



**Universität Paderborn**

Fachbereich 17 – Mathematik/Informatik  
Arbeitsgruppe Softwaretechnik  
Warburger Str. 100  
33098 Paderborn

# **Simulation von Netzwerken am Beispiel von UDP**

Studienarbeit  
zur Erlangung des Grades  
**Bachelor of Computer Science**  
für den integrierten Studiengang Informatik

von

Volker Pech  
Am Rippinger Weg 12  
33098 Paderborn

vorgelegt bei  
Prof. Dr. Wilhelm Schäfer

Paderborn, April 2001

Ich versichere hiermit, dass ich die Arbeit selbstständig angefertigt habe und keine anderen, als die angegebenen und bei Zitaten kenntlich gemachten Quellen und Hilfsmittel benutzt wurden. Diese Arbeit wurde, in gleicher oder ähnlicher Form, keiner anderen Prüfungsbehörde vorgelegt und von dieser als Teil einer Prüfungsleistung angenommen.

Volker Pech (31.04.2001)

# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung .....</b>	<b>1</b>
1.1	Wozu benötigt man Simulationen .....	1
1.2	Simulation von Netzwerken .....	2
1.3	Netzwerke in Java .....	2
1.4	Anwendungsmöglichkeiten .....	3
1.5	Aufbau der Studienarbeit .....	4
<b>2</b>	<b>Netzwerke .....</b>	<b>5</b>
2.1	LANs und WANs .....	5
2.2	Das ISO/OSI Kommunikationsmodell .....	5
2.3	Kommunikationsmodelle in der Praxis .....	8
2.4	IEEE 802 .....	9
2.5	Busarbitrierung .....	10
2.6	Die MAC-Schicht .....	14
2.7	Die LLC-Schicht .....	15
2.8	Fehlererkennung .....	15
2.9	TCP/IP .....	16
2.10	Kommunikationsgeräte .....	21
2.11	Busse in der Automatisierungstechnik .....	23
<b>3</b>	<b>Technische Realisierung der Simulation .....</b>	<b>27</b>
3.1	UDP mit Java .....	27
3.2	Aufruf der Simulation .....	30
3.3	Simulationsansatz .....	32
3.4	Hauptschleife der Simulation .....	34
3.5	Übertragungsprozesse .....	36
3.6	Einsatz des Schedulers .....	37
3.7	Arbitrierungsklassen .....	38
<b>4</b>	<b>Anwendung der Simulationsumgebung .....</b>	<b>42</b>
4.1	Einfluss der Hardware .....	42
4.2	Ausführen der Simulationsumgebung .....	43
4.3	Die Ausgabe der Simulation .....	45
4.4	Testanwendungen und Statistiken .....	48
<b>5</b>	<b>Zusammenfassung und Ausblick .....</b>	<b>53</b>
5.1	Fähigkeiten der Simulationsumgebung .....	53
5.2	Einbettung neuer Protokolle .....	54
5.3	Simulation komplexer Netzwerke .....	54
5.4	Messung, Darstellung und Analyse .....	55
5.5	Nebenläufigkeit .....	55
	<b>Literaturverzeichnis .....</b>	<b>56</b>

# Abbildungsverzeichnis

<b>1</b>	<b>Einleitung .....</b>	<b>1</b>
1.1	Aufbau einer Prozessoreinheit .....	3
<b>2</b>	<b>Netzwerke .....</b>	<b>5</b>
2.1	Das 7-Schichten ISO/OSI Basisreferenzmodell .....	6
2.2	Kommunikationsmodelle im OSI-Basisreferenzmodell .....	8
2.3	Einordnung der IEEE 802-Protokolle in das OSI-Referenzmodell .....	9
2.4	Reines Aloha .....	10
2.5	CSMA/CD Algorithmus .....	12
2.6	Kollisionserkennung bei CSMA/CD .....	13
2.7	Arbitrierungsverfahren im Vergleich .....	14
2.8	Rahmenstruktur aller Ethernet-Varianten .....	14
2.9	TCP/IP Protokollfamilie .....	16
2.10	IP-Datagrammstruktur .....	17
2.11	UDP-Paketformat .....	18
2.12	Anwendungsdienste die UDP verwenden .....	19
2.13	Anwendungsdienste die TCP einsetzen .....	20
2.14	TCP-Paketformat .....	20
2.15	Vergleich von TCP und UDP .....	21
2.16	Kommunikationsgeräte .....	22
2.17	Leitungslängen beim PROFIBUS .....	25
<b>3</b>	<b>Technische Realisierung der Simulation .....</b>	<b>27</b>
3.1	Konstruktoren der Klasse <i>DatagramSocket</i> .....	27
3.2	Methoden der Klasse <i>DatagramSocket</i> .....	28
3.3	Konstruktoren der Klasse <i>DatagramPacket</i> .....	29
3.4	Einsatz des package <i>simnet</i> .....	30
3.5	Strategy Pattern für <i>CommonSocket</i> .....	31
3.6	Einsatz der Klasse <i>SimSocket</i> .....	31
3.7	Grobes Schema für das virtuelle Netzwerk .....	32
3.8	Genaueres Schema für das virtuelle Netzwerk .....	32
3.9	Die Klassen <i>SimSocket</i> , <i>Buffer</i> und <i>BufferTransfer</i> .....	33
3.10	Die Klasse <i>Wire</i> .....	35
3.11	Die Klasse <i>Task</i> .....	36
3.12	Die Klasse <i>Scheduler</i> .....	37
3.13	Strategy Pattern für <i>CommonArbitration</i> .....	38
3.14	Die Klasse <i>CSMACDArbitration</i> .....	39
<b>4</b>	<b>Anwendung der Simulationsumgebung .....</b>	<b>42</b>
4.1	Aufruf des virtuellen Netzwerkes .....	45
4.2	Anmeldung eines neuen Netzwerkteilnehmers .....	46
4.3	Übertragung von Daten .....	47
4.4	Ausgaben der Klasse <i>SimSocket</i> .....	48
4.5	Übertragungsschema .....	48
4.6	Einfluss der Übertragungsrate .....	49

4.7 Einfluss der Bitfehlerwahrscheinlichkeit .....	50
4.8 Interferenz von 2 Video-Servern .....	50
4.9 Textübertragung ohne Fehlererkennung .....	51
<b>5 Zusammenfassung und Ausblick .....</b>	<b>53</b>
5.1 TCP/IP Protokollfamilie .....	53

# **1 Einleitung**

## **1.1 Wozu benötigt man Simulationen ?**

Viele Entwicklungen in der Welt der modernen Technik sind heutzutage sehr komplex. Der Mensch hat dadurch viele Möglichkeiten, die neu entwickelte Technologie individuell seinen Bedürfnissen anzupassen. Es entsteht allerdings oft der Nachteil der Unüberschaubarkeit. Der Benutzer stellt folgende Fragen:

- Kann ich mein Problem mit dieser neuen Technologie lösen ?
- Wie effizient kann ich mein Problem mit dieser Technologie lösen ?
- Wie setze ich die neue Technologie ein ?
- Gibt es andere Technologien die evtl. günstiger oder effizienter sind ?
- Wie lerne ich am schnellsten mit der Technologie umzugehen ?

Gerade neue Technologien bieten eine Vielzahl von Anwendungsmöglichkeiten, die oft bei ihrer Einführung auf dem Markt noch gar nicht erforscht sind. Unüberschaubarkeit kann dazu führen, dass sie ineffizient oder sogar zweckentfremdet eingesetzt werden. Sie kann auch dazu führen, dass neue geeignetere Technologien gar nicht eingesetzt werden.

Viele Technologien lassen sich heute äußerst realistisch durch Simulationen untersuchen. Es existieren zum Beispiel Simulationen von Flügen, Autorennen, Besiedlungsentwicklungen und Windkanal-Messungen. Auch das Verhalten von busbasierten Fahrzeugsteuerungen sowie die Arbeitsweise von Fertigungsrobotern in der Industrie lassen sich durch Simulationen im Vorfeld erforschen.

Selbstverständlich kann man auch auf eine Simulation verzichten und eine Technologie in einer realen Umgebung untersuchen. Es ergeben sich dabei allerdings folgende Nachteile:

- Kostenintensiver Aufbau einer Messumgebung
- Aufwendige, oft ungenaue Messverfahren
- Mangel an Flexibilität
- Schlecht Reproduzierbarkeit von speziellen Verhaltensweisen
- Beschränkung des Tests auf wenig Szenarien aus Zeit- und Kostengründen
- Fehlverhalten der Messumgebung führt zu hohen Umbaukosten

Um eine gute Simulation zu erstellen, muss man versuchen, die Realität auf ein möglichst exaktes Modell abzubilden. Eine große Rolle spielt hierbei sicher auch die Technik, mit deren Hilfe die Simulation entwickelt wird. Es kommt darauf an, dass sich diese Technik in Bezug auf die Simulation möglichst transparent verhält, damit es nicht zu einer Verfälschung der Ergebnisse kommt. Da es bei Simulationen oft darum geht, viele Berechnungen in kurzer Zeit durchzuführen, werden in der Regel Computer zu ihrer Realisierung verwendet. Simulationen auf Computern bieten folgende Vorteile:

- Die einfache Erforschung von Verhaltensweisen einer Technologie
- Der Vergleich verschiedener Einsatzarten der Technologie
- Das Erstellen von Statistiken
- Schulung von Menschen zur Vorbereitung auf den realen Umgang mit der Technologie
- Test verschiedener Szenarien
- Geringe Anschaffungskosten

## 1.2 Simulation von Netzwerken

Schon seit langer Zeit existieren Netzwerke. Aus einem Forschungsprojekt, das 1969 ins Leben gerufen wurde und vom US-Verteidigungsministerium finanziert wurde, entstand die TCP/IP Protokollreihe, die auch für das Internet verwendet wird. Sie basiert auf der Grundidee der Paketvermittlung [REVS]. Ein spezielles Protokoll dieser Protokollreihe ist UDP (*User Datagram Protocol*). UDP bietet ein vergleichsweise einfaches Verfahren um Pakete über ein Netzwerk zu verschicken. Da TCP/IP unzählige Möglichkeiten bietet, hat sich die Vernetzung von Computern seit 1969 stark entwickelt. Heute werden TCP/IP Netzwerke sowohl für das Internet als auch für interne Netzwerke von vielen Organisationen verwendet. Die Struktur dieser Netzwerke wird dabei immer komplexer, so dass der Aufbau oft mit großem Aufwand und hohen Kosten verbunden ist.

Will man also ein Netzwerk aufbauen, ist es schwer festzustellen, welche Netzwerkkomponenten benötigt werden, oder welche Netzstruktur zu empfehlen ist. Wenn man selbst Anwendungen implementieren möchte, kann auch die Wahl der beteiligten Protokolle hierbei sehr entscheidend sein. Um zu untersuchen, ob ein Netzwerk den gestellten Anforderungen genügt, muss man das gesamte Netz erst komplett verdrahten. Es entsteht so ein hoher Aufwand, der sich in der Regel nicht finanzieren lässt.

Will man ein existierende Netzwerk erweitern, so ist es sehr schwierig festzustellen, wie sich das Netzwerk mit weniger Änderungen an die neuen Anforderungen anpassen lässt. Auch hier hat man nicht die Möglichkeit einer langen Erprobungsphase und wird sich so in der Regel für eine zu aufwendige Lösung entscheiden.

Diese Arbeit soll zur Lösung dieser Probleme beitragen, indem sie einen Ansatz für eine Testumgebung liefert, die ein komplettes Netzwerk simuliert. Die Testumgebung bietet folgende Möglichkeiten:

- Existierende Applikationen, die auf ein Netzwerk zugreifen lassen sich an eine Netzwerk-Simulation anschließen. Alle Applikationen können dabei prinzipiell auf nur einem einzigen Rechner ablaufen.
- Es können virtuelle Applikationen erzeugt werden, die beliebige Geräte repräsentieren und ebenfalls an die Netzwerksimulation angeschlossen werden.

Es entfällt somit ein großer Teil des Aufwandes, der bisher nötig war, um ein Netzwerk vernünftig zu gestalten und ideal zu konfigurieren.

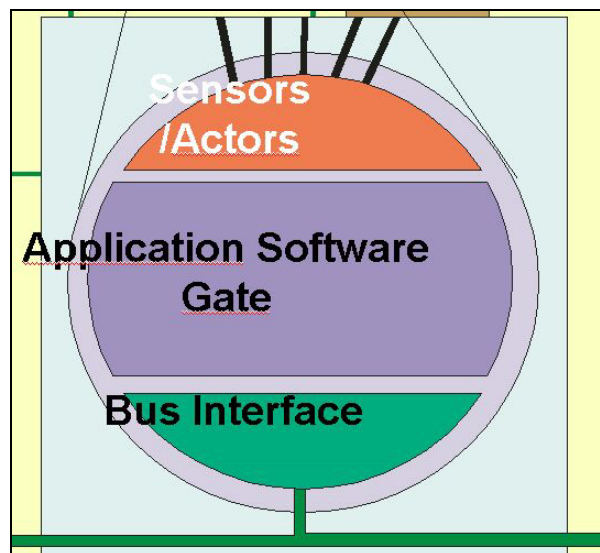
## 1.3 Netzwerke in Java

Die objektorientierte Sprache Java hat sich in den letzten Jahren im Netzwerk-Bereich stark etabliert. Eine Anforderung der Studienarbeit bestand darin, Java-Anwendungen eine Möglichkeit zu bieten, über ein virtuelles Netzwerk zu kommunizieren. Java findet man nicht nur auf Computern mit einer Virtual Machine, sondern auch in kleinen Bausteinen wie Java Cards, die von Netzwerken abhängig sind. Da Java eine plattform-unabhängige Sprache ist, eignet sie sich ideal für den Einsatz in inhomogenen Netzwerken.

In dieser Arbeit soll gezeigt werden, wie Applikationen die auf der Basis von UDP kommunizieren, auf eine einfache Weise an ein virtuelles Netzwerk angeschlossen werden können. Die Daten, die normalerweise über ein reales Netzwerk fließen, werden dabei einfach an die Simulation umgelenkt.

## 1.4 Anwendungsmöglichkeiten

Eine Anwendung der Simulation findet man bei der Analyse und Validierung von Fertigungssystemen. Das Forschungsprojekt *ISILEIT* (Integrative Spezifikation von verteilten Leitsystemen der flexibel automatisierten Fertigung, s.a. [ISILEIT]) befasst sich mit diesem Thema. Auch hier sind Ansätze gefragt, die das zeitliche Verhalten eines Netzwerkes mit einbeziehen, um das Verhalten von Fertigungssystemen exakter zu simulieren. Man kann ebenfalls untersuchen, wie sich Übertragungsfehler oder Pufferüberläufe, bei denen Daten verloren gehen, auf die Funktionalität des Fertigungssystems auswirken. So lässt sich hier relativ schnell feststellen, welchen Aufwand man betreiben muss, um eine schnelle und fehlerfreie Kommunikation zu ermöglichen. Die Simulation des Netzwerkes eignet sich dazu, in eine übergeordnete Simulation, welche ein komplettes Leitsystem simuliert, eingebettet zu werden. In Leitsystemen findet man für die Steuerung eine Vielzahl von Prozessoreinheiten, die sowohl Sensoren als auch Aktoren enthalten und über einen Feldbus kommunizieren. Abbildung 1.1 zeigt den Aufbau einer Prozessoreinheit die sich in 3 Schichten gliedern lässt:



**Abbildung 1.1:** Aufbau einer Prozessoreinheit [MSMF]

Die oberste Schicht stellt das Interface zu den Sensoren und Aktoren dar. Die mittlere Schicht kümmert sich um die Steuerungs-Logik und führt die Berechnungen aus. Auf der untersten Schicht befindet sich schließlich das Bus-Interface, welches die Prozessoreinheiten an den Feldbus anschließt. Eine Anforderung an die Studienarbeit bestand darin, einen allgemeinen Ansatz für die Simulation dieser unteren Schicht zu liefern.

Ein anderes, interessantes Anwendungsfeld ist zur Zeit die Vernetzung von Haushaltsgeräten. Hier wird ein jederzeit einwandfrei funktionierendes Netzwerk vorausgesetzt. Auch im Haushaltsbereich treten mehrere Echtzeitanforderungen auf. So ist es zum Beispiel nicht wünschenswert, dass eine Türklingel sich erst wenige Sekunden nach der Betätigung des Klingelknopfes meldet. Da die Auslastung des Netzwerkes hier sehr individuell ist, kann eine Testumgebung maßgeschneiderte Ansätze für verschiedene Kunden liefern. Man sollte hier auch die Übertragung digitaler Medien, die oft ein großes Datenvolumen aufweisen, mit in das Szenario einbeziehen. Es wird schnell klar, dass hier adäquate Voraussagen für ein langfristiges Übertragungsverhalten, nur von einer guten Simulation geliefert werden können, da sich die Schwächen eines solchen Systems unter Umständen erst nach langer Zeit zeigen.



## 1.5 Aufbau der Studienarbeit

Die vorliegende Beschreibung gliedert sich im Weiteren in 4 Kapitel. Kapitel 2 liefert eine Einführung in die relevanten Grundlagen eines Netzwerkes. In Kapitel 3 folgt dann eine ausführliche Beschreibung der Implementierung. Kapitel 4 befasst sich mit konkreten Anwendungsbeispielen, die zeigen sollen wie man die Testumgebung einsetzt. Kapitel 5 liefert schließlich einige Ausblicke in Bezug auf eine Weiterentwicklung der Simulationsumgebung.

## 2 Netzwerke

### 2.1 LANs und WANs

Bei Netzwerken geht es im allgemeinen um die Technik, mit der Rechner untereinander verbunden werden, und um die Art und Weise auf die sie kommunizieren. Diese Rechner können nah nebeneinander innerhalb eines Raumes stehen, aber auch weit entfernt in 2 verschiedenen Kontinenten beheimatet sein. Für alle Anforderungen wurden in der Vergangenheit unterschiedliche Lösungen entwickelt [REVS].

Je nach Entfernung der Partner werden diese über ein lokales Netzwerk (LAN – *Local Area Network*) oder ein Weitverkehrsnetz (WAN – *Wide Area Network*) verbunden. Im allgemeinen liegt zwischen zwei Kommunikationspartnern ein komplexes Kommunikationsnetzwerk, das mit Hilfe unterschiedlicher Kommunikationsgeräte aufgebaut ist [REVS].

Damit sich die einzelnen Computer und Kommunikationsgeräte der verschiedenen Hersteller auch verstehen, war es notwendig sich auf gemeinsame Normen zu einigen. Diese Normen müssen sowohl die Hardware als auch die Software berücksichtigen. Um den verschiedenen Anforderungen an das Kommunikationssystem flexibel begegnen zu können, haben sich in der Vergangenheit verschiedene mehrschichtige Kommunikationsmodelle, welche die Hard- und Software umfassen entwickelt [REVS].

Auf die Kommunikationsmodelle wird in dieser Arbeit insoweit eingegangen werden, wie es für das Verständnis der Simulation notwendig ist.

### 2.2 Das ISO/OSI Kommunikationsmodell

Da das weltweit wichtigste Weitverkehrsnetz zur Zeit das Internet ist, über das ein weitverzweigter Datenaustausch stattfindet, braucht man Vereinbarungen, wie die Kommunikation statzufinden hat. Bei den Vereinbarungen im Bereich der Rechnerkommunikation spricht man von Kommunikationsmodellen bzw. Protokollen [REVS].

Alle gängigen Kommunikationsarchitekturen sind heute als Schichtenmodell realisiert, da sich Kommunikationsmodelle gut in unabhängige Schichten mit eigenen Aufgaben aufteilen lassen. Jede Schicht realisiert eine Bibliothek mit Funktionen für die darüber liegende Schicht, wobei die „oberste“ Schicht das Anwendungsprogramm ist. Die „unterste“ Schicht beschreibt den Zugriff auf das physikalische Medium. Der große Vorteil der Schichtenarchitektur liegt darin, dass sich das Protokoll einer Schicht austauschen lässt, ohne dass die anderen Schichten davon betroffen sind [REVS].

Das wichtigste Referenzmodell zur Rechnerkommunikation wurde von der International Standard Organisation (ISO – eine Institution der UNO) 1984 nach Vorarbeiten der CCITT veröffentlicht und wurde unter dem Begriff

*OSI-Basisreferenzmodell bzw. OSI-7-Schichtenmodell*  
(*Basic Reference Modell for Open Systems Interconnection*)

bekannt [REVS].

Als Referenzmodell definiert das Dokument nur die Aufgaben der Protokolle auf den einzelnen Schichten, nicht jedoch die Protokolle, die Implementierung der Schichten selbst [REVS]. Abbildung 2.1 gibt eine Übersicht über die Schichten und Aufgaben.

Schicht	Name	Aufgabe
7	Anwendungsschicht ( <i>Application Layer</i> ) DIN:Verarbeitungsschicht	Festlegung des Dienstes des Kommunikationspartners für das jeweilige Anwendungsprogramm (z.B. Dateiübertragung, E-Mail,...)
6	Darstellungsschicht ( <i>Presentation Layer</i> )	Festlegung der Strukturen der Anwenderdaten inklusive Formatierung, Verschlüsselung, Zeichensetzung
5	Sitzungsschicht ( <i>Session Layer</i> ) DIN:Kommunikationssteuerschicht	Kommunikationsablauf und -abbau, d.h. Auf- und Abbau von logischen Kanälen auf dem physikalischen Transportsystem
4	Transportschicht ( <i>Transport Layer</i> )	Steuerung der Datenstroms durch Bereitstellen von fehlerfreien logischen Kanälen auf dem physikalischen Transportsystem
3	Vermittlungsschicht ( <i>Network Layer</i> )	Festlegung eines Weges für einen Datenstrom durch das Netzwerk
2	Sicherungsschicht ( <i>Data Link Layer</i> )	Sicherstellung eines korrekten Datenstromes durch Festlegung eines Datenformats für die physikalische Übertragung (Kanalkodierung) und durch Festlegung der Zugriffsart auf das Netzwerk
1	Bitübertragungsschicht ( <i>Physical Layer</i> )	Festlegung des Übertragungsmediums (elektrische und mechanische Eigenschaften)

**Abbildung 2.1:** Das 7-Schichten ISO/OSI-Basisreferenzmodell [REVS]

### **1 Bitübertragungsschicht:**

Die Bitübertragungsschicht dient zur Übertragung von Bitfolgen ohne jeglichen Rahmen. Folgende Aspekte müssen berücksichtigt werden [REVS]:

- Kanalkodierung (NRZ, Manchester, ...)
- Modulationsart bzw. Basisbandübertragung
- Multiplexverfahren (Frequenzmultiplex/Zeitmultiplex)
- Betriebsarten (Simplex/ (Halb-)Duplex,synchron/asynchron, seriell, parallel).

### **2 Sicherungsschicht (Verbindungsschicht)**

Die Dienste auf der Sicherungsschicht sorgen für die Übertragung von Rahmen (einige 100 Byte), die Bitweise an die Bitübertragungsschicht weitergegeben werden, sowie die Behandlung von Übertragungsfehlern [REVS].

### **3 Vermittlungsschicht (Netzwerkschicht)**

Die Vermittlungsschicht dient der Paketübertragung vom Sender zum Empfänger (Ende-zu-Ende- bzw. Peer-to-Peer Verbindung). Alle höheren Schichten sehen keinen Verbindungsrechner auf dem Übertragungsweg mehr. Die unteren beiden Schichten dienen dagegen lediglich der Kommunikation zwischen zwei benachbarten Knoten im Netzwerk. Der Transport von Ende zu Ende über Zwischenknoten hinweg wird auf dieser Schicht drei durchgeführt.

Der Datentransport über mehrere Teilstrecken kann verbindungslos oder verbindungsorientiert erfolgen. Bei der verbindungslosen Kommunikation (z.B. IP Protokoll) werden Pakete wie bei der Briefpost einzeln durch das Netz gesendet. Man spricht hierbei von *Datagram Service*. Jedes Paket enthält die vollständige Zieladresse, was einerseits den Overhead erhöht, andererseits aber die Übertragungsstrecke nur für die Zeit der Übertragung eines Pakets in Anspruch nimmt. Die in diese Klasse fallenden Dienste sind prinzipiell unzuverlässig, d.h. Verlust, Verfälschung und Änderung der Reihenfolge der Pakete muss von höheren Schichten behandelt werden. Es wird auch keine Fehlererkennung durchgeführt [REVS].

#### **4 Transportschicht**

Auf der Transportschicht findet man prinzipiell eine zuverlässige, verbindungsorientierte Kommunikation mit Fehlersicherung von Endpunkt zu Endpunkt. Die Kommunikation auf dieser Schicht ist verbindungsunabhängig, da das Routing von Schicht 3 abgenommen wird. Die Dienste der Schicht 4 sind bereits unabhängig von der Netzwerktechnik. Zu den Aufgaben der Dienste gehören [REVS]:

- Namensgebung für beide Endpunkte
- Adressierung der Teilnehmer
- Fehlerbehandlung
- Multiplexing von verschiedenen Datenströmen auf dem Kanal
- Synchronisation der Endpunkte
- bei Fehler in tieferen Schichten Wiederaufbau der Verbindung
- Internetworking: Protokollumsetzung zwischen unterschiedlichen Netzen durch einen Gateway-Rechner (s.u.)

#### **5 Sitzungsschicht**

Die Sitzungsschicht dient ebenfalls dem Datentransfer. Es werden Sitzungen auf- und abgebaut, so dass mehrere Prozesse auf das logische Transportsystem zugreifen können. Genauso wie die beiden höchsten Schichten, ist diese meist im Betriebssystem enthalten [REVS].

#### **6 Darstellungsschicht**

Die Darstellungsschicht beinhaltet Durchreichtedienste zwischen der Anwendungs- und der Sitzungsschicht. Zu den Aufgaben gehören Datenkonvertierung (z.B. ASCII ↔ EBCDIC, float ↔ integer), die Darstellung der Daten auf dem Bildschirm und möglicherweise Verschlüsselung und Komprimierung [REVS].

#### **7 Anwendungsschicht**

Die oberste Schicht enthält Funktionen mit denen der Anwender auf das Kommunikationssystem zugreifen kann. Vielfach bekannte Dienste sind *E-Mail*, *ftp (File Transfer)* und virtuelle Terminals [REVS].

Jede Schicht kommuniziert, logisch betrachtet mit der gleichen Schicht des Kommunikationspartners. (man spricht hier von *virtueller Peer-to-Peer Verbindung*). Die zu verschickenden Datenblöcke enthalten in jeder Schicht einen zusätzlichen Nachrichtenkopf (*Header*), der den Protokollablauf auf der entsprechenden Schicht beim Empfänger steuert. Daten und Header werden zusammen Paket genannt. Die Sicherungsschicht fügt neben dem Header noch einen Trailer hinzu. Man spricht auf dieser Schicht meist von Rahmen (*Frame*) statt von Paket [REVS].

## 2.3 Kommunikationsmodelle in der Praxis

Das OSI-Basisreferenzmodell, wie es oben vorgestellt wurde, bildet lediglich einen Rahmen mit Vorgaben für tatsächliche Protokolle auf den verschiedenen Ebenen. Das Referenzmodell definiert selbst keine Dienste. Auf welchen Ebenen des Referenzmodells die verschiedenen Protokolle, die wir in der Praxis finden, anzusiedeln sind soll mit Abbildung 2.2 gezeigt werden [REVS].

OSI	ISO 7498 CCITT X.200	TCP/IP	Novell Netware	IBM NETBIOS	Microsoft LAN	DEC DECNET
Anwendungsschicht	-FTAM (File Transp Acc and Mgmt) -JTM (Job Transfer and Manipulation) -VTP (Virtual Terminal Pcl) -CCITT X.400	-NFS(Network File Service)  -Telnet	-Anwenderprogramm  -MS-DOS	-Anwenderprogramm	-Anwenderprogramm	-Anwenderprogramm
Darstellungsschicht	-ISO 8822 -ISO 8823	-FTP (File Transfer Protocol)	-OS/2	-MS-DOS -IBM Svr Msg Block (SMB)	-Microsoft LAN Manager	
Sitzungsschicht	-ISO 8326 -ISO 8327		-Netware Shell -NetBIOS Emulation	-NetBIOS	-NetBIOS	-Sitzung
Transportschicht	-ISO 8072 -ISO 8073	-TCP -UDP	-SPX (Sequenced Packet Exg)	PC LAN Support Program	-TCP	-Netzwerk und Transport
Vermittlungsschicht	-ISO 8473 -CCITT X.25	-IP	-IPX (Internetwork Packet Exg)		-IP	
Sicherheitsschicht	-CCITT X.25	-IEEE 802.2/ ISO 8802				
		-IEEE 802.3 (CSMA/CD)				
Bitübertragungsschicht	-Ethernet -CCITT X.21					

**Abbildung 2.2:** Kommunikationsmodelle im OSI-Basisreferenzmodell [REVS]

Die Protokolle der TCP/IP Spalte sind die Grundlagen des Internet und der Kommunikation unter UNIX. Diese Protokolle sind älter als das OSI-Referenzmodell und passen nicht ganz so gut in die 7 Schichten. Die Protokolle, welche in der Simulation berücksichtigt wurden, sind in der Tabelle besonders hervorgehoben. Im folgenden werden zunächst die unteren Protokolle des TCP/IP Modells erläutert, bis schließlich die Transportschicht mit dem UDP Protokoll behandelt wird. In der Sicherungsschicht geht es vor allem um die Thematik der Arbitrierung, der Fehlererkennung und um das Verpacken in einzelne Datenrahmen [REVS].

## 2.4 IEEE 802

Der IEEE 802-Standard deckt die untersten drei OSI-Schichten ab. Auf der untersten Schicht enthält die Norm Standards für Zugriffsprotokolle auf verschiedene physikalische Medien, wie beispielsweise einen CSMA/CD basierten Bus. Schicht 2 wird in 2 Teilschichten für die Medienzugriffskontrolle (*MAC – Medium Access Control*) und die Verbindungskontrolle (*LLC – Logical Link Control*) aufgeteilt. Abbildung 2.3 zeigt die Zuordnung der IEEE 802-Protokolle zum OSI-Basisreferenzmodell.

Vermittlungsschicht	Netzwerkverwaltung	802.1 (High Level Interface) (IP)			
Sicherungsschicht	Logische Verbindungssteuerung	802.2 (LLC: Logical Link Control)			
	Medium-Zugriffssteuerung	MAC :Medium Access Control			
		802.3	802.4	802.5	802.6
		Zugriff auf physikalisches Medium			
Bitübertragungsschicht	elektronischer und mechanischer Anschluß	802.3 CSMA/CD (Ethernet)	802.4 Token-Bus	802.5 Token-Ring	802.6 MAN

**Abbildung 2.3:** Einordnung der IEEE 802-Protokolle in das OSI-Referenzmodell [REVS]

Auch hier wird wieder deutlich hervorgehoben, mit welchen Protokollen sich die Simulation auseinandersetzt [REVS].

IEEE 802.3 wurde 1983 veröffentlicht. Die Norm wurde an verschiedene Übertragungsmedien, d.h. Kabeltypen angepasst. Während man noch vor einigen Jahren hauptsächlich mit Übertragungsraten von 10 MBit/s gearbeitet hat, findet man heutzutage vor allem Medien mit Übertragungsraten von 100 MBit/s. Auch schnellere Medien sind bereits entwickelt worden [REVS].

Die Übertragungsmedien unterscheiden sich außerdem durch den Signaltyp, die maximale Segmentlänge, die Anzahl der Stationen pro Segment sowie durch Kabel und Steckerart und durch die angewendete Topologie. Unabhängig vom verwendeten Medium beschreibt die Norm eine Basisbandübertragung mit CSMA/CD-Arbitrierung. Auf das Thema der Arbitrierung soll im folgenden Abschnitt genauer eingegangen werden.

## 2.5 Busarbitrierung

Der Begriff „Busarbitrierung“ beschreibt das Verfahren, nach dem Anforderungen an den Bus kontrolliert werden und wie anschließend die Zuteilung des Busses erfolgt.

Die Vergabe der Buskontrolle kann auf verschiedene Arten aufgeführt werden. Sie lässt sich auf zwei Arten klassifizieren [REVS]:

- zentral ↔ dezentral
- kontrolliert ↔ zufällig

bei der IEEE 802.3 Norm wird ausschließlich eine dezentrale, zufällige Busarbitrierung verwendet.

**dezentral:** Die Busteilnehmer müssen sich selbst mittels geschickter Verfahren darum kümmern wer die Buskontrolle erhält.

**zufällig:** Die Vergabe des Busses wird nicht nach festgelegten Regeln kontrolliert

Folgende Protokolle entsprechen dieser Klassifikation:

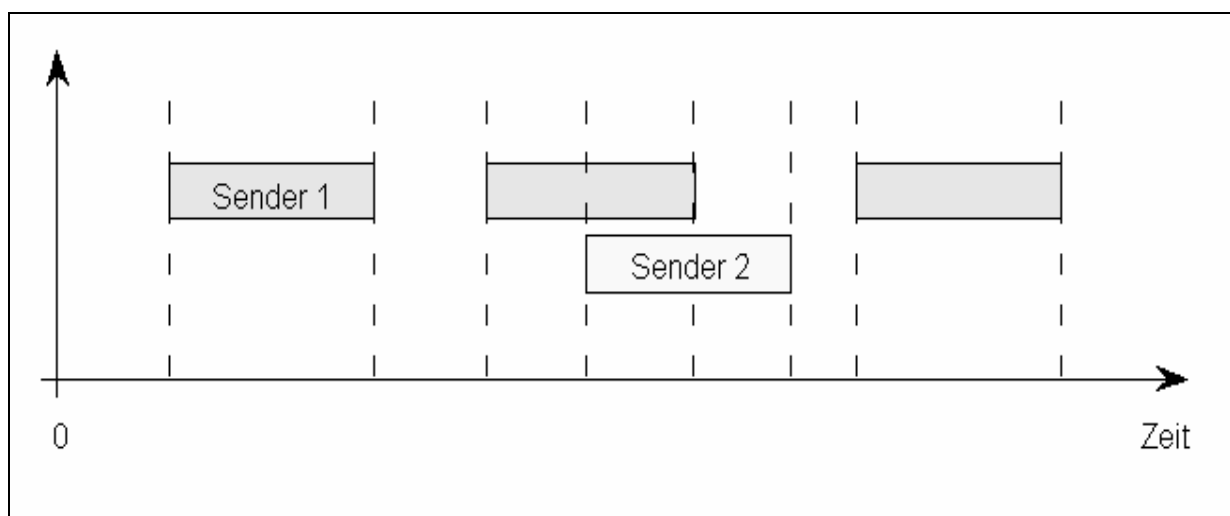
-Aloha

-CSMA/CD, CSMA/CA

### Aloha:

Das erste Medienzugriffsverfahren nach zufälligen Strategien, entstand 1970 auf Hawaii und wurde primäre in einem funkbasierten Forschungsnetz zur Verbindung von Inseln eingesetzt. *Reines Aloha* zeichnet sich dadurch aus, dass alle sendebereiten Teilnehmer sofort senden, ohne eine Erlaubnis einzuholen und ohne Überprüfung ob das Medium zur Zeit belegt ist. Es ist offensichtlich, dass dieses primitive Verfahren Probleme aufwirft, sobald zwei Teilnehmer gleichzeitig senden. Durch Überlagerung der Signale werden diese verfälscht und von den Empfängern nicht mehr erkannt. Um eine korrekte Informationsübertragung zu gewährleisten, muss eine Kollisionserkennung vorhanden sein [REVS].

Bei den Aloha-Verfahren wird die Konflikterkennung durch Versenden von Quittungssignalen gelöst. Nach korrektem Empfang einer Nachricht, was durch Fehlerprüfverfahren erkannt werden kann, sendet der Empfänger ein Quittungssignal zum Sender zurück. Bleibt diese Quittung aus, erkennt der Sender, dass die Übertragung gestört wurde und sendet sein Datenpaket noch einmal. Dies wird solange wiederholt bis eine Quittung vorliegt [REVS].



**Abbildung 2.4:** Reines Aloha

In Abbildung 2.4 sieht man, dass sowohl Sender 1 als auch Sender 2 erst ein vollständiges Datenpaket übertragen müssen, bevor sie erkennen dass es zu einer Kollision gekommen ist. Mit statistischen Methoden lässt sich leicht zeigen, dass die Fehlerrate, bedingt durch Kollision, schnell mit der Belegung des Übertragungskanals zunimmt. Die Berechnungen zeigen, dass das Verfahren nur dann eingesetzt werden kann, wenn ein Kanal weniger als 20% ausgelastet ist [REVS].

Das einfache Verfahren ist nie aus dem Forschungsstadium herausgekommen, da durch leichten Mehraufwand wesentlich bessere Verfahren möglich sind. Durch sehr einfache Erweiterungen lässt sich die Effizienz des Verfahrens erheblich steigern. *Slotted-Aloha* teilt beispielsweise die Zeit in Zeitscheiben (engl. Slots) ein, wobei nur zu Beginn einer Zeitscheibe versucht werden darf eine Übertragung zu starten. die Kanalkapazität erhöht sich auf etwa 35 % [REVS].

Die am weitesten verbreiteten Nachfolgeprotokolle sind die CSMA-Verfahren, die den Kanal dahingehend abhören, ob er zur Zeit belegt ist.

### CSMA:

CSMA steht für *Carrier Sense Multiple Access*. Das Busarbitrierungsverfahren ist eine Erweiterung von Aloha. Jede sendewillige Station hört den Kanal ab, bevor sie selbst – bei freiem Kanal – mit der Übertragung beginnt. Da zum Zeitpunkt des Freiwerdens des Kanals weiterhin Kollisionen auftreten können, müssen diese auch bei CSMA behandelt werden. Hierzu gibt es zwei Implementierungen [REVS]:

- CSMA/CD (collision detect): nur Kollisionserkennung
- CSMA/CA (collision avoidance): Kollisionsvermeidung

Für beide Unterklassen gibt es leicht unterschiedliche Ansätze [REVS]:

- *Nicht-persistente Verfahren*  
Bei diesem Verfahren wird, wenn der Kanal besetzt ist eine zufällige Zeitspanne gewartet, bevor der Sendeversuch mit dem Abhören des Kanals von vorne beginnt. Die Kollisionswahrscheinlichkeit wird dadurch vermindert. Ein Nachteil besteht allerdings darin, dass der Kanal während der Wartezeit ungenutzt ist.
- *p-persistente Verfahren*  
Wird bei diesem Verfahren der Kanal bei einem Sendewunsch frei, wird nur mit der Wahrscheinlichkeit  $p \leq 1$ , sofort gesendet. Es wird mit der Wahrscheinlichkeit  $1-p$  für eine Zeitspanne  $t$  gewartet, bevor der Teilnehmer erneut mit dem Senden beginnt.  $t$  ist so gewählt, dass ein Bit genügend Zeit hat den Kanal zu durchlaufen. Der sendewillige Teilnehmer kann damit feststellen, ob ein zweiter Teilnehmer ebenfalls senden will. Belegt ein zweiter Teilnehmer während der Zeit  $t$  den Bus, wird erneut gewartet, bis der Kanal frei ist und dann mit dem Buszugriff von vorne begonnen.

Der Nachteil beider Verfahren ist, dass keine direkte Kollisionserkennung während der Arbitrierungsphase durchgeführt wird. Eine Kollision wird erst durch eine fehlende Quittung erkannt, d.h. nachdem ein ganzes Paket gesendet wurde. Dies wird bei CSMA/CD und erst recht bei CSMA/CA vermieden [REVS].



## CSMA/CD:

CSMA/CD (CD steht für Collision Detect) ist ein Verfahren zur direkten Erkennung von Kollisionen bereits während der Arbitrierungsphase. Nachdem der Übertragungskanal frei geworden ist, versucht ein sendewilliger Teilnehmer, durch Senden eines Datenpakets, den Bus zu belegen. Gleichzeitig liest der Sender den Kanal mit, und überprüft, ob das von ihm gesendete Signal mit dem gelesenen übereinstimmt. Sind beide Signale gleich, trat keine Störung bzw. Kollision auf, und der Teilnehmer erhält, nach Ablauf einer bestimmten Arbitrierungszeit, bis zum Ende der Übertragung seines Datenpakets den Bus, ohne weiter gestört zu werden [REVS].

Durch das gleichzeitige Lesen während des Aussendens des Arbitrierungssignals, kann der Teilnehmer sehr schnell erkennen, ob eine Kollision aufgetreten ist. Erkennt der Sender eine solche Kollision, so bricht er seine Übertragung ab und sendet ein Störsignal (*Jam-Signal*). Dieses Störsignal wird von allen anderen Teilnehmern, die ebenfalls begonnen haben zu senden, empfangen, worauf auch diese ihre Übertragung abbrechen. Nach einer Kollision beginnt der Teilnehmer nach einer zufällig gewählten Zeitspanne erneut mit dem Abhören. Ausgefeiltere Algorithmen sorgen hier dafür, dass sich das Zeitintervall bei jeder Kollision erhöht [REVS]. Abbildung 2.5 zeigt anschaulich den Ablauf des CSMA/CD Verfahrens. CSMA/CD ist hier 1-persistent da bei einem freien Kanal auf jeden Fall gesendet wird.

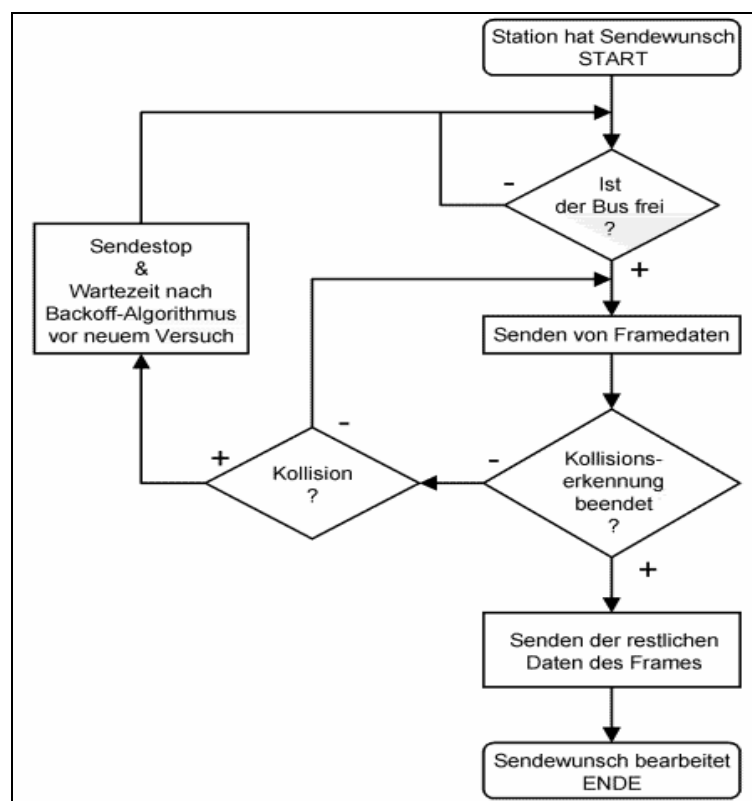


Abbildung 2.5: CSMA/CD Algorithmus [GRUPP]

Eine Kollision wird nach spätestens doppelter maximaler Signallaufzeit erkannt (Siehe Abbildung 2.6). Die Arbitrierungszeit und damit auch die minimale Paketlänge betragen  $2\tau$ . Damit  $\tau$  bekannt ist muss die maximale Leitungslänge festliegen [REVS].

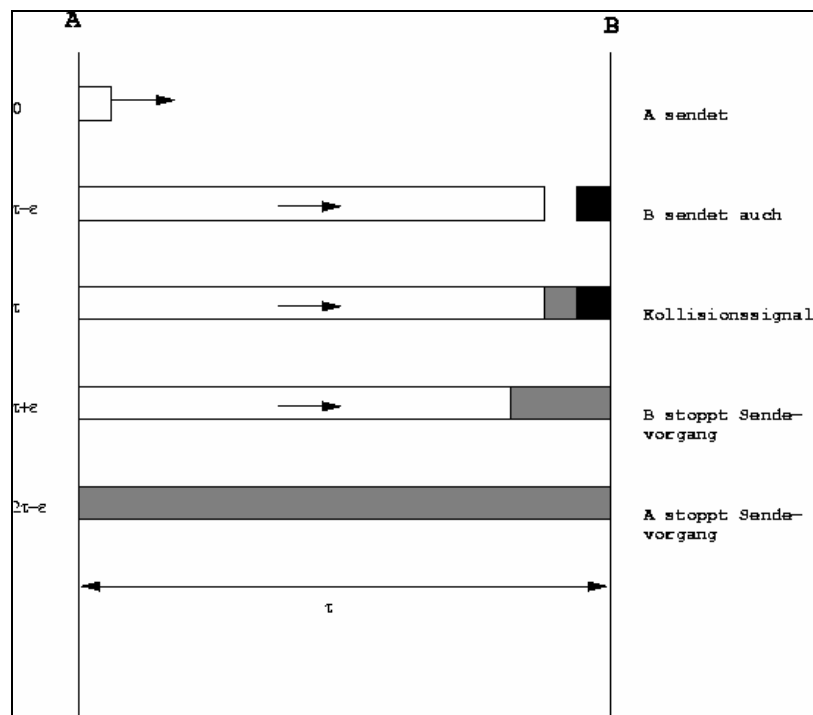


Abbildung 2.6: Kollisionserkennung bei CSMA/CD [KONE]

Die bekannteste Anwendung von CSMA/CD ist das *Ethernet* Protokoll der TCP/IP Protokollreihe. *Ethernet* verwendet 1-persistentes CSMA/CD. Das Kollisionserkennungsverfahren begrenzt auch bei *Ethernet* die maximale Buslänge [REVS].

### CSMA/CA:

CSMA/CA (CA steht für *Collision Avoidance*) erkennt nicht nur Kollisionen sondern vermeidet sie durch eine prioritätsgesteuerte Busvergabe. Eine begrenzte Paketlänge ermöglicht dem Teilnehmer mit der höchsten Priorität dabei ein Echtzeitverhalten. Der Bus kann allerdings für Teilnehmer niedriger Priorität blockiert werden, wenn der Teilnehmer mit höchster Priorität ständig senden würde [REVS].

### Vergleich:

Abbildung 2.7 vergleicht alle behandelten Arbitrierungsverfahren. Die hier gezeigten Kurven beruhen auf mathematischen Berechnungen. Es treten folgende Größen auf:

- $S$  = (mittlerer) Durchsatz pro Rahmen = Erwartungswert der erfolgreichen Paketübertragungen pro Zeiteinheit.
- $G$  = (mittlerer) Gesamtverkehr der Systems = Erwartungswert aller Paketübertragungen (neue Pakete und Wiederholungen) pro Zeiteinheit [REVS].

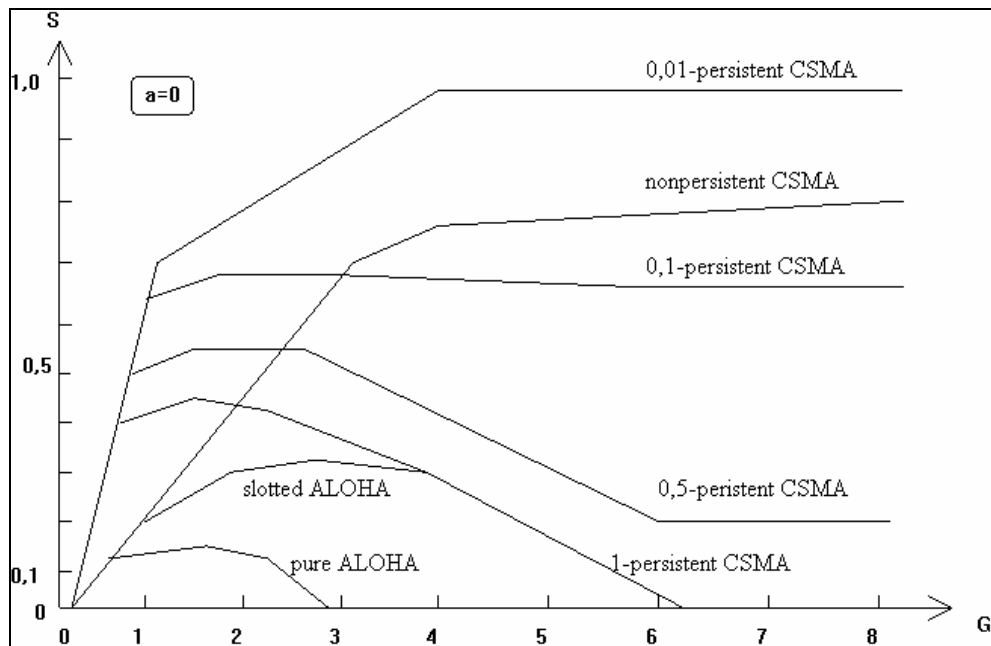


Abbildung 2.7: Arbitrierungsverfahren im Vergleich [RENE]

Man kann hier deutlich erkennen, das CSMA dem Aloha Verfahren bereits weit überlegen ist. Außerdem sieht man, dass bei einem hohen Gesamtverkehr im System ein kleiner Persistenz-Wert von Vorteil ist. Dieses Verfahren wird allerdings nicht vom Ethernet-Protokoll unterstützt [REVS].

## 2.6 Die MAC Schicht

Die MAC Schicht bildet den Zugriff auf unterschiedliche physikalische Medien nach „oben“ auf eine gemeinsame Schnittstelle zur LLC Schicht ab. Für die verschiedenen Arbitrierungsschemata gibt es MAC-Klassen mit wahlfreiem, verteilt gesteuertem und zentral gesteuertem Zugriff. Zu den Aufgaben der Protokolle auf der MAC-Schicht gehören [REVS]:

- Adressierung, wobei differenziert wird zwischen der Adressierung einer einzelnen Station, einer Gruppe von Stationen (Multicast) und Broadcast. Die MAC Adresse ist eine physikalische Adresse, die eine Station im Netzwerk eindeutig identifiziert.
- Erkennung des Rahmentyps durch Interpretation des Rahmen-Headers.
- Rahmenkontrolle durch Interpretation der Prüfsumme im Rahmen-Trailer.
- Kopieren von Rahmen: die adressierte Station kopiert die Daten des Rahmens vom Netzwerk in einen lokalen Puffer, um sie höheren Schichten bereitzustellen.

Als Beispiel für die Rahmenstruktur auf der MAC-Schicht soll das Rahmenformat von *Ethernet* beschrieben werden. Abbildung 2.8 stellt die Rahmenstruktur dar:

Bits						
64	48	48	16	variabel	variabel	32
Präambel	Zieladresse	Quelladresse	Typ	LLC-Header	Datenbereich	CRC

Abbildung 2.8: Rahmenstruktur aller Ethernet-Varianten [REVS]

Der Datenbereich kann bei *Ethernet* bis zu 1500 Byte betragen.

Verwandte Protokolle sind die Rahmenformate der 802.3 Norm, der Novell Netware und von SNAP. Diese in der Praxis am häufigsten anzutreffenden Rahmenformate sind sehr ähnlich, aber doch nicht identisch, was mehr oder weniger große Kompatibilitätsprobleme aufwirft und nur durch entsprechende Konfiguration der Netzwerktreiber zu lösen ist [REVS].

## 2.7 Die LLC Schicht

Die LLC Schicht sorgt für eine logische Verbindungssteuerung zwischen zwei benachbarten Stationen. Ein Datenpaket wird hierbei nicht über eine Vermittlungsstelle, d.h. eine dritte Station, gesendet. Die Vermittlungsstationen werden erst bei den Routing-Ansätzen auf Schicht 3 betrachtet [REVS].

Für den Datenaustausch zweier Stationen die direkt über eine Leitung miteinander verbunden sind, soll zunächst einschränkend angenommen werden, dass beide über eine Punkt-zu-Punkt-Verbindung direkt miteinander verbunden sind. Das Arbitrierungsproblem wird hierbei auf der darunterliegenden CSMA/CD Schicht gelöst [REVS].

Zur Datenübertragung zwischen zwei benachbarten Stationen wird ein *Leitungsprotokoll* (*Data Link Protocol*) benötigt. In der Vergangenheit wurden hierfür mehrere Protokolle mit unterschiedlichem Aufwand und Zuverlässigkeit entwickelt [REVS].

Zur Realisierung einer zuverlässigen Übertragung auf Schicht 2 gibt es in der Praxis drei Ansätze: fehlerkorrigierende Codes, Echoüberwachung und Wiederholungsanforderung, wovon das dritte Verfahren am weitesten verbreitet ist. Hierbei quittiert der Empfänger eine korrekt empfangene Nachricht durch ein Quittungssignal (*ACK*). Im Fehlerfall werden die Daten noch einmal gesendet. Dieses Verfahren ist so sicher wie das verwendete Fehlererkennungsverfahren [REVS].

## 2.8 Fehlererkennung

Prinzipiell muss man davon ausgehen, dass alle technischen Geräte und physikalischen Übertragungswege fehlerbehaftet sind. Beispiele für Störungen auf Übertragungswegen sind [REVS]:

- thermisches Rauschen in Halbleitern
- elektromagnetische Einstrahlungen (Übersprechen, Motoren, Zündanlagen, Blitze,...)
- radioaktive Einstrahlungen (Höhenstrahlung,...)

Oft treten solche Fehler gehäuft während eines Zeitabschnitts auf (Fehlerbündel, Error Bursts). Die Fehlerrate steigt im allgemeinen mit der Übertragungsgeschwindigkeit (Frequenz) an. Heutzutage muss man mit folgenden typischen Bitfehlerwahrscheinlichkeiten<sup>1</sup> rechnen [REVS]:

- verdrehte Zweidrahtleitungen in industrieller Umgebung: ca. 0.0001
- Telex (50 Baud): ca. 0.000001
- digitale Standleitung: ca. 0.000000000001

---

<sup>1</sup> Unter der Bitfehlerwahrscheinlichkeit versteht man die Wahrscheinlichkeit, dass ein einzelnes Bit falsch ist. Der Kehrwert davon ist demnach die Anzahl von Bits, die, statistisch gesehen, gesendet werden können, ohne dass ein Fehler auftritt.

Auch wenn diese Störungen nicht immer zu vermeiden sind, gibt es doch Maßnahmen zur Sicherstellung einer korrekten Datenübertragung. Hierzu kann man einerseits versuchen, durch technische Maßnahmen (Abschirmung, Potentialtrennung,...), die Bitfehlerwahrscheinlichkeit zu reduzieren, was allerdings noch keine korrekte Übertragung garantiert. Die Garantie<sup>2</sup> erlangt man heute durch zusätzliche Softwaremaßnahmen, wobei man eine Nachricht auf Fehler untersucht und im Fehlerfall auf geeignete Weise einschreitet. Datensicherung besteht immer aus 2 Schritten [REVS]:

- Fehlererkennung und
- Fehlerkorrektur

Bei TCP/IP wird die zyklische Redundanzprüfung CRC (Cyclic Redundancy Check) eingesetzt die bereits bei einer 16-Bit Prüfsumme folgende Bitfehler erkennt [REVS]:

- alle ungeraden Bitfehler für Blöcke bis 4095 Byte
- alle 2-Bit Fehler für Blöcke bis 3095 Byte
- alle Fehlerbündel bis 16 Bit Länge
- 99,997% aller längeren Fehlerbündel

Die Daten verdeutlichen, dass CRC nahezu alle Bitfehler mit einer annähernd 100%-igen Sicherheit erkennt.

## 2.9 TCP/IP

Oberhalb von Ethernet liegt in der Vermittlungsschicht IP (*Internet Protocol*). IP bildet zusammen mit TCP auch die Basis für das Internet. Folgende Abbildung zeigt die Zuordnung der TCP/IP Protokollfamilie zum OSI-Referenzmodell. Abbildung 2.9 zeigt auch die bei TCP/IP verwendeten Schichtennamen [REVS].

OSI-Schichten	TCP/IP-Schichten	TCP/IP-Protokolle			
Anwendungsschicht (Application Layer)	Anwendungsschicht (Application Layer)	Telnet	FTP	SMTP	NFS
Darstellungsschicht (Presentation Layer)					
Sitzungsschicht (Session Layer)					
Transportschicht (Transport Layer)	Transportschicht (Transport Layer)	Transmission Control Protocol (TCP)		User Datagram Protocol (UDP)	
Vermittlungsschicht (Network Layer)	internet	Internet Protocol (IP)			
Sicherungsschicht (Data Link Layer)	Netzwerkschicht (Network Layer)	Ethernet	Token-Ring	andere	
Bitübertragungsschicht (Physical Layer)					

**Abbildung 2.9:** TCP/IP Protokollfamilie [REVS]

<sup>2</sup> 100%-ige Garantie kann nicht gegeben werden. Allerdings lässt sich durch die heute eingesetzten Methoden die Restfehlerwahrscheinlichkeit so weit reduzieren, dass man dies als sicher ansieht.

Ebenfalls in der Transportschicht befindet sich das UDP Protokoll, welches sich grundlegend von TCP unterscheidet. Auf die orange hinterlegten Schichten soll im folgenden eingegangen werden.

### **Internet Protocol (IP):**

Das *Internet Protocol* bietet auf der Vermittlungsschicht eine verbindungslose, ungesicherte Übertragung von Datagrammen (Paketen). Die Sicherung der Datenübertragung muss daher auf den höheren Ebenen durchgeführt werden. Die Datagramme werden bei IP an direkt erreichbare Stationen gesendet. Im Internet werden die Datagramme auf der Sicherungsschicht in Rahmen verpackt, deren Größe bei Ethernet auf 1500 Byte beschränkt ist. Die Größe der IP-Datagramme auf der höheren Ebene ist auf eine Rahmengröße von 64 kByte beschränkt. Ist die Rahmengröße zu klein, so können Datagramme auch fragmentiert werden, wobei der Datagrammkopf in jeden Rahmen vollständig übernommen werden muss. Fragmentierte Datagramme können auf verschiedenen Wegen verschickt werden. Sie werden erst beim Empfänger wieder zusammengesetzt. Die Reihenfolge der Fragmente steht hierzu im Datagrammkopf zu Verfügung. Geht ein Fragment im Netzwerk verloren, gilt das gesamte Datagramm als verloren und muss vollständig neu gesendet werden [REVS].

Das *Internet Protocol* verwendet 32-Bit-Adressen, die alle Rechner im Internet eindeutig identifizieren. Da die IP-Adressen abhängig von den Subnetzen sind, muss ein Rechner beim Umhängen in ein neues Netz auch eine neue IP-Adresse erhalten. Er erhält dabei aber *keine* neue Hardware-Adresse. Die Hardware-Adressen auf der Sicherungsschicht identifizieren den Rechner weltweit eindeutig, unabhängig vom Netz dem sie angehören. Mit Hilfe des ARP (*Address Resolution Protocol*) kann ein Rechner über eine IP-Adresse, die Hardware-Adresse, die ihm möglicherweise unbekannt ist, ermitteln [REVS].

Abbildung 2.10 zeigt den Aufbau eines IP-Datagramms:

#### **IP-Paketformat**

Bits											
0...3	4...7	8...11	12...15	16	17	18	19	20..23	24..28	29..31	
Version	Kopflänge	Diensttyp			Datagrammlänge						
Identifikation				M	D			Fragmentabstand			
Lebenszeit		Transportprotokollnummer			Kopfprüfsumme						
Senderadresse											
Empfängeradresse											
Optionen										Füllbits	
Daten											
...											

**Abbildung 2.10:** IP-Datagrammstruktur [REVS]

## User Datagram-Protocol (UDP):

Einige TCP/IP-Anwendungen benötigen neben dem grundlegenden bestmöglichen Datagramm-Zustelldienst, der von IP zur Verfügung gestellt wird, noch weitere Dienste. Einige Anwendungen verwenden als Transportschichtprotokoll das User Datagram.Protocol (UDP). UDP sorgt dafür, dass eine Dateneinheit, die als *User-Datagram* bezeichnet wird, von einem Anwendungsprozess des Quell-Hosts zu einem Anwendungsprozess des Ziel Hosts geschickt wird. UDP fügt dem untergeordneten IP-Zustelldienst einen minimalen Zuverlässigkeitsmechanismus in Form einer optionalen Prüfsumme hinzu. UDP unternimmt jedoch keinen Versuch die, um die erfolgreiche Zustellung von User-Datagrammen sicherzustellen. Jedes User-Datagramm wird in einem einzelnen IP-Datagramm durch ein Internet befördert [TCPIP].

Abbildung 2.11 zeigt das Format eines UDP-Datagramms:

### UDP-Paketformat

Bits							
0...3	4...7	8...11	12...15	16...19	20...23	24...27	28...31
Sender-Port (optional)				Empfänger-Port			
Länge				UDP-Prüfsumme			
Daten							
...							

**Abbildung 2.11** : UDP-Paketformat [REVS]

Der von UDP gebotene Dienst ist ein verbindungsunabhängiger Datenzustelldienst, den man sich als Black Box vorstellen kann. Ein UDP-Anwender an einem Ende wirft ein User-Datagramm in die Black Box ein. Wenn dann nichts schief geht, kommt am anderen Ende der Black Box eine identische Kopie des User-Datagramms heraus und wird dort vom UDP-Anwender empfangen [TCPIP].

Der UDP-Dienst kann User-Datagramme verlieren, sie in einer anderen Reihenfolge zustellen als sie abgeschickt wurden, oder sie vervielfältigen. Da solche Fehler auftreten können, kann UDP nicht als zuverlässiger Dienst angesehen werden. Die Anwendung selbst muss für etwaige Zuverlässigkeitsprüfungen sorgen, wenn sie das UDP-Transportschichtprotokoll verwendet [TCPIP].

Anwendungen, die UDP einsetzen, übergeben Nachrichten einfach dem Internet, und die TCP/IP-Kommunikationssoftware muss ihnen nicht mitteilen, ob diese Nachrichten auch angekommen sind. Eine solche Anwendung kümmert sich entweder nicht darum, ob die Nachrichten empfangen wurden, oder sie verwendet eigene Routinen, um Endpunkt-Endpunkt-Prüfungen vorzunehmen. Alle verteilten Client-Server-Anwendungen, die ein einfaches Frage-Antwort Protokoll verwenden, kommen für UDP in Frage. Die Client-Anwendungskomponente stellt eine Frage und die Server-Anwendungskomponente sendet eine Antwort an sie zurück. Die Tatsache, dass der Server eine Antwort zurücksendet, ist schon eine Bestätigung dafür, dass er die Anfrage erhalten hat. Empfängt der Client innerhalb eines bestimmten Zeitraums keine Antwort, so kann der die Anfrage erneut senden [TCPIP].

Mehrere Anwendungsdienste verwenden bei der Erledigung ihrer Aufgaben UDP. Einige davon sind zusammen mit ihren UDP-Anschlußzuweisungen in Abbildung 2.12 aufgeführt.

Dienst	Port	Beschreibung des Dienstes
Echo	7	Meldet dem Absender ein empfangenes User-Datagramm
Discard	9	Löscht ein empfangenes User-Datagramm
Daytime	13	Gibt einen Wert für das Datum und die Uhrzeit zurück
Quote	17	Gibt eine Zeichenkette mit dem „Spruch des Tages“ zurück
Chargen	29	Gibt eine Zeichenkette mit zufällig gewählter Länge zurück
Nameserver	53	Nameserver-Prozess des Domain Name Service
Bootps	67	Server-Port zum Downloaden von Konfigurationsdaten
Bootpc	68	Client-Port zum Empfang von Konfigurationsdaten
TFTP	69	Server-Prozess des Trivial File Transfer Protokolls (TFTP)
SunRPC	111	Wird zur Implementierung Sun RPC-Dienstes verwendet

**Abbildung 2.12:** Anwendungsdienste die UDP verwenden [TCPIP]

Paketverluste entstehen entweder durch Übertragungsfehler oder durch einen Pufferüberlauf. Die einzige Fehlerprüfung unter UDP erfolgt über das Prüfsummenfeld. Stimmt der vom empfangenden Host errechnete Wert nicht mit der im User-Datagramm enthaltenen Prüfsumme überein, so wird das User-Datagramm vom empfangenden Host entfernt [TCPIP]. Ein Prozess, der für Datenübertragungen UDP einsetzt, reserviert normalerweise Platz in einem Pufferspeicher, um die am Port des Prozesses eintreffenden Datagramme in eine Warteschlange zu stellen. Im allgemeinen weiß ein Serverprozeß, der zur Kommunikation mit Clients UDP verwendet nicht, wie viele User-Datagramme ihm zu einer bestimmten Zeit geschickt werden. Treffen am Port des Servers mehr User-Datagramme ein, als vorgesehen, so werden sie entfernt [TCPIP].

Insgesamt ist UDP also ein einfaches, verbindungsunabhängiges und unzuverlässiges Übertragungsprotokoll auf der Transportschicht. In all diesen Punkten unterscheidet sich UDP von TCP.

### **Transmission Control Protocol (TCP):**

Während UDP für bestimmte Netzerkennungen hilfreich ist, benötigen viele andere Anwendungen eine Kommunikationssoftware, die einen zuverlässigen, geordneten Datenübertragungsdienst bereitstellen kann. In diesem Zusammenhang bedeutet *zuverlässig*, dass der Ziel-Prozess entweder alle abgeschickten Nachrichten oder einen Hinweis darüber empfängt, dass ein Fehler aufgetreten ist. Geordnet bedeutet, dass der Dienst dem Empfänger Pakete in der Reihenfolge zustellt, in der sie abgesendet wurden [TCPIP].

Ein zuverlässiger, geordneter Datenzustelldienst wird häufig als *verbindungsorientiert* bezeichnet. Bei einem solchen Datenübertragungsdienst wird zwischen dem Transportschicht-Prozess des Quell-Host und des Ziel-Host ein logischer Zusammenhang, eine sogenannte *Verbindung* hergestellt. Das Transportschichtenprotokoll führt dann alle Endpunkt-Endpunkt-Steuerungen durch, die zur Bereitstellung eines zuverlässigen, geordneten Datenübertragungsdienstes erforderlich sind [TCPIP].

Beim TCP-Datenübertragungsdienst handelt es sich um einen Vollduplex-Datenübertragungsdienst, bei dem Daten gleichzeitig in beiden Richtungen zwischen zwei miteinander in



Verbindung stehenden Prozessen hin- und herfließen können. TCP geht nicht davon aus, dass die zu übertragenden Daten eine bestimmte Struktur besitzen. Bytes, die an einem Ende in die Leitung gegeben werden, kommen am anderen Ende einfach in der gleichen Reihenfolge wieder heraus. Bei TCP kommt entweder eine identische Kopie des Bytestroms aus der Leitung, oder die Verbindung wird unterbrochen, und die TCP-Anwender werden über das Auftreten des Fehlers informiert [TCPIP].

Viele Anwendungen benötigen die Zuverlässigkeitsprüfungen eines verbindungsorientierten Datenübertragungsprotokolls wie TCP. Eine Anwendung zum Beispiel, die eine Dateiübertragung implementiert, ist für ein verbindungsorientiertes Protokoll bestens geeignet. Mehrere Anwendungsdienste verwenden zur Ausführung ihrer Funktionen TCP. Einige davon sind zusammen mit den ihnen zugewiesenen TCP Ports in Abbildung 2.13 aufgelistet [TCPIP]:

Dienst	Port	Beschreibung des Dienstes
Discard	9	Dienst zur Entfernung empfangener Anwenderdaten
Chargen	19	Dienst, der eine Zeichenkette mit zufällig gewählter Länge zurückgibt
FTP	20	Dienst zur Übertragung von FTP-Dateiübertragungen
FTP	21	Dienst zur Implementierung von FTP-Steuerrouinen
Telnet	23	Dienst zur Implementierung des Telnet-Dienstes zum Fern.Login
SMTP	25	Dienst zur Implementierung des SMTP-Dienstes für elektronische Post
X400	103	Dienst zur Implementierung des X400-Dienstes für elektronische Post

**Abbildung 2.13:** Anwendungsdienste die TCP einsetzen [TCPIP]

Das TCP Protokoll teilt den Bytestrom zur Übertragung in eine Reihe von Dateneinheiten auf, die *Segmente* genannt werden. Segmente werden zur Übertragung mit Hilfe der Dienste von IP in IP-Datagramme eingeschlossen. TCP selbst legt den zu übertragenden Daten keine Struktur auf, und die Segmentstruktur der Daten bleibt vor den beiden TCP-Anwendern verborgen, denen die Daten als kontinuierlicher Strom erscheinen [TCPIP].

Abbildung 2.14 zeigt das Format eines TCP-Segments:

#### TCP-Paketformat

TCP-Headerformat															
0...3		4...7		8...11		12...15		16...19		20...23		24...27		28...31	
Sender-Port								Empfänger-Port							
Sequenznummer															
Quittungsnummer															
Datenabstand	reserviert	U	A	P	R	S	F	Fenstergröße							
		R	C	S	S	Y	I								
		G	K	H	T	N	N								
TCP-Prüfsumme								Urgent-Zeiger							
Optionen												Füllbits			
Daten															
...															

**Abbildung 2.14:** TCP-Paketformat [REVS]

Um zu verstehen, wie TCP im Detail für eine sichere Übertragung sorgt, muss man sich mit dem Bestätigungsverfahren auseinandersetzen, welches verwendet wird um sicherzustellen, dass alle Segmente fehlerfrei und in der richtigen Reihenfolge zugestellt werden. Auch hier werden zur Fehlererkennung Prüfsummen eingesetzt [TCPIP].

Bestätigungen werden bei TCP auch verwendet, um Flusssteuerungsroutinen zu implementieren, die die relative Geschwindigkeit von Sender und Empfänger anpassen. Wird eine Verbindung aufgebaut, einigt man sich auf eine Fenstergröße zur Übertragung in jeder Richtung. Die Fenstergröße bestimmt die Anzahl der Bytes, die der Sender übertragen kann, ehe er eine Bestätigung empfängt. Mit einer Anpassung der Fenstergröße können Sender und Empfänger auf veränderte Bedingungen im Netzwerk reagieren [TCPIP].

TCP verfügt außerdem über eine Blockierungsregelung, die dafür sorgt die Netzlast zu begrenzen, Blockierungen zu verhindern oder diese abzubauen [TCPIP]. Alles in allem ist TCP also ein ausgefeiltes, verbindungsorientiertes und zuverlässiges Übertragungsprotokoll auf der Transportschicht. Abbildung 2.15 vergleicht noch einmal die Protokolle UDP und TCP :

UDP	TCP
unsichere Datenübertragung	zuverlässige Die Übertragung: TCP garantiert Zuverlässigkeit bzgl. Datenverfälschung und -verlust. Die Zuverlässigkeit ist für den Benutzer transparent
verbindungslos: im wesentlichen Abbildung des IP auf die Transportschicht, d.h. Übertragung einzelner Pakete	verbindungsorientiert: Aufbau und Abbau einer virtuellen Verbindung zwischen beiden Stationen vor und nach der Informationsübertragung stromorientiert: Bit-/Byte-Folgen kommen in gleicher Reihenfolge an, wie sie vom Sender abgeschickt wurden

**Abbildung 2.15:** Vergleich von TCP und UDP [REVS]

## 2.10. Kommunikationsgeräte

Will man nun komplexere Netzwerke aufbauen, so muss man sich neben den verschiedenen Protokollschichten auch mit der Hardware beschäftigen, die zum Aufbau von Netzwerken notwendig ist. Man benötigt Kommunikationsgeräte und Verbindungsrechner, die zum Zusammenschluss einzelner Stationen oder ganzer Netzwerke dienen [REVS].

Die verschiedenen Gerätetypen unterscheiden sich in ihrer Einordnung im OSI-7-Schichtenmodell und damit in ihrer Funktionalität, Flexibilität und Geschwindigkeit. Am einen Ende finden wir reine elektrische Verstärker zur Signalregenerierung, denen am anderen Ende die sogenannten Gateways, die Teilnetzwerke mit ganz unterschiedlichen Protokollstapeln verbinden können, gegenüberstehen. Die wesentlichen Gerätearten zum Aufbau von Rechnernetzen sollen in diesem Abschnitt kurz beschrieben werden. Diese sind [REVS]:

Gerätetyp	auf OSI-Schicht
Repeater	1
Bridge	2
Router	3
Switch	2 oder 3
Gateway	bis zu allen sieben

**Abbildung 2.16:** Kommunikationsgeräte [REVS]

### **Repeater:**

Repeater sind die einfachsten Netzwerk-Verbindungskomponenten. Diese Geräte sind auf das verwendete Transportmedium zugeschnitten und für den Datenfluss, d.h. für die Protokolle der OSI-Schichten 2 bis 7 völlig transparent. Ein Repeater kann nur gleichartige Netze verbinden (z.B. *Ethernet*↔*Ethernet* oder *Token-Ring*↔*Token Ring*) [REVS].

### **Bridges:**

Bridges sind etwas intelligentere Geräte als Repeater. Sie können gleich- und verschiedenartige Netzwerke miteinander verbinden. Wie Repeater besitzen sie allerdings keine eigenen Netzwerk- oder sonstige Adressen und können somit nicht adressiert oder explizit angesprochen werden. Im Gegensatz zu Repeatern lesen und interpretieren Bridges die Hardware-Adressen (MAC Adressen). Bridges reduzieren den Gesamttraffic indem sie die Frames nur an die Kabellegmente weiterleiten, an denen sich die Empfangsstation befindet [REVS].

### **Router:**

Router bearbeiten die untersten 3 Protokollschichten. Die Kopplung von Teilnetzen erfolgt auf der Vermittlungsschicht. Während Bridges auf der Schicht 2 die physikalischen Geräteadressen zur Unterscheidung von lokalen und nicht-lokalen Rahmen interpretieren, betrachten Router die Adressen der Vermittlungsschicht (*IP*-Adressen) um Datenpakete gezielt auf günstigen Wegen durch das Netzwerk zu leiten. Router sind selbst aktive Teilnehmer im Netz. Sie besitzen auch eigene Adressen und können gezielt angesprochen werden [REVS].

### **Gateways:**

Gateways nach ihrer ursprünglichen Definition kann man als erweiterte Router ansehen. Neben ihrer Eigenschaft als Router können Gateways jedoch die Daten zwischen Netzwerken mit unterschiedlichen Protokollen bzw. Protokollfamilien transportieren. Im Prinzip decken Gateways damit alle sieben Schichten des OSI-Schichtenmodells ab und führen eine Protokollumsetzung durch [REVS].

### **Switches:**

Die Basiskomponenten zum Aufbau komplexer Kommunikationsnetzwerke aus kleineren Teilnetzen sind die oben erläuterten Bridges, Router und Gateways. Je nach Gleich- bzw. Verschiedenartigkeit der zu verbindenden Teilnetze genügt es, diese auf OSI-Schicht 2 zu brücken oder es sind größere Protokollumsetzungen notwendig (siehe Gateways). Zur Wegfindung in komplexen Netzen dient die Router-Funktionalität.

Eine weitere Geräteklasse, die *Switches*, ändert diese Einteilung nicht. Es gibt Switches, die wie Bridges auf Schicht 2 arbeiten und es gibt Switches auf Schicht 3 als Konkurrenz zu

Routern. Der Unterschied zwischen Switches und anderen Gerätetypen liegt vor allem darin, zu welchen Netztopologien man die Stationen verschalten kann. In diesem Zusammenhang dienen Switches der sternförmigen Anbindung von Rechnern oder LAN-Segmenten an ein sogenanntes Backbone<sup>3</sup> [REVS].

### **Hubs:**

Hubs arbeiten genauso wie Repeater auf der untersten Schicht des ISO/OSI Modells. Sie dienen dazu einen Datenstrom auf mehrere Leitungen zu verteilen. Es findet hier eine sternförmige Anbindung auf der untersten Ebene statt. Hubs bilden so eine einfache Möglichkeit, mehrere Komponenten per Steckverbindung an ein Netzwerk anzuschließen.

## **2.11 Busse in der Automatisierungstechnik**

In diesem Abschnitt wird noch ein spezielles Anwendungsfeld für Busse betrachtet: der Steuerungs- und Automatisierungsbereich. Bedingt durch die Arbeitsumgebung, werden in diesem Bereich spezielle Anforderungen an die Busse gestellt, die nicht unbedingt mit denen in der allgemeinen Datenverarbeitung übereinstimmen. So kann beispielsweise bei der Steuerung einer kritischen Anlage die Robustheit bei der Übertragung, eine wesentlich stärkere Rolle spielen als die Übertragungsrate [REVS].

Aus Sicht der Kommunikation zerfällt die Automatisierung in 2 wesentliche Klassen mit zum Teil unterschiedlichen Anforderungen [REVS]:

- Steuerung von:
  - Fabriken, Kraftwerken und anderen Anlagen
  - Gebäuden
  - Fahrzeugen, Flugzeugen, etc.
  - ...
- Fertigung: CIM (Computer Integrated Manufacturing)

CIM ist momentan das Schlagwort in der rechnerintegrierten Produktion. In den letzten Jahren gab es mehrere Entwicklungsstufen im Automatisierungsbereich, angefangen von mikroprozessorgesteuerten Automaten, über Robotereinsatz, bis zu vollautomatisierten Fertigungsprozessoren bzw. -systemen. CIM wird in dieser Entwicklung häufig als oberste Stufe angesehen, wobei der Computer den gesamten Entwurfs- und Fertigungsprozeß steuert. Die extremste Vision wäre eine menschenleere Fabrik [REVS].

Die Prozesssteuerung, als zweiter wichtiger Aspekt in der Automatisierung, kann als Teil von CIM angesehen werden. Der gesamte CIM-Bereich wird heute hierarchisch unterteilt, wobei die Sensoren und Aktoren auf der untersten Ebene angesiedelt sind. Auch die Prozesssteuerung kann man in dieser Hierarchie im unteren Bereich ansiedeln. Am oberen Ende der Hierarchie finden wir die Leit-, Betriebs- und Planungsebenen [REVS].

Die unterschiedlichen Komponenten bzw. Ebenen des CIM haben selbstverständlich auch unterschiedliche Anforderungen an das Kommunikationssystem. Es muss wohl nicht näher begründet werden, dass die Kommunikation mit einem Roboter in der Fabrikhalle ganz andere Anforderungen besitzt als beispielsweise die Kopplung zweier Workstations im Planungsbüro. Da sich diese unterschiedlichsten Anforderungen wohl kaum durch ein

---

<sup>3</sup> Ein *Backbone* ist ein firmenweites schnelles Netz (Bus oder Ring), das die abteilungsinternen LANs miteinander verbindet.

Kommunikationsmedium und ein Kommunikationsprotokoll geeignet realisieren lassen, sind heutzutage alle Kommunikationsstrukturen im Fertigungsbereich hierarchisch strukturiert [REVS].

Den unterschiedlichen Anforderungen an das Kommunikationssystem wurde durch die Entwicklung ebenenspezifischer Bussysteme und Netzwerke Rechnung getragen. Die einzelnen anwendungsspezifischen Kommunikationsmedien sind zu einer hierarchischen Kommunikationsstruktur zusammengesetzt. Auf den höheren Leitebenen werden heute meist Standards wie Ethernet, FDDI und Token-Ring verwendet [REVS].

Die prinzipielle Struktur der digitalen Automatisierungssysteme auf den unteren Ebenen ist ein System bestehend aus Leitstationen, Prozessstationen und einem Kommunikationssystem, das diese Stationen verbindet. Letzteres kann als BUS, als hierarchisches Netzwerk mit Punkt-zu-Punkt Verbindungen oder als Kombination dieser beiden realisiert sein. Die Anforderungen an das Kommunikationssystem auf Feldebene sind durch die industrielle Umgebung begründet [REVS]:

- Umempfindlichkeit gegenüber Störungen und Beschädigungen (z.B. großer Temperaturbereich, Industrieluft, Meeresluft, mechanische Schwingungen, hohe Induktivitäten durch Motoren)
- Fehlertoleranz (z.B. durch redundante Auslegung)
- leichte und schnelle Wartbarkeit, Fehlerdiagnosemöglichkeit
- Buszugriffsverfahren muss meist Echtzeitverhalten garantieren (dies erfordert ein deterministisches Arbitrierungsverfahren)
- Möglichkeit zur ereignisorientierten Kommunikation durch Interruptmöglichkeiten
- auf Einsatzbereich zugeschnittene Übertragungsgeschwindigkeit
- Wirtschaftlichkeit (Busankopplung darf nicht mehr als ca. 10% bis 20% der Kosten des Automatisierungsgerätes betragen)

### **Feldbusse:**

Oberhalb der Sensor/Aktor-Ebene sind die Feldbusse angesiedelt, wobei sich die beiden Ebenen nicht immer genau abgrenzen lassen. Oft werden Sensor/Aktor-Busse zu den Feldbussen gerechnet. Neben den weiter oben angesprochenen allgemeinen Anforderungen an Busse in der Automatisierungstechnik wird für Feldbusse gefordert [REVS]:

- Die Ausdehnung liegt zwischen einigen Metern und einigen Kilometern
- das Bussystem sollte so flexibel sein, dass zusätzliche Busteilnehmer problemlos eingebracht werden können
- harte Zeitanforderungen mit garantierter maximaler Reaktionszeit des Systems, Echtzeitfähigkeit und Reaktionszeiten im Millisekunden bis Sekundenbereich (je nach Anwendung)
- aus wirtschaftlichen Gründen sind serielle Busse den Parallelbussen vorzuziehen
- da Bussysteme von Natur aus störanfälliger sind als die bisher verwendete sternförmige Verkabelung, sind Maßnahmen notwendig, um die Ausfallwahrscheinlichkeit zu reduzieren und die Zuverlässigkeit zu erhöhen (z.B. entsprechende Kodierung, Fehlererkennung, störunempfindliches Übertragungsmedium wie etwa Lichtwellenleiter).

Aus der Gruppe der Feldbusse sollen vier Beispiele genannt werden. Diese sind [REVS]:

- PROFIBUS
- CAN
- LON und
- EIP

Im folgenden soll auf den *PROFIBUS* etwas genauer eingegangen werden

### **PROFIBUS (Process Field Bus):**

Der *PROFIBUS* wurde herstellerübergreifend von 14 Herstellern und 5 wissenschaftlichen Instituten entwickelt und ist in Deutschland eine nationale Feldbusnorm (DIN E 19245). Es handelt sich dabei um einen Multi-Master-Bus mit *Token-Passing*-Zugriffsverfahren. Die physikalische Netzstruktur ist eine Linienstruktur [REVS].

Das Forschungsprojekt *ISILEIT* (s.a. [ISILEIT]) arbeitet an einem integrierten Entwurf, sowie an der Analyse und der Validierung verteilter Fertigungssysteme. Die verteilten Fertigungssysteme kommunizieren dabei mit Hilfe des *PROFIBUS*.

Als Übertragungsmedium ist für den PROFIBUS entweder Lichtwellenleiter oder Shielded – Twisted-Pair vorgesehen, wobei im letzten Fall sie Bussegmente passiv abgeschlossen werden müssen. Abbildung 2.17 zeigt die maximale Länge in Abhängigkeit von der Übertragungsgeschwindigkeit [REVS].

Übertragungsgeschwindigkeit [kBit/s]	max. Länge [m]
9,6 / 19,2 / 93,75	1.200
187,5	600
500	200

**Abbildung 2.17:** Leitungslängen beim PROFIBUS [REVS]

An der Busarbitrierung nach dem Token-Passing-Verfahren nehmen alle aktiven Teilnehmer (z.B. SPS-Steuerung) teil. Nach Erhalt des Busses kann der aktuelle BUS-Master mit beliebigen passiven Teilnehmern (z.B. Sensoren/Aktoren) kommunizieren. Der Bus-Master kann während eines Zyklus ein- oder mehrmals einen Datenaustausch durchführen bzw. initiieren, wobei die Gesamtkommunikationsdauer von verschiedenen zeitlichen Randbedingungen abhängt. Hierzu wird die als Parameter vorgegebenen Token-Soll-Umlaufzeit mit der gemessenen, tatsächlichen Umlaufzeit verglichen. Die Busbelegungszeit richtet sich nach den verbleibenden Zeitreserven. Jeder Master darf jedoch zumindest eine hochpriorie Nachricht absenden. Weitere „normale“ Nachrichten sind dagegen nur erlaubt, wenn die Token-Soll-Umlaufzeit noch nicht überschritten ist. Durch dieses Verhalten ist ein gut vorhersagbares Echtzeitverhalten möglich [REVS].

Die Übertragung beim PROFIBUS erfolgt zeichenorientiert. Ein Zeichen besteht aus 11 Bit (→UART<sup>1</sup>-Zeichen), die sich aus 8 Daten- und 3 Steuerbits zusammensetzen. Aus mehreren Zeichen werden Telegramme gebildet wobei es hier mehrere Telegrammformate gibt.

---

<sup>1</sup> UART = Universal Asynchronous Receiver-Transmitter – s. serielle Schnittstelle

Die OSI-Schichten 3 bis 7 sind beim PROFIBUS nicht vollständig ausgeprägt. Direkt auf der Sicherungsschicht, auf der die Telegrammformate definiert sind, sitzt die Anwendungsschicht die man in weitere Schichten unterteilen kann [REVS].

Da jedes Zeichen bereits einen 3-Bit-Overhead hat und die Telegramme mehr oder weniger viele Steuerzeichen besitzen entsteht beim PROFIBUS insgesamt ein erheblicher Overhead. Aufgrund des geringen Datendurchsatzes sind zeitkritische Anwendungen nicht möglich [REVS].

## 3 Technische Realisierung der Simulation

### 3.1 UDP mit Java

Dieser Abschnitt soll zunächst erklären, wie UDP in Java behandelt wird und welche Möglichkeiten Java in Bezug auf UDP bietet. Zum einen soll hier dargestellt werden, wie UDP in Java funktioniert, da ein großer Teil der Simulation auf dem UDP-Protokoll basiert. Zum anderen tragen die Erläuterungen des realen Einsatzes von UDP aber auch dazu bei, den Ansatz der Simulation von UDP besser zu verstehen.

Da sich UDP für viele Anwendungsfälle eignet, ist auch Java in der Lage auf dieses Protokoll zuzugreifen. Für sämtliche Netzanwendungen ist in Java das package *java.net* verantwortlich, das sowohl UDP als auch TCP unterstützt. Zum Versenden von UDP Paketen werden in Java Sockets definiert, über die sich der UDP-Transfer konfigurieren lässt. Erzeugt wird ein Socket mit Hilfe der Klasse *DatagramSocket* aus dem package *java.net*. Dabei gibt es folgende Konstruktoren:

#### Konstruktoren der Klasse *DatagramSocket*

Constructor Summary	
<b><i>DatagramSocket</i></b> ()	Constructs a datagram socket and binds it to any available port on the local host machine.
<b><i>DatagramSocket</i></b> (int port)	Constructs a datagram socket and binds it to the specified port on the local host machine.
<b><i>DatagramSocket</i></b> (int port, <i>InetAddress</i> laddr)	Creates a datagram socket, bound to the specified local address.

Abbildung 3.1: Konstruktoren der Klasse *DatagramSocket*

Man hat wie man sehen kann zunächst die Möglichkeit einen Port festzulegen. Bei einem Port handelt sich um einen allgemeinen Anschluss an das Netzwerk, wobei der Rechner auf dem die Anwendung residiert keine Rolle spielt. Dieser Port bezieht sich auf eine spezielle Instanz von *DatagramSocket*, die von einem laufenden Prozess genutzt wird. Einem Rechner im Netzwerk kann mal also mehrere Ports zuweisen, wobei schon ein einzelner Prozess mehr als einen Port besitzen kann. Legt man den Port nicht explizit fest, so wird einfach ein Port, der gerade zur Verfügung steht, verwendet. Zusätzlich zu dem Port kann bei Bedarf eine URL vom Typ *InetAddress* definiert werden, unter welcher sich der Socket ansprechen lässt. Entfällt die Definition einer URL, so verwendet der Socket automatisch die URL des lokalen Rechners im Netzwerk. Der Socket ist also logisch betrachtet ein Anschluss an das Netz, um Datenpakete an das Netz zu übertragen und Datenpakete zu empfangen.

Abbildung 3.2 zeigt die Methoden mit denen man auf ein Objekt der Klasse *DatagramSocket* zugreifen kann. Für den eigentlichen Transfer sorgen dabei die Methoden *send(DatagramPacket p)* und *receive(DatagramPacket p)*. *send(DatagramPacket p)* versendet ein Paket, welches mit Hilfe der Klasse *DatagramPacket* definiert wird. Analog zum eigentlichen UDP Protokoll enthält hierbei das Paket die Ziel-Adresse und den Ziel-Port. Bei *receive(DatagramPacket p)* wird ein empfangenes Paket an ein Objekt der Klasse *DatagramPacket* übergeben. Während *send(DatagramPacket p)* in der Lage ist, einfach laufend Pakete zu versenden, ohne dass diese irgendwo ankommen müssen, ist *receive-*



(*DatagramPacket p*) darauf angewiesen, etwas zu empfangen, da ansonsten der aufrufende Prozess blockiert wird. Diese Tatsache lässt sich jedoch mit der Methode

## Methoden der Klasse *DatagramSocket*

Method Summary	
void	<b><u>close()</u></b> Closes this datagram socket.
void	<b><u>connect(InetAddress address, int port)</u></b> Connects the socket to a remote address for this socket.
void	<b><u>disconnect()</u></b> Disconnects the socket.
InetAddress	<b><u>getInetAddress()</u></b> Returns the address to which this socket is connected.
InetAddress	<b><u>getLocalAddress()</u></b> Gets the local address to which the socket is bound.
int	<b><u>getLocalPort()</u></b> Returns the port number on the local host to which this socket is bound.
int	<b><u>getPort()</u></b> Returns the port for this socket.
int	<b><u>getReceiveBufferSize()</u></b> Get value of the SO_RCVBUF option for this socket, that is the buffer size used by the platform for input on the this Socket.
int	<b><u>getSendBufferSize()</u></b> Get value of the SO_SNDBUF option for this socket, that is the buffer size used by the platform for output on the this Socket.
int	<b><u>getSoTimeout()</u></b> Retrive setting for SO_TIMEOUT.
void	<b><u>receive(DatagramPacket p)</u></b> Receives a datagram packet from this socket.
void	<b><u>send(DatagramPacket p)</u></b> Sends a datagram packet from this socket.
void	<b><u>setReceiveBufferSize(int size)</u></b> Sets the SO_RCVBUF option to the specified value for this DatagramSocket.
void	<b><u>setSendBufferSize(int size)</u></b> Sets the SO_SNDBUF option to the specified value for this DatagramSocket.
void	<b><u>setSoTimeout(int timeout)</u></b> Enable/disable SO_TIMEOUT with the specified timeout, in milliseconds.

**Abbildung 3.2:** Methoden der Klasse *DatagramSocket*

*setSoTimeout(int timeout)* umgehen, die dafür sorgt, dass die Methode *receive(DatagramPacket p)* nur für eine vordefinierte Zeit wartet.

Die Zuverlässigkeit von UDP hängt entscheidend von der Größe der verwendeten Puffers ab. In Java hat man die Möglichkeit, mit Hilfe der Methoden *setReceiveBufferSize(int size)* und *setSendBufferSize(int size)* sowohl den Empfangspuffer als auch den Sendepuffer in Bytes festzulegen. Man kann so einerseits für eine zuverlässigere Übertragung sorgen und andererseits vermeiden, dass der Speicher des Rechners nutzlos verwaltet wird. Die maximale Puffergröße ist hierbei auf die maximale Größe eines UDP-Paketes beschränkt. Da UDP Pakete in IP Pakete eingeschlossen werden, liegt dieser Wert bei etwa 64 kByte. Die übrigen Methoden der Klasse *DatagramSocket* dienen zur Benutzung von Remote-Adressen und sollen hier nicht genauer betrachtet werden.

Ebenfalls entscheidend für die UDP-Übertragung in Java ist die Klasse *DatagramPacket*. Hier sollen nur die Konstruktoren gezeigt werden, um den Aufbau eines Objektes dieser Klasse zu erklären. Wie man leicht erkennt, enthält ein Objekt der Klasse *DatagramPacket* die drei folgenden Attribute:

- Einen Port der dem Prozess zugewiesen wird
- Einen Daten-Array für die eigentlichen Nutzdaten
- Eine vordefinierte Länge

### Konstruktoren der Klasse *DatagramPacket*

Constructor Summary	
<b>DatagramPacket</b> (byte[] buf, int length)	Constructs a DatagramPacket for receiving packets of length length.
<b>DatagramPacket</b> (byte[] buf, int length, <a href="#">InetAddress</a> address, int port)	Constructs a datagram packet for sending packets of length length to the specified port number on the specified host.
<b>DatagramPacket</b> (byte[] buf, int offset, int length)	Constructs a DatagramPacket for receiving packets of length length, specifying an offset into the buffer.
<b>DatagramPacket</b> (byte[] buf, int offset, int length, <a href="#">InetAddress</a> address, int port)	Constructs a datagram packet for sending packets of length length with offset ioffsetto the specified port number on the specified host.

**Abbildung 3.3:** Konstruktoren der Klasse *DatagramPacket*

Da die Instanz von *DatagramSocket* den Port des Senders kennt, lässt sich mit diesen Angaben bereits ein UDP-Paket definieren. Die URL, welche in Java mit einem Objekt der Klasse *InetAddress* definiert wird, ist erst bei der Erzeugung des IP-Datenpakets von Bedeutung. Java gewährt hier einen Zugriff auf die Vermittlungsschicht. Die Klasse *DatagramPacket* wird ebenfalls bei der Realisierung der Simulation eingesetzt.

## 3.2 Aufruf der Simulation

Der letzte Abschnitt hat gezeigt, wie Java mit UDP umgeht und welche Möglichkeiten Java dabei bietet. Mit den oben gezeigten Klassen kann auf ein real existierendes Netzwerk zugegriffen werden. Es stellt sich nun also die Frage, wie man Daten, die normalerweise über ein reales Netz versendet werden, an ein virtuelles Netz schicken kann, welches das reale Netz simuliert. Eine wichtige Anforderung an diesen Ansatz war es, existierenden Anwendungen die per UDP kommunizieren, eine Möglichkeit zu geben, ihre Daten alternativ über ein simuliertes UDP-Netzwerk zu verschicken. Die Verbindung dieser Anwendungen, mit der Netzwerksimulation sollte sich dabei einfach realisieren lassen, damit eine aufwendige Neuentwicklung dieser Anwendungen entfällt. Ebenfalls sollte jederzeit die Möglichkeit bestehen sowohl über das reale Netzwerk, als auch über die Simulation zu kommunizieren, um den Einsatz der Simulation möglichst flexibel zu gestalten. Diese Anforderungen beziehen sich auf die Anwendung der Simulation und sagen noch nichts über den eigentlichen Kern der Simulation aus, der in den weiteren Abschnitten erläutert wird.

Da Java die UDP-Kommunikation mit Hilfe von Sockets realisiert, werden auch in der Simulation Sockets zum Verschicken und Empfangen von Objekten des Typs *DatagramPacket* verwendet. Hierfür sorgt in der Simulation die Klasse *CommonSocket*, welche einen Socket definiert der wahlweise über ein reales Netzwerk oder über ein virtuelles Netzwerk kommunizieren kann. Die Klasse *CommonSocket* befindet sich, wie alle anderen Klassen, die von der Simulation benötigt werden, im Package *simnet*. Abbildung 3.4 zeigt, wie man durch einfache Änderungen im bestehenden Programm-Code auf die Netzwerksimulation zugreifen kann:

```
import java.io.*;
import java.net.*;
import simnet.*;

public class VideoServer1 {

    static CommonSocket socket;
    static long sleeptime=5000;

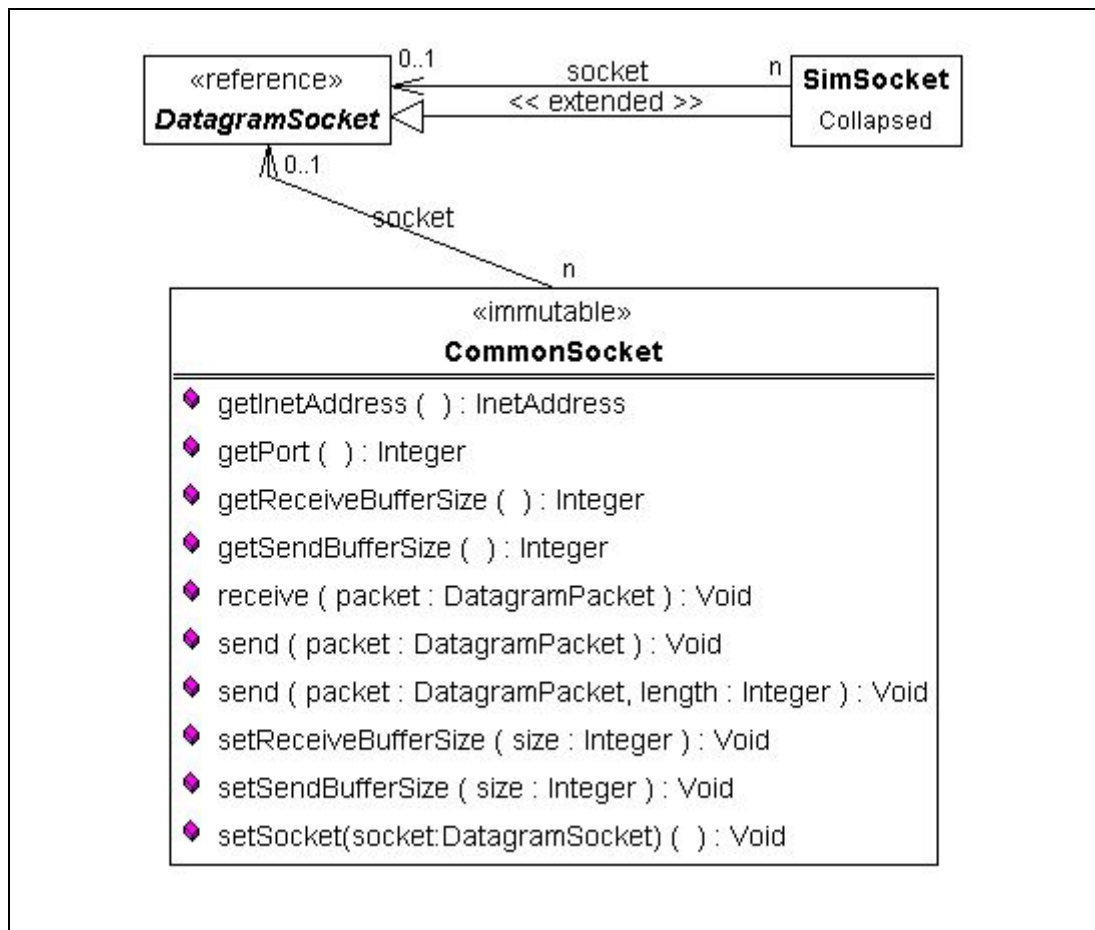
    public static void main(String args[]) throws IOException, InterruptedException {

        socket=new CommonSocket();
    }
}
```

**Abbildung 3.4:** Einsatz des package *simnet*

Während man in realen UDP-Anwendungen ein Objekt der Klasse *DatagramSocket* definiert, wird hier ein Objekt der Klasse *CommonSocket* definiert. Die Anwendung kommuniziert nun über einen allgemeinen Socket und die eigentliche Realisierung des Verschickens bleibt transparent.

Ein Design-Pattern, welches sich für diese Anwendung hervorragend eignet, ist das *Strategy-Pattern*. Bei einem *Strategy-Pattern* müssen mehrere Klassen existieren, welche die gleiche Spezifikation aufweisen und sich nur im Detail unterscheiden. Es existieren somit verschiedene Strategien zum Lösen des gleichen Problems. Um den Wechsel zwischen den verschiedenen Strategien zu ermöglichen, bilden alle Klassen die eingesetzt werden können, eine Vererbungshierarchie und dienen somit alle zur Erzeugung von Objekten desselben Typs. Abbildung 3.5 verdeutlicht den Aufbau des *Strategy-Patterns*.



**Abbildung 3.5:** Strategy Pattern für *CommonSocket*

Die Klasse *CommonSocket* dient hier als Wrapper-Klasse für ein Objekt vom Typ *DatagramSocket*. Die Klasse *SimSocket* erbt von der Klasse *DatagramSocket*. Für ein Objekt vom Typ *CommonSocket* lässt sich mit Hilfe der Methode *setSocket(DatagramSocket socket)* festlegen, welche Art von Socket, bzw. welche Art der Übertragung verwendet werden soll. Objekte vom Typ *SimSocket* dienen hierbei zur Kommunikation über die Netzwerksimulation. Die übrigen Methoden entsprechen den Methoden der Klasse *DatagramSocket* und lassen sich sowohl für Objekte der Klasse *DatagramSocket* als auch für Objekte der Klasse *SimSocket* aufrufen.

Will man auf das virtuelle Netzwerk zugreifen, so weist man dem Objekt der Klasse *CommonSocket* ein Objekt der Klasse *SimSocket* zu:

```

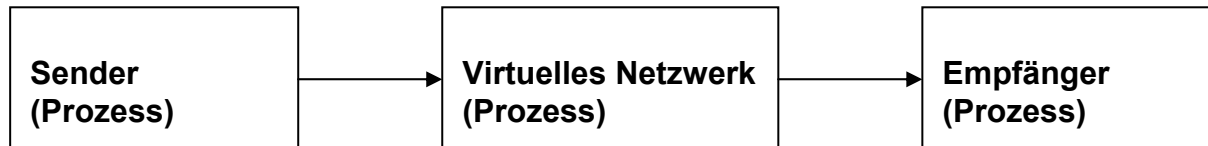
socket=new CommonSocket();
socket.setSocket(new SimSocket("VideoServer1"));
  
```

**Abbildung 3.6:** Einsatz der Klasse *SimSocket*

Die Arbeitsweise von Objekten des Typs *SimSocket* dient zu eigentlichen Realisierung der Simulation und soll nun im folgenden Abschnitt erläutert werden.

### 3.3 Simulationsansatz

Nachdem erklärt wurde, wie man Daten an die Netzwerksimulation übergeben kann, soll nun verdeutlicht werden, was mit diesen Daten passiert. In der Struktur der Programmierung herrscht hier eine klare Trennung zwischen den Applikationen, die Daten verschicken bzw. empfangen können, und dem Netzwerk, welches die Daten weiterleitet. Sämtliche Anwendungen sowie das Netzwerk selbst bilden hierbei einzelne Prozesse, welche normalerweise auf einem Rechner realisiert werden. Es ergibt sich so folgendes stark abstrahiertes Schema:



**Abbildung 3.7:** Grobes Schema für das virtuelle Netzwerk

Selbstverständlich ist die Anzahl der Teilnehmer am virtuellen Netzwerk grundsätzlich nicht begrenzt. Außerdem kann eine Anwendung sowohl Sender als auch Empfänger sein. Um von vornherein den Rechenaufwand der Simulation zu reduzieren, ist es sinnvoller, anstatt ganzer Datenpakete nur die Länge des zu verschickenden Datenpakets an das virtuelle Netzwerk zu übermitteln. Die Länge eines Datenpakets ist im Grunde die einzige Größe eines Pakets, welche Einfluss auf die Dauer der Übertragung hat. Das Schema lässt sich damit folgendermaßen verfeinern:



**Abbildung 3.8:** Genauer Schema für das virtuelle Netzwerk

Der Sender übergibt zunächst die Länge des Datenpakets an das virtuelle Netzwerk, welches die Aufgabe hat den Zeitpunkt zu bestimmen, zu dem die Übertragung vollständig abgeschlossen ist. Erst wenn die Simulation meldet, dass alle Daten übertragen wurden, wird das reale Paket versendet. Für die Realisierung dieses Ansatzes sorgt die Klasse *SimSocket* die, wie schon im vorherigen Abschnitt erwähnt wurde, anstelle der Klasse *DatagramSocket* verwendet wird. Wird ein Objekt der Klasse *SimSocket* erzeugt, so meldet sich dieses Objekt beim virtuellen Netzwerk an und kann gegebenenfalls relevante Eigenschaften des virtuellen Netzwerkes abfragen. Neben der Methode *send(DatagramPacket packet)* zum Verschicken von Datenpaketen, bei denen die Länge des Datenpakets implizit durch seine Länge bestimmt wird existiert in der Klasse *SimSocket* außerdem die Methode *send(DatagramPacket packet, int length)*, bei der die Länge des Datenpaketes explizit festgelegt werden kann. Das Datenpaket kann so flexibel genutzt werden. Man kann so das Übertragungsverhalten in Abhängigkeit vom Datenvolumen untersuchen, ohne einen Inhalt für die Daten festzulegen, der keinen Einfluss auf das Übertragungsverhalten hat.

Abbildung 3.9 zeigt die Einordnung der Klasse *SimSocket* in das Klassendiagramm. Um mit dem virtuellen Netzwerk zu kommunizieren, verwendet ein Objekt der Klasse *SimSocket* das UDP-Protokoll, was man an den Assoziationen mit den Klassen *DatagramSocket* und *DatagramPacket* erkennen kann. Die Klasse *Buffer* simuliert den Übertragungspuffer, auf den die jeweilige Anwendung zugreifen kann. Definiert man

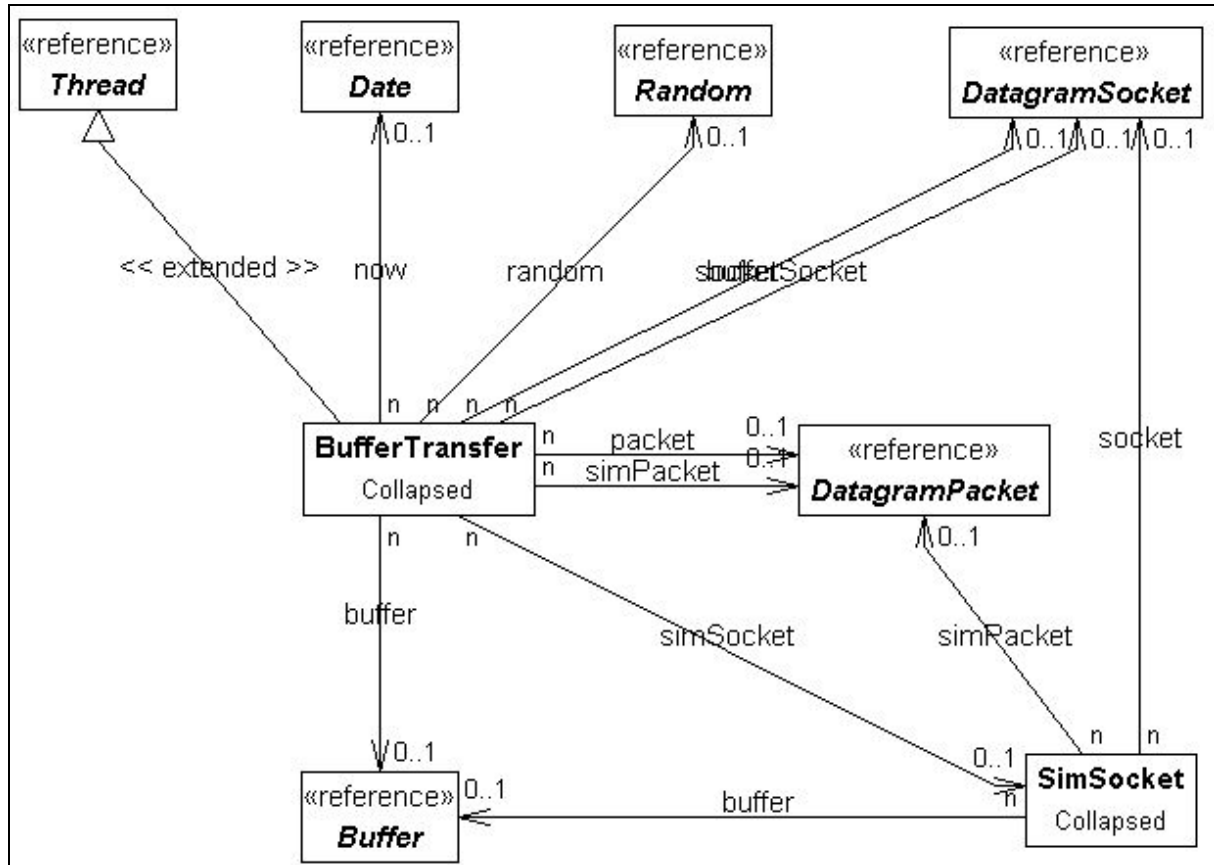


Abbildung 3.9: Die Klassen *SimSocket*, *Buffer* und *BufferTransfer*

nach dem oben aufgeführten Übertragungsschema den Sender als einen einzelnen Prozess, so arbeitet die Simulation genaugenommen unrealistisch. Nicht nur der Empfänger sondern auch der Sender müssten in diesem Fall warten, bis die Übertragung abgeschlossen ist, was natürlich nicht der realen Arbeitsweise von UDP entspricht. Um diesen Umstand zu vermeiden, übergibt die Klasse *SimSocket* das Datenpaket an die Klasse *BufferTransfer*, welche als *Thread* realisiert ist. Der Anwendung, die auf *SimSocket* zugreift, wird somit ein unmittelbares Fortfahren ermöglicht. Da es sich bei dem Pufferspeicher um eine begrenzte Ressource handelt und eventuell mehrere Objekte vom Typ *BufferTransfer* auf diese Ressource zugreifen, ist eine Synchronisation der einzelnen Prozesse notwendig. Für diese Synchronisation sorgt die Klasse *Buffer* die als Monitor dient und so die Belegung des Puffers kontrolliert. Ist ein neu erzeugtes Datenpaket zu groß für den Sendepuffer, so wird der Verlust dieses Paketes bestätigt.

Um das Konzept noch offener zu gestalten, besteht für ein Objekt der Klasse *BufferTransfer* die Möglichkeit, die zu verschickenden Daten mit Bitfehlern zu versehen. So lassen sich später die Auswirkungen von schlechten Übertragungen und gegebenenfalls fehlenden Fehler-erkennungsverfahren beobachten. Da diese Einstellung im Prinzip nicht von der Anwendung selbst, sondern von der Übertragung abhängt, kann die Fehlererkennung im virtuellen Netzwerk selbst ein- bzw. ausgeschaltet werden. Der nötige Parameter wird übergeben, wenn ein Objekt der Klasse *SimSocket* erzeugt wird und sich beim virtuellen Netzwerk anmeldet. Dieses Konzept zeigt, wie eine enge Bindung zwischen Socket und

Netzwerk erzeugt werden kann, um Berechnungen, die nicht im virtuellen Netzwerk ausgeführt werden können, dennoch von dort aus zu steuern. An dieser Stelle sei erwähnt, dass bei einer TCP/IP Übertragung normalerweise immer das CRC Verfahren eingesetzt wird. Es können so die Auswirkungen einer nicht vorhandenen Fehlererkennung beobachtet werden.

Insgesamt ergibt sich also ein transparentes Konzept, bei dem Daten in ein virtuelles Netzwerk umgelenkt werden können. Da die UDP Kommunikation mit Hilfe von Java auch auf einem einzelnen Rechner realisierbar ist können dabei alle Prozesse lokal ausgeführt werden. Im folgenden soll nun dargestellt werden, wie der Prozess für das virtuelle Netzwerk aufgebaut ist.

### 3.4 Hauptschleife der Simulation

Nach dem bereits erörterten Übertragungsschema wird ein Datenpaket, welches die Länge der tatsächlich zu übertragenden Daten enthält an die Simulation übergeben. Im einfachsten Fall simuliert das Netzwerk die Übertragung des Paketes und bestätigt anschließend die vollständige Übertragung. In der Praxis entsteht jedoch häufig der Fall, dass mehrere Anwendungen gleichzeitig auf das virtuelle Netz zugreifen. Bei der Planung der Implementierung gab es zunächst zwei verschiedene Ansätze:

Ein **inkrementeller Ansatz** beruht im wesentlichen auf einer Schleife, bei der in jedem Durchlauf eine Zähler um eine Zeiteinheit weitergezählt wird. In jedem Durchlauf der Schleife haben die einzelnen Übertragungsprozesse die Möglichkeit einen Teil ihrer Bytes zu übertragen. Jeder Prozess enthält also einen Zähler, der bei jedem Durchlauf für den Fall, dass der Prozess das Übertragungsmedium für sich beansprucht, weitergezählt wird. Gleichzeitig muss in jedem Durchlauf der Schleife die Möglichkeit bestehen, neue Datenpakete zu empfangen, aus denen neue Übertragungsprozesse erzeugt werden. Damit die Simulation realistisch erscheint, muss das Zeitinkrement sehr klein gewählt werden. Die Folge davon ist, dass die Schleife eine extrem hohe Anzahl von Durchläufen pro Zeiteinheit erzeugt. Zusätzlich müssen in jedem Durchlauf der Schleife alle Prozesse, die gerade Daten übertragen, berücksichtigt werden. Es entsteht so ein großer Rechenaufwand, der dazu führt, dass die in der Simulation gezählte Zeit stark von der realen Zeit abweicht. Es kann außerdem schnell passieren, dass Datenpakete nicht entgegengenommen werden, da ein Großteil der Zeit für die Rechenarbeit der Simulation verwendet wird. Der inkrementelle Ansatz verspricht für die Simulation eines Netzwerkes wenig Erfolg und wurde aus diesem Grund verworfen.

Bessere Möglichkeiten bietet hier ein **Scheduling-Ansatz**, der für eine effiziente Verwaltung mehrerer Prozesse sorgt. Der Scheduling-Ansatz wandelt ein anliegendes Übertragungsvolumen ebenfalls in einen Prozess um, der solange in der Simulation verbleibt, bis die Übertragung abgeschlossen ist. Der Rechenaufwand ist hier deutlich geringer, da die Simulation nur in Konfliktsituationen einen neuen Ablauf der Übertragungen festlegen muss. Es wird also zunächst berechnet, wann ein relevantes Ereignis wie ein Konflikt oder das Ende einer Übertragung, eintritt. Danach muss die Simulation nur noch „warten“, bis dieser Zeitpunkt erreicht wird. Es entsteht so deutlich weniger Rechenaufwand, so dass das Verhalten der Simulation auch bei mehreren Teilnehmern stabil bleibt.

Den Kern der Simulation bildet in der Implementierung die Klasse *Wire* (siehe Abbildung 3.10). Da die Klasse nur ein Kabel bzw. Netzsegment simuliert, ein komplexes Netzwerk dagegen sich aus mehreren Segmenten zusammensetzt, erbt die Klasse *Wire* von der Klasse *Thread*. So wird der Einsatz mehrerer Segmente, die gemeinsam verwaltet werden können, prinzipiell ermöglicht. Auch bei der Klasse *Wire* zeigen die Assoziationen zu den Klassen *DatagramSocket* und *DatagramPacket*, dass *Wire* das UDP Protokoll zur Kommunikation mit dem Socket einsetzt. Bei der Instanziierung von *SimSocket* findet die erste Kommunikation



mit der Klasse *Wire* statt, wobei die Klasse *Wire* gegebenenfalls den Namen des Teilnehmers empfängt. Dieses Konzept enthält die Möglichkeit, dem Netz Informationen über seine Teilnehmer zukommen zu lassen, was später zu einer anschaulicheren Darstellung der Simulation führen kann. *Wire* sendet daraufhin den Parameter für das CRC Verfahren an den Socket, da hier eventuell die Simulation einer nicht vorhandenen Fehlererkennung stattfindet. Im weiteren Verlauf empfängt die Instanz der Klasse *Wire* nun jeweils einen Wert, der die Länge eines zu übertragenden Pakets enthält. Mit einem Objekt der Klasse *Date* wird hier der Zeitpunkt für den Übertragungsanfang gemessen. Als nächstes wird ein Objekt vom Typ *Task* erzeugt, welches den Übertragungsprozess darstellt. Dieses Objekt wird an eine Instanz von *Scheduler* übergeben, der den neuen Prozess in den Ablauf der Simulation einbindet. Hat eine Instanz von *Task* den Zustand einer abgeschlossenen Übertragung erreicht, so sendet sie eine Bestätigung an ihren Socket und existiert daraufhin nicht mehr.

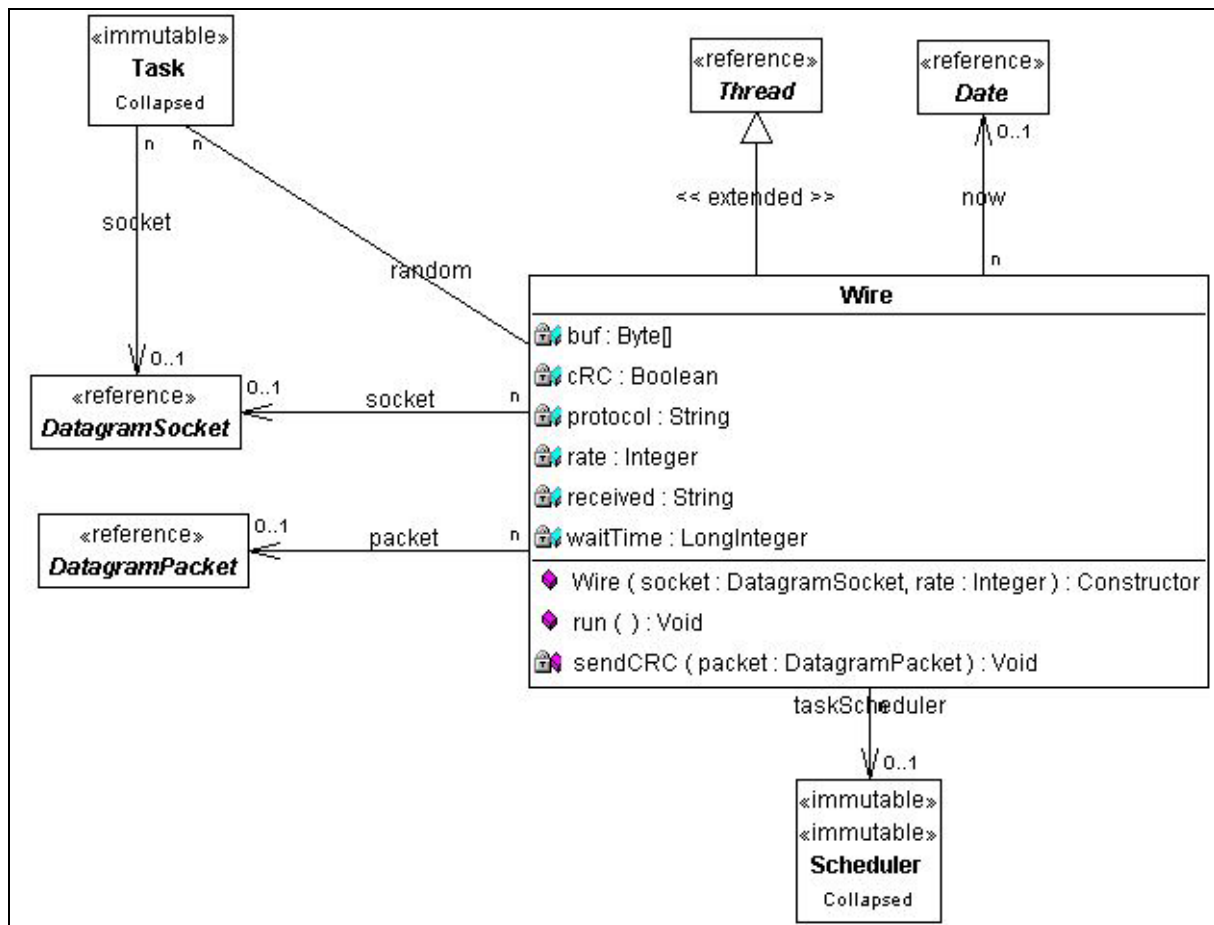


Abbildung 3.10: Die Klasse *Wire*

Bei der Umsetzung des Konzepts entstehen folgende Schwierigkeiten: da jederzeit neue Pakete eintreffen können, muss die Klasse *Wire* praktisch immer empfangsbereit sein. Verwendet man die Methode *receive()*, so ist der Prozess blockiert und kann sich nicht mehr um das Versenden bzw. Verwalten von Instanzen der Klasse *Task* kümmern. Solange keiner der Teilnehmer mehr sendet, bleiben diese Übertragungen hängen. Ein scheinbarer Ausweg besteht darin, zunächst die Prozesse im Scheduler abzuarbeiten, bis alle Übertragungen abgeschlossen sind. Dieses würde allerdings zu gravierenden Fehlern führen, da so während der Übertragung keine Daten mehr empfangen werden können. Einen Ausweg liefert die Methode *setSoTimeout()*, mit deren Hilfe man das Warten auf den nächsten Empfang begrenzen kann. Die entscheidende Frage ist hier: Wie lange soll gewartet werden? Während hier lange Wartezeiten zu Verzögerungen des Übertragungsverhaltens führen, entsteht bei



kurzen Wartezeiten eine hohe Frequenz beim Durchlaufen der Hauptschleife, so dass das Prinzip wieder dem inkrementellen Ansatz entspricht. Sinnvoller ist es hier die, Wartezeit so zu bemessen, dass genau solange gewartet wird, bis z.B. eine Übertragung abgeschlossen ist. Bei einem Scheduling-Ansatz lässt sich diese Information leicht abfragen. Einzelheiten hierzu befinden sich in Abschnitt 3.7.

Eine einwandfreie Übertragung der Daten ist somit gewährleistet. Die Rate, mit der die Bytes übertragen werden, lässt sich über das Attribut *rate* vom Typ Integer festlegen. Außerdem kann man festlegen, ob die Fehlererkennung CRC eingesetzt wird und mit welcher Wahrscheinlichkeit Bitfehler auftreten. Wie der Scheduler mit Instanzen vom Typ *Task* weiter verfährt, wird in Abschnitt 3.6 erläutert. Zunächst soll aber gezeigt werden, wie sich ein Objekt der Klasse *Task* zusammensetzt.

### 3.5 Übertragungsprozesse

Ein Übertragungsprozess entspricht in der Simulation einem Objekt vom Typ *Task*. Die wichtigsten Attribute für diesen Objekttyp bilden zunächst die Länge des Pakets, der Ankunftszeitpunkt und der Socket, von dem die Daten gesendet wurden. Ein Task sorgt nach dem ISO/OSI Modell für die Simulation der Übertragungsschritte auf Schicht 3 und 4. Dabei werden folgende Klassen verwendet:

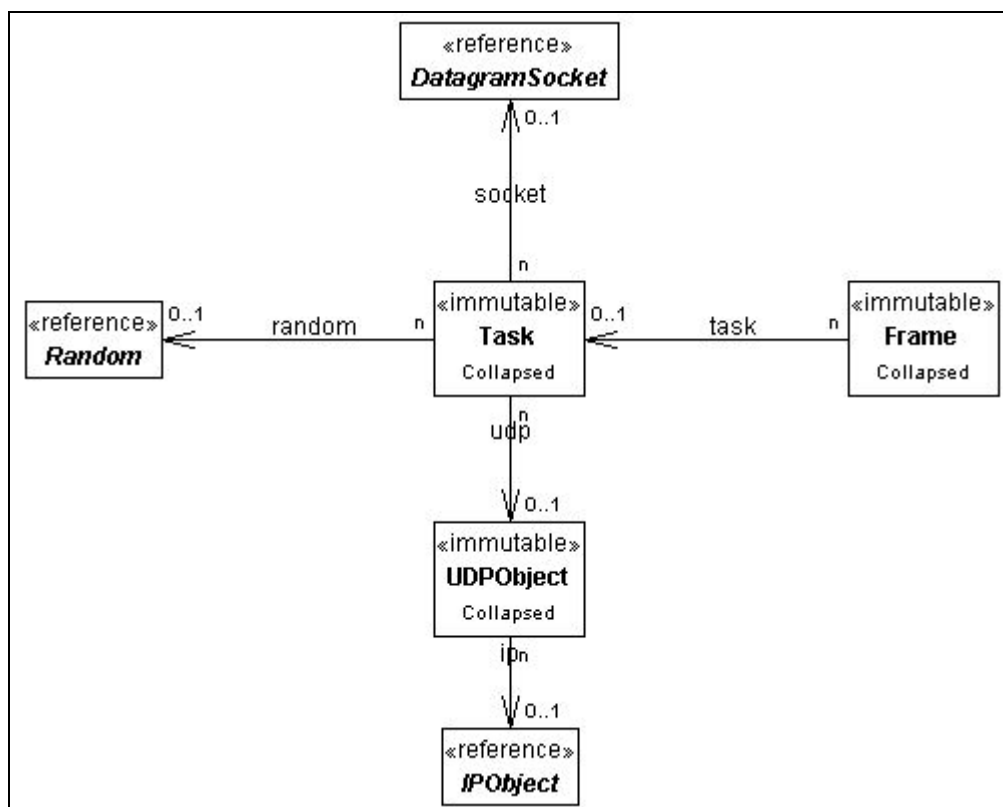


Abbildung 3.11: Die Klasse *Task*

Der Ankunftszeitpunkt einer Übertragung kann in Java nur in Millisekunden gemessen werden. Da ein realistisches Übertragungsverhalten besser im Mikrosekundenbereich erzeugt werden kann, wird zunächst eine zufällige Anzahl von Mikrosekunden auf den Ankunftszeitpunkt aufaddiert, wobei dafür gesorgt wird, dass der Wert des Ankunftszeitpunktes sich nur erhöhen kann. Da die Simulation prinzipiell immer ein nichtdeterministisches Verhalten

aufweist, wird das Ergebnis hierdurch nicht verfälscht. Da die Übertragung einer Instanz von *Task* im Verlauf der Simulation durch Konflikte unterbrochen werden kann, speichert die Instanz in diesem Fall den Zeitpunkt, zu dem die nächste Arbitrierung stattfinden soll. Ein Übertragungsprozess kann schließlich am Ende der Übertragung den Abschlusszeitpunkt bestimmen und so die benötigte Übertragungszeit berechnen.

Jede Instanz von *Task* erzeugt eine Instanz vom Typ *UDPObject*. Es wird hier der Overhead errechnet, der durch das Einfügen des UDP-Headers entsteht. Entsprechend dem UDP Format wird hier auch der Port gespeichert, wobei es sich hier allerdings nicht um den Port des Empfängers, sondern um den Port des Senders handelt, da der Sender später über das Ende der Übertragung informiert werden muss. Eine Instanz vom Typ *UDPObject* erzeugt daraufhin eine Instanz vom Typ *IPObject*. Die *IPObject* Instanz speichert die URL des Senders und berechnet die Anzahl der MAC-Frames. Wir befinden uns hier also auf Schicht 2 der ISO/OSI Modells. Fragmente vom Typ *Frame* beziehen sich immer auf einen Task, was sich im Folgenden als sinnvoll erweisen wird.

### 3.6 Einsatz des Schedulers

Der Scheduler sorgt dafür, dass mehrere Übertragungsprozesse in der richtigen zeitlichen Abfolge auf das Medium zugreifen. Ein Scheduler ist zunächst einmal unnötig, da die Übertragungsprozesse nacheinander eintreffen und somit auch in derselben Reihenfolge mit der Arbitrierung beginnen. Der Scheduler übergibt die Übertragungsprozesse in dieser Reihenfolge an eine Arbitrierungsverfahren vom Typ *CommonArbitration*, welches den eigentlichen Zugriff auf das Medium darstellt. Dabei besteht ein enger Zusammenhang zwischen dem Scheduler und der Arbitrierungsklasse.

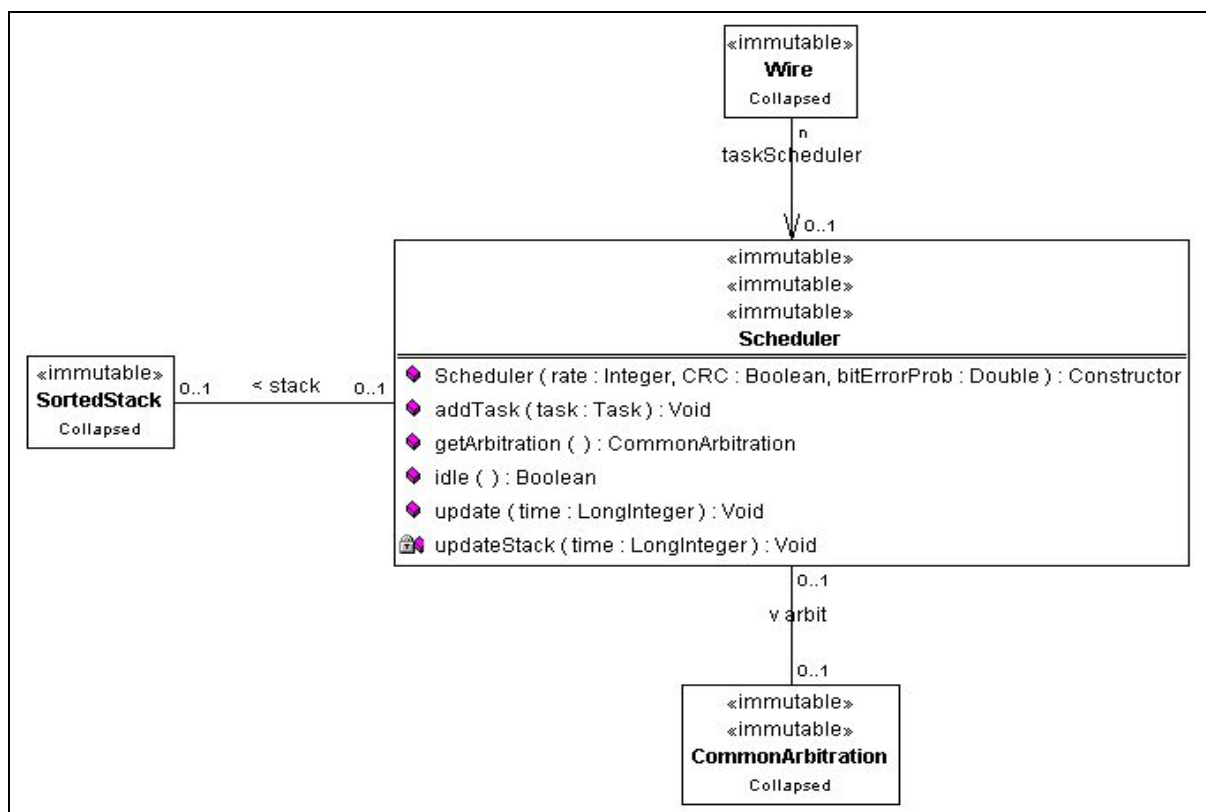
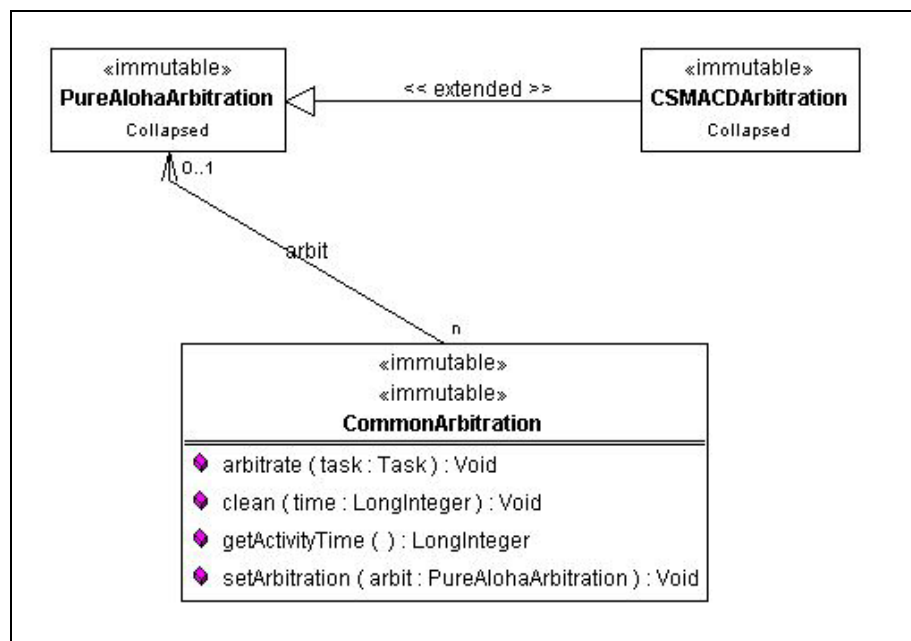


Abbildung 3.12: Die Klasse Scheduler

Entsteht im weiteren Verlauf jedoch ein Konflikt, so müssen die Prozesse nach einer oft zufällig gewählten Zeitspanne einen erneuten Zugriff auf das Medium erhalten. Die am Konflikt beteiligten Prozesse werden also erneut an den Scheduler übergeben, der diese mit Hilfe einer Datenstruktur vom Typ *SortedStack* verwaltet. Die Übertragungsprozesse werden so entsprechend ihrer nächsten Zugriffszeit in einem Stack abgelegt. Erst wenn eine neue, aktuelle Zeit an den Scheduler übergeben wird, prüft dieser, ob alte Übertragungsprozesse aus dem Stack genommen werden müssen. So muss auch beim Aufruf des Schedulers mit einer neuen Instanz vom Typ *Task* überprüft werden, ob sich noch Übertragungsprozesse im Stack befinden, die bevorzugt zu verwenden sind.

### 3.7 Arbitrierungsklassen

Bei der Arbitrierung befinden wir uns nach dem ISO/OSI Modell auf Schicht 2. Es findet hier der eigentlich Zugriff auf das Medium statt, bei dem die einzelnen Bits bzw. Bytes übertragen werden. Wie im letzten Abschnitt dargestellt wurde, übergibt der Scheduler die Übertragungsprozesse an ein Objekt der Klasse *CommonArbitration*. Um bei der Arbitrierung mehrere Verfahren einsetzen zu können, wurde hier wieder ein Strategy Pattern verwendet. Abbildung 3.13 verdeutlicht dieses Pattern und zeigt den allgemeinen Aufruf der Arbitrierung.



**Abbildung 3.13:** Strategy Pattern für *CommonArbitration*

Mit Hilfe der Methode *setArbitration()* hat man die Möglichkeit, sich bei der Instanziierung zwischen den Klassen *PureAlohaArbitration* und *CSMAcDArbitration* zu entscheiden. Diese Klassen entsprechen dem reinen Aloha-Zugriffsverfahren und dem CSMAcD-Zugriffsverfahren (IEEE 802.3), das bei Ethernet eingesetzt wird. Die weiteren Methoden sind unabhängig vom eingesetzten Verfahren. Die Klasse *CSMAcDArbitration* erbt von der Klasse *PureAlohaArbitration*, welche die gleiche Spezifikation aufweist. Die Methode *arbitrate(Task task)* übergibt einen Übertragungsprozess an das Medium, welcher entweder zu einer Belegung des Mediums bis zu einer berechneten Zeit führt, oder einen Konflikt hervorruft. Die Methode *clean(long time)* übergibt nur die aktuelle Zeit an das Medium und überprüft, wie weit die Übertragung bereits fortgeschritten ist. Ebenfalls sehr wichtig ist die Methode

*getActivityTime()*, mit deren Hilfe die Hauptschleife abfragen kann, wann der Empfang unterbrochen werden muss, um gegebenenfalls die Abarbeitung eines Übertragungsprozesses abzuschließen. Es soll nun anhand der Klasse *CSMACDArbitration* erklärt werden, wie die Arbitrierung im einzelnen implementiert wurde.

Die Klasse *CSMACDArbitration* simuliert den Zugriff auf das Medium nach dem CSMACD Verfahren. Bei der Instanzzierung dieser Klasse werden folgende Parameter übergeben:

- das aufrufende Objekt der Klasse *Scheduler* (bidirektionale Assoziation)
- die Übertragungsrate in Byte/Millisekunde
- ein *boolean* Wert, der entscheidet, ob das CRC Verfahren eingesetzt wird
- die Bitfehlerwahrscheinlichkeit
- die CSMACD Persistenz

Die Übergabe des Scheduler-Objekts wird benötigt, um Übertragungsprozesse, die an einem Konflikt beteiligt sind, wieder an den Scheduler zu übergeben. Alle anderen Parameter entsprechen den Größen in einem realen Netzwerk und benötigen keine weitere Erklärung. Die Auswirkungen dieser Parameter sollen im weiteren Verlauf erklärt werden. Folgende Abbildung 3.14 zeigt die Klasse *CSMAArbitration* und ihre Assoziationsklassen:

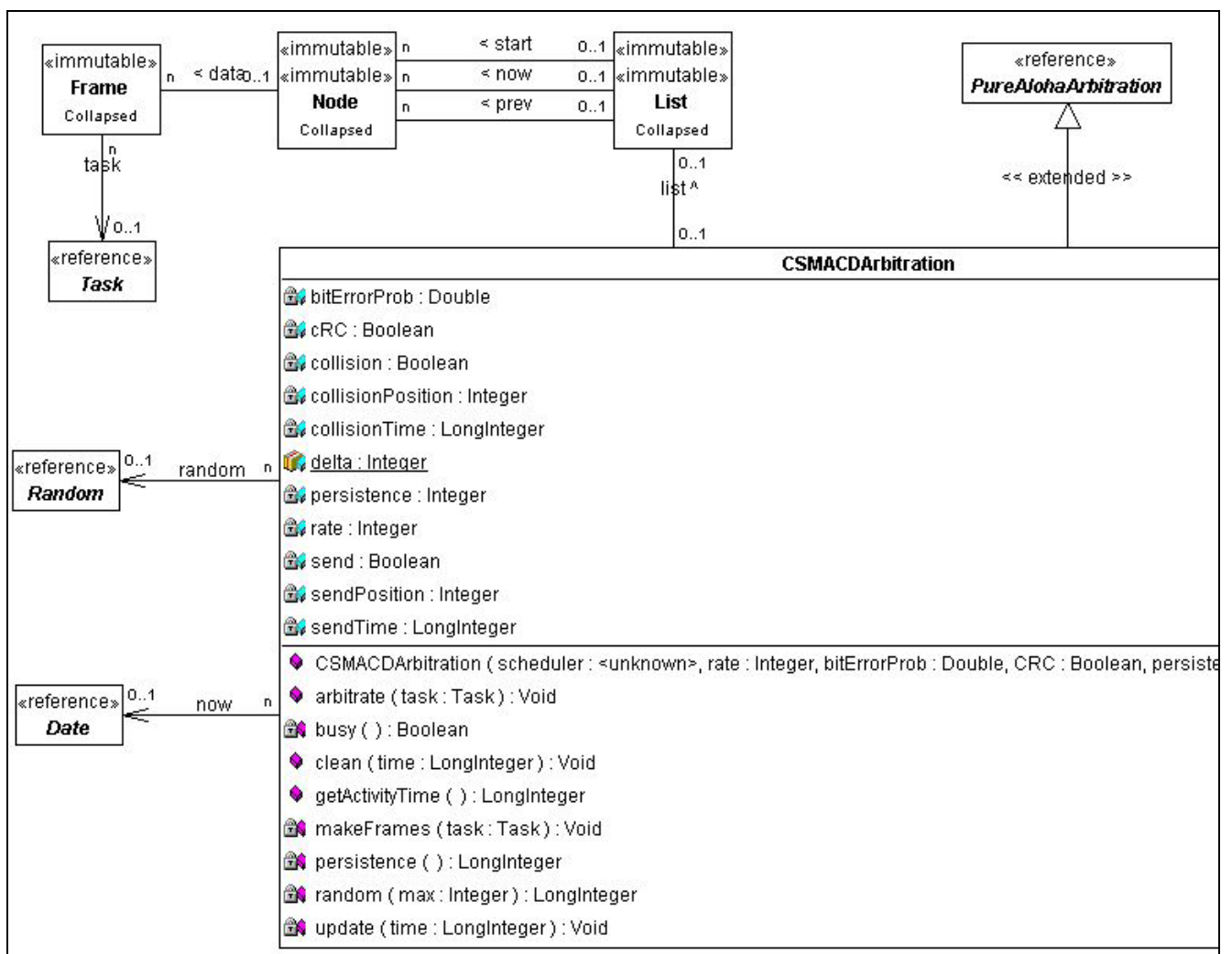


Abbildung 3.14: Die Klasse *CSMACDArbitration*

### Ankunft eines neuen Übertragungsprozesses:

Soll auf das Medium zugegriffen werden, so wird die Methode *arbitrate(Task task)* aufgerufen, welche den Übertragungsprozess an eine Instanz von CSMAArbitration übergibt. Die Methode *update(long time)* sorgt zunächst dafür, dass festgestellt werden kann, in welchem Zustand sich das Medium beim Eintreffen des Übertragungsprozesses befindet. Ist das Medium belegt, so wechselt die Arbitrierung in einen Kollisionszustand. Sowohl der Prozess, der gerade das Medium belegt als auch der neu ankommende Prozess kollidieren und müssen wieder an den Scheduler übergeben werden. Die Zeit für den neuen Versuch wird dabei zufällig festgelegt. Ist das Medium nicht belegt, so müssen drei Fälle unterschieden werden:

1. Die Arbitrierung befindet sich im Kollisionszustand, der Zeitpunkt der Kollision wurde jedoch noch nicht erreicht (Kollision ist vorprogrammiert, wenn zwei Prozesse warten), so dass der neue Übertragungsprozess ebenfalls kollidiert.
2. Der Übertragungsprozess, der das Medium belegt, sendet gerade sein letztes Frame, so dass der neue Prozess nur noch warten muss, bis das Medium frei ist.
3. Das Medium ist frei und kann uneingeschränkt genutzt werden.

Ist das Medium frei, so geht der neue Übertragungsprozess davon aus, dass er das Medium uneingeschränkt nutzen kann. Es werden sofort alle *Frame*-Instanzen an das Medium übergeben. Dieses wird mit einer Listenstruktur realisiert, die den zeitlichen Verlauf des Mediums darstellt und in dem die Instanzen vom Typ *Frame* abgelegt werden. Jedes Frame speichert dabei seinen Startpunkt, die Übertragungsdauer und einen Verweis auf sein dazugehöriges Objekt der Klasse *Task*. Der Zeitverlauf der einzelnen Frames wird nun vor allem durch die Übertragungsrate und durch die Bitfehlerwahrscheinlichkeit festgelegt. Die Übertragungsrate wirkt sich dabei auf die Übertragungsdauer der einzelnen Frames aus und legt somit fest wie lange das Medium von einem Frame belegt ist. Die Bitfehlerwahrscheinlichkeit legt fest, wie groß die Wahrscheinlichkeit ist, dass ein Frame ein weiteres Mal übertragen werden muss. Da bei einem Bitfehler das gesamte Frame neu übertragen wird, hat die Bitfehlerwahrscheinlichkeit einen großen Einfluss auf die Belegung des Mediums und auf die Gesamtdauer des Übertragungsprozesses. Wird CRC nicht verwendet, so entsteht hier kein Einfluss auf die Übertragungsdauer, was sich aber wie bereits oben erklärt auf die Zuverlässigkeit der Übertragung auswirkt. Es soll hier noch mal betont werden, dass in der Praxis bei TCP/IP immer CRC verwendet wird. Da das CRC Verfahren äußerst effizient arbeitet, geht man in der Simulation von folgendem Verhalten aus:

- Wird CRC verwendet, so werden alle Bitfehler erkannt, was in jedem Fall ein erneutes Versenden des Frames hervorruft. Die Übertragung arbeitet völlig zuverlässig.
- Wird CRC nicht verwendet, so wird kein Bitfehler erkannt, so dass jedes Frame nur einmal übertragen wird. Die Bitfehler wirken sich auf die Übertragung aus.

Diese Implementierung dürfte den meisten Anforderungen genügen, da CRC bei realistischen Störeinflüssen praktisch alle Bitfehler erkennt. Im folgenden soll nun das Verfahren erklärt werden, nach dem der Zustand des Mediums aktualisiert wird bzw. das Medium wieder in den freien Zustand versetzt wird.

### **Abarbeiten eines Übertragungsprozesses:**

Jedes Mal wenn der Scheduler die Methode *update* aufruft, wird in der Arbitrierungsklasse die Methode *clean(long time)* aufgerufen. Da hierbei kein neuer Übertragungsprozess anliegt, wird nur die Methode *update(long time)* aufgerufen um den Fortschritt der Übertragung zu ermitteln. Dabei wird die Liste einfach von Anfang bis zu dem übergebenen Zeitpunkt durchlaufen. Wird dabei das letzte Frame des Übertragungsprozesses erreicht, wird der Zeitpunkt für das Ende der Übertragung an den Übertragungsprozess übergeben. Der Übertragungsprozess berechnet nun die Gesamtdauer der Übertragung und sendet schließlich ein Bestätigungssignal an seinen Socket, damit das reale Paket übertragen werden kann. Der Übertragungsprozess ist somit abgeschlossen.

### **Einfluss der Leitungslänge:**

Bei dem bisherigen Verfahren wurde der Einfluss der Leitungslänge vernachlässigt, was gerade bei langen Leitungen zu Ungenauigkeiten führt. Da bei der Simulation der Arbitrierung genau genommen immer maximal 2 Übertragungsprozesse im Bezug zueinander stehen, lässt sich der Einfluss, den die Länge des Mediums auf die Übertragung hat, einfach simulieren. Um die Position des Übertragungsgerätes am Netz festzulegen, weist man jeder Anwendung, die auf das Netz zugreift, eine bestimmte Position zu. Kennt jeder Übertragungsprozess die Position seiner Anwendung, so lässt sich die Verzögerung zwischen beiden Übertragungsprozessen berechnen. Die Klasse *CSMAArbitration* berechnet diese Verzögerungszeit anhand der Positionen der Übertragungsprozesse und übergibt sie an die Variable *delta*. Der Wert von *delta* wird dann für alle weiteren Berechnungen verwendet und wirkt sich vor allem auf den Zeitpunkt aus, zu dem ein Übertragungsprozess einen Konflikt erkennt.

### **Einfluss der Persistenz:**

Das *persistence*-Attribut bildet ein mächtiges Werkzeug, da es dazu dient, weitere Zugriffsverfahren zu definieren. Da sich das Attribut frei bestimmen lässt, sind Zugriffsverfahren wie z.B. 0.1 persistentes CSMA/CD oder 0.001 persistentes CSMA/CD leicht zu realisieren. Auch 0 persistentes CSMA/CD, das etwas aus dem Rahmen fällt, stellt für die Simulation kein Problem dar. Der Wert für die Persistenz lässt sich demnach frei wählen, um die Umsetzung des CSMA/CD-Zugriffsverfahrens für eine Anwendung optimal zu konfigurieren.

### **Arbitrierung mit dem reinen Aloha-Zugriffsverfahren:**

Wird an Stelle der Klasse *CSMAArbitration* die Klasse *PureAlohaArbitration* instanziiert, so erfolgt die Arbitrierung mit einem reinen Aloha-Zugriffsverfahren. Die Implementierung ist hier etwas einfacher, da bei diesem Verfahren keine Kollisionen erkannt werden und auch keine Persistenz berücksichtigt werden muss. Das Medium wird hier ebenfalls mit Hilfe einer Liste beschrieben. Auf den Einfluss der Leitungslänge wurde hier zunächst verzichtet, da sich diese aufgrund der fehlenden Konflikterkennung weniger stark auf das Übertragungsverhalten auswirkt.

## 4 Anwendung der Simulationsumgebung

### 4.1 Einfluss der Hardware

Da die Berechnungen, welche in der Simulation stattfinden, recht komplex sein können, hat die Hardware, mit deren Hilfe der Simulationsprozess ausgeführt wird, einen gewissen Einfluss auf die Verwendbarkeit der Simulation.

Die Simulation gerät an ihre Grenzen, wenn sehr viele Anwendungen gleichzeitig auf das virtuelle Netzwerk zugreifen, da die Berechnungen die während der Simulation stattfinden, nicht unendlich schnell ablaufen können. Werden die Berechnungen nicht schnell genug ausgeführt, so entstehen aufgrund des Scheduling-Ansatzes zunächst keine Fehler bei der internen Berechnung der Übertragungszeit. Sobald allerdings die Berechnungszeit für die Übertragung eines Prozesses seine errechnete Übertragungszeit übersteigt, erfahren die Anwendungen zu spät von der erfolgreichen Übertragung. Dieses kann folgende Auswirkungen haben:

- Sendepuffer werden zu spät freigegeben und es gehen Pakete verloren, die in der Realität problemlos übertragen werden.
- Empfänger warten zu lange auf ihre Daten, was die Funktionalität eines Systems stören kann.
- Zeitmessungen in den Anwendungen liefern falsche Ergebnisse.

Diese Nachteile entstehen vor allem dann, wenn man das Übertragungsverhalten in Echtzeit simulieren will. In den meisten Anwendungsfällen dürften die beteiligten Anwendungen von so einem Verhalten ausgehen, da alle Anwendungen in einer realen Umgebung in Echtzeit ablaufen. Will man dennoch komplexere Simulationen mit weniger CPU-Leistung erzielen, so empfiehlt es sich, die zeitlichen Abläufe der Anwendungen um einen bestimmten Faktor zu verlängern. Um die Simulation an diese virtuelle Zeit anzupassen, muss man die gewünschte Übertragungsrate dann durch diesen Faktor dividieren. Die gesamte Simulation arbeitet so in Zeitlupe und es lassen sich so beliebig komplexe Anwendungsfälle simulieren.

Der für die Simulation verfügbare Speicher hat keinen Einfluss auf die Simulation, da die verwendeten Datenstrukturen bei der Simulation realistischer Anwendungen nur wenig Speicher benötigen.

Prinzipiell ist es möglich, die Simulation, die in gewisser Weise einen *Server* darstellt, auf einem separat ans Netzwerk angeschlossenen Rechner zu verwenden. Alle von den Anwendungen aufgerufenen Instanzen der Klasse *SimSocket* bilden somit mehrere *Clients*, die auf diesen Server zugreifen. Folgendes Beispiel aus der Physik soll die Probleme bei dieser Art der Anwendung deutlich machen:

Es soll versucht werden, mit einem Thermometer, welches eine Eigentemperatur von 40° C hat, die Temperatur einer Flüssigkeit in einem kleinen Reagenzglas zu messen. Die Temperatur dieser Flüssigkeit beträgt vor dem Eintauchen des Thermometers 10° C. Taucht man das Thermometer in die Flüssigkeit ein, so haben nach einer gewissen Zeit sowohl die Flüssigkeit als auch das Thermometer eine Temperatur von 15° C. Genau diese Temperatur misst das Thermometer und liefert somit ein falsches Ergebnis! Um ein genaues Messergebnis zu erhalten, muss das Volumen des Thermometers gleich 0 oder zumindest vernachlässigbar klein sein.

Auch bei der Simulation wird das simulierte Übertragungsverhalten durch das Vorhandensein eines realen Netzwerkes verfälscht, da hierdurch eine zusätzliche Verzögerung hervorgerufen wird. Um bei der Simulation ein genaues Messergebnis zu

erhalten, muss die Übertragungsrate eines realen Netzwerkes folglich unendlich groß sein oder zumindest so groß, dass der Einfluss auf die Simulation vernachlässigbar klein ist. Da einem die Simulationsumgebung die Möglichkeit bietet, virtuelle Pakete mit einer definierten Länge zu verschicken die aber real nur wenige Bytes lang sind, ist diese Art der Anwendung jedoch nicht ausgeschlossen.

## 4.2 Ausführen der Simulationsumgebung

Aufruf innerhalb der Anwendungen die auf die Simulation zugreifen:

Wie bereits der Ansatz gezeigt hat, kommunizieren alle Anwendungen über eine Instanz der Klasse *CommonSocket* mit dem virtuellen Netzwerk. An diese Instanz werden keine Werte übergeben, so dass die Instanzierung in jedem Fall folgendermaßen aussieht:

```
name = new CommonSocket()
```

Um den Socket für das Versenden an die Simulation zu benutzen, wird eine Instanz der Klasse *SimSocket* an den allgemeinen Socket übergeben. Hierzu dient die Methode *setSocket(..)*:

```
name.setSocket (new SimSocket(...))
```

An die Instanz von *SimSocket* können wahlweise ein Port, eine URL und ein Name übergeben werden. Folgende Beispiele zeigen den Einsatz der Klasse *SimSocket*:

- `new SimSocket(5443, "Kühlschrank")`

⇒ Eine Anwendung mit dem Namen "Kühlschrank" soll über den Port 5443 mit der Netzwerksimulation kommunizieren.

- `new SimSocket("Videorecorder")`

⇒ Eine Anwendung mit dem Namen "Videorekorder" wird über einen Port, der gerade zur Verfügung steht, mit der Netzwerksimulation verbunden.

- `new SimSocket(5336, "aeg", "Toaster")`

⇒ Eine Anwendung mit dem Namen „Toaster“ kommuniziert über die URL „aeg“ und den Port 5336 mit der Netzwerksimulation.

Alle Methoden der Klasse *SimSocket* entsprechen prinzipiell den Methoden der Klasse *DatagramSocket* und müssen somit nicht angepasst werden. Will man die Übertragung von Paketen beliebiger Größe simulieren so benutzt man die Methode *send(DatagramPacket packet, int length)*, wie folgendes Beispiel zeigt.

```
socket.send(packet, 65536)
```

Es wird ein virtuelles Paket mit einer Größe von 64 kByte an die Netzwerksimulation übergeben. Die Instanz packet vom Typ *DatagramPacket* kann beliebig groß sein und beliebi-



ge Daten enthalten. Sie wirkt sich somit nicht auf das Verhalten der Simulation aus und kann vielseitig verwendet werden.

#### Aufruf der Netzwerksimulation:

Im Folgenden soll nun gezeigt werden, wie man den Prozess für die Simulation des virtuellen Netzwerkes aufruft. Um diesen Prozess zu starten wird einfach eine Instanz vom Typ *Wire* erzeugt, die daraufhin als Thread gestartet wird. Der Aufruf hat die folgende Form:

- `new Wire(DatagramSocket socket,int rate,boolean CRC,double BitErrorProb,String Arbitration,int persistence).start();`

Zunächst wird ein Objekt der Klasse *DatagramSocket* übergeben, welches den Port und die URL festlegt. Beide Werte bilden die Adresse, an die sich alle Instanzen vom Typ *SimSocket* wenden müssen, um ihre Datenpakete zu verschicken. Es muss also hier eine Übereinstimmung der entsprechenden Werte geben. Das Attribut *rate* legt die Übertragungsrate des simulierten Netzwerkes in Bytes pro Millisekunde fest. *CRC* entscheidet, ob eine Bitfehlererkennung verwendet wird oder nicht und *BitErrorProb* legt die Wahrscheinlichkeit fest, mit der ein Bit falsch übertragen wird. *Arbitration* bestimmt das Arbitrierungsverfahren und *persistence* definiert gegebenenfalls die Persistenz eines Arbitrierungsverfahrens.

Folgende Beispiele zeigen anschaulich den Einsatz der Klasse *Wire*:

- `new Wire(socket, 12500, true, 0.0001, "CSMACD", 1)`

⇒ Es wird ein Netzkabel mit einer Übertragungsrate von 100 Mbit simuliert. Bitfehler treten mit einer Wahrscheinlichkeit von 1/10000 auf und werden durch die Fehlererkennung aufgefunden. Für die Arbitrierung sorgt 1 persistentes CSMACD.

- `new Wire(socket, 1250, false, 0.001, "PureAloha", 1)`

⇒ Es wird ein Netzkabel mit einer Übertragungsrate von 10 Mbit simuliert. Bitfehler treten mit einer Wahrscheinlichkeit von 1/1000 auf und werden durch die Fehlererkennung nicht erkannt. Für die Arbitrierung sorgt das reine Aloha-Zugriffsverfahren.

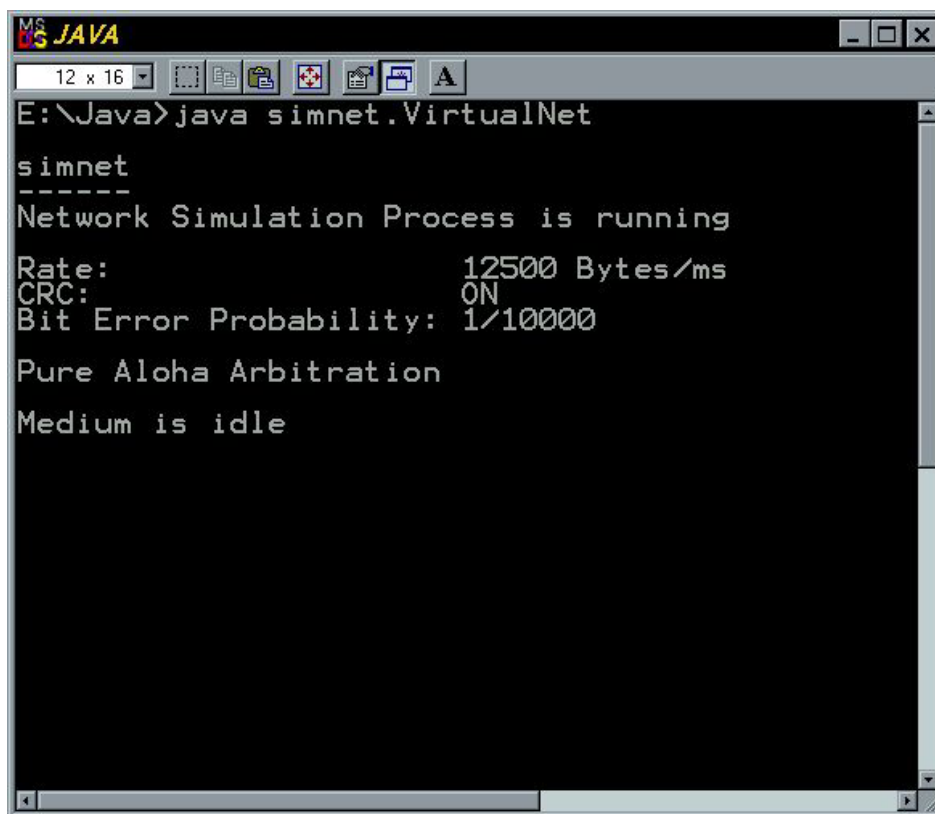
Es lassen sich so sehr verschiedene Übertragungsmedien mit unterschiedlichen Zugriffsverfahren simulieren. Wie sich die Einstellungen auf das Übertragungsverhalten auswirken, wird in Abschnitt 4.4 anhand von Testergebnissen gezeigt.

### 4.3 Die Ausgabe der Simulation

Um die Arbeit mit der Simulationsumgebung zu erleichtern und um Messungen zu ermöglichen, erzeugt die Simulationsumgebung verschiedene Ausgaben. Diese Ausgaben liefern sowohl der Simulationsprozess für das Übertragungskabel, als auch die Instanzen der Klasse *SimSocket*, die zum Anschluss an dieses Kabel dienen. Das Programm beschränkt sich dabei darauf, Texte über die Konsole auszugeben, was sicherlich keine optimale Darstellung der Simulation ermöglicht. Es lassen sich allerdings wie man im nächsten Abschnitt sieht bereits einige interessante Szenarien beschreiben. Im folgenden sollen die Ausgaben des Programms im allgemeinen erklärt werden.

#### Aufruf des virtuellen Netzwerkes:

Beim Start der Simulation erhält man folgende Ausgabe:



```
MS JAVA
12 x 16
E:\Java>java simnet.VirtualNet

simnet
-----
Network Simulation Process is running

Rate:                12500 Bytes/ms
CRC:                  ON
Bit Error Probability: 1/10000

Pure Aloha Arbitration

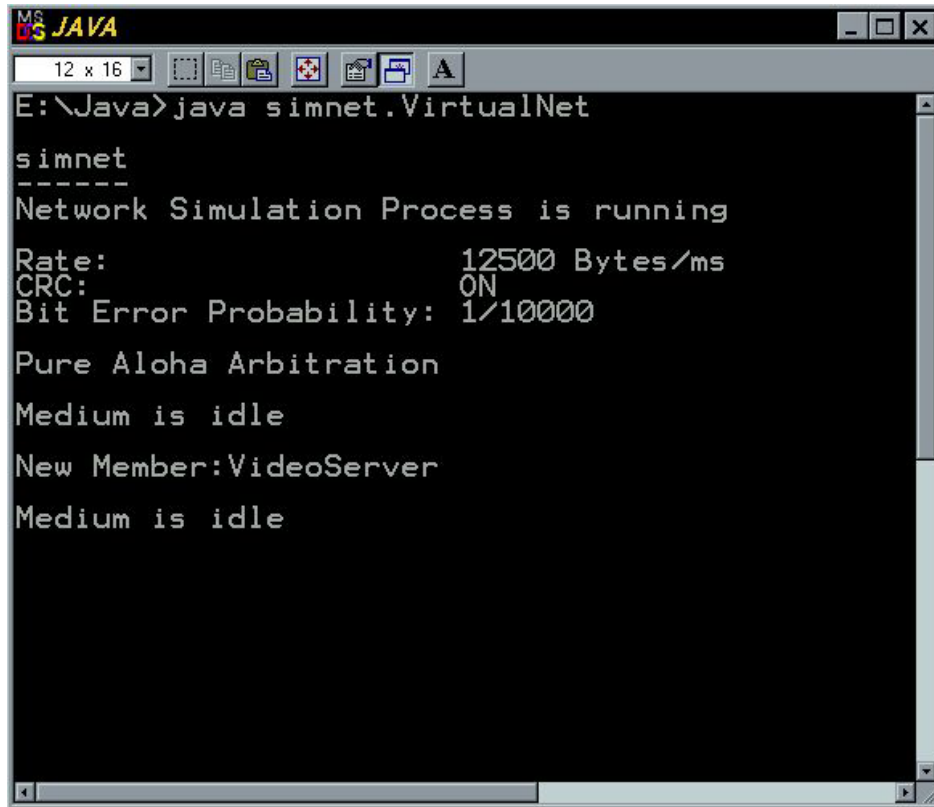
Medium is idle
```

**Abbildung 4.1:** Aufruf des virtuellen Netzwerkes

Das virtuelle Netzwerk kündigt seine Bereitschaft an und gibt noch mal alle relevanten Einstellungen für die Übertragung aus. Man kann sich so vergewissern, dass man die richtigen Werte an die Simulation übergeben hat. Die Meldung „Medium is idle“ signalisiert, dass im Moment keine Übertragung stattfindet.

### Anmeldung eines neuen Netzwerkteilnehmers:

Greift eine neue Anwendung über den entsprechenden Socket auf das virtuelle Netzwerk zu, so erhält man die Ausgabe in Abbildung 4.2:



```
MS JAVA
12 x 16
E:\Java>java simnet.VirtualNet

simnet
-----
Network Simulation Process is running

Rate:                12500 Bytes/ms
CRC:                  ON
Bit Error Probability: 1/10000

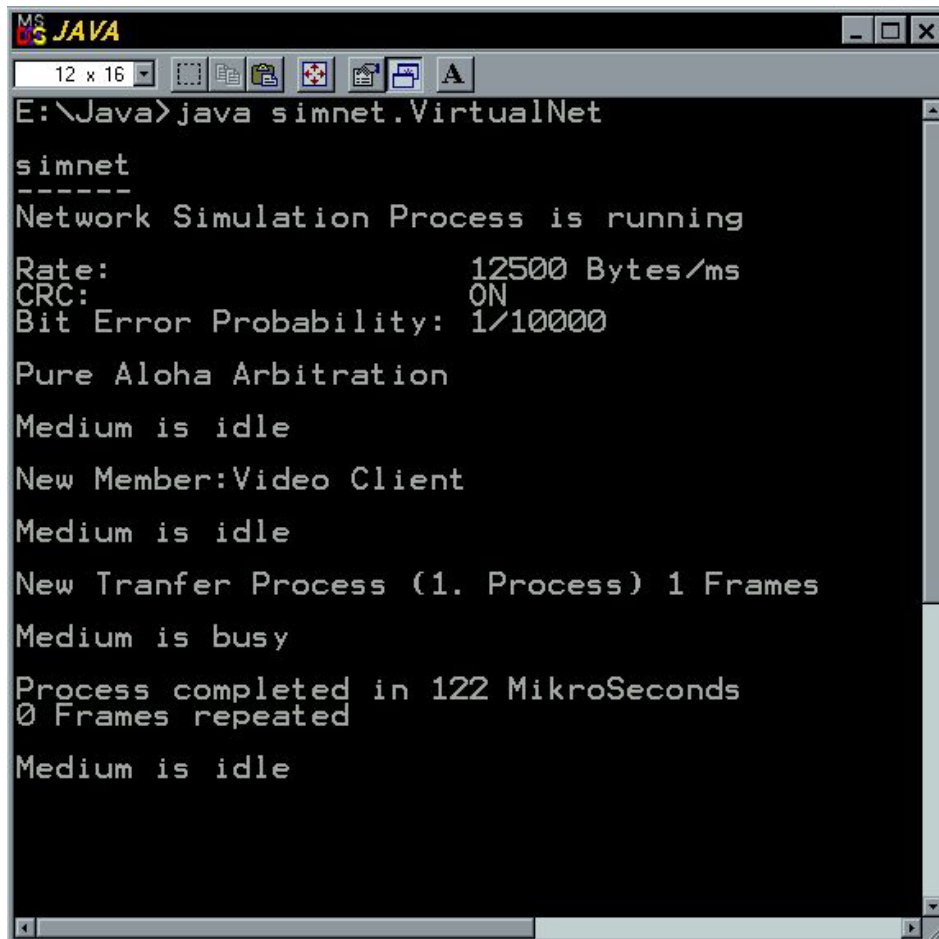
Pure Aloha Arbitration
Medium is idle
New Member:VideoServer
Medium is idle
```

**Abbildung 4.2:** Anmeldung eines neuen Netzwerkteilnehmers

Hat sich die Anwendung mit ihrem Namen beim Netzwerk angemeldet, so gibt die Simulation diesen Namen aus. Man erhält so eine Bestätigung für einen erfolgreichen Anschluss an das virtuelle Netzwerk.

## Übertragung von Daten:

In Abbildung 4.3 kann man erkennen, auf welche Weise die Simulation die Übertragung von Daten darstellt:

A screenshot of a Java command window titled "MS JAVA". The window shows the execution of the command "java simnet.VirtualNet". The output text is as follows:

```
E:\Java>java simnet.VirtualNet

simnet
-----
Network Simulation Process is running

Rate:                  12500 Bytes/ms
CRC:                   ON
Bit Error Probability: 1/10000

Pure Aloha Arbitration
Medium is idle
New Member:Video Client
Medium is idle
New Tranfer Process (1. Process) 1 Frames
Medium is busy
Process completed in 122 MikroSeconds
0 Frames repeated
Medium is idle
```

**Abbildung 4.3:** Übertragung von Daten

Trifft ein neues UDP-Paket ein, so erhält man zunächst die Ausgabe „New Transfer Prozess“. Es wird außerdem angezeigt, wie viele Übertragungsprozesse gerade an der Arbitrierung teilnehmen, und wie viele Rahmen versendet werden sollen. Die Abbildung zeigt die Übertragung eines Startsignals, das ein Video-Client an einen Server schickt. Da das Signal aus wenigen Bytes besteht, wird hier nur ein Frame an das Netzwerk übergeben. Während der Übertragung signalisiert die Simulation mit der Ausgabe „Medium is busy“, dass das Netzwerk belegt ist. Ist die Übertragung abgeschlossen, so wird die Übertragungsdauer und die Anzahl der Rahmen, die aufgrund von Bitfehlern erneut verschickt werden müssen, ausgegeben.

### Ausgaben der Klasse *SimSocket*:

Wird eine Anwendung an das virtuelle Netzwerk angeschlossen, so erhält man hier ebenfalls einige Ausgaben, die den Zustand des Netzteilnehmers beschreiben. In Abbildung 4.4 ist die Ausgabe des Video-Clients dargestellt, der auf das virtuelle Netzwerk zugreift:



```
MS JAVA
12 x 16
E:\Java>java VideoClient

SimSocket:Announce Member
Simsocket:CRC ON
Buffer:65481
Buffer:Send begins
Buffer:Send ends
```

Abbildung 4.4: Ausgaben der Klasse *SimSocket*

Die Instanz der Klasse *SimSocket* signalisiert ihrerseits die Anmeldung beim Netzwerk und zeigt den CRC-Status an, den sie von diesem Netzwerk empfängt. Jedes Mal, wenn eine Übertragung stattfindet, wird der verbleibende Pufferplatz ausgegeben. „Send begins“ signalisiert daraufhin den Beginn der Übertragung. Meldet das virtuelle Netzwerk das Ende der Übertragung, so erfolgt die Ausgabe „Send ends“. Falls der Pufferplatz nicht mehr ausreicht, erhält man die Ausgabe „Packets lost“. Die Übertragung ist somit gescheitert.

## 4.4 Testanwendungen und Statistiken

Anhand der folgenden Anwendungen soll verdeutlicht werden, wie sie sich die Testumgebung für verschiedene Szenarien einsetzen lässt.

### Szenario 1: Digitale Videoübertragung mit einem VideoServer

Das virtuelle Netzwerk soll dafür eingesetzt werden um die Kommunikation zwischen einem Video-Client und einem Video-Server zu simulieren. Der Video-Client sendet ein Start-Signal an den Video-Server. Dieser beginnt daraufhin damit, Video-Packages an den Video-Client zu senden. Das Verschicken der Video-Packages läuft dabei nach folgendem Schema ab:

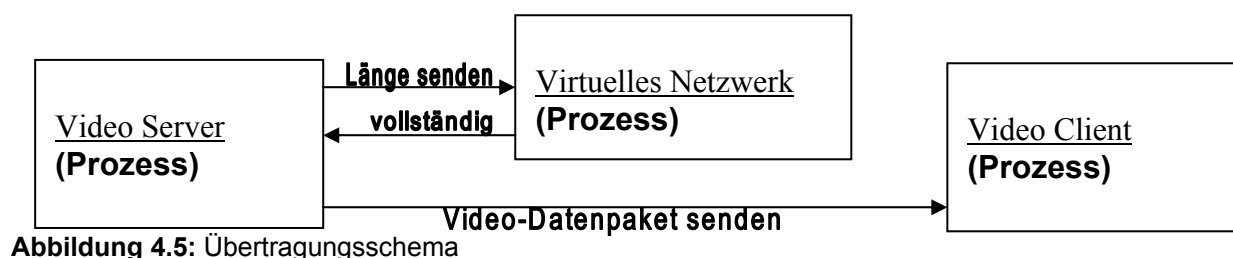


Abbildung 4.5: Übertragungsschema

Mit folgenden Befehlen lässt sich der Video-Server an das virtuelle Netzwerk anschließen, dabei wird der Sendepuffer auf eine Größe von 4 Mbyte gesetzt:

```
socket=new CommonSocket();
socket.setSocket(new SimSocket(3001,"VideoServer"));
socket.setSendBufferSize(4194304);
```

Der Video-Server sendet nun alle 5 s ein Video-Package der Größe 1 Mbyte an den Video-Client, wobei jeweils der folgende Befehl ausgeführt wird:

```
socket.send(packet,1048576);
```

Es soll nun zunächst untersucht werden, wie sich die Übertragungsrate auf das Verhalten der Übertragung auswirkt. Bei den Messungen wurde die Bitfehlerwahrscheinlichkeit für das Kabel auf 0.0001 festgelegt, CRC wurde aktiviert. Für die Arbitrierung wurde CSMA/CD eingestellt. Die Netzwerksimulation zeigt die benötigte Übertragungszeit für ein Package an, während der Video-Client immer den zeitlichen Abstand der ankommenden Packages misst. Abbildung 4.6 gibt eine Übersicht über die Messergebnisse, wobei jeweils ein Ausschnitt aus einer Messreihe abgebildet wird.

Übertragungsrate	Übertragungszeiten (µs)	Package-Abstände (ms)
<b>6 MBit/s</b>	...5180367,4693299,5217301, 4933842,4788897,4654653,...	...5000,5210,4780,5000,9170, 3190,7360,270,8020,...
<b>10 Mbit/s</b>	...2866296,2861416,2950476, 2911436,2995616,2818012,...	...4710,5160,4840,5260,5110, 5110,4830,4890,5050,...
<b>50 Mbit/s</b>	...567819,566774,589119, 578871,605223,584727,...	...4940,5060,4940,5050,5000, 5000,4990,5000,5060,...
<b>100 Mbit/s</b>	...295169,289679,298097, 297853,300395,288388,...	...4990,5000,5000,5000,5000, 5050,4940,5000,5000,...
<b>1000 Mbit/s</b>	...30053,28865,29825, 27761,28313,28589,...	...5000,5000,5000,4990,5000, 5000,4990,5000,5000,...

**Abbildung 4.6:** Einfluss der Übertragungsrate

Schaut man sich die Übertragungszeiten an, so fällt einem auf, dass sich diese Zeiten bei größeren Übertragungsraten immer weiter verkleinern. Dieses Ergebnis entspricht der Erwartung, da höhere Übertragungsraten im allgemeinen dazu dienen die Übertragung zu beschleunigen. Da die Bitfehler zufällig auftreten, und Pakete eventuell neu versendet werden müssen, weichen die einzelnen Übertragungszeiten voneinander ab.

Die zeitlichen Abstände mit denen die Packages den Video-Client erreichen hängen ebenfalls von der Übertragungsrate ab. Sehr große Schwankungen treten hier bei einer Übertragungsrate von 6 MBIT/s auf. Da die Übertragungsdauer ab einer bestimmten Zahl von Bitfehlern den zeitlichen Sende-Abstand von 5 s überschreitet, kommt es zu Kollisionen, die zu extremen Verzögerungszeiten der Prozesse führen. Im weiteren Verlauf der Simulation führt diese Verzögerungszeiten sogar an einigen Stellen zu einem Überlauf des Sendepuffers, so dass die Simulation den Verlust von Rahmen meldet. Erhöht man die Übertragungsrate, so lässt sich klar erkennen, dass die Schwankungen immer kleiner werden. Diese Messung zeigt, dass sich Bitfehler bei einer höheren Übertragungsrate weniger stark auf das Übertragungsverhalten auswirken. Das erneute Versenden von Rahmen nimmt hier weniger Zeit in Anspruch.

Als nächstes soll untersucht werden, wie sich die Bitfehlerwahrscheinlichkeit auf das Übertragungsverhalten auswirkt. Die Übertragungsrate wird dabei konstant auf 100 Mbit/s festgelegt. Man erhält hier folgende Messungen:

Bitfehlerwahrsch.	Übertragungszeiten ( $\mu$ s)	Package-Abstände (ms)
<b>0</b>	konstant 86671	...5000,5000,5000,5000, 4990,5000,5000,5000,...
<b>0.00001</b>	...97651,96431,97824, ,98383,96919,99603,...	...5000,5000,5000,4990, 5060,5000,4990,5000,...
<b>0.0001</b>	...295169,289679,298097, 297853,300395,288388,...	...4710,5160,4840,5260,5110, 5110,4830,4890,5050,...
<b>0.000328</b>	...4567010,4841530,4973656, 4654595,4949196,4859149,...	...4990,5000,5000,5000, 5000,5060,4990,5000,...

**Abbildung 4.7:** Einfluss der Bitfehlerwahrscheinlichkeit

Die Bitfehlerwahrscheinlichkeit hat, wie man sehen kann, einen großen Einfluss auf die Übertragungszeit. Treten keine Bitfehler auf, so ist die Übertragungszeit immer konstant. Dieses entspricht dem Verhalten bei einer ausgeschalteten Fehlererkennung. Erhöht man die Bitfehlerwahrscheinlichkeit, so steigt die Übertragungszeit explosionsartig an. Schon kleine Änderungen wirken sich dabei stark auf das Übertragungsverhalten aus. Bei einer Bitfehlerwahrscheinlichkeit von 0.000328 befinden wir uns an der Grenze, da die 5 Sekunden, die der Video-Server bis zum Senden des nächsten Packages wartet, beinahe überschritten werden. In diesem Fall führt das erneute Versenden von Datenpaketen, bei dem ebenfalls Bitfehler auftreten, zu einem Zusammenbruch des Netzes.

### **Szenario 2: Zwei Video-Server**

Es soll nun wieder ein Video-Server Daten an einen Video-Client schicken. Als Störeinfluss sendet jedoch ein weiterer Video-Server seine Daten an das Kabel. Bei dem zweiten Video-Server wurde auf einen Client verzichtet, da der Client nur Daten empfängt und somit keinen Einfluss auf das Übertragungsverhalten hat. Der zweite Video-Server wird auf die gleiche Weise wie Video-Server eins an das Netzwerk angeschlossen. Er unterscheidet sich vom dem ersten Video-Server durch die Wartezeit beim Senden, die hier genau 1 s beträgt. Außerdem sei erwähnt, dass der zweite Video-Server seine Packages an einen anderen Port, und somit nicht an den Video-Client schickt.

Es soll nun untersucht werden, wie sich die Übertragung des zweiten Video-Servers auf das Übertragungsverhalten des ersten Video-Servers auswirkt. Die Größe des Packages, welches der zweite Video-Server jeweils verschickt soll dabei variiert werden. Beide Puffer werden so eingestellt, dass immer nur ein Package im Puffer abgelegt werden kann. So wird eine totale Überlastung des Netzwerkes vermieden. Das simulierte Kabel soll nun eine Übertragungsrate von 100 Mbit/s und eine Bitfehlerwahrscheinlichkeit von 0.0001 besitzen. CRC ist aktiv und für die Arbitrierung wird zunächst CSMA/CD verwendet. Hier wurde nur gemessen, in welchen zeitlichen Abständen die Packages beim Video-Client eintreffen. Man erhält folgende Messungen:

Package-Größe (Video-Server 2)	Package-Abstände (ms) (Video Server 1→Video-Client)
1 Mbyte	5000/5330/23290/41130/.../43010/5000
512 kByte	5050/5000/29116/2850/3140/4900
128 kByte	5050/5000/4990/5000/5060/...

**Abbildung 4.8:** Interferenz von 2 Video-Servern

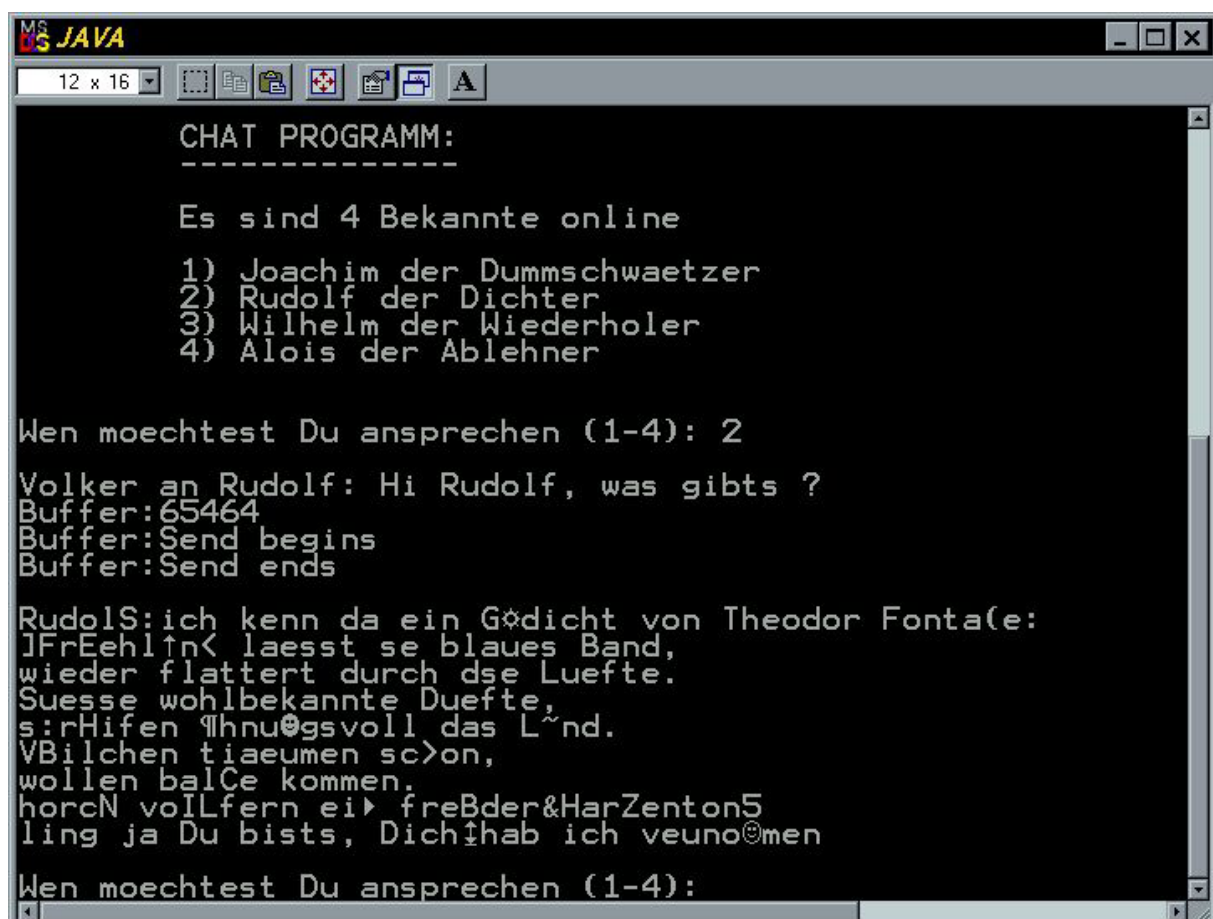
Man kann hier erkennen, dass sich Packages der Größe 128 kByte nicht auf das Übertragungsverhalten des ersten Video-Servers auswirken. Beträgt die Größe der Packages 512 kByte, so entstehen bei den Übertragungszeiten bereits größere Unregelmäßigkeiten, die

man jedoch noch mit einem relativ kleinen Puffer ausgleichen kann. Überträgt der zweite Video-Server 1 Mbyte Packages, so ist die Arbitrierung völlig überfordert, was man an bis zu 8-fachen Übertragungszeiten erkennt. Das Netzwerk ist also deutlich überlastet und lässt sich für diesen Zweck nicht einsetzen.

Ein interessantes Ergebnis erhält man auch, wenn man anstelle der CSMA/CD Arbitrierung die reine Aloha-Arbitrierung verwendet. Bei diesem weniger ausgereiften Verfahren entstehen hier teilweise 20-fache Übertragungszeiten.

### Szenario 3: Chat-Programm

In diesem Beispiel soll untersucht werden, wie sich eine fehlende Fehlererkennung auf ein Chat-Programm auswirkt, bei dem Texte über ein Netzwerk versendet werden. In Szenario 1 haben wir bereits gesehen, dass sich die Fehlererkennung bei einer großen Bitfehlerwahrscheinlichkeit äußerst negativ auf die Übertragungszeit auswirkt. Es kann somit unter Umständen Sinn machen auf die Fehlererkennung zu verzichten. Die Simulation stellt dabei nicht nur die Vorteile einer schnelleren Übertragung dar, sondern ist auch in der Lage die Dateien, welche über das virtuelle Netzwerk übertragen werden, mit Fehlern zu versehen. Abbildung 4.9 zeigt wie sich Bitfehler bei der Textübertragung auswirken:

The image shows a screenshot of a Java application window titled "MS JAVA". The window contains a text-based chat interface. The text displayed is as follows:

```
CHAT PROGRAMM:
-----

Es sind 4 Bekannte online
1) Joachim der Dummschwaetzer
2) Rudolf der Dichter
3) Wilhelm der Wiederholer
4) Alois der Ablehner

Wen moechtest Du ansprechen (1-4): 2
Volker an Rudolf: Hi Rudolf, was gibts ?
Buffer:65464
Buffer:Send begins
Buffer:Send ends

Rudolf:ich kenn da ein Gœdicht von Theodor Fonta(e:
]FrEehlin< laesst se blaues Band,
wieder flattert durch dse Luefte.
Suesse wohlbekannte Duefte,
s:rHifen ¶hnu@gsvoll das L~nd.
VBilchen tiaeumen sc>on,
wollen balCe kommen.
horcN voILfern ei> freBder&HarZenton5
ling ja Du bists, DichIhab ich veuno@men

Wen moechtest Du ansprechen (1-4):
```

The text shows a list of four contacts, a selection of contact 2 (Rudolf), and a message from Volker to Rudolf. The response from Rudolf contains several characters that appear to be corrupted or misinterpreted, likely due to the high bit error rate mentioned in the text. The window has a standard Java Swing look with a title bar, menu bar, and toolbar.

**Abbildung 4.9:** Textübertragung ohne Fehlererkennung

Für den Test wurde eine sehr hohe Bitfehlerwahrscheinlichkeit von 1/100 eingestellt. Die Bitfehler wirken sich hier deutlich auf die Übertragung eines bekannten Gedichtes von



Theodor Fontane aus. Bei einer Bitfehlerwahrscheinlichkeit von  $1/10000$  ist der Effekt kaum noch zu bemerken.

Auch bei der digitalen Videoübertragung könnten Protokolle, die auf eine Fehlererkennung verzichten, zum Einsatz kommen. Das menschliche Auge verhält sich relativ unempfindlich gegenüber Bitfehlern, so dass diese Fehler bis zu einer gewissen Häufigkeit nicht wahrgenommen werden. Da gerade digitale Videoübertragungen große Datenmengen verschicken, kann sich eine fehlende Fehlererkennung sogar positiv auf die Bildqualität auswirken, da Jitter-Effekte so evtl. vermieden werden.

Die 3 Beispiele haben gezeigt, dass sich die Simulation vielseitig einsetzen lässt. Da man beliebige Anwendungen schreiben kann, die auf die Simulation zugreifen, sind der Phantasie beim Entwickeln von Test-Szenarien keine Grenzen gesetzt.

## 5 Zusammenfassung und Ausblick

### 5.1 Fähigkeiten der Simulationsumgebung

Mit Hilfe des package *simnet* hat man die Möglichkeit, das Verhalten eines Netzwerkes, das von mehreren Teilnehmern benutzt wird, zu simulieren. Genauer gesagt handelt es sich hierbei um ein Netzwerk, das auf der TCP/IP-Protokollreihe basiert. Abbildung 5.1 zeigt, welche TCP/IP-Protokolle simuliert werden und welche Schichten von TCP/IP damit abgedeckt werden. Man kann ebenfalls erkennen, auf welcher OSI-Schicht man sich jeweils befindet. Auf der Transportschicht wird das UDP-Protokoll simuliert, welches auf die Protokolle der unteren Schichten zugreift. Um eine Übertragung mit dem UDP-Protokoll zu simulieren, muss also mindestens jeweils eines der Protokolle auf den darunterliegenden Schichten simuliert werden. So kann man schließlich auf ein virtuelles Medium zugreifen. Auf der internet-Schicht beschränkt sich die Auswahl auf das Internet Protocol (IP), welches somit in jedem Fall simuliert werden muss. Auf der Netzwerkschicht wird das Ethernet Protocol simuliert, welches die CSMA/CD Arbitrierung und das Fehlererkennungsverfahren CRC verwendet. Anstelle der CSMA/CD Arbitrierung kann alternativ die reine Aloha Arbitrierung verwendet werden. CRC kann wahlweise ausgeschaltet werden. An den beiden letzten Optionen lässt sich erkennen, dass es sich um ein flexibles Konzept handelt, das über die Grenzen von TCP/IP hinaus erweitert werden kann. Als Übertragungsmedium auf der Bitübertragungsschicht simuliert das package *simnet* ein Kabel, für das man eine Übertragungsrate und eine Bitfehlerwahrscheinlichkeit festlegen kann.

OSI-Schichten	TCP/IP-Schichten	TCP/IP-Protokolle			
Anwendungsschicht (Application Layer)	Anwendungsschicht (Application Layer)	Telnet	FTP	SMTP	NFS
Darstellungsschicht (Presentation Layer)					
Sitzungsschicht (Session Layer)					
Transportschicht (Transport Layer)	Transportschicht (Transport Layer)	Transmission Control Protocol (TCP)		User Datagram Protocol (UDP)	
Vermittlungsschicht (Network Layer)	internet	Internet Protocol (IP)			
Sicherungsschicht (Data Link Layer)	Netzwerkschicht (Network Layer)	Ethernet	Token-Ring	andere	
Bitübertragungsschicht (Physical Layer)					

**Abbildung 5.1:** TCP/IP Protokollfamilie [REVS]

## 5.2 Einbettung neuer Protokolle

Da die Implementierung der Simulation nach einem modularen Konzept aufgebaut ist, entspricht jedes simulierte Protokoll einer Klasse im package *simnet*. Um ein neues Protokoll auf einer Schicht zu implementieren, muss also nur eine neue Klasse erzeugt werden, die dieses Protokoll repräsentiert. Alle Klassen, die Protokolle auf einer gleichen Schicht darstellen, müssen die gleiche Spezifikation aufweisen und lassen sich so problemlos austauschen.

Auf der Bitübertragungsschicht hat man so z.B. die Möglichkeit, weitere Zugriffsverfahren zu implementieren. Eine interessante Erweiterung wäre z.B. die Entwicklung von Klassen für kontrollierte Zugriffsverfahren wie Token-Ring oder Token-Bus.

Auf der Transportschicht befindet sich neben dem UDP Protokoll das TCP-Protokoll. Da Java mit Hilfe von *streams* über TCP kommunizieren kann, ist es auch hier möglich, einen Ansatz zu implementieren, der die Daten an die Simulation übergibt. Alle Klassen, die den Protokollen auf den unteren Schichten entsprechen, können dabei weiter verwendet werden. Will man die Simulation auch für TCP verwenden, muss man sich also nur noch mit der Arbeitsweise von TCP auseinandersetzen.

Um Protokolle auf den oberen Schichten, wie zum Beispiel Telnet, FTP oder SMTP zu simulieren, sind keine Änderungen in der Implementierung der Simulationsumgebung nötig, da die Simulation nur die unteren vier Schichten darstellt. Es müssen hier lediglich Applikationen geschaffen werden, die auf die Protokolle, welche auf der Transportschicht simuliert werden, zugreifen. Es lassen sich so bereits problemlos Protokolle simulieren, die auf den UDP Dienst zugreifen.

Da man alle Protokolle auch in das ISO/OSI Schichtenmodell einordnen kann, hat man prinzipiell die Möglichkeit, alle Protokolle die sich in dieses Modell einordnen lassen auch in die Simulation einzubetten. Man kann so auch andere Arten von Netzwerken, wie zum Beispiel Feldbussysteme simulieren. Bei Feldbussen werden auf einigen Schichten Protokolle verwendet, die denen der TCP/IP-Protokollreihe sehr ähneln. Es reicht so in einigen Fällen aus die vorhandenen Module anzupassen. Will man die Simulation in einer übergeordneten Simulation von Leitsystemen einsetzen, so muss man sich mit den hier verwendeten Protokollen im Detail auseinandersetzen.

## 5.3 Simulation komplexer Netzwerke

Eine deutliche Einschränkung der Simulation besteht darin, dass immer nur ein Kabel für sich betrachtet wird. Ein reales Netzwerk besteht aber fast immer aus mehreren Kabeln und verschiedenen Kommunikationsgeräten. Um die Simulation für diesen Zweck zu erweitern, sind sicher viele Ansätze möglich. Da sich die einzelnen Datenpakete in einem Netzwerk sehr schnell ausbreiten und so mehrere Kabel gleichzeitig beanspruchen können, reicht es nicht aus, mehrere Kabel unabhängig voneinander zu betrachten. Es muss also für die unterste Schicht eine Kontrollinstanz eingeführt werden, die einen Scheduling-Ansatz für mehrere Leitungen liefert. Diese Kontrollinstanz muss auch die Verwendung von Hubs und Repeatern simulieren, die ebenfalls auf der untersten Schicht anzusiedeln sind.

Kommunikationsgeräte auf den höheren Schichten lassen sich einfacher simulieren, da sie selbst Teilnehmer am Netzwerk darstellen und nach ihrer Implementierung nur an die Netzwerksimulation angeschlossen werden müssen.

## 5.4 Messung, Darstellung und Analyse

Zur Messung des Übertragungsverhaltens bietet die Simulation bisher nur wenige Werkzeuge. Es lässt sich lediglich die Übertragungsdauer eines Prozesses berechnen. Da die Simulation ein reales Übertragungsverhalten widerspiegelt, kann man Messungen auch „von außen“ durchführen, indem die Anwendungen, die auf das Netzwerk zugreifen, für die Erfassung der Messdaten sorgen. Eine sinnvolle Erweiterung der Simulation wäre die Entwicklung einer Messumgebung, die Folgendes leisten sollte:

- eine eindrucksvolle Darstellung des momentanen Netzwerkzustandes
- die automatische Erfassung mehrerer Messdaten
- die automatische Erstellung von Statistiken
- die Ausführung komplexer Berechnungen mit Hilfe der Messdaten

Gerade bei der Darstellung sollte hierbei nicht auf den Einsatz einer graphischen Benutzeroberfläche verzichtet werden, da die textuelle Ausgabe nur sehr schlecht den momentanen Übertragungszustand des Netzwerkes wiedergeben kann. Der Benutzer hätte so außerdem die Möglichkeit, interaktiv mit der Messumgebung zu arbeiten.

## 5.5 Nebenläufigkeit

Da die Simulation in der Regel auf einem 1-Prozessor System durchgeführt wird, findet man bei der Verwendung mehrerer Prozesse keine echte Nebenläufigkeit. Es entstehen so ständig kleine Verzögerungen, die sich auch auf das Übertragungsverhalten auswirken. Dieser fehlerhafte Einfluss hängt also von der Anzahl der Netzwerkteilnehmer ab, die somit beschränkt werden sollte. Ab einer gewissen Anzahl von Prozessen, die über das virtuelle Netzwerk kommunizieren, liefert die Simulation keine realistischen Ergebnisse mehr. Systeme mit mehreren Prozessoren bieten hier einen echten Vorteil, da man die Prozesse auf die einzelnen Prozessoren verteilen kann.

## Literaturverzeichnis

- [REVS] Bernd Schürmann: Rechnerverbindungsstrukturen. Bussysteme und Netzwerke. Vieweg Verlag 1997.
- [TCPIP] James Martin, Joe Leben. TCP/IP-Netzwerke. Architektur, Administration und Programmierung. Prentice Hall 1994.
- [LONE] Franz-Joachim Kauffels. Grundlagen, Standards, Perspektiven. DATACOM Buchverlag 1995.
- [MSMF] Ulrich A. Nickel, Jörg Niere. Modelling and Simulation of a Material Flow System.
- [GRUPP] Homepage Andreas Grupp.  
<http://www.elektronikschule.de/~grupp/index.html>
- [KONE] Prof. Dr. G. Timmer, FH Osnabrück. Kommunikationsnetze.  
<http://gtsun.et.fh-osnabrueck.de/lehre/kommunikationsnetze/kn-skript/node78.html>
- [RENE] Reiner Dumke, Achim Winkler, Universität Magdeburg. Rechnernetze Vorlesungsskript.  
<http://ivs.cs.uni-magdeburg.de/rn/rn1script/chapter4.html>