrameStartHeader` das Bit zur Auslösung eines Interrupts während der Bearbeitung des Deskriptors gesetzt wird, da die Karte den `FrameStartHeader` sowohl vor als auch nach der Verarbeitung untersucht.

### 2.5.4 Die eigentliche Übertragung der Daten auf das Netz

Das Paket wird von der Netzwerkkarte versendet, sobald das komplette Paket im Sende-FIFO Puffer enthalten ist oder falls sich bereits mehr Bytes im Puffer befinden, als durch ein entsprechendes Register, das `TxStartThresh` Register, angegeben wurde. Mit Hilfe dieses Registers kann gesteuert werden, ab welchem Füllgrad des Sendepuffers das Versenden der Daten auf das Netzwerk gestartet werden soll.

Zum Senden selbst muss die Sende-Einheit aktiviert worden sein. Nach einem Start des Computers ist diese deaktiviert, d.h., es können zwar mittels der DPDs Daten an die Karte übertragen werden, welche in den Sende-Puffer kommen, aber es werden keine Daten auf das Netzwerk übertragen. Werden mehr Daten an die Karte übertragen, als der Puffer aufnehmen kann, so tritt ein Puffer-Überlauf auf. Das Aktivieren der Sende-Einheit erfolgt wieder durch einen speziellen Befehl, der in das Kommando-Register geschrieben wird.

Tritt beim Versenden der Daten ein Fehler auf, so wird ein Interrupt ausgelöst und entsprechende Einträge in einem Status-Register (`TxStatus`) gesetzt. Der Treiber muss dann die Sendeeinheit erneut aktivieren und bei einigen Fehlern ist es zusätzlich auch noch nötig, die Sendeeinheit mit einem Reset-Befehl zurückzusetzen. Hierbei wird natürlich auch der Sendepuffer gelöscht, wogegen bei Fehlern, die zur Behandlung keinen Reset benötigen, der Inhalt des Puffers erhalten bleibt und die Daten nach der Fehlerbehandlung übertragen werden.

Die Anzahl der Interrupts, die die Karte erzeugt, um den Treiber anzuzeigen, dass ein Download beendet wurde, kann, wie schon oben erwähnt, beeinflußt werden. So kann nur bei bestimmten DPDs die Erzeugung von Interrupts aktiviert werden oder das Bit wird nur beim letzten DPD in der Liste gesetzt, so dass ein Interrupt ausgelöst wird, wenn die ganze Liste abgearbeitet wurde. Dabei muss dann natürlich darauf geachtet werden, dass dieses Bit wieder auf null gesetzt wird, falls an das Ende der Liste ein weiterer DPD angefügt wird. Es gibt also verschiedene Möglichkeiten, dies zu steuern, wobei es dem Entwickler überlassen bleibt, zwischen der Reaktionszeit einerseits und der Anzahl der Interrupts andererseits abzuwägen. Bei den Karten der Serie 3C90xB gibt es darüber hinaus noch eine weitere Möglichkeit. Hier kann anstatt des `dnComplete` Bits ein spezielles Register, das `CountDown` Register verwendet werden. Dieses Register ermöglicht es, zeitgesteuert einen Interrupt auszulösen, indem man die Zeit, zu welcher der Interrupt ausgelöst werden soll, in dieses Register einträgt. Der Treiber wird nun den *dnComplete* Interrupt maskieren und stattdessen einen Interrupt durch das `CountDown` Register auslösen lassen (zu eineren genaueren Erläuterung der Interrupts siehe 2.6). Hierbei kann z.B. geprüft werden, wie lange die Übertragung einer bestimmten Anzahl von Bytes dauert und man kann nun abhängig von der zu sendenden Datenmenge das `CountDown` Register beispielsweise mit der Hälfte dieser Zeit setzen. In der Interruptbehandlung wird dann wieder über das `dnComplete` Bit geprüft, welche DPDs schon verarbeitet wurden.

### 2.5.5 Der Empfang von Daten

Das Konzept, wie empfangene Daten an den Treiber weitergegeben werden bzw. wie sie verwaltet werden, ist dem Verfahren beim Senden von Daten sehr ähnlich. Auch hier wird wie schon oben erwähnt eine Liste von Deskriptoren verwaltet, die von der Karte verarbeitet wird. Diese Liste besteht aus sogenannten "upload packet descriptors" (UPD). Der Vorgang des Datentransfers vom Empfangspuffer der Netzwerkkarte in den Hauptspeicher des Rechners wird analog zum Download mit "Upload" bezeichnet.

Im Gegensatz zu den DPDs gibt es beim Upload nur einen Typ von Deskriptor, welcher analog zu den DPD Type 0 auf eine acht-Byte Adresse ausgerichtet sein muss. Diese Deskriptoren beschreiben in der selben Weise wie die DPDs Speicherbereiche. Im Gegensatz zu den DPDs sind hier natürlich keine Daten enthalten, sondern es sind die vom Treiber reservierten Speicherbereiche, in die empfangene Daten kopiert werden sollen. Ein UPD kann wiederum 63 solcher

Speicherfragmente beschreiben. Die UPDs können wie die DPDs über ihren ersten Eintrag verkettet werden und auf den Anfang der Liste zeigt wieder ein spezielles Register, das `UpListPtr` Register.

Nach einem Systemstart befindet sich die entsprechende Upload-Funktionseinheit ebenso wie die Download-Funktionseinheit in einem Idle-Zustand. Die Verarbeitung einer Liste von UPDs wird durch das Schreiben eines Wertes ungleich null (eben die Adresse des Listenanfangs) angestoßen.

Das zweite Feld eines UPD, der `UpPktStatus`-Eintrag, ist ähnlich dem eines DPD; nur gibt hier nicht der Treiber Anweisungen an die Karte, sondern die Netzwerkkarte kann durch dieses Feld den Treiber über bestimmte Eigenschaften informieren, wie z.B., dass der Upload eines Pakets schon beendet ist oder, dass bestimmte Fehler wie ein Empfangspufferüberlauf oder ein CRC-Fehler aufgetreten sind.

Danach folgen bis zu 63 Paare einer Speicheradresse sowie einer Längenangabe dieses Speicherbereichs. Hiermit werden die Speicherbereiche beschrieben, in die die Karte mittels DMA die empfangenen Daten überträgt.

Schon wie beim Download ist auch der Upload nach einem Reset nicht aktiviert und die Upload-Funktionseinheit muss erst vom Treiber durch einen speziellen Befehl eingeschaltet werden.

Die Benachrichtigung über das Ende eines Uploads geschieht normalerweise über einen *upComplete* Interrupt, was jedoch einstellbar ist. Im entsprechenden Interrupt-Handler kann der Treiber nun anhand des `upComplete` Bits im `UpPacketStatus` Eintrag der UPDs feststellen, welche UPDs benutzt wurden, da ja wie schon im Download-Fall bis zum Eintritt in den Handler mehrere UPDs benutzt worden sein können. Diese Methode ist das bevorzugte Vorgehen, da die andere Möglichkeit, anhand des Wertes des `UpListPtr` Registers denjenigen UPD zu finden, bis zu dem die UPDs benutzt wurden, zu aufwenig ist. Gegenüber dem Verhalten, dass nach jedem UPD ein Interrupt ausgelöst wird, kann die Netzwerkkarte auch derart konfiguriert werden, dass schon nach einer bestimmten Anzahl von empfangenen Bytes ein Interrupt ausgelöst wird. Es ist sogar möglich, während dem Upload der Daten diese schon aus dem Empfangspuffer herauszukopieren, bevor der komplette Upload beendet ist. Dabei kann über ein Register (`UpPktStatus` Register - wenn der Upload jedoch schon fertig ist, dann sollte der `UpPktStatus` Eintrag im UPD gelesen werden) bestimmt werden, wieviele Daten schon empfangen wurden und somit schon verwendet werden können.

Das übliche Vorgehen eines Treibers ist wie schon beim Download das Erzeugen mehrerer UPDs und die Verwaltung dieser in einer Liste, um sie wiederverwenden zu können. Eine Besonderheit beim Upload ist jedoch die Möglichkeit, diese in einem Ring anzuordnen, da die Karte entweder implizit einen speziellen Befehl ausführt (*UpStall*), falls sie auf einen bereits verwendeten UPD stößt oder solange das `upComplete` Bit überprüft (Polling), bis dieses wieder gelöscht wird.

## 2.6 Interrupts

Die Netzwerkkarte kann das System auf bestimmte Ereignisse auf der Karte durch das Auslösen eines Interrupts aufmerksam machen, wobei dann ein entsprechender Interrupt-Handler die Verarbeitung dieses Ereignisses übernehmen muss. Dabei kann ein Interrupt durch verschiedene Gründe ausgelöst werden: bei Karten der 3C90xB Serie gibt es acht (bzw. neun, falls der nur bei einer ASIC-Version implementierte *linkEvent* mitgezählt wird) verschiedene Auslöser für einen Interrupt. Durch das Konzept der Interruptverwaltung der Netzwerkkarte ist hier jedoch eine feinere Unterscheidung notwendig: es gibt die sogenannten "Auslöser" bzw. "Anzeigen", dass etwas eingetroffen ist, über das der Treiber benachrichtigt werden muss. Diese Benachrichtigung kann als Interrupt erfolgen oder aber der Treiber selbst muss in einem Register nachsehen oder es findet keinerlei Benachrichtigung statt, d.h. der auslösende "Event" wird unterdrückt. Das spezifische Verhalten ist durch drei Register konfigurierbar.

Im `IndicationEnable` Register kann für jede Ursache eingestellt werden, ob sie ignoriert werden soll (Bit im `IntStatus` Register wird nicht gesetzt) oder ob das Auftreten eines "Events" als gesetztes Bit im `IntStatus` Register erscheinen soll. Das `IntStatus` Register gibt Auskunft über aufgetretene "Events". Hier kann der Treiber im Interrupt-Handler auslesen, welche Quelle den Interrupt ausgelöst hat. Das Erscheinen eines "Events" als gesetztes Bit im `IntStatus` Register jedoch löst einen Interrupt noch nicht aus. Dies geschieht nur dann, wenn im `InterruptEnable` Register dies für die entsprechende Quelle konfiguriert ist. Durch diese drei Register ist also eine sehr feine Konfiguration möglich: es können manche Dinge komplett ignoriert werden und erscheinen somit nicht im `IntStatus` Register. Für die anderen Quellen wird ein Bit im `IntStatus` Register gesetzt, es kann aber zusätzlich eingestellt werden, welche dieser Quellen dann auch einen Interrupt auslösen soll.

Wurde ein Interrupt ausgelöst, so ist ein Bit, das sogenannte `interruptLatch`, immer gesetzt. Es zeigt einen Interrupt an und soll fehlerhafte Interrupts auf dem Bus verhindern ("spurious interrupts"). Dieses Bit (`interruptLatch`) muss immer explizit durch einen speziellen Befehl gelöscht werden, d.h. der Interrupt muss allgemein durch einen Befehl bestätigt werden. Dann muss noch abhängig von der Ursache des Interrupts das entsprechende Bit (welches die Ursache des Interrupts anzeigt) durch bestimmte Handlungen zurückgesetzt und der Interrupt dadurch bestätigt werden.

Die folgende Tabelle erklärt die neun möglichen Ursachen für einen Interrupt, wobei natürlich wie oben beschrieben festgelegt werden kann, welche im `IntStatus` Register erscheinen und welche von diesen dann einen Interrupt auslösen:

**Tabelle 1: Interrupts**

| Interrupt-Name | Beschreibung |
|---|---|
| *hostError* | Tritt ein katastrophaler Fehler auf dem Bus auf, wird dieses Bit gesetzt; in diesem Fall hilft nur ein kompletter Reset der Karte |
| *txComplete* | Durch ein Bit im `FramestartHeader` eines DPDs kann festgelegt werden, dass nach erfolgreicher Versendung der Daten dieses Bit gesetzt wird und dadurch je nach Konfiguration ein Interrupt ausgelöst wird; außerdem wird es durch einen Fehler bei der Datenübertragung gesetzt |
| *rxComplete* | Dieses Bit wird gesetzt, wenn ein oder mehrere komplette Pakete im Empfangspuffer angekommen sind; im Gegensatz zu den anderen Interrupt-Quellen wird dies automatisch von der Upload-Funktionsseinheit bestätigt; dieses Bit wird normalerweise maskiert und kein Interrupt ausgelöst |
| *rxEarly* | Die Karte kann so konfiguriert werden, dass schon nach einer bestimmten Anzahl empfangener Bytes ein Interrupt ausgelöst wird und nicht erst nach dem vollständigen Empfang; der dadurch ausgelöste Interrupt ist vom Typ *rxEarly* |
| *intRequested* | Es ist möglich, explizit durch ein Kommando oder durch einen Timer auf der Karte einen Interrupt auszulösen; das intRequested-Bit zeigt diesen Fall an |
| *updateStats* | Die Karte verfügt über einige Statistik-Register (siehe Kap. 2.7); nähern sich diese einem Überlauf, so wird dieses Bit gesetzt; der Treiber sollte dann die Statistiken auslesen, was diese automatisch zurücksetzt und diesen Interrupt bestätigt |
| *linkEvent* | Eine Veränderung des Link-Status wird durch dieses Bit angezeigt; allerdings ist dies nur bei einer bestimmten ASIC-Version der 3C90xB-Serie implementiert |

**Tabelle 1: Interrupts**

| Interrupt-Name | Beschreibung |
|---|---|
| *dnComplete* | Wurde ein Download von zu versendenden Daten aus dem Speicher zur Karte beendet, informiert dieses Bit darüber |
| *upComplete* | Wie *dnComplete*, nur wird hier die Beendigung eines Uploads angezeigt |

Zusätzlich zu diesen neun Einträgen im `IntStatus`-Register, die über die Quelle eines Interrupts informieren, gibt es noch drei weitere Einträge in diesem Register. Das schon erwähnte `interruptLatch` zeigt das generelle Auftreten eines Interrupts an, der wie in der Tabelle beschrieben dann noch nach verschiedenen Ursachen unterschieden werden kann; ferner kann noch festgestellt werden, ob ein an die Karte abgesetztes Kommando sich noch in Bearbeitung befindet und welches das derzeit aktuelle Registerfenster ist.

## 2.7 Statistiken

Zu statistischen, aber auch zu Diagnosezwecken kann die Netzwerkkarte in einem Satz von Registern bestimmte Ereignisse zählen. So kann gezählt werden, wieviele Bytes insgesamt ohne Fehler übertragen wurden, wie oft das Trägersignal verloren wurde, wie oft ein Paket auf die Übertragung warten musste, da das Netz schon durch die Übertragung einer anderen Karte belegt war, oder wieviele Pakete fehlerlos übertragen werden konnten. Außerdem können noch verschiedene Arten von Kollisionen auf dem Netz gezählt werden sowie einige andere Statistiken (z.B. Anzahl der SQE Pulse Fehler). Die oben beschriebenen Daten sind alle auf das Versenden von Daten bezogen. Auch im Empfangsfall werden einige Statistiken erstellt, wenn auch nicht so viele wie beim Senden. Hier wird ebenfalls protokolliert, wieviele Bytes und Frames fehlerlos empfangen wurden, außerdem noch wieviele Pakete nicht empfangen werden konnten, da der Empfangspuffer schon voll war sowie für den Betrieb mit 100BASE-TX oder 100BASE-FX, wieviele Pakete mit einem fehlerhaften “start-of-stream delimiter” empfangen wurden.

Das Zählen all dieser Werte wird durch einen speziellen Befehl an die Karte aktiviert. Das Lesen aus einem der Register setzt dieses zurück, Schreiben in eines der Register addiert den geschriebenen Wert zum Inhalt. Es wird ein *updateStats* Interrupt ausgelöst, wenn ein Register sich dem Bereich eines Überlaufs nähert, d.h. wenn der Registerwert die Hälfte seines Maximalwertes erreicht.

## 2.8 Befehle und das Befehls-Register

Mittels Befehlen können auf der Netzwerkkarte bestimmte Funktionseinheiten aktiviert oder deaktiviert werden, Interrupts bestätigt, sowie einige Konfigurationsparameter gesetzt werden. Befehle werden in das `Command` Register geschrieben, welches ein 16 Bit breites Register ist, das an der gleichen Position liegt wie das `IntStatus` Register. Auf das `Command` Register kann nur schreibend zugegriffen werden.

Die Befehle sind spezielle Bitkombinationen, die sich aus einem Kommando-Teil zusammensetzen (die oberen fünf Bit) sowie möglicherweise je nach Befehl einem optionalen Parameter. Befehle können außer durch die verschiedenen Befehlsklassen, die weiter unter erläutert werden, noch nach ihrer Ausführungszeit unterschieden werden. Die meisten Befehle werden schneller ausgeführt, als das System einen weiteren Schreib- oder Lesezugriff ausführen kann. Einige Befehle brauchen jedoch mehr Zeit und somit muss der Treiber sicherstellen, dass der Befehl abgearbeitet wurde, bevor weitere Aktionen ausgeführt werden.

Insgesamt kann man 5 verschiedene Befehlsklassen unterscheiden.

So gibt es Befehle zum Reset der Karte, wobei hier die gesamte Karte zurückgesetzt werden kann (*GlobalReset*), nur der Empfangsteil (*RxReset*) oder der Sendeteil (*TxReset*), wobei bei diesen drei Befehlen mittels des Parameters noch feiner bestimmt werden kann, welche Teile zurückgesetzt werden sollen.

Der zweite Typ von Befehlen bestimmt das Senden. Hier kann die Download-Einheit ein- und ausgeschaltet (*DnUnStall* und *DnStall*), bestimmte Threshold-Werte für den Sendepuffer gesetzt (*SetTxReclaimThresh* und *SetTxStartThresh*) sowie das Versenden der Pakete ein- und ausgeschaltet werden (*TxEnable* und *TxDisable*).

Die dritte Klasse hat im Prinzip analoge Befehle für die Empfangseinheit. Dort können ebenso sowohl der Datentransfer vom Empfangspuffer in den Hauptspeicher (“upload”) als auch das eigentliche Empfangen aktiviert und deaktiviert (*UpUnStall*, *UpStall*, *RxEnable*, *RxDisable*) und ein Threshold-Wert gesetzt werden (*SetRxEarlyThresh*). Zusätzlich kann das Filterregister über einen Befehl geschrieben werden (*SetRxFilter*) und somit ist einstellbar, welche Arten von Paketen die Karte empfangen soll (nur Pakete, die an die Karte gerichtet sind; Multicast; Broadcast; alle Pakete (“promiscuous mode)”; Multicast-Pakete, die über einen Hashfilter auf der Karte nochmals überprüft werden sowie jede sinnvolle Kombination dieser Möglichkeiten) und es können einzelne Bits im HashFilter gesetzt werden (*SetHashFilterBit*). Durch diesen Hashfilter ist genauer einstellbar, welche Multicast-Pakete man empfangen möchte. Hierbei wird auf die Zieladresse eines Multicast-Pakets eine CRC-Prüfsumme berechnet und von diesem je nach Kartentyp die unteren sechs oder acht Bit als Index in den Hashfilter verwendet. Ist das dort adressierte Bit gesetzt, wird das Paket akzeptiert, sonst verworfen.

Die vierte Klasse von Befehlen ist für die Interruptbehandlung zuständig. Es können Interrupts bestätigt (*AcknowledgeInterrupt*) sowie ein Interrupt gezielt ausgelöst werden (*RequestInterrupt*). Darüberhinaus werden das `IndicationEnable` Register und das `InterruptEna-`

`ble` Register durch zwei Befehle geschrieben (*SetIndicationEnable* und *SetInterruptEnable*). Diese Register bestimmen, welche Interrupt-Quellen überhaupt sichtbar sind und welche einen Interrupt auslösen (siehe Kap. 2.6).

Die fünfte Klasse von Befehlen enthält Befehle, mit denen man noch einige Dinge, die keiner der anderen Klassen zugeordnet werden konnten, konfigurieren kann. So kann der DC-DC Konverter ein- oder ausgeschaltet werden (*EnableDcConverter* und *DisableDcConverter*). Dies sind zwei sehr spezielle Befehle, die nur bestimmte Karten abhängig vom ASIC betreffen und auch nur in der 10BASE2 Betriebsart wirksam sind. Die letzten drei Befehle sind dagegen wieder allgemein gültig. So kann wie schon erwähnt das Zählen von verschiedenen Ereignissen ein- und ausgeschaltet (*StatisticsEnable* und *StatisticsDisable*) sowie das aktive Registerfenster ausgewählt werden (*SelectRegisterWindow*).

## 2.9 Autonegotiation und das MII-Interface

Die Netzwerkkarten der Serien 3C90x und 3C90xB beherrschen das automatische Aushandeln der Übertragungsgeschwindigkeit sowie des Duplexmodus zwischen zwei Teilnehmern, die am selben Twisted-Pair Segment angeschlossen sind (nach dem Standard IEEE 802.3u). Jedoch ist diese Funktionalität zwischen Karten der beiden Serien verschieden, ferner gibt es bei der Serie 3C90xB auch noch Unterschiede im Registerlayout abhängig vom verwendeten ASIC.

Der Grund dafür ist, dass bei Karten der 3C90x-Serie die Autonegotiation Funktionen durch einen eigenen PHY-Chip von National Semiconductor implementiert werden, während bei den Karten der 3C90xB-Serie dies schon im ASIC integriert ist. Nach dem IEEE-Standard funktioniert Autonegotiation bei 10BASE-T, 100BASE-TX sowie 100BASE-T4. Bei den Karten mit 3C90xB ist 100BASE-T4 jedoch nicht implementiert.

Auf den Karten dieser Serie erscheinen die Register, mit welchen das automatische Aushandeln der Übertragungsparameter konfiguriert werden kann, als reguläre PHY-Einheit, obwohl sie auf dem Chip selbst schon integriert sind. Sie werden somit wie die Register eines externen Chips, der ein Medium Independent Interface (MII) besitzt, angesprochen. Eine 3C90xB-Karte kann jedoch auch eine PHY-Einheit besitzen, die nicht im Chip integriert ist und somit ebenso durch das MII angesprochen wird.

Das Medium Independent Interface ist eine serielle Schnittstelle, um Register auf einer PHY-Einheit mit einer MII-Schnittstelle anzusprechen. Diese Schnittstelle besitzt 2 Leitungen, die den ASIC auf der Netzwerkkarte mit einem MII-konformen PHY verbinden. Der Zugriff auf dieses Interface erfolgt über drei Bits im `PhysicalMgmt` Register, wobei dabei Bitmuster geschrieben bzw. gelesen werden, die den benötigten physischen Wellenformen entsprechen. Die seriellen Bitsequenzen, mit denen Register geschrieben und gelesen werden, heißen Frames. Diese beginnen immer mit einer Preamble von 32 gesetzten Bits, um sich zu synchronisieren. Danach kommen je nach Lese- oder Schreiboperation verschiedene Felder, die u.a. die Registeradresse beschreiben, den PHY adressieren (hier feste Adresse von 0x18h) und natürlich z.B. die eigentlichen zu sendenden Daten enthalten. Dabei bestehen dieses Teile wiederum aus mehreren Schreib-, Lese- oder Z-Zyklen. Ein Schreib- bzw. Lesezyklus schreibt bzw. liest ein Bit vom

Interface, wobei solch ein Zyklus aus mehreren Operationen auf dem `PhysicalMgmt` Register besteht. Ein Z-Zyklus beendet eine Übertragung und löscht dabei das Bit, welches die Übertragungsrichtung (Schreiben oder Lesen) angibt.

Wie bereits erwähnt, unterscheiden sich die ASICs der 3C90xB-Serie im Registerlayout bezüglich des PHY. Dies äußert sich in der Anzahl der verwendeten Register, die zur Kontrolle der Autonegotiation zur Verfügung stehen und auch in der Benennung der Register. Jedoch haben alle drei verschiedenen ASIC-Versionen einige Register, die bei allen ASICs die gleiche Adresse haben und auch von der Bedeutung fast gleich sind, jedoch ist ihr Inhalt nicht immer identisch (Adresse 00H bis 06h). Konzeptuell besteht aber bei allen ASICs die Möglichkeit zu bestimmen, welche Fähigkeiten der Karten überhaupt angeboten werden sollen (z.B. ist es möglich, 100BASE-TX von der automatischen Aushandlung auszuschließen) und zu erfahren, welche Möglichkeiten der Partner bei der Autonegotiation angeboten hat. Darüberhinaus kann die entsprechende Logik zurückgesetzt werden oder nochmals gestartet werden und es können einige Statusinformationen über die Aushandlung ausgelesen werden.

## 2.10 Die Konfiguration der Netzwerkkarte

Nach einem Reset muss die Netzwerkkarte konfiguriert werden, bevor Daten versendet oder empfangen werden können. Dabei werden zwei verschiedene Arten der Konfiguration unterschieden: die Konfiguration der Karte selbst und die PCI-Konfiguration.

Jedes PCI-Gerät besitzt einen 256 Byte großen Bereich von Konfigurationsregistern (64 32-bit Register). Davon sind die ersten 64 Byte für jedes Gerät gleich, also von der PCI-Spezifikation fest vorgeschrieben. Die restlichen 192 Byte sind geräteabhängig und dienen der speziellen Konfiguration des Geräts. Dieser 256 Byte große Bereich kann über einen speziellen PCI Bustransfer, einen sogenannten Konfigurationszyklus gelesen und geschrieben werden. Der Zugriff erfolgt dabei über zwei Register, die im I/O-Adreßraum des Systems liegen: das CONFIG_ADDRESS Register sowie das CONFIG_DATA Register. Über das CONFIG_ADDRESS Register wird das entsprechende Register adressiert, wobei dann im CONFIG_DATA Register die Daten erscheinen, falls aus dem adressierten Register gelesen werden soll oder die Daten aus dem CONFIG_DATA Register werden an die Zieladresse im Fall eines Schreibzugriffes geschrieben. Die Auswahl erfolgt dadurch, dass in jedem Fall erst das Register adressiert wird und dann entweder aus CONFIG_DATA gelesen wird oder in CONFIG_DATA geschrieben wird. Dabei besteht die Adresse, die in CONFIG_ADDRESS geschrieben wird, aus mehreren Einträgen: es kann einer von 256 Bussen ausgewählt werden (in hierarchisch strukturierten Systemen mit mehreren PCI-PCI-Brücken möglich), dann eines von 32 PCI-Geräten (PCI-Agent), nun eine von maximal 8 Funktionen (bei Multifunktionskarten kann es mehr als eine Funktion geben) und nun schließlich eines der 64 möglichen Register. Die zwei niedrigwertigsten Bits geben ferner noch einen Typ von Zugriffszyklus an: Typ 0 spricht Geräte am lokalen PCI-Bus an, ein Typ 1 Zugriff bedeutet, dass die adressierte Einheit hinter einer Brücke liegt, also auf einer anderen hierarchischen Ebene. Der Konfigurationszyklus selbst wird durch das höchstwertigste Bit des CONFIG_ADDRESS Registers gesteuert. Eine 1 initiiert einen solchen Zyklus, eine 0 bewirkt, dass ein gewöhnlicher I/O-Zyklus auf den normalen I/O-Adreßraum ausgeführt wird [Mes98].

Bezüglich der beiden Arten von Konfigurationszyklen unterscheiden sich die Karten der Serie 3C90x wieder von der Serie 3C90xB. Die Serie 3C90x antwortet nur auf Typ 0 Zyklen und hierbei auch nur auf die Gerätefunktion 0. Die Serie 3C90xB verarbeitet beide Arten von Konfigurationszyklen.

Jedoch muss man sich um diese Konfiguration als Treiberentwickler nicht mehr kümmern, da die PCI-Konfiguration schon entweder vom BIOS bzw. von der POST-Routine (Power on Self Test) durchgeführt wird. Man kann somit nach einem Systemstart die benötigten Werte, die vom BIOS bzw. der POST-Routine während des Starts gesetzt werden, wie z.B. der zugewiesene Interrupt, die I/O-Basisadresse, die Basisadresse für einen speicherbasierten Zugriff oder auch die Cacheline-Größe, auslesen. Der Treiber selbst muss nur die Netzwerkkarten-spezifischen Dinge konfigurieren. Hierbei ist die Auswahl des Medienports (Transceiver) der wichtigste Punkt.

Die Auswahl des Transceivers kann dabei durch zwei Arten geschehen, wobei ein Eintrag (`xcvrSelect`) des `InternalConfig` Register eine zentrale Rolle spielt. Hier können verschiedene physikalische Medien fest eingestellt werden, wie z.B. 10BASE-T oder 10BASE2. Der Inhalt dieses Registers wird auch im EEPROM der Karte gespeichert. Somit ist es möglich, einen Wert hier einzutragen und die Karte verwendet dadurch auch nach jedem Reset bzw. Start des Systems diesen voreingestellten Wert.

Die zweite, elegantere Möglichkeit ist das AutoSelect Verfahren. Dieses wird durch das Setzen eines Bits im `InternalConfig` Register konfiguriert, welches den Treiber anweist, den Eintrag in `xcvrSelect` zu ignorieren und stattdessen den Medienport dadurch zu bestimmen, indem die Ports nach verschiedenen Verfahren getestet werden und der aktive Port gesucht wird. Hierbei ist je nach möglichem Anschluß ein anderes Verfahren nötig, um festzustellen, ob dieser Anschluß eine Verbindung zum Netz hat. Wird kein aktiver Anschluß gefunden, so nimmt der Treiber die in `xcvrSelect` gewählte Einstellung. Die verschiedenen Möglichkeiten, die der Treiber ausprobieren muss, kann aus dem `MediaOptions` Register bestimmt werden, welches alle auf der Karte vorhandenen Anschlüsse speichert.

Die "einfachsten" Anschlüsse sind 10BASE-T oder 100BASE-TX, welche durch die Auto-negotiation Logik ausgehandelt werden. Normalerweise ist die Autonegotiation, welche nach einem Reset gestartet wird, beim Laden des Treibers schon beendet. Dieser muss nur noch durch das Prüfen einiger Bits feststellen, ob sie schon beendet ist und kann dann die Karte entsprechend konfigurieren (z.B. noch das `fullDuplexEnable` Bit setzen).

Ein möglicher anderer Anschluß sind über MII angeschlossene PHY. Hier ist das Vorgehen abhängig vom jeweiligen Gerät, das über die genormte MII-Schnittstelle angesprochen werden kann.

Bei 100BASE-FX ist das Vorgehen komplizierter als bei Twisted-Pair Anschlüssen. Hier müssen einige Registerwerte überprüft werden und durch das Senden selbstadressierter Pakete wird versucht, anhand einer Statistik abzuschätzen, ob eine Netzverbindung besteht oder nicht.

Bei AUI und 10BASE2 wird ebenso durch das Senden eines selbstadressierten Pakets versucht, zu bestimmen, ob ein korrekt terminiertes Netzwerk angeschlossen ist. Hierbei wird im Gegensatz zu 100BASE-FX jedoch nur ein Paket verschickt, anhand dessen die Entscheidung getroffen wird.

Nach der Bestimmung des Netzwerkanschlusses sind noch einige andere Dinge zu konfigurieren: es muss die Ethernet-Adresse in ein Register eingetragen werden, wobei die Adresse der Karte im EEPROM abgespeichert ist. Allerdings kann im Prinzip auch jede andere beliebige Adresse eingetragen werden, wenngleich man dadurch Gefahr läuft, eine Adresse zu wählen, die auch eine andere Netzwerkkarte hat, welche im gleichen Netz liegt. Somit kann die Eindeutigkeit der Ethernetadresse verletzt sein (wobei nach Spezifikation weltweit die Adresse eindeutig sein muss - wenngleich wohl nur Probleme auftreten, falls im selben Netz eine Adresse doppelt ist).

Nun muss noch eingestellt werden, welche Pakete die Karte empfangen soll. Hier sind neben an die Adresse der Karte adressierten Paketen noch Multicast- und Broadcastempfang möglich, ferner die feinere Auflösung von Multicast-Paketen über den Hashfilter.

Je nach Umgebung und ausgehandelter Verbindung muss eventuell noch der Vollduplex-Modus aktiviert werden und es kann auch eine Flußkontrolle aktiviert werden, welche durch PAUSE Frames nach IEEE 802.3x gesteuert wird.

Nach den hier erläuterten Einstellungen sowie nach den schon besprochenen Punkten wie Erzeugung der zur Verwaltung benötigten DPDs und UPDs sowie dem Start der entsprechenden Funktionseinheiten ist die Karte betriebsbereit [3CO98].

## 2.11 Zusammenfassung

Dieses Kapitel gab einen umfassenden Überblick über die Netzwerkkarte. Ausgehend von einer Beschreibung der Hardware-Architektur und der einzelnen Komponenten wurden die jeweiligen Funktionseinheiten näher betrachtet. Es wurde ein Überblick über die Registerbelegung der Karte gegeben und die Funktionsweise beim Versand und Empfang von Daten über Deskriptoren beschrieben. Dabei wurden verschiedene Möglichkeiten einer Treiberimplementierung angesprochen. Das mehrstufige Interrupt-Konzept der Karte wurde eingehend erläutert, die Statistikzähler der Karte wurden beschrieben und die Steuerung über Befehle erklärt. Ein Überblick über die MII-Schnittstelle und das Vorgehen bei der Initialisierung und Konfiguration der Karte beendeten dieses Kapitel.

# 3 Die Netzwerkprotokolle

Neben der Ansteuerung der Ethernet-Netzwerkkarte besteht die zweite Aufgabe der Studienarbeit darin, das TCP-Protokoll zu implementieren. In diesem Kapitel soll ein Überblick über die TCP/IP-Protokollsuite gegeben werden, wobei das Hauptgewicht auf dem TCP-Protokoll liegt. Es werden alle in der Studienarbeit implementierten Protokolle kurz umrissen sowie ihr Paketformat näher erläutert.

## 3.1 Referenzmodelle

In diesem Abschnitt werden kurz die beiden bekanntesten Netzwerkarchitekturmodelle, das ISO OSI Modell sowie das TCP/IP Modell erläutert, um die in den folgenden Kapiteln dargestellten Protokolle einordnen zu können und einen Einblick in das zu Grunde liegende Konzept zu erlangen.

### 3.1.1 ISO OSI Referenz Modell

Das Open Systems Interconnection (OSI) Referenz Modell der International Standards Organisation (ISO) ist wohl das bekannteste Modell einer Netzwerkarchitektur. Das Modell an sich ist noch keine Netzwerkarchitektur, da es nur definiert, was auf jeder Schicht getan werden soll, die exakten Dienste und Protokolle jedoch nicht spezifiziert. Diese wurden jedoch auch von der ISO erarbeitet und sind in einem weiteren Standard veröffentlicht.

Das OSI-Modell besitzt drei zentrale Konzepte: eine Schicht bietet *Dienste* für höhere Schichten an. Das *Interface* einer Schicht definiert, wie auf eine Schicht zugegriffen werden kann, welche Parameter erwartet werden und welche Ergebnisse möglich sind. Die *Protokolle* werden von einer Schicht benutzt, um die Dienste anbieten zu können. Hierbei sind diese auch auswechselbar, da sie in der Schicht gekapselt sind und diese nur über das Interface angesprochen werden kann.

Dabei kam das OSI Modell vor den Protokollen, ist also nicht auf einen bestimmten Satz von Protokollen ausgerichtet. Es ist somit sehr allgemein und eignet sich auch sehr gut, um Protokollsuiten anhand des OSI Modells einzuordnen und zu vergleichen.

Das Modell definiert sieben Schichten, wobei hierbei generell einige Grundsätze gelten: Jede Schicht stellt eine andere Abstraktionsebene dar. Die Aufgaben einer Schicht sind wohldefiniert und der Informationsfluß zwischen den Schichten soll so gering wie möglich gehalten werden. Dabei wurde die Anzahl der Schichten derart gewählt, dass sich unterschiedliche Funktionen auch in unterschiedlichen Schichten finden; aber natürlich sollte die Anzahl der Schichten nicht zu groß werden, um die Architektur nicht zu unübersichtlich werden zu lassen [Tan96].

Abb. 3.1 zeigt den Schichtenaufbau des Modells. Jede Schicht bietet Dienste für die nächsthöher liegende Schicht an und benutzt selbst die Dienste der direkt darunterliegenden Schicht.

Im folgenden eine kurze Erläuterung der einzelnen Schichten:

Die *Physical* Schicht kümmert sich um die eigentliche Übertragung der puren Bits über das Kommunikationsmedium und legt Dinge wie z.B. wieviele Volt eine logische Eins darstellen, fest.

Auf der *Data link* Schicht wird der Versand von Bitblöcken angeboten. Ferner beinhaltet diese Schicht eine erste Fehlerkontrolle sowie eine Flußkontrolle.

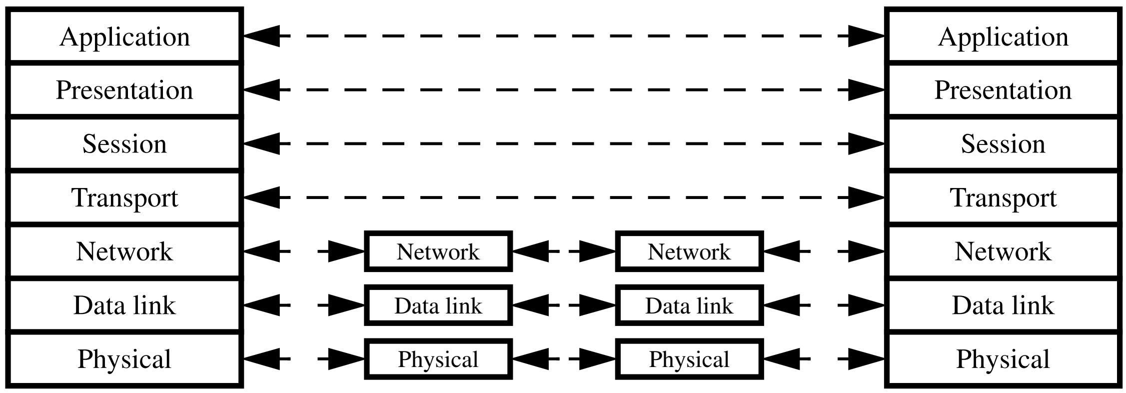


**Abb. 3.1 Das OSI Referenz Modell (aus [Kau94])**

Möglicherweise ist auch eine Nachrichtenwiederholung implementiert. Bei Broadcast-Netzen wie dem Ethernet kümmert sich eine Unterschicht dieser Schicht, die *Medium Access* Schicht, um den Zugriff auf das Medium.

Auf der dritten Ebene, der *Network* Schicht, wird eine virtuelle Verbindung zwischen zwei Einheiten angeboten. Hierbei ist der wichtigste Aspekt das Routing der Pakete von der Quelle zum Ziel. Bei Broadcast-Netzen ist dieses Problem natürlich einfach und diese Schicht muss nicht viel tun.

Die Aufgabe der *Transport* Schicht ist, die Daten der *Session* Schicht zu übernehmen und eventuell noch in kleinere Einheiten zu unterteilen und diese an die *Network* Schicht zu übergeben. Dabei hat sich die Schicht darum zu kümmern, dass die Daten am anderen Ende der Verbindung auch korrekt ankommen. Ab dieser Ebene besteht die Verbindung von Sender zu Empfänger (wie in Abb. 3.1 dargestellt), während bei den unteren drei Schichten die Verbindung von Maschine zu Maschine bestanden hat. Ab der Transport Schicht erscheint es so, als dass ein Programm auf dem sendenden Rechner direkt mit einem Programm auf dem empfangenden Rechner kommuniziert.

Die *Session* Schicht erzeugt und verwaltet Sitzungen. Es werden unter anderem Dienste wie Synchronisation (Verwaltung von Sicherungspunkten bei Datenübertragung großer Datenmengen) oder die Verwaltung des Zugriffs bei einem unidirektionalem Datenaustausch angeboten.

Die *Presentation* Schicht übernimmt die Wandlung spezieller Datenformate eines Rechners in ein gemeinsames Netzwerkdatenformat und zurück.

Die *Application* Schicht schließlich umfasst eine Vielzahl von verschiedenen Anwendungen mit ihren speziellen Protokollen (z.B. Datentransfer (FTP), Email (SMTP, POP), ...).

### 3.1.2 Das TCP/IP Referenz Modell

Das Hauptziel bei der Entwicklung von TCP/IP war die problemlose Verbindung vieler verschiedener Netze. Das TCP/IP Modell ist nach seinen beiden Hauptprotokollen benannt und im Gegensatz zum oben beschriebenen OSI Modell waren bei TCP/IP die Protokolle zuerst da und das Modell ist quasi nur eine Beschreibung der Protokolle. Auch die klare Trennung zwischen Dienst, Schnittstelle und Protokoll findet sich bei TCP/IP nicht wieder.

Das TCP/IP Modell hat im Gegensatz zum OSI Modell nur 4 Schichten, wobei die unterste Schicht, die "*host-to-network*" Ebene, in dieser Beschreibung noch feiner unterteilt wird in die "*physical*" Schicht und die "*data link*" Schicht (dies ist die Klassifizierung nach dem OSI Modell).

In Abb. 3.2 ist das Modell dargestellt. TCP/IP kann auf vielen verschiedenen Netzwerken genutzt werden und die Aufgaben der untersten Schicht werden im Prinzip vom jeweiligen Netz definiert. In den Beispielen wird immer von einem Ethernet ausgegangen und hier ist dann unter anderem festzulegen, in welchem Format Daten übertragen werden oder wie der Zugriff auf das Netz erfolgt. Dies sind alles Punkte, die in der Ethernet-Norm spezifiziert wurden. Allerdings werden wohl unabhängig vom verwendeten Netzwerk Dienste wie der vom ARP-Protokoll (siehe Kap. 3.3) angebotene benötigt werden, da kein Netzwerk auf dieser Ebene mit den "logischen" IP-Adressen umgehen kann. Daher wird ein Dienst benötigt, der die Abbildung von logischen Adressen in Hardware-Adressen anbietet. ARP wird nicht nur bei Ethernet verwendet, sondern es gibt auch andere Broadcast-Netzwerke, bei denen ARP verwendet wird. In Abb. 3.2 wurde die unterste Schicht wie oben beschrieben in "*Physical Layer*" und "*Data Link Layer*" aufgeteilt und beispielsweise bei dem "*Physical Layer*" ein Übertragungsmedium eingezeichnet, um die Aufgabe dieser Schicht darzustellen, wobei allerdings nicht vergessen werden sollte, dass bei diesem Modell keine exakten Angaben wie bei dem OSI Modell über diese Schicht gemacht werden.

Das TCP/IP Modell ist wie auch das OSI Modell ein geschichtetes Modell, das heißt, Dienste bzw. Protokolle einer höheren Schicht setzen wieder auf dem Dienst der niedrigeren Schicht auf.

Im folgenden wird eine kurze Beschreibung der drei weiteren Schichten gegeben. Die einzelnen Protokolle werden in den folgenden Abschnitten näher erläutert.

Die *Network* Schicht definiert ein Paketformat und bietet den Dienst der Paketvermittlung vom Sender zum Zielrechner an. Außerdem ist diese Schicht auch dafür verantwortlich, für das gesamte Netzwerk einen logischen Adressraum anzubieten, damit jeder Teilnehmer eindeutig adressiert werden kann. Es ist klar, dass hier Probleme wie beispielsweise das Routing von Paketen von einem Rechner zu einem anderen Rechner oder das Vermeiden von Engpässen beim Transport der Daten die Hauptaufgabe sind. Das Protokol auf dieser Ebene ist das IP-Protokoll (siehe Kap. 3.2).

Auf der Transport Schicht gibt es zwei Protokolle, ein zuverlässiges, verbindungsorientiertes Protokoll (TCP, siehe Kap. 3.5) und ein verbindungsloses, unsicheres Protokoll (UDP, siehe Kap. 3.4). Die generelle Aufgabe dieser Schicht ist wie beim OSI Modell, es verschiedenen Einheiten wie z.B. Prozessen auf verschiedenen Rechner zu ermöglichen, miteinander zu kommunizieren, wobei je nach Anforderungen an die Verbindung eines der beiden Protokolle gewählt werden kann.

Beim TCP/IP Modell gibt es keine *Session* und keine *Presentation* Schicht. Es ist jedoch so, dass manche Anwendungen ihre eigenen *Presentation* oder *Session* Schicht Funktionen mitbringen, wie z.B. bei NFS (Network File System) das XDR (External Data Representation). Aber explizit sind diese beiden Schichten im TCP/IP Modell nicht vorhanden. Daher ist schon die vierte Schicht die *Application* Schicht. Hier finden sich alle möglichen Arten von Anwendungen, wie z.B. Email (SMTP), Datenübertragung (FTP) oder natürlich das World Wide Web (HTTP). Diese Protokolle setzen je nach Anforderung wieder auf UDP oder TCP auf.

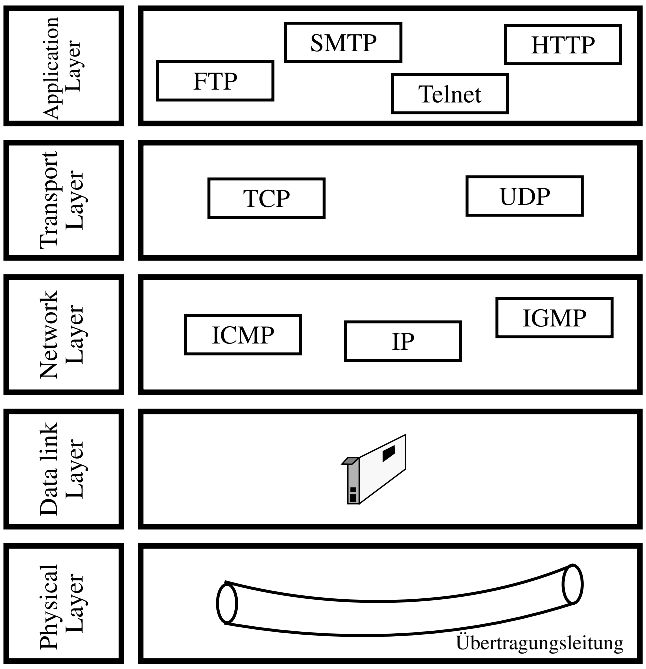


**Abb. 3.2 Das TCP/IP Referenz Modell (aus [TJ97])**

## 3.2 Das IP-Protokoll

Das Internet-Protokoll (IP) stellt für die semantisch auf einer höheren Ebene angesiedelten Protokolle wie UDP oder TCP die Grundlage dar, da es die notwendigen Dienste für den Versand der Daten zwischen den Hosts und Routern anbietet. Unter dem IP-Protokoll liegt nur noch die Netzwerkhardware, in unserem Fall also das Ethernet. Dort werden die IP-Pakete in Netzwerkframes eingepackt und dann versendet.

Das Internet selbst ist ein Netz von vielen kleineren Teilnetzwerken unterschiedlichster Architektur und Netzwerktechnologie. IP bietet die Möglichkeiten für eine netzübergreifende Verständigung, da es einen Mechanismus zur Adressierung der am Datenaustausch beteiligten Rechner bietet, der unabhängig vom Standort oder von der Architektur des Netzes, in dem sich der Rechner befindet, ist. Darüberhinaus wird geregelt, wie die Daten vom Sender zum Ziel gelangen und wie dabei die Daten verschiedene Teilnetze durchqueren. Natürlich wird von IP die Form des Datenaustauschs selbst festgelegt: es wird ein Format definiert, das den eigentlichen Nutzdaten einen Protokollkopf ("Header") voranstellt, der die für die obigen Aufgaben benötigten Daten enthält.

IP bietet einen verbindungslosen und nicht gesicherten Transport der Daten an. Das heißt, es besteht keinerlei Verbindung zwischen den Partnern einer Datenübertragung auf der Ebene des IP, es muss nicht erst beim Empfänger der Versand von Daten angemeldet werden. Beim Versand der Daten können Pakete verloren gehen, wovon sowohl Sender als auch Empfänger nichts mitbekommen. Beim Empfänger kommt einfach kein Paket an, der Sender erwartet keine Bestätigung oder ähnliches und kann so nicht feststellen, ob ein Versand erfolgreich war.

### 3.2.1 Die Adressierung von IP

Um Daten versenden zu können, müssen Sender und Empfänger eindeutig bestimmt werden. Hierzu werden die sogenannten IP-Adressen verwendet. Diese sind im Prinzip ein 32Bit-Wert, welcher aber zur besseren Lesbarkeit als durch vier Punkte getrennte Dezimalzahlen dargestellt wird, wie z.B. 131.188.34.118 (faui40c.informatik.uni-erlangen.de).

IP-Adressen können nochmals in 5 verschiedene Adreßklassen unterteilt werden, abhängig von den höchstwertigsten Bits der Adresse. Diese Klassen werden mit den Buchstaben von A bis E gekennzeichnet. Die Klassen A bis C sind von der Bedeutung sehr ähnlich: die Adresse wird aufgespalten in einen Netzwerk-ID Teil und einen Host-ID Teil. Mit der Netzwerk-ID wird ein spezielles Netzwerk adressiert, mit der Host-ID ein bestimmter Rechner innerhalb dieses Netzes. Die Adreßklassen A bis C unterscheiden sich darin, wieviele Bits für die Netzwerk-ID und wieviele Bits für den Host-Teil verwendet werden.

Die Klasse D bezeichnet Multicast-Adressen, welche aber meist nicht permanent, sondern dynamisch vergeben werden. Die Klasse E schließlich ist für die Entwicklung neuer Adreßmechanismen und neuer Protokolle reserviert.

Darüberhinaus gibt es noch einige spezielle IP-Adressen z.B. für einen Broadcast im lokalen Netz oder als Loopback-Adresse (Rückkopplung).

Zusätzlich zur IP-Adresse besitzt ein Rechner noch eine Subnet-Maske. Unabhängig von den oben genannten Klassen ist eine Adresse in einen Netzwerk- und in einen Hostteil aufspaltbar, wobei sich die Klassen in der Größe des jeweiligen Teils unterscheiden (von Klasse A mit acht Bits für Netzwerk und 24 für die einzelnen Rechner bis zu Klasse C mit 24 Bits für das Netzwerk und acht Bits für die einzelnen Rechner). Mittels der Subnet-Maske kann nun der Hostteil weiter unterteilt werden in einen Subnetz-Teil und einen Host-Teil [ML94]. Dies hat einerseits administrative Vorteile, da das Netz so besser in kleinere, übersichtlichere Teilnetze aufgeteilt werden kann, andererseits wird dadurch die Größe der Routingtabellen reduziert, da externe Router nur den Weg zum Netzwerk kennen müssen und keinerlei Wissen über die weitere Aufteilung in Subnetze besitzen müssen [Ste94].

Dennoch hat die Aufteilung der IP-Adressen in die verschiedenen Klassen heute für das Routing keine Bedeutung mehr, da Anfang des letzten Jahrzehnts Wege gesucht wurden, um dem sehr starken Wachstum des Internets zu begegnen und die Routingtabellen nichts ins Unendliche wachsen zu lassen. Beim Konzept des CIDR ("classless interdomain routing") werden die Adressen nicht mehr in Klassen aufgeteilt, sondern das Routing wird anhand der ganzen 32-Bit Adresse zusammen mit Maskierungsoperationen (32-Bit Maske) entschieden. Dabei soll erreicht werden, dass viele einzelne Routen auf einer hierarchisch höheren Ebene (beispielsweise bei einem Backbone) zu einer Route aggregriert werden können, um so die Routingtabellen zu verkleinern. Dazu müssen die Routen alle die selben höhersignifikanten Bits gemeinsam haben, d.h. die Adressen haben alle einen gemeinsamen Präfix, der aber nun ohne Klasseneinteilung nicht mehr auf eine bestimmte Länge festgelegt ist. Außerdem musste zur Verwirklichung dieses Konzept auch die Vergabe von Adressen entsprechend angepasst werden: dabei müssen nun Adressen in zusammenhängenden Bereichen vergeben werden, damit dieser Bereich gegenüber anderen Routern als eine einzige Route angegeben werden kann und nicht die vielen einzelnen Adreßbereiche aufgelistet werden müssen, was zu größeren Routingtabellen führen würde [RL93], [HKC96], [FLY93].

### 3.2.2 Das Paketformat von IP

Abb. 3.3 gibt einen Überblick über den Header des Internet-Protokolls. Er besteht aus einem festen 20 Byte großen Teil sowie einem optionalen Teil mit variabler Länge. Falls die Optionen nicht auf eine 32 Bit-Grenze enden, so werden sie mit Füllbytes aufgefüllt, so dass die an den Header folgenden Daten immer an einer 32 Bit-Grenze beginnen.

Die einzelnen Felder haben folgende Bedeutung: Im *Version*-Feld wird die verwendete Version des IP-Protokolls vermerkt. Derzeit ist das noch die Version vier, aber es wird schon seit längerer Zeit an der Version sechs gearbeitet. Der *IHL* Eintrag gibt die Länge des Headers in 32-Bit-Worten an und ist somit ein Offset zu den Daten. Mit *Type of Service* ist ein Eintrag vorhanden, mit dem Anforderungen an die Priorität des Pakets und den gewünschten Übertragungsweg gestellt werden können, falls dies unterstützt wird. Praktisch wird dieses Feld von den Routern ignoriert. Der *Total length* Wert gibt die komplette Länge von Daten und Header zusammen an. *Identification* ist zur Zuordnung eines Fragments zu einem IP-Paket notwendig. Dabei haben alle Fragmente des gleichen Pakets denselben Wert. Die beiden Bits *DF* und *MF* behandeln

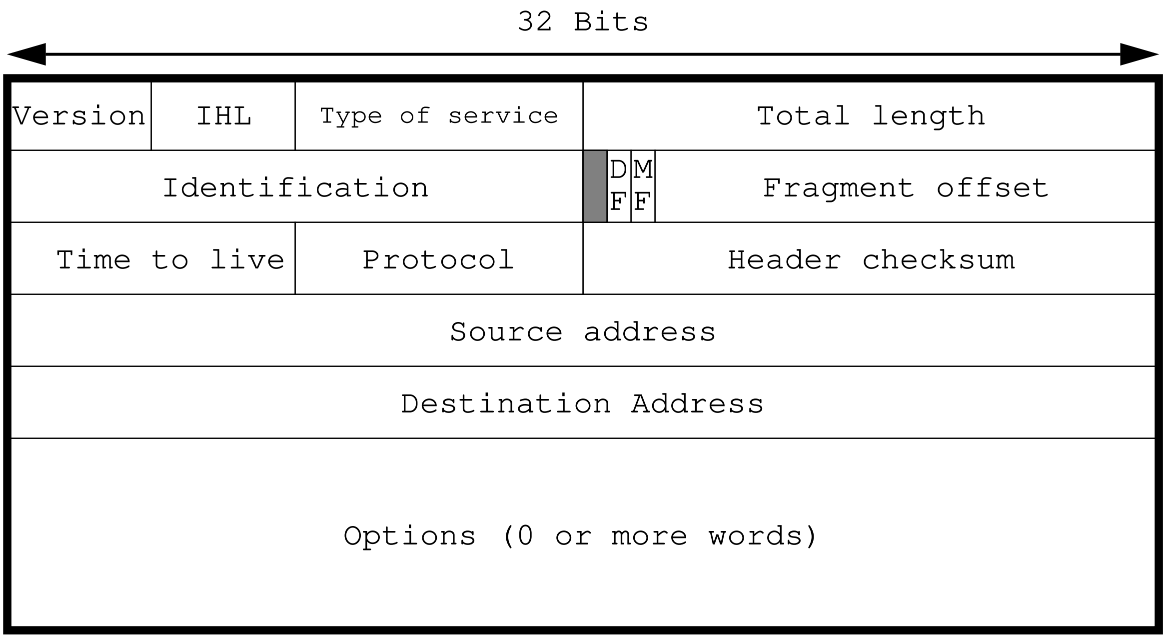


**Abb. 3.3 Der IP Header (aus [Pos81])**

Fragmente. *DF* bedeutet, dass der Router ein Datenpaket nicht fragmentieren soll, weil beispielsweise das Zielnetz nicht damit umgehen kann, *MF* zeigt an, dass die Daten ein Fragment eines größeren Pakets sind. Alle Fragmente bis auf das letzte haben dann dieses Bit gesetzt. Mit *Time to live* kann die Lebensdauer eines Pakets beschränkt werden. In der Praxis wird dieser Wert an jedem Rechner, den das Paket passiert, dekrementiert und bei Erreichen des Wertes null wird das Paket verworfen. Der *Protocol* Eintrag zeigt an, welcher Art die enthaltenen Daten sind und welches Protokoll diese weiterverarbeiten soll (z.B. TCP oder UDP) und dient somit dem Multiplexing der Protokolle über IP. Die *Header checksum* überprüft nur den Header und ermöglicht somit immerhin, Fehler in diesem zu erkennen. Auf diesen Wert folgen dann die Adressen des Senders sowie des adressierten Rechners. Mit den bereits erwähnten Optionen besteht eine Möglichkeit, das Protokoll zu erweitern, ohne gleich den Header neu festlegen zu müssen. Im RFC 791 [Pos81] sind acht verschiedene Optionen spezifiziert.

## 3.3 Das ARP-Protokoll

Zwar besitzt jeder an das Internet angeschlossene Rechner mindestens eine IP-Adresse, jedoch sind diese nur oberhalb der Netzwerkhardware angesiedelt. Die Netzwerkkarten selbst verstehen diese Adressen nicht und jedes unter IP benutzte Netzwerk (z.B. X.25 oder Ethernet) hat ein anderes Adressformat. Die Abbildung der Internet-Adressen auf die physischen Adressen des jeweiligen verwendeten Netzwerks ist die Aufgabe des “Adress Resolution Protocols” (ARP). Es gibt verschiedene Fälle, in denen ARP benötigt wird. Will ein Rechner Daten an einen Rechnern schicken, der im selben Netz liegt, so wird dessen physikalische Netzwerkadresse benötigt. Sollen die Daten an einen Rechner geschickt werden, der nicht im selben Netz liegt, so werden diese über einen Router weitergeleitet, der im selben Netzwerk liegt. Auch in diesem Fall wird eine physikalische Adresse benötigt, nur eben diejenige des Routers.

Das ARP-Protokoll geht nun folgendermaßen vor, um die physikalische Adresse des Zielrechners herauszufinden: es sendet ein spezielles Paket mittels Broadcast an alle Rechner im gleichen Netz. In dieses Paket ist die IP-Adresse des gesuchten Rechners eingetragen. Derjenige Rechner, der diese IP-Adresse hat, trägt nun in dieses Paket seine physikalische Netzwerkadresse ein und sendet dieses Paket zurück. Unter Umständen kennt er jedoch auch nicht die physikalische Adresse des Rechners, an den er die Antwort zurücksenden will und müßte nun seinerseits eine ARP-Anfrage abschicken. Deswegen liefert aus Effizienzgründen der anfragende Rechner gleich seine Adresse mit. Ferner haben die Rechner einen ARP-Cache, in dem sie solche Paare von IP-Adresse/Netzwerkadresse speichern und somit nicht immer eine ARP-Anfrage versenden müssen. Außerdem können auch die Rechner, an die ein ARP-Paket nicht gerichtet ist, dennoch den Inhalt auslesen und das Ergebnis in ihren Cache eintragen. Somit haben sie die benötigen Informationen für den Datenversand an andere Rechner wahrscheinlich schon im Cache.

### 3.3.1 Das Paketformat von ARP

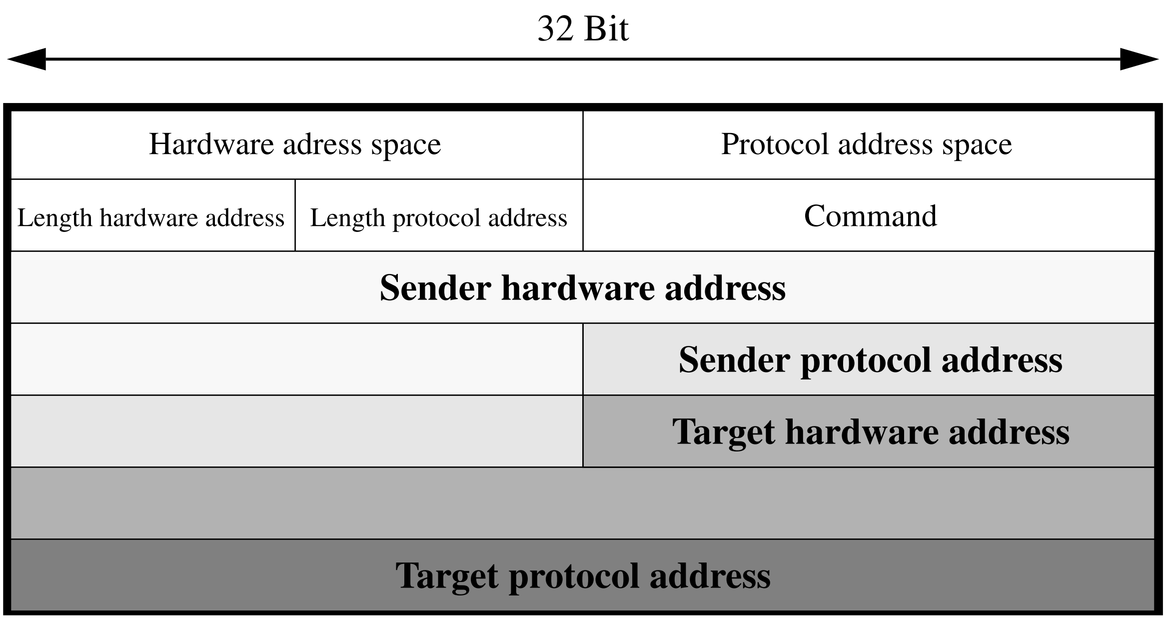


**Abb. 3.4 ARP-Paketformat (aus [Plu82])**

In Abb. 3.4 ist grafisch ein ARP-Paket dargestellt. Der Aufbau des Pakets ist relativ einfach. Das Paket besitzt einen kleinen, immer gleich großen Kopf sowie einen je nach Netzwerk variablen Teil. Die ersten beiden Felder *Hardware address space* und *Protocol address space* geben Auskunft über die im Paket verzeichneten Protokolle der jeweiligen Ebene. Beispielsweise steht eine Eins bei *Hardware address space* für Ethernet und 0x800 bei *Protocol address space* bezeichnet das IP-Protokoll. Danach werden die Längen der Adressen der jeweiligen Protokolle angegeben. Nach dem *Command* Eintrag, der eine ARP-Anfrage (eingetragener Wert = 1) oder eine ARP-Antwort (eingetragener Wert = 2) kennzeichnet, kommt der variable Teil. Dieser Teil enthält die Hardware- und die Protokolladressen von Sender und Empfänger und dessen Länge ist natürlich abhängig von dem verwendeten Netzwerk und dem Protokoll. Die Abb. 3.4 zeigt ein Paket für den typischen Fall IP-Protokoll über Ethernet.

### 3.3.2 Das RARP-Protokoll

Manchmal tritt auch der Fall auf, dass zu einer Ethernet-Adresse die korrespondierende IP-Adresse gesucht wird. Dies ist beispielsweise bei Rechnern ohne Festplatte, die beim Booten ihre Protokoll-Adresse erfahren möchten, aber nur ihre Hardware-Adresse, die fest vorgegeben ist, kennen, der Fall. RARP verwendet das gleiche Paketformat wie ARP. Es werden zwei neue Commando-Codes definiert, die beiden bisherigen Codes werden an den normalen ARP-Code übergeben. Im Gegensatz zu ARP muss bei RARP ein spezieller Rechner im Netz vorhanden sein, der eine Tabelle mit Abbildungen zwischen Hardware-Adresse und Protokoll-Adresse besitzt, um auf RARP-Anfragen antworten zu können.

## 3.4 Das UDP-Protokoll

Das User Datagram Protocol (UDP) ist neben TCP wohl das wichtigste Transportprotokoll im Internet. UDP bietet einen Versand von Daten ohne den Aufbau einer Verbindung, d.h. die zu übertragenden Daten können einfach in ein Paket gepackt werden und mittels IP versendet werden. UDP ist immer dann vorteilhaft, wenn ein geringer Overhead notwendig ist oder nur wenig gesendet wird wie z.B. eine Anfrage und eine Antwort darauf. Bei manchen Anwendungen wie z.B. Audio-Streaming kann im Fehlerfall ein Paket auch nicht nochmals übertragen werden, weil dafür in Echtzeit-Audio die Zeit fehlt und ein kleines Knacksen o.ä. nicht ins Gewicht fällt. Dafür hat man bei UDP die selben Unsicherheiten des IP-Protokolls, da UDP auf IP aufsetzt und nur wenig in seinem eigenen Protokollkopf zu den Daten des IP-Headers hinzufügt. Es ist einerseits nicht sichergestellt, dass ein Paket ankommt und andererseits erhält der Sender über den erfolgreichen Transport keinerlei Rückmeldung.

### 3.4.1 Das Paketformat von UDP

In Abb. 3.5 ist das Format des UDP-Headers dargestellt. Auf den Pseudo-Header wird weiter unten eingegangen. Der Header selbst ist sehr einfach gehalten. Die Daten werden mit einem acht-Byte Header versehen und dann in ein IP-Paket gepackt. Die ersten zwei Einträge geben den Ursprungs-Port und den Ziel-Port an, um auf einem Rechner verschiedene “Dienste” adressieren zu können. Darauf folgt die Länge des gesamten Datagramms, also Kopf sowie Daten. Hier steht somit mindestens immer der Wert acht. Danach kann eine Checksumme eingetragen werden. Bei UDP ist diese optional und falls keine Checksumme berechnet wurde, trägt dieses

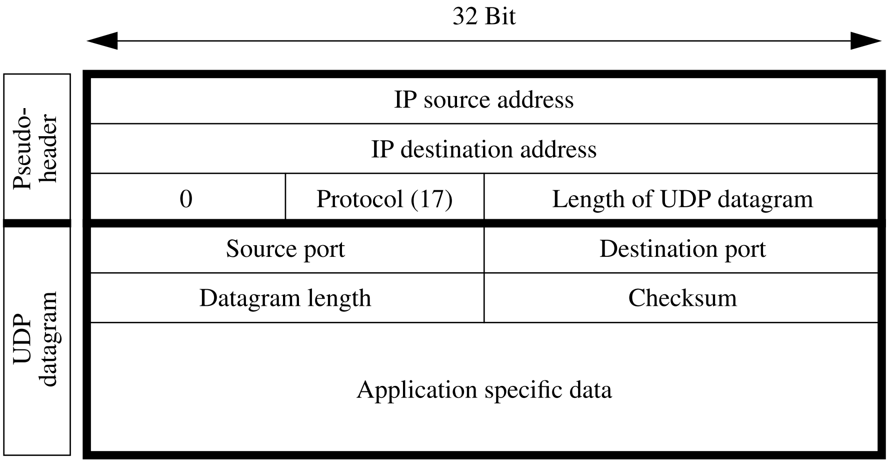


**Abb. 3.5 UDP-Datagramm mit Pseudo-Header (aus [Pos80])**

Feld den Eintrag null. Bei der Berechnung der Checksumme wird jedoch nicht allein über den UDP-Header und die zu sendenden Daten ein Prüfwert berechnet, sondern dem Paket wird der in Abb. 3.5 mit eingezeichnete Pseudo-Header vorangestellt. Dieser Teil dient nur zur Berechnung der Prüfsumme und wird nicht mitversendet. In diesem Pseudo-Header sind noch die IP-Adressen eingetragen, ein fester Wert null an Byte neun, der Protokoll-Code für UDP und die Länge des Datagramms. Mit Hilfe dieses zusätzlichen Headers kann der Empfänger auf Ebene des UDP noch überprüfen, ob die IP-Daten des Pakets verfälscht wurden und beispielsweise die Zieladresse falsch ist und somit das Paket gar nicht für diesen Rechner bestimmt ist.

Die Checksumme selbst wird als Summe der Einerkomplemente aller 16-Bit Worte berechnet. Von dieser Summe wird wiederum das Einerkomplement gebildet und in den Header eingetragen [BB88].

## 3.5 Das TCP-Protokoll

Das Transmission Control Protocol (TCP) ist ein sicheres, stromorientiertes Übertragungsprotokoll, das es ermöglicht, zwischen Prozessen auf verschiedenen Rechnern Daten auszutauschen. Dabei setzt das Protokoll selbst auf Diensten des IP-Protokolls auf, welches einen unsicheren Datagramm Dienst bietet. TCP wurde unter anderem entwickelt, um eine sichere und zuverlässige Datenübertragung über ein unsicheres Netzwerk wie z.B. das Internet zu ermöglichen. Im Gegensatz zu einem LAN sind hier Ausfälle von einzelnen Rechner wahrscheinlicher schon allein wegen der großen Anzahl der Rechner, ferner besteht solch ein Netz aus vielen Teilnetzen unterschiedlicher Topologie mit vielen verschiedenen Parametern wie MTU (Maximum Transfer Unit) oder Bandbreite. TCP kann mit einer Vielzahl von möglichen Fehlern umgehen und schottet Anwendungen oder andere Protokolle, die TCP benutzen, von den Unsicherheiten des Netzwerks ab. Dazu arbeitet TCP verbindungsorientiert, d.h. vor dem Versand von Daten muss erst eine virtuelle Verbindung aufgebaut werden und nach Beendigung des Transfers wieder geschlossen werden.

Die Möglichkeiten, die TCP anbietet und auf welche Art und Weise das Protokoll auf einem unsicheren Netzwerk eine zuverlässige Datenübertragung erreicht, werden in diesem Abschnitt näher erläutert.

### 3.5.1 Grundlegende Konzepte

Bei TCP werden die zu versendenden Daten, die als Octets bezeichnet werden (8-bit Bytes), in sogenannten Segmenten versendet. Prinzipiell können diese Segmente beliebig lang werden, aber in der Praxis setzt TCP auf IP auf, welches die maximale Länge schon beschränkt und selbst wiederum in einer effizienten Implementierung abhängig ist von der MTU des unterliegenden Netzwerks. Ferner hat der TCP-Teil des empfangenden Rechners auch nur einen Puffer bestimmter Länge reserviert und kann ein zu großes Segment möglicherweise gar nicht annehmen. Deshalb kann die maximale Segmentgröße beim Verbindungsaufbau auch mit ausgehandelt werden.

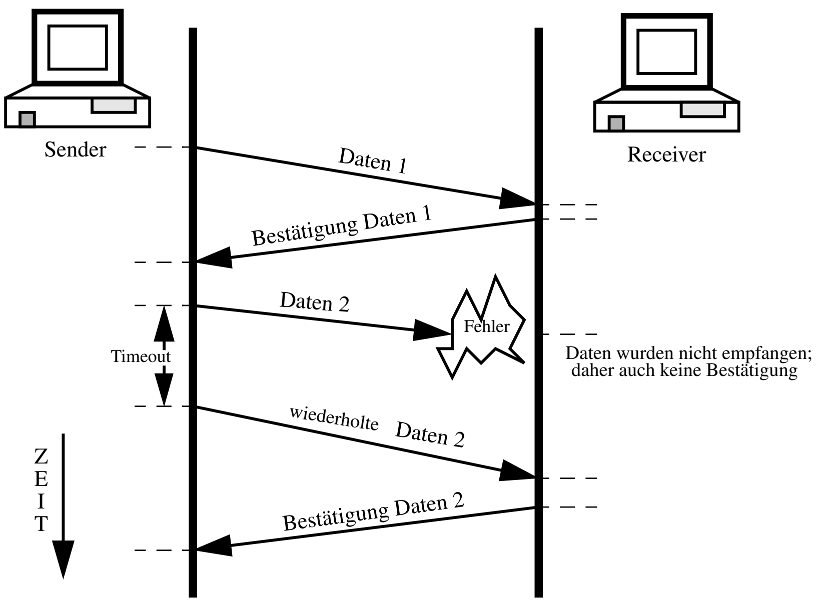


**Abb. 3.6 Positives Bestätigungsschema von TCP (aus [Siy97])**

TCP ist stromorientiert und stellt sicher, dass der Strom der Daten in der selben Reihenfolge beim Empfänger ankommt, wie er gesendet wurde. Dabei werden nicht die einzelnen versendeten Segmente verwaltet, sondern die Octets selber, d.h. dass das TCP keine Blockgrenzen o.ä. kennt, sondern die Daten durchnummeriert werden und anhand dieser Nummern das Protokoll den Versand verwaltet. Auch in welchen Segmentgrößen die Daten versendet werden, liegt allein in der Verantwortung des TCP. Eine Anwendung kann mehrere Sendeaufrufe machen, die das TCP aber selbst puffert und diese Daten zusammen in ein Segment packt (natürlich unter Berücksichtung der maximalen Segmentgröße) oder es kann für jeden Aufruf sofort ein Segment versenden. Dabei gibt es jedoch auch die Möglichkeit, mittels eines Push-Flags zu veranlassen, dass die Daten nicht gepuffert und sofort versendet werden sollen. Der empfangende Rechner kann jedoch nicht feststellen, in welchen Schritten die Daten vom Sender geschrieben wurden (z.B. auf einmal oder mehrere kleinere Pakete). Die Daten kommen für ihn als Strom von Bytes an.

Ebenso wie schon UDP verwendet auch TCP Ports, um den Empfänger auf dem Zielrechner zu adressieren. Dabei ist eine Verbindung vollständig beschrieben durch die beiden Sockets[1] auf dem sendenden Rechner sowie auf dem Zielrechner.

Eine der wichtigsten Aufgaben des Protokolls ist der zuverlässige Transport der Daten. Um dies zu ermöglichen, muss TCP mit verfälschten Daten, verlorenen Daten, mit doppelten Daten und mit Daten, die nicht in der richtigen Reihenfolge ankommen, umgehen können. Hierzu wird ein positives Bestätigungsschema verwendet, d.h. der Empfang der Daten wird bestätigt und bei Ausbleiben einer Bestätigung sendet der Rechner die Daten erneut.

In Abb. 3.6 ist dieses Vorgehen dargestellt. In der Grafik bleibt die Bestätigung für den Empfang der Daten zwei aufgrund eines Paketverlusts aus, weswegen der Sender nach Ablauf eines Timers die Daten erneut sendet und dann eine Bestätigung erhält. Bestätigungen müssen nicht sofort auf den Empfang eines neuen Segments erfolgen, sondern es ist möglich, mehrere empfangene Segmente auf einmal zu bestätigen, um so Netzbandbreite zu sparen. Jedoch muss natürlich berücksichtigt werden, dass eine zu lange Verzögerung beim Versand der Bestätigung zur Folge hat, dass der andere Rechner die Daten erneut sendet.

Dieses Verfahren wird durch die Nummerierung des Datenstroms implementiert (sogenannte Sequenzierung; die Nummern werden als Sequenznummer bezeichnet). Eine Bestätigung ist dann auch eine Nummer, die die Position im Datenstrom angibt, bis zu der alle Daten empfangen wurden. Mit Hilfe der Nummerierung kann der Empfänger die Daten auch gegebenenfalls in die richtige Reihenfolge bringen oder einfach Duplikate erkennen. Verfälschte Daten werden durch eine Prüfsumme im TCP-Header erkannt.

### 3.5.2 Verbindungsaufbau und Verbindungsabbau

Bevor Daten übertragen werden können, muss zwischen den beiden beteiligten Rechnern erst eine virtuelle Verbindung aufgebaut werden. Dies ist notwendig, da beide Seiten Zustandsinformationen über die Verbindung speichern müssen, um mit den oben beschriebenen Fehlern umgehen zu können. Daher müssen anfangs entsprechende Werte synchronisiert werden, bevor mit der eigentlichen Übertragung begonnen werden kann. Aus demselben Grund muss auch eine Verbindung explizit abgebaut werden, wenn keine weiteren Daten mehr versendet werden sol-

1. dies bezeichnet die Kombination der Internetadresse und dem Port; die heutige Programmierschnittstellen in Unix-Systemen, die als Sockets bezeichnet werden, haben ihren Namen von der Bezeichnung im ursprünglichen Request for comment [TCP81]

len. Da die Daten vom Modell her gesehen als kontinuierlicher Strom übertragen werden, gibt es kein explizites Endesymbol o.ä. und es muss dem Gegenüber mitgeteilt werden, dass die Verbindung abgebaut werden soll. Dabei ist es auch möglich, dass zuerst nur eine Seite die Verbindung beendet und somit keine weiteren Daten mehr senden kann. Datenempfang ist auf dieser Seite jedoch immer noch möglich, so lange bis auch der andere Rechner die Verbindung abgebaut hat. Es ist natürlich möglich, dass durch einen Fehler oder Rechnerabsturz eine Verbindung abrupt beendet wird. Dies fällt jedoch unter die Fehlermöglichkeiten und kann auch erkannt werden.

Beim Verbindungsaufbau gibt es zwei verschiedene Möglichkeiten. Es gibt eine sogenannte passive und eine aktive OPEN Anforderung. Ein passives OPEN bedeutet hierbei, dass der Rechner bzw. Prozeß Verbindungen von anderen Rechnern annimmt; meist sind das beliebige Rechner, aber es ist auch möglich, dies einzuschränken. Das aktive OPEN dagegen initiiert eine Verbindung und sendet ein Synchronisationspaket an den anderen Rechner. Meistens ist dies ein Rechner, der mittels eines passiven OPEN auf eingehende Verbindungswünsche wartet. Es ist aber auch möglich, dass beide Rechner ein aktives OPEN machen. Auch in diesem Fall kommt eine Verbindung zustande.

| TCP A | | | | TCP B |
|---|---|---|---|---|
| CLOSED | | | | LISTEN |
| SYN-SENT | → | <SEQ=100><CTL=SYN> | → | SYN-RECEIVED |
| ESTABLISHED | ← | <SEQ=300><ACK=101><CTL=SYN,ACK> | ← | SYN-RECEIVED |
| ESTABLISHED | → | <SEQ=101><ACK=301><CTL=ACK> | → | ESTABLISHED |
| ESTABLISHED | → | <SEQ=101><ACK=301><CTL=ACK><DATA> | → | ESTABLISHED |

**Abb. 3.7 3-Wege Handshake beim Verbindungsaufbau (aus [TCP81])**

Der Verbindungsaufbau erfolgt mittels eines Drei-Wege-Handshake. Dabei werden die beiden initialen Sequenznummern der Partner synchronisiert. Das prinzipielle Vorgehen ist, dass durch ein aktives OPEN der erste Rechner ein Paket schickt, das durch ein gesetztes Flag (SYN-Flag) anzeigt, dass ein Verbindungsaufbau gewünscht wird. Dabei wird die initiale Sequenznummer mitübertragen. Der zweite Rechner bestätigt den Empfang und überträgt seinerseits seine initiale Nummer, welche im dritten Schritt vom ersten Rechner bestätigt wird.

In Abb. 3.7 wird das Vorgehen dargestellt. Hierbei sind links und rechts am Rand die jeweiligen Zustände des TCP angeschrieben. Das Zustandsmodell von TCP wird in Kap. 3.5.10 genauer erklärt und ist hier nicht von so großer Bedeutung. Die Pfeile geben die Richtung des Pakets an, welches in der Mitte dargestellt ist. Hierbei sind nur die relevanten Teile wie die Sequenznummer u.ä. eingezeichnet; die restlichen Einträge wurden aus Übersichtsgründen weggelassen.

In dieser Grafik ist der normale Vorgang dargestellt, bei dem ein Rechner mittels eines aktiven OPEN (hier Rechner A) mit einem Rechner in Verbindung tritt, der auf Verbindungen mit einem passiven OPEN wartet (Rechner B). Daher ist Rechner B schon in dem sogenannten LISTEN Zustand, in dem er auf eingehende Verbindungen wartet. Rechner A sendet nun ein Paket mit dem gesetzten SYN-Flag und seiner initialen Sequenznummer, in diesem Beispiel 100. Rechner B antwortet auf den Verbindungswunsch mit ebenfalls gesetztem SYN-Flag sowie dem ACK-Flag, das eine Bestätigung anzeigt. Rechner B sendet seinerseits seine eigene initiale Sequenznummer 300 und bestätigt mit der Angabe von “ACK=101” und dem gesetzten ACK-Flag, dass er das Paket von A empfangen hat und im nächsten Paket eine Sequenznummer von 101 erwartet. Darauf bestätigt Rechner A im nächsten Schritt dies wiederum mit einem Paket mit der ACK Nummer 301 und eigener Sequenznummer 101, wie von Rechner B erwartet. Ab nun können Daten übertragen werden, wie in der Grafik in der letzten Zeile gezeigt.

Das hier erläuterte Vorgehen ist der gewöhnliche Fall. Es ist jedoch auch möglich, dass beide Rechner quasi gleichzeitig mittels eines aktiven OPEN miteinander in Kontakt treten wollen. Auch dann kommt eine Verbindung zustande, allerdings ist der Verbindungsaufbau etwas komplizierter.

Auch beim Schliessen einer Verbindung wird nach einer Handshake-Methode verfahren. Dabei gibt es drei prinzipielle Möglichkeiten, um eine Verbindung zu schliessen. Es kann explizit vom Benutzer initiiert werden oder es wird vom anderen Rechner ein Paket empfangen, das anzeigt, dass die Verbindung geschlossen wird. Außerdem ist es möglich, dass beide Rechner simultan die Verbindung schliessen möchten.

| | | | | |
|---|---|---|---|---|
| TCP A | | | | TCP B |
| ESTABLISHED | | | | ESTABLISHED |
| (Close)<br>FIN-WAIT-1 | → | <SEQ=100><ACK=300><CTL=FIN,ACK> | → | CLOSE-WAIT |
| FIN-WAIT-2 | ← | <SEQ=300><ACK=101><CTL=ACK> | ← | CLOSE-WAIT |
| TIME-WAIT | ← | <SEQ=300>>ACK=101><CTL=FIN,ACK> | ← | (Close)<br>LAST-ACK |
| TIME-WAIT | → | <SEQ=101><ACK=301><CTL=ACK> | → | CLOSED |
| (2 MSL)<br>CLOSED | | | | |

**Abb. 3.8 Normale Verbindungsabbausequenz (aus [TCP81])**

In jedem Fall werden dabei Pakete mit gesetztem FIN Flag versendet, welches anzeigt, dass die Verbindung geschlossen werden soll. Wird der Empfang des FIN Pakets von der Gegenseite bestätigt, so ist diese Seite für das Senden schon geschlossen. Ist dies auf beiden Seiten passiert, so ist die Verbindung beendet.

In Abb. 3.8 ist dies dargestellt. Die einzelnen Zustände sind hier nicht so wichtig, vielmehr soll das Prinzip erklärt werden. Anfangs bekommt ein Rechner (hier A) einen Close-Aufruf vom Benutzer. Daraufhin wird ein Paket mit gesetztem FIN Flag versendet. Der Rechner kann keine weiteren neuen Daten mehr versenden, stellt jedoch sicher, dass alle bis zum Close-Aufruf versendeten Daten noch sicher übertragen werden und kann auch noch Daten von Rechner B empfangen. Den Empfang des FIN-Paketes bestätigt Rechner B. Dieser Rechner sendet nachdem er von seiner Benutzerseite ebenfalls die Aufforderung erhalten hat, die Verbindung zu schliessen, seinerseits auch ein Paket mit FIN Flag. Dies wird von Rechner A bestätigt. Rechner B hat seine Verbindung abgebaut und auch Rechner A kann nach einem Timeout von 2 MSL (maximalen Segmentlebenszyklen auf dem Netzwerk) die Verbindung beenden, das heißt, alle Zustandsinformationen über die Verbindung löschen.

### 3.5.3 Das Paketformat von TCP

Wie schon bei den anderen besprochenen Protokollen wird auch den Daten, die mit TCP versendet werden, ein Header mit Protokollinformationen vorangestellt. In Abb. 3.9 ist das Format des Headers dargestellt, das im folgenden kurz erläutert werden soll.

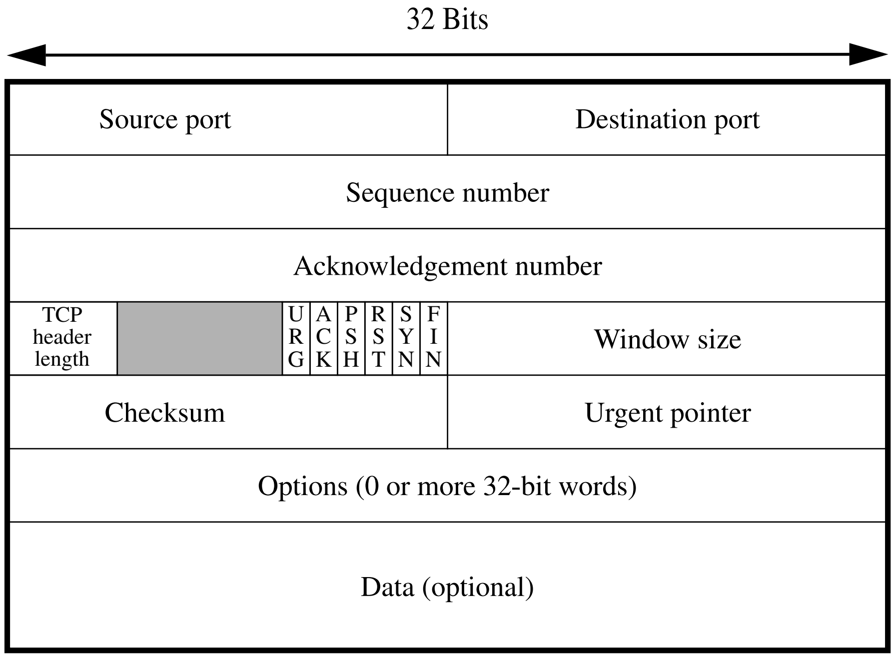


**Abb. 3.9 Das TCP-Header Format (aus [TCP81])**

Die ersten beiden Einträge sind wieder die Portnummer der Quelle und des Ziels. Darauf folgen die Sequenznummer und die Acknowledgement-Nummer. Diese beiden Nummern werden zur Sicherstellung einer zuverlässigen Übertragung benötigt. Die Sequenznummer gibt die Nummer des ersten Octets in diesem Segment an. Falls das SYN Flag gesetzt ist (beim Verbindungsaufbau) ist die Sequenznummer die initiale Sequenznummer und das erste Daten-Octet hat die Nummer Sequenznummer + 1.

Die Acknowledgement-Nummer - 1 gibt die Position im Datenstrom an, bis zu dem alle Octets empfangen wurden, d.h. die Acknowledgement-Nummer selbst ist die Sequenznummer, die vom Rechner, mit dem kommuniziert wird, bei dem nächsten übertragenen Segment erwartet wird.

Der Eintrag “TCP header length” zeigt an, wo die Daten im empfangenen Paket beginnen und zeigt somit auch die Größe des Headers an. Dabei wird der Wert in 32-Bit Worten angegeben.

Der Reserved Eintrag wird derzeit nicht benutzt und muss den Wert null haben.

Darauf folgen die sechs Control-Bits mit folgenden Bedeutungen:

- URG: das Urgent Pointer Feld ist gültig.
- ACK: das Acknowledgement-Nummer Feld ist gültig.
- PSH: Push Funktion.
- RST: Die Verbindung soll zurückgesetzt werden.
- SYN: Verbindungsaufbau; die Sequenznummern sollen synchronisiert werden.
- FIN: Verbindungsabbau; es folgen keine weiteren Daten mehr vom Sender.

Mittels des URG Flags und des Urgent Pointer Eintrags kann eine wichtige Nachricht an den Empfänger übertragen werden. Dabei muss die Möglichkeit bestehen, die Applikationsschicht asynchron über den Empfang dieser Daten zu informieren, damit diese notwendige Schritte unternehmen kann. Der Sinn dieser Funktion ist, dass z.B. ein Befehlsabbruch wie bei einer Telnet-Sitzung sofort zur Anwendung durchgereicht wird und nicht möglicherweise an das Ende des Empfangspuffers kommt, weil die Anwendung die vorherigen Daten noch nicht verarbeitet hat. Das URG Flag gibt an, dass der Eintrag im Urgent Pointer Feld gültig ist. Dieser zeigt dabei auf das letzte Octet der “Urgent Data”.

Das TCP-System hat die Möglichkeit, die von einer Anwendung übergebenen Daten, welche versendet werden sollen, zu puffern und erst bei einer ausreichenden Datenmenge ein Paket zu versenden. Genauso kann auf der Empfangsseite gepuffert werden und die Daten nicht sofort an die Applikation weitergereicht werden. Mittels des Push-Flags kann dieses Verhalten beeinflusst werden. Auf der Sendeseite bedeutet dies, dass die Daten gleich versendet werden sollen, weil die Anwendung hohe Latenzanforderungen hat oder ein Kommunikationsdeadlock verhindert werden soll. Auf der Empfängerseite bedeutet das gesetzte Flag, dass die Daten gleich an die Anwendung weitergereicht werden sollen.

Ein gesetztes RST Flag bedeutet, dass ein schwerwiegender Fehler aufgetreten ist und die Verbindung sofort abgebrochen wird. Ein RST Flag veranlaßt den Rechner, sofort die Verbindung zu beenden und alle belegten Ressourcen freizugeben, was natürlich auch Datenverlust zur Folge haben kann. Dies ist nicht die normale Art, eine Verbindung zu beenden, sondern wird nur dann benutzt, wenn Fehler aufgetreten sind, die nicht behoben werden können, wie z.B. der Absturz eines Rechners. Für ausführlichere Erläuterungen siehe Kap. 3.5.9.

In dem *Window* Eintrag des Headers wird die Anzahl der Octets, angefangen bei dem Octet mit der Acknowledgement-Nummer, eingetragen, die das empfangende TCP derzeit bereit ist, aufzunehmen. Dieses Feld wird zur Flußkontrolle benötigt. Mehr dazu siehe in Kap. 3.5.4.

Im *Checksum* Feld wird eine Prüfsumme eingetragen, um fehlerhafte Pakete erkennen zu können. Diese Prüfsumme ist wieder das Einerkomplement der Summe der Einerkomplemente der 16-bit Wörter des TCP Pakets [BB88]. Dabei wird dem Paket wieder ein Pseudo-Header wie in Kap. 3.4.1 bei UDP vorangestellt, allerdings ist hier der Protokol-Eintrag sechs für TCP und im Längenfeld ist die TCP Länge eingetragen, die die Länge des Headers und der Daten in Octets angibt.

Der nächste Eintrag ist der *Urgent Pointer*, der oben bereits beschrieben wurde.

Darauf können noch *Optionen* im Header folgen, die insgesamt eine ganzzahlige Anzahl von Byte belegen. Falls die Optionen nicht auf eine vier-Byte Grenze enden, wird mit Fülldaten aufgefüllt, damit die folgenden Daten an einer 32-Bit Grenze beginnen. Im ursprünglichen RFC [TCP81] wurden drei Optionen spezifiziert. Eine Option signalisiert das Ende der Optionen-Liste, *NOP* ("No Operation") ist eine weitere Option, um die Optionen notfalls auf eine 32-Bit Grenze aufzufüllen und die dritte ist der schon angesprochene *Maxium Segment Size* Wert, mit dem die TCP-Einheiten der verschiedenen Rechner die maximal akzeptierte Segmentgröße aushandeln können. Später kamen weitere Optionen dazu, u.a. deshalb, da für bestimmte Verbindungsarten (Satellitenlink) bzw. neuere Hochgeschwindigkeitsnetzwerke die im Header von TCP vorgesehenen Einträge nicht mehr ausreichend waren. Mit der Definition von neuen Optionen konnten diese Probleme gelöst werden, ohne den Header ändern zu müssen und ferner bleibt dabei auch die Abwärtskompatibilität erhalten, da die neuen Optionen derart definiert wurden, dass alte Systeme, die diese nicht kennen, sie ignorieren können und trotzdem eine normale Datenübertragung möglich ist. Hinzugekommen sind unter anderem eine Option, um den Window-Eintrag, welcher nur ein 16 bit Wert ist, zu vergrößern und eine Option, um eine genauere Berechnung der Laufzeiten von Paketen zu ermöglichen. Allerdings ist die maximale Größe des Optionsfeldes mit 40 Byte beschränkt und dadurch können nicht beliebig viele Optionen kombiniert werden, da die neueren Optionen relativ viel Platz benötigen. Somit ist die Erweiterbarkeit des Protokolls durch die Optionen zwar prinzipiell möglich, aber es werden wohl kaum viele neue Optionen spezifiert werden, da z.B. die Verwendung der Timestamp-Option allein schon den gesamten Platz für Optionen verbrauchen kann. Eine genauere Betrachtung u. a. der neueren Optionen befindet sich in Kap. 3.5.11 [Bra89].

### 3.5.4 Flußkontrolle

TCP bietet mit dem *Window* Eintrag im Header eine Möglichkeit zur Flußkontrolle, d.h. jede TCP-Einheit kann anderen mitteilen, wieviele Daten sie derzeit verarbeiten kann. Somit können auch Rechner zuverlässig miteinander kommunizieren, die beispielsweise unterschiedliche

CPU-Leistung oder verschiedene Netzwerkbandbreiten besitzen, ohne dass die Gefahr besteht, dass der eine Rechner den anderen mit zu vielen Daten “überschwemmt” oder der andere Rechner zu wenig sendet, da er meint, dass sein Gegenüber mit mehr nicht zurechtkommen würde. Das verwendete Verfahren ist ein sogenanntes “Sliding window” Verfahren. Bei TCP sind alle Daten durchnummeriert (Sequenznummern); ein Rechner sendet nun seinem Gegenüber die Anzahl von Daten, die er akzeptiert. Dieser Wert bezieht sich dabei immer auf die bisher als empfangen bestätigten Daten, die als Acknowledgement-Nummer im Paket verzeichnet sind. Dabei kann man sich diesen Wert wie ein Fenster über den Datenstrom vorstellen. Links von diesem Fenster liegen schon versendete und auch bestätigte Daten. Rechts vom Fenster befinden sich die Daten, die noch nicht versendet werden können, da sie außerhalb des Fensters liegen. Die aktuelle Position, bis zu der Daten versendet wurden, liegt irgendwo innerhalb des Fensters. Diese Daten sind jedoch noch nicht bestätigt. Rechts von dieser Position bis zum Ende des Fensters befinden sich die Daten, die noch ohne Verzögerung versendet werden können. In Abb. 3.10 ist dies nochmals in einer Grafik dargestellt.

Dieses Fenster kann sich nun nach rechts verschieben, wenn der Empfang von Daten bestätigt wurde. Somit fallen wieder neue Daten in das Fenster, die nun ohne Verzögerung übertragen werden können.

Eine TCP-Implementierung sollte jedoch ein Fenster nicht schrumpfen, d.h. den rechten Fensterrand nach links verschieben, ohne dass sich das Fenster selbst verschoben hat. Solch eine Verkleinerung des Fensters, ohne dass sich das Fenster verschoben hat, also Daten bestätigt wurden, kann dazu führen, dass der nutzbare Fensterbereich (der Bereich ab der Endposition der bisher gesendeten Daten bis zum Ende des Fensters) sogar negativ wird und somit die TCP-Implementierung der sendenden Seite verwirrt. In solch einem Fall sollte der sendende Rechner keine neuen Daten mehr senden, sondern die alten unbestätigten Daten bis zum Fensterende wiederholen [TCP81].

Mittels des *Window* Eintrags kann nun die übertragene Datenmenge kontrolliert werden, indem der Empfänger einen kleineren Wert einträgt, wenn sich sein Puffer für den Empfang füllt. Es ist auch möglich, eine *Window* Größe von null anzugeben, was bedeutet, dass derzeit keine Daten verarbeitet bzw. empfangen werden können. Kann der Empfänger wieder Daten annehmen, weil beispielsweise die Daten vom Anwendungsprozeß entgegen genommen wurden, so trägt der Empfänger wieder einen größeren Wert in das *Window* Feld ein. Aber auch im Falle einer *Window* Größe von null muss der sendende Rechner in regelmäßigen Abständen Pakete senden (sogenanntes “Probing”) und der empfangende Rechner muss trotz dieser Fenstergröße das empfangene Paket bestätigen und seine nächste erwartete Sequenznummer angeben. Der Grund für dieses Verhalten ist, dass die Änderung der Fenstergröße von einem Fenster mit Größe null auch zuverlässig dem anderen Rechner mitgeteilt wird, da Acknowledgement-Segmente ohne Daten nicht zuverlässig übertragen werden und somit ein Deadlock auftreten kann, da der Rechner ewig darauf wartet, dass das *Window* wieder den Versand von Daten erlaubt.

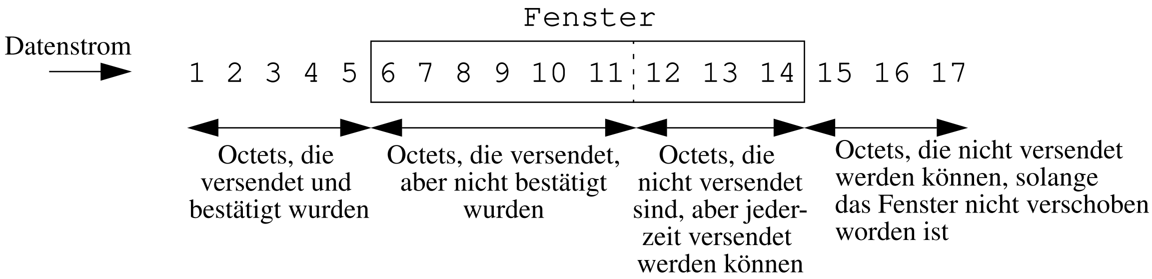


**Abb. 3.10 Sliding Window Verfahren (aus [Siy97])**

Bei Verbindungen, bei denen der Datenfluß nur in eine Richtung erfolgt, wird die Fenstergröße immer in Paketen übertragen, welche die gleiche Sequenznummer tragen, da der andere Rechner ja selbst keine Daten sendet. Daher können die Pakete nicht in die richtige Reihenfolge gebracht werden, falls sie nicht in der Sendereihenfolge eintreffen und die Information über die Fenstergröße kann dann auf veralteten Informationen beruhen. Dies ist aber einigermaßen leicht zu umgehen, indem immer die Fenstergröße aus dem Paket mit der höchsten Acknowledgement-Nummer genommen wird und so sichergestellt wird, dass die aktuellste Information verwendet wird.

Die Möglichkeit mit dem *Window* Eintrag den Umfang der Datenübertragung zu steuern, kann einen großen Einfluß auf die Performanz der Übertragung haben. Ein Problem einfacher Implementierungen sowie einige Hinweise zur Flußkontrolle bezüglich Performanz werden im folgenden Kapitel gegeben.

### 3.5.5 Das Silly-Window Syndrome

Das sogenannte "Silly-Window Syndrome" ist ein bekanntes Verhalten früher TCP-Systeme, die sich zwar nach der Spezifikation korrekt verhalten, aber unter gewissen Umständen wegen des einfachen *Window*-Managements eine sehr geringe Performanz erreichen. Es kann passieren, dass das TCP-System pro Octet ein Segment verschickt und jedes Octet dann einzeln bestätigt wird. Die Performanz und natürlich auch das Nutzdaten- zu Protokolldaten-Verhältnis ist unter diesen Umständen natürlich sehr schlecht.

Das beschriebene Verhalten kann aus zwei Gründen auftreten: einerseits kann auf der Empfängerseite der Puffer voll sein und eine Applikation liest daraus Byte für Byte. Das TCP-System schickt nachdem ein Byte Platz im Puffer freigeworden ist, einen *Window*-Wert von eins. Der andere Rechner füllt darauf den Puffer wieder mit einem Byte und dies geht dann solange, bis die sendende Seite keine Daten mehr zu versenden hat und der Puffer auf Empfangsseite sich lehrt. Eine weitere Möglichkeit, dass es zu diesem Verhalten kommt, ist der Fall, dass eine Applikation die Daten Byte-weise erzeugt und das TCP-System diese sofort versendet und somit auch immer nur ein Octet pro Segment verschickt. Man sieht, dass im ersten Fall die Ursache auf der Empfängerseite liegt, der zweite Fall wird dagegen durch die Sender-Seite verursacht.

Es gibt einige Möglichkeiten, das oben beschriebene Verhalten zu verhindern. Auf der Empfängerseite kann beispielsweise das Versenden einer neuen Fenstergröße solange hinausgezögert werden, bis sie einen größeren Wert angenommen hat, z.B. das Minimum der halben Fenstergröße und der maximalen Segmentgröße. Wird dies eventuell noch mit einem Timeout kombiniert, nachdem auf jeden Fall eine Änderung der Fenstergröße weitergegeben werden muss, auch wenn dieses noch nicht obiger Entscheidungsregel entspricht, so wird dadurch keine zu kleine Fenstergröße mehr bekanntgegeben und das Problem wird auf der Empfängerseite nicht mehr auftreten. Außerdem können auf der Empfängerseite auch die Acknowledgement-Pakete etwas verzögert werden. So wird einerseits Netzbandbreite gespart, da nicht für jedes empfangene Paket ein Acknowledgement geschickt wird, ferner ist die Wahrscheinlichkeit groß, dass in der Verzögerungszeit der Puffer durch die Applikation mehr geleert werden kann und somit keine zu kleine *Window*-Größe bekannt gegeben wird. Allerdings besteht hierbei die Gefahr, dass bei einer zu langen Verzögerung der Sender annimmt, dass die Pakete verloren gegangen sind und sie nochmals sendet. Daher wird in der Praxis die Verzögerung, die maximal in Kauf genommen werden kann, kaum eine halbe Sekunde überschreiten.

Eine dritte Möglichkeit, die allerdings nicht von den Standards empfohlen wird, ist, eingehende Pakete sofort zu bestätigen, aber die Window-Größe solange nicht zu ändern, bis ein bestimmter Minimalwert erreicht worden ist.

Die drei beschriebenen Vorgehensweisen versuchen, dass Problem auf der Empfängerseite zu verhindern. Aber auch auf der Senderseite gibt es hierfür einen Algorithmus, den sogenannten "Nagle avoidance algorithm". Der Algorithmus versucht Situationen zu verhindern, die z.B. bei einer Verbindung zu einem Editor über Telnet auftreten können. Hier werden Daten Byte für Byte erzeugt, beispielsweise, wenn der Benutzer eine Taste drückt. Wird nun jeder Tastendruck in einem TCP Segment versendet, wird sehr viel Netzlast erzeugt, der ein ganz geringer Nutzdatenanteil gegenübersteht (1 Byte Daten, 40 Byte IP- und TCP-Header für das erste Datenpaket, dann die Bestätigung dieses Pakets sowie ein Paket für Auffrischen des Editorbildschirms wegen dem neuen Zeichen und wiederum eine Bestätigung dafür). Der Algorithmus von Nagle versucht die Daten gewissermassen zu sammeln. Er sendet das erste Segment (also beispielsweise das erste Byte) und puffert die weiteren Bytes so lange, bis das erste Segment bestätigt wird. Darauf werden die gepufferten Daten versendet und wieder bis auf die nächste Bestätigung gewartet. Dabei "taktet" sich der Algorithmus quasi selbst. Je schneller die Acknowledgement-Pakete zurück kommen, desto schneller können auch wieder neue gesendet werden. Zusätzlich können aber Daten auch nach einer gewissen Zeit versendet werden, falls bis dahin keine Bestätigung erhalten wurde oder wenn mehr Daten gepuffert sind als z.B. die halbe Fenstergröße oder falls ein Segment komplett gefüllt werden kann. Dieser Vorgehen hat vor allem dann Erfolg, wenn der Nutzer einigermassen flott tippt und das Netzwerk langsam ist. Jedoch ist die Reaktion des Editors anfangs etwas ungewöhnlich und es gibt Situationen, wo dieser Algorithmus eher Nachteile bringt, z.B. bei einer X-Window Sitzung, bei der die Mausbewegungen so schnell wie möglich weitergeleitet werden sollten. Daher fordert der RFC [Bra89], dass der Algorithmus abschaltbar sein muss. Dabei wird man mit dem Algorithmus beispielsweise bei einer Telnet-Verbindung über ein lokales Netz sowieso eher nicht konfrontiert werden, da die “Roundtriptime” (Zeit von Versand des Pakets bis zu dessen Bestätigung) so gering ist, dass man wohl mindestens 50 Zeichen pro Sekunde tippen müsste, um hier Auswirkungen wie Verzögerungen durch das Puffern und Warten auf die Bestätigung zu bemerken.

### 3.5.6 Vermeidung der Überlastung des Netzwerks

Neben dem bereits in Kap. 3.5.4 beschriebenen *Window* Eintrag zur Flußkontrolle verwalten moderne TCP-Implementierungen noch ein zweites Fenster, um Überlastung und als Folge die Verstopfung des Netzes zu verhindern. Der verfolgte Ansatz ist dabei relativ simpel: ist das Netz überlastet, werden keine neuen Pakete dem Netzwerk übergeben, solange kein altes Paket das Netzwerk verlassen hat, was bei TCP bedeutet, solange keine Bestätigung zurück gekommen ist.

Das Modell, um eine Überlastung zu erkennen, ist ebenso relativ einfach: in der heutigen Zeit hat der Paketverlust weniger Übertragungsfehler als Ursache, da abgesehen von verbindungsloser Übertragung die Netze wie z.B. die glasfaserbasierten Netzwerke sehr zuverlässig sind. Vielmehr wird ein Paketverlust als Indikator für eine Überlastung angesehen.

Neben dem gewöhnlichen Window-Wert, der im Prinzip die Empfangskapazität des Kommunikationspartners beschreibt wird ein zweiter Window-Wert für die Netzwerkkapazität verwaltet (sogenanntes “congestion window”). Auch dieser beschreibt die Menge an Daten, die der Sender versenden kann. Tatsächlich kann dann das Minimum der beiden Fenster versendet werden.

Um Überlastung des Netzes zu vermeiden, wird der zweite Fensterwert folgendermassen verwaltet: anfangs wird er mit der maximalen Segmentgröße initialisiert. Dann wird ein Segment versendet. Wird dieses innerhalb der Timeout-Zeit bestätigt, so wird der Fensterwert verdoppelt. Nun beträgt der Wert zwei mal die maximale Segmentgröße und es werden zwei Segmente verschickt, die nun innerhalb der Timeout-Zeit bestätigt werden müssen. Geschieht dies, wird das Fenster wieder verdoppelt. Die Fenstergröße wächst also exponentiell. Dies tut sie solange, bis nicht mehr alle Pakete innerhalb des Timeouts bestätigt werden. Dies ist der Punkt, an dem man davon ausgeht, dass zuviel Daten versendet wurden. Die *Window*-Größe muss also verkleinert werden. Um sich an den optimalen Wert heranzutasten, wird noch ein weiterer Parameter verwendet, ein spezieller Schwellenwert, der anfangs mit 64K initialisiert ist. Erreicht nämlich das *Window* den Schwellenwert, wird es nicht mehr exponentiell vergrößert, sondern nur noch linear um eine maximale Segmentgröße pro erfolgreiche Bestätigung innerhalb des Timeouts. Werden nicht alle Bestätigungen erhalten, so wird der Schwellenwert halbiert und die Fenstergröße auf ein Segment zurückgesetzt. Der Vorgang beginnt erneut von vorne und das Fenster wird die neue halbierte Schwelle anhand seines anfangs exponentiellen Wachstums schnell erreichen oder es wird erneut zurückgesetzt, falls der Wert immer noch zu groß war und somit Pakete verloren gingen. Falls dann die Schwelle passend zur Netzwerkkapazität ist und keine Pakete mehr verloren gehen, wird das “congestion window” dann solange wachsen, bis es die Fenstergröße zur Flußkontrolle des Empfängerrechners erreicht hat [Ste94].

### 3.5.7 Initiale Sequenznummern

Anhand der Sequenznummern ordnet TCP die Datenpakete und bringt sie gegebenenfalls in die richtige Reihenfolge. Falls jedoch Pakete einer alten Verbindung noch auf dem Netzwerk sind, die die gleichen Sequenznummern tragen wie die Pakete einer neuen Verbindung, so kann TCP dies nicht erkennen und liefert somit falsche Daten an die Anwendung. Dies kann leicht passieren, falls eine Verbindung geschlossen wird und kurz später erneut aufgebaut und dabei jedesmal mit denselben Sequenznummern begonnen wird; um dies zu verhindern, fordert der RFC [Bra89], dass die anfangs gewählten Nummern von einer Uhr abhängig gemacht werden, die einen 32-bit Zähler etwa alle vier Millisekunden erhöht und somit alle 4,55 Stunden sich wiederholt. Unter der Annahme, dass Segmente nicht länger als die maximale Segmentlebensdauer auf dem Netz bleiben und diese geringer ist als 4,55 Stunden, sind die Sequenznummer nun eindeutig.

Stürzt ein Rechner jedoch ab, verliert er normalerweise sein Wissen über bereits verwendete Sequenznummern und läuft Gefahr, dass er Sequenznummern benutzt, die noch im Umlauf sind, falls er einfach wieder beginnt, Segmente zu versenden. Für diesen Fall fordert der Standard, dass das System mindestens eine maximale Segmentlebensdauer (MSL - maximum segment lifetime) abwartet, bevor wieder gesendet wird, wobei die MSL auf zwei Minuten festgelegt wurde. Falls beispielsweise auf stabilem Speicher protokolliert wird, welche Nummern schon verwendet wurden, muss nach einem Absturz natürlich nicht gewartet werden.

Dadurch, dass bei TCP jedes Octet eine Nummer erhält, werden die Sequenznummern sehr schnell verbraucht. Es muss jedoch sichergestellt sein, dass sich die Sequenznummern nicht wiederholen, bevor die zu diesen Nummern gehörigen Daten nicht erfolgreich versendet und bestätigt worden sind und mögliche Wiederholungen der Segmente ebenfalls vom Netz verschwunden sind. Die Lösung hierzu war, den Sequenznummernraum sehr groß zu machen und somit die Wahrscheinlichkeit von Wiederholungen zu minimieren. Dazu wurde ein Raum von 2^32 Sequenznummern gewählt. Dieser wird bei einer Senderate von 2MBit/sec in 4.5 Stunden aufgebraucht. Bei den allerdings heute gebräuchlichen Netzen z.B. mit 100MBit/sec wiederholen sich die Nummern schon nach 5.4 Minuten und bei einem Gigabit-Ethernet ist die Zeit, bis die Nummer verbraucht sind schon unter der damals im RFC angenommenen MSL von zwei Minuten. Für dieses Problem wurde allerdings ein Algorithmus entwickelt, der auch noch bei Netzwerken, die wesentlich schneller sind als heutige Technologien, verhindert, dass durch zu schnellen Verbrauch der Sequenznummern fälschlicherweise alte Pakete akzeptiert werden. Dieser wird in Kap. 3.5.11 beschrieben.

Obige Probleme müssen bei der Implementierung berücksichtigt werden und es ist beispielsweise abzuwägen, ob nach einem Absturz geraume Zeit gewartet wird, um sicherzustellen, keine noch auf dem Netz sich befindenden Sequenznummern zu verwenden oder einfach wieder zu senden mit dem Risiko, dass beim Empfänger möglicherweise Verwirrung gestiftet wird (hierbei wird angenommen, dass das System keinerlei Informationen mehr über vor dem Absturz verwendete Sequenznummern besitzt). Allerdings sollte man nicht vergessen, dass solche Probleme nur auftreten, falls dieselbe Verbindung wie vorher erneut nach einem Absturz geöffnet wird (quasi eine erneute Inkarnation dieser Verbindung) und außerdem war die Wahl der MSL mit 2 Minuten eine willkürliche Festlegung, mit der man einen sicheren Puffer schaffen wollte.

### 3.5.8 Timer-Verwaltung von TCP

Timer sind eine essentielle Eigenschaft von TCP, da sie z.B. zur Implementierung der sicheren Datenübertragung benötigt werden. Mittels des "Retransmission Timers" werden Pakete, die nicht innerhalb der Zeit des Timers bestätigt werden, erneut gesendet. Hierzu wird für jedes versendete Segment ein Timer gestartet; wird das Paket nicht innerhalb des Timeouts bestätigt, wird davon ausgegangen, dass das Paket auf dem Netz verloren gegangen ist. Deshalb wird das Paket erneut versendet und der Timer zurückgestellt. Eventuell sollte hierzu eine obere Grenze vorgegeben werden, wie oft ein Paket wiederholt versendet wird. Falls dann immer noch keine Bestätigung eingetroffen ist, muss ein schwerwiegenderes Problem vorliegen, z.B. dass der Zielrechner abgestürzt ist.

Wie lange jedoch gewartet werden soll, bevor das Paket erneut versendet wird, hängt stark vom Zielrechner und dem Netzwerk bzw. den Netzwerken auf dem Weg zu diesem Rechner ab. Daher kann kein fester Timeout-Wert gewählt werden, sondern dieser muss dynamisch ermittelt werden, um nicht unnötig Pakete zu wiederholen oder aber zu lange zu warten. Die heute verwendete Lösung basiert auf einem Vorschlag von Jacobson kombiniert mit einer Verbesserung von Karn. Das zu Grunde liegende Prinzip ist, zu messen, wie lange es vom Versand eines Pakets bis zu dessen Bestätigung dauert. Hierzu wird ebenso ein Timer verwendet, der gestartet wird, wenn ein Paket versendet wird. Für jede Verbindung berechnet TCP eine Variable RTT, die eine Schätzung der "Round trip time" darstellt. Diese Variable wird nun nach folgender Formel rekursiv berechnet:

$$RTT = \alpha RTT + (1 - \alpha)M$$

Dabei ist $\alpha$ ein Faktor, mit dem der alte Wert gewichtet wird; gewöhnlich wird für diesen Wert 7/8 gewählt.

Aber auch mit einer guten Abschätzung der "Round trip time" ist die Wahl des optimalen Timeouts relativ schwierig. Anfangs wurde RTT mit einem zu wählendem Faktor $\beta$ multipliziert, meist war dieser zwei. Allerdings stellte sich diese Vorgehensweise als zu unflexibel heraus und der geeignete Timeout wird deswegen nach einem etwas aufwendigeren Verfahren bestimmt. Hierzu muss eine Variable D nach folgender Formel bestimmt werden:

$$D = \alpha D + (1 - \alpha)|RTT - M|$$

Dieses D ist ein Maß für die Abweichung, wenngleich sie nicht exakt der Standardabweichung entspricht. $\alpha$ ist dabei wieder ein Gewichtungsfaktor und muss nicht dem obigen entsprechen. Mit der Berechnung von D über die Differenz der erwarteten "Round trip time" und der tatsächlichen Zeit wird der Timeout-Wert meist nach folgender Formel bestimmt:

$$Timeout = RTT + 4D$$

Die Wahl des Faktor vor D ist wie schon öfters eine Entscheidung aus der Praxis, unter anderem auch deshalb, weil es sehr schnell durch einen "Bitshift" zu implementieren ist. Außerdem hat die Praxis gezeigt, dass ein Timeout zusammengesetzt aus der geschätzten "Round trip time" plus 4 Standardabweichungen fast alle Pakete erfasst und ziemlich nahe einem optimalen Wert liegt. Für die Initialisierung von RTT und des Timeout werden im Standard Werte vorgegeben, die sich in der Praxis bewährt haben (RTT = 0 sec, Timeout = 3 sec, D muss derart gewählt werden, dass es zu diesem Timeout führt).

Die Ergänzung von Karn zu diesem Algorithmus behandelt das Problem, welches auftritt, wenn für ein Segment der Timeout abläuft und dadurch das Segment erneut gesendet wird. Eine daraufhin erhaltene Bestätigung kann nicht mehr dem ursprünglichen Segment oder seiner Wiederholung zugeordnet werden, was bei falscher Zuordnung die Berechnung von RTT stark verfälschen kann. Daher wird bei Segmenten, die mehrfach gesendet wurden, der Wert von RTT nicht aufgefrischt, sondern der Timeout jedesmal verdoppelt, bis die Segmente ohne eine wiederholte Versendung bestätigt werden.

Neben diesem Timeout-Timer verwendet TCP noch einige andere Timer, um bestimmte Funktionen zu unterstützen. Wie bereits erwähnt, werden ACK-Pakete ohne Daten nicht sicher übertragen. Im Falle eines *Window*-Eintrags von null wartet der sendende Rechner auf eine Änderung des Fensters, um wieder senden zu können. Geht jedoch diese Änderung auf dem Netzwerk verloren, so wird dieser Rechner ewig warten. Um dies zu verhindern wir ein weiterer Timer eingesetzt, der nach seinem Ablauf ein Probe-Segment an den anderen Rechner versendet, der darauf mit der Fenstergröße antwortet. Durch diese Art des "Polling" wird der sendende Rechner dann sicher von einer geänderten Fenstergröße erfahren.

Ein weiterer Timer, der allerdings nicht unbedingt eingesetzt werden muss, ist der Timer zum Versand von sogenannten "Keep-alive" Segmenten. Falls eine Verbindung lange Zeit unbenutzt war und keinerlei Daten versendet wurden, wird nach Ablauf dieses Timers überprüft, ob der Kommunikationspartner noch da ist und seinerseits die Verbindung auch noch aufrecht erhält. Falls nicht, wird die Verbindung abgebrochen. Dieses Vorgehen ist umstritten, da es Bandbreite kostet und es auch dazu führen kann, dass eine ansich intakte Verbindung abgebrochen wird, da gerade zur Zeit des Probings Kommunikationsprobleme des Netzwerks bestehen [TCP81], [Ste94], [Siy97].

### 3.5.9 Reset-Segmente und die Verarbeitung von Reset-Segmenten

In Kap. 3.5.3 wurde das RST-Bit schon kurz erläutert. Ein gesetztes RST-Bit zeigt im Allgemeinen das Auftreten eines Fehlers an, woraufhin die Verbindung abgebrochen werden soll. Ein Segment mit gesetztem RST-Flag wird immer dann versendet, wenn ein Segment eintrifft, bei dem man sicher weiß, dass es nicht für die Verbindung bestimmt ist.

Man kann dabei drei verschiedene Situationen unterscheiden, abhängig davon, in welchem Zustand das empfangende TCP sich befindet.

- Im CLOSED Zustand existiert keinerlei Verbindung und auf alle ankommenden Segmente (außer, es ist auch ein Reset-Segment) wird ein RST zurückgesendet. Die Sequenznummer des RST-Segments hängt vom empfangenen Segment ab. Ist dort ein gültiger Acknowledgement-Wert eingetragen, so wird dieser übernommen, ansonsten hat das RST Segment Sequenznummer null und das Acknowledgement wird auf die Summe der Sequenznummer des empfangenen Segments und dessen Länge gesetzt. Dabei bleibt das System im selben Zustand (hier CLOSED).
- In unsynchronisierten Zuständen (LISTEN, SYN-SENT, SYN-RECEIVED) wird ein RST gesendet, wenn ein empfangenes Segment etwas bestätigt, was noch gar nicht versendet wurde. Die Wahl der Sequenznummer und eventuell der Acknowledgement-Nummer erfolgt wie oben. Auch hier wird der Zustand nicht verlassen.
- Falls das System sich in einem synchronisierten Zustand befindet (ESTABLISHED, FIN-WAIT-1, FIN-WAIT-2, CLOSE-WAIT, CLOSING, LAST-ACK, TIME-WAIT) wird bei falschen Segmenten (nicht akzeptierbare Acknowledgement-Nummer oder Sequenznummer außerhalb des Fensters) kein RST gesendet, sondern nur ein leeres ACK-Segment, in dem die aktuelle Sende-Sequenznummer und die nächste erwartete Sequenznummer verzeichnet ist. Der Zustand wird beibehalten.

In den Zuständen außer dem CLOSED-Zustand können nach dem ursprünglichen RFC [TCP81] auch RST Segmente versendet werden, falls die “Precedence” und “Security” Einträge, die aus dem IP-Header entnommen werden können, nicht übereinstimmen. Allerdings werden diese Einträge in der Praxis auch auf IP-Ebene von Routern nicht beachtet und daher wird hier nicht weiter darauf eingegangen.

Bei den verschiedenen Fällen wurde schon darauf eingegangen, wie die Sequenznummer des RST Segments gewählt werden muss. Ein Reset Segment wird anhand seiner Sequenznummer überprüft und ist nur gültig, falls diese innerhalb des Fensters liegt. Eine Ausnahme ist ein RST als Antwort auf eine initiale Synchronisationsaufforderung (SYN Segment); hier sind die Sequenznummern ja noch nicht synchronisiert, das RST Segment ist dann gültig, wenn es mittels der Acknowledgement-Nummer das SYN bestätigt.

Bei Empfang eines RST geht das System in einen anderen Zustand über, vorrausgesetzt natürlich das RST Segment war gültig nach den oben genannten Bedingungen. Normalerweise wird die Verbindung abgebrochen und in den CLOSED Zustand gewechselt, nur beim LISTEN Zustand wird dieser beibehalten oder auf diesen zurückgesprungen, falls das System vorher in diesem war und beim Empfang von RST in SYN-RECEIVED war.

### 3.5.10 Zustände des TCP-Systems

Wie in den vorherigen Kapiteln schon mehrfach angesprochen, liegt TCP das Modell eines endlichen Zustandsautomaten zugrunde bzw. ist die Funktion von TCP gut mittels eines Zustandsautomaten darstellbar. Dabei ist ein endlicher Zustandsautomat ein System, welches interne Zustände besitzt und durch entsprechende Ereignisse von einem Zustand in einen anderen wechseln kann. Beim Zustandsübergang kann eine Ausgabe, eine Reaktion erfolgen. In Abb. 3.11

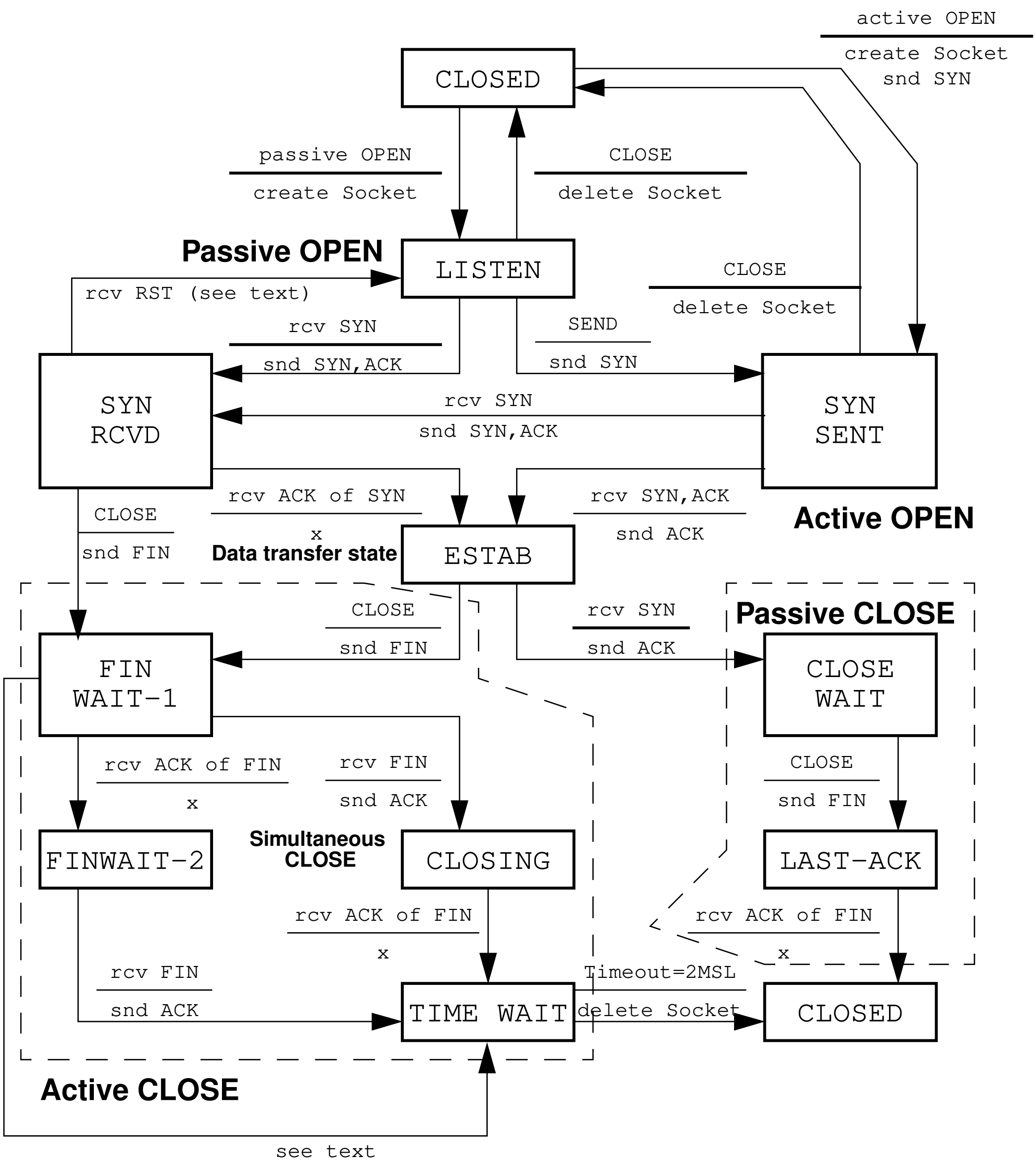


**Abb. 3.11 Der TCP-Zustandsgraph (aus [Siy97] und [TCP81])**

sind die Zustände des TCP-Systems sowie die möglichen Übergänge dargestellt. Diese Graphik soll nur als Überblick dienen und bietet eine Möglichkeit, konkrete Fälle wie den Verbindungsaufbau in Abb. 3.7 am Zustandsgraphen nachzuvollziehen.

In diesem Graphen wurden schon die Korrekturen aus RFC 1122 [Bra89] gegenüber RFC 793 [TCP81] eingearbeitet. Das Konzept der Graphik ist wie folgt: Zustände sind als Rechtecke dargestellt, Pfeile zwischen ihnen geben Art und Richtung eines Zustandsübergangs an. Dabei gibt die Beschriftung eines Pfeiles an, was den Zustandsübergang auslöst und was während des Übergangs getan wird. Oberhalb des horizontalen Strichs steht der Auslöser, darunter die Aktion, die während des Übergangs stattfindet.

Dabei können Auslöser einerseits Aufrufe eines Anwenders sein oder der Empfang einer Nachricht. Beispielsweise wird durch ein “passive OPEN” vom Benutzer vom CLOSED Zustand in den LISTEN Zustand übergegangen und dabei ein Socket erzeugt. Socket bezeichnet in diesem Kontext die Verwaltungsstruktur für die Verbindung; der Begriff wurde gegenüber der Originalgraphik geändert, wo dies TCB (Transmission Control Buffer) genannt wurde, da der Begriff Socket näher der später dargestellten Implementierung entgegen kommt. Vom Zustand LISTEN wird nach SYN RCVD übergegangen, wenn ein SYN Paket empfangen wurde (rcv SYN). Dabei wird ein Paket mit gesetztem SYN und gesetztem ACK Flag versendet (snd SYN, ACK).

In der Graphik finden sich zwei Stellen, an denen auf den Text hingewiesen wird (see text: einmal langer Pfeil am unteren Rand, einmal bei Passive OPEN). Diese beiden Übergänge sind Korrekturen gegenüber der ursprünglichen Graphik, da diese Übergänge nicht eingezeichnet waren, aber in den Zustandsbeschreibungen aufgeführt wurden.

Die generelle Struktur in Abb. 3.11 wird schnell deutlich. Der obere Bereich enthält die Zustände, die beim Verbindungsaufbau durchlaufen werden und in der Mitte befindet sich CONNECTION ESTABLISHED; hier findet die Datenübertragung statt. Im unteren Bereich befinden sich die Zustände, die bei einem Abbau der Verbindung durchlaufen werden, ebenso wie beim Aufbau mit einer Unterscheidung zwischen dem aktiven und dem passiven Vorgang.

### 3.5.11 Neuere Erweiterungen des TCP-Protokolls

Der ursprüngliche RFC, in dem TCP beschrieben wurde, ist mittlerweile fast 20 Jahre alt. Seitdem hat sich beispielsweise die Leistungsfähigkeit von Netzwerken um mehrere Zehnerpotenzen verbessert und die TCP/IP Protokolle sind auch durch die rasante Entwicklung des Internet zu den wohl meist benutzten Protokollen geworden. TCP hat sich seit seiner ersten Spezifikation weiterentwickelt, einerseits weil nach einiger Erfahrung mit verschiedenen Implementierungen und dem Betrieb unter verschiedensten Rahmenbedingungen festgestellt werden konnte, wo Schwachpunkte liegen, die beispielsweise die Performanz des Protokolls beeinträchtigen, andererseits wird heute TCP auf Netzwerken betrieben, von denen man zur Zeit der Protokollspezifikation noch nichts ahnen konnte. Dies zeigt sich schon bei dem in Kap. 3.5.7 beschriebenen Problem mit dem schnellen Verbrauch der Sequenznummern bei Gigabit-Netzwerken. In späteren RFCs wurden u.a. Fehler in der ursprünglichen Definition behoben, darüber hinaus einige neue Algorithmen eingeführt und alte wie z.B. der Algorithmus zur Bestimmung der “Roundtriptime” verbessert. Dieses Kapitel gibt einen kurzen Überblick über neuere Erweiterungen und Algorithmen des TCP-Protokolls (wobei mit “neuere” diejenigen Algorithmen gemeint sind, die erst in späteren RFCs beschrieben wurden), wobei auch die neu eingeführten Optionen besprochen werden, mittels derer das Protokoll in einer abwärtskompatiblen Weise erweitert wurde. Dies ist ein Kernpunkt aller hier besprochenen Erweiterungen. Diese sind immer derart gestaltet, dass auch alte Systeme, die sich spezifikationsgemäß verhalten, damit zurechtkommen und eine Datenübertragung möglich ist, wenngleich vielleicht dann nicht mit optimaler Performanz.

#### 3.5.11.1 Bestimmung der MTU entlang des Sendewegs ("path MTU discovery")

Normalerweise orientiert sich TCP beim Versand von Segmenten an der MTU des zugrunde liegenden Netzwerks. Bei einem Transport über viele Zwischenstationen jedoch ist nicht klar, wie groß ein Segment nun maximal sein sollte. Darüberhinaus kann jedes IP-Paket anders weitergeleitet werden. Neuere Systeme bieten die Möglichkeit, die sogenannte "path MTU" zu bestimmen, d.h. die maximale Transfereinheit auf dem gesamten Weg vom Sender zum Empfänger. Dabei beginnt TCP nach Verbindungsaufbau mit der MTU des benutzten Netzwerks oder der maximalen Segmentgröße des Empfängers, falls dieser diese Option mit übermittelt hat (sonst wird standardmäßig 536 angenommen). Bei versendeten Paketen wird das *DF* Flag im IP-Header gesetzt, welches Router anweist, das Paket nicht zu fragmentieren. Falls nun doch ein Router auf dem Weg zum Ziel das Paket fragmentieren müßte, so sendet er einen ICMP-Fehler zurück. Daraufhin verkleinert das TCP seine Segmentgröße und übermittelt die Daten erneut. Um der Möglichkeit Rechnung zu tragen, dass sich die Route zu einem anderen Rechner ständig ändern kann, wird mittels eines Timers nach einer bestimmten Zeit nach Verkleinerung der MTU versucht, diese wieder zu vergrößern.

#### 3.5.11.2 Schnelleres Wiederholen von verlorenen Paketen ("fast retransmit and fast recovery")

Eine Modifikation des Algorithmus zur Vermeidung von Überlastungen des Netzwerks (siehe Kap. 3.5.6) sind die sogenannten "Fast Retransmit and Fast Recovery" Algorithmen. Hierbei wird ausgenutzt, das TCP bei Empfang eines Segments, dass nicht in der normalen Reihenfolge ankommt, eine Bestätigung schickt, die die nächste erwartete Sequenznummer angibt. Dies ist quasi ein Duplikat eines vorher versendeten Acknowledgements, da TCP das neu erhaltene Segment erst dann bestätigen kann, wenn alle Daten bis zu diesem Segment ohne Lücken dazwischen empfangen wurden. Kommt nun mehrmals dasselbe Acknowledgement-Paket, ist das ein starker Hinweis darauf, dass ein Paket verloren gegangen ist. Daraufhin wird das Segment, das verloren gegangen zu sein scheint erneut versendet, ohne auf das Ablaufen eines Timer zu warten. Dabei wird auch versucht, eine Überlastung zu vermeiden, allerdings nicht nach dem sogenannten "slow start" Schema aus Kap. 3.5.6 (Rücksetzen des "congestion window" auf eins), da noch ein Datenfluß zwischen den Rechnern existiert, da sonst auch nicht die Duplikate des Acknowledgement-Pakets versendet worden wären. Dabei wird das "congestion window" auf die halbe Größe plus die dreifache Segmentgröße zurückgesetzt und wächst dann linear bei jeder Ankunft eines weiteren schon erhaltenen Acknowledgement-Pakets. Wird schließlich das wiederholt gesendete Segment bestätigt, dass das Loch beim Empfänger geschlossen hat, so wird das “congestion window” auf die Größe des Fensters zurückgesetzt, den es hatte, als das Paket verloren gegangen ist (dieser wurde zu Beginn des Algorithmus gespeichert).

#### 3.5.11.3 Selektive Empfangsbestätigungen (“SACK”)

Eine noch moderne Version, Pakete erneut zu übertragen, ohne das Ende eines Timers abzuwarten ist die *Selective Acknowledgement* Option (SACK). Hierbei werden empfangene Pakete selektiv bestätigt, d.h. es können auch einzelne Segmente bestätigt werden, obwohl insgesamt noch Löcher in den empfangenen Daten vorhanden sind, da beispielsweise ein Segment verloren gegangen ist. Der Sender braucht daraufhin nur noch die Pakete erneut zu senden, die verloren gegangen sind. Dies ist ein erheblicher Vorteil gegenüber den gewöhnlichen kumulativen Acknowledgements, mit denen eine solche Information nicht zur Verfügung steht. Implementiert wird diese Bestätigungsmethode mit zwei neuen Optionen. Die erste, “SACK-permitted” wird in einem SYN-Pakete gesendet und gibt an, dass das System diese Art von Acknowledgements verarbeiten kann. Wurde eine Verbindung aufgebaut, die SACK versteht (angezeigt durch die Option im SYN-Paket), dann kann die eigentliche SACK-Option versendet werden. Diese besteht aus einem neuen Optionstyp, einer Längenangabe und daraufhin folgen Bereichsangaben, die schon empfangene Daten beschreiben. Die Beschreibung erfolgt wieder mit Sequenznummern in der Form [linker Rand des ersten Blocks][rechter Rand des ersten Blocks] usw. Durch die Beschränkung des TCP-Optionenfelds auf 40 Byte sind maximal 4 Einträge in einem Segment möglich. Werden zusätzlich andere Optionen wie die weiter unten beschriebene *Timestamp*-Option verwendet, so sind entsprechend weniger Einträge möglich. An der Bedeutung des normalen Acknowledgement Eintrages ändert diese neue Option nichts. Dieser Wert wird auch weiterhin nur dann geändert, wenn ein zusammenhängender Bereich von Daten beim Empfänger vorliegt. SACK dient dazu, Segmente zu bestätigen, die nicht-zusammenhängende Bereiche repräsentieren und dadurch den Sender zu veranlassen, nur noch die fehlenden Teile erneut zu versenden [MMF96].

#### 3.5.11.4 Expansion des Sende- und Empfangsfensters (“window scale option”)

Einige weitere Neuerungen wurden als Lösungen für Probleme entwickelt, die beim Datentransfer über sogenannte “long fat pipes” auftreten [JK88]. Wird eine Verbindung zwischen Sender und Empfänger als eine Röhre betrachtet, so kann man die Kapazität dieser Röhre aus dem Produkt der Bandbreite mit der Verzögerungszeit berechen. Dieser Wert stellt die Menge an Daten dar, die die Röhre ausfüllen und korrespondiert dadurch auch mit den benötigten Puffer- bzw. Fenstergrößen. Betrachtet man die Daten für moderne Übertragungstrecken wie z.B. eine transkontinentale Gigabitstrecke oder eine DS1-Satellitenverbindung, so ergeben sich hier Werte im Bereich von 10^6. Allerdings ist der Window-Wert auf 16 Bit festgelegt und beschränkt somit das Fenster auf maximal 65535 Bytes. Somit kann solch eine Verbindung nicht mehr voll ausgelastet werden und dies wird in Gegenwart von Paketverlusten noch schlimmer, da durch den Timer-Ablauf und die implementierten Methoden wie das Zurücksetzen des “Congestion window” auf ein Segment (“slow start algorithm”) die Kapazität nicht ausgenutzt werden kann. Es wurde daher eine neue Option definiert, mit der das Fenster auf 32 Bit expandiert werden kann. Dabei wird ein "scale factor" übermittelt, der als 16-Bit Wert einen Shift des Fensterwertes angibt, d.h. bei Sender und Empfänger wird das Fenster als 32 Bit Wert betrachtet und der 16 Bit Eintrag im Header wird mittels des "scale factor" entsprechend nach links bzw. rechts verschoben (Multiplikation des Fensterwertes mit einer Zweierpotenz). Dabei darf der Faktor nur beim Verbindungsaufbau übermittelt werden und gilt dann für die ganze Verbindung [JBB92].

#### 3.5.11.5 Exaktere Bestimmung der Paketlaufzeiten ("RTTM - round-trip time measurement")

In Kap. 3.5.8 wurde u.a. behandelt, wie die Timeouts zur Wiederholung des Versands von Paketen bestimmt werden, wobei dieser Berechnung eine möglichst genaue Bestimmung der Paketlaufzeiten zu Grunde liegt. Die meisten Implementierungen machen dazu eine Messung pro Sendefenster. Bei kleinen Fenster genügt dies auch, werden allerdings große Fenster wie bei modernen Übertragungswegen notwendig, verwendet, so ist damit keine exakte Messung mehr möglich. Dies ist auch theoretisch erklärbar, da die Paketversenderate im Prinzip mit der Fensterrate abgetastet wird, wobei das Abtasttheorem verletzt wird, da die Abtastrate viel zu gering ist. Wiederum wurde zur Lösung dieses Problems eine neue Option definiert, die sogenannte "Timestamp" Option. Die Option besteht im wesentlichen aus einem Zeiteintrag und der entsprechenden Antwort darauf. Der sendende Rechner trägt einen 32 Bit Zeitstempel in das erste Feld ein und der Empfänger schickt diesen Zeitstempel dann im zweiten Feld zurück. Das Verfahren ist symmetrisch, d.h. Zeitstempel können jederzeit in beide Richtungen versendet werden. Bei Erhalt eines Segments mit Zeitstempel kann der Empfänger durch leichte Differenzbildung mit der aktuellen Zeit die exakte Laufzeit bestimmen. Wiederum wird die Anwendung dieser Methode beim Verbindungsaufbau ausgehandelt. Für die Zeitstempel wird eine "virtuelle Uhr" verwendet, deren Wert monoton steigt und deren Frequenz innerhalb weiter Grenzen liegen darf. RFC 1323 [JBB92] empfiehlt eine Inkrementierung im Bereich einer Millisekunde bis zu einer Sekunde. Man beachte, dass die Uhren für diesen Algorithmus nicht synchronisiert sein müssen.

#### 3.5.11.6 Methode gegen den zu schnellen Verbrauch des Sequenznummernraums ("PAWS")

In Kap. 3.5.7 wurde das Problem erläutert, dass Sequenznummern bei sehr schnellen Übertragungswegen schneller verbraucht werden, als die maximale Lebenszeit eines Segments ist und es dadurch passieren kann, dass ein altes, verzögertes Paket fälschlicherweise akzeptiert wird. Mittels PAWS ("protection against wrapped sequence numbers") wird dies verhindert. Dazu wird ebenso die in Kap. 3.5.11.5 dargestellte Timestamp Option verwendet. Dabei ist die Idee sehr einfach: ein Paket kann als altes, verspätetes Paket verworfen werden, wenn sein Zeitstempel älter ist als ein Zeitstempel, der bis jetzt über die Verbindung empfangen wurde. Dazu muss der Zeitstempel monoton ansteigen und sich mindestens um eins pro Fenster erhöhen (normalerweise ist die Uhr sicher schneller als ein Fenster). Dabei wurde die Methode derart gestaltet, dass auf die gleichen Datenstrukturen, die schon bei dem Verfahren aus Kap. 3.5.11.5 benötigt werden, zurückgegriffen werden kann [JBB92].

### 3.5.12 Zusammenfassung

In diesem Kapitel wurde das TCP-Protokoll sehr ausführlich und umfassend behandelt. Ausgehend von den Grundlagen wie der Sequenzierung der Daten und dem Versand in Segmenten wurden die Konzepte zur Sicherung des Datentransports besprochen, u.a. das positive Bestätigungsschema kombiniert mit Timeouts sowie die Prüfsumme zur Erkennung von Übertragungsfehlern. Der Ablauf des Verbindungsaufbaus als auch des Verbindungsabbaus wurden an Beispielen erläutert. Die einzelnen Einträge im Header eines TCP-Segments wurden besprochen und aufbauend darauf wurde auf die speziellen Eigenschaften eingegangen, u.a. das Urgent Flag, das Push Flag sowie das Verfahren zur Flußkontrolle. Es wurde das Problem des "Silly window syndrome" sowie Ansätze zur Lösung wie der Algorithmus von Nagle oder das verzögerte Weitergeben von Fenstergrößeänderungen behandelt. Weitere Fragestellungen von TCP wie die Wahl der initialen Sequenznummern, Ansätze zur erweiterten Flußkontrolle, um Überlastungen des Netzwerks zu vermeiden und die Verwaltung der Timer (Timeout-Berechnung und Ergänzung von Karn) wurden eingehend betrachtet. Darauf folgend ist die Verarbeitung und der Versand von Reset-Segmenten dargestellt worden und ein Überblick über die Zustandsmaschine, die TCP konzeptionell zu Grunde liegt, gegeben worden. Neuere Erweiterungen wie "path MTU discovery", schnellere Wiederholung von verloren gegangenen Paketen sowie Optionen zur Erweiterung des Fensters, zur selektiven Bestätigung, exakteren Laufzeitmessung und zum Umgang mit den Sequenznummern bei Systemen mit sehr hohen Übertragungsraten wurden im letzten Abschnitt kurz vorgestellt und zeigen auf, dass TCP seit seinem Bestehen an verschiedenste Systeme angepasst wurde und auch nach der Protokollspezifikation vor 20 Jahren weiter an Verbesserungen und Erweiterungen gearbeitet wird.

# 4 Die Implementierung des Netzwerkkartentreibers

## 4.1 Überblick

Der Treiber zur Ansteuerung der Netzwerkkarte ist sehr umfangreich. Dies liegt unter anderem daran, dass der Treiber nicht nur mit der in Kap. 2 beschriebenen Netzwerkkarte funktioniert, sondern allgemein die ganze Serie von Netzwerkkarten des Typs 3C90x und 3C90xB betreiben kann und somit auch Methoden anbieten muss, um die verschiedenen Fähigkeiten bestimmter Karten zu erkennen und zu unterstützen. Besonders komplex ist beispielsweise die Detektion des aktiven Anschlusses. Während das Vorgehen bei einem Twisted-Pair Anschluß noch relativ einfach ist, ist die Vorgehensweise bei Glasfaseranschluss wieder völlig anders, was sich natürlich in der Komplexität der Methoden niederschlägt. Die Implementierung orientiert sich an einem Treiber für das Linux-Betriebssystem [3CO00].

Der Treiber besteht aus mehreren Hauptklassen, welche im Prinzip den Zugriff auf eine spezielle Funktionseinheit kapseln (beispielsweise Lese- und Schreibzugriff auf das EEPROM) sowie vielen weiteren Exception-Klassen, Informationsklassen zum Speichern von Konfigurationsinformationen und Hilfs-Klassen, deren Methoden in mehreren anderen Klassen benötigt werden (beispielsweise zum Setzen von Bits). Die beiden zentralen Klassen sind `ComInit`, welche die Karte(n) im Rechner detektiert und dann einige Grundkonfigurationen vornimmt sowie die Klasse `D3C905`, die den eigentlichen Treiber darstellt. Im folgenden wird ein Überblick über die wichtigsten Klassen gegeben, bevor darauf aufbauend einige Klassen genauer betrachtet werden:

- `ComInit`: diese Klasse durchsucht den PCI-Bus nach unterstützten Netzwerkkarten und instanziiert für jede Karte ein Treiber-Objekt. Danach werden für jede Karte einige grundlegende Dinge wie z.B. das Busmastering aktiviert und der verwendete IRQ sowie die Basisaddresse für den Registerzugriff bestimmt.

- `D3C905`: dies ist der eigentliche Treiber, der, unterstützt von den vielen anderen Klassen, Methoden zur Initialisierung der Karte, zum Senden und zum Empfangen von Daten zur Verfügung stellt. Die Interruptverwaltung wird von dieser Klasse durchgeführt, wobei dies neben der Benachrichtung über den Empfang von Daten auch die Behandlung von Fehlersituationen beinhaltet.

- `Befehl`: in dieser Klasse sind alle Befehlskombinationen definiert und es werden Methoden angeboten, um auf Register zuzugreifen und Befehle an die Karte abzusetzen. Außerdem gibt es Methoden für die Interruptverwaltung.

- `Register`: Diese Methode kapselt alle Adressen für die verschiedenen Register und Registerfenster, wobei über symbolische Namen darauf zugegriffen werden kann. Methoden werden von dieser Klasse nicht zur Verfügung gestellt.

- `Eeprom`: der Zugriff auf das Eeprom wird von dieser Klasse erledigt. Es befinden sich alle symbolischen Namen für den Eeprom-Zugriff in dieser Klasse und es werden Methoden zur Verfügung gestellt, um lesend oder schreibend auf das Eeprom zugreifen zu können.
- `Mii`: der Zugriff auf das Medium Independent Interface erfolgt über diese Klasse. Es sind wiederum alle zum Zugriff auf das MII-konforme Gerät benötigten Adressen als symbolische Namen verfügbar und es werden Methoden zum Lesen und Schreiben von MII-Registern angeboten.
- `DpdListEntry`: diese Klasse stellt einen Download-Deskriptor dar. Die angebotenen Methoden erleichtern die Verwaltung der Deskriptoren und bieten auch Möglichkeiten zum Debugging.
- `UpdListEntry`: wie `DpdListEntry`, nur eben ein Upload-Deskriptor.
- `NicInformation`: diese Klasse enthält mehrere weitere Informationsklassen und ist die am meisten benutzte Klasse zum Speichern von Einstellungen und Konfigurationswerten der Netzwerkkarte.
- `WaitTimer`, `NicTimer`: verschiedene Klassen, die das Timer-Interface implementieren. Das Timer-Konzept wird in Kap. 4.7 näher erläutert.
- `NicPciInformation`, `NicHardwareInformation`, `CapabilitiesWord`, und weitere: dies sind Klassen, die Informationen über spezielle Eigenschaften speichern, so speichert beispielsweise `NicPciInformation` die I/O-Basisadresse und den Interrupt und in `NicHardwareInformation` werden Informationen über die verfügbaren Anschlüsse oder eventuell behobene Hardwarefehler abgespeichert.

Schließlich gibt es schon wie oben erwähnt noch einige Exceptionklassen wie z.B. `D3C905Exception` oder `LinkSpeedException`.

## 4.2 Detektion der Karten und grundlegende Initialisierung - Klasse `ComInit`

Wie bereits oben erwähnt, dient diese Klasse dazu, den PCI-Bus des Rechners nach unterstützten Netzwerkkarten abzusuchen und für diese einen Treiber zu instanziieren. Der Start des Netzwerksystems gestaltet sich daher relativ einfach. Es muss nur ein Objekt dieser Klasse erzeugt werden und dessen Methode *probe()* aufgerufen werden. Diese Methode benötigt als Argument eine Referenz auf den PCI-Bus, der abgesucht werden soll. Mittels der Methode *getDevicesByClass()* der PCI Klasse wird der Bus nach Geräten der Klasse PCI_BASE_CLASS_NETWORK abgesucht. Von den gefundenen Netzwerkkarten wird geprüft, ob der Hersteller 3COM ist, d.h. es wird nach der entsprechenden Vendor-ID gesucht. Daraufhin wird anhand der Device-ID des/der gefundenen Geräte(s) geprüft, ob es ein vom Treiber unterstützes Modell ist.

Daraufhin werden alle den oben genannten Kriterien entsprechenden Karten abgespeichert (`DeviceInfo` Objekt) und für jede Karte ein Treiber-Objekt der Klasse `D3C905` instanziiert. Abschließend wird für jede Karte die Methode *doBasicConfig()* aufgerufen. Diese Methode überprüft das Busmastering und aktiviert es, falls es nicht automatisch durch das Bios aktiviert worden ist. Daraufhin werden die der Karte vom Bios zugewiesene I/O-Basisaddresse und die Speicher-Basisaddresse für den Registerzugriff ausgelesen sowie der Interrupt der Karte bestimmt. Abschließend werden einige weitere Konfigurationsdaten ausgelesen bzw. gesetzt, unter anderem die PCI-Latenz, die Hardware-Revision der Karte und die Unterstützung von Flußkontrolle. Die entsprechenden Werte werden in einem Objekt der Klasse `NicInformation` gespeichert.

Nach Ablauf dieser Methode ist die erste Stufe der Initialisierung beendet. Über Methoden der Klasse `ComInit` kann bestimmt werden, wieviele Karten gefunden wurden und über die Referenz auf den Treiber (Objekt der Klasse `D3C905`) kann nun auf die Karte zugegriffen werden. Um Daten zu senden bzw. empfangen zu können, muss allerdings die Karte erst vollständig konfiguriert werden, was eine Methode der Treiberklasse übernimmt.

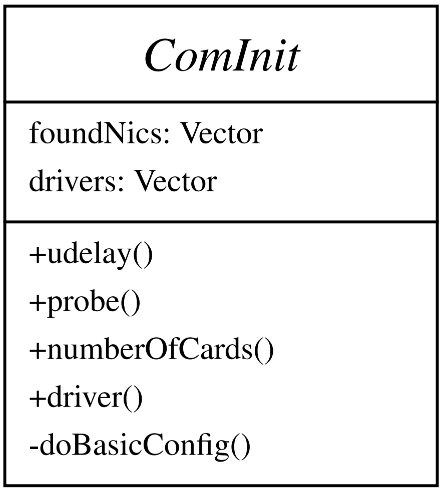


**Abb. 4.1 Die Klasse `ComInit`**

Abb. 4.1 zeigt nochmals die für die Anwendung dieser Klasse wichtigen Methoden. *udelay()* wird von Methoden des Treibers benötigt, um eine bestimmte Zeit zu warten, bevor beispielsweise ein Registerinhalt gültig ist. Diese Methode ist derzeit auf die Geschwindigkeit des verwendeten Entwicklungsrechners abgestimmt und müßte für andere Systeme noch angepasst werden. Mittels *numberOfCards()* kann bestimmt werden, wieviele Karten gefunden und konfiguriert wurden um dann mit *driver()* eine Referenz auf den der jeweiligen Karte zugeordneten Treiber zu erhalten.

## 4.3 Zugriff auf das Eeprom - Klasse `Eeprom`

Wie schon in Kap. 2.3 beschrieben, erfolgt der Zugriff auf die Inhalte über ein spezielles Eeprom-Kommando Register und ein Eeprom-Datenregister. Die Klasse `Eeprom` bietet zwei Methoden für den lesenden sowie schreibenden Zugriff auf die Eeprom-Register an. Diese Methoden schalten hierzu auf das entsprechende Registerfenster um, um dann nach der Prüfung eines speziellen Bits (EEPROM_BUSY_BIT) mittels Lese- und Schreibbefehlen der `Befehl`-Klasse auf die beiden Register zuzugreifen. Dabei wird geprüft, ob diese Operation innerhalb eines Timeouts beendet wird (private Methode *CheckIfEEPROMBusy()*). Bei schreibenden Zugriff muss vorher noch der alte Inhalt des entsprechenden Registers gelöscht werden.

## 4.4 Steuerung der Karte über Befehle - Klasse `Befehl`

In Kap. 2.8 wurde das Befehls-Konzept der Netzwerkkarte erläutert. Die Klasse Befehl enthält alle “Bitmuster” der Befehle und bietet mittels symbolischer Namen einen Zugriff darauf. Die Klasse bietet Befehle für den Lese-/Schreibzugriff auf die Register der Karte, wobei Byte-, Wort- sowie Langwortzugriffe möglich sind. Der Zugriff selbst erfolgt mit Hilfe der schon zur Verfügung stehenden `Port`-Klasse unter Berücksichtigung der Basisadresse über den I/O-Adreßraum.

Natürlich sind in dieser Klasse Methoden für das Absetzen von Kommandos an die Karte enthalten, wobei hier zwei Versionen existieren. Eine normale Version, die den Befehl sofort absetzt und eine Version, die innerhalb eines Timeouts auf die Abarbeitung des Befehls wartet, was bei Befehlen nötig ist, die länger dauern als die Zeit, die das System für einen weiteren Zugriff benötigt (siehe auch Kap. 2.8). Zusätzlich enthält diese Klasse noch Methoden für die Interruptverwaltung: es können mittels eigener Methoden sowohl die Interrupts als auch die sogenannten “Indications” aktiviert bzw. deaktiviert werden und alle Interrupts bestätigt werden.

## 4.5 Zugriff auf das Medium Independent Interface - Klasse `Mii`

Das Schreiben und Lesen der Register des Medium Independent Interface erfolgt über diese Klasse. Da diese technisch einfache Schnittstelle allerdings nur aus 2 Leitungen besteht, ist die Programmierung relativ aufwendig, was sich unter anderem daran zeigt, dass jeweils zu Beginn eines Transfers ein Synchronisationszyklus durchlaufen werden muss und auch bestimmte Timing-Anforderungen eingehalten werden müssen. Natürlich ist der Zugriff über dieses Interface gegenüber gewöhnlichen Registern nicht sehr performant, da ein Register Bit für Bit seriell übertragen werden muss.

Mit Hilfe dieser Klasse ist es nicht nur möglich, auf einfache Weise auf das MII zuzugreifen und somit Register zu lesen oder zu schreiben, sondern es werden auch Methoden zum vereinfachten Umgang mit der automatischen Aushandlung der Übertragungsgeschwindigkeit (Autonegotiation) zur Verfügung gestellt, da auf die Register, welche den Autonegotiation Mechanismus kontrollieren, über dieses serielle Interface zugegriffen werden muss. Die Methode *CheckMI-*

*lAutoNegotiationStatus()* überprüft anhand des *AutoNegStatus* Registers, ob eine Verbindung besteht (sogenanntes “Link up”). Falls dies nicht der Fall ist, wird anhand spezieller Register versucht zu bestimmen, was die Ursache dafür ist. Die Methode *CheckMIIConfiguration()* überprüft, ob die ausgehandelten Übertragungsparameter gültig sind. Dazu wird der aktuelle Verbindungsstatus bestimmt und mit den Daten des *MediaOptions* Register verglichen, welches die auf der Karte verfügbaren Anschlüsse beschreibt (z.B. 10BASE-T Anschluss). Falls bei diesem Test ein Fehler auftritt, so wird die Karte entsprechend neu konfiguriert und der Autonegotiation-Prozess erneut angestossen.

*FindMIIPhy()* dient dazu, ein PHY-Device zu finden und zu prüfen, ob daran etwas angeschlossen ist und *ProgrammMII()* dient dazu, Register entsprechend der Konfiguration aus dem Eeprom zu setzen und beispielsweise Vollduplex-Übertragung einzustellen.

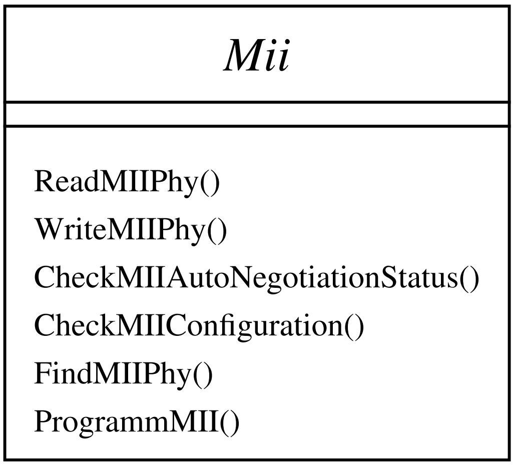


**Abb. 4.2 Überblick über Klasse `Mii`**

In Abb. 4.2 ist sind die wichtigen Methoden von Mii nochmals dargestellt. Intern verwendet diese Klasse wiederum die Methoden von `Befehl`, um auf das *PhysicalMgmt* Register zuzugreifen, welches die “Registerschnittstelle” zum seriellen Interface darstellt. Die oben beschriebenen Methoden werden wiederum von der Treiberklasse `D3C905` benötigt, um die Karte zu initialisieren, da dazu unter anderem die Autonegotiation Register benötigt werden oder geprüft werden muss, welcher Anschluss benutzt wird.

## 4.6 Die Download- und Uploaddeskriptoren - `DpdListEntry` und `UpdListEntry`

Ein zentraler Punkt in der Ansteuerung der Karte ist die Verwaltung der Deskriptoren für die Speicherfragmente, die zu versendende Daten beinhalten oder freien Speicher für den Empfang von Daten aus dem Netzwerk beschreiben. Aus Gründen der Abwärtskompatibilität wird der Downloaddeskriptortyp 0 verwendet. Außerdem wird die zusätzliche Möglichkeit zum zeitgesteuerten Download von Daten der Deskriptoren Typ 1 nicht benötigt.

Die beiden Klassen `DpdListEntry` sowie `UpdListEntry` dienen zum Zugriff und zur Verkettung dieser Deskriptoren. Sie sind strukturell gleich aufgebaut, unterscheiden sich natürlich jedoch in den jeweils spezifischen Einträgen eines Downloaddeskriptors bzw. eines Uploaddeskriptors. Die eigentlichen Deskriptoren werden durch Objekte der bereits zur Verfügung stehenden `Memory`-Klasse dargestellt, welche einen direkten Zugriff einen Hauptspeicherbereich des Rechners ermöglichen. Ein Objekt der Klasse `DpdListEntry` bzw. `UpdListEntry` enthält somit zwei `Memory`-Objekte. Den eigentlichen Deskriptor sowie ein `Memory`-Objekt des Speichers, welcher Daten enthält bzw. aufnehmen soll. Darüber hinaus wird die Verkettung gespeichert sowie einige weitere Daten, auf die häufiger zugegriffen wird. Es werden Methoden angeboten, um die Objekte miteinander zu verketten, bestimmte Felder des Deskriptors wie z.B. den *UpPacketStatus* auszulesen oder das Speicherobjekt zu entnehmen bzw. neuen Speicher einzufügen. Je nach der Art eines Zugriffs kann die Antwort anhand der gespeicherten Werte erfolgen (z.b. die *PacketLength* bei einem Download-Deskriptor, da dieser Werte nach "Einhängen" eines `Memory`-Objektes bis zur Änderung des zugrunde liegenden `Memory`-Objekts gleich bleibt) oder muss zum `Memory`-Objekt, welches den Deskriptor darstellt, weitergereicht werden. Anhand einer *toString()* Methode können die Deskriptoren mit allen Einträgen ausgegeben werden, um im Debugging-Fall einen Überblick zu bekommen.

In Abb. 4.5 sind die Download- und die Uploaddeskriptoren im Zusammenhang mit den anderen Klassen dargestellt, wobei die Deskriptoren auf die wesentlichen Inhalte reduziert wurden.

## 4.7 Das Timer-Konzept - Erweiterung der `Clock`-Klasse

Bei der Entwicklung des Treibers wurde häufig die Möglichkeit benötigt, asynchron nach Ablauf einer bestimmten Zeitspanne bestimmte Register zu überprüfen oder aber in regelmäßigen Abständen bestimmte Werte abzuspeichern bzw. aufzufrischen (beispielsweise Statistiken). Allgemein bedeutet dies, dass ein System nötig ist, mit dem zu bestimmten Zeitpunkten eine beliebige Methode aufgerufen wird.

Da im metaXa- bzw. JX-System bislang keine Timer implementiert waren, wurde das System um ein einfaches Timer-Konzept erweitert. Neben der Definition eines Interfaces, das Objekte, die zeitgesteuert eine Methode aufrufen möchten, implementieren müssen, wurde die `Clock`-Klasse erweitert, welche Methoden für den Zugriff auf den Timer-Interrupt zur Verfügung stellt.

In der `Clock`-Klasse wird alle 10 ms (Intervallzeit des Timer-Interrupts) durch den Interrupthandler dieser Klasse ein Zähler (sogenannter (Clock-)Tick) erhöht, welcher als eine “Uhr” mit einer Granularität von 10ms verwendet werden kann.

### 4.7.1 Implementierung der Timer

Die Implementierung der Timer besteht aus mehreren Komponenten:

- dem Interface `Timer`, welches jedes Objekt, das die Timer-Funktionalität benötigt, implementieren muss (siehe Abb. 4.3). Dieses enthält nur zwei Methoden. Die Methode *timer()* wird nach Ablauf des Zeitintervalls aufgerufen. Dabei kann ein beliebiges Argument übergeben werden, dass von der entsprechenden Timer-Methode auf den erwarteten Argument-Typ umgewandelt (“cast”) werden muss. Die Methode *equals()* wird benötigt, um feststellen zu können, ob ein Objekt eine Instanz einer bestimmten Klasse ist, welche dieses Interface implementiert. Dies wird unter anderem dann benötigt, wenn man noch nicht abgelaufene Timer-Objekte aus der Warteschlange löschen möchte.

```
public Interface Timer {
    public void  timer(Object arg);
    public boolean equals(Object obj);
}
```

**Abb. 4.3 Das `Timer`-Interface**

- aus Objekten der Klasse `TimerEntry`, die in einer verketteten Liste von der `Clock`-Klasse verwaltet werden. Diese Klasse enthält neben der Verkettung ein das `Timer`-Interface implementierendes Objekt sowie ein Argumentobjekt, das beim Aufruf der *timer()*-Methode übergeben wird. Darüber hinaus ist die “expire”-Zeit eingetragen, die angibt, zu welchem Zeitpunkt die *timer()*-Methode aufgerufen werden soll. Dabei ist dies eine absolute Zeit angegeben in Ticks der `Clock`-Klasse, d.h. derzeit mit einer Auflösung von 10ms. Möchte man beispielsweise, dass die Methode in einer Sekunde ausgeführt wird, so wird man dort die (aktuelle Zeit in Ticks + 100) eintragen. Der eigentliche Aufruf der jeweiligen *timer()*-Methode wird über die Methode *doCall()* eines `TimerEntry` ausgeführt. In Abb. 4.4 ist ein `TimerEntry` reduziert auf die wesentlichen Komponenten dargestellt.

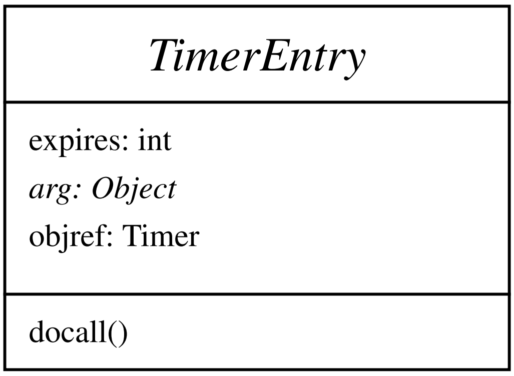


**Abb. 4.4 Timer-Entry**

- den neuen Methoden der `Clock`-Klasse, mit denen `TimerEntry` Objekte in die Liste der `Clock`-Klasse eingefügt werden können. Dabei wird ein neuer Eintrag sortiert nach der "expire"-Zeit eingefügt, d.h. am Anfang der Liste steht immer derjenige Eintrag, dessen "expire"-Zeit als nächstes abläuft. Im Interrupt-Handler der `Clock`-Klasse wird nun zusätzlich zur Inkrementierung des Tick-Wertes die Liste geprüft nach Timern, die ablaufen. Werden entsprechende gefunden, wird ihre *timer()*-Methode aufgerufen. Danach wird dieser Eintrag aus der Liste entfernt, d.h. wenn man einen Timer benötigt, der in bestimmten Zeitabständen aufgerufen wird, so muss dieser Timer sich selbst mit einer neuen "expire"-Zeit eintragen.

### 4.7.2 Anwendung der Timer - Klassen `NicTimer` und `WaitTimer`

Mittels dieser beiden Timer erledigt der Treiber Aufgaben, die zyklisch wiederholt werden müssen bzw. überprüft asynchron benötigte Registerinhalte. `NicTimer` wird am Ende der Initialisierung von *NICOpen()* erzeugt und gestartet (siehe Kap. 4.8.1 weiter unten). Dieser Timer wird jede Sekunde aufgerufen (indem er sich selbst wieder in die Timer-Liste einfügt) und liest dabei die Statistikregister der Netzwerkkarte aus, wodurch sie wieder zurückgesetzt werden (mittels *UpdateStatisticsEvent()* - siehe Kap. 4.8.3 weiter unten). Zusätzlich wird alle fünf Sekunden der Zustand der Verbindung geprüft und gegebenenfalls angepasst (beispielsweise Änderung des Übertragungsgeschwindigkeit von 10MBit auf 100MBit).

Objekte der Klasse `WaitTimer` werden an mehreren Stellen des Treibers benötigt, beispielsweise in *TestAdapter()* bei der Initialisierung des Treibers (siehe Kap. 4.8.1 weiter unten). Diese Timer testen einen Registerwert auf eine Veränderung und tauschen diesen Wert über eine Informationsklasse (*WaitTimerGlobal*) mit dem Treiber aus. Beispielsweise wird im Fall der Methode *TestAdapter()* anhand des *DownListPtr* Registers überprüft, ob die Karte das Testpaket angenommen hat. Ändert sich der Wert dieses Registers nicht innerhalb einer bestimmten Zeitspanne, so liegt bei der Download-Funktionseinheit ein Fehler vor.

## 4.8 Die Treiberklasse - `D3C905`

In der Klasse `D3C905` ist die eigentliche Funktionalität des Treibers enthalten. Die Aufgaben dieser Klasse umfassen die Konfiguration der Netzwerkkarte und den Start dieser, natürlich das Senden und Empfangen von Daten und somit auch die Interruptverwaltung und Interruptbehandlung sowie der Umgang mit möglicherweise im Betrieb auftretenden Fehlern, wobei es auch Methoden gibt, um etwa eine einzelne Funktionseinheit zurückzusetzen oder die komplette Karte wieder in den Startzustand zu versetzen. Diese Klasse verwaltet die Liste der Download- und Uploaddeskriptoren und enthält eine Queue, mit der empfangene Pakete an den Netzwerkstack weitergereicht werden. Um die vielfältigen Aufgaben zu erledigen, bedient sich die Klasse der Methoden der oben beschriebenen Hilfsklassen, um etwa auf die MII-Schnittstelle oder auf das Eeprom zugreifen zu können.

Für einen Überblick über die Klasse D3C905 im Zusammenspiel mit allen Hilfsklassen siehe Abb. 4.5.

### 4.8.1 Konfiguration und Start der Karte

Bevor die Karte Daten empfangen bzw. versenden kann, muss sie mittels der Methode *NIC-Open()* in den Betriebszustand versetzt werden. Diese Methode ruft intern eine Reihe von Hilfsmethoden auf, die die Eigenschaften der Karte bestimmen, initialisieren und die korrekte Funktionsweise testen. Im folgenden wird ein Überblick über den Ablauf gegeben:

(1) *GetAdapterProperties()*: hier werden die grundsätzlichen Eigenschaften der Karte bestimmt. Es wird überprüft, welches Modell vorhanden ist und anhand dieser Information werden einige Konfigurationsparameter gesetzt (z.B. Anzahl der Hashfilter-Bits). Durch einen Zugriff auf ein Registerfenster wird geprüft, ob ein Zugriff auf die Register fehlerfrei funktioniert. Danach werden Eigenschaften wie Vollduplex-Übertragung, Powermanagement oder das Vorhandensein bekannter Hardwarebugs gepüft und entsprechende Informationen in Hilfsklassen abgespeichert. Am Ende wird die Hardware-Adresse der Karte aus dem Eeprom gelesen und für die spätere Konfiguration gespeichert.

(2) *BasicInitializeAdapter()*: es werden alle Funktionseinheiten für die weitere Konfiguration deaktiviert (da dies u.a. auch für das Testen des Adapters im nächsten Schritt notwendig ist) sowie alle Statistikregister durch Lesen ihrer Werte auf 0 zurückgesetzt. Darüberhinaus wird die in *GetAdapterProperties()* bestimmte Ethernet-Addresse in das entsprechende Adressregister geschrieben.

(3) *TestAdapter()*: mittels dieser Methode wird die Funktionsfähigkeit der Karte getestet. Dazu wird die Karte in den sogenannte FIFO-Loopback Modus geschaltet, in dem der Sendepuffer direkt mit dem Empfangspuffer verbunden ist. Nun kann der normale Betrieb simuliert werden, indem ein Testpaket erzeugt wird und mittels der Deskriptoren versendet wird. Dabei wird einerseits überprüft, ob die Hardware das Paket entgegen nimmt und andererseits, falls dies erfolgreich war, ob das Paket empfangen wird und sein Inhalt unverfälscht ist. Zum Überprüfen, ob das Paket von der Hardware entgegen genommen wurde, wird ein Timer (`WaitTimer`) verwendet, der innerhalb eines Timeouts das *DnListPtr* Register überprüft.

(4) *SetupMedia()*: nun muss der Anschluß und dessen Geschwindigkeit bestimmt werden (z.B. 10BASE-T) und entsprechende Konfigurationsregister gesetzt werden. Hierbei ist das in Kap. 2.10 beschriebene *xcvrSelect* Register von Bedeutung. Je nachdem, ob die sogenannte AutoSelection konfiguriert ist (spezielles Bit im *InternalConfig* Register) wird mittels der Hilfsmethode *MainAutoSelectionRoutine()* ein automatisches Erkennen des aktiven Anschlusses versucht. Das grobe Vorgehen hierzu ist in Kap. 2.10 beschrieben. Anhand des *MediaOptions* Register kann bestimmt werden, welche Anschlüsse geprüft werden müssen, wobei je nach Anschlussart dann ein anderes Vorgehen notwendig ist, um erkennen zu können, ob dieser aktiv ist und welche Parameter zu setzen sind. Falls jedoch Autoselection nicht aktiviert ist, wird die Karte über die “erzwungene” Konfiguration initialisiert.

(5) *SoftwareWork()*: in diese Methode sind einige kleinere Konfigurationsaufgaben ausgelagert. Es werden einige spezielle Eigenschaften der neueren Serie 3C90xB abgeprüft und abgespeichert.

(6) *StartAdapter()*: mit dieser Methode wird die Initialisierung schließlich beendet. Es wird für das *CountDown* Register die entsprechende Geschwindigkeit für die Dekrementierung dieses Registers abhängig von der Verbindungsgeschwindigkeit eingestellt. Ferner werden die Anfangsadressen der Deskriptorlisten in die entsprechenden Register eingetragen, die Funktionseinheiten wieder aktiviert und abschließend der Receive-Filter gesetzt, um die gewünschten Arten von Paketen zu empfangen (nur an die Karte gerichtete Pakete, Multicast-Pakete, ...).

Nach der erfolgreichen Initialisierung durch diese sechs Methoden ist die Karte betriebsbereit und es können Daten mittels der *transmit()* Methode gesendet werden. Der interne Ablauf beim Versand vom Daten wird im nächsten Kapitel erläutert.

### 4.8.2 Versand von Daten

Die Klasse bietet die Methode *transmit()* zum Versand von Daten an. Dabei wird vorrausgesetzt, dass die übergebenen Daten nicht die maximale Länge eines Ethernet-Pakets überschreiten, d.h. die eventuell notwendige Aufteilung in Pakete passender Größe muss auf einer höheren Ebene schon erledigt worden sein. Der interne Ablauf beim Versand gestaltet sich dank des flexiblen Deskriptorkonzepts relativ einfach: es wird in der Liste der Downloaddeskriptoren der nächste freie Deskriptor gesucht, die übergebenen Daten in ein `Memory`-Objekt kopiert und dieses Objekt wird in den Deskriptor eingetragen. Beim Versand wird das Konzept des “Download-Polling” (siehe hierzu auch Kap. 2.5.1) verwendet. Die Übertragung der Daten in den Sendepuffer wird durch das Schreiben der Addresse des neuen Deskriptors in das *DnNextPtr* Feld des vorherigen Deskriptors gestartet. Dadurch wird ein Registerzugriff eingespart und stattdessen nur ein Speicherschreibbefehl benötigt. Die Benachrichtigung über den erfolgten Transfer erfolgt “aktiv”, d.h. es wird anhand der zu übertragenden Bytes das *CountDown* Register gesetzt. Bei Ablauf dieses Zählers wird ein Interrupt ausgelöst, in dessen Behandlung geprüft wird, ob der Deskriptor zur erneuten Verwendung wieder bereit steht (siehe dazu auch Kap. 4.8.3).

### 4.8.3 Empfang von Daten - Interruptverwaltung

In Kap. 2.6 wurde das Interrupt-Konzept ausführlich vorgestellt. Es gibt mehrere Auslöser für einen Interrupt und es ist sehr gut konfigurierbar, über welche Ereignisse man mittels eines Interrupts informiert werden möchte. Dieses mehrstufige Konzept schlägt sich auch im Interrupt-Handler nieder. Es gibt einen zentralen Handler nach Interface-Definition (*interruptHandler()*), welcher überprüft, ob wirklich ein Interrupt aufgetreten ist, um sogenannte “spurious interrupts” zu vermeiden. Nach der Maskierung der Interrupts für die Netzwerkkarte wird die Art des Interrupts bestimmt und je nach Interrupt eine eigene Methode zur Behandlung dieses Interrupts aufgerufen. Dabei dürfen nur 5 der 8 (bzw. 9 mit dem meist nicht implementierten *linkEvent*) möglichen Interrupts auftreten. Die Interrupts *rxComplete*, *linkEvent* und *dnComplete* zeigen einen Fehler an, da *rxComplete* automatisch bestätigt wird (siehe Kap. 2.6), *linkEvent* nur bei einer speziellen ASIC-Version verfügbar ist und *dnComplete* maskiert wird, da die Implementierung stattdessen einen Interrupt durch das *CountDown* Register auslöst.

Wurden neue Daten empfangen, wird dies durch einen *upComplete* Interrupt angezeigt. Die Methode *UpCompleteEvent()* behandelt diesen Interrupt und nimmt die empfangenen Daten entgegen. Dabei wird die Liste der Uploaddeskriptoren durchlaufen und anhand des *upComplete* Bits im *upPacketStatus* Eintrag erkannt, welche Deskriptoren neue Daten addressieren (Bit ist gesetzt und zeigt einen beendeten Upload an). Nach einer Überprüfung, ob das Paket fehlerfrei empfangen wurde (ebenfalls anhand des *upPacketStatus*), werden die empfangenen Daten in die `EtherPacketsQueue` (siehe Kap. 4.9) eingestellt, aus der sie asynchron vom Stack entnommen werden können. Nachdem die Deskriptoreinträge wieder zurückgesetzt worden sind und ein neuer Speicherbereich für den nächsten Datenempfang allokiert worden ist, werden am Ende noch einige Statistikdaten gespeichert. Damit ist die Bearbeitung des Interrupts beendet und der Deskriptor wieder bereit für den erneuten Empfang.

Die Behandlung der übrigen Interrupts ist ähnlich einfach. *CountDownTimerEvent()* ist für die Behandlung eines selbstausgelösten Interrupts zuständig. Da das *CountDown* Register in Abhängigkeit der zu versendenden Daten gesetzt wurde, sollte unter normalen Umständen bei Auftreten dieses Interrupts der Download beendet sein. Zur Interruptbehandlung wird nun die Liste der Downloaddeskriptoren analog dem obigen Vorgehen durchlaufen und anhand des *dnComplete* Bits des *FrameStartHeader* Eintrages werden die entsprechenden DPDs zur erneuten Verwendung gelöscht, d.h. der Speicher der versendeten Daten wird freigegeben und die Headereinträge werden zurückgesetzt. Sind noch nicht alle Daten versendet worden, wird am Ende erneut das *CountDown* Register gesetzt.

*txComplete* Interrupts werden von *TxCompleteEvent()* behandelt. Bei diesem Interrupt müssen keine Deskriptoren bearbeitet werden, da dies bei den jeweiligen Down- bzw. Upload-Interrupts schon erledigt wird. In diesem Fall werden nur einige Statistikdaten gespeichert und zwar nur im Fehlerfall, da dieser Interrupt sowohl nach Beendigung des Datenversands als auch nach Auftreten eines Fehlers ausgelöst wird. Mögliche Fehler sind unter anderem die Überschreitung der maximalen Anzahl von Kollisionen oder ein Hardwarefehler der Sendeeinheit.

Der *hostError* Interrupt weist auf einen katastrophalen Fehler hin. In diesem Fall kann die behandelnde Methode *HostErrorEvent()* nur einen globalen Reset der Karte durchführen und sie erneut initialisieren.

Der *updateStats* Interrupt benachrichtigt über einen drohenden Überlauf der Statistikregister. in *UpdateStatisticsEvent()* müssen dann nur die Statistikregister ausgelesen werden (und bei Interesse gespeichert werden), was diese automatisch zurücksetzt.

### 4.8.4 Zusätzliche Methoden und weitere Konfigurationsmöglichkeiten

Neben den bereits vorgestellten Methoden enthält die Klasse D3C905 noch einige weitere Methoden, die nur für bestimmte Karten notwendig sind bzw. Methoden, mit denen einige spezielle Eigenschaften wie die Definition von zu empfangenden Multicast-Adressen über den Hashfilter eingestellt werden können (siehe auch Kap. 2.10). Im folgenden wird darüber ein kurzer Überblick gegeben.

Mittels *ReStartAdapter()* kann die Netzwerkkarte erneut initialisiert werden, beispielsweise falls ein Fehlerfall aufgetreten ist (siehe auch oben *hostError* Interrupt). Diese Methode benützt hierzu wieder dieselben Methoden, mit denen auch *NICOpen()* anfangs die Netzwerkkarte initialisiert.

Wird das Betriebssystem beendet oder soll die Netzwerkkarte deaktiviert werden, so sollte *NICClose()* aufgerufen werden. Dabei werden die Sende- und Empfangseinheit abgeschaltet, noch nicht versendete Daten übertragen, alle Deskriptoren gelöscht und die Speicherobjekte freigegeben und schließlich vom Treiber benutzte Timer, welche noch nicht abgelaufen sind oder zyklisch aktiviert werden, gelöscht.

Mittels *IndicateToOSLinkStateChange()* kann das Betriebssystem - falls von diesem unterstützt - über Veränderungen der physischen Verbindung informiert werden. Derzeit gibt diese Methode nur entsprechende Meldungen aus.

Die Methode *NICSetReceiveMode()* ermöglicht die Konfiguration, welche Art von Ethernet-Paketen empfangen werden sollen. Dazu kann über einen Vector von `MultiCast`-Objekten eine Menge von `Multicast`-Adressen übergeben werden, welche von der Karte empfangen werden sollen. Darüberhinaus sind auch die “gewöhnlichen” Empfangsmöglichkeiten wie an die Karte adressierte Pakete und Broadcasts über einen Eintrag der `MultiCast`-Objekte einstellbar.

## 4.9 Weitergabe empfangener Daten an den Stack - Klasse `EtherPaketsQueue`

Wird der Treiber durch einen Interrupt über den Empfang neuer Daten benachrichtigt, so wird eine Möglichkeit benötigt, diese Daten an den Netzwerkstack zur weiteren Verarbeitung weiterzugeben. Die einfachste Möglichkeit ist sicherlich, eine vom Stack angebotene Empfangsmethode aufzurufen, die anhand des Pakettyps (IP, ARP, ...) die weiteren Schritte bestimmt. Jedoch tritt dadurch das Problem auf, das die Verarbeitung des Pakets im Interruptkontext abläuft und somit die Karte für weitere Interrupts gesperrt ist und möglicherweise durch die Dauer der Verarbeitung Daten verloren gehen. Abgesehen davon sind der Treiber und der Netzwerkstack schon konzeptionell verschieden, so dass eine derartige Verbindung nicht existieren sollte. Ferner soll die Ablaufdauer des Interrupthandler so kurz wie möglich sein, da der nächste Interrupt erst behandelt werden kann, wenn die Behandlung des vorherigen abgeschlossen wurde.

Daher dient die Klasse `EtherPacketsQueue` als eine Art Vermittler zwischen Netzwerktreiber und Netzwerkstack. Der Treiber kann hierbei mittels der Methode *add()* ein neu empfangenes Ethernet-Paket einreihen. Der `EtherPacketsQueue` ist ein Thread zugeordnet (welcher sich anfangs bei dieser Klasse registrieren muss - Methode *register()*), der dadurch aufgeweckt wird und dieses Paket entgegen nehmen kann und nun die weitere Verarbeitung anstoßen kann. Hat der Thread das Paket verarbeitet oder abhängig von der Implementierung an einen weiteren Thread, welcher die Verarbeitung übernimmt, weitergereicht, so blockiert dieser, falls keine weiteren Pakete in der Queue stehen und kann durch einen Aufruf von *add()* erneut aktiviert werden.

## 4.10 Zusammenfassung

Dieses Kapitel gab einen Überblick über den Aufbau des Treibers. Dabei wurden nach einer Erläuterung des generellen Aufbaus die wichtigsten Klassen vorgestellt und näher betrachtet. Es wurden sowohl einzelne Methoden vorgestellt als auch der interne Ablauf näher erklärt. Bei der Klasse D3C905 wurden wichtige Schritte im Betrieb der Karte dargestellt, unter anderem die Initialisierung, das Senden und Empfangen und zusätzliche Konfigurationsmöglichkeiten. Grafische Darstellungen der Haupt-Klassen, die einen Überblick über die wichtigsten Methoden liefern, sowie eine Grafik (siehe Abb. 4.5) zur Darstellung des Zusammenhangs zwischen den jeweiligen Klassen runden diese Kapitel ab.

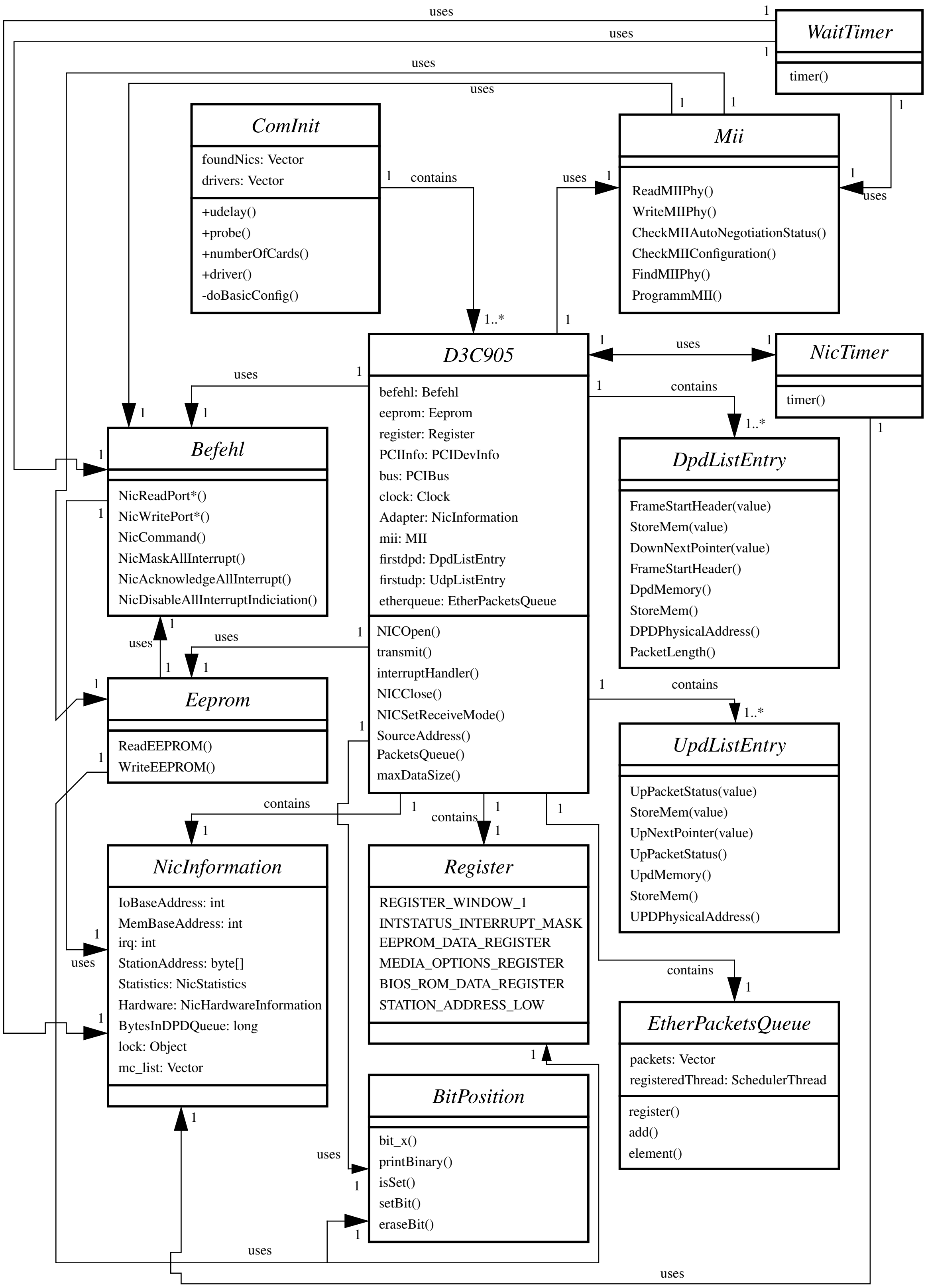


**Abb. 4.5 Zusammenhang zwischen D3C905 und Hilfsklassen**

# 5 Die Implementierung des Netzwerkstacks

## 5.1 Überblick

Der Netzwerkstack baut auf einigen Klassen auf, die die grundlegenden Schichten schon zur Verfügung gestellt haben. So konnten die Format-Klassen auch für den TCP-Teil benutzt werden, ferner war schon eine IP-Klasse vorhanden, die den Empfang bzw. den Versand von IP-Paketen ermöglichte. Die Implementierung fügt sich daher in das schon bestehende Konzept ein und erweitert teilweise auch die schon vorhandenen Klassen, wo dies notwendig ist.

Der Netzwerkstack folgt durchgehend dem Konzept der Formatklassen, die in Kap. 5.2 näher erläutert werden. Dies bedeutet, es gibt für jedes Protokoll eine Formatklasse (benannt nach dem Schema <Protokollname>Format.java, also z.B. IPFormat.java), welche einen einfachen Zugriff auf die Header-Einträge des jeweiligen Protokolls erlaubt. Zusätzlich wird jedes Protokoll im Prinzip durch eine Klasse dargestellt. Dazu existieren noch einige Hilfsklassen, welche allgemein benötigte Methoden bzw. Konstanten anbieten oder spezielle Dienste auslagern (z.B. TCP Retransmission). Auf Benutzerebene erfolgt der Zugriff über die vom Java Development Kit 1.1 (JDK1.1, siehe [JDK1]) bekannte Socket-Schnittstelle. Hierzu wurden mehrere Klassen implementiert, die bis auf eine kleine Einschränkung (nähere Erläuterungen siehe Kap. 5.8.1) konform zu dieser Schnittstelle sind und intern auf die Implementierung des Netzwerkstacks zugreifen.

Die Implementierung des eigentlichen TCP-Protokolls orientiert sich an [Tin86].

Im folgenden wird ein kurzer Überblick über die zentralen Klassen gegeben, bevor in den folgenden Kapiteln näher auf die wichtigsten Klassen eingegegangen wird und spezielle Abläufe beim Empfang bzw. Versand von Daten dargestellt werden.

- `Format`: die Basisklasse für die protokollspezifischen Format-Klassen. Diese Klasse bietet eine Vielzahl von Methoden zum Zugriff auf das interne Byte-Array und zur Manipulation des Inhaltes.
- `EtherFormat`, `ARPFormat`, `IPFormat`, `UDPFormat`, `TCPFormat`: spezielle Formatklassen für die jeweiligen Protokolle. Die Klassen bieten Zugriffsmethoden entsprechend den jeweiligen Header-Einträgen.
- `Ether`: die "Ethernet-Schicht" ermöglicht das Senden und Empfangen von Ethernet-Paketen.
- `ARP`: repräsentiert das Address Resolution Protocol. IP-Adressen können aufgelöst werden, ferner speichert ein Cache bereits bekannte Hardwareadressen und diese Klasse antwortet auf ARP-Anfragen, die nach der eigenen Hardwareadresse fragen.
- `IP`: diese Klasse entspricht einer rudimentären IP-Schicht, wobei Methoden zum Empfangen und Versenden von IP-Paketen angeboten werden.

- `UDP`: Klasse zum Versand und Empfang von UDP-Paketen. Dabei wird derzeit nicht das UDP-Interface des JDK unterstützt, sondern es ist nur ein sehr einfaches Konzept implementiert.
- `TCP`: diese Klasse implementiert das TCP-Protokoll und ist die größte Klasse. Der Benutzer kommt mit dieser Klasse normalerweise nicht direkt in Berührung, da seine Schnittstelle das Socket-Interface ist. Allerdings ist es möglich, direkt entsprechende Methoden von `TCP` aufzurufen und beispielsweise eine Verbindung zu initiieren.
- `Flusher`: ein Timer, der von der `TPC`-Klasse für den Versand von sehr kleinen Datensegmenten benötigt wird.
- `MyInputStream`, `MyOutputStream`: Streams zum Lesen bzw. Schreiben von Daten über die TCP-Verbindung. Bieten dasselbe Interface wie ein Input- bzw. Outputstream von JDK1.1.
- `TcpRetransmitSegment`, `Retransmit`: Hilfsklassen für die `TCP`-Klasse zur Implementierung des erneuten Versands von Segmenten bei Ausbleiben einer Bestätigung.
- `Socket`, `ServerSocket`: die Schnittstelle für den Benutzer, um die TCP-Implementierung zu benutzen. Das Interface entspricht prinzipiell dem Interface der korrespondierenden Klassen des JDK 1.1.

## 5.2 Datenkapselung über `Format`-Klassen

Eine grundlegende Frage bei dem Netzwerkstack ist die Frage nach dem Datenaustausch zwischen den einzelnen Schichten und die Frage nach der Repräsentation der Daten. Diese Implementierung verwendet dazu sogenannte Formatklassen. Die Basisklasse ist die abstrakte Klasse `Format`. Die Daten werden in einem Byte-Array in der Formatklasse gespeichert, wobei zusätzlich ein Offset verwaltet wird, d.h. eine Startposition für dieses Array. Die `Format`-Klasse bietet die grundlegenden Methoden, um auf das Array zuzugreifen. Hierzu gibt es eine Vielzahl von Methoden, um beispielsweise einen Integer, einen Short oder ein Byte zu schreiben oder zu lesen (z.B. *writeByte()*, *writeInt()*). Ferner gibt es auch die Möglichkeit, verschiedene Arten von Arrays zu schreiben (z.B. IntegerArray). Dabei kann bei allen Methoden ein Offset angegeben werden, zu dem noch transparent der interne Offset, welcher bei der Objekterzeugung übergeben wurde, addiert wird.

Auf dieser Klasse bauen die jeweiligen protokollspezifischen Formatklassen auf. Diese bieten nun entsprechend dem im jeweiligen Protokollkopf verwendeten Einträgen Methoden, um diese zu schreiben und zu lesen und darüberhinaus natürlich auch Zugriff auf die eigentlichen Daten zu erhalten. Darüber hinaus enthalten manche Formatklassen noch zusätzlich in diesem Protokoll benötigte Dienste, wie z.B. bei IP oder TCP die Berechnung einer Checksumme.

Um beim Datentransfer von Schicht zu Schicht ein Umkopieren der Daten und erneutes Allokieren eines größeren Arrays zu vermeiden, wird auf Ebene des TCP-Protokolls Speicher für die unteren Schichten mitreserviert. Die Methode *requiresSpace()* jeder Formatklasse gibt den benötigten Platz für den entsprechenden Protokollkopf an. Auf Ebene des TCP-Protokolls wird nun für alle noch nachfolgenden Protokollköpfe Speicher mitreserviert und der Protokollkopf des TCP-Protokolls wird ab einem entsprechenden Offset geschrieben. Damit bleibt noch Platz am Anfang des Arrays, in welchen die weiteren Protokolle ihren Header einfügen können. Mittels des anzugebenden Offset-Wertes bei der Konstruktion eines Format-Objektes kann immer der jeweils benötigte Startpunkt angegeben werden, um dann transparent auf die Informationen zuzugreifen bzw. sie schreiben zu können. Die entsprechenden Offsets für das jeweilige Protokoll sind wieder über die *requiresSpace()*-Methoden zu bestimmen. Im Prinzip sind die Format-Klassen nur eine Art "Maske", welche immer über dasselbe Array gelegt wird, jeweils mit einem anderen Offset, um transparent die jeweiligen Protokolleinträge zu lesen und schreiben.

## 5.3 Die Ethernetschicht - Klasse `Ether`

Diese Klasse bietet die Dienste zum Versenden und Empfangen von Ethernetpaketen an und ist über die `EtherPacketsQueue` (siehe Kap. 4.9) die Schnittstelle zwischen dem Netzwerkkartentreiber und dem Netzwerkstack. Die Klasse enthält die innere Klasse `Receiver`, welche die Datenentgegennahme aus der `EtherPacketsQueue` übernimmt. `Receiver` ist von `SchedulerThread` abgeleitet und benötigt ebenso wie die `Ether`-Klasse eine Referenz auf ein Netzwerkgerät (Klasse `NetworkDevice`), z.B. auf den Netzwerktreiber. An diesem registriert sich der Thread für die Entgegennahme von Daten und entnimmt in einer Endlosschleife die in die Queue gestellten Ethernetpakete. Falls die Queue leer ist, so ruft er *wait()* auf. Für empfangene und aus der Queue entnommene Ethernetpakete ruft der Thread die Methode *receive()* der Ether-Klasse auf.

Die *receive()*-Methode prüft, ob der Pakettyp ein IP-Paket oder ein ARP-Paket anzeigt. Falls keiner der beiden Typen zutrifft, wird das Paket fallengelassen und somit ignoriert. IP-Pakete werden an die statische *receive()*-Methode der IP-Klasse weitergereicht, ARP-Pakete an die *receive()*-Methode der ARP-Klasse, allerdings nur, falls sich die ARP-Schicht an der Ether-Klasse registriert hat. Sonst wird auch das ARP-Paket ignoriert.

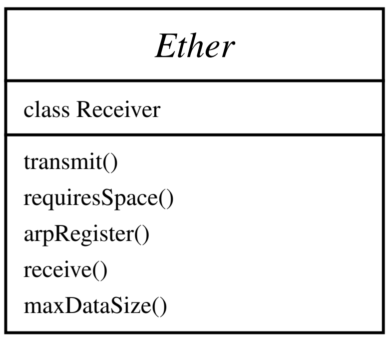


**Abb. 5.1 Die Ether-Klasse**

Zum Versand von Ethernetpaketen bietet die Klasse drei Varianten der *transmit()*-Methode. Die für den “normalen Gebrauch” bestimmte Methode nimmt als Parameter die Ziel-Hardwareadresse entgegen, die zu versendenden Daten und den Typ des Pakets, das versendet werden soll. Die eigene Hardwareadresse wird von dieser Methode automatisch eingefügt. Die zweite Variante erlaubt es jedoch, auch eine beliebige andere Hardwareadresse als Quelladresse anzugeben. Dies sollte allerdings im normalen Betrieb nicht verwendet werden, weil somit eine Störung des Netzwerks sehr wahrscheinlich ist und ein Paket die korrekte Absenderadresse haben sollte. Die dritte Variante von *transmit()* nimmt nur zu versendende Daten entgegen. Dies bedeutet, dass diese Daten schon ein korrektes Ethernetpaket enthalten müssen, ansonsten werden die ersten zwölf Bytes der Daten von den anderen Systemen als Quell- bzw. Zieladresse interpretiert.

Zusätzlich zu diesen Methoden gibt es noch einige Hilfsmethoden, u.a. zur Bestimmung der maximalen Größe eines Paketes, wobei diese Information vom zu Grunde liegenden Übertragungssystem abhängt und deshalb der Methodenaufruf an den jeweiligen Treiber weitergereicht wird.

Abb. 5.1 zeigt die wichtigsten Methoden dieser Klasse im Überblick.

## 5.4 Auflösung der IP-Adressen - Klasse `ARP`

Die Klasse `ARP` bietet die Dienste an, um zu IP-Adressen die entsprechenden Ethernet-Adressen zu finden und stellt im Prinzip die Grundlage dar, um IP-Pakete versenden zu können. Dazu bedient sich `ARP` der inneren Klasse `ARPCache`, welche bereits bekannte Zuordnungen von IP-Adresse zu Ethernet-Adresse abspeichert.

Um zu einer IP-Adresse die korrespondierende Ethernet-Adresse zu erhalten, bietet die Klasse die Methode *lookup()* an. Diese Methode benützt die Methode gleichen Namens des `ARPCache`. Dabei wird der Cache nach der abgefragten Adresse durchsucht. Ist die gesuchte Adresse bereits im Cache enthalten, so kann die Antwort sofort zurückgegeben werden. Ist die gesuchte Adresse noch nicht bekannt, wird mittels der von der `ARP`-Klasse angebotenen Methode *sendQuery()* eine ARP-Anfrage gestellt. Aus Gründen einer einfachen Implementierung wird nun innerhalb eines Timeouts der Cache durchsucht. Normalerweise sollte in einem lokalen Netz sehr schnell die gewünschte Antwort erhalten werden und somit auch im Cache auftauchen. Kann jedoch innerhalb des Timeouts die gesuchte Adresse nicht bestimmt werden, so wird mittels einer Exception dem Aufrufer von *lookup()* mitgeteilt, dass die gesuchte Adresse nicht bekannt ist.

Die *receive()*-Methode der Klasse `ARP` beantwortet Anfragen nach der eigenen Ethernet-Adresse. Diese Adresse sowie die eigene IP-Adresse können über entsprechende Methoden der `IP`-Klasse bzw. der `Ether`-Klasse bestimmt werden. Wird eine Anfrage empfangen, so wird ein Antwortpaket mit der eigenen Hardwareadresse erzeugt und versendet. Zusätzlich werden Antwortpakete auf ARP-Anfragen anderer Rechner empfangen und die enthaltene Information dem Cache mittels der *add()*-Methode hinzugefügt. Somit ist die Wahrscheinlichkeit groß, dass bei einer Anfrage an den Cache bereits die Information vorliegt, ohne dass selbst ein ARP-Paket versendet werden musste.

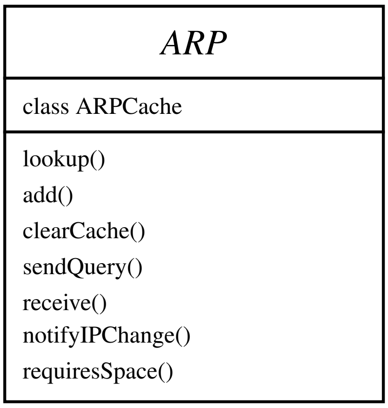


**Abb. 5.2 Die Klasse ARP**

Um Konfigurationsänderungen zu berücksichtigen, dürfen die Einträge im Cache keine beliebig lange Gültigkeit besitzen. Nach einer gewissen Zeit muss ein Eintrag entfernt werden, da es ja möglich ist, dass eine IP-Adresse einem anderen Rechner zugeordnet wird oder dieser Rechner eine andere Netzwerkkarte bekommt, wodurch sich seine Hardwareadresse ändert. Aus Gründen einer einfacheren Implementierung als auch aus Performanzgründen wird hierbei ein einfaches Konzept verfolgt. Die Klasse `ARP` bietet mit der Methode *clearCache()* die Möglichkeit, alle Einträge des Cache zu löschen. Diese Methode kann natürlich auch vom Benutzer dazu verwendet werden, nach einer Konfigurationsänderung an einem anderen Rechner explizit den Cache zu löschen und somit eine richtige Zuordnung sicherzustellen. Im Konstruktor der `ARP`-Klasse wird jedoch auch ein Timer gestartet, der alle 15 Minuten den `ARPCache` löscht. Der Nachteil, dass alle 15 Minuten eine höhere Last durch die erneute Anfrage nach Hardwareadressen durch dieses Vorgehen ausgelöst wird und dadurch, dass alle Adressen gelöscht werden, auch welche die erst für sehr kurze Zeit im Cache gehalten werden, wird wohl durch die höhere Geschwindigkeit dieses Vorgehens ausgeglichen. Sonst wäre es beispielsweise nötig, für jede neue Eingabe in den Cache gleichzeitig einen Timer zu erzeugen (2 Objekte - der `Timer` selbst und ein `TimerEntry`, eventuell auch noch Argumentobjekt) und zu starten, welche nach Ablauf explizit diesen Eintrag wieder aus dem Cache entfernt.

Neben den bereits erwähnten Methoden enthält `ARP` noch einige Hilfsmethoden, u.a. um beispielsweise über Konfigurationsänderungen der eigenen IP-Adresse informiert zu werden (*notifyIPChange()*) oder den Platzbedarf eines kompletten ARP-Pakets zu bestimmen (*requiresSpace()*).

In Abb. 5.2 ist die `ARP`-Klasse nochmals im Überblick dargestellt.

## 5.5 Die Internet-Protokoll Schicht - Klasse `IP`

Dies ist eine sehr einfache Klasse, die im Prinzip nur die Möglichkeit zum Versenden und zum Empfangen von `IP`-Paketen bietet. Weitergehende Möglichkeiten wie das Routing von IP-Paketen sind nicht implementiert. In `IP` ist die eigene IP-Adresse gespeichert, wobei es eine Möglichkeit gibt, diese zur Laufzeit zu ändern (*changeSourceAddress()* - dabei wird auch die `ARP`-Klasse benachrichtigt - siehe oben) und natürlich diese abzufragen (*getmyselfIP()*). Die Methode *receive()* für den Empfang von IP-Paketen ist als "static" deklariert, wodurch diese von der Ethernetschicht direkt aufgerufen werden kann. Die Methode unterscheidet nun, ob das Paket ein UDP- oder ein TCP-Paket enthält und leitet die Daten an die entsprechende *receive()*-Methode weiter. Während die Methode der `UDP`-Klasse (siehe Kap. 5.6) ebenfalls "static" ist, ist dies bei TCP nicht möglich, da die Verarbeitung eines TCP-Pakets vergleichsweise sehr aufwendig ist und nicht innerhalb einer "static"-Methode erfolgen kann, da zur Laufzeit einige Parameter wie z.B. eine `Socket`-Instanz benötigt werden. Daher bietet `IP` mittels *tcpRegister()* eine Methode, mit der sich die TCP-Schicht bei der IP-Schicht registrieren muss, damit IP die entsprechenden Pakete weiterleiten kann. Ist keine TCP-Schicht registriert, so wird das Paket verworfen.

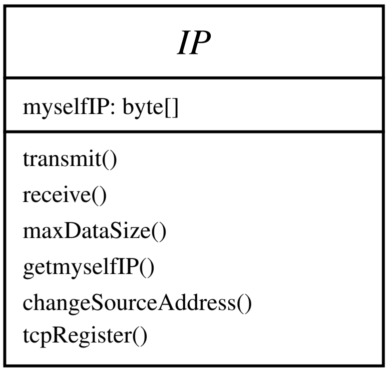


**Abb. 5.3 IP-Klasse**

`IP` bietet natürlich auch eine *transmit()*-Methode zum Versand von IP-Paketen. Dabei wird eine IP-Adresse als Zieladresse übergeben, zu welcher durch die in Kap. 5.4 beschriebenen Methoden die entsprechende Hardwareadresse bestimmt wird. Schlägt dies fehl, so schlägt auch der Versand des IP-Pakets fehl und wird durch eine Exception angezeigt.

## 5.6 Paketorientierte Datenübertragung - Klasse UDP

Die UDP-Klasse bietet auf einer einfachen Ebene die Möglichkeit zum Versand und zum Empfang von UDP-Paketen. Dabei wird nicht das Konzept eines `DatagramPacket` des JDK1.1 angeboten (siehe [JDK1]), sondern ein einfacheres Prinzip verfolgt.

Der Versand von Daten erfolgt einfach über die *transmit()*-Methode unter Angabe von Ziel-Adresse sowie Ziel- und Quell-Port. Sind die zu übertragenden Daten größer als ein Paket maximaler Größe aufnehmen kann (wobei sich die maximale Größe an dem zu Grunde liegenden Übertragungsmedium orientiert- also hier an einem Ethernetpaket, da eine Aufteilung in mehrere IP-Pakete auf der IP-Schicht nicht erfolgt), so werden die Daten auf mehrere UDP-Pakete verteilt.

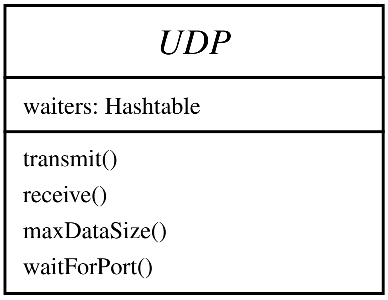


**Abb. 5.4 Klasse UDP**

Der Empfang von UDP-Paketen erfolgt über die Klasse `Waiter`. Mittels *waitForPort()* kann eine Portnummer und ein Empfangspuffer angegeben werden, in welchen die Daten übertragen werden. Die *receive()*-Methode prüft nun auf das Vorhandensein eines `Waiter` für den entsprechenden Port und überträgt die Daten in dessen Pufferspeicher. Danach wird der `Waiter` aus der Hashtable, die die `Waiter` verwaltet, entfernt und mittels seiner *notify()*-Methode über den Erhalt neuer Daten benachrichtigt.

## 5.7 Die Implementierung des TCP-Protokolls - Klasse `TCP`

Diese Klasse bietet, unterstützt durch einige weitere Hilfsklassen, eine einfache Implementierung des TCP-Protokolls an. Wie bereits erwähnt, bildet diese Klasse im Gegensatz beispielsweise zur `UDP`-Klasse noch nicht die Schnittstelle zum Benutzer; es ist jedoch auch möglich, direkt auf die Methoden dieser Klasse zuzugreifen und somit quasi “von Hand” eine TCP-Verbindung aufzubauen.

Eng verbunden mit dieser Klasse ist die `Socket`-Klasse (siehe zur näheren Erläuterung auch Kap. 5.8), welche die meisten Parameter für die Verarbeitung von TCP-Daten beinhaltet, beispielsweise die aktuelle Sequenz- oder Acknowledgement-Nummer. Somit ist eine Referenz zu einem `Socket`-Objekt für die meisten Methoden ein notwendiges Argument.

Beim Erzeugen einer `TCP`-Instanz ist eine Referenz zur IP-Schicht notwendig, da sich `TCP` an dieser für den Empfang von TCP-Paketen registrieren muss (siehe Erläuterung in Kap. 5.5). Im Konstruktor der Klasse `TCP` wird dann noch zur Implementierung der wiederholten Übertragung von Segmenten bei ausbleibender Bestätigung ein Retransmit-Timer gestartet. Das Konzept wird näher in Kap. 5.7.4 erläutert.

### 5.7.1 Verbindungsaufbau - Methoden *open()* und *listen()*

`TCP` bietet Methoden, die im Prinzip verschiedenen Zuständen eines TCP-Systems zuzuordnen sind. Mit der Methode *open()* kann ein aktiver Verbindungsaufbau initiiert werden. Die korrespondierende Methode für einen passiven Verbindungsaufbau ist *listen()*. Die Methode *open()* ermöglicht es, direkt einen lokalen Port anzugeben, über welchen die Verbindung aufgebaut werden soll oder es kann dem System überlassen werden, einen noch unbenutzten Port zu finden. Wird schon ein Port vorgegeben, wird mittels der Methode *checkPort()* geprüft, ob dieser Port schon belegt ist. Im Fehlerfall wird dem Benutzer durch die `PortUsedException` mitgeteilt, dass der Port schon belegt ist. Soll das System den Port bestimmen, wird dies durch die Methode *randomPort()* erledigt, die auch prüft, ob der Port noch zur Auswahl verfügbar ist. Sind jedoch schon sehr viele Ports belegt (Anzahl in der Nähe von 65000) kann dies sehr lange dauern, da nun die Wahrscheinlichkeit, einen freien Port zu treffen gering ist und die Methode somit sehr viele Versuche benötigen kann, einen entsprechenden Port zu finden. Praktisch wird der Fall wohl seltener auftreten, da diese TCP-Implementierung kaum für einen Server benutzt werden wird, der mehrere tausend Verbindungen offen hält.

Die Methode *open()* setzt nach Prüfung des Ports die entsprechenden Variablen im übergebenen `Socket`-Objekt, unter anderem die Adressen und Ports, den Zustand (Übergang in SYNSENT) und wählt über einen einfachen Zufallsmechanismus eine initiale Sequenznummer (*getISS()*) um Sequenznummerattacken, welche durch die Wahl einer immer gleichen initialen Sequenznummer leicht möglich wären, zu entgegnen.

Intern verwaltet `TCP` eine Liste von Sockets, um beispielsweise die Ports prüfen zu können und eingehende Pakete an den entsprechenden `Socket` weiterzuleiten. Durch die *open()*-Methode wird der Socket in diese Liste eingetragen. Am Ende wird nun ein SYN-Paket durch die *send()*-Methode versendet, welche ihre benötigten Daten aus dem Socket entnehmen kann (genauere Erläuterung zu *send()* siehe Kap. 5.7.4).

Die Methode *listen()* hat intern einen ähnlichen Ablauf. Bei *listen()* kann natürlich nur der lokale Port angegeben werden, der wie schon beschrieben überprüft wird (falls er schon belegt ist, wird dies wieder durch eine Exception mitgeteilt). Bei *listen()* ist es möglich, anhand eines Timeout-Wertes zu bestimmen, wie lange auf eingehende Verbindungen gewartet werden soll, wobei durch den Wert null angegeben werden kann, dass *listen()* "unendlich" lange warten soll und nicht durch einen Timeout abgebrochen wird. Auch bei dieser Methode werden entsprechende Parameter im `Socket`-Objekt gesetzt, der Zustand geht in LISTEN über und am Ende wird der `Socket` der Liste der Sockets hinzugefügt. Ein Paket wird bei einem *listen()*-Aufruf natürlich nicht versendet, es ist nur der Zustand und die Parameter des `Socket` entscheidend, um Verbindungsanforderungen durch ein *open()* der Gegenseite annehmen zu können.

### 5.7.2 Verbindungsende und Verbindungsabbruch - *close()* und *abort()*

Eine Verbindung wird normalerweise nach Spezifikation mittels eines *close()*-Aufrufes des Benutzers beendet. Die Methode *close()* der Klasse `TCP` prüft in diesem Fall den Zustand, in dem sich das System befindet. Befindet sich das System im ESTABLISHED-Zustand, d.h. besteht eine Verbindung oder wurde ein Paket zum Verbindungsaufbau empfangen (SYNREC), so wird ein Paket mit gesetztem FIN-Flag versendet und der Zustand geht in den FINWAIT-1 Zustand über. Befand sich das System in keinem der beiden genannten Zustände, geschieht nichts.

Zum Verbindungsabbruch existiert die Methode *abort()*. Um eine Verbindung beispielsweise im Fehlerfall abzubrechen, wird ein Paket mit gesetztem RST-Flag erzeugt und versendet, vorrausgesetzt das `TCP`-System befand sich nicht im LISTEN-Zustand oder bereits im CLOSED-Zustand. Die Zustandsvariablen des `Socket` werden zurückgesetzt, der `Socket` über den Verbindungsabbruch informiert (über die Methode *abortConnection()* - siehe Kap. 5.8.3) und der `Socket` aus der Liste der Sockets entfernt.

### 5.7.3 Der Empfang von TCP-Segmenten - *tcpHandler()*

Die Methode *tcpHandler()* entspricht den *receive()*-Methoden beispielsweise der `IP`- oder `UDP`-Klasse, wobei diese Methode natürlich bei `TCP` wesentlich komplexer ausfällt. Die Methode erwartet ein komplettes IP-Paket, d.h. die *receive()*-Methode der IP-Schicht darf ihren eigenen Header vor der Weitergabe noch nicht entfernen, wie das z.B. bei der Übergabe von `Ether` zu `IP` getan wird. `TCP` benötigt einige Einträge des IP-Protokollkopfs, um die Prüfsumme über den sogenannten Pseudo-Header berechnen zu können.

Bevor jedoch diese Prüfsumme berechnet wird, wird erst anhand der `Socket`-Liste entschieden, für welche Verbindung (und somit für welches `Socket`) das empfangene Segment bestimmt ist. Zuerst wird nach Sockets mit einem *open()*-Aufruf gesucht (erkennbar daran, dass unter anderem der Ziel-Port nicht null ist, wie dies bei einem *listen()*-Aufruf der Fall ist, da bei passivem Verbindungsaufbau ja der Port des Gegenübers noch nicht bekannt ist). Wird hier kein `Socket` gefunden, wird nach einem `Socket` mit *listen()*-Aufruf gesucht. Wird hierbei auch kein entsprechender `Socket` gefunden, liegt ein Fehler vor, das Segment wird von keiner Verbindung erwartet und daher verworfen.

Ansonsten wurde ein passender `Socket` gefunden und es wird mittels der Hilfsklasse `PseudoHeader` (wieder eine Formatklasse, die einfachen Zugriff auf die entsprechenden Felder bietet und die Prüfsumme berechnen kann) die Prüfsumme über den Pseudo-Header kombiniert mit der Prüfsumme über das komplette TCP-Segment berechnet. Dabei erfolgt dies in zwei Schritten. Zuerst wird nur die Prüfsumme über das TCP-Segment berechnet, welche in ein entsprechendes Feld des `PseudoHeader` eingefügt wird (Erweiterung des `PseudoHeader` gegenüber der Spezifikation). Da die Prüfsummenberechnung additiv ist und somit aufgeteilt werden kann, wird nun die Prüfsumme über den `PseudoHeader` mit der Erweiterung berechnet, was der kompletten Prüfsumme entspricht. Wird hierbei ein Fehler festgestellt, wird das Paket verworfen.

Wird kein Prüfsummenfehler festgestellt (d.h., die Prüfsumme ist 0xffff) so wird geprüft, ob das Paket das RST-Flag gesetzt hat. In diesem Fall wird die Verbindung geschlossen, wieder der `Socket` benachrichtigt und aus der Liste der Sockets entfernt.

Ist dies kein Reset-Paket, hängt das weitere Vorgehen vom Zustand des TCP-Systems ab. Die Verarbeitung wird durch ein großes switch-Konstrukt abhängig vom Zustand, der im jeweiligen `Socket` gespeichert ist, bestimmt.

Es wird im folgenden nicht auf jeden Zustand eingegangen, da das Vorgehen im Prinzip immer gleich ist: es werden die gesetzten Flags geprüft, danach je nach Zustand die Sequenznummer und Acknowledgement-Nummer ausgelesen und entsprechend inkrementiert. Es findet gegebenenfalls ein Zustandsübergang statt und am Ende wird ein Antwortpaket versendet.

In den Zuständen, in denen Daten empfangen werden können (also in ESTABLISHED, FINWAIT1 und FINWAIT2), wird zur Bearbeitung noch eine weitere Methode verwendet, die die empfangenen Daten weiterleitet und hier noch näher betrachtet werden soll.

Der interessanteste Fall ist der gewöhnliche Empfang von Daten, wenn die Verbindung normal aufgebaut ist und das System sich somit im Zustand ESTABLISHED befindet. Hier wird durch die im Paket enthaltene Acknowledgement-Nummer bestimmt, wieviele Daten von der Gegenseite bestätigt wurden und somit nicht mehr für die erneute Übertragung von diesem `TCP` gepuffert werden müssen. Dem `Socket` wird dies über die Methode *ackReceived()* mitgeteilt (die Implementierung der "Retransmission" wird näher unter Kap. 5.7.4 erklärt - an dieser Stelle sei bemerkt, dass ausgelöst durch diese Methode entsprechend bestätigte Daten aus dem Puffer gelöscht werden). Es wird dann die Sequenznummer entsprechend der erhaltenen Bestätigung erhöht und die weitere Verarbeitung an die Methode *processData()* weitergeleitet.

*processData()* berechnet nun, wieviele neue Daten empfangen wurden. Dabei ist es möglich, dass ein Segment ausserhalb der sequenziellen Reihenfolge empfangen wurde ("out-of-order"). Mann könnte nun dieses Paket schon speichern und selektiv bestätigen (SACK Option - siehe Kap. 3.5.11) oder auf die Pakete, die die Lücke im Strom bis zu diesem Paket füllen, warten, und dann komplett die Bestätigung schicken (das normale Vorgehen einer aufwendigeren Implementierung). Diese Implementierung unterstützt jedoch keine out-of-order-Segmente, daher wird dieses Segment verworfen und da eine Bestätigung dafür ausbleibt, auch wenn die "Lükken-Segmente" empfangen wurden, später von der Gegenseite erneut übertragen. Es werden also nur Segmente verarbeitet, die Daten enthalten, die an die bereits empfangenen Daten ansetzen, wobei es möglich ist, dass das Segment einen Teil bereits empfangener Daten enthält. Diese "alten Daten" werden erkannt und übersprungen, nur die neuen Daten werden an den `Socket` weitergeleitet. Dies wird über die Methode *dataReceived()* des `Socket` erledigt (nähere Erläuterung siehe Kap. 5.8.3). Kann der Puffer im `Socket` diese Daten nicht mehr aufnehmen, da die Applikation die empfangenen Daten nicht schnell genug entgegennehmen kann, werden die Daten wieder verworfen und es wird auf eine Wiederholung des Segments gewartet.

Nach der Verarbeitung bzw. Weiterleitung der empfangenen Daten wird anhand der neu gesetzten Werte wie z.B. der Acknowledgement-Nummer im `Socket`-Objekt eine Bestätigung zurückgesendet. Dabei wurde vorher geprüft, ob durch ein gesetztes FIN-Flag im empfangenen

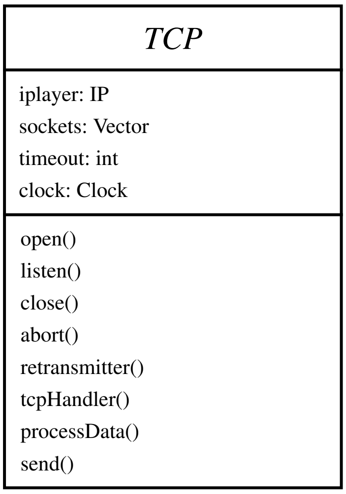


**Abb. 5.5 Die TCP-Klasse**

Segment die Verbindung von der Gegenseite beendet wird. In diesem Fall werden vorher die entsprechende Zustände angepasst (z.B. Übergang aus ESTABLISHED nach LASTACK) und entsprechend selbst ein FIN-Paket versendet.

### 5.7.4 Der Versand von Daten und "Retransmission" - Methoden *send()*

Der Versand eines TCP-Segments erfolgt über die beiden *send()*-Methoden. Sie sind beide vom Aufbau gleich, wobei die eine Methode nur für den Versand von beispielsweise Bestätigungssegmenten dient, während die andere Methode zusätzlich Daten versendet. Dabei wird wieder vorausgesetzt, dass die übergebenen Daten die maximale Paketgröße, die durch die Größe des Ethernet-Pakets abzüglich aller benötigten Header bestimmt ist, nicht überschreitet. Da die Methode durch den Outputstream des `Socket` aufgerufen wird, wird dies dort berücksichtigt (siehe Kap. 5.8.3).

Die *send()*-Methode zum Versand von Daten prüft zu Beginn, ob der Zustand des TCP-Systems ESTABLISHED ist; nur bei einer bestehenden Verbindung können in einem Segment auch Daten enthalten sein. Dann wird entsprechend der Größe der zu versendenden Daten und dem benötigten Platz für alle Header (also `TCP`, `IP` und `Ether`) Speicher reserviert (die zweite Variante benötigt nur Platz für alle Header). Es werden die entsprechenden Protokollkopfeinträge geschrieben, u.a. die Sequenznummer, Acknowledgement-Nummer oder die Fenstergröße. Hierbei wird immer ein fester Wert von 1024 eingetragen. Die derzeitige Implementierung beherrscht keine Flußkontrolle über Änderungen der Fenstergröße. 1024 ist ein guter Kompromiß, da die Empfangspuffer wesentlich mehr aufnehmen können und somit die Übertragung nicht sofort in einen Engpaß geraten sollte. Falls doch zu viele Daten empfangen werden und nicht mehr entgegengenommen werden können, wird wie schon in Kap. 5.7.3 beschrieben, von der

Methode *processData()* das Segment verworfen und somit nicht bestätigt, was eine erneute Übertragung von der Gegenseite zur Folge hat. Dadurch wird natürlich eine erhöhte Netzlast erzeugt und es werden möglicherweise Pakete sehr oft übertragen, bevor sie wirklich verarbeitet werden können, aber unter dem Ziel, einen einfachen und nur grundlegenden TCP-Teil zu implementieren, scheint diese Vorgehensweise tragbar.

Beim Verbindungsaufbau wird von der Methode *send()* die *Maximum Segment Size* Option (siehe Kap. 3.5.3) verwendet, um der Gegenseite die maximal akzeptable Segmentgröße mitzuteilen. Hierbei wird als maximale Größe 1400 Bytes angegeben. Somit wird sichergestellt, keine Pakete verarbeiten zu müssen, die die Größe eines maximalen Ethernetpakets überschreiten.

Am Ende werden je nach Methode die Daten eingefügt, mittels des Pseudo-Headers die Prüfsumme berechnet und über die *transmit()*-Methode der Klasse `IP` versendet.

Nach dem Versand puffert die *send()*-Methode (nur die Variante, die Daten versendet) das erzeugte Segment, um es im Fall eines Ausbleiben einer Bestätigung erneut versenden zu können. Zur Implementierung der verwendeten Strategie dienen Objekte der Klassen `TcpRetransmitSegment`, `RetransmitArg` und `Retransmit`. Jeder `Socket` verwaltet eine Liste von sogenannten `TcpRetransmitSegment`-Objekten. Diese Objekte enthalten das in der *send()*-Methode erzeugte Segment, dazu die entsprechende erste und letzte Sequenznummer der im Paket enthaltenen Daten, den Zeitpunkt (wieder in "Ticks" der `Clock`-Klasse), zu dem das `TcpRetransmitSegment` in die Liste eingefügt wurde, sowie die Anzahl der erneuten Übertragungen des Segments. Bei der Erzeugung der Klasse `TCP` wird ein Retransmit-Timer gestartet (Klasse `Retransmit`). Dieser ruft alle drei Sekunden die Methode *retransmitter()* der Klasse `TCP` auf.

Diese Methode implementiert die komplette "Retransmission"-Strategie. Es wurde oben erwähnt, dass nur von der *send()*-Methode, welche Daten versendet, `TcpRetransmitSegment`-Objekte in die Liste eingefügt werden. Dies hat den Grund, dass es noch einen zweiten Mechanismus für Zustände gibt, in denen keine Daten versendet werden (z.B. beim Verbindungsaufbau). Hierzu gibt es das Marker-Flag `unhappy` in der `Socket`-Klasse, das beim Versand eines Pakets in diesen Zuständen gesetzt wird. Die Methode *retransmitter()* prüft nun zuerst dieses Flag und ruft gegebenenfalls erneut *send()* mit dem Socket als Argument auf, worauf wieder ein initiales Segment beispielsweise beim Verbindungsaufbau versendet wird. Dies hat den Vorteil, dass solch ein kleines Segment nicht erst in die Liste eingefügt werden muss, um bei ausbleibender Bestätigung erneut versendet zu werden, sondern schnell erneut von der *send()*-Methode erzeugt werden kann, da dazu ja nur die entsprechenden Zustandsdaten aus dem `Socket`-Objekt benötigt werden, welche sich nicht geändert haben, da ja keine Antwort erfolgte. Sind jedoch schon Daten versendet worden, ist dieses Vorgehen nicht praktikabel, da nun jedesmal erneut ein Datensegment erzeugt werden müsste (Umkopieren der Daten, erneute Prüfsummenberechnung usw.). Daher wird das fertige Segment in einem `TcpRetransmitSegment` gespeichert und kann nun von *retransmitter()* direkt an die *transmit()*-Methode von `IP` übergeben werden. *retransmitter()* prüft also nach dem Test des `unhappy`-Flags, ob in der Liste des übergebenen `Socket` `TcpRetransmitSegment`-Objekte enthalten sind. Falls ein Segment schon länger als eine Aufrufzeit (RETRANSMITTIME) in der Liste steht, wird das

Segment erneut versendet und der entsprechende Zähler der Klasse `TcpRetransmitSegment` erhöht. Dieses Vorgehen dient dazu, Segmente, die gerade vor dem Aufruf von *retransmitter()* in die Liste kamen, nicht sofort zu wiederholen, sondern erst nach Ablauf des festen Timeouts. Überschreitet der Zähler der wiederholten Versendungen einen vorgegebenen Wert, wird davon ausgegangen, dass ein Fehler bei der Verbindung besteht und die Verbindung wird geschlossen. Somit wird verhindert, dass Segmente unendlich oft wiederholt werden. Nach Behandlung der Liste von *TcpRetransmitSegment*-Objekten wird gegebenenfalls noch der Timeout des `Socket` dekrementiert (bei *listen()* ist die Angabe eines Timeouts möglich). Ist die Zeit für den *listen()*-Aufruf abgelaufen, wird wiederum die Verbindung beendet.

Die Liste der `TcpRetransmitSegment`-Objekte hat natürlich eine beschränkte Größe. D.h. es kann nur eine bestimmte Anzahl von Daten gepuffert werden und erst wenn Bestätigungen für diese Daten erhalten worden sind und wieder Platz in der Liste besteht, können neue Daten versendet werden. Dazu prüft die *send()*-Methode zu Beginn, ob noch Platz in der Liste besteht. Falls nicht, ruft *send()* die Methode *waitForBufferSpace()* des `Socket`-Objekts auf und blokkiert damit, bis wieder Platz besteht. Diese Benachrichtigung über neu verfügbaren Platz geschieht durch ein entsprechendes *notify()*, wenn ein Segment bestätigt wurde und aus der Liste entfernt wurde (*deleteRetransmit()* des `Socket` - siehe auch Kap. 5.8).

Die derzeitige “Retransmission”-Strategie benutzt eine fest vorgegebene Zeit[1], um Pakete erneut zu versenden, d.h. es ist derzeit kein adaptiver Mechanismus implementiert, der anhand der Messung von Paketlaufzeiten die Zeit für eine wiederholte Übertragung dynamisch anpasst. Eine Implementierung einer solchen Strategie wäre erheblich aufwendiger und stellt somit eine Option für zukünftige Ergänzungen dieser Implementierung dar. Dabei ist allerdings nicht klar, ob dadurch eine signifikante Verbesserung der Netzlast erzielt werden kann, wobei auch die erhöhte Komplexität und damit schlechteres Laufzeitverhalten des Stacks berücksichtigt werden muss, der ja komplett in Java implementiert wurde.

## 5.8 Die Benutzerschnittstelle - Socket-Klassen und Input-/Outputstreams

### 5.8.1 Einschränkungen

Wie schon erwähnt, sind die von JDK1.1 [JDK1] bekannten `Socket`-Klassen einerseits die Schnittstelle zum Benutzer als auch Argument für die Methoden der `TCP`-Klasse, welche den Zustand des System für die jeweilige Verbindung und weitere Parameter enthalten.

Diese Implementierung bietet eine `Socket`- sowie eine `ServerSocket`-Klasse, die bis auf kleine Einschränkungen dasselbe Interface wie die `Socket`-Klassen von JDK1.1 bieten. Eine Einschränkung ist, dass nicht alle Konstruktoren unterstützt werden. Nach Spezifikation gibt es Konstruktoren, denen ein symbolischer Name für den Rechner, mit dem eine Verbindung auf-

1. die vorgegebene Zeit sollte eventuell noch an das jeweilige verwendete Netzwerk bzw. dessen durchschnittliche Latenzzeit angepasst werden, um einen guten Kompromiß zwischen einerseits der Vermeidung von unnötig wiederholten Übertragungen und andererseits der Wartezeit auf Daten im Falle eines wirklichen Paketverlusts zu finden.

gebaut werden soll, übergeben wird. Da jedoch derzeit keine Namensauflösung über DNS vorhanden ist, kann dies nicht unterstützt werden. Die entsprechenden Konstruktoren geben daher eine Fehlermeldung aus und werfen eine Exception.

Darüber hinaus wird der Konstruktor, dem eine vom Benutzer spezifizierte Implementierung eines `Socket` (Klasse `SocketImpl`) übergeben wird, ebenfalls nicht unterstützt.

Die dritte Einschränkung, die jedoch relativ leicht behoben werden kann, ist die Tatsache, dass alle Konstruktoren als zusätzliches Argument eine Referenz auf eine TCP-Implementierung benötigen (also ein Objekt der Klasse `TCP`). Damit ist es möglich, pro `Socket` eine andere Implementierung zu verwenden, bei einem `Socket` beispielsweise die oben beschriebene, bei einem anderen `Socket` eine erweiterte Version. Da derzeit keine systemglobale Implementierung vorhanden ist bzw. es keine Möglichkeit gibt, durch einen Methodenaufruf am Kern eine Referenz auf eine Implementierung zu erhalten, muss dies bei der Objekterzeugung übergeben werden. Es wäre allerdings leicht möglich, dies zu ändern, indem beim Start des Systems immer eine Implementierung instanziiert wird, auf die dann zurückgegriffen werden kann. Eine Art "Mantelklasse" für ein `Socket` könnte dann ein Objekt der hier beschriebenen `Socket`-Klasse instanziieren mit der bekannten TCP-Implementierung und dieses zurückgeben.

Ferner sind einige erweiterte Konzepte, die nicht auf die vorhandene TCP-Implementierung abgebildet werden können, nur als leere Funktionen vorhanden (*set-/getSoLinger()*, *setSocketImplFactory()*,...).

### 5.8.2 Aufbau und Funktionsweise

Die Klasse `Socket` verwaltet intern alle benötigten Informationen über eine Verbindung, d.h. Zustand des Systems, die entsprechenden Adressen und Ports, gegebenenfalls ein Timeout-Wert, die Sequenz- und Acknowledgement-Nummern, das Markerflag `unhappy` sowie einige weitere Parameter, die spezifisch für diese Implementierung sind (Grund für Verbindungsabbruch und entsprechendes Markerflag). Außerdem enthält ein `Socket` die in Kap. 5.7.4 beschriebene Liste von `TcpRetransmitSegment`-Objekten sowie Methoden, um diese zu verwalten (*addRetransmit()* zum Hinzufügen eines neues Segmentes, *deleteRetransmit()* zum Entfernen bei erhaltener Bestätigung, *getRetransmitEnumeration()* zum Durchsuchen, sowie *ackReceived()*, welche die entsprechenden Segmente nach Erhalt einer Bestätigung mittels *deleteRetransmit()* löscht).

Durch die Erzeugung eines `Socket`-Objektes wird eine entsprechende TCP-Verbindung aufgebaut. Dazu wird mittels *open()* von `TCP` aktiv eine Verbindung initiiert. Diese Methode setzt auch einen Timeout-Wert im übergebenen `Socket`. Innerhalb dieses Timeouts wird nun der Zustand überprüft, der nach kurzer Zeit in ESTABLISHED übergeben sollte. Geschieht dies nicht innerhalb der vorgegebenen Zeit oder geht das System in einen Zustand, der "hinter" ESTABLISHED liegt, d.h. z.B. in einen Verbindungsabbauzustand, so liegt ein Fehler vor. Es konnte keine Verbindung aufgebaut werden, was durch eine Exception angezeigt wird. Anson-

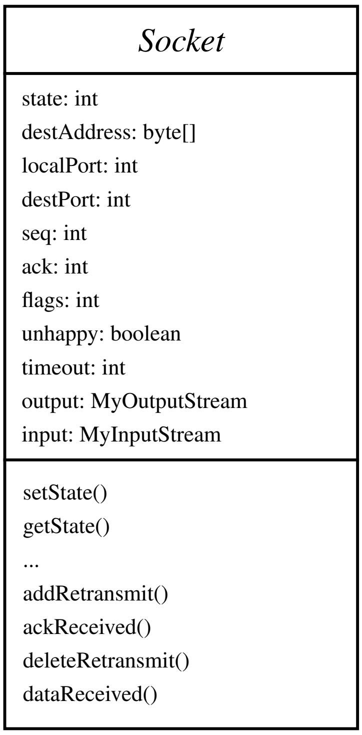


**Abb. 5.6 Socket-Klasse im Überblick mit zusätzlichen Methoden**

sten ist nach Rückkehr aus dem Konstruktor eine TCP-Verbindung aufgebaut worden, über die nun Daten übermittelt werden können. Dies geschieht in der bekannten Weise über Input- bzw. Outputstream-Objekte, welche in Kap. 5.8.3 erläutert werden.

Ein Abbau einer Verbindung erfolgt durch den Aufruf von *close()*, was direkt an die TCP-Implementierung weitergereicht wird. Ferner gibt es natürlich die bekannten Methoden, um Zustandsinformation zu lesen und zu schreiben, wie beispielsweise die Adressen und Port-Nummern. Für das TCP-System gibt es darüber hinaus Methoden, um die für die Implementierung relevante Information zu verarbeiten (Zustand, Sequenznummer, ...).

Die Klasse `ServerSocket` verwendet die beschriebene `Socket`-Klasse. Bei einem erfolgreichen Verbindungsaufbau wird von der Methode *accept()* ein `Socket` zurückgegeben, über das dann die Kommunikation in normaler Weise erfolgt. Die in Kap. 5.8.1 beschriebenen Einschränkungen sind bei `ServerSocket` geringer. Jedoch wird auch hier der Umgang mit `SocketImpl` nicht unterstützt. Ansonsten sind jedoch alle Methoden implementiert, auch *set-/getSoTimeOut()*.

`ServerSocket` verwaltet intern eine sogenannte “Backlog-Queue”, in der eingehende Verbindungsanforderungen gepuffert werden. Die maximale Anzahl von gepufferten Verbindungsanforderungen beträgt 50, welche auch mittels des Konstruktor-Parameters nicht überschritten werden kann. Wird kein Wert angegeben, wer maximal drei Anforderungen gepuffert. In diese Queue werden Objekte der Klasse `Socket` gespeichert, auf welche dann ein *listen()* der `TCP`-Klasse ausgeführt wird. Verbindungsanforderungen werden nun ganz normal über das TCP-System an diese Sockets weitergereicht. Mittels *accept()* kann eine Verbindung entgegen genommen werden. Dabei kann durch *setSoTimeout()* vor dem Aufruf dieser Methode eine Zeit festgelegt werden, die *accept()* maximal auf eine Verbindung wartet. Wird kein Timeout-Wert angegeben, kehrt *accept()* erst zurück, falls eine Verbindung aufgebaut wurde. *accept()* durchsucht nun die oben beschriebene Backlog-Queue nach Sockets, deren Zustand sich auf ESTABLISHED geändert hat, welche also eine fertige Verbindung beschreiben und gibt den ersten gefundenen `Socket` zurück, wobei wieder ein neuer `Socket` in die Queue eingefügt wird. Wird keine aufgebaute Verbindung gefunden und ist ein Timeout gesetzt worden, so wird nach Ablauf dieser Zeit mittels einer Exception die Funktion beendet.

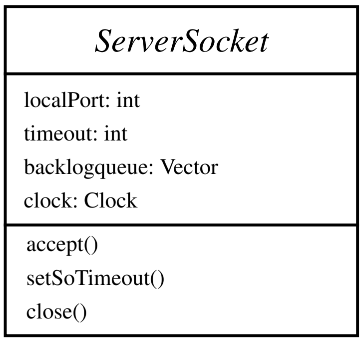


**Abb. 5.7 Das ServerSocket**

### 5.8.3 Datenübertragung über Streams

Die Schnittstelle zum Lesen und Schreiben über eine aufgebaute Verbindung stellen sogenannte Streams dar, ein Inputstream für das Lesen sowie ein Outputstream zum Schreiben. Es wurden dafür zwei Klassen implementiert, die das Interface der JDK1.1 Streams anbieten und intern auf die TCP-Implementierung zugreifen.

Mittels *getOutPutStream()* bzw. *getInputStream()* der `Socket`-Klasse wird eine Instanz von `MyOutputStream` bzw. `MyInputStream` zurückgeliefert. Diese Streams verwenden intern die *send()*-Methode von `TCP`, um die Daten zu versenden. Dabei ist eine einfache Strategie zur Verhinderung des "Silly window syndrome" implementiert (siehe Kap. 3.5.5). Aufrufe zum Versand von Daten mittels *write()* werden nicht sofort an *send()* weitergeleitet, falls die Datenmenge kleiner 40 Bytes ist. Stattdessen werden diese Daten in einen internen Puffer kopiert. Erreicht der Puffer eine Füllung von mindestens 250 Bytes, werden die Daten versendet. Damit nicht der Fall eintritt, dass die Daten zu lange auf den Versand warten müssen bzw. nie gesendet werden, weil der Puffer sich nicht weiter füllt, wird ein Timer gestartet (Klasse `Flusher` und `FlushArg`), der alle zehn Sekunden mittels eines Aufrufs der Methode *flush()* von `MyOutputStream` den Puffer versendet. D.h. kleine Datenfragmente werden nicht sofort versendet, sondern gepuffert, wobei dieser Puffer gesendet wird, wenn er eine bestimmte Größe überschreitet oder ein Timer aufgerufen wird, je nachdem was zuerst auftritt.

Sind die Daten, die übertragen werden sollen, größer als 40 Bytes, werden sie sofort mittels *send()* von `TCP` versendet, wobei dann weiter geprüft wird, ob ein Paket ausreichend ist (maximale Paketgröße abzüglich der Protokollköpfe) oder ob mehere Aufrufe an *send()* mit jeweils entsprechender Datengröße notwendig sind. Dabei kann *send()* wie bereits beschrieben blockieren, falls kein Platz mehr für weitere `TcpRetransmitSegment`-Objekte besteht. Wird eine Bestätigung erhalten, kann *send()* ein weiteres Segment versenden und somit wird auch *write()* deblockiert.

*write()*-Aufrufe prüfen zu Beginn, ob die Verbindung überhaupt noch besteht, das heisst, ob nicht ein Fehler aufgetreten ist. In Fehlerfällen wurde das `Socket` von der TCP-Schicht mittels *abortConnection()* der Klasse `Socket` informiert, wobei auch ein Grund als String übergeben werden kann. *write()* prüft dies durch die Methode *connectionAborted()* der `Socket`-Klasse, gibt gegebenenfalls eine Meldung aus und wirft eine Exception.

`MyInputStream` ist der Stream zum Lesen von empfangenen Daten. Intern werden diese gepuffert; falls dieser Puffer voll ist, werden empfangene Segmente vom TCP-System verworfen und dadurch auch nicht bestätigt, was eine Wiederholung zur Folge hat (siehe Kap. 5.7.3). Mittels der Methode *insertReceived()* werden die Daten vom `Socket` (das `TCP`-System übergibt die Daten mittels *dataReceived()* an den `Socket`, der sie dann an den Stream weiterleitet) an den Stream weitergeleitet und im internen Puffer gespeichert. Diese Methode prüft vor Datenübername, ob dafür genügend Platz zur Verfügung steht und gibt entsprechend *true* bzw. *false* zurück. Der Puffer wird dabei intern als Ringpuffer verwaltet.

Mittels der bekannten verschiedenen *read()*-Methoden können Daten von der Verbindung gelesen werden. Sind bei Aufruf der Methode keine Daten im Puffer vorhanden, so blockiert die Methode an einem internen "lock-Objekt" und wird bei Datenempfang über *insertReceived()* wieder aufgeweckt.

## 5.9 Zusammenfassung

Dieses Kapitel hat einen umfangreichen Überblick über die Implementierung des Netzwerkstacks gegeben. Dabei wurde beginnend bei der Schnittstelle zum Netzwerktreiber beschrieben, wie von diesem Daten an den Stack übergeben werden. Das allen Klassen zugrunde liegende Konzept der `Format`-Klassen wurde dargestellt. Darauf folgend wurde eine Darstellung der unteren Schichten gegeben: die `IP`-Klasse sowie die Adressauflösung über `ARP`, dann die Implementierung des `UDP`-Protokolls. Einen großen Teil nahm die Erklärung der TCP-Implementierung in Anspruch. Aufbauend auf den prinzipiellen Zusammenhängen wurde der interne Ablauf bei den wichtigsten Operationen erklärt: aktiver als auch passiver Verbindungsaufbau, der Empfang von Segmenten sowie die Verarbeitung der Daten. Dabei wurden auch Einschränkungen dieser Implementierung wie feste "Retransmission"-Timer oder statische Fenstergröße angesprochen. Die Implementierung der Benutzerschnittstelle über das `Socket`-Interface sowie die dazu gehörigen Input- und Outputstreams schließen die Darstellung des Systems. Grafiken der wichtigsten Klassen gaben einen schnellen Überblick über den jeweiligen Aufbau. Zwei große Übersichtsgrafiken über den Ablauf des Datenempfangs bzw. den Datenversands runden diesen Überblick ab.

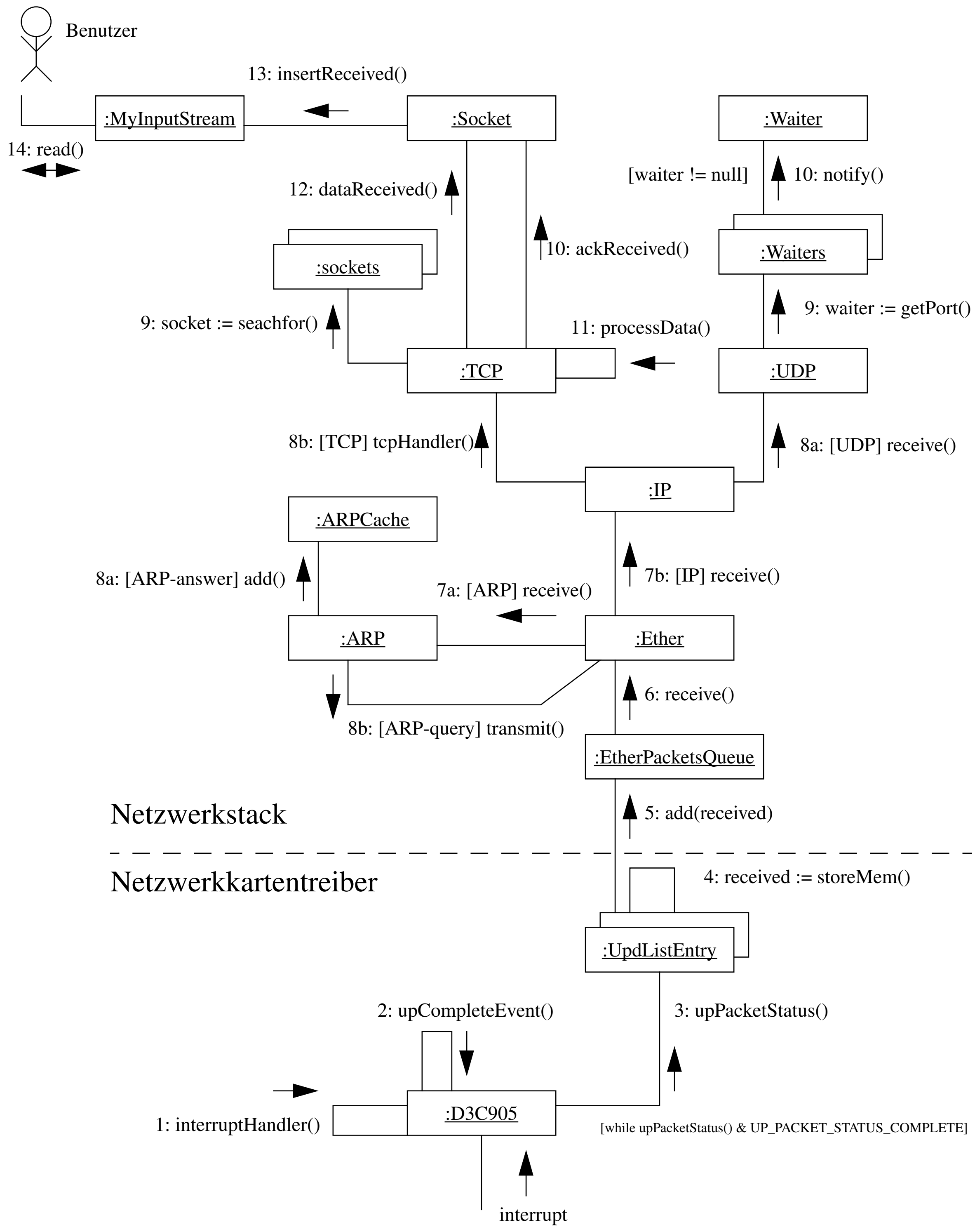


**Abb. 5.8 Ablauf bei Datenempfang**

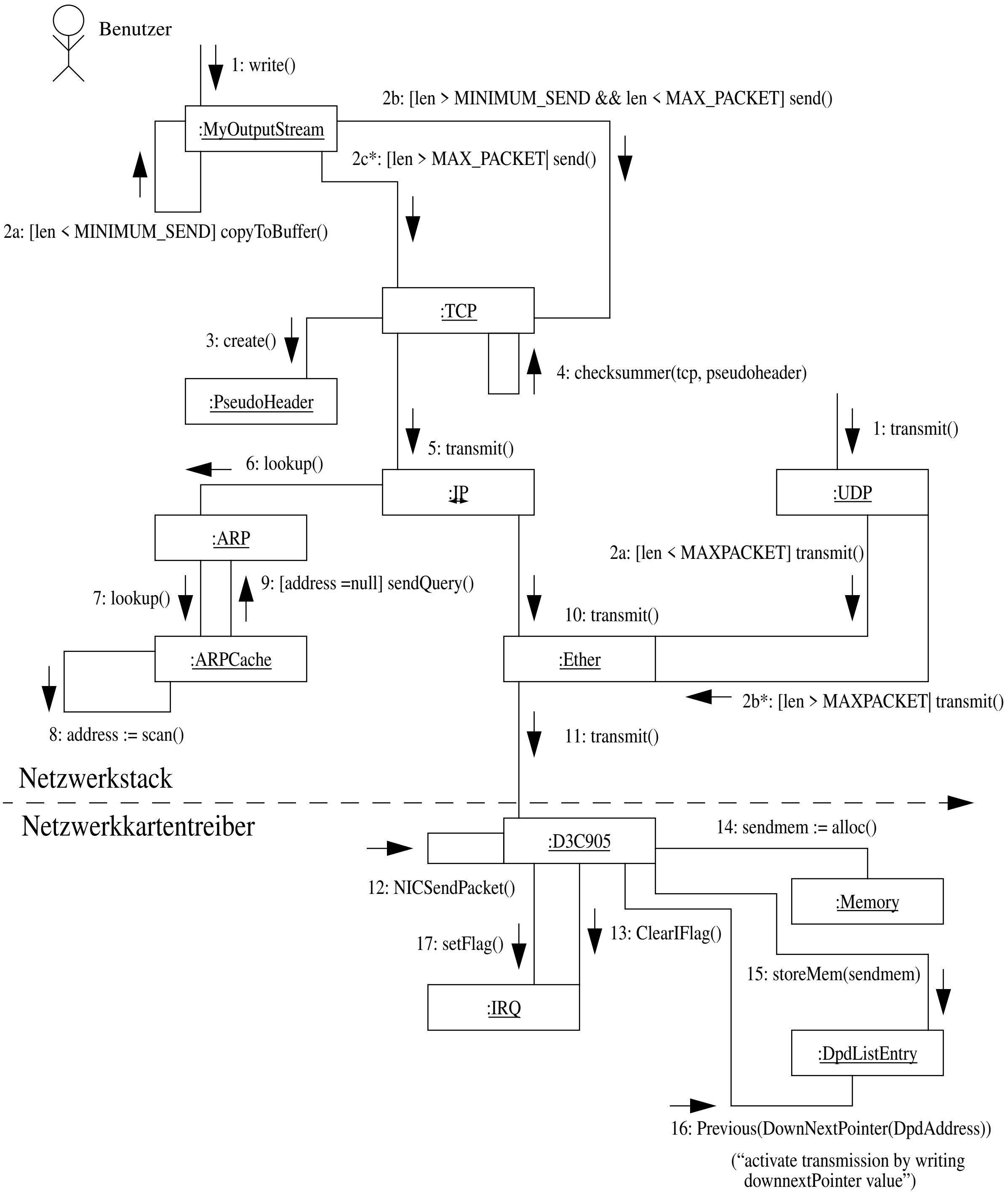


**Abb. 5.9 Ablauf beim Versand von Daten über UDP oder TCP**

# 6 Abschließende Betrachtungen

Dieses Kapitel beschreibt nochmals kurz die erzielten Ergebnisse und stellt mögliche Erweiterungen für die Zukunft vor.

## 6.1 Der Netzwerkkartentreiber

Der implementierte Treiber ist komplett und bietet mehr Funktionalität, als für die zur Verfügung stehende Karte benötigt wird, da auch viele Methoden vorhanden sind, die nur für spezielle Versionen der Kartenserie notwendig sind (beispielsweise Konfiguration eines Glasfaser-Anschlusses). Dabei konnten natürlich nicht alle Möglichkeiten getestet werden, da nicht alle vom Treiber unterstütze Hardware zur Verfügung stand.

Der Treiber ist dadurch natürlich sehr komplex, wobei versucht wurde, durch Definition von Klassen für verschiedene Teile der Hardware (z.B. Mii, Eeprom) eine gute Unterteilung zu bieten. Es ist möglich, die größte Klasse D3C905 kompakter zu implementieren, wenn alle nicht für die Karte benötigten Methoden entfernt würden. Ein weiter Punkt zur Vereinfachung bietet die Konfiguration der Karte. Es wäre natürlich möglich, fest die Konfiguration vorzugeben und dadurch könnte die gesamte automatische Detektion z.B. des Anschlusses oder der Eigenschaften entfallen. Natürlich ist dies mit einem Verlust der Flexibilität des Treibers verbunden, wobei der erzielbare Performanzgewinn sich auch nur beim Start des Systems auswirken würde.

Die Implementierung der Deskriptoren und somit des Sende- und Empfangskonzepts bietet kaum Raum für Verbesserungen, um eine höhere Geschwindigkeit zu erreichen. Es wäre gegebenenfalls möglich, die Anzahl der Methodenaufrufe für die Erzeugung eines Deskriptors zu verringern, was allerdings die Verständlichkeit des Treibers eher verringern würde und nicht allzuviel Gewinn erwarten läßt.

## 6.2 Der Netzwerkstack

Der Netzwerkteil dieser Arbeit bietet sicherlich wesentlich mehr Raum für Erweiterungen und Verbesserungen, was schon dadurch begründet ist, dass eine Implementierung des TCP-Protokolls mit all seinen in Kap. 3.5 beschriebenen Erweiterungen sehr komplex ist. Der implementierte Stack genügt jedoch sicherlich dem Ziel der Implementierung eines einfachen TCP-Systems.

Aufbauend auf dem vorhanden TCP-Stack wären als erstes eine Erweiterung um Flußkontrolle über die Empfangs-Fenstergröße und adaptive Timeouts für die “Retransmission” von Paketen, deren Bestätigung ausgeblieben ist, zu nennen. Darauf folgend bietet das TCP-Protokoll mit seinen Erweiterungen genügend Optionen für weitere Ergänzungen, u.a. Unterstützung von mehr Optionen als nur der Maximum Segment Size Option oder Implementierung von Algorithmen zur Verhinderung von Netzwerkengpässen (“network congestion control” - siehe u.a. [JK88]).

Auch auf Ebene des IP-Protokolls sind Erweiterungen möglich, da hier nur grundlegende Fähigkeiten zum Versand und Empfang von IP-Paketen vorhanden sind. Eine Implementierung des Routings von IP-Paketen sollte als erstes in Angriff genommen werden. Unabhängig davon wären Namensauflösung über DNS und die Ergänzung um das ICMP-Protokoll ein weiteres Ziel.

Der Datenaustausch im Stack und der Zugriff auf die Daten erfolgt über die beschriebenen Formatklassen. Dieses Konzept wurde bei TCP verwendet, da die schon vorhandenen Teile ebenso darauf aufbauten und TCP sich hier gut integrieren sollte. Eine Umstellung des Format-Konzepts auf von der Memory-Klasse abgeleitete Klassen, die einen ähnlich komfortablen Zugriff bieten, könnte die Performanz des Systems steigern, da dann kein Umkopieren der Daten mehr notwendig wäre und dieses Memory-Objekt quasi direkt über einen Deskriptor des Treibers adressiert werden könnte. Auch die Überlegung, ob direkt Deskriptoren an den Stack weitergegeben werden, die den entsprechenden Speicher beschreiben, wäre möglich. Allerdings wird dadurch die konzeptionelle Eigenständigkeit des Treibers aufgegeben und sehr eng mit dem Stack verzahnt, was eine Anpassung an andere Netzwerkhardware sehr schwierig machen würde. Zudem würde die Verwaltung der Deskriptorlisten komplexer, da sowohl Stack als auch Treiber darauf Zugriff hätten.

## 6.3 Zusammenfassung

Das implementierte System zeigt, dass auch so grundlegende Betriebssystemkomponenten wie ein Treiber oder das Netzwerksystem komplett in Java implementiert werden können. Selbstverständlich ist die Performanz nicht direkt mit einem gewöhnlichen System, das in C oder Assembler implementiert ist, vergleichbar. Dennoch ist das System natürlich voll einsetzbar. Die angesprochenen Erweiterungen zeigen Möglichkeiten für die Komplettierung des Netzwerksystems, wobei allerdings beachtet werden sollte, inwiefern die zusätzliche Komplexität sich auf das Laufverhalten auswirkt und somit möglicherweise die erwarteten Vorteile zunichte macht.

# 7 Literaturverzeichnis


| | |
|---|---|
| 3CO98 | 3COM Corporation: 3C90x and 3C90xB NICs Technical Reference, 3COM Corporation, Santa Clara, 1998. |
| 3CO00 | 3COM Corporation: Linuxtreiber für die 3C90x-Serie, online verfügbar unter Adresse http://support.3com.com/infodeli/tools/nic/linuxdownload.htm, 2000 |
| Bra89 | R. Braden (Editor), Requirements for Internet Hosts - Communication Layers, RFC 1122, 1989 |
| BB88 | R. Braden, D. Borman, Computing the Internet Checksum, RFC 1071, 1988 |
| Eck98 | Bruce Eckel: Thinking in Java, Prentice Hall PTR, NJ, 1998, online verfügbar unter http://www.bruceeckel.com |
| FLY93 | V. Fuller, T. Li, J. Yu, K. Varadhan, Classless Inter-Domain Routing (CIDR): an Address Assignment and Aggregation Strategy, RFC 1519, 1993 |
| GK00 | Michael Golm, Jürgen Kleinöder: JX: Eine adaptierbare Java-Betriebssystemarchitektur, Universität Erlangen, IMMD 4, 2000 |
| Gol00 | Michael Golm, Projektbeschreibung JX und zu vergebende Arbeiten, Universität Erlangen, IMMD4, 2000 |
| Hau97 | Dr.-Ing Franz J. Hauk, Skript zur Vorlesung Systemprogrammierung 1997, http://www4.informatik.uni-erlangen.de/Lehre/WS97/V_SP1/d100-e44-2.ps |
| HKC96 | K Hubbard, M. Kosters, D. Conrad, D. Karrenberg, J. Postel, Internet Registry IP Allocation Guidelines, RFC 2050, 1996 |
| JBB92 | V. Jacobson, R. Braden, D. Borman, TCP Extensions for High Performance, RFC 1323, 1992 |
| JDK1 | API-Spezifikation des Java Development Kit 1.1, http://java.sun.com/products/jdk/1.1/docs/api/packages.html, Sun Microsystems |
| JK88 | Van Jacobson, Michael J. Karels, Congestion Avoidance and Control, Berkeley, 1988 |
| Kau94 | Franz-Joachim Kauffels: Lokale Netze - Grundlagen, Standards, Perspektiven, Markt & Technik Verlag, Haar bei München, 1994. |
| Kopp98 | H. Kopp, Design und Implementierung eines maschinenunabhängigen Just-in-Time-Compilers für Java, Diplomarbeit (masters thesis), Universität Erlangen, IMMD 4, Oct. 1998 |
| Mes98 | Hans Peter Messmer: PC Hardware - 5. Auflage, Addison Wesley Longman GmbH, Bonn, 1998. |

ML94 James Martin, Joe Leben: TCP/IP-Netzwerke - Architektur, Administration und Programmierung, Prentice-Hall Inc., Eaglewood Cliffs, 1994

MMF96 M. Mathis, J. Mahdavi, S. Floyd, A. Romanow, TCP Selective Acknowlegment Options, RFC 2018, 1996

Plu82 D. Plummer, An Ethernet Address Resolution Protocol, RFC 826, 1982

Pos80 J. Postel, User Datagram Protocol, RFC 768, 1980

Pos81 J. Postel, DARPA Internet Program Protocol Specification - Internet Protocol, RFC 791, 1981

RL93 Y. Rekhter, T. Li, An Architecture for IP Address Allocation with CIDR, RFC 1518, 1993

Ste94 W. Richard Stevens: TCP/IP Illustrated, Volume 1 - The Protocols. Addison-Wesley Publishing Company, Reading, 1994.

Siy97 Karanjit S. Siyan: Inside TCP/IP Third Edition, New Riders Publishing, Indianapolis, 1997.

Sun Webseite der Firma Sun Microsystems zu Java, http://www.java.sun.com

SW95 W. Richard Stevens, Gary R. Wright: TCP/IP Illustrated, Volume 2 - The Implementation, Addison-Wesley Publishing Company, Reading, 1995.

Tan96 Andrew. S. Tanenbaum: Computer Networks Third Edition, Prentice-Hall, Inc., Upper Saddle River, 1996.

TCP81 Arpanet Working Group Request For Comment, DDN Network Information Center, Transmission Control Protocol Specification, RFC 793, 1981

Tin86 TinyTCP, Eine einfache Implementierung des TCP-Protokolls, erhältlich unter http://www.csonline.net/bpaddock/tinytcp/default.htm, (c) by IMAGEN Corporation, 1986

TJ97 Tischer, Jennrich: Internet intern - Technik & Programmierung, Data Becker, Düsseldorf, 1997.