Technische Universität Braunschweig

Institut für Betriebssysteme und Rechnerverbund

Diplomarbeit

simuLan

Modellierung und Simulation lokaler Netzwerke

von cand. inform. Ralf Schmidt-Dannert

Aufgabenstellung und Betreuung :

Prof. Dr. H. Langendörfer

Braunschweig
22. Juli 1993

Zusammenfassung

In den vergangenen Jahren wurden immer mehr und komplexere Rechensysteme entworfen und eingesetzt. Die Erforschung leistungsfähiger offener Kommunikationssysteme ermöglichte ihren zunehmenden Einsatz in Bereichen wie Bürokommunikation, Betriebsdatenerfassung und CAD/CAM.
Mit der zunehmenden Akzeptanz dieser Systeme in öffentlichen, industriellen und wissenschaftlich-technischen Umgebungen steigt das Bedürfnis nach Leistungsbewertung und Leistungsvorhersage solcher Netzwerke. Die Modellierung und Simulation stellt ein vielfältiges Instrument für die Lösung der dabei auftretenden Fragestellungen dar.
In dieser Arbeit werden die Möglichkeiten, die das Simulationswerkzeug simuLan[1] zur effizienten Modellierung und Simulation lokaler Netzwerke bietet, vorgestellt.
Dazu wird ein auf dem *Ethernet* basierendes lokales Netzwerk mit mehreren angeschlossenen Rechnern modelliert. Das Modell ist in allen wichtigen Systemparametern frei konfigurierbar und ermöglicht die Bestimmung von Leistungswerten, wie Durchsatz, Verzögerungszeit u.a., in den verschiedenen, an dem ISO-Standard orientierten, modellierten Schichten. Die Realisierung des Modells durch eigenständige Module für die verschiedenen Schichten ermöglicht eine einfache Erweiterbarkeit und Wiederverwendbarkeit für weitergehende Untersuchungen.

Abstract

Over the last years more and more complex computing systems were developed and implemented. The examination of efficient open communication systems made their increasing use in fields like office communication and CAD/CAM possible.
With the increasing acceptance of such systems in public, industrial, scientific and technical environments the necessity of perfomance evaluation and performance prediction of such networks is growing.
In the present study the possibilities of the tool simuLan[2] for efficient modelling and simulation of lokal area networks are shown.
A model of a local area network based on the `Ethernet` with serveral connected computers is modeled and simulated. The values of all important system parameters are freely choosable. This modell allows the determination of performance values like throughput, delay, etc., in the various modelled layers, which are guided along the ISO-Standard. The realisation of this modell by independent moduls for the different layers makes extension and reuse for further investigations simple.

---

[1] simuLan wurde an der Technischen Universität Braunschweig entwickelt
[2] simuLan was developed at the technical university of Braunschweig

Inhaltsverzeichnis

Abkürzungen

CEP	: Connection Endpoint
CPU	: Central Processing Unit
CRC	: Cyclic Redundancy Check
CSMA	: Carrier Sense Multiple Access
CSMA/CD	: Carrier Sense Multiple Access with Collision Detection
DLL	: Data Link Layer
DUD	: Data Unit Descriptor
FCS	: Frame Check Sequence
FIFO	: First In First OUT
IEEE	: Institute of Electrical and Electronics Engeneers
ISO	: International Organization for Standardization
ITU	: Indivisible Time Unit
LAN	: Local Area Network
LIFO	: Last In First Out
LLC	: Logical Link Control
MAC	: Media Access Control
MAN	: Metropolitan Area Network
OSI	: Open Systems Interconnection
PDU	: Protocol Data Unit
SAP	: Service Access Point
SFD	: Start of Frame Delimiter

Kapitel 1

Einleitung

"So eine Arbeit wird eigentlich nie fertig, man muß sie für fertig erklären, wenn man nach Zeit und Umständen das Möglichste getan hat".
(J.W. Göthe, Italienische Reise, 16.03.1767)

In den vergangenen Jahren wurden immer mehr und komplexere Rechensysteme entworfen und eingesetzt. Die Erforschung leistungsfähiger offener Kommunikationssysteme ermöglichte ihren zunehmenden Einsatz in Bereichen wie Bürokommunikation, Betriebsdatenerfassung und CAD/CAM.
Mit der zunehmenden Akzeptanz dieser Systeme in öffentlichen, industriellen und wissenschaftlich-technischen Umgebungen steigt das Bedürfnis nach Leistungsbewertung und Leistungsvorhersage solcher Netzwerke.

Typische Fragestellungen, die sich damit ergeben, sind:

- „Ist es sinnvoll, die vorhandenen Rechner zu vernetzen, um neue Funktionalitäten oder eine bessere Ausnutzung der vorhandenen Ressourcen zu ermöglichen?"
- „Welches Netz mit welcher Topologie ist bei den gegebenen Rahmenbedingungen einsetzbar?"
- „Wo liegen in dem geplanten oder schon installierten System die kritischen Stellen in bezug auf Durchsatz und Wartezeiten?"

Die vorliegende Arbeit zeigt Lösungsansätze für die Beantwortung dieser und ähnlicher Fragestellungen auf - unter Verwendung des an der Technischen Universität Braunschweig entwickelten Simulationswerkzeugs simuLan [Pet91].
Dazu wird ein auf dem *Ethernet* basierendes lokales Netzwerk mit mehreren angeschlossenen Rechnern modelliert. Das Modell ist in allen wichtigen Systemparametern frei konfigurierbar und ermöglicht die Bestimmung von Leistungswerten, wie Durchsatz, Verzögerungszeit u.a., in den verschiedenen, an dem ISO-Standard orientierten, modellierten Schichten. Die Realisierung des Modells durch eigenständige Module für die verschiedenen Schichten ermöglicht eine einfache Erweiterbarkeit und Wiederverwendbarkeit für weitergehende Untersuchungen.

Im Gegensatz zu anderen Modellierungsprojekten s. [Mac87], [Lan88], [Gon85], [GR88a], [HL82] wird in dieser Arbeit Wert auf eine sehr realitätsgetreue Modellierung, besonders des `Ethernet`, gelegt.

Nach dieser kurzen Übersicht folgt im Kapitel 2 eine Einführung in das Gebiet der Leistungsanalyse unter besonderer Berücksichtigung der simulativen Modellierung. Daran anschließend wird in Kapitel 3 näher auf lokale Netze eingegangen. Im Kapitel 4 wird dann kurz das verwendete Simulationswerkzeug simuLan mit seinen Möglichkeiten anhand eines Modellierungsprojektes, in dem ein einfaches Netzwerk modelliert wird, vorgestellt.
Das nächste Kapitel (5) bildet den Schwerpunkt dieser Arbeit und beschreibt die Modellierung

des `Ethernet`. Dort wird zunächst das `Ethernet` beschrieben und anschließend ein Modell entwickelt. Nach Beschreibung der Implementaion werden eine Validierung durchgeführt und die zugehörigen Simulationsergebnisse angegeben. Den Abschluß bildet die Simulation eines modifizierten `Ethernet` mit einer Übertragungsrate von 20 Mbit/s.
Im Kapitel 6 wird ein einfaches Schichtmodell für die Protokollschicht *2* beschrieben und implementiert, daß den Zugang einer Station zum Netz serialisiert.
Das nachfolgende Kapitel 7 beschreibt ein allgemeines Modell für verschiedene Rechnertypen. Nach der Beschreibung des dazugehörigen Moduls, wird eine Verifikation durchgeführt.
Das Kapitel 8 faßt die bei der Modellierung und Simulation mit simuLan gemachten Erfahrungen zusammen.
Der Anhang enthält das Glossar, die Indizes und eine Auflistung aller ‚**C**'-Quelldateien die im Rahmen dieser Arbeit entwickelt wurden. Die dazugehörigen Programmquellen sind auf der beigefügten Diskette enthalten.

Kapitel 2

Leistungsanalyse in Rechnensystemen

Dieses Kapitel gibt eine kurze Einführung in das Gebiet der Leistungsanalyse von Rechnersystemen (-netzen). Für weitergehende Informationen zu diesem Gebiet verweise ich auf [Agr85], [Fer78], [Fis73], [Fis78], [Kob78], [Lav83], [Mee86], [Pid89], [SC81].

2.1 Überblick über die Leistungsanalyse von Rechensystemen

Bei der Entwicklung und Planung von Rechensystemen sind Leistungsabschätzungen von großer Wichtigkeit, um kostspielige Fehlentwicklungen zu vermeiden. Durch die immer weiter zunehmende Komplexität der heutigen Rechnensysteme und den Einsatz in neuen Bereichen gewinnt das Gebiet der *Leistungsanalyse* (*Performance Analysis*) deshalb immer mehr an Bedeutung.

> Die Leistung eines Systems ist einer der Schlüsselfaktoren, die bei Entwurf, Entwicklung, Konfigurierung und Tuning von Computersystemen berücksichtigt werden müssen. [Lav83]

Die Leistungsfähigkeit eines Rechensystems ist über *Leistungsgrößen* beschreibbar. Das Ziel der Leistungsanalyse ist deshalb die Bestimmung dieser Größen, die eine Analyse und Bewertung des Systems zulassen. Einige wichtige Leistungsgrößen sind Durchsatz, Auslastung und mittlere Antwortzeiten.

In der Leistungsanalyse werden zwei grundsätzliche methodische Ansätze unterschieden:

- Messung
- Modellierung

Nach Fertigstellung eines bestimmten Rechensystems kann die Leistung über die direkte Messung mit Hardware-, Software- und Hybridmonitoren bestimmt werden. Für die Leistungsanalyse eines Systems oder einer Komponente bei dem/der eine Messung nicht möglich ist, z.B. während der Planungs- und Entwicklungsphase, muß man entweder auf Erfahrungswerte zurückgreifen oder *Leistungsmodelle* des Systems entwickeln. In den heutigen Rechensystemen sind die Wechselwirkungen so komplex, daß nur Formen der Modellierung eine Vorhersage der Leistungsfähigkeit ermöglichen. Ein großer Vorteil der Verwendung von Leistungsmodellen ist, neben den ermittelten quantitativen Vorhersagen, der Einblick in die Struktur und das Verhalten des Systems. [Lav83]

Der folgende Abschnitt geht etwas genauer auf die Modellierung von Rechensystemen ein, wobei zunächst die Begriffe Rechensystem und Modell erläutert werden und anschließend näher auf den Prozeß der Modellbildung und Analyse eingegangen wird.

2.2 Modellierung von Rechensystemen

Definition:

> Ein *Rechensystem* (kurz: *System*) ist eine Menge von miteinander verbundenen Hardware- (Zentral- und I/O-Prozessoren, Speicher, Peripherie, Schnittstellen) und Software-Komponenten (die Programme, welche das Betriebssystem ausmachen). Diese Komponenten werden *Ressourcen* genannt. Jede Komponente besitzt seine eigenen Attribute, die *Systemparameter*, auch wenn diese nicht durch eine einzelne Zahl ausgedrückt werden können. Ein Beispiel ist der Algorithmus, nach dem eine Komponente arbeitet. [Fer78]
>
> Ein *Modell* ist eine Abstraktion eines Systems und steht für eine bestimmte Sicht eben dieses Systems. Die Beschreibung des Modells geschieht durch Ausdrücke, die spezifisch für die verwendete Modellierungsmethode sind. [Mac87]

Die Methode der Modellierung ist in allen Stadien des Rechnersystementwurfs einsetzbar, da die Modellbildung unabhängig von der Verfügbarkeit des untersuchten Systems ist.

In der Modellierung werden zwei verschiedene Ansätze verwendet:

- *analytische* Modellierung: Das System wird durch eine Menge von Gleichungen beschrieben, die sich mit mathematischen Verfahren lösen lassen. Oftmals kann man Systeme in Komponenten zerlegen, für die die Lösung der beschreibenden Gleichungen bekannt ist. Dieses Verfahren kann allerdings nur für ,,relativ" einfache Systeme verwendet werden, da sich die Wechselwirkungen in komplexeren Systemen nicht mehr ohne zu weitgehende Vereinfachungen mit mathematischen Gleichungen geschlossen darstellen lassen.
  Der große Vorteil der analytischen Methode gegenüber der simulativen liegt in dem deutlich geringeren Rechenzeitbedarf.
- *simulative* Modellierung: Das dynamische Verhalten der Komponenten des zu untersuchenden Systems und ihr Zusammenwirken werden durch Computer-Programme beschrieben. Diese Methode wird in der vorliegenden Arbeit angewandt, da nur diese die für die zu analysierenden Systeme benötigte realitätsgetreue Modellierung ermöglicht.
  Problematisch ist bei diesem Verfahren die Laufzeit der Simulation, sobald komplexere Systeme mit vielen Wechselwirkungen nachgebildet werden sollen.

Hybrid-Modelle kombinieren die Vorteile der analytischen Modelle mit denen der Simulation, indem einzelne Komponenten durch ihre Kenngrößen bzw. deren Verteilungsfunktionen beschrieben werden, ohne näher auf ihr ,,Innenleben" einzugehen.

Ein Simulationssprojekt kann in folgende Phasen (s. Abb. 2.1) unterteilt werden:

Systembeschreibung
$\downarrow$
Abstraktion und Modellbildung
$\downarrow$
Festlegen der Eingabedaten
$\downarrow$
Entwicklung des Simulationsprogramms
$\downarrow$
Programm-Debugging
$\downarrow$
Verifikation (Programm gegen Modell)
$\downarrow$
Validierung (Modell gegen System)
$\downarrow$
Analyse der Ausgabewerte
$\downarrow$
Problemanalyse

Abbildung 2.1: Der Modellierungs- und Analyseprozeß (aus [Mac87]).

Systembeschreibung: Hier wird das zu modellierende System mit allen relevanten Parametern beschrieben.

Abstraktion und Modellbildung : An diesem Punkt wird eine Darstellung des Systems entwickelt, die alle für die Leistungsanalyse wichtigen Parameter berücksichtigt. Dabei wird aber noch nicht die Methode der Modellierung festgelegt.

Festlegen der Eingabedaten: Für die Modellparameter werden die benötigten Werte entweder durch Beobachtung eines realen Systems oder anhand von Erfahrungswerten ermittelt.

Entwicklung des Simulationsprogramms: Hierbei sind folgende Punkte besonders zu beachten:

- Entwurf des Simulationsmodells
- Programmorganisation
- Verwaltung der Parameter
- Tools für Debugging
- Instrumentierung.

Programm-Debugging:

Verifikation: An dieser Stelle wird geprüft, ob das Simulationsprogramm auch wirklich das System in der gewünschten Art nachbildet. Dazu gehört einerseits die Überprüfung des Datenflusses im Modell und andererseits ein Vergleich mit evtl. Ergebnissen analytischer Modellierung.

Validierung: Hier muß gezeigt werden, daß das Modell eine ausreichend genaue und korrekte Darstellung des realen Systems ist.

Analyse der Ausgabewerte: Auswertung (z.B. graphisch) der vom Modell gelieferten Werte $\rightarrow$ Erkennung von Problemen des modellierten Systems.

Problemanalyse: Eventuelle Probleme im modellierten System die durch die Simulation erkannt wurden, analysieren, beheben und gegebenenfalls das veränderte System erneut simulieren.

Kapitel 3

Lokale Netze

> "At the beginning of the 1990s, it is becoming clear that the computing environment will continue to undergo fundamental changes, driven primarily by technological progress. Roughly speaking, we can divide the history of modern computing into eras:
>
> - 1970s: Timesharing (1 computer with many users)
> - 1980s: Personal computing (1 computer per user)
> - 1990s: Parallel computing (many computers per user)."
>
> (A.S. Tanenbaum, im Intro zur Dokumentation des verteilten Betriebssystems "Amoeba")

Noch vor wenigen Jahren war das Konzept für eine Mehrbenutzer-Datenverarbeitungsanlage die Installation eines mehr oder weniger leistungsfähigen „Großrechners", an den „dumme" Terminals angeschlossen wurden. Mit der Verfügbarkeit leistungsfähiger, preisgünstiger Personal Computer und Workstations sowie der Entwicklung bzw. Verbesserung der benötigten Kommunikations-Hardware (Treiber- und DMA-Prozessoren) hat sich dieses Konzept dahingehend gewandelt, daß anstelle eines großen mehrere „kleine" Rechner eingesetzt werden. Die Vorteile werden insbesondere bei speziellen Anwendungen, wie z.B. Grafik, Textverarbeitung oder Datenbankmanagement, deutlich.

Diese „kleinen" Rechner stehen über ein Kommunikationsnetz miteinander in Verbindung. Dessen wichtigste Kennzeichen sind [Lan83]:

- geringe räumliche Ausdehnung
- hohe Übertragungsraten
- niedrige Fehlerraten.

Neben dem Datenaustausch zwischen den unabhängigen Systemen soll mit diesem Netz auch der `gemeinsame` Zugriff auf alle vorhandenen Ressourcen (Platten, Plotter, Drucker, Spezialrechner ...) sowie eine Lastverteilung ermöglicht werden.
Kommunikationsnetze mit den vorher beschriebenen Charakteristika bezeichnet man als *Lokale Netze (Local Area Networks, LAN).*

Im weiteren werden zunächst der Aufbau und die Funktionsweise lokaler Netze skizziert, und anschließend wird genauer auf das ISO/OSI-Referenzmodell und die Protokollschichten des `ARPA Internet` eingegangen.

3.1 Überblick über lokale Netze

Definition:

> „*Lokale Netze sind Datenkommunkiationssysteme, welche einer Anzahl von unabhängigen Einrichtungen eine zumeist partnerschaftlich orientierte Kommunikation hoher Datenrate auf relativ begrenztem geographischen Gebiet ermöglicht.*" (aus [Lan83])

LAN's zeichnen sich aus durch:

- eine geographische Ausdehnung von einigen 10 Metern bis zu einigen Kilometern
- einen Kommunikationskanal mittlerer oder hoher Datenübertragungsrate (1Mbit/s bis zu wenigen 100 Mbit/s)
- eine niedrige Fehlerrate (Bitfehlerwahrscheinlichkeit: 10^{-8} bis 10^{-12})
- einfache Möglichkeiten, das Netz um weitere Rechner, Peripherie zu erweitern
- den Besitz und Gebrauch durch eine einzelne Organisation
- einfache und flexible Erweiterbarkeit
- einfacher Netzzugang für den Benutzer.

Haupteinsatzgebiete lokaler Netzwerke sind die Bürokommunikation, die Betriebsdatenerfassung sowie der Verbund verschiedener Rechner und ihrer Peripherie. Letzteres wird durch die Normungsbestrebungen der vergangenen Jahre stark gefördert.

Detaillierte Informationen über die angesprochenen Aspekte lokaler Netze, die den in diesem Abschnitt gebotenen Umfang überschreiten, sind z.B. [IEE88a], [IEE88b], [IEE88c], [IEE85], [IEE89], [Lan83], [Tan89], [Com88] zu entnehmen.

3.2 Übertragungsmedien

Infolge der geringen Netzausdehnung können Übertragungsverfahren mit einer hohen Datenrate verwendet werden. Einsatz als Übertragungsmedium in lokalen Netzen finden vornehmlich

- das traditionelle verdrillte Kupferkabel (twisted pair)
- das Koaxialkabel (mit Basisband- und Breitbandverfahren)
- die Glasfaser, die hohe Datenraten (bis 10 GBit/s) und große zu überbrückende Entfernungen (bis zu 100 km) zuläßt.

3.3 Netztopologien

Lokale Netze bestehen aus einzelnen Stationen (Rechnern), die durch Übertragungseinrichtungen miteinander verbunden sind. Abb. 3.1 zeigt die wichtigsten heute verwendeten Topologien lokaler Netze.

Bei einem *sternförmigen lokalen Netz* laufen alle Übertragungen über die Zentralstation, die als Steuerstation (Monitorstation) dient und die anfallenden Vermittlungs- und Verteilungsaufgaben übernimmt. Nachteilig sind die geringe Ausfallsicherheit aufgrund der zentralen Stellung der Monitorstation, die Begrenzung des Gesamtdurchsatzes sowie die hohen Anschaffungskosten für den Zentralrechner.

Bei einem *ringförmigen lokalen Netz* erfolgt der Datenaustausch `unidirektional` über Leitungen, die jeweils zwei direkt benachbarte Stationen verbinden. Die Übertragungssteuerung und

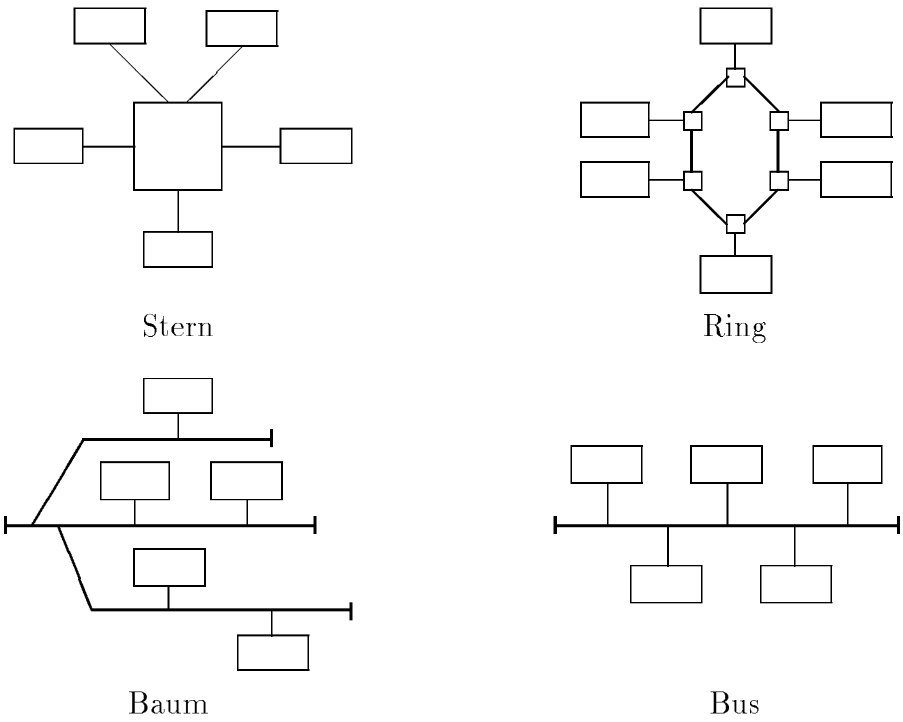


Abbildung 3.1: Topologien für lokale Netzwerke

-überwachung in einem Ring ist dezentral organisiert, so daß die Teilnehmer ständig aktiv sein müssen, um ankommende Nachrichten zu empfangen und weiterzuleiten. Diese Topologie wird in letzter Zeit bei Verwendung der Glasfaser als Übertragungsmedium bevorzugt eingesetzt. Um die Ausfallsicherheit zu erhöhen, werden meistens doppelte Ringe benutzt. Nachteilig bei Ringen ist der hohe Aufwand bei der Fehlererkennung und -behandlung.

Bei der *Bus-Topologie* findet der Datenaustausch über ein einziges Übertragungsmedium, an das alle Stationen passiv gekoppelt sind, statt. Die Stationen werden nur im Sendefall aktiviert. Dadurch ist eine Erweiterung des Netzes um weitere Rechner während des laufenden Betriebs problemlos möglich. Nachteilig ist bei dieser Topologie der Aufwand für das Zugriffsverfahren.

Die *Baum-Topologie* ist eine Erweiterung der Busstruktur. Mehrere Netzwerke werden dabei miteinander verbunden.

3.4 Zugriffsverfahren

Bei paralleler Nutzung eines gemeinsamen Übertragungsmediums durch mehrere Stationen müssen Verfahren zur Koordination des Netzzugangs implementiert werden. Es wird zwischen deterministischen und Random-Access Verfahren unterschieden. [Lan83]

deterministische Verfahren: Alle sendewilligen Stationen erhalten bei diesem Zugriffsverfahren in „fairer" Weise eine Sendeberechtigung zum Netz, d.h. jeder Benutzer kann nach endlicher Wartezeit senden. Konflikte durch sich gegenseitig störende Übertragungen können nicht auftreten.
Ein typisches Beispiel ist das *Token-Verfahren.*

Random-Access Verfahren: Die Benutzer haben bei diesem Verfahren jederzeit Zugriff auf das Übertragungsmedium. Hierdurch können Konflikte entstehen, wenn mehrere Stationen gleichzeitig auf das Medium zugreifen. Das Konzept von Random-Access-Zugriffsverfahren muß also den Zugang zum Übertragungskanal, die Konflikterkennung und -bereinigung umfassen.
Ein typisches Beispiel ist das *CSMA/CD (Carrier Sense Multiple Access with Collission Detection)*-Verfahren. Auf den bekanntesten Vertreter dieses Verfahrens, das *Ethernet*, wird in Kapitel 5 noch näher eingegangen.

3.5 Normen

Das zentrale Normungsdokument für Zugriffsprotokolle lokaler Netze ist der IEEE (Institute of Electrical and Electronics Engeneers) Standard 802 , der weitgehend von der ISO (International Organization for Standardization) übernommen wurde. Er umfaßt im Augenblick die Dokumente 802.1 bis 802.5. Eine Erweiterung um den Standard 802.6 (metropolitan area networks, MAN) ist in Vorbereitung. Die Beziehung der verschiedenen Standards zueinander ist aus Abb. 3.2 zu ersehen.

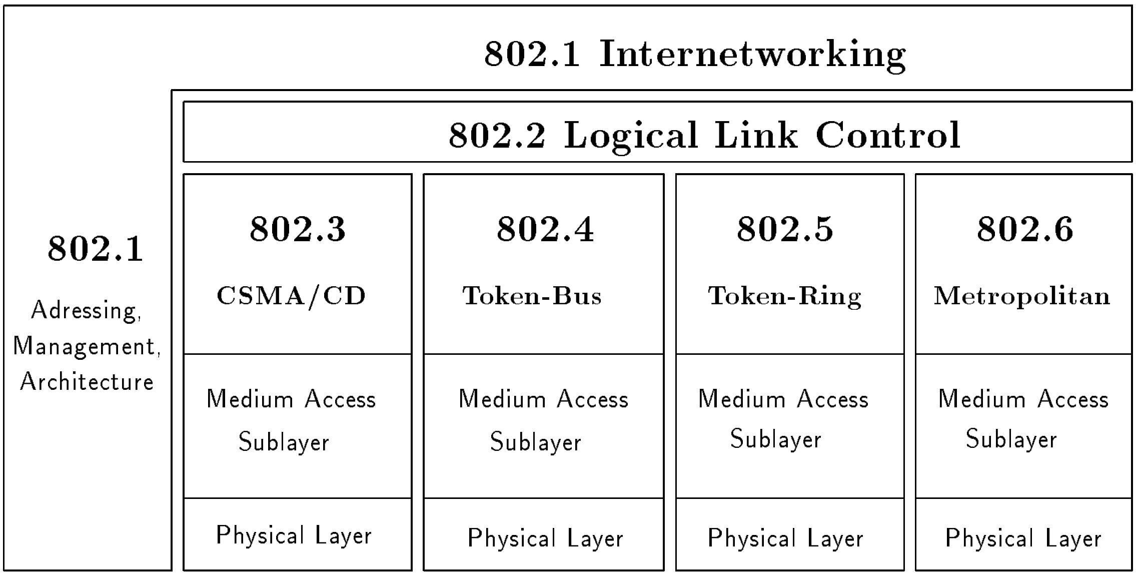


Abbildung 3.2: Die Familie der IEEE 802-Standards (aus [Kel86])

802.1: Hier wird eine Übersicht über den Aufbau des Standards gegeben und die Unterteilung der ISO-Schicht 2 (*data link layer, DLL*) in die zwei Teilschichten *logical link control (LLC)* und *media access control (MAC)* begründet. Diese Unterteilung ist notwendig, da sich einerseits verschiedene Zugriffsverfahren mit unterschiedlichen Topologien etabliert haben, andererseits die Funktionalität des DLL unabhängig vom verwendeten Zugriffsprotokoll sein soll. Die Abb. 3.3 Seite 10 zeigt diese Aufteilung am Beispiel des Standards IEEE802.3.

802.2: Dieses Dokument enthält Normungsvorschläge für das `LLC`, das als Schicht *2b* bezeichnet wird. Es werden drei verschiedene Protokolltypen durch die Angabe von Funktionsumfang und Dienstprimitiven spezifiziert.

802.3: Hier wird das Zugriffsverfahren *CSMA/CD* spezifiziert, das auf einem allen Stationen zugänglichen Bus arbeitet. Auf dieses Verfahren wird in Kapitel 5 noch näher eingegangen.

802.4: Diese Norm beschreibt das *Token-Bus*-Zugriffsverfahren, das als Übertragungsmedium ebenfalls einen allgemein zugänglichen Bus benutzt. Der Zugriff erfolgt im Gegensatz zu `CSMA/CD` allerdings nicht Random-Access, sondern deterministisch. Die angeschlossenen Stationen werden in einem logischen Ring angeordent, d.h. jeder Station wird ein Nachfolger zugewiesen. Auf dem logischen Ring kreist ein *Token* (ein spezielles Bitmuster), dessen Besitz einer Station die Sendeberechtigung zuteilt. Liegt bei Empfang des Tokens ein Sendewunsch vor, so beginnt die Station zu senden, andernfalls wird das Token sofort an den Nachbarknoten weitergegeben. Nach Beendigung der Übertragung generiert die Sendestation ein neues Token und gibt dieses an den Nachfolger weiter.

802.5: Das hier beschriebene *Token-Ring*-Zugriffsverfahren arbeitet ähnlich dem Token-Bus. Da das Übertragungsmedium als Ring organisiert ist, muß der Nachfolger der Station nicht explizit bekannt sein.
Im Gegensatz zum Token-Bus arbeitet jede Station ,,passiv", indem sie ankommende Signale auf das nächste Ringsegment weiterleitet. Nur bei Empfang des Tokens und

vorliegendem Sendewunsch sendet eine Station eigene Daten und „vernichtet" sie nach einem Ringumlauf wieder. Ein weiterer Unterschied ist, daß das Token nicht vom Ring genommen und anschließend neu generiert wird, sondern ständig als Frei- oder Besetzt-Token auf dem Ring kreist. Diese Unterscheidung wird durch Invertierung eines Bits im Token durchgeführt.

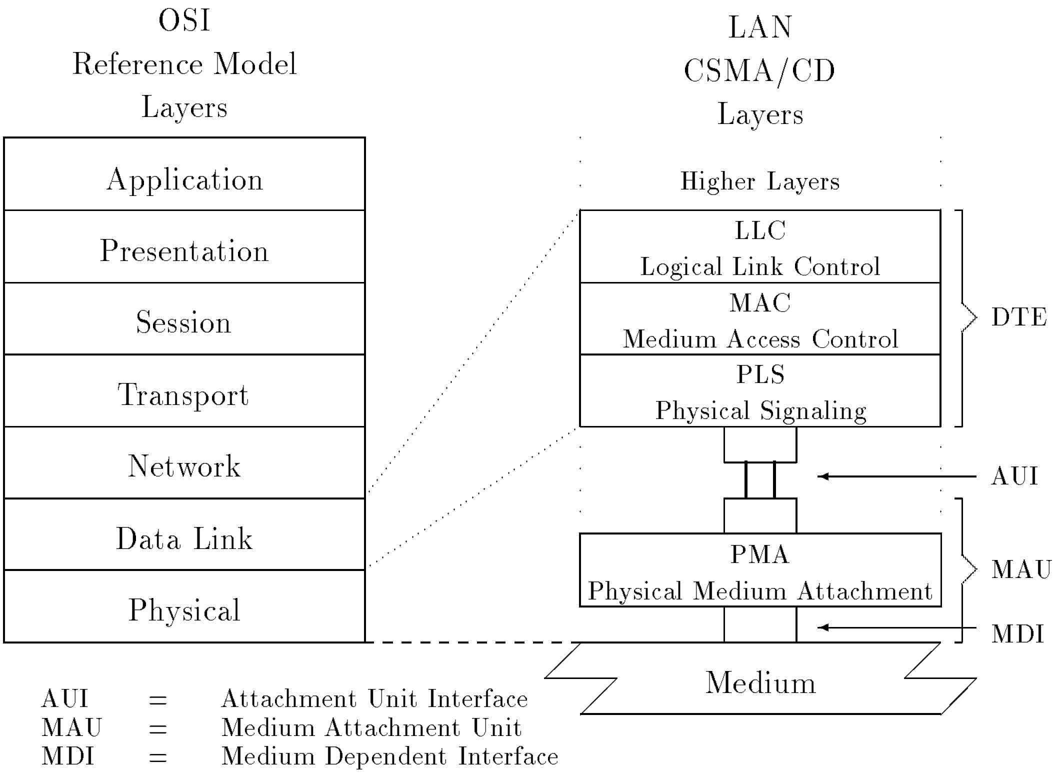


Abbildung 3.3: Service Specifikation Relations to the LAN Model (aus [IEE88c])

3.6 Das ISO/OSI-Referenzmodell

Um die Komplexität von Netzwerkentwürfen zu verringern, sind die meisten Netzwerke in *Funktionschichten (Layer)*, im folgenden kurz *Schichten* genannt, organisiert, wobei jede dieser Schichten auf der vorhergehenden aufbaut. Die Anzahl der Schichten, der Name und die Funktion jeder Schicht sind von Netzwerk zu Netzwerk verschieden. Für jedes Netzwerk gilt allerdings, daß es die Aufgabe einer Schicht *N* ist, der Schicht *N+1* bestimmte Dienste anzubieten und so diese von den Details, wie die angebotenen Dienste real implementiert sind, abzuschirmen.

Die Funktionen der Schicht N werden durch eine oder mehrere identische N-Instanzen (N-entities) geleistet. Eine N-Instanz bietet ihren Funktionsumfang der Schicht *N+1* in Form von Diensten (Services) an und benutzt ihrerseits Dienste der Schicht *N-1*. Die Dienste sind nur über sogenannte *Dienstzugangspunkte (Service Access Points, SAP)* zugänglich.

Der Ablauf der Kommunikation zwischen N-Instanzen, die zu verwendenden Datenformate und die Zuordnung Dienstprimitiv - PDU (s. u.) werden in einem *N-Protokoll* beschrieben. Der äußere Rahmen eines Protokolls ist durch die Schnittstellen bzw. Dienstprimitive vorgegeben.

Die physikalische Verbindung zweier Systeme ermöglicht eine direkte Verbindung auf der Schicht *1*. Durch den hierarchischen Aufbau kann auch eine N-Instanz mit ihrer *Partnerinstanz (peer-entity)* - nach einem festgelegten *Schichtenprotokoll* - logisch kommunizieren, indem sie die direkte Kommunikation mit der Schicht *N-1* benutzt. Diese Interaktion zwischen benachbarten Schichten wird durch Dienstprotokolle geregelt.

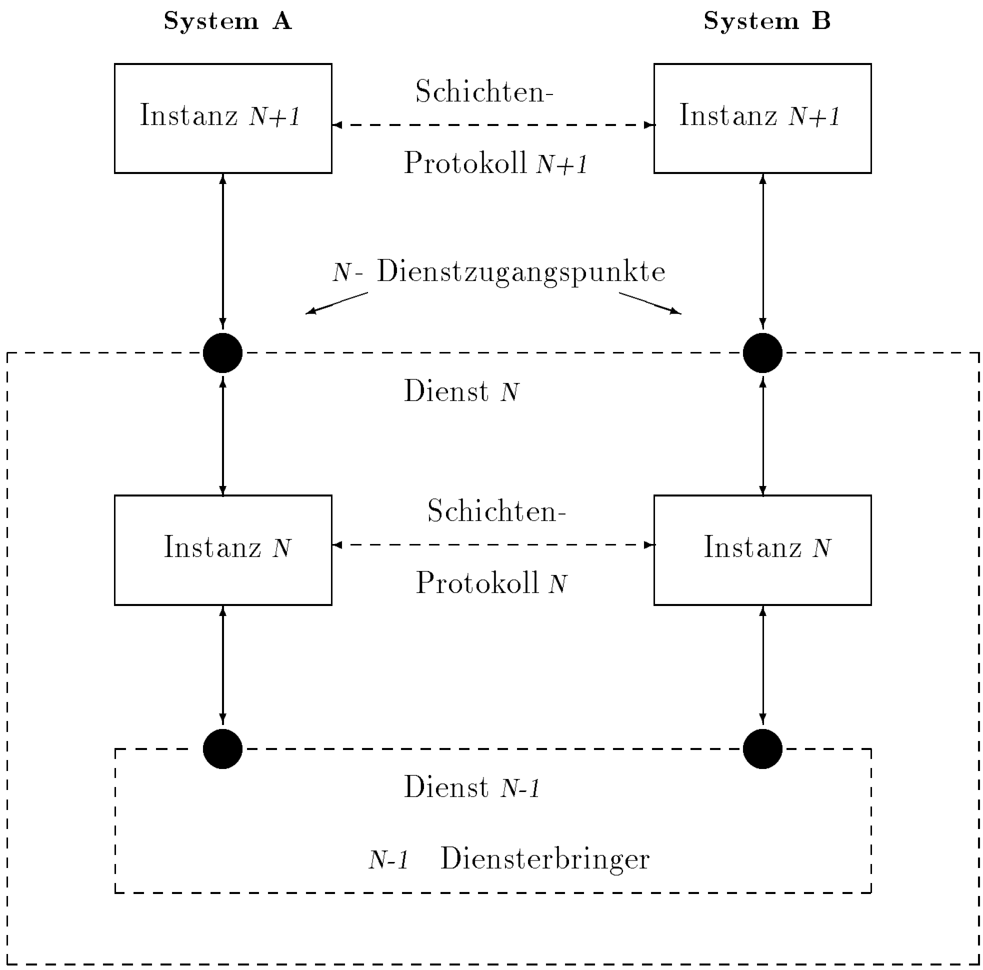


Abbildung 3.4: Modell einer Protokollschicht und ihrer Interaktion (aus [GKS+85])

Die Datenübertragung zwischen peer-entities erfolgt über *Protokolldateneinheiten(protocol data units, PDU)*, die von den tieferliegenden Schichten nicht interpretiert werden. Die Datenübertragung ist also transparent.
Das Ende einer Verbindung zwischen Partner-Instanzen bildet innerhalb der SAP's ein *Verbindungsendpunkt*, (*Connection Endpoint*, *CEP*).

Das von der *ISO* vorgeschlagene *ISO/OSI (Open Systems Interconnection) Reference Model* (ISO7498: Basic Reference Model for Open Systems Interconnections) ordnet jeder der in diesem Modell enthaltenen 7 Schichten (s. Abb. 3.5) bestimmte Aufgaben zu.
Das Modell soll dazu beitragen, den Informationsaustausch zwischen Systemen zu reglementieren und damit technologische Unterschiede verschiedener Systeme zu überwinden. Es liefert weder eine exakte Beschreibung der Dienste einer Schicht noch eine genaue Definition der zu verwendenden Protokolle zwischen diesen.

Den einzelnen Schichten werden folgende Aufgaben zugeordnet:

1) *Physikalische Schicht (Physical Layer)*: Spezifizierung von Spannungen, Strömen, Medium, zeitlichem Verhalten und Steckernormen. Es wird eine transparente, ungesicherte Übertragung von Bit-Folgen zur Verfügung gestellt.

2) *Leitungsschicht (Data Link Layer)*: Es werden die Formate und Verfahren des Datenaustausches festgelegt. Hauptaufgabe ist die Sicherung der Bit-Folgen gegen Übertragungsfehler, die bei physikalischen Übertragungen auftreten können. Bei lokalen Netzen ist diese Schicht von großer Bedeutung, da sie auch den Zugriff auf das Übertragungsmedium verwaltet.

3) *Netzwerkschicht (Network Layer)*: In dieser Schicht wird das Routing realisiert, welches

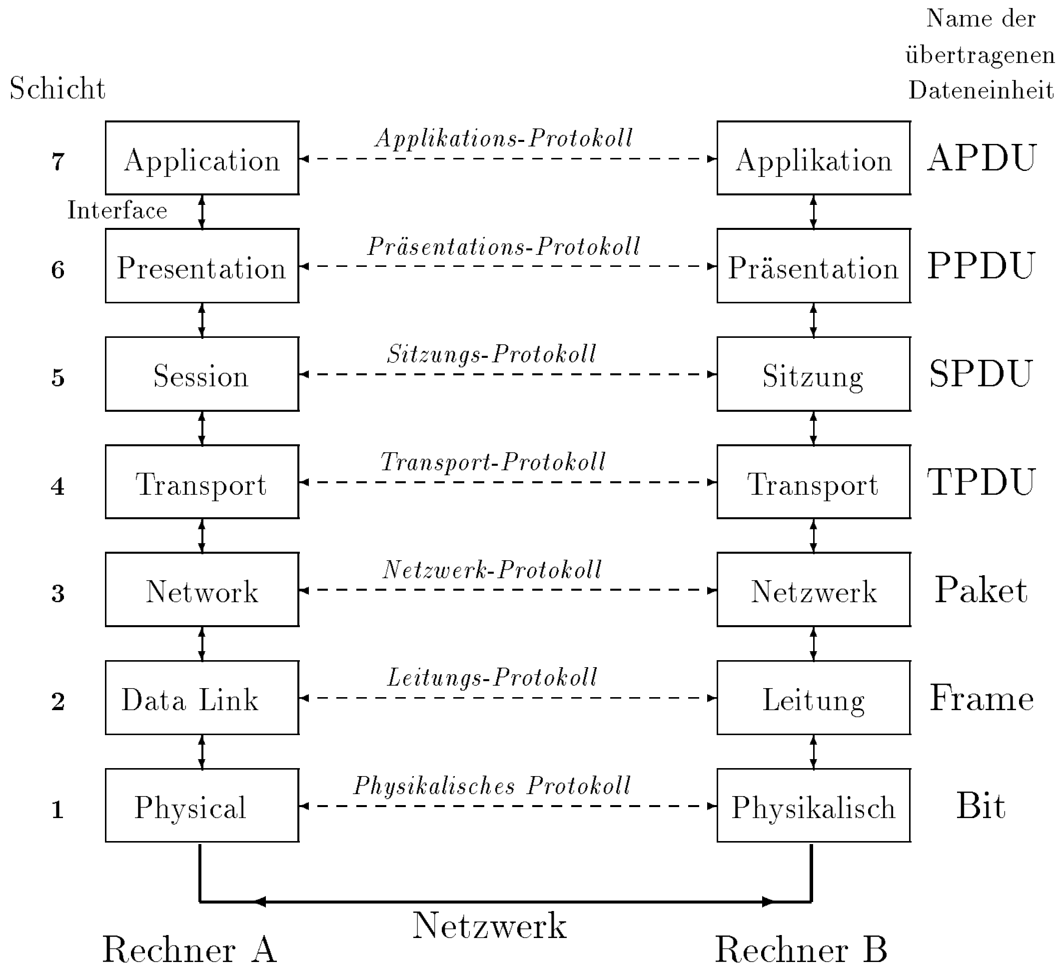


Abbildung 3.5: Das ISO/OSI-Referenzmodell

die Kommunikation zweier Rechner über ein ganzes Netz ermöglicht. Bei lokalen Netzen entfällt das Routing.

4) *Transportschicht (Transport Layer)*: Die Transportschicht stellt der Schicht 5 eine fehlerfreie, effiziente Ende-zu-Ende Kommunikation zur Verfügung. Hier werden auf der Senderseite zu große Nachrichten in Pakete zerlegt und im Empfänger wieder zusammengesetzt. Zusätzlich sind Verfahren der Flußkontrolle in dieser Schicht implementiert.

5) *Sitzungsschicht (Session Layer)*: Die Sitzungsschicht bietet Dienste zur Eröffnung, geordneten Durchführung und Beendigung einer Sitzung; sie synchronisiert die miteinander kommunizierenden Instanzen.

6) *Darstellungsschicht (Presentation Layer)*: Es werden Aufgaben der Gerätesteuerung und Anpassungen an gerätespezifische Eigenschaften realisiert. Typische Beispiele sind die Codeumwandlung EBCDIC $\leftrightarrow$ASCII und die Verschlüsselung von Daten.

7) *Anwendungsschicht (Application Layer)*: Hier sind Dienste wie *elektronische Post*, *Dateientransfer* und *virtuelle Terminals* angesiedelt. Diese Dienste werden von den Anwenderprozessen direkt genutzt.

3.7 Protokollschichten im ARPA Internet

Das im ARPA Internet verwendete Schichtenmodell wurde nicht von einem Standardisierungskommitee vorgeschlagen, sondern es entwickelte sich aus den Anforderungen der Praxis.
Aus Abb. 3.7 Seite 13 sind die fünf Schichten und die zwischen ihnen ausgetauschten Dateneinheiten ersichtlich.

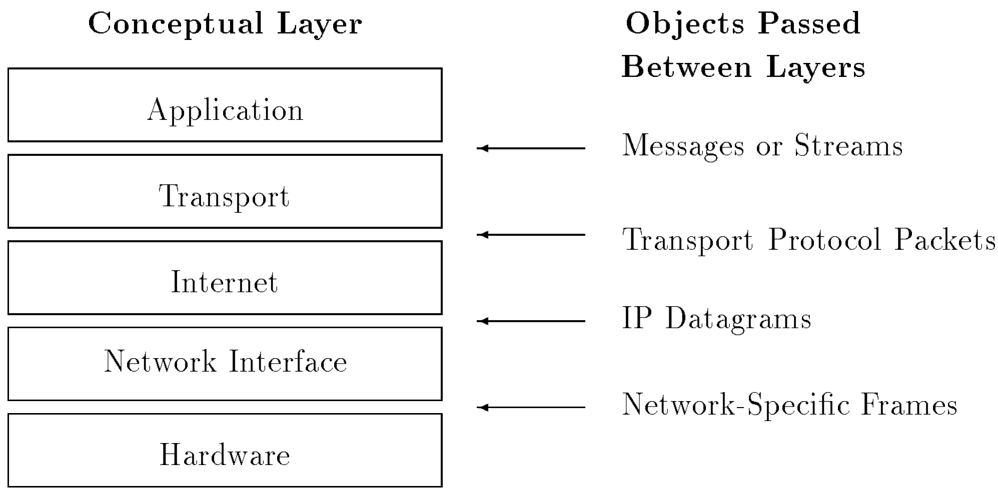


Abbildung 3.6: The 5 conceptual layers of hardware and protocol software used in the Internet, and the form of objects passed bewtween layers. The layer labeled *Network Interface* is sometimes called the *data link* layer. [Com88], pg. 108

Application Layer: Die Benutzerprogramme verwenden die Transportschicht, um Daten zu senden und zu empfangen. Jedes Benutzerprogramm verwendet ein eigenes Datenformat, entweder als Folge von Bytes oder als Folge von einzelnen Nachrichten.

Transport Layer: Diese Schicht realisiert eine fehlerfreie Ende-zu-Ende Verbindung zwischen der Senderapplikation und der Empfängerapplikation. In dieser Schicht werden Elemente wie *Acknowledges*, *Timer*, *Retransmit* und *Window Flow Control* implementiert. Der zu sendende Datenstrom wird in *Pakete* aufgeteilt. Jedes Paket erhält zusätzliche Informationen die angeben, welches die sendende und welches die empfangende Applikation ist. Ein Bsp.-Protokoll ist ,,TCP".

Internet Layer: Diese Schicht behandelt die Kommunikation zwischen zwei Rechnern. Sie akzeptiert Anfragen von der Transport-Schicht, ein *Paket* zu senden. Das *Paket* wird in ein IP *Datagramm* verpackt und dann entsprechend dem Wegfindungsalgorithmus an das entsprechende Netzwerk-Interface übergeben. Außerdem werden ankommende *Datagramme* geprüft und die enthaltene Information wird extrahiert. Zusätzlich werden *ICMP* Messages in dieser Schicht behandelt. Das Protokoll dieser Schicht ist ,,IP".

Network Interface Layer: Diese Schicht ist verantwortlich für das Senden und Empfangen von IP *Datagrammen*. Hier werden die *Datagramme* in netzwerkspezifische *Frames* verpackt. Das heißt, daß beim Sender evtl. eine zusätzliche Aufsplittung eines *Datagramms* in mehrere *Frames* nötig werden kann, da Datagramme bis zu 2^{16} Bytes groß sein dürfen. Falls ein *Frame* während der Übertragung zerstört wird, muß das komplette dazugehörige *Datagramm* von der Schicht **4** erneut übertragen werden.

Hardware: Diese Schicht spezifiziert die pysikalische Verbindung einschließlich elektrischen Parametern, wie Spannung und Strom.

Ein Unterschied zwischen dem ISO- und dem Internet-Modell liegt in der Realisierung der zuverlässigen Datenübertragung. Im ISO-Modell wird auf allen Protokollschichten eine zuverlässige Übertragung mit Prüfsummen, Acknowledgements, Timeouts und Retransmits angestrebt.
Im Internet-Modell wird Zuverlässigkeit als Aufgabe der Ende-zu-Ende-Übertragung in der Transportschicht angesehen. Dadurch vereinfacht sich das Verständnis und die Implementation der Protokollsoftware erheblich.

In [Com88] (Kapitel 10) vergleicht D. Comer die beiden Schichten-Konzepte. Kapitel 1 enthält einen Überblick über Geschichte und Organisation des ARPANET / Internet.

Kapitel 4

Das Modellierungstool simuLan - eine praktische Einführung

4.1 Einführung

In diesem Kapitel wird das in den späteren Kapiteln benötigte und vorausgesetzte Wissen über die grundlegenden Konzepte der Simulation mit simuLan anhand eines einfachen, aber vollständigen Simulationsprojektes vermittelt.

simuLan wurde an der TU Braunschweig entwickelt. Dieses Tool ist auf die Simulation lokaler Netzwerke zugeschnitten und ermöglicht die Modellierung aller üblichen Topologien (*Bus, Ring, Stern*), solange es sich um ein einzelnes Netz handelt und nicht um einen Netzverbund.

Einige wichtige Eigenschaften sind:

- Arbeitet nach dem Prinzip der ,,Discrete Event Simulation". Dabei wird angenommen, daß Zustandsänderungen des Systems durch einzelne *Ereignisse* ausgelöst werden. Jedes Ereignis tritt zu einer bestimmten Zeit ein, wodurch die Reihenfolge ihrer Bearbeitung festgelegt ist. Die aktuelle Simulationszeit wird jeweils auf die Zeit des nächsten zu bearbeitenden Ereignisses gesetzt und die Zustandsänderungen gemäß der Ereignisbeschreibung berechnet. Dabei werden ggf. neue, zukünftige Ereignisse erzeugt.
- Hohe Performanz und Flexibilität, da alle Komponenten in ,,**C**" implementiert sind.
- Baukastensystem – Modelle von Rechnernetzen können einfach aus vordefinierten Modellkomponenten zusammengesetzt werden.
- Flexible Instrumentierung – die Modellkomponenten enthalten Meßpunkte, an denen auf Wunsch Meßwerte ermittelt und in wählbarem Format ausgegeben werden.
- Statistische Absicherung – für die ermittelten Meßwerte können Vertrauensgrenzen automatisch ermittelt werden.
- Wiederholung von Simulationsläufen mit variierten Parametern.
- Einfache Erweiterbarkeit – ein Skelett für neue Modellkomponenten läßt sich nach einem vorgegebenem Schema erzeugen, für die Ausformulierung in ,,C" steht eine Anzahl von Grundkomponenten zur Verfügung.
- Interaktive Fehlersuche.
- Gezieltes Verfolgen der Ereignisse in bestimmten Modulen (Event Traceing) – für Fehlersuche und Verifikation.

In [Pet91] ist eine genaue Beschreibung der Konzepte und verwendeten Verfahren von simuLan sowie ein Programmierhandbuch, in dem alle Funktionalitäten anhand von Beispielen ausführlich beschrieben sind, enthalten.

4.2 Grundlagen

4.2.1 Darstellung der Zeit

In simuLan wird die Zeit in *ITU (Indivisible Time Unit)* gemessen. Diese ganzzahlige Einheit wird über die Variable *itu_per_second (ITU pro Sekunde)* mit der real simulierten Zeit (in Sekunden) in Verbindung gebracht. Der Wert dieser Variable wird über die Konfigurationsdatei (s. [Pet91]) vorgegeben.
Durch die Verwendung dieser skalierbaren internen Zeiteinheit ist es möglich, verschiedene Systeme mit sehr unterschiedlichen Zeitauflösungen (ns - Tage) zu simulieren, ohne an der Simulationsumgebung etwas ändern zu müssen.
Die bisher simulierte Zeit in `ITU` steht in der globalen Variablen *simul_time* zur Verfügung.

4.2.2 Beschreibung eines *Ereignis*: *EVENT* und *SADU*

simuLan basiert auf der ereignisorientierten Simulation. Jedes Ereignis (z.B. Ankunft eines Jobs an der CPU, Ende einer Übertragung) wird durch eine Datenstruktur *EVENT* und angehängte spezifische Informationen beschrieben.
Die wichtigsten Elemente des `EVENT`s sind:

- *event_time*: Dies ist der Zeitpunkt, zu dem das Ereignis auftritt, das durch den `EVENT` beschrieben wird.
- *event_handler()*: Ein Zeiger auf die Funktion, die zum Zeitpunkt *event_time* das Ereignis bearbeitet. Sie erhält als Parameter einen Zeiger auf den `EVENT`.
- *k_nr*: *K_nr* enthält die Nummer des Netzknotens, der den `EVENT` in die Eventliste gestellt hat. Bei `EVENT`s, die von einem Netzmodul behandelt werden, ist dieses die Nummer des Senders.
- *event_info*: Die über diesen Zeiger erreichbaren Informationen beschreiben das Ereignis für den *event_handler* näher. Der *event_handler* weiß, was für ein Objekt am *event_info* hängt.

Bei Ereignissen, die von Protokollschichtmodulen bearbeitet werden, hängt an *event_info* immer eine Datenstruktur *SADU (Service Access Data Unit)*, die folgende zusätzliche Informationen enthält:

- *status*: Über diese Variable können Statusinformationen zu dieser Nachricht an den Sender zurückgegeben werden. Dazu gehört z.B. die Information, ob die Nachricht fehlerfrei übertragen werden konnte.
- *pdu_size*: Größe der Nachricht in einer für die Schicht spezifischen Einheit (Bit, Byte, Paket, ...).
- *receiver*: Die Knotennummer des Empfängers, der diese Nachricht erhalten soll.
- *next*: Pointer auf zusätzliche Information zu dieser Dateneinheit.

Die Abb. 4.1 zeigt schematisch die Datenstruktur, die ein Ereignis in einem Protokollschichtmodul beschreibt. Die Struktur *N*-PDU enthält die privaten Informationen der höheren Protokollschicht *N*.
WICHTIG !

> *Datenstrukturen werden in* simuLan *dynamisch über spezielle Funktionen allokiert und wieder freigegeben. Um zu verhindern, daß Strukturen überhaupt nicht oder mehrfach an die Speicherverwaltung zurückgegeben werden (Programm stürzt ab !), dürfen diese nur in den Prozeduren, die die Senderseite simulieren, allokiert und freigegeben werden. Auf der Empfängerseite benötigte Informationen müssen dort aus den Strukturen in eigene Datenbereiche kopiert werden.*

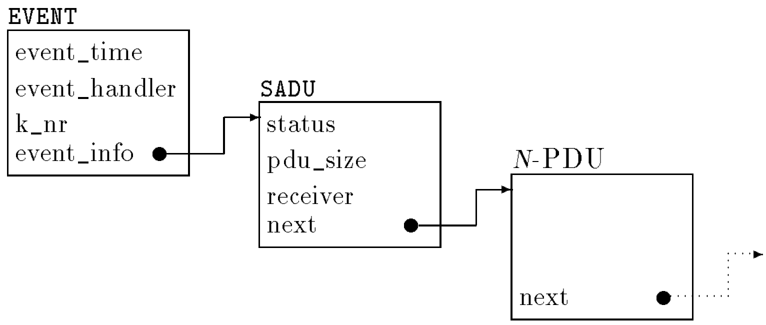


Abbildung 4.1: Verkettung `EVENT`, `SADU`, `N-PDU`, ...

4.2.3 Repräsentation der Topologie

Die Netzwerktopologie wird durch Nachbarschaftsbeziehungen dargestellt. Eine Möglichkeit der Darstellung benutzt die Signallaufzeit zwischen zwei Stationen. Diese wird über netzspezifische *calc_laufzeit*-Funktionen in den Netzmodulen nachgebildet. Diese Funktionen erhalten die Sender- und Empfängernummer als Parameter und geben die benötigte Signallaufzeit (in `ITU`) zwischen diesen beiden Knoten zurück.

4.2.4 Verkehrsmatrizen

Ein wichtiger Parameter für die Modellierung von Netzwerken ist die Angabe, welche der Netzknoten miteinander kommunizieren. (Untersuchung von z.B. Client-Server-Beziehungen). Es ist üblich, für jedes Paar *(S, E)* eine Wahrscheinlichkeit anzugeben, mit der eine vom Sender S erzeugte Nachricht an den Empfänger E gerichtet ist. Daraus ergibt sich zunächst eine quadratische *Verkehrsmatrix*. In der Praxis zeigt sich jedoch, daß diese Matrix meistens dünn besetzt ist. In simuLan kann daher einem Knoten eine Liste von ,,bevorzugten" Partnerknoten und dazugehörigen Wahrscheinlichkeiten zugeordnet werden. Falls die Summe der Wahrscheinlichkeiten kleiner als eins ist, wird für die Differenz eine Gleichverteilung über alle Knoten angenommen.

In den Fällen, in denen eine solche detaillierte Beschreibung nicht benötigt wird, kann der Empfänger der Nachrichten eines Senders gleichverteilt aus allen simulierten Netzknoten ausgewählt werden.

4.2.5 Das Schicht-Modul

In simuLan wird ein System durch eine Menge miteinander interagierender Protokollschichtmodelle definiert. Jedes Modell modelliert dabei, in Anlehnung an das ISO/OSI-Modell (s. Kapitel Abschnitt 3.6 [*Das ISO/OSI-Referenzmodell*], Seite 10), eine bestimmte Protokollschicht des Netzwerkes.
Dabei ist folgende Entsprechung zwischen dem ISO/OSI-Modell und dem simuLan-Modell vorgesehen:

simuLan-Schicht		ISO/OSI-Schicht	
1	↔	1 und 2a	(Physical-, Mac-Layer)
2	↔	2b	(Logical Link Control)
3	↔	3	(Network Layer)
4	↔	4	(Transport Layer)
5	↔	5,6,7	(Session-, Application-, Presentation Layer)

Das Modell, welches die Schichten *1* und *2a* modelliert, wird im folgenden als *Netzmodell* bezeichnet. Von dem dazugehörigen Netzmodul existiert in einem modellierten Rechnernetz genau eine Instanz. Von den darüberliegenden Schichtmodulen kann dagegen in jedem simulierten Netzknoten eine Instanz existieren. (siehe Abschnitt 4.3.1 [*Das Netzmodul PTP*], Seite 21)

Die oberste simuLan-Schicht *5* wird auch als *,,Knotenmodell"* bezeichnet. In ihr wird die lasterzeugende Anwendung (die Vorgänge ,,im Inneren des Knotens") modelliert. (siehe Abschnitt

4.4.1 [*Das Knotenmodell K0*], Seite 25)

Die hierarchische Abfolge der Modelle und damit die Folge der Protokollschichten im modellierten System kann frei über die Konfigurationsdatei vom Modellierer festgelegt werden. simuLan kann allerdings nicht feststellen, ob die gewählte Folge sinnvoll ist.

Jedes Modell wird durch ein *Modul*, eine Anzahl von Funktionen und Datenstrukturen, die in der Programmiersprache „**C**" geschrieben sind, realisiert.

simuLan setzt im allgemeinen für jedes Modell die Existenz der unten beschriebenen Prozeduren mit den angegebenen Funktionalitäten voraus.
Heißt das dazugehörige Modul z.B. XLayer und realisiert die Schicht *N*, sind dieses:

*XLayer*_**init():** Diese Prozedur wird vor jedem Simulationslauf nach dem Einlesen der Konfiguration einmal aufgerufen. Hier werden Datenstrukturen und Variablen initialisiert.

*XLayer*_**cleanup():** Nach jedem Simulationslauf wird diese Prozedur einmal aufgerufen. An dieser Stelle werden z.B. allokierte Speicherbereiche freigegeben und interne Meßwerte ausgegeben.

*XLayer*_**rc():** Findet simuLan beim Auswerten der Konfigurationsdatei das vorher bekannt gemachte Schlüsselwort *XLayer*, wird diese Funktion aufgerufen. Die Aufgabe ist dann die Auswertung der in der Konfigurationsdatei für dieses Modul angegebenen Parameter, eine Plausibilitätsprüfung derselben und die Initialisierung bestimmter Datenstrukturen des *XLayer*-Moduls mit diesen Werten. Zusätzlich sollte diese Funktion eine Beschreibung aller Parameter und ihrer Defaultwerte ausgeben können.

*XLayer*_**sap():** Diese Prozedur stellt den *SAP* (s. Abschnitt 3.6) der Schicht *N* dar, die durch das Modul *XLayer* modelliert wird. Sie wird also immer dann aufgerufen, wenn die Schicht *N+1* eine *(N+1)*-`PDU`(s. Abschnitt 3.6) senden möchte. Diese *(N+1)*-`PDU` wird entsprechend dem modellierten Protokoll verpackt und unter Verwendung der Schicht *N-1* gesendet.

*XLayer*_**pdu_received():** Die Empfänger-Schicht *N-1* ruft diese Prozedur auf, um der *XLayer*-Schicht *N* (im Empfänger) eine empfangene Nachricht zu übergeben. Die Nachricht wird hier dann soweit wie möglich bearbeitet und eine dabei entstandene *N+1*-`PDU` an die Schicht *N+1* weitergegeben.

*XLayer*_**pdu_sended():** Nach erfolgreicher Übertragung einer *N*-`PDU` ruft die Schicht *N-1* diese Funktion auf. Anschließend wird die nächste zu sendende *N*-`PDU` an die Schicht *N-1* gegeben und eventuell eine „Fertig"-Meldung an die Schicht *N+1* übermittelt.

Für die Speicherung der Modellparameter und des aktuellen Zustandes werden in jedem Modul normalerweise die folgenden Datenstrukturen benötigt:

*XLayer*_**DUD:** In dieser Struktur, *DUD* steht für **D**ata **U**nit **D**escriptor, sind alle Informationen über genau eine bestimmte *N*-`PDU` gespeichert, die für Verwaltung, Übertragung und die Meßwertgewinnung in der Schicht *N* benötigt werden.
Dazu gehören z.B.:

- Größe der zu übertragenden Datenmenge in Bit
- Zeitpunkt an dem diese *N*-`PDU` gesendet wurde
- Anzahl der „erlebten" Kollissionen
- Zeiger *next* auf die angehängten Informationen der Schicht *N+1*.

*XLayer*_**INFO:** Diese Struktur enthält die schichtspezifischen Parameter und Meßpunkte (s. Abschnitt 4.2.7). Bei einem Netzmodul ist zum Beispiel der Protokolloverhead und bei einem Knotenmodul die Verteilungsfunktion, nach der der Zeitpunkt des nächsten Sendewunsches bestimmt wird, vermerkt.

Der Datenaustausch zwischen zwei benachbarten Schichtmodulen *N*, *N+1* geschieht **nur** über den `EVENT` (s. Abschnitt 4.2.2) und die daran angehängten Datenstrukturen.

4.2.6 Die Datenstrukturen *net_params* und *NODE*

Das zu simulierende Netzwerk besteht aus einem Netz, modelliert durch ein *Netzmodell* (siehe Abschnitt 4.3 [*Das Netzmodul PTP*], Seite 21) und *N* damit verbundenen Rechnern. Für die Verwaltung dieser Objekte gibt es in simuLan die globalen Datenstrukturen *net_params* und *NODE* (s. Anhang C). In *net_params* werden allen Modulen diejenigen Informationen bereitgestellt, die für **alle** implementierten Netzmodule vorhanden und öffentlich bekannt sein müssen. Die Datenstruktur `NODE` stellt die Informationen bereit, die für jeden einzelnen simulierten Netzknoten global bekannt sein müssen. Durch *nodearry[]* macht simuLan ein Array von *N* `NODE`s global verfügbar, über das auf die Beschreibung eines Knotens *i* mit *nodearray[i]* zugegriffen werden kann. Ein Knoten wird über den Index „**i**" der dazugehörigen `NODE`-Struktur in *nodearray[]* eindeutig identifiziert.

Die Datenstruktur *net_params* enthält im einzelnen:

name: Der Name des simulierten Netzes, der in der Ausgabe erscheint.

bits_per_second: Gibt die nominelle Übertragungsgeschwindigkeit des Netzes in Bits pro Sekunde an. (Eingabeparameter)

itu_per_bit: Beschreibt die Zeitdauer für die Übertragung eines Bits in `ITU` (s. Abschnitt 4.2.1). (wird aus *bits_per_second* berechnet)

max_data_len: Dies ist die maximale Nachrichtengröße (Nutzdaten) in Bits, die in eine Dateneinheit des Netzmoduls paßt.

min_data_len: Die minimale Größe (Nutzdaten) in Bits, die eine zu übertragende Nachricht haben muß, ist in dieser Variablen vermerkt. Kürzere Nachrichten werden durch Anhängen von Dummyinformation auf *min_data_len* vergrößert.

net_init(): Zeiger auf die Initialisierungsfunktion des Netzmoduls (s. Abschnitt 4.2.5).

net_cleanup(): Zeiger auf die Funktion die nach Beendigung der Simulation aufgerufen wird (s. Abschnitt 4.2.5).

l1_sap(): Zeiger auf die Funktion die den `SAP` darstellt. Über diese Funktion werden an das Netzmodul Sendeaufträge übergeben (s. Abschnitt 3.6 und Abschnitt 4.2.5).

Die wichtigsten Elemente einer Datenstruktur vom Typ `NODE` sind:

name: Enthält für jeden Netzknoten den Namen, der über die Konfigurationsdatei vorgegeben wird.

k_init(): Entspricht *XLayer_init()* des `Knotenmoduls`.

k_cleanup(): Entspricht *XLayer_cleanup()* des `Knotenmoduls`.

calc_empfaenger(): Zeigt auf eine Funktion, die den Empfänger für eine von dem Knoten erzeugte Nachricht ermittelt (s. Abschnitt `Verkehrsmatrizen` in [Pet91]).

traffic_list: Zeiger auf die zu *calc_empfaenger()* gehörende Empfängerliste (s. Abschnitt [`Verkehrsmatrizen`] in [Pet91]).

l4_sap(),l3_sap(),l2_sap(): Diese Funktionszeiger entsprechen *XLayer_sap()* (s. Abschnitt 4.2.5) der Protokollschichten *2*, *3* und *4*, die in dem durch `NODE` beschriebenen Netzknoten für eine Übertragung benutzt werden. Die entsprechende Netzzugangsfunktion *l1_sap()* ist in *net_params* enthalten.

l5_pdu_received(),l4_pdu_received(), l3_pdu_received(),l2_pdu_received(): *XLayer_pdu_received()* (s. Abschnitt 4.2.5) entsprechende Funktionszeiger der Prokollschichten *2–5*.

l5_pdu_sended(),l4_pdu_sended(),l3_pdu_sended(),l2_pdu_sended(): Die Funktionszeiger entsprechen *XLayer_pdu_sended*(s. Abschnitt 4.2.5) der Schichten *2–5*.

l4_info,l3_info,l2_info: Das für Protokollschicht $N = 2,3,4$ in diesem Netzknoten benutzte `Schichtmodul` kann an diese Zeiger knotenbezogene, private Informationen anhängen.

k_info: Das benutzte `Knotenmodul` hängt an diesen Zeiger private, für den Knoten spezifische Informationen.

4.2.7 Meßpunkte

Um die Gewinnung, Verarbeitung und Ausgabe von Meßdaten möglichst flexibel und doch einheitlich zu gestalten, werden an den für „interessant" befundenen Stellen einer Modellkomponente *Meßpunkte* einprogrammiert. An diesen Meßpunkten können jeweils mehrere verschiedene Messungen durchgeführt werden. Zu jeder Messung gehören eine *Meßfunktion*, Ausgabekanäle und Ausgabefunktionen sowie ggf. eine Funktion zur statistischen Absicherung. In der Beschreibung des zu simulierenden Systems (in der Konfigurationsdatei) wird angegeben, ob und welche Messungen an einem Meßpunkt durchgeführt werden sollen. Zusätzlich wird dort vorgegeben, wohin die Meßwerte ausgegeben und wie die Ausgaben formatiert werden sollen. Außerdem kann angegeben werden, daß für einen Meßwert eine statistische Absicherung durchgeführt werden soll (s. [Pet91]).
Die für eine Meßfunktion `XYZ` benötigten Prozeduren und Datenstrukturen sind:

probept_tab: Diese Tabelle enthält für jede Meßfunktion `XYZ` einen Eintrag. In ihm sind das Konfigurationsschlüsselwort (**xyz**), ein Zeiger auf *rc_XYZ()* sowie eine kurze Beschreibung der Funktion `XYZ` enthalten.

rc_XYZ(): Diese Funktion wird von simuLan aufgerufen, wenn in der Konfigurationsdatei das Wort **xyz** gefunden wird. Sie initialisiert die zu einer Messung gehörende Datenstruktur *PROBE* (s. [Pet91]).

XYZ(): Dieses ist die eigentliche Meßfunktion. Sie berechnet aus den Informationen in dem übergebenen `EVENT` einen Meßwert und speichert ihn in `PROBE`→*akt_val* ab. Zusätzlich kann der Wert auch in eine Datei oder eine Pipe geschrieben werden.

XYZ_erg(): Die Ausgabe der in `PROBE` berechneten Ergebnisse geschieht in dieser Funktion.

Alle für einen Meßpunkt **M** aktivierten Meßfunktionen lassen sich mit dem vordefinierten Makro *MEASURE()* folgendermaßen aufrufen:

```
PROBEPOINT M;
EVENT *event_p;
/* ... */
MEASURE(M, event_p);
```

(*event_p* ist ein Zeiger auf den `EVENT`, der das Ereignis ausgelöst hat, das zur Aktivierung des Meßpunktes **M** geführt hat.)

Eine weitergehende Einführung in das Konzept und die Möglichkeiten der Meßpunkte ist [Pet91] zu entnehmen.

4.2.8 Warteschlangen

Die Simulationsumgebung simuLan stellt zwei Arten von Warteschlangen zur Verfügung. Diese sind:

NQUEUE: Die *normale Warteschlange*. Zu einer `NQUEUE` gehören *n* Server, die jeweils einen Kunden bedienen können.

TQUEUE: Die *Ticket Queue* stellt *n Tickets* oder *Tokens* zur Verfügung, von denen jeder Kunde eine bestimmte Anzahl anfordert.

Als Bedienstrategien stehen jeweils *FIFO* (*First In First Out*) und *LIFO* (*Last In First Out*) zur Verfügung. Der globale Parameter `glob_params.max_q_length` gibt eine maximale Schlangenlänge an, bei deren Überschreiten ein Fehler ausgelöst und der Simulationslauf beendet wird. Die Elemente einer Warteschlange sind Datenstrukturen vom Typ `EVENT` (siehe Abschnitt 4.2.2 [*EVENT*], Seite 15).
Ein Kunde wird in die Warteschlange gestellt, falls kein Server mehr frei ist (`NQUEUE`) oder die Anzahl benötigter Tickets größer als die Anzahl der vorhanden freien Tickets ist (`TQUEUE`). Ein Ticket kann z.B. einer Speichereinheit entsprechen.

In jeder Warteschlange kann während eines Simulationslaufes die Zahl der bedienten Kunden, die mittlere Wartezeite, die mittlere Schlangenlänge und die Auslastung des dazugehörigen Servers ermittelt werden. Diese Funktion wird normalerweise über die Konfiguration aktiviert.

Weitere Informationen über Warteschlangen in simuLan sind [Pet91] zu entnehmen.

4.2.9 Die Konfigurationsdatei

In der *Konfigurationsdatei* wird das Simulationsexperiment mit allen seinen Parametern, z.B. Netz- und Knotentyp, Anzahl Knoten, Ausgabedateien für die Ergebnisse, ... vollständig beschrieben. Sie kann Kommentare enthalten, die mit einem # beginnen. Danach wird alles bis zum nächsten Zeilenende überlesen.

Da die Syntax sich mit jedem neuen Schichtmodul und jedem neuen Meßpunkt erweitert, wird hier auf eine vollständige Beschreibung der Syntax und Semantik verzichtet. Stattdessen ist simuLan in der Lage, eine Schablone für eine `Konfigurationsdatei` mit allen verfügbaren Sprachelementen (kommentiert) auszugeben, aus der mit einem Texteditor die gewünschte Modellbeschreibung erstellt werden kann.

Allgemein besteht die `Konfigurationsdatei` aus den zwei Teilen Vorspann und Hauptteil.
Der Vorspann wird von simuLan `genau einmal` gelesen und enthält folgende Elemente:

- Die Definition aller benötigten Ausgabedateien mit den dazugehörigen symbolischen Namen.
- Die Definition aller *Laufvariablen* mit Start- und Endwert. Diese Laufvariablen ermöglichen es, automatisch mehrere Simulationsläufe mit veränderten Systemparametern hintereinander auszuführen.

Der Hauptteil beschreibt das zu simulierende Netzwerk. Er muß mindestens ein `Netzmodell` und ein `Knotenmodell` enthalten. Die Angabe der Paramter für das entsprechende Modul erfolgt allgemein in der Form:

```
<Protokollschicht> <Modulname>
        <Parametername> <Parameterwert>
        ...
        END<modul>
```

Die Parameter für ein Modul können im allgemeinen in beliebiger Reihenfolge und beliebig oft angegeben werden. Für den Simulationslauf wird jeweils der als letztes für einen Parameter angegebene Wert verwendet.

Ein Beispiel einer vollständigen `Konfigurationsdatei` wird in Abschnitt 4.5.2 angegeben.
In [Pet91] werden der Aufbau, die Strukturen und die dazugehörigen Einlesefunktionen näher beschrieben.

4.2.10 Verfolgen von Ereignissen und Fehlersuche

Das Verfolgen von Ereignissen und die damit verbundene Informationsverwaltung sind sehr laufzeitintensiv. Daher kann zur Übersetzungszeit festgelegt werden, ob die dazugehörigen Funktionen mit in das Programm simuLan eingebunden werden oder nicht [Pet91].
In der Version mit den Funktionen zur Informationsausgabe kann über die Angabe von '-D<modulename>=<flag>' in der Kommandozeile, die Ausgabe in bestimmten Funktionen im

Modul '`modulname`' aktiviert werden. Der Parameter '`flag`' wird als Zeichenkette an das entsprechende Modul übergeben, die diesen entsprechend auswerten muß.

Im Anhang *Debugflags* sind für jedes in dieser Arbeit implementierte Schichtmodul die definierten Werte von '`flag`' und die jeweilige Bedeutung angegeben.

Eine genaue Beschreibung und ein Beispiel sind in [Pet91] im Abschnitt [Debugging] angegeben.

4.3 Das Netzmodell *PTP*

In diesem Abschnitt wird, nach einigen einführenden Bemerkungen zu *Netzmodulen*, ein allgemein verwendbares Netzmodell entwickelt, das ein vollvermaschtes (fiktives) Point-to-Point-Netzwerk nachbildet.

4.3.1 Einführung

In einem *Netzmodul* werden die Schichten *1* und *2a (MAC)* des ISO/OSI-Schichtenmodells modelliert.

Allgemeine Bemerkungen:

- In simuLan werden alle Knoten durch **ein** Netz verbunden. Daher existiert nur eine Netzbeschreibung.
- Da das Netzmodul die niedrigste Schicht in der Protokollhierarchie realisiert, werden die Funktionen *XLayer_pdu_received()* und *XLayer_pdu_sended()* nicht benötigt. Diese Funktionen stellen die Schnittstelle zur nächst tieferen Protokollschicht dar (s. Abschnitt 4.2.5).
- Mit der `globalen` Datenstruktur *net_params* stellt simuLan allen Modulen Informationen zur Verfügung, die für alle implementierten Netzmodelle vorhanden und öffentlich bekannt sein müssen (s. Abschnitt 4.2.6 und [Pet91]).

4.3.2 Beschreibung des Modells

In einem vollvermaschten Point-To-Point-Netzwerk (s. Abb. 4.2), wie es hier modelliert wird, ist jeder Knoten direkt mit jedem anderen Knoten im Netzwerk über zwei Leitungen, eine zum Senden und eine zum Empfangen, verbunden.

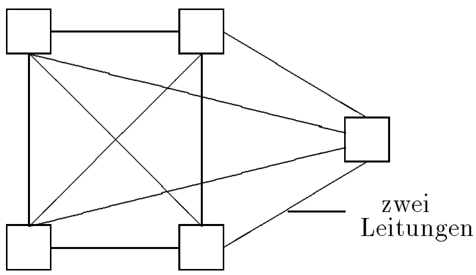


Abbildung 4.2: Ein vollvermaschtes Point-To-Point-Netzwerk

Die Aufgabe des Netzes ist es, von der simuLan-Schicht *N*$\geq$*2* (Sender) eine zu sendende Dateneinheit, im Folgenden *Paket* genannt, zu übernehmen und sie dem Empfänger zuzustellen. Nachdem das `Paket` fehlerfrei übertragen wurde, ist zusätzlich der Sender davon zu benachrichtigen.

Von folgendem Übertragungsprotokoll wird ausgegangen:

- Die Größe einer Netzdateneinheit und damit die des `Pakets` (`SADU`$\rightarrow$*pdu_size*) ist nach oben nicht beschränkt.

- Die Mindestgröße des `Paket`s ist durch *net_params.min_data_len* (s. Abschnitt 4.2.6) vorgegeben. Falls das `Paket` kleiner ist, wird es entsprechend vergrößert.
- Der Protokolloverhead für jedes `Paket` ist in *ptp_info→overhead* vermerkt. Die Anzahl der zu übermittelnden Bits ergibt sich aus der Summe der `Paket`größe und dieses Overheads.
- Das Netzzugriffsverfahren ist sehr einfach, da es für jede mögliche Verbindung eine eigene Leitung gibt und es somit keine Konkurrenz um den Zugang zum Übertragungsmedium gibt. Es wird aus diesem Grund nicht explizit simuliert.

Alle Nachrichtenlängen sind in **Bit** anzugeben.

4.3.3 Schnittstellen

Aus Abb. 4.3 sind die Schnittstellen zur Schicht *2* sowie der Datenfluß innerhalb des PTP-Moduls ersichtlich. Die Schreibweise `(*...)()` besagt, daß die entsprechende Schnittstellenfunktion der Nachbarschicht über einen Funktionszeiger aufgerufen wird. Die Schicht $N \geq 2$ im Sender erzeugt ein `Paket` und übergibt dieses zum Senden an die Funktion *ptp_data_encapsulation()*, die der in Abschnitt 4.2.5 beschriebenen Prozedur `XLayer_sap()` entspricht.

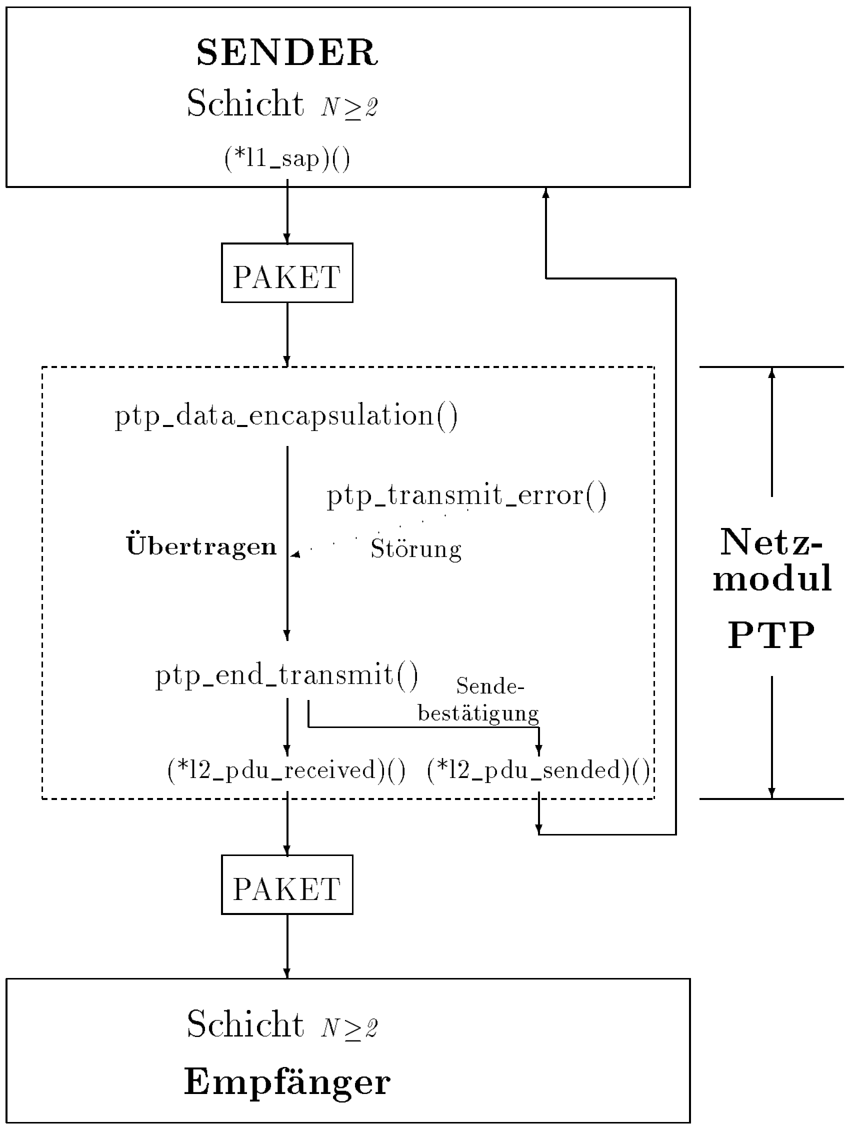


Abbildung 4.3: Schnittstellen und Datenfluß im PTP-Netzmodul

Ein `Paket` besteht dabei zumindest aus dem `EVENT` und einer daran angehängten `SADU` (s. Abschnitt 4.2.2). Für den Informationsaustausch zwischen höheren Schichten ($N \geq 2$) können an dem Zeiger `SADU`→*next* Datenstrukturen angehängt sein. Diese dürfen durch die Schicht *1* nicht verändert werden.
Nach der erfolgreichen Übertragung gibt die Funktion *ptp_end_transmit()* zunächst das `Paket` an die Empfängerschicht $N \geq 2$ und benachrichtigt anschließend die Schicht $N \geq 2$ im Sender.
Während der Übertragung auftretende Störungen auf dem Kanal werden durch die Funktion *ptp_transmit_error()* simuliert.

4.3.4 Realisierung

In dem Anhang D ist das vollständige Programmlisting zu finden.

4.3.4.1 ptp_data_encaspulation()

Die Funktion *ptp_data_encapsulation()* bekommt einen Zeiger auf den `EVENT`, der das `Paket` beschreibt, übergeben. Dieses `Paket` muß nun, entsprechend dem in Abschnitt 4.3.2 definierten Übertragungsprotokoll, in die im Netz verwendete Dateneinheit, im folgenden `PTP-FRAME` genannt, umgewandelt werden. Dazu wird von der Speicherverwaltung eine Datenstruktur vom Typ *ptp_FRAME* angefordert und zwischen `SADU` und eventuellen nachfolgenden Datenstrukturen eingehängt. Dieser `ptp_FRAME` entspricht der in Abschnitt 4.2.2 beschriebenen `XLayer_DUD`.

Die Abb. 4.4 zeigt die zu einem `PTP-FRAME` gehörenden Datenstrukturen.

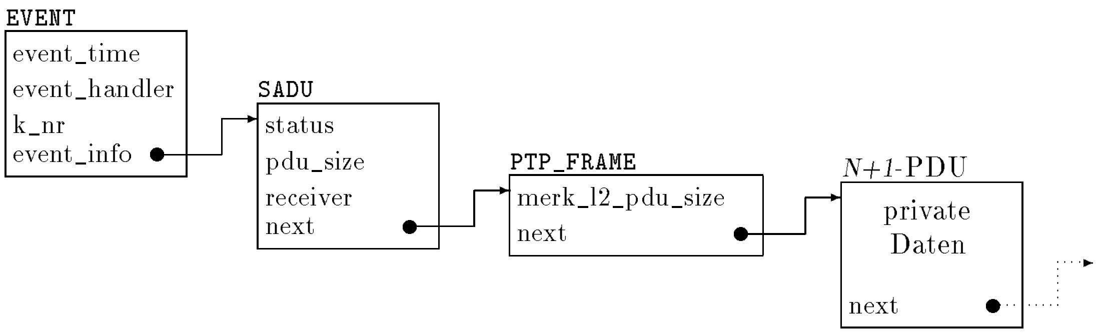


Abbildung 4.4: Die Datenstrukturen eines `PTP-FRAME`s

Die `PTP-FRAME`-Größe (*frame_size)* und damit die Anzahl der zu übertragenden Bits ergibt sich aus: *ptp_info.overhead + max(*`SADU`→*pdu_size, (net_params.min_data_len))*

Nachdem die `Paket`-Größe in `PTP-FRAME`→*merk_l2_pdu_size* gerettet wurde, wird diese Größe in `SADU`→*pdu_size* abgespeichert.

Das nächste Ereignis für diesen `PTP-FRAME` ist das Ende der Übertragung. Die Funktion *ptp_-end_transmit()* simuliert dieses Ereignis. Ein Zeiger auf diese Funktion wird in `EVENT`→*event_-handler* eingetragen. In `EVENT`→*event_time* wird der Zeitpunkt, an den dem die Übertragung beendet wird, abgespeichert. Er ergibt sich aus:

`EVENT`→*event_time + frame_size ⋆ net_params.itu_per_bit.*

Die alte Zeit in `EVENT`→*event_time* muß von der aufrufenden Prozedur gesetzt werden. So ist es möglich, Verzögerungszeiten zwischen den Schichten einfach zu simulieren. Das Produkt aus *frame_size* und *net_params.itu_per_bit* liefert die Zeit (in `ITU`), die zum Senden von *frame_size* Bits benötigt wird. Der Zeitpunkt, an dem der `PTP-FRAME` vollständig beim Empfänger angekommen ist, ergibt sich aus dem Zeitpunkt, an dem die Übertragung beendet wurde **plus** der Signallaufzeit zwischen Sender und Empfänger. Diese Signallaufzeit wird in dem einfachen hier entwickelten Modell vernachlässigt.

4.3.4.2 ptp_transmit_error()

Die Funktion *ptp_transmit_error()* erzeugt nach einer vorgegebenen Verteilungsfunktion ([Pet91]) Störungen auf dem Übertragungskanal. Der Zeitpunkt der Störung wird in der modulinternen Variablen *ptp_stoerzeit* vermerkt und kann in der Funktion *ptp_end_transmit()* ausgewertet werden. Zum Schluß wird der Zeitpunkt der nächsten Störung ermittelt und für diesen ein Aufruf von *ptp_transmit_error()* eingeplant.

4.3.4.3 ptp_end_transmit

Die Funktion *ptp_end_transmit()* simuliert das Ende der Übertragung und die Ankunft eines `PTP-FRAME`s in der `MAC`-Schicht des Empfängers (s. Abschnitt 3.5).

Nach dem Aufruf aller aktivierten Meßfunktionen des Meßpunktes *ptp_probept_end_transmit* mit dem Makro `MEASURE()` (s. Abschnitt 4.2.7) wird die Anzahl der durch das Netz übertragenen `PTP-FRAME`s erhöht.
Anschließend wird durch Ausketten des `ptp_FRAME` alle schichtspezifische Information aus der übertragenen Dateneinheit entfernt und in `SADU`→*pdu_size* die alte `Paket`-Größe wieder eingetragen.
Nun wird geprüft, ob die Übertragung gestört war. Dieses trifft zu, falls *ptp_stoerzeit (s.o.)* größer als die Startzeit der Übertragung ist. Bei ungestörter Übertragung wird das `Paket` an die Empfängerschicht $N \geq 2$ übergeben. Die dazugehörige Funktion wird über *nodearray[*`SADU`→*-receiver].l2_pdu_received* ermittelt. Der das `Paket` beschreibende `EVENT` mit den angehängten Informationen darf von dieser Prozedur nicht verändert werden (s. auch S. 15). Im Falle einer Störung wird diese Funktion **nicht** aufgerufen, da auch in der Realität fehlerhaft empfangene `Paket`e nicht an die höheren Schichten weitergereicht werden.
Durch den Aufruf von *nodearry[*`EVENT`→*k_nr].l2_pdu_sended()*, mit dem `EVENT` als Parameter, wird nun noch der Sender des `Paket`s über den Abschluß der Übertragung benachrichtigt.

4.3.4.4 Die Rahmenfunktionen des Netzmoduls *PTP*

Die Rahmenfunktionen des Netzmoduls `PTP` sind *ptp_rc()*, *ptp_init()*, *ptp_cleanup()* und *ptp_print_frame()*.

Die Aufgaben der Funktion *ptp_rc()* sind:

- Falls über die Kommandozeile angegeben, alle variablen Parameter des `PTP`-Moduls mit einer Erklärung ausgeben.
- Die Datenstruktur *net_params* (s. Abschnitt 4.3.1) mit Defaultwerten vorbelegen.
- Die Funktion *rcreadkeywds()* (s. [Pet91]) mit der Tabelle der definierten Schlüsselwörter für das `PTP`-Modul aufrufen. Dabei werden gegebenenfalls die Defaultwerte ausgegeben.
- Die eingelesenen Parameter auf Plausibilität überprüfen.

Die Funktion *ptp_init()* hat folgende Aufgaben:

- Die Liste der freien Datenstrukturen vom Typ `ptp_FRAME` initialisieren (s. auch Kapitel [Speicherverwaltung] in [Pet91]).
- Die lokalen Variablen, in denen die Anzahl der übertragenen Bits bzw. `PTP-FRAME`s berechnet werden, mit dem Wert Null vorbelegen.
- Falls die Simulation von Störungen auf dem Übertragungsweg gewünscht ist, wird in Form eines `EVENT`s das erste Störungsereignis in die Ereignisliste eingefügt.

ptp_cleanup() gibt die Anzahl der übertragenen Bits und `PTP-FRAME`s aus.

Die Funktion *ptp_print_frame()* ermöglicht die Ausgabe des Inhaltes eine `PTP-FRAME`s und der angehängten Datenstrukturen. Diese Funktion ist für die Fehlersuche notwendig.

4.3.4.5 Die Meßfunktion *ptp_throughput()*

Die Funktionen *ptp_throughput()*, *ptp_mean_throughput()*, *ptp_rc_throughput()* gehören zu dem in *ptp_end_transmit()* eingebauten Meßpunkt *ptp_probept_end_transmit*. Dabei liest *ptp_rc_throughput()* die benötigten Parameter ein, und *ptp_mean_throughput()* gibt die gewonnenen Ergebnisse am Ende eines Simulationslaufes aus.
In der eigentlichen Meßfunktion *ptp_throughput()* wird die Auslastung des `PTP`-Netzes in Prozent nach folgender Gleichung bestimmt:

$$\frac{bisherige\ Zeit\ ,während\ der\ Daten\ \text{über}\ das\ Netz\ \text{übertragen}\ wurden\ (ITU)}{bisherige\ Simulationszeit\ (ITU)} \cdot 100$$

Für weitergehende Informationen zu Meßpunkten und ihrer Implementation s. Abschnitt 4.2.7 und [Pet91].

4.4 Das Knotenmodell *K0*

In diesem Abschnitt wird, nach einigen einführenden Bemerkungen, ein `Knotenmodell` entwickelt, das einen sehr einfachen ,,Rechner" nachbildet.

4.4.1 Einführung

In einem `Knotenmodul` werden die ISO/OSI-Schichten *5*, *6* und *7* (s. Abschnitt 3.6) modelliert. Diese finden ihre Entsprechung in der simuLan-Schicht *5*, in der alle lasterzeugenden Anwendungen simuliert werden.

Allgemeine Bemerkungen:

- Zu jedem *Rechnerknoten* in einem modellierten Netzwerk gehört genau eine `Knotenmodul`-Instanz.
- In simuLan können **gleichzeitig** verschiedene Knotenmodelle, die durch unterschiedliche `Knotenmodul`e realisiert werden, in der Modellbeschreibung enthalten sein und simuliert werden.
- Da ein `Knotenmodul` die höchste Schicht in der Protokollhierarchie realisiert, wird die Funktion *XLayer_sap()* nicht benötigt, da diese die Schnittstelle zu der nächst höheren Schicht darstellen würde (siehe Abschnitt 4.2.5 [*Schichtmodul*], Seite 16).
- Für jeden simulierten *Rechnerknoten* gibt es in dem globalen Array *nodearry[]* eine Datenstruktur `NODE`, die diesen Knoten mit allen Eigenschaften, z.B. verwendete Übertragungsprotokolle, ..., vollständig beschreibt (siehe Abschnitt 4.2.6 [*NODE*], Seite 18).

4.4.2 Beschreibung des Modells

Der einfache ,,Rechnerknoten" *K0* (s. Abb. 4.5) modelliert ein einfaches, direkt an das Netz angeschlossenes Terminal.

Der ,,Hacker" sitzt vor dem Bildschirm, denkt ein Weile nach und gibt dann Zeichenfolgen über die Tastatur ein. Diese `Dateneinheiten`, im folgenden `Nachrichten` genannt, werden dann von dem Terminal über die Protokollschicht *N-1* an einen Verarbeitungsrechner gesendet. Falls während der Übertragung ein erkennbarer Fehler aufgetreten ist, wird die `Nachricht` erneut gesendet. Der Benutzer muß auf das Ende der Übertragung warten, bevor er weitere Daten senden kann.
Am Terminal ankommende `Nachrichten` werden gezählt und anschließend vergessen. Meldungen wie ,,Sie behindern den Rechenbetrieb !" vom Operator interessieren den ,,Hacker" sowieso nicht.

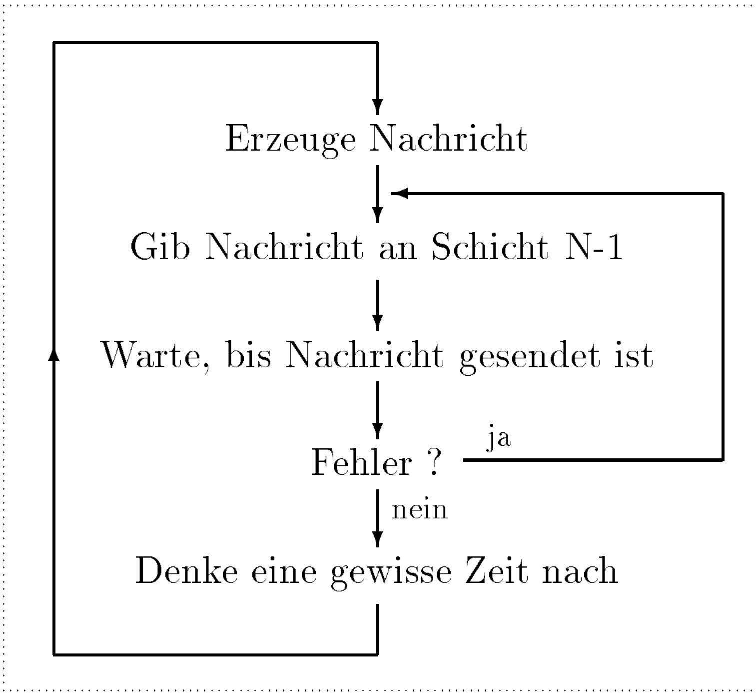


Abbildung 4.5: Die Aktionsfolge im K0-Knoten

4.4.3 Schnittstellen

Aus der Abb. 4.6 sind die Schnittstellen zur Protokollschicht *4* sowie der Datenfluß innerhalb des `K0-Moduls` ersichtlich. Die Schreibweise **(*...)()** besagt, daß die entsprechende Schnittstellenfunktion der Nachbarschicht über einen Funktionszeiger aufgerufen wird.

Die durch das `Knotenmodul` modellierte simuLan-Schicht *5* erzeugt in *k0_pdu_sended()* eine `Nachricht` und sendet diese nach einer gewissen ,,Denkzeit". Dazu wird die Zugangsfunktion l*4*_sap() zur Schicht *4* mit dem die `Nachricht` beschreibenden `EVENT` als Parameter (siehe Abschnitt 4.2.2 [`EVENT`], Seite 15) aufgerufen.
Nach dem Senden ruft die Protokollschicht *4* die Senderfunktion *k0_pdu_sended()* (über l5_pdu_sended()) auf, um den Sender von dem Ende der Übertragung in Kenntnis zu setzen und so die Erzeugung einer neuen `Nachricht` anzustoßen.
Die Funktion *k0_pdu_received()* (über l5_pdu_received()) wird von der Schicht *4* genau dann aufgerufen, wenn sie eine empfangene `Nachricht` an den dazugehörigen Empfängerknoten abliefern will.

4.4.4 Realisierung

In der Datenstruktur *K0_MESSAGE*, die in jedem `EVENT`, der eine `Nachricht` beschreibt, enthalten ist, speichert das `K0-Knotenmodul` private Informationen zu dieser `Nachricht` ab. Im Augenblick ist dieses nur die Zeit, zu der die `Nachricht` das erste Mal gesendet wird.

In dem Anhang D ist das vollständige Programmlisting zu finden.

4.4.4.1 k0_pdu_sended()

Die Prozedur *k0_pdu_sended()* modelliert das Ende einer Sendung und entspricht der Funktion *XLayer_pdu_sended()* (s. Abschnitt 4.2.5) der hier modellierten simuLan-Schicht *5*.
Nach dem Aufruf durch die Schicht *4* wird als erstes anhand `SADU`→*status* überprüft, ob die `Nachricht` fehlerfrei übertragen wurde. Dieser Status wird von der Schicht *4* gesetzt.
Im Fehlerfall wird dieser zurückgesetzt und anschließend die `Nachricht` erneut an die Schicht *4* zum Senden übergeben, indem die Funktion *nodearry[k_nr].l4_sap()* (s. Abschnitt 4.2.6) mit der `Nachricht` als Parameter aufgerufen wird.
Nach einer fehlerfreien Übertragung werden alle aktivierten Meßfunktionen des Meßpunktes

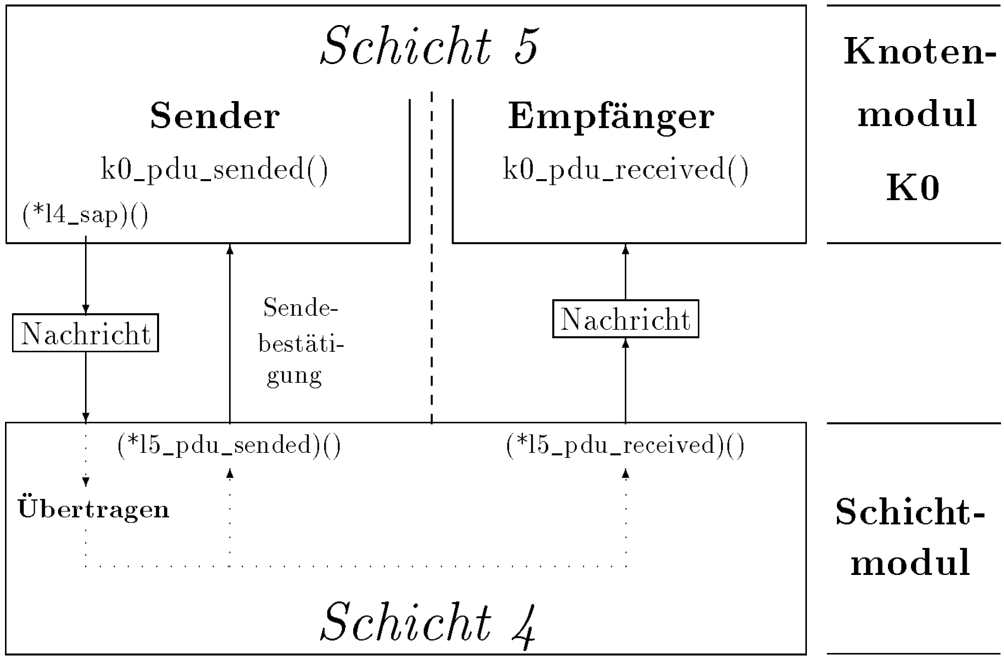


Abbildung 4.6: Schnittstellen und Datenfluß im K0-Knotenmodul

k0_send_probept über das Makro `MEASURE()` (siehe Abschnitt 4.2.7 [*Meßpunkte*], Seite 19) aufgerufen, die Anzahl gesendeter `Nachricht`en erhöht und dann der Zeitpunkt des nächsten Sendewunsches berechnet.
Dieser ergibt sich aus: `EVENT`→*event_time* + *USE_ZUFALL(inter_sende_distr)*.
Das Makro `USE_ZUFALL` (s. [Pet91]) liefert entsprechend der in *inter_send_distr* angegebenen Verteilungsfunktion eine Zeit in `ITU` zurück. Die Funktion wird vom Modellierer über die Konfigurationsdatei vorgegeben und stellt die Verteilung der „Denkzeiten" zwischen zwei Sendeaufträgen dar.

Nach dem Abspeichern der Sendezeit in *k0_message→start_time* wird die Größe der zu übertragenden `Nachricht` (in Bits) mit `USE_ZUFALL(`*msg_size_distr*`)` ermittelt. Die dabei verwendete Verteilungsfunktion *msg_size_distr* ist über die Konfiguration frei wählbar.
Nun wird über die Funktion *calc_empfaenger()* (siehe Abschnitt 4.2.6 [*NODE*], Seite 18) noch die Nummer des Empfängerknotens bestimmt und in `SADU`→*receiver* gemerkt.
Zum Schluß wird die vollständig ausgefüllte `Nachricht` über *(⋆nodearray[k_nr].l4_sap)()* (siehe Abschnitt 4.2.6 [*NODE*], Seite 18) an die Schicht *4* zum Senden übergeben.

4.4.4.2 k0_pdu_received()

Die Prozedur *k0_pdu_received()* modelliert die Ankunft einer `Nachricht` am simulierten Netzknoten und entspricht *XLayer_pdu_received()* (siehe Abschnitt 4.2.5 [*Schichtmodul*], Seite 16). Die einzige Funktion ist das Zählen der empfangenen `Nachricht`en. Die Summe wird in *k0_info→pak_received* abgespeichert, um sie am Simulationsende ausgeben zu können.

4.4.4.3 Die Rahmenfunktionen des Knotenmoduls *K0*

Die Rahmenfunktionen des Knotenmoduls `K0` sind *k0_rc()*, *k0_init()*, *k0_cleanup()*, *k0_print_info()* und *k0_print_message()*.

Die Prozedur *k0_rc()* initialisiert, entsprechend dem übergebenen Parameter *count*, *count* **identische** `Knotenmodule` vom Typ `K0`.
Dabei fallen folgende Aufgaben an:

- Falls über die Kommandozeile angegeben, alle variablen Parameter eines `K0-Moduls` mit einer Erklärung ausgeben.

- Die Datenstruktur *k0_rc_info*, in der die Werte aus der Konfiguration zwischengespeichert werden, entspricht *k_info* (siehe Abschnitt 4.2.6 [*NODE*], Seite 18) mit den Defaultwerten vorbelegen.
- Die Funktion *rcreadkeywords()* mit der Tabelle der definierten Schlüsselwörter für das `K0-Modul` aufrufen.
- Die eingelesenen Parameter auf Plausibilität prüfen.
- Die Datenstruktur `NODE` in *nodearray[]* (siehe Abschnitt 4.2.6 [*NODE*], Seite 18) und den angehängten *k_info* der *count* `K0`-Knoten entsprechend dem Inhalt von *k0_rc_info* initialisieren.

Die Aufgabe der Funktion *k0_init()* ist es, den Zyklus *DENKEN – SENDEN – DENKEN* vor Beginn des Simulationslaufes anzustoßen. Dazu wird eine komplette `Nachricht`, bestehend aus `EVENT`, `SADU`, `K0_MESSAGE` erzeugt und analog zu *k0_message_sended()* initialisiert. Anschließend wird diese neue `Nachricht` zur Übertragung an die Schicht *4* übergeben.
Zu dem Zeitpunkt, an dem diese `Nachricht` komplett gesendet wurde, wird automatisch *k0_pdu_sended()* aufgerufen, die dann das nächste ,,Denken" und Senden simuliert.

Die Funktion *k0_cleanup()* gibt, nach Beendigung eines Simulationslaufes, den Knotennamen und die Anzahl gesendeter und empfangener Nachrichten aus.

Die Funktionen *k0_print_info()* und *k0_print_message()* geben den Inhalt der für dieses Modul spezifischen Datenstrukturen *K0_INFO* bzw. *K0_MESSAGE* aus. Diese Funktionen werden für Debuggingzwecke benötigt.

4.4.4.4 Die Meßfunktion *k0_msg_delay()*

Die Funktionen *k0_msg_delay()*, *k0_msg_delay_erg()*, *k0_rc_msg_delay()* und *k0_send_probept_-printf()* gehören zu dem in *k0_pdu_sended()* eingebauten Meßpunkt *k0_send_probept*. Dabei liest *k0_rc_msg_delay()* die benötigten Parameter ein, und *k0_msg_delay_erg()* gibt die gewonnenen Ergebnisse am Ende eines Simulationslaufes aus. Falls zusätzliche Informationen ausgegeben werden sollen, z.B. Knotennummer, aktuelle Simulationszeit, …, wird die Funktion *k0_send_probept_printf()* verwendet, deren Steuerinformation vom Modellierer über die Konfigurationsdatei vorgegeben wird.
In der Meßfunktion *k0_msg_delay()* wird die Zeitdauer für die Übertragung (in `ITU`) einer `Nachricht` folgendermaßen bestimmt:

Übertragungszeit = Aktuelle Simulationszeit - Sende-Startzeit.

Für weitergehende Informationen zu Meßpunkten und ihrer Implementation siehe Abschnitt 4.2.7 [*Meßpunkte*], Seite 19 und [Pet91].

4.5 Das fertige Simulationsprogramm

In diesem Abschnitt wird die Simulation eines vollvermaschten Netzes mit mehreren Netzknoten durchgeführt. Das Netz wird dabei durch das oben beschriebene Netzmodell `PTP` und die Knoten durch das Knotenmodell `K0` beschrieben.
Es wird allerdings keine Auswertung der Ergebnisse vorgenommen, da es in diesem Kapitel um eine Einführung in die Konzepte von simuLan geht und nicht um die Durchführung eines realistischen Simulationsprojektes.

4.5.1 Beschreibung des Modells

Aus Abb. 4.7 ist die Topologie des zu simulierenden Netzwerkes ersichtlich.

Das Netzwerk besteht aus den fünf Rechnern *Hugo1, Hugo2, Hugo3, Hugo4* sowie *Claudia*, die jeder mit jedem über zwei Leitungen, eine zum Senden und eine zum Empfangen, verbunden sind.

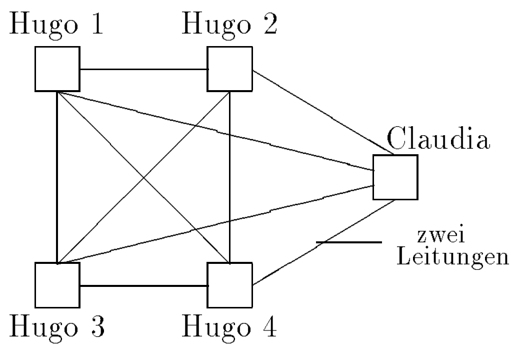


Abbildung 4.7: Das simulierte Point-To-Point-Netzwerk

Die genaue Beschreibung des modellierten Systems erfolgt parallel zur Beschreibung der benötigten Elemente der `Konfigurationsdatei`, um so einen Einblick in den Aufbau dieser Datei und die damit verbundenen Möglichkeiten zu geben.

4.5.2 Die Konfigurationsdatei

Unten ist die vollständige Konfigurationsdatei für dieses Simulationsprojekt angegeben. In den nachfolgenden Unterabschnitten werden die einzelnen Elemente näher erläutert.

```
Die Konfigurationsdatei für das  PTP-Netz mit 5  K0-Knoten.

#
## simuLan Configuration Syntax
#

## simuLan version 0.98

## -- Meta-Configuration: -- (Vorspann)

# Ergebnisdatei
FILE erg "simple_net_erg"

# Datei, in der Laufzeiten der Nachrichten von ,Claudia' gespeichert werden
FILE mess "msg_delay"

# Laufvariable für die Nachrichtengröße des Knotens Claudia
FOR int S FROM 512 TO 1536 STEP 512

## --- Body of Configuration --- (Hauptteil)

BEGINconf

ITU             1e9                     # indivisible time-units per Second
max_events      1e6                     # Max. Number of Events to simulate
max_simul_time  1000 s                  # max time to simulate
max_cpu_time    50 s                    # max cycles to succ

NODES           <<Imm Hook>>    # read the Node-Configs

   NAME "Hugo%d" COUNT 4
   LAYER5 "k0"

## Parameters for k0-node
    zw_ank_distr                        # prob. distr. for messages'
                                        # interarrival-times
```

```
        expntl 0.5 s

    msg_size_distr                      # prob. distr. for message sizes
        expntl 50 bit

    calc_empfaenger                # Function to determine the
                                   # receiver-nodenumber
        uniform_empf               # All nodes same probability as receiver

   ENDk0                               # end of k0-descr.


   NAME "Claudia"
   LAYER5 "k0"

## Parameters for k0-node
    zw_ank_distr                        # prob. distr. for messages'
                                        # interarrival-times
        expntl 0.3 s

    msg_size_distr                      # prob. distr. for message sizes
        konstant $S bit

        calc_empfaenger                 # Function to determine the
                                        # receiver-nodenumber
        listen_empf                     # receiver choosen from given
                                        # traffic-list

    traffic                             # traffic distribution for this node
        partner Hugo1 0.3
        partner Hugo2 0.4
        partner Hugo3 0.1
        partner Hugo4 0.2


    send_probept                           # used in k0_pdu_sended
        MEASURE NULL msg_delay
           FORMAT mess "%t% %R% %l%  \n" # "simul_time Receiver delay "
           STATISTIC WAIT erg
              bmva
                delete_count 50
              ENDbmva

   ENDk0                               # end of k0-descr.

# End of NODES

NET                                         # read the Net-Config
  PTP                                       # simple Point-to-Point net
    bits_per_second     1e6                 # Net Speed
    error_distr                             # ditribution of transmission
        expntl 2 s                          # errors

    min_data_len        50 bit          # Minimum size of PTP-FRAME in bits
    overhead            25 bit          # Overhead for each FRAME

    end_transmit_probept                    # Messpunkt ptp_end_transmit
        MEASURE NULL ptp_throughput
        STATISTIC NOWAIT erg
```

```
          siesta
               delete_count   500            # avoid initial bias
               values_needed  5500           # number of values to collect
       ENDsiesta
ENDptp                               # end of ptp-description
##
END
```

4.5.2.1 Der Vorspann

Hier werden die Ausgabedateien und Laufvariablen für simuLan deklariert.
In die Datei '`simple_net_erg`' mit dem symbolischen Namen '`erg`' werden nach dem Ende jedes Simulationslaufes die Ergebnisse der beiden aktivierten Statistiken (s. Abschnitte 4.5.2.4 und 4.5.2.5) geschrieben.
Die Datei '`msg_delay`' mit dem symbolischen Namen '`mess`' enthält bei Simulationsende alle in dem `Knotenmodul` *Claudia* (s. Abschnitt 4.5.2.4) gesammelten Verzögerungszeiten der gesendeten Nachrichten.

Die Laufvariable S gibt die Größe einer `Nachricht` des Knotens *Claudia* für die drei Simulationsläufe an. Beim ersten Lauf sind dies *512*, beim zweiten *1024* und beim letzten *1536* Lauf Bits.

4.5.2.2 Allgemeine Simulationsparameter

Hier werden die Parameter festgelegt, die für jeden Simulationslauf neu gesetzt werden.

ITU: Gibt die Anzahl `ITU` (siehe Abschnitt 4.2.1 [*ITU*], Seite 15) pro Sekunde und damit den minimalen Abstand zwischen zwei `Ereignissen` an. Die verwendete Auflösung ist hier *1 ns*.

max_events: Nach spätestens einer Millionen simulierten Ereignissen wird ein Simulationslauf beendet.

max_simul_time: Die maximale simulierte Zeit beträgt eintausend Sekunden. Bei Überschreitung dieser Zeit (*simul_time*≥*max_simul_time*) wird der Simulationslauf beendet.

max_cpu_time: Nach fünfzig Sekunden realer CPU-Zeit endet der Simulationslauf.

4.5.2.3 Die Knoten *Hugo*⋆

Die folgenden Zeilen aus der Konfigurationsdatei definieren die Knoten *Hugo*⋆:

```
NAME "Hugo%d" COUNT 4
LAYER5 "k0"

## Parameters for k0-node
    zw_ank_distr                       # prob. distr. for messages'
        expntl 0.5 s                   # interarrival-times

    msg_size_distr                     # prob. distr. for message sizes
        expntl 50 bit

    calc_empfaenger                # Function to determine the
                                   # receiver-nodenumber
        uniform_empf               # All nodes same probability as receiver

ENDk0                                  # end of k0-descr.
```

simuLan ermöglicht mit dem Konstrukt ,,*NAME* Hugo%d *COUNT* 4" die einfache Definition der vier **identischen** Netzknoten *Hugo1–Hugo4*.
Die Parameter dieser *K0*-Knoten mit ihren Werten sind im einzelnen:

zw_ank_distr: Dieser Parameter beschreibt die Verteilung der ,,Denkzeiten" (s. Abschnitt 4.4.1). Hier werden negativ exponential verteilte ,,Denkzeiten" mit einem Mittelwert von 0.5 Sekunden angenommen.

msg_size_distr: Die Verteilung der Größen der `Nachricht`en, die von diesem Knoten gesendet werden, wird durch diesen Parameter beschrieben. Es wird eine negative Exponentialverteilung mit dem Mittelwert von 50 Bit benutzt.

calc_empfaenger: Über diese Funktion wird der Empfängerknoten für eine `Nachricht` bestimmt. Die hier verwendete Prozedur *uniform_empf()* wählt aus **allen** im Netz vorhandenen Knoten zufällig einen aus.

4.5.2.4 Der Knoten *Claudia*

Der `K0`-Netzknoten *Claudia* wird durch folgende Zeilen in der Konfigurationsdatei vollständig definiert:

```
NAME "Claudia"
LAYER5 "k0"

## Parameters for k0-node
    zw_ank_distr                        # prob. distr. for messages'
                                        # interarrival-times
        expntl 0.3 s

    msg_size_distr                      # prob. distr. for message sizes
        konstant $S bit

    calc_empfaenger                     # Function to determine the
                                        # receiver-nodenumber
        listen_empf                     # receiver choosen from given
                                        # traffic-list

    traffic                             # the traffic-list
        partner Hugo1 0.3
        partner Hugo2 0.4
        partner Hugo3 0.1
        partner Hugo4 0.2

    send_probept                           # used in k0_pdu_sended
        MEASURE NULL msg_delay
           FORMAT mess "%t% %R% %1%  \n" # "simul_time Receiver delay"
           STATISTIC WAIT erg
              bmva
                delete_count 50
              ENDbmva

ENDk0                                   # end of k0-descr.
```

Die angegebenen Parameter haben folgende Bedeutungen und Werte:

zw_ank_distr: Es wird eine negative Exponentialverteilung der ,,Denkzeiten" mit dem Mittelwert 0.3 Sekunden gewählt.

msg_size_distr: Die von *Claudia* erzeugten `Nachricht`en haben eine **konstante** Größe während **eines** Simulationslaufes. Die Größe wird durch die `Laufvariable` *S* vorgegeben. Auf den aktuellen Wert kann mit *$S* zugegriffen werden. Im ersten Lauf sind die `Nachricht`en 512 Bit, im zweiten 1024 Bit und im letzten 1536 Bit groß.

calc_empfaenger: Der Empfänger einer gesendeten `Nachricht` wird über *listen_empf()* bestimmt. Diese Funktion wählt den Empfänger nach der über den Parameter *traffic* eingelesenen Liste aus.

traffic: Über diesen Parameter kann vom Modellierer festgelegt werden, zu welchem Knoten im Netz mit welcher Wahrscheinlichkeit *Claudia* eine `Nachricht` sendet.
In diesem Modell ist *Hugo1* mit 0.3, *Hugo2* mit 0.4, *Hugo3* mit 0.1 und *Hugo4* mit der Wahrscheinlichkeit 0.2 Empfänger einer `Nachricht`. (siehe Abschnitt 4.2.4 [*Verkehrsmatrizen*], Seite 16)

send_probept: Hier werden die `Meßfunktionen` für den dazugehörigen `Meßpunkt` *k0_send_probept* (siehe Abschnitt 4.4.4.4 [*Die Meßfunktion k0_msg_delay()*], Seite 28) mit allen benötigten Parametern definiert.
Die Zeile „*MEASURE NULL msg_delay*" aktiviert für diesen Meßpunkt die `Meßfunktion` *k0_msg_delay()*, die die Übertragungszeit einer `Nachricht` bestimmt. Der Dateizeiger `NULL` besagt, daß die einzelnen Meßwerte nicht in einer Datei gesammelt werden sollen.
Mit „*FORMAT mess "%t% %R% %l% \n"*" wird erreicht, daß für jeden gefundenen Meßwert in die Datei '`mess`' eine Zeile der Form
(simul_time) (Empfänger-name) (Meßwert)
geschrieben wird.
Zur statistischen Absicherung des Meßwertes wird die Statistik *bmva* aktiviert, die ihre Ergebnisse in die Datei '`erg`' schreibt. Das Schlüsselwort `WAIT` bewirkt, daß mindestens solange simuliert wird, bis dieses Statistikmodul genügend Werte gesammelt hat. Die ersten 50 Meßwerte werden vergessen.
Eine genauere Beschreibung des Moduls *bmva* ist [Pet91] zu entnehmen.

4.5.2.5 Die Netzbeschreibung

Das Netz wird beschrieben durch:

```
NET                                          # read the Net-Config
  PTP                                        # simple Point-to-Point net
    bits_per_second     1e6                  # Net Speed
    error_distr                              # ditribution of transmission
        expntl 2 s                           # errors

    min_data_len        50 bit       # Minimum size of PTP-FRAME in bits
    overhead            25 bit       # Overhead for each FRAME

    end_transmit_probept                     # Messpunkt ptp_end_transmit
        MEASURE NULL ptp_throughput
        STATISTIC NOWAIT erg
           siesta
                delete_count   500           # avoid initial bias
                values_needed  5500          # number of values to collect
        ENDsiesta

ENDptp                               # end of ptp-description
```

Die Parameter des Netzmoduls *PTP* sind:

bits_per_second: Die Übertragungsgeschwindigkeit des Netzwerkes wird auf $1 \cdot 10^6$ Bits pro Sekunde gesetzt.

error_distr: Die Verteilung der Zeiten zwischen zwei Übertragungsfehlern ist negativ exponential verteilt mit dem Mittelwert von 2 Sekunden.

min_data_len: Die minimale Größe des Datenbereiches eines `PTP-FRAME`s ist 50 Bit (siehe Abschnitt 4.2.6).

overhead: Für jedes `Paket` müssen zusätzlich 25 Bit Overhead übertragen werden (siehe Abschnitt 4.2.6).

end_transmit_probept: Dieser `Meßpunkt` entspricht dem Meßpunkt *ptp_probept_end_transmit* in der Funktion *ptp_end_transmit()*. Die aktivierte Meßfunktion, die die Auslastung des Netzwerks in Prozent bestimmt, ist *ptp_throughput()* (s. Abschnitt 4.3.4.5). Die Meßwerte werden nicht in eine Datei ausgegeben, sondern nur von dem Statistikmodul *siesta* ausgewertet. Dieses liefert den Mittelwert aller Meßwerte. Das Schlüsselwort `NOWAIT` besagt, daß diese Statistik keinen Einfluß auf die Simulationsdauer hat. Die ersten 500 Werte werden bei der Mittelwertbildung nicht berücksichtigt, um Verfälschungen aus der Einschwingphase zu vermeiden. Es werden maximal 5500 Werte zur Mittelwertbildung herangezogen.

Eine genauere Beschreibung der Statistikmodule sowie ihrer Parameter ist [Pet91] zu entnehmen.

4.5.3 Der Simulationsablauf

Nach dem Entwickeln der benötigten Schichtmodule und der Spezifikation des zu simulierenden Systems über die Konfigurationsdatei kann die Simulation nun angestartet werden.

Der Aufruf erfolgt mit:

simulan -p -o <output> <config>

Der optionale Parameter *-o output* gibt eine Datei an, die nach der Simulation die Ausgaben von simuLan enthält, die sonst nach *stdout* geschrieben werden. In der Datei '`config`' steht die Konfigurationsbeschreibung. Mit der Option *-p* bietet simuLan die Möglichkeit, eine vollständige Konfigurationsbeschreibung des simulierten Systems mit allen Werten (auch die Defaultwerte) ausgegeben zu bekommen.

Der Aufruf *simulan -h* liefert alle möglichen Aufrufparameter mit einer Kurzbeschreibung. Eine ausführliche Beschreibung dieser Parameter und ihrer Verwendung, z.B. zu Debuggingzwecken, ist [Pet91] zu entnehmen.

Kapitel 5

Das Netzmodell *CSMA*

5.1 Einführung

Das in diesem Kapitel entwickelte Protokollschichtmodell *CSMA* modelliert Netze, die nach dem Zugriffsverfahren *1-persistent CSMA/CD*[1] arbeiten. Eine Erweiterung auf *p-persistent CSMA/CD* wäre relativ einfach zu implementieren.
Ein Ziel dieser Arbeit ist die möglichst realitätsgetreue Simulation dieses Zugriffsverfahrens, besonders der Kollisionserkennung und -behandlung, ohne vereinfachende Annahmen. Die Validierung erfolgt durch Modellierung eines *Ethernet* und anschließendem Vergleich der erhaltenen Meßergebnisse mit Werten aus der Literatur.
Nach einer Einführung in die Grundlagen des wohl bekanntesten LANs, das nach dem `CSMA/CD` Verfahren arbeitet, dem `Ethernet`, wird das `CSMA`-Modul ausführlich beschrieben. Anschließend folgt ein Abschnitt über die Validierung, in dem auch einige Simulationsergebnisse angegeben sind. Im letzten Abschnitt wird dann ein modifiziertes `Ethernet` mit einer Übertragungsrate von 20 Megabit modelliert und der Einfluß dieser Modifikation auf den Durchsatz des Netzes dargestellt.

5.2 Das Vorbild: *Ethernet*

Der Vorgänger des beim *Ethernet* verwendeten Zugriffsprotokolls *CSMA/CD* war Abramsons *ALOHA*-System, das auf Hawaii Verwendung findet. Später wurde die Möglichkeit des *Carrier-Sense* zu diesem Protokoll hinzugefügt. Im Jahr 1976 baute die Firma XEROX ein `CSMA/CD`-System , um mehr als 100 Rechner über ein 1 km langes Kabel zu verbinden (Metcalfe und Boggs, 1976). Dieses System wurde `Ethernet` genannt. Das XEROX `Ethernet` war so erfolgreich, daß XEROX, DEC und Intel einen Standard für das heute weit verbreitete 10 Mbit `Ethernet` entwarfen. Dieser bildete die Basis des *IEEE 802.3*-Standards (siehe Abschnitt 3.5 [*NORMEN*], Seite 9).
Die beiden Standards unterscheiden sich insoweit, daß `802.3` eine ganze Familie von `1-persistent CSMA/CD` Systemen beschreibt, die mit Übertragungsgeschwindigkeiten von 1 bis 10 Mbit pro Sekunde auf unterschiedlichen Medien arbeiten, während `Ethernet` ein spezielles Produkt ist, daß den `802.3`-Standard implementiert. [Tan89]

Das `Ethernet` ist das bevorzugte Verfahren für lokale Netze mit Bustopologie. Dabei kann ein solcher Bus aus bis zu 5 Segmenten von jeweils maximal 500 Metern Länge bestehen, die durch *Repeater* miteinander verbunden sind. Ein `Repeater` empfängt, verstärkt und sendet ankommende Signale in beiden Richtungen.

Die übliche Konfiguration für einen Rechner am `Ethernet` (Thick Ethernet) ist aus Abb. 5.1 ersichtlich. Der *Transceiver*, der direkt mit dem Übertragungsmedium verbunden ist, enthält die Elektronik, die für das physikalische Senden von Bits sowie die Carrier- und Kollisionserkennung

[1] Carrier Sense Multiple Access with Collision Detection

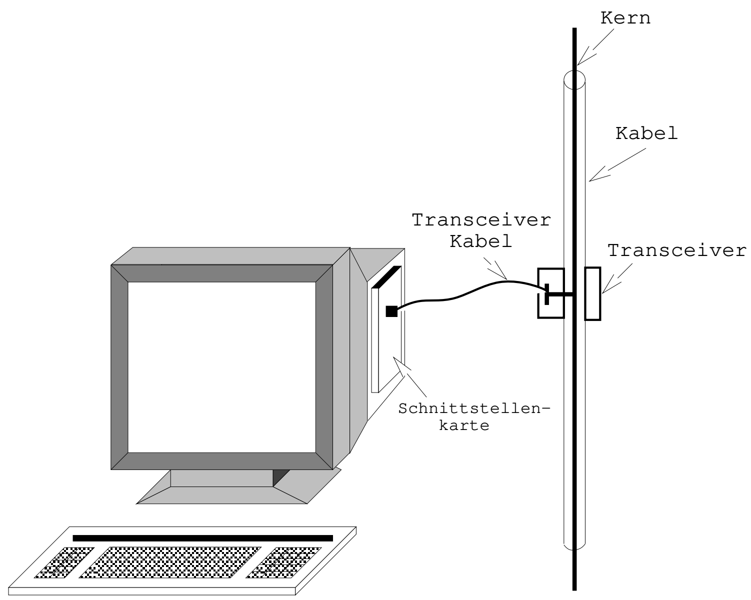


Abbildung 5.1: Anschlußkonfiguration Rechner – Ethernet

benötigt wird. Über das ,,Transceiver Kabel" ist der `Transceiver` mit der *Schnittstellenkarte* im Computer verbunden. Die Funktionen dieser Karte sind:

- Verpacken der zu sendenden Daten in einem `Frame`, wozu auch das Eintragen der Sender- und Empfängeradresse gehört.
- Berechnen der Prüfsumme von ausgehenden `Frame`s.
- Senden von `Frame`s an den und Empfangen von `Frame`s von dem Transceiver.
- Prüfen der Prüfsumme ankommender `Frame`s.

Die Funktionen Carrier- und Kollisionserkennung des Transceivers und die bisher beschriebenen Aufgaben der Interfacekarte sind in der *ISO/OSI*-Schicht *2a* (MAC) (siehe Abschnitt 3.5 [*NORMEN*], Seite 9) anzusiedeln und somit durch das unten entwickelte `Netzmodul CSMA` zu modellieren.

Neuere Schnittstellenkarten verwalten zusätzlich mehrere Puffer für eingehende `Frame`s, eine Warteschlange von zu sendenden Puffern, sowie den DMA-Transfer von und zum Hostrechner.

Wie oben schon erwähnt, arbeitet das `Ethernet` nach dem Zugriffsverfahren `1-persistent CSMA/CD`, das nun beschrieben wird.

Nach dem Verpacken der zu übertragenden Daten in einem `Frame` prüft der Sender, ob der Kanal frei ist, das heißt, ob für eine bestimmte Zeit kein Trägersignal (Carrier) vorhanden war. Falls der Kanal belegt ist, wartet die Station, bis er frei wird (*defer*), andernfalls beginnt sie sofort mit dem Senden. Wenn sich die Übertragungen von zwei oder mehr Stationen überlappen, tritt eine *Kollision* auf. Sobald eine sendende Station eine Kollision erkennt, gibt sie eine Sequenz von 32 Bit, das *JAM-Signal*, auf das Netz. So wird sichergestellt, daß die Kollision von allen anderen sendenden Stationen am Netz erkannt wird.
Im Anschluß an das Senden des `JAM-Signals` wird die Übertragung abgebrochen und nach einem zufälligen Zeitraum ein neuer Übertragungsversuch gestartet (,,Backoff"). Dieser Zeitraum wird mit jedem Sendeversuch größer, um so die Netzlast, die von diesem Knoten erzeugt wird, zu verringern. Die Dauer dieses ,,Wartens" wird nach dem *truncated binary exponential backoff*-Verfahren bestimmt.
Dieses Verfahren funktioniert folgendermaßen: n sei die Anzahl der bisherigen Sendeversuche für den `Frame`.
Die Verzögerungszeit ist ein ganzzahliges Vielfaches der *Slottime*, die bei dem `Ethernet` normalerweise 512 Bit-Zeiten groß ist. Die `Slottime` ist so gewählt, daß sie größer als die doppelte

maximale Signallaufzeit zwischen zwei Netzknoten in einem maximal ausgebauten `Ethernet` ist. Nach dem kollisionsfreien Senden von `Slottime` Bits kann demnach keine Kollision mehr auftreten.
Die Anzahl der `Slottimes`, die vor dem n-ten Sendeversuch gewartet werden muß, wird aus dem Intervall $[0 \ldots 2^k]$ mit $k = min\ (n\ ,\ backoff\_limit)$ gewählt.
Die maximale Anzahl von Übertragungsversuchen für ein `Frame`, und damit der maximale Wert von n, wird durch *attempt_limit* festgelegt. Für das `Ethernet` gilt *attempt_limit* = 16. Die Variable *backoff_limit*, mit *backoff_limit*$\leq n$, legt die maximale Wartezeit zwischen zwei Sendeversuchen fest und hat beim `Ethernet` den Wert 10.

Zwischen dem Ende einer Übertragung und dem Beginn der nächsten muß eine Pause der Größe *Inter Frame Gap* eingehalten werden. Diese dient als Erholungszeit für andere `CSMA/CD`-Teilschichten und für das Übertragungsmedium. Normalerweise beträgt diese Zeitdauer beim `Ethernet` $9.6\mu s$. Dieses entspricht 96 Bit-Zeiten.

Der Aufbau eines `Ethernet-Frame` ist in Abb. 5.2 zu sehen:

Ethernet	**Bit**	**Byte**	**Summme min**	**Summe max**
Preamble	56	7	7	
SFD	8	1	8	
Destination Address	48	6	14	
Source Address	48	6	20	
Length	16	2	22	
LLC min. Data	368	46	68	
LLC max. Data	12000	1500		1522
CRC	32	4	72	1526

$\rightarrow$**Overhead** = 208 Bit = 26 Byte.

Abbildung 5.2: Format eines `Frames` im `Ethernet`

Die einzelnen Felder eines `Frames` haben folgende Bedeutung:

- Die **Preamble** dient zur Synchronisation der Stationen.
- Das Feld **SFD** (*Start of Frame Delimiter*) kennzeichnet den Beginn des `Frames`.
- **Destination Address** spezifiziert die Empfängeradresse(n).
- **Source Address** enthält die Senderadresse.
- In **Length** steht die Größe des zu übertragenden *LLC* (*Logical Link Control*)- Datenblocks (in Bytes).
- **LLC Data** enthält die zu sendenden Daten.
- Das Feld **PAD** enthält zusätzliche Füllinformation, falls die Nachrichtengröße kleiner als die minimale erlaubte Größe *min_data_len* ist. Für `Ethernet` sind dies 46 Byte.
- **FCS** (*Frame Check Sequence*) enthält die *CRC* (*Cyclic Redundancy Check*)-Prüfsumme über die Felder `Destination Address` bis einschließlich `PAD`.

Ein `Frame` muß mindestens 64 Byte groß sein, damit eventuelle Kollisionen von allen Rechnern am Netz erkannt werden können. Die maximale `Frame`-Größe ist 1526 Byte. Damit können in einem `Frame` bis zu 1500 Byte „Nutzdaten" transportiert werden.

Wie aus der Abb. 5.2 ersichtlich, ist der Overhead für jede zu sendende Nachricht der `LLC`-Schicht 26 Byte. Davon stammen 18 Byte aus der `MAC`-Schicht und 8 Byte aus der physikalischen Schicht.

5.3 Modellierung des *CSMA/CD*-Protokolls

In diesem Abschnitt wird zunächst näher auf die mit dem `CSMA/CD`-Verfahren verbundenen Kollisionen eingegangen. Anschließend wird das verwendete Simulationsmodell mit den dazugehörigen Ereignissen entwickelt.

In der Literatur ([Mac87], [Tan89], [Lan88]) wird bei der Modellierung des `Ethernet` häufig davon ausgegangen, daß die Signallaufzeiten, die sich aus der Topologie des Netzwerks ergeben, keinen signifikanten Einfluß auf die zu bestimmenden Leistungswerte (Durchsatz, Verzögerungszeiten, ...) haben. In [GT88] und [BMK88] wird dagegen gezeigt, daß die Netztopologie sehr wohl die oben genannten Faktoren beeinflußt. In dieser Arbeit werden Vereinfachungen dieser Art nicht gemacht, sondern im Gegenteil großer Wert auf die Modellierung dieser Aspekte gelegt.

Die wichtigsten Probleme bei der Modellierung des `CSMA/CD`-Verfahrens sind:

- Wie sollen die Signallaufzeiten auf dem Kabel simuliert werden?
- Wie kann festgestellt werden, ob zu einem gegebenen Zeitpunkt für eine Station der Kanal frei oder belegt ist?
- Auf welche Weise läßt sich eine Kollision mit allen unten aufgeführten Aspekten darstellen?
- Wie wird das `Backoff`-Ereignis mehrerer Stationen mit dem anschließenden erneuten Senden modelliert?

Bevor auf die Lösung dieser Fragen eingegangen werden kann, wird in Abschnitt 5.3.1 der Ablauf einer Kollision näher beschrieben. In Abschnitt 5.3.2, Seite 39 wird das genaue Simulationsmodell des CSMA/CD-Verfahrens entwickelt. Im letzten Abschnitt (5.3.3, Seite 44) wird dann eine optimierte Version dieses Modells beschrieben.

5.3.1 Der zeitliche Ablauf einer Kollision

Eine *Kollision* zweier `Frames` auf dem Netz kann nur auftreten, wenn eine Station in einem sehr kurzen Intervall nach einer anderen zu senden beginnt. Dieses Intervall wird *Kollisionsfenster* genannt.
Die Abb. 5.3 zeigt die bei einer Kollision auftretenden Ereignisse im einzelnen. Die X-Achse repräsentiert die Zeit und die Y-Achse den physikalischen Abstand τ zwischen den beiden Stationen $\mathbf{S_1}$ und $\mathbf{S_2}$.

Der Übertragungskanal wird zur Zeit $\mathtt{t} \leq \mathtt{t}_1$ als frei (*idle*) angenommen. T_{jam} ist die Zeitdauer des `JAM`-Signals, T_{ifg} die Länge des `Inter Frame Gap`.
Zum Zeitpunkt $\mathtt{t_1}$ beginnt $\mathbf{S_1}$ mit dem Senden ihres `Frames`. Aufgrund der endlichen Ausbreitungsgeschwindigkeit elektrischer Signale im Kabel (ca. 0.8 c) erreicht diese Nachricht $\mathbf{S_2}$ nicht vor dem Zeitpunkt $\mathtt{t_3}$. Währenddessen beginnt $\mathbf{S_2}$ zur Zeit $\mathtt{t_2}$ zu senden. Bei $\mathtt{t_3}$ erreicht das Übertragungssignal von $\mathbf{S_1}$ die Station $\mathbf{S_2}$. Diese erkennt, daß eine Kollision aufgetreten ist und sendet das `JAM`-Signal. Im Anschluß daran beginnt $\mathbf{S_2}$ bei $\mathtt{t_5}$ mit dem `Backoff`.
Der `Frame` der Station $\mathbf{S_2}$ erreicht $\mathbf{S_1}$ zum Zeitpunkt $\mathtt{t_4}$. Nun erkennt auch diese das Auftreten einer Kollision, sendet ihrerseits zunächst das `JAM`-Signal und beginnt bei $\mathtt{t_6}$ mit dem `Backoff`. Da $\mathbf{S_2}$ erst bei $\mathtt{t_8}$ den Kanal als frei erkennt, kann der nächste Sendeversuch frühstens bei $\mathtt{t_8} + T_{ifg}$ stattfinden. Für $\mathbf{S_1}$ ergibt sich $\mathtt{t_7} + T_{ifg}$ als der frühste mögliche Sendezeitpunkt.

Die `Frames` der Stationen $\mathbf{S_1}$ und $\mathbf{S_2}$ kollidieren nur, wenn das Intervall $\mathtt{t_2}$ - $\mathtt{t_1}$ kleiner als die Signallaufzeit zwischen $\mathbf{S_1}$ und $\mathbf{S_2}$ ist, andernfalls wird der Kanal von $\mathbf{S_2}$ als belegt erkannt und gewartet, bis dieser wieder frei ist.
Der ungünstigste Fall einer Kollision ist der, daß das Signal der Station $\mathbf{S_1}$ die Station $\mathbf{S_2}$ gerade dann erreicht, wenn diese anfängt zu senden ($\mathtt{t_2} \approx \mathtt{t_3}$). Unter diesen Bedingungen sendet $\mathbf{S_1}$ für die Zeit $2\tau + T_{jam}$ und $\mathbf{S_2}$ für die Zeit T_{jam}, bevor ihre `Backoff`-Wartezeit beginnt.

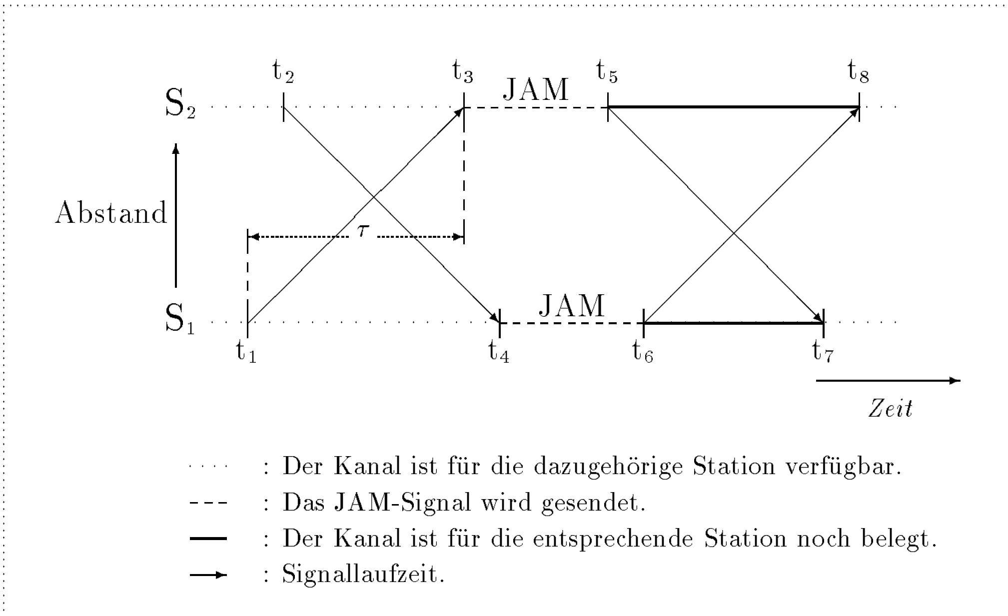


Abbildung 5.3: Kollision bei 2 Stationen

5.3.2 Das genaue Simulationsmodell

Ein System wird durch eine Menge möglicher Systemzustände und die Zustandsübergänge zwischen diesen beschrieben. Angestoßen werden die Zustandsänderungen durch *Ereignisse*. Ein Beispiel ist das Freiwerden des Übertragungskanals, das alle darauf wartenden Rechner veranlaßt, mit dem Senden ihres `Frame`s zu beginnen.
Bei der in simuLan verwendeten ereignisorientierten Simulation ist der aktuelle Zustand des Systems in den Datenstrukturen und den nächsten zu simulierenden Ereignissen enthalten. Die Aktionen, die zu einer bestimmten Zustandsänderung gehören, werden durch eine zu dem jeweiligen auslösenden Ereignis gehörende *Ereignisfunktion* und deren Parameter beschrieben.

Eine Zusammenfassung der in dem `CSMA-Netzmodul` zu simulierenden Aufgaben der Protokollschicht *1* der jeweiligen Stationen ergibt:

- Vor Sendebeginn wird gewartet, bis der Kanal als frei erkannt wird.
- Nach dem Start der Übertragung muß geprüft werden, ob eine Kollision auftritt.
- Falls es zu einer Kollision kommt, wird `JAM` gesendet und anschließend in den `Backoff`-Zustand gewechselt.
- Nach einer fehlerfreien Übertragung ist der Kanal freizugeben und die Zeitspanne `Inter Frame Gap` abzuwarten, bevor erneut gesendet werden darf.
- Bei einem `Backoff` wird der nächste Sendeversuch entsprechend der berechneten `Backoff`-Wartezeit verzögert.

Jeder der genannten Punkte entspricht einer in sich abgeschloßenen Aktionenfolge zu einem festgelegten Zeitpunkt, die durch ein bestimmtes Ereignis ausgelöst wird. Es bietet sich an, diese jeweils durch eine spezifische Funktion zu modellieren.

In der Abb. 5.4 sind die Zustände des in dieser Arbeit realisierten CSMA/CD-Modells mit allen definierten Zustandsübergängen und den dazugehörigen auslösenden Ereignissen angegeben.

Dabei haben die Zustände die in Tabelle 5.1 angegebene Bedeutung:

Die in dem CSMA-Modell definierten Ereignisse sind:

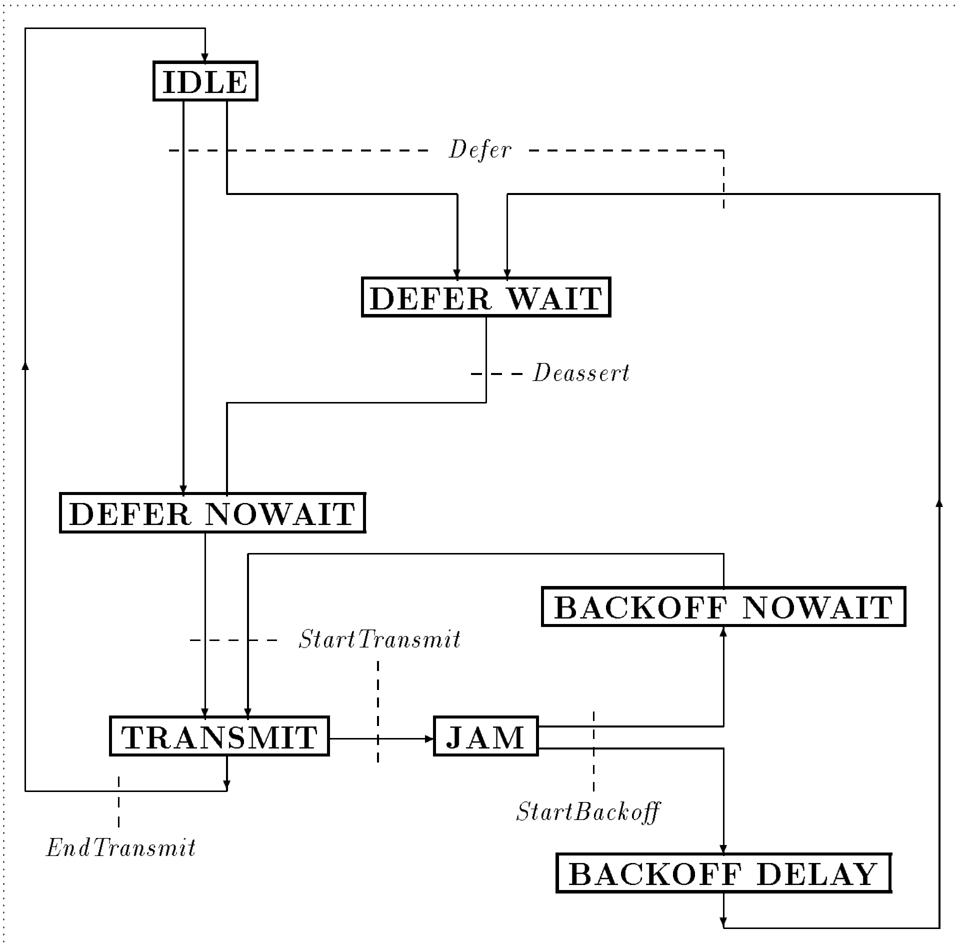


Abbildung 5.4: Zustände und Ereignisse im CSMA-Modell

Zustand	Erläuterung
IDLE	Es gibt nichts zu tun.
DEFER NOWAIT	Ein Sendeauftrag ist eingetroffen und der Kanal ist frei.
DEFER WAIT	Ein Sendeauftrag ist eingetroffen und der Kanal ist belegt. Es wird auf das Freiwerden des Kanals gewartet.
TRANSMIT	Eine Übertragung läuft.
JAM	Es ist eine Kollision eingetreten, die bearbeitet wird.
BACKOFF NOWAIT	Nach einer Kollision wurde eine `Backoff`- Wartezeit von Null `Slottimes` (siehe Abschnitt 5.2 [*JAM*], Seite 36) ermittelt.
BACKOFF DELAY	Die berechnete Backoff-Wartezeit ist größer als Null.

Tabelle 5.1: Die Zustände des *CSMA-Modells*

- Das Ereignis Defer entspricht dem Eintreffen eines Sendewunsches (neu oder nach dem Ablauf einer `Backoff`-Wartezeit). Daraufhin wird der Kanalzustand überprüft. Falls der Kanal frei ist, ist der nächste Zustand DEFER NOWAIT, andernfalls DEFER WAIT.
- StartTransmit entspricht dem Ereignis, daß ein `Frame` von der Station gesendet werden soll und der Kanal von dieser Station als frei erkannt wurde. Es können allerdings schon andere Übertragungen laufen, die aber durch die Signalverzögerung von der aktuellen Station noch nicht feststellbar sind. Die Folgezustände sind TRANSMIT oder JAM. TRANSMIT bedeutet, daß noch **keine** andere Station bisher zu senden begonnen hat. Nach JAM wird verzweigt, wenn es durch die neu angestartete Sendung zu einer Kollision kommt.
- Das Ereignis EndTransmit tritt bei erfolgreicher Beendigung einer Übertragung ohne Kollision ein.

- Das Ereignis StartBackoff wird nach dem Ende des JAM-Signals für jede an einer Kollision beteiligte Station erzeugt. Die möglichen Folgezustände sind BACKOFF NOWAIT und BACKOFF DELAY. Der erste wird angenommen, wenn die Backoff-Wartezeit Null ist, andernfalls wird in den zweiten Zustand übergegangen.
- Das Ereignis Deassert wird ausgelöst, wenn die letzte sendende Station den Kanal freigibt.

Die getroffene Zuordnung zwischen Ereignis und Ereignisfunktion ist:

Defer - *csma_defer()*
StartTransmit - *csma_start_transmit()*
EndTransmit - *csma_end_transmit()*
StartBackoff - *csma_start_backoff()*
Deassert - *csma_deassert()*.

Der Rest dieses Abschnitts zeigt die grundlegenden Probleme, die sich bei der Modellierung des CSMA/CD-Verfahrens ergeben auf, und gibt die bei der Realisierung verwendeten Methoden zur Lösung derselben an. Die Implementierung wird dann in Abschnitt 5.4 beschrieben.

Die größte Schwierigkeit bei der Simulation des CSMA/CD-Verfahrens stellt die Signallaufzeit dar. Diese bewirkt, daß sich die durch ein Ereignis angestoßenen Aktionen an verschiedenen Stellen im simulierten Kanal zu unterschiedlichen Zeitpunkten auswirken.
Das beste Beispiel ist das Ereignis StartTransmit, das einen Übertragungsstart anstößt. Die Zeit, bis das erzeugte Übertragungssignal die anderen Stationen am Kanal erreicht und damit eventuell zur Erkennung einer Kollision führt, hängt von dem physikalischen Abstand dieser Stationen zur Senderstation ab.
Eine Frage, die sich daraus ergibt, ist: ,,Wie wird erreicht, daß alle Stationen zum richtigen Zeitpunkt, wenn das Signal die Station erreicht, von dieser anderen Übertragung erfahren und entsprechend reagieren?"

Ein weiteres Problem ist, daß eingeplante Ereignisse wieder gelöscht und durch andere ersetzt werden müssen. In der Ereignisfunktion *csma_start_transmit()* wird zum Beispiel für den Zeitpunkt des Übertragungsendes ein Ereignis EndTransmit in die Liste zukünftiger Aktionen eingefügt. Falls es nun zu einer Kollision kommt, muß dieses Ereignis entfernt und stattdessen ein Ereignis StartBackoff für die dazugehörige Station eingeplant werden. Das EndTransmit-Ereignis muß in jedem Fall zunächst eingeplant werden, da bei Beginn der Übertragung noch nicht bekannt ist, ob auch andere Stationen senden wollen. Das Löschen und Ersetzen ist, insbesondere bei vielen eingeplanten Ereignissen, eine **sehr** zeitaufwendige Aktion und sollte deshalb so selten wie möglich verwendet werden.

Zwei Lösungsmöglichkeiten für diese Probleme sind:

1. In *csma_start_transmit()* wird für jede Station im Netz ein Ereignis eingeplant, das dieser mitteilt, daß von der aktuellen Station ein Übertragungssignal angekommen ist. Der Zeitpunkt dieses Ereignisses läßt sich einfach aus der Summe der aktuellen Simulationszeit und der Signallaufzeit bestimmen. Falls die Station, die dieses Ereignis bekommt, gerade sendet, führt dieses dann zu der Kollisionsbehandlung. In dieser muß das eingeplante Ereignis EndTransmit gelöscht und ein neues Ereignis StartBackoff eingeplant werden.
2. In der Variablen *channel* wird der Zustand des Kanals, FREI oder BELEGT, vermerkt. Der Kanal wird dann als BELEGT gekennzeichnet, wenn mindestens eine Station eine Übertragung begonnen hat. Eine zweite Variable *first_collision* enthält Information darüber, ob im aktuellen Kollisionsfenster (siehe Abschnitt 5.3.1 [*Kollision*], Seite 38) schon vorher eine Kollision erkannt wurde oder nicht. In *csma_start_transmit()* werden dann, entsprechend dem Wert von *channel*, folgende Aktionen durchgeführt:

FREI: Über die Größe des zu sendenden Frames wird der Zeitpunkt des Übertragungsendes bestimmt und für diese Zeit ein Ereignis EndTransmit eingeplant. Außerdem werden die Variablen *channel* auf BELEGT und *first_collision* auf TRUE gesetzt.

`BELEGT`: Hier muß nun über *first_collision* entschieden werden, ob es sich um die erste oder eine spätere Kollision in diesem Kollisionsfenster handelt.
Handelt es sich um die erste Kollision, muß das Ereignis EndTransmit für den **ersten** Sender gelöscht und durch ein Ereignis StartBackoff zu einem neu berechneten Zeitpunkt ersetzt werden. Für die zweite Station, die die Kollision verursacht hat, wird anschließend auch ein Ereignis StartBackoff eingeplant.
Im zweiten Fall muß „nur" der Zeitpunkt der Kollisionserkennung ermittelt und ein Ereignis StartBackoff für diese Zeit für die entsprechende Station eingeplant werden.

Die erste Lösung erfordert die Simulation **sehr** vieler Ereignisse. Insbesondere die Benachrichtigung aller Stationen vom Start einer Übertragung erfordert viel Rechenaufwand, wobei ein Großteil der Ereignisse eigentlich unnötig ist, da die meisten angesprochenen Stationen gerade nicht senden wollen. Auch die Berechnung des Zeitpunktes, an dem eine Station erkennt, daß der Kanal wieder frei ist, ist ein Problem.

Die zweite Methode erfordert sehr viel weniger Rechenaufwand für die Ereignisverwaltung. Der Rechenaufwand für das Ereignis StartTransmit ist allerdings deulich größer, da hier der Zeitpunkt der Kollisionserkennung berechnet werden muß. Auf die Art und Weise, wie das geschieht, wird später noch eingegangen. Ein ähnlicher Aufwand ist allerdings auch für die Ereignisfunktion zu erwarten, die durch das „Benachrichtigungsereignis" aus 1) aktiviert wird. Die Bestimmung der Zeit, zu der der Kanal wieder frei wird, ist mit dem gleichen Aufwand wie bei Lösung 1) verbunden.

Aufgrund der zu erwartenden besseren Performanz wurde die Lösung 2) implementiert, zumal für diese Methode eine Vereinfachung vorgenommen werden kann, die eine deutliche Reduzierung der Programmlaufzeit zur Folge hat (s. Abschnitt 5.5.5.1, Seite 72).

Mit Hilfe des in Abb. 5.5 angegebenen Szenarios wird nun die gewählte Methode näher beschrieben.

Die X-Achse repräsentiert die Zeit und die Y-Achse den physikalischen Abstand (Signallaufzeit in `ITU`) (siehe Abschnitt 4.2.1 [*Darstellung der Zeit*], Seite 15) zwischen den Stationen **A**, **B** und **C**. Der Abstand zwischen **A** und **C** beträgt 80 `ITU`, zwischen **C** und **B** 100 `ITU` (siehe Abschnitt 4.2.1 [*ITU*], Seite 15).

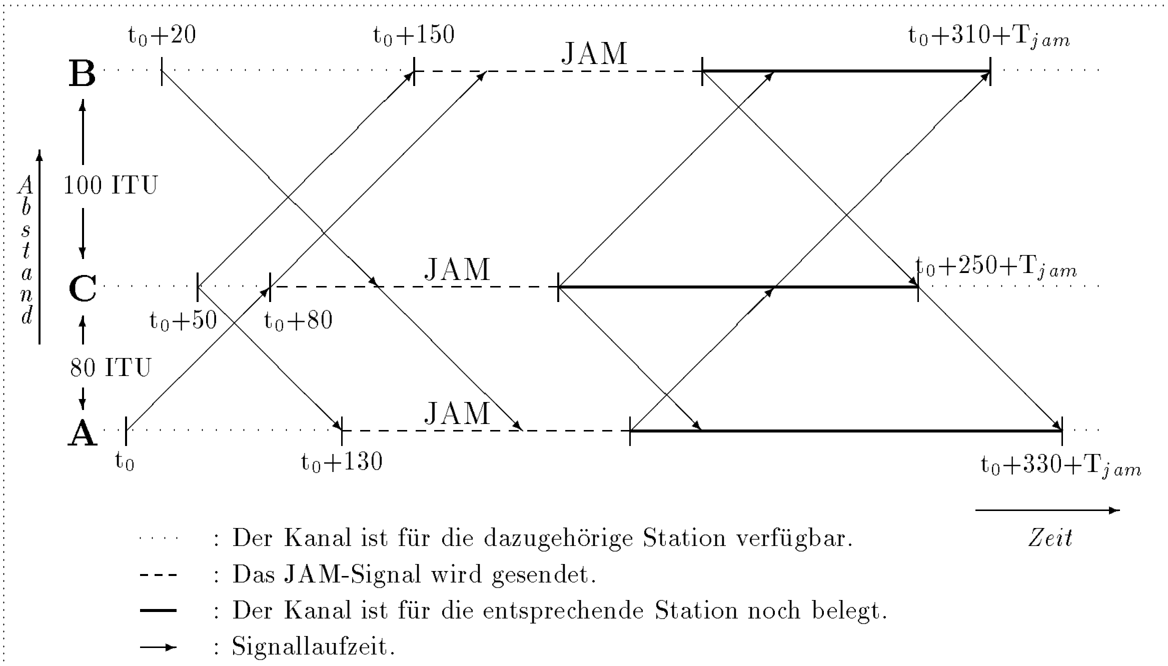


Abbildung 5.5: Kollision bei mehr als zwei sendenden Stationen

Der Übertragungskanal wird zur Zeit $\mathtt{t}<\mathtt{t}_0$ als frei (*idle*) angenommen.
Von folgendem Szenario wird ausgegangen:

- Station **A** sendet bei $\mathtt{t}_0$ ein `Frame` der Länge 10^6 `ITU`.
- Station **B** sendet bei $\mathtt{t}_0 + 20$ `ITU`.
- Station **C** sendet bei $\mathtt{t}_0 + 50$ `ITU`.

Die sich aus diesem Ablauf ergebenden Zeiten für Kollisionserkennung, Beginn des Backoff und Erkennen, daß der Kanal frei ist, werden nun einmal für jede Station vollständig angegeben und dann einzeln hergeleitet.
Eine Station hört, wegen der Signallaufzeiten, die jeweils anderen Stationen zu den Zeitpunkten:

A hört **C** bei:	$\mathtt{t}_0 + 130$	`ITU`
A hört **B** bei:	$\mathtt{t}_0 + 200$	`ITU`
B hört **C** bei:	$\mathtt{t}_0 + 150$	`ITU`
B hört **A** bei:	$\mathtt{t}_0 + 180$	`ITU`
C hört **A** bei:	$\mathtt{t}_0 + 80$	`ITU`
C hört **B** bei:	$\mathtt{t}_0 + 120$	`ITU`.

Die Zeit, zu der eine Station die Kollision erkennt und beginnt das `JAM`-Signal zu senden, ergibt sich aus dem Zeitpunkt, an dem das erste Signal einer anderen Station bei dieser Station eintrifft.
Dies sind:

$$\begin{aligned} &\mathbf{A}: \min(130, 200) \implies 130 \\ &\mathbf{B}: \min(150, 180) \implies 150 \\ &\mathbf{C}: \min(80, 120) \implies 80. \end{aligned}$$

Sobald eine Station ihr `JAM` gesendet hat, geht sie in den Backoff-Zustand. Station **B** sendet bis $\mathtt{t}_0 + T_{jam} + 150$ das `JAM`-Signal und ist damit die am längsten sendende Station. Der früheste Zeitpunkt des nächsten Sendeversuches ergibt sich aus der Summe der Zeit, zu der die Station erkennt, daß der Kanal frei ist, und T_{ifg} (siehe Abschnitt 5.2 [*JAM*], Seite 36).
Diese Zeiten sind im einzelnen:

$$\begin{aligned} &\mathbf{A}: t_0 + 150 + T_{jam} + L_{BA} + T_{ifg} \\ &\mathbf{B}: \max(TA_{last}, TC_{last}) + L_X + T_{ifg} \\ &\mathbf{C}: t_0 + 150 + L_{BC} + T_{jam} + T_{ifg} \end{aligned}$$

(TA_{last} bzw. TC_{last} ist der Zeitpunkt, zu dem **A** bzw. **B** zu senden aufhören. Dies ist $\mathtt{t}_0 + T_{jam} + 130$ bzw. $\mathtt{t}_0 + T_{jam} + 80$. L_X ist die dazugehörige Laufzeit zur Station **B**.)

Um die Berechnung der Zeiten durchführen zu können, wird eine Liste aller Sender in einem Kollisionsfenster mit den dazugehörigen Startzeiten ihrer Übertragung geführt. In der Variablen *channel* wird der Zustand des Kanals vermerkt und in *first-collision*, ob es sich um die erste bemerkte Kollision in diesem Kollisionsfenster handelt. Die Signallaufzeiten zwischen zwei Knoten werden über zur Programmlaufzeit wählbare Funktionen bestimmt (siehe Abschnitt 5.4.16 [*calc_laufzeit-Funktionen*], Seite 56).

Wenn Station **A** zu senden beginnt, wird *channel* auf `BELEGT` und *first_collision* auf `TRUE` gesetzt. Zusätzlich wird für die Zeit $t_0 + 10^6$ das Ereignis EndTransmit eingeplant. Dieses initiiert auch die Freigabe des Kanals nach Abschluß der Übertragung (Ereignis Deassert). Zum Schluß wird noch die Nummer von **A** sowie die aktuelle Simulationszeit in der Liste der Sender vermerkt.

Zum Zeitpunkt $\mathtt{t}_0 + 20$ beginnt Station **B** zu senden (StartTransmit). Da der Kanal schon belegt ist, kommt es zur ersten Kollision.
Nun wird für Station **A** das Ereignis EndTransmit gelöscht und ein Ereignis StartBackoff für $\mathtt{t}_0 + T_{jam} + 200$ eingeplant. Diese Zeit ergibt sich aus der aktuellen Simulationszeit + Signallaufzeit $\mathtt{L}_{\mathtt{BA}} + T_{jam}$. Auch für Station **B** wird ein Ereignis StartBackoff eingeplant, allerdings schon für $\mathtt{t}_0 + T_{jam} + 180$, da **B** das Übertragungssignal von **A** schon bei $\mathtt{t}_0 + 180$ hört. Anschließend wird der Name von **B** und die aktuelle Simulationszeit in die Liste der in diesem Kollisionsfenster sendenden Stationen aufgenommen.
Zum Schluß muß noch ein Ereignis Deassert in die Ereignisliste eingefügt werden. **A** ist die letzte sendende Station, so daß sich die Zeit des Deassert-Ereignisses zu $\mathtt{t}_0 + T_{jam} + 200$ ergibt.

Der Sendebeginn der Station **C** bei $\mathtt{t_0}+50$ zeigt eine neue Schwierigkeit bei der Simulation auf. Für **A** und **B** ist das Ereignis StartBackoff für $\mathtt{t_0}+T_{jam}+200$ bzw. $\mathtt{t_0}+T_{jam}+180$ eingeplant, da der zusätzliche Sendebeginn von **C** nicht bekannt war. Nun muß für jede dieser Stationen geprüft werden, ob sie die Kollision nun früher erkennt, und wenn ja, muß das Ereignis StartBackoff auf die neue Zeit umgeplant werden. Parallel zu dieser Prüfung muß der Zeitpunkt bestimmt werden, an dem **C** die Kollision erkennt. Abschließend wird, falls nötig, die Zeit des Ereignisses Deassert korrigiert. Die Zeit, zu der **A** bzw. **B** das Signal der Station **C** hört, ist $\mathtt{t_0}+130$ bzw. $\mathtt{t_0}+150$. Da sowohl **A** als auch **B** die Kollision nun früher erkennen, ergibt sich die neue Zeit für das Ereignis StartBackoff zu $\mathtt{t_0}+T_{jam}+130$ bzw. $\mathtt{t_0}+T_{jam}+150$.
Die Station **C** hört das Übertragungssignal von **A** bei $\mathtt{t_0}+80$ und das von **B** bei $\mathtt{t_0}+120$. Die Zeit für das Ereignis StartBackoff für die Station **C** ergibt sich aus
$min(\mathtt{t_0}+80, \mathtt{t_0}+120)+T_{jam}$ zu $\mathtt{t_0}+T_{jam}+80$.

Da **B** zuletzt mit dem Senden aufhört, ist die Zeit des Ereignisses StartBackoff dieser Station auch die neue Zeit für das Ereignis Deassert. Zum Schluß wird die Nummer der Station **C** sowie die aktuelle Simulationszeit in die Senderliste eingetragen.

Die Tabelle 5.2 faßt die wichtigsten Aktionen mit den dazugehörigen Zeiten noch einmal zusammen.
(E) steht für EndTransmit neu einplanen, *(S)* für StartBackoff neu einplanen und *(C)* für die Änderung der Zeit des Ereignisses StartBackoff. Das Ereignis Deassert wird zur Zeit $\mathtt{t_0}+150+T_{jam}$ für die Station **B** ausgelöst.

Zeit	Station **A**	Station **B**	Station **C**
$\mathtt{t_0}$	$(E)\ \mathtt{t_0}+10^6$		
$\mathtt{t_0}+20$	$(S)\ \mathtt{t_0}+T_{jam}+200$	$(S)\ \mathtt{t_0}+T_{jam}+180$	
$\mathtt{t_0}+50$	$(C)\ \mathtt{t_0}+T_{jam}+130$	$(C)\ \mathtt{t_0}+T_{jam}+150$	$(S)\ \mathtt{t_0}+T_{jam}+80$

Tabelle 5.2: Zusammenfassung des Szenario mit drei Stationen

Das Ereignis Deassert simuliert das Freiwerden des Kanals. Dabei werden alle auf dieses Ereignis wartenden Stationen benachrichtigt. Die Signallaufzeit wird auch hier berücksichtigt. Für eine genaue Beschreibung siehe Abschnitt 5.4.12 [*csma_deassert()*], Seite 56.

5.3.3 Das laufzeitoptimierte Modell

Nach der Implementierung der im vorhergehenden Abschnitt beschriebenen Methode und Durchführung einiger Simulationsläufe stellte sich heraus, daß eine weitere Verbesserung der Programmlaufzeit wünschenswert sei.

Die meiste Rechenzeit wurde für die Bestimmung der Ereigniszeiten im Zusammenhang mit Kollisionen und dem damit verbundenen Durchsuchen der Senderliste benötigt. Eine Vereinfachung sollte also an dieser Stelle ansetzen.

Den Ansatzpunkt für die Optimierung liefert die Beobachtung, daß an den meisten Kollisionen nur zwei Stationen beteiligt sind. Es sollte demnach ausreichen, **diesen** Fall genau zu simulieren und die seltenen Fälle von Kollisionen mit mehr als zwei beteiligten Stationen vereinfacht darzustellen.
Für eine große Anzahl Stationen trifft diese Annahme nicht mehr zu, da es zu sehr vielen Kollisionen, auch mit mehr als zwei beteiligten Stationen kommt. Zusätzlich hat die Größe der zu übertragenden `Frame`s einen Einfluß auf die Anzahl der an einer Kollision beteiligten Stationen. In Kapitel 5.5, Seite 60 wird jedoch gezeigt, daß die gewählte Vereinfachung trotzdem sehr gut die Realität darstellt und für die meisten modellierten Systeme völlig ausreichend ist.

Diese Überlegungen führen zu folgender Methode der optimierten Bestimmung der Ereigniszeiten:
Die Kollision von **zwei** Stationen in einem Kollisionsfenster wird wie vorher **genau** simuliert. Falls eine i-te (i>2) Station ebenfalls in diesem Kollisionsfenster zu senden beginnt, bleibt ein eventueller Einfluß auf den Zeitpunkt der Ereignisse StartBackoff der j-ten (j<i) Station und Deassert unberücksichtigt. Alle Ereignis-Zeiten werden also relativ zu der Zeit berechnet, bei der die **erste** Station in diesem Kollisionsfenster zu senden begonnen hat. Das Führen einer

Liste aller aktuell sendenden Stationen und das ständige Durchsuchen derselben entfallen damit vollständig.

Die Funktionsweise der neuen Methode wird wieder anhand des Szenarios aus Abb. 5.5, Seite 42 erläutert.

Für die Beschreibung des Kanalzustandes wird wieder *channel* benötigt. Auch *first_collision* ist erforderlich, um feststellen zu können, ob es sich um die erste oder eine spätere Kollision in dem aktuellen Kollisionsfenster handelt. Die Liste wird durch die Struktur *first_sender* ersetzt, in der der Name und der Sendezeitpunkt des **ersten** Senders vermerkt wird.

Station **A** beginnt als erste bei $\mathtt{t_0}$ zu senden. Ihr Name und die aktuelle Simulationszeit werden in *first_sender* gemerkt. Das Ereignis EndTransmit wird für $\mathtt{t_0} + 10^6$ eingeplant und *channel* sowie *first_collision* gesetzt.

Bei $\mathtt{t_0} + 20$ beginnt **B** zu senden und löst damit eine Kollision aus. Da es sich um die erste in diesem Kollisionsfenster handelt, wird das Ereignis EndTransmit von **A** gelöscht und durch ein StartBackoff-Ereignis ersetzt. Die Zeit für dieses Ereignis ergibt sich zu

$$\mathtt{t_0} + 20 + L_{BA} + T_{jam} = \mathtt{t_0} + 200 + T_{jam}.$$

Der Zeitpunkt für StartBackoff der Station **B** ergibt sich wie bei der genauen Methode zu $\mathtt{t_0} + 180 + T_{jam}$. Zum Schluß wird noch das Ereignis Deassert zur Zeit $\mathtt{t_0} + 200 + T_{jam}$ eingeplant und vermerkt, daß es von **A** ausgelöst wird.

Bisher lieferte das optimierte Verfahren die gleichen Ereigniszeiten wie das genaue. Die Änderungen werden deutlich, wenn **C** bei $\mathtt{t_0} + 50$ zu senden beginnt.
Der Zeitpunkt für das Ereignis StartBackoff der Stationen **A** und **B** wird **nicht** verändert und auch das Ereignis Deassert bleibt gleich. Die Zeit für das Ereignis StartBackoff der Station **C** wird relativ zu **A** berechnet, da diese die erste sendende Station war. Damit ergibt sich für StartBackoff von **C** die Zeit: $\mathtt{t_0} + 80 + T_{jam}$.

In der Tabelle 5.3 sind die sich ergebenden Zeiten für die genaue und die optimierte Methode noch einmal vergleichend aufgeführt. *SB* steht für den Zeipunkt des Ereignisses StartBackoff. Die unter *Fs* angegebene Zeit **plus** T_{ifg} ist der frühste Zeitpunkt, zu dem ein neuer Sendeversuch unternommen werden kann. In diesem Fall ist die `Backoff`-Wartezeit gleich Null.

Station	*SB* (genau)	*SB* (opt)	*Fs* (genau)	*Fs* (opt)
A	$\mathtt{t_0} + T_{jam} + 130$	$\mathtt{t_0} + T_{jam} + 200$	$\mathtt{t_0} + T_{jam} + 330$	$\mathtt{t_0} + T_{jam} + 360$
B	$\mathtt{t_0} + T_{jam} + 150$	$\mathtt{t_0} + T_{jam} + 180$	$\mathtt{t_0} + T_{jam} + 310$	$\mathtt{t_0} + T_{jam} + 380$
C	$\mathtt{t_0} + T_{jam} + 80$	$\mathtt{t_0} + T_{jam} + 80$	$\mathtt{t_0} + T_{jam} + 250$	$\mathtt{t_0} + T_{jam} + 280$

Tabelle 5.3: Vergleich der Ereigniszeiten der genauen und der optimierten Methode

Die Werte in der Spalte *Fs* (opt) ergeben sich folgendermaßen:
Für die Stationen **B** und **C** wird die Zeit über das Ende des `JAM`-Signal der Station **A** und die entsprechenden Laufzeiten L_{AB} bzw. L_{AC} bestimmt.
Der dazugehörige Zeitpunkt für **A** wird über das Ende des `JAM` der Station **B** berechnet. Bei $\mathtt{t_0} + \mathtt{L_{AB}}$ erkennt **B** die Kollision und beendet das `JAM` T_{jam} `ITU` später. Das Ende dieses Signals erreicht **A** nach zusätzlichen $\mathtt{L_{BA}}$ `ITU`. Damit ergibt sich der Zeitpunkt des Ereignisses StartBackoff der Station **A** zu: $\mathtt{t_0} + L_{AB} + T_{jam} + L_{BA}$.

Die endgültige Implementierung dieser Methode wird im nächsten Abschnitt beschrieben.
Die Simulationsergebnisse der beiden Modellvarianten werden in Abschnitt 5.5.6, Seite 76 verglichen.

5.4 Realisierung

In den folgenden Funktionsbeschreibungen wird das in Kapitel 4 vermittelte Grundlagenwissen über simuLan vorausgesetzt und nur noch die für die Modellierung spezifischen Sachverhalte beschrieben.

Bei Funktionen, die sich in der Implementierung des genauen Modells und des optimierten Modells unterscheiden, wird zunächst die genaue Version beschreiben. Im jeweils direkt darauffolgenden Abschnitt wird die Implementation für die optimierte Version angegeben. In der Überschrift sind diese Abschnitte durch den Anhang „*(optimiert)*" gekennzeichnet.
Die betroffenen Funktionen sind *csma_defer(), csma_deassert()*, *csma_start_transmit()* und *csma_start_backoff()*.

Ein `CSMA-FRAME` an eine Funktion „übergeben" bedeutet im folgenden, einen Zeiger auf die dazugehörige Datenstruktur `EVENT` als Parameter an diese Funktion zu übergeben (siehe Abschnitt 5.7 [*csma_data_encapsulation()*], Seite 48).

In Anhang D ist das vollständige Programmlisting enthalten.

5.4.1 Schnittstellen

Aus Abb. 5.6, Seite 47 sind die Schnittstellen zur simuLan-Schicht *2* sowie der Datenfluß innerhalb des CSMA-Moduls ersichtlich. Die Schreibweise **(*...)()** besagt, daß die entsprechende Schnittstellenfunktion der Nachbarschicht über einen Funktionszeiger aufgerufen wird.

Die Schicht $N \geq 2$ im Sender erzeugt ein `Paket` und übergibt dieses zum Senden an die Funktion *csma_data_encapsulation()* (über *(* net_params.l1_sap)()*). Diese verpackt das `Paket` in einem `CSMA-FRAME` (siehe Abschnitt 5.7 [*csma_data_encapsulation()*], Seite 48) und plant dann für diesen ein Ereignis DEFER ein.

Ein `Paket` wird beschrieben durch einen `EVENT` und einer daran angehängten `SADU` (siehe Abschnitt 4.2.2 [*EVENT*], Seite 15). Für den Informationsaustausch zwischen höheren Schichten ($N \geq 2$) können an dem Zeiger `SADU`→*next* Datenstrukturen angehängt sein. Diese dürfen durch diese Schicht *1* nicht verändert werden (siehe Abschnitt 4.2.2 [*Wichtig*], Seite 15).

Wenn die Funktion *csma_defer()*, ausgelöst durch ein Ereignis DEFER, feststellt, daß der Kanal belegt ist, wird der `CSMA-FRAME` in die Warteliste *defer_wait_list* eingefügt. Andernfalls wird für diesen ein Ereignis StartTransmit eingeplant.
In der Funktion *csma_deassert()*, die durch *csma_end_transmit()* oder *csma_start_backoff()* aktiviert wird, werden alle `EVENT`s aus dieser Liste entfernt und für das Ereignis StartTransmit, das die Funktion *csma_start_transmit()* aktiviert, eingeplant.

Die Funktion *csma_start_transmit()* wird durch das Ereignis StartTransmit aktiviert und überprüft, ob schon eine andere Station sendet. Falls schon eine Übertragung läuft, wird für den `CSMA-FRAME` ein Ereignis StartBackoff eingeplant und sonst ein Ereignis EndTransmit.

In der Funktion *csma_start_backoff()* wird für den `CSMA-FRAME`, bei einer `Backoff`-Wartezeit von Null *Slottimes*, ein Ereignis StartTransmit und andernfalls ein Ereignis Defer eingeplant.

Nach der erfolgreichen Übertragung gibt die Funktion *csma_end_transmit()* das `Paket` aus dem `CSMA-FRAME` an die Schicht $N \geq 2$ im Empfänger und benachrichtigt anschließend die Schicht $N \geq 2$ im Sender.

Während der Übertragung auftretende Störungen auf dem Kanal, werden durch die Funktion *csma_transmit_error()* simuliert.

5.4.2 Modellparameter

In der Tabelle 5.4 sind die Parameter des Netzmodells `CSMA` sowie jeweils eine kurze Erläuterung dazu angegeben. Die aktuellen Werte für einen Simulationslauf werden über die Konfigurationsdatei (siehe Abschnitt 4.2.9 [*Konfigurationsdatei*], Seite 20) festgelegt. Diese Parameter sind, soweit nicht in der Struktur *net_params* enthalten, in der Struktur *csma_info* gespeichert.
Die Parameter und Einheiten gibt simuLan beim Aufruf mit der Option „**-x**" aus.

5.4.3 Variablen

Die Tabelle 5.5 gibt die lokalen Variablen des Netzmoduls `CSMA` an. Alle Zählvariablen sind in der Struktur *csma_info* enthalten.

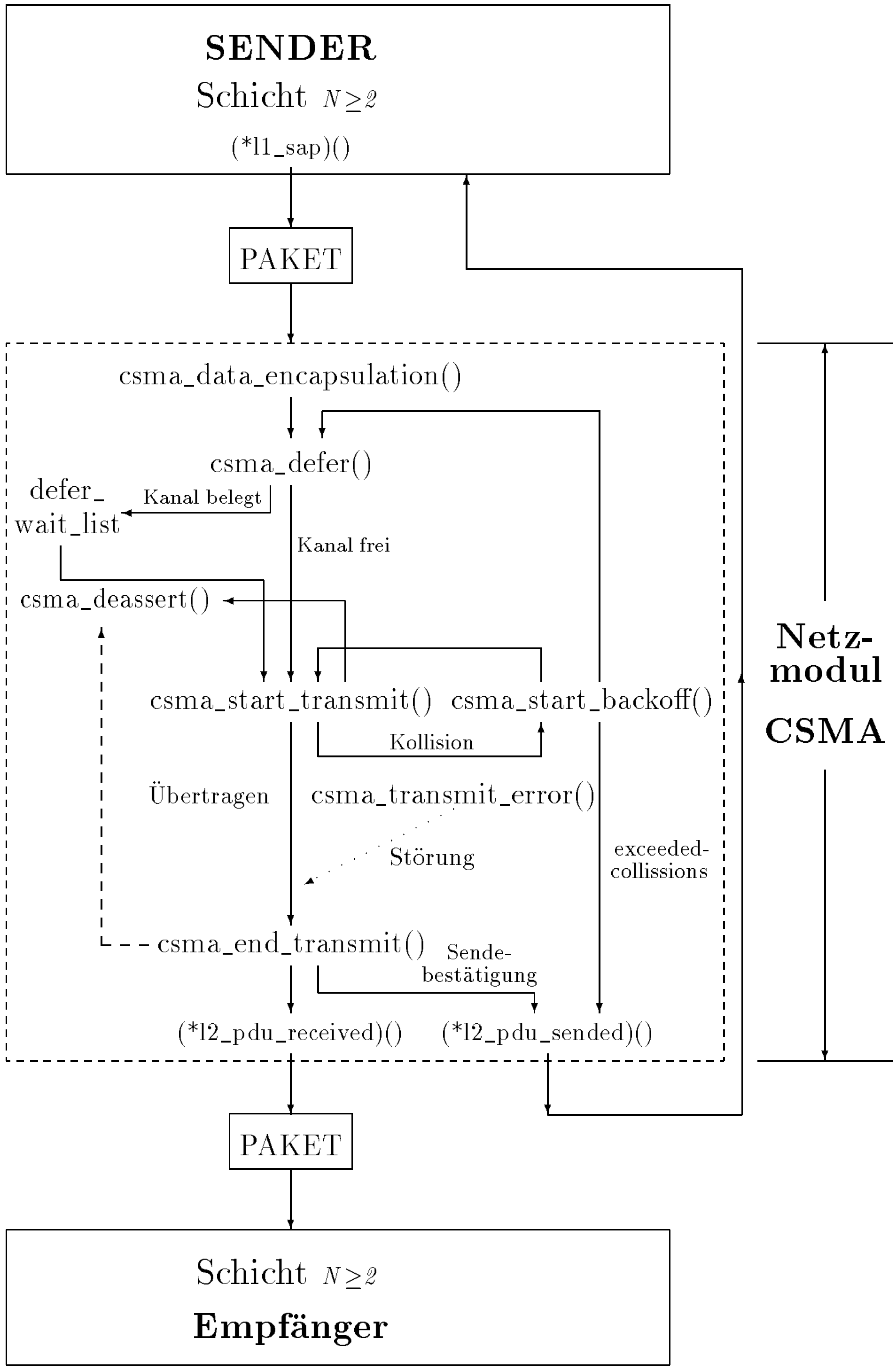


Abbildung 5.6: Schnittstellen und Datenfluß im CSMA-Netzmodul

5.4.4 csma_data_encapsulation()

Die Funktion *csma_data_encapsulation()* realisiert die Dienstzugangsfunktion des `CSMA`-Netzmoduls. Ein Zeiger auf diese wird während des Einlesens der Konfiguration in *net_params.l1_sap* eingetragen (siehe Abschnitt 4.2.6 [*net_params*], Seite 18). Sie bekommt einen Zeiger auf den `EVENT`, der das zu sendende `Paket` beschreibt, übergeben. Dieses `Paket` wird in einem `CSMA-FRAME`, der der im Netz verwendeten Dateneinheit entspricht, verpackt.
Ein `CSMA-FRAME` wird durch die in Abb. 5.7 Seite 48 dargestellte Datenstruktur beschrieben. Die `CSMA-FRAME`-Größe und damit auch die Anzahl der zu übertragenden Bits ergibt sich aus: *csma_info.overhead + max(*`SADU->`*pdu_size, net_params.min_data_len).*

Parameter	Erläuterung
bits_per_second	Nominelle Übertragungsgeschwindigkeit des Mediums in Bits pro Sekunde.
inter_frame_gap	Ruhezeit zwischen dem Ende einer Übertragung und dem Beginn der nächsten (Bit).
jam_time	Länge des `JAM`-Signals (Bit).
min_send_bits	Minimale Anzahl zu sendender Bits, bevor auf eine Kollision mit dem Senden des `JAM` reagiert wird.
min_data_len	Mindestgröße des Datenbereiches eines `CSMA-FRAME` (Bit).
max_data_len	Maximale Größe des Datenbereiches eines `CSMA-FRAME` (Bit).
overhead	Anzahl zusätzlicher Bits für Verwaltungsinformation, die jedem zu übertragenden `Paket` hinzugefügt wird.
slot_time	Größe des Kollisionsfensters (siehe Abschnitt 5.3.1 [*Kollision*], Seite 38).
attempt_limit	Maximale Anzahl von Wiederholungen für das Senden eines `Frame`s nach einer Kollision.
backoff_limit	Legt die maximale Wartezeit nach einer Kollision bei dem *binary truncated exponential backoff* Verfahren fest.
itu_pro_meter	Ausbreitungsgeschwindigkeit des Signals im Kabel.
calc_backoff	Funktion, die die Wartezeit nach einer Kollision berechnet (siehe Abschnitt 5.4.15 [*csma_backoff()*], Seite 56).
error_distr	beschreibt die Verteilung der Zeiten zwischen zwei Übertragungsfehlern (siehe Abschnitt 5.4.14 [*csma_transmit_error()*], Seite 56).
calc_laufzeit	Eine Funktion zur Berechnung der Signallaufzeit zwischen zwei Stationen (siehe Abschnitt 5.4.16 [*Laufzeitfunktion*], Seite 56).
end_transmit_probept	Meßpunkt in der Funktion *cmsa_end_transmit()* (siehe Abschnitt 5.4.18 [*Meßpunkte*], Seite 59).

Tabelle 5.4: Die Parameter des Netzmoduls *CSMA*

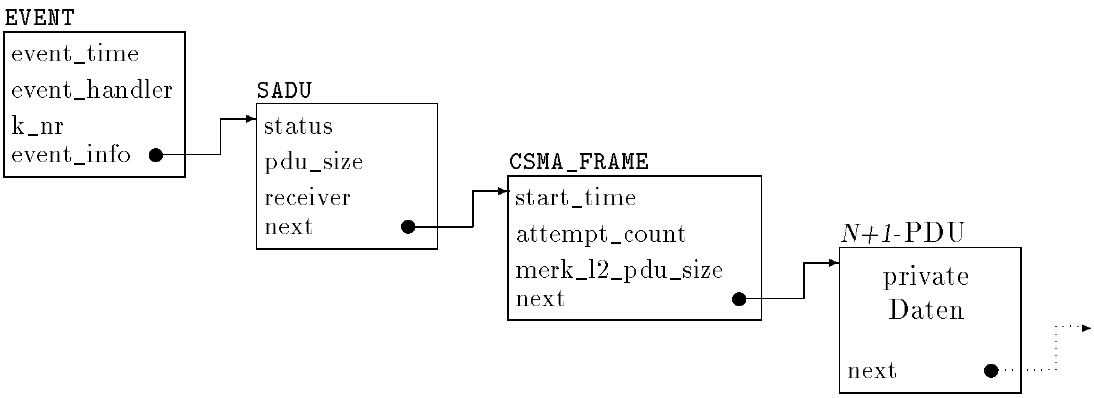


Abbildung 5.7: Datenstruktur die einen `CSAM-FRAME` beschreibt

Anschließend wird in dem `CSMA-FRAME` die Startzeit der Übertragung gesetzt und die Anzahl der bisherigen Kollisionen mit 0 initialisiert.

Das nächste Ereignis für diesen `CSMA-FRAME` ist Defer. Die Funktion *csma_defer()* ist die dazugehörige Ereignisfunktion. Ein Zeiger auf diese wird in `EVENT`→*event_handler* eingetragen. Die Zeit des Ereignisses steht in `EVENT`→*event_time* und muß von der aufrufenden Prozedur gesetzt werden. So ist es möglich, Verzögerungszeiten zwischen den Schichten einfach zu simulieren.

5.4.5 csma_defer()

Diese Funktion erhält als Parameter einen `CSMA-FRAME` übergeben und simuliert das Testen des Kanalzustands mit den entsprechenden Folgeaktionen.
Die Abb. 5.8 zeigt vereinfacht den Kontrollfluß.

Variablen	Bedeutung
channel_status	Zustand des Kanals. `FREI`: Es sendet **keine** Station `BELEGT`: Mindestens eine Station sendet
first_collision	Flag für die erste Kollision in einem Kollisionsfenster.
csma_stoerzeit	Zeitpunkt der letzten Übertragungsstörung. Diese Variable ist nur definiert, falls die Simulation von Störungen aktiviert ist.
defer_wait_list	Liste der Stationen, die auf das Freiwerden des Kanals warten.
started_trans	Liste der Stationen, die gerade senden; wird nur bei der genauen Methode verwendet.
first_sender	Diese Struktur enthält die Nummer der Station (*station_nr*), die als erste gesendet hat, und den Zeitpunkt (*zeit*) des Übertragungsstarts. Wird nur bei der optimierten Methode verwendet.
s1_sense_idle	Enthält die Zeit, zu der die erste sendende Station in einem Kollisionsfenster nach einer Kollision erkennt, daß der Kanal wieder frei ist.
last_sender	Diese Struktur enthält die Nummer der Station (*station_nr*), die zuletzt gesendet hat, und den Zeitpunkt (*zeit*) des Übertragungsendes.
simul_frames	Anzahl der bisher simulierten `CSMA-FRAME`s.
nutz_bits_transmitted	Anzahl Bits in den erfolgreich übertragenen `CSMA-FRAME`s einschließlich des Overhead und eventueller Padding-Bits.
collision_count	Diese Variable enthält die Anzahl bisher simulierter Kollisionen.
exceeded_collisions	Anzahl `CSMA-FRAME`s, die mit *attempt_limit* Sendeversuchen nicht gesendet werden konnten, da es immer zu einer Kollision kam.
dropped_packets	Diese Variable ist nur definiert, wenn die Simulation von Übertragungsfehlern aktiviert ist. In dem Fall enthält sie die Anzahl gestörter `CSMA-FRAME`s.
nodes_abstand[]	Array, in dem die Stationsabstände gespeichert sind.

Tabelle 5.5: Die Modulvariablen des Netzmoduls *CSMA*

$\mathtt{L_{FA}}$ sei die Signallaufzeit zwischen Station $\mathbf{S_A}$ und Station $\mathbf{S_{first}}$, und T_{first} sei die Zeit, zu der $\mathbf{S_{first}}$ mit dem Senden begonnen hat.

Als erstes wird geprüft, ob die sendwillige Station $\mathbf{S_A}$ den Kanal als frei oder belegt erkennt. Dazu wird die Liste *started_trans* durchgegangen und für jede enthaltene Station $\mathbf{S_L}$ geprüft, ob ihr Signal die aktuelle Station $\mathbf{S_A}$ schon erreicht hat. Dies ist dann gegeben, wenn $simul\_time \leq T_{first} + L_{LA}$ gilt. In diesem Fall wird der `CSMA-FRAME` in die *defer_wait_list* gestellt.

Falls die Station $\mathbf{S_A}$ den Kanal als frei erkennt, wird der Zeitpunkt T_{start} des Ereignisses StartTransmit berechnet. Die Zeit T_{idle}, zu der $\mathbf{S_X}$ erkennt, daß der Kanal wieder frei ist, ergibt sich aus der Summe der Zeit, zu der die letzte Station $\mathbf{S_{last}}$ ihre Übertragung beendet hat, und der Signallaufzeit zwischen $\mathbf{S_{last}}$ und $\mathbf{S_A}$. Mit den Informationen über den letzten Sender aus der Struktur *last_sender* und der Laufzeit $\mathtt{L_{LA}}$ zwischen $\mathbf{S_{last}}$ und $\mathbf{S_A}$ ist dies:

$$T_{idle} = time\_last\_idle.zeit + L_{LA}.$$

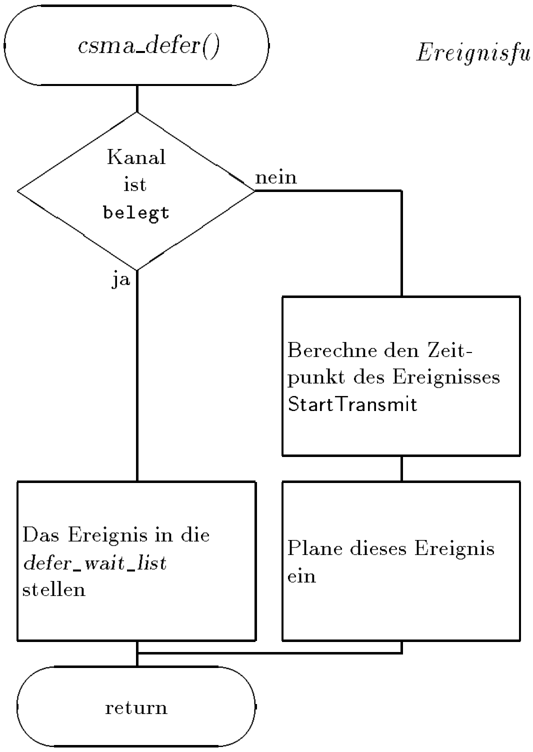
Nach einer Übertragung muß die Zeit T_{ifg} (Zeitdauer des `Inter Frame Gap`) abgewartet werden, bevor erneut gesendet werden darf. Damit ergibt sich der frühste Sendezeitpunkt zu $T_f = T_{idle} + T_{ifg}$.

Falls der Kanal schon länger als die Zeit $T_{idle} + T_{ifg}$ im Zustand `FREI` ist ($T_f < simul\_time$), ist $T_{start} = simul\_time$, andernfalls gilt $T_{start} = T_f$.

Nachdem der `EVENT`→*event_handler* auf *csma_start_transmit()* gesetzt wurde, wird der `CSMA-FRAME` für das Ereignis StartTransmit zur Zeit T_{start} eingeplant.

5.4.6 csma_defer() *(optimiert)*

Diese Funktion erhält als Parameter einen `CSMA-FRAME` übergeben und simuliert das Testen des Kanalzustands mit den entsprechenden Folgeaktionen.



Ereignisfunktion des Ereignisses Defer

Abbildung 5.8: Kontrollfluß in der Funktion *csma_defer()*

Die Abb. 5.8, Seite 50 zeigt den Kontrollfluß.

L_{FA} sei die Signallaufzeit zwischen der aktuellen Station $\mathbf{S_A}$ und Station $\mathbf{S_{first}}$, und T_{first} sei die Zeit, zu der $\mathbf{S_{first}}$ mit dem Senden begonnen hat.

Als erstes wird geprüft, ob die sendwillige Station $\mathbf{S_A}$ den Kanal als frei oder belegt erkennt. Falls der Kanal belegt ist, d.h. mindestens eine andere Station sendet schon, wird geprüft, ob das Signal der ersten sendenden Station $\mathbf{S_{first}}$ in diesem Kollisionsfenster die Station $\mathbf{S_A}$ schon erreicht hat. Dies ist dann gegeben, wenn $simul\_time \leq T_{first} + L_{FA}$ gilt. In diesem Fall wird der `EVENT` in die *defer_wait_list* gestellt.

Falls die Station $\mathbf{S_A}$ den Kanal als frei erkennt, wird der Zeitpunkt T_{start} des Ereignisses StartTransmit berechnet.
Die Zeit T_{idle}, zu der $\mathbf{S_A}$ erkennt, daß der Kanal wieder frei ist, ergibt sich aus der Summe der Zeit, zu der die letzte Station $\mathbf{S_{last}}$ ihre Übertragung beendet hat und der Signallaufzeit zwischen $\mathbf{S_{last}}$ und $\mathbf{S_A}$. Mit den Informationen über den letzten Sender aus der Struktur *last_sender* und der Laufzeit L_{LA} zwischen $\mathbf{S_{last}}$ und $\mathbf{S_A}$ ist dies:

$$T_{idle} = last\_sender.zeit + L_{LA}.$$

Nach einer Übertragung muß die Zeit T_{ifg} abgewartet werden, bevor erneut gesendet werden darf. Damit ergibt sich der frühste Sendezeitpunkt zu $T_f = T_{idle} + T_{ifg}$.

Falls der Kanal schon länger als die Zeit $T_{idle} + T_{ifg}$ im Zustand `FREI` ist ($T_f < simul\_time$), ist $T_{start} = simul\_time$, andernfalls gilt $T_{start} = T_f$.

Nachdem `EVENT`→*event_handler* auf *csma_start_transmit()* gesetzt wurde, wird der zu sendende `CSMA-FRAME` für das Ereignis StartTransmit zur Zeit T_{start} eingeplant.

5.4.7 csma_start_transmit()

Diese Funktion erhält als Parameter einen `CSMA-FRAME` übergeben und simuliert den Start einer Übertragung, falls der Kanal im Zustand `FREI` ist und sonst eine Kollision.

csma_start_transmit() — *Ereignisfunktion des Ereignisses* StartTransmit

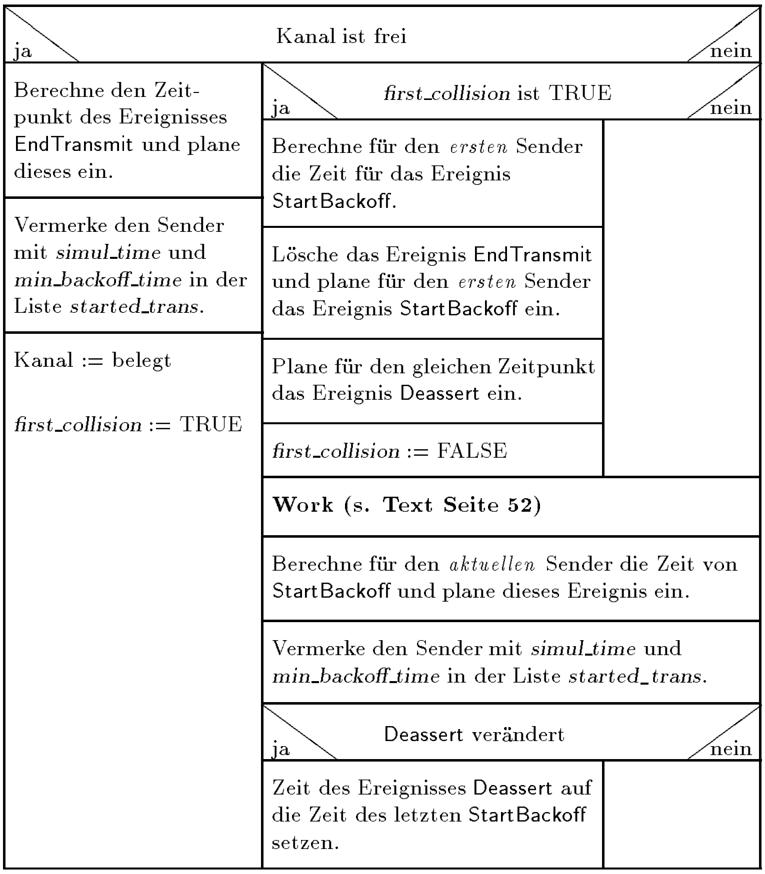


Abbildung 5.9: Kontrollfluß in der Funktion *csma_start_transmit()*

Station $\mathbf{S_A}$ ist diejenige, welche senden möchte. Station $\mathbf{S_1}$ ist der erste Sender in diesem Kollisionsfenster. $\mathtt{L_{AB}}$ beschreibt die Signallaufzeit zwischen den Stationen **A** und **B**.

Falls der Kanal frei ist, wird der Zeitpunkt für das Ereignis EndTransmit bestimmt. Dieser ergibt sich aus der Summe von *simul_time* und der Zeit, die für die Übertragung der Bits des **`CSMA-FRAME`**s benötigt wird. Nach Einplanen dieses Ereignisses wird *channel_status* auf **`BELEGT`** und *first_collision* auf **`TRUE`** gesetzt. Anschließend wird die aktuelle Station $\mathbf{S_A}$ und der Zeitpunkt, an dem die Station nach einer eventuellen Kollision frühstens in den `Backoff`-Zustand wechselt, in der Liste *started_trans* vermerkt. Diese Zeit wird für die Kollisionsbehandlung benötigt.
Bei der Bestimmung dieses Zeitpunktes ist der Parameter *min_send_bits* (siehe Abschnitt 5.5 [*Variablen*], Seite 49) zu berücksichtigen. Dieser gibt an, wieviele Bits mindestens gesendet werden, bevor der Kanal auf eine Kollision hin überprüft wird.
Damit ergibt sich der frühste Zeitpunkt, an dem eine Kollision erkannt werden kann, zu *simul_time*+ *min_send_bits* (siehe 5.3.1, Seite 38, für den Fall $t_2 \approx t_3$).
Da nach Erkennung der Kollision noch das **`JAM`**-Signal gesendet wird, ergibt sich:

$$min\_backoff\_time = simul\_time + min\_send\_bits + jam\_time.$$

Im Falle eines belegten Kanals wird zunächst geprüft, ob es sich um die erste Kollision in diesem Kollisionsfenster handelt.
Bei der ersten Kollision muß das Ereignis EndTransmit aus der Liste geplanter Ereignisse gelöscht und für die dazugehörige Station $\mathbf{S_1}$ der Zeitpunkt $T_{backoff\_1}$ des Ereignisses StartBackoff bestimmt werden. Die Berechnung wird in Abschnitt 5.3.2 [*Das laufzeitoptimierte Modell*], Seite 42, ausführlich beschrieben.

Diese Methode liefert:

$$T_{backoff\_1} = simul\_time + L_{A1} + jam\_time.$$

Da hier zusätzlich der Parameter *min_send_bits* berücksichtigt werden muß, ergibt sich für das Ereignis StartBackoff der Station $\mathbf{S_1}$ der Zeitpunkt aus:

$\max(\mathit{min\_backoff\_time}, T_{backoff\_1})$.

Für die gleiche Zeit wird ein Ereignis Deassert für $\mathbf{S_1}$ eingeplant, da diese die letzte sendende Station ist. Zum Schluß wird *first_collision* noch auf `FALSE` gesetzt.

Die im folgenden beschriebenen Aktionen sind für die erste und alle späteren Kollisionen im aktuellen Kollisionsfenster identisch. Im Flußdiagramm sind diese in der Aktion **Work** zusammengefaßt.

Die Berechnung wird in Abschnitt 5.3.2 [*Das genaue Simulationsmodell*], Seite 42, ausführlich beschrieben.

Für jede in der Liste *started_trans* eingetragene Station wird geprüft, ob diese durch das Senden der Station $\mathbf{S_A}$ die Kollision früher erkennt; und wenn dieses zutrifft, wird der Zeitpunkt des dazugehörigen Ereignisses StartBackoff entsprechend korrigiert.

Um den Zeitpunkt $T_{backoff\_A}$ von StartBackoff der Station $\mathbf{S_A}$ zu berechnen, wird die Station $\mathbf{S_K}$ mit der kleinsten Laufzeit $\mathtt{L_{KA}}$ in der Liste *started_trans* ermittelt. Für $T_{backoff\_K}$ gilt dann mit T_{start_K} als Sendestartzeit der Station $\mathbf{S_K}$:

$T_{backoff\_A} = \mathit{simul\_time} + \mathit{jam\_time} + \max(\mathit{min\_send\_bits}, T_{start\_K} + L_{KA})$.

Nachdem das Ereignis StartBackoff für $\mathbf{S_A}$ bei $T_{backoff\_A}$ eingeplant wurde, muß geprüft werden, ob der Kanal jetzt zu einem anderen Zeitpunkt als bisher freigegeben wird. In diesem Fall muß das Ereignis Deassert entsprechend geändert werden. Zum Abschluß wird für $\mathbf{S_A}$ der Wert von *min_backoff_time* bestimmt und zusammen mit der Nummer von $\mathbf{S_A}$ und der aktuellen Simulationszeit in der Liste *started_trans* abgespeichert.

5.4.8 csma_start_transmit() *(optimiert)*

Diese Funktion erhält als Parameter einen `CSMA-FRAME` übergeben und simuliert den Start einer Übertragung, falls der Kanal im Zustand `FREI` ist, und sonst eine Kollision.

Der Kontrollfluß ist Abb. 5.10 zu entnehmen.

Station $\mathbf{S_A}$ ist diejenige, welche senden möchte. Station $\mathbf{S_1}$ ist der erste Sender in diesem Kollisionsfenster.

Falls der Kanal frei ist, wird der Zeitpunkt für das Ereignis EndTransmit bestimmt. Dieser ergibt sich aus der Summe von *simul_time* und der Zeit, die für die Übertragung der Bits des `CSMA-FRAME`s benötigt wird. Nach Einplanen dieses Ereignisses wird die aktuelle Station $\mathbf{S_A}$ in der Struktur *first_sender* mit Nummer und der aktuellen Simulationszeit vermerkt. Zum Schluß wird *channel_status* auf `BELEGT` und *first_collision* auf `TRUE` gesetzt.

Im Falle eines belegten Kanals wird zunächst geprüft, ob es sich um die erste Kollision in diesem Kollisionsfenster handelt.

Bei der ersten Kollision muß das Ereignis EndTransmit aus der Liste geplanter Ereignisse gelöscht und für die dazugehörige Station $\mathbf{S_1}$ der Zeitpunkt $T_{backoff\_1}$ des Ereignisses StartBackoff bestimmt werden.

Die Berechnung dieser Zeit wird in Abschnitt 5.3.3 [opt-mod], Seite 44 ausführlich beschieben. Diese Methode liefert:

$T_{backoff\_1} = \mathit{simul\_time} + L_{A1} + \mathit{jam\_time}$.

Da hier zusätzlich der Parameter *min_send_bits* berücksichtigt werden muß, ergibt sich für das Ereignis StartBackoff der Station $\mathbf{S_1}$ der Zeitpunkt:

$\max((\mathit{simul\_time} + \mathit{min\_send\_bits} + \mathit{jam\_time}), T_{backoff\_1})$.

Für die gleiche Zeit wird ein Ereignis Deassert für $\mathbf{S_1}$ eingeplant, da diese die letzte sendende Station ist.

Anschließend wird der Zeitpunkt T_1 bestimmt, an dem $\mathbf{S_1}$ erkennt, daß der Kanal wieder frei ist. Die Berechnung wurde ausführlich in Abschnitt 5.3.3 beschrieben und ergibt:

$T_1 = T_{start} + L_{1A} + \mathit{jam\_time} + L_{A1}$.

Diese Zeit wird in *s1_sense_idle* für die Verwendung in *csma_start_backoff()* gespeichert. Zum Schluß wird *first_collision* noch auf `FALSE` gesetzt.

csma_start_trans_mit() — Ereignisfunktion des Ereignisses StartTransmit

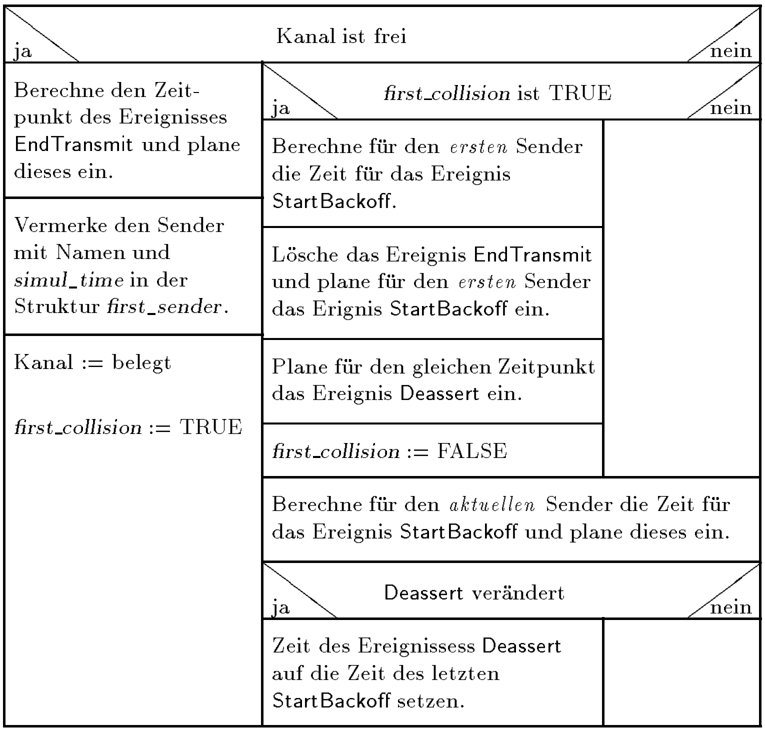


Abbildung 5.10: Kontrollfluß in der optimierten Funktion *csma_start_transmit()*

Die im folgenden beschriebenen Aktionen sind für die erste und alle späteren Kollisionen im aktuellen Kollisionsfenster identisch.
Für die Station $\mathbf{S_A}$ muß jetzt die Zeit $T_{backoff\_A}$ des Ereignisses StartBackoff bestimmt werden.
Die in Abschnitt 5.3.3 [*Das laufzeitoptimierte Modell*], Seite 44 beschriebene Berechnung liefert:
$T_{hilf} = T_{start} + L_{1A} + jam\_time.$

Unter Berücksichtigung des Parameters *min_send_bits* ergibt sich:
$T_{backoff\_A} = \max(T_{hilf}, min\_send\_bits).$

Abschließend wird das Ereignis StartBackoff für die Zeit $T_{backoff\_A}$ eingeplant.

5.4.9 csma_start_backoff()

Diese Funktion erhält als Parameter einen `CSMA-FRAME` übergeben und simuliert den Übergang in den `Backoff`-Zustand.

Die Abb. 5.11 zeigt den Kontrollfluß in der Funktion.

Zunächst wird geprüft, ob die Anzahl der Übertragungsversuche größer als die durch *attempt_limit* (siehe Abschnitt 5.5 [*Variablen*], Seite 49) vorgegebene maximal erlaubte Anzahl ist. Trifft dieses zu, wird durch Ausketten des `CSMA-FRAME` die schichtspezifische Information aus der übertragenen Dateneinheit entfernt und die alte `Paket`-Größe wieder eingetragen. Nach Setzen des `Paket`-Status auf `SEND_FAILED` wird das `Paket` über *(∗nodearry[*`EVENT`*→k_nr].l2_pdu_sended)()* an den Sender zurückgegeben.

Falls die maximale Anzahl noch nicht erreicht ist, wird über die Funktion *(∗calc_backoff)()* die `Backoff`-Wartezeit T_{wait} bestimmt.
Wenn T_{wait} größer als Null ist, wird der `CSMA-FRAME` für das Ereignis Defer zum Zeitpunkt $simul\_time + T_{wait}$ eingeplant.
Andernfalls wird der `CSMA-FRAME` für das Ereignis StartTransmit eingeplant und die Anzahl der

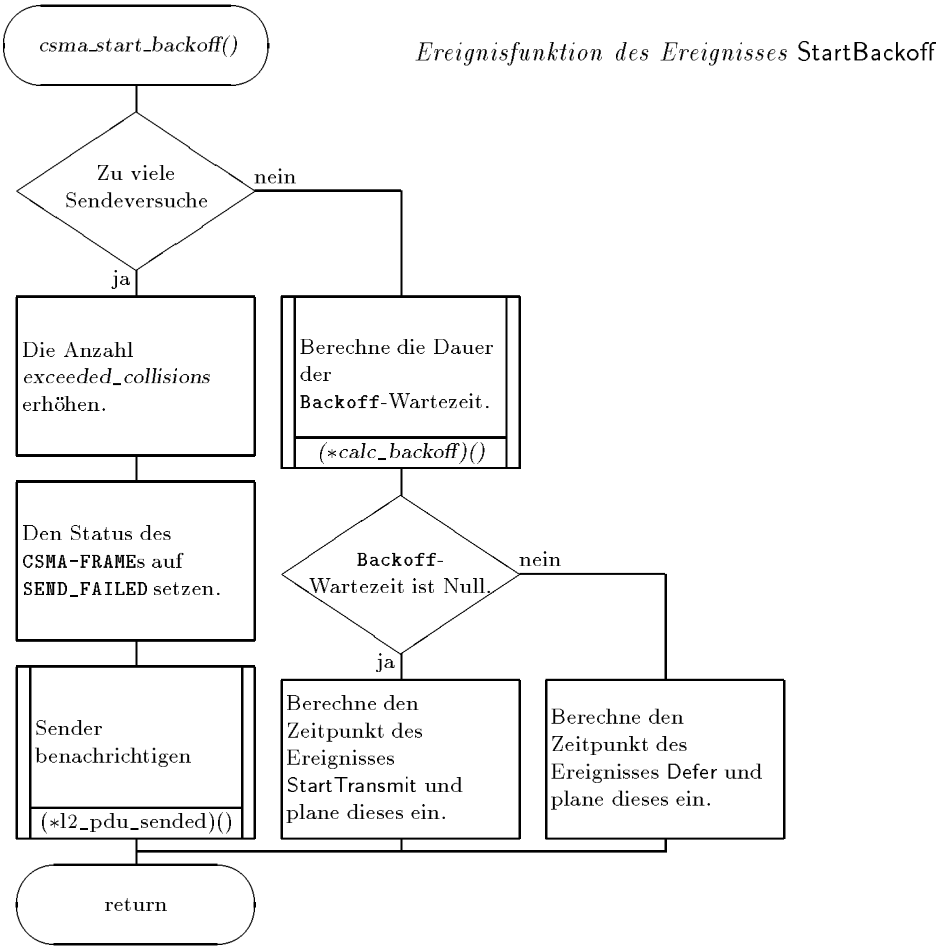


Abbildung 5.11: Kontrollfluß in der Funktion *csma_start_backoff()*

Übertragungsversuche erhöht.
Das Ereignis Deassert enthält die Station $\mathbf{S_D}$, die als letzte in diesem Kollisionsfenster sendet, sowie die Zeit $T_{Deassert}$, zu der der Kanal von $\mathbf{S_D}$ freigegeben wird. Die aktuelle Station $\mathbf{S_A}$ erkennt das Freiwerden des Kanals somit bei $T_{Deassert} + L_{DA}$. Da anschließend noch die Zeit *inter_frame_gap* gewartet werden muß, ergibt sich die Zeit für StartTransmit zu $T_{Deassert} + L_{DA} + inter\_frame\_gap$.

5.4.10 csma_start_backoff() *(optimiert)*

Diese Funktion erhält als Parameter einen `CSMA-FRAME` übergeben und simuliert den Übergang in den `Backoff`-Zustand.

Die Abb. 5.11, Seite 54 zeigt den Kontrollfluß in der Funktion.

Zunächst wird geprüft, ob die Anzahl der Übertragungsversuche größer als die durch *attempt_limit* (siehe Abschnitt 5.5 [*Variablen*], Seite 49) vorgegebene maximale Anzahl ist. Trifft dieses zu, wird durch Ausketten des `CSMA-FRAME` die `CSMA`-spezifische Information aus der übertragenen Dateneinheit entfernt und die alte `Paket`-Größe wieder eingetragen. Nach Setzen des `Paket`-Status (`SADU`→*status*) auf `SEND_FAILED` wird das `Paket` über *(*nodearry[*`EVENT`→*k_nr].l2_pdu_sended)()* an den Sender zurückgegeben.

Falls die maximale Anzahl Versuche noch nicht erreicht ist, wird über die Funktion *(*csma_info.calc_backoff)()* die `Backoff`-Wartezeit T_{wait} bestimmt.
Wenn T_{wait} größer als Null ist, wird der `CSMA-FRAME` für das Ereignis Defer zum Zeitpunkt $simul\_time + T_{wait}$ eingeplant.
Andernfalls wird der `CSMA-FRAME` für das Ereignis StartTransmit eingeplant und die Anzahl der

Übertragungsversuche erhöht.
Das Ereignis Deassert enthält die Station $\mathbf{S_D}$, die als letzte in diesem Kollisionsfenster sendet, sowie die Zeit $T_{Deassert}$, zu der der Kanal von $\mathbf{S_D}$ freigegeben wird. Falls die aktuelle Station $\mathbf{S_A}$ mit $\mathbf{S_D}$ identisch ist, wird das Ereignis StartBackoff für die in *csma_start_transmit()* berechnete Zeit (in *s1_sense_idle*) eingeplant (siehe Abschnitt 5.3.3 [*Das laufzeitoptimierte Modell*], Seite 44).
Andernfalls erkennt die aktuelle Station $\mathbf{S_A}$ das Freiwerden des Kanals bei $T_{Deassert} + L_{DA}$. Da anschließend noch die Zeit *inter_frame_gap* abgewartet werden muß, ergibt sich die Zeit für StartTransmit zu:

$T_{Deassert} + L_{DA} + inter\_frame\_gap.$

5.4.11 csma_end_transmit()

Diese Funktion erhält als Parameter einen `CSMA-FRAME` übergeben. Sie simuliert das Ende einer erfolgreichen Übertragung und die Ankunft eines `Frame`s in der `MAC`-Schicht (siehe Abschnitt 3.5 [*Normen*], Seite 9) des Empfängers.

Die Abb. 5.12 zeigt den Kontrollfluß in der Funktion.

csma_end_transmit() — Ereignisfunktion des Ereignisses EndTransmit

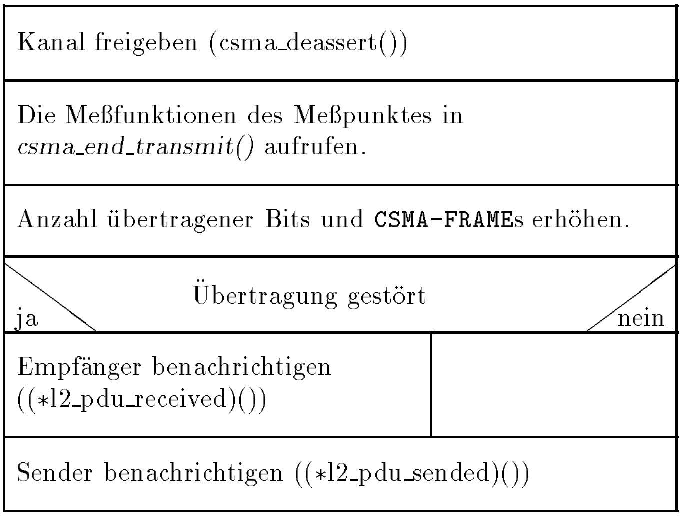


Abbildung 5.12: Kontrollfluß in der Funktion *csma_end_transmit()*

Nach Aufruf der Funktion *csma_deassert()*, die die Kanalfreigabe simuliert, werden alle aktivierten Meßfunktionen des Meßpunktes *csma_probept_end_transmit* mit dem Makro `MEASURE()` (siehe Abschnitt 4.2.7 [*Meßpunkte*], Seite 19) aufgerufen.
Anschließend wird durch Ausketten des `CSMA-FRAME` alle schichtspezifische Information aus der übertragenen Dateneinheit entfernt und die alte `Paket`-Größe in der `SADU` wieder eingetragen.
Nun wird geprüft, ob die Übertragung gestört war. Dieses trifft zu, falls *csma_stoerzeit* (siehe Abschnitt 5.5 [*Variablen*], Seite 49) größer als die Startzeit der Übertragung ist. Bei ungestörter Übertragung wird das `Paket` an die Empfängerschicht $N \geq 2$ übergeben. Die dazugehörige Funktion wird über *(*nodearray[*`SADU`*→receiver].l2_pdu_received)()* ermittelt. Der das `Paket` beschreibende `EVENT` mit den angehängten Informationen darf von dieser Prozedur nicht verändert werden (siehe Abschnitt 4.2.2 [*WICHTIG*], Seite 15).
Im Falle einer Übertragungsstörung wird diese Funktion **nicht** aufgerufen, da auch in der Realität fehlerhaft empfangene `Paket`e nicht an die höheren Schichten weitergereicht werden. Das `Paket` erhält allerdings den Status `ERROR_IN_TRANSMIT` , der vom Sender ausgewertet werden kann.
Durch den Aufruf von *(*nodearry[*`EVENT`*→k_nr].l2_pdu_sended)()*, mit dem `Paket` als Parame-

ter, wird zum Schluß ncoch der Sender des `Pakets` von dem Ende der Übertragung benachrichtigt.

5.4.12 csma_deassert()

Diese Funktion simuliert das Freigeben des Kanals. Sie wird entweder direkt in der Funktion *csma_end_transmit()* aufgerufen oder durch ein in *csma_start_transmit()* eingeplantes Ereignis Deassert aktiviert.

Zunächst werden alle Stationen aus der Liste *started_trans* gelöscht. Danach werden die Elemente der Struktur *last_sender* (siehe Abschnitt 5.5 [*Variablen*], Seite 49) entsprechend *simul_time* und der Station $\mathbf{S_D}$, die das Ereignis Deassert ausgelöst hat, gesetzt.

Anschließend werden alle `EVENT`s aus der *defer_wait_list* entfernt und für das Ereignis StartTransmit zum Zeitpunkt T_{start_X} eingeplant. Da nach dem Ende einer Übertragung eine Pause der Länge *inter_frame_gap* einzuhalten ist, ergibt sich für T_{start_X} der Station $\mathbf{S_X}$: $T_{start\_X} = simul\_time + L_{DX} + inter\_frame\_gap$.

5.4.13 csma_deassert() *(optimiert)*

Diese optimierte Funktion unterscheidet sich von der genauen Version nur darin, daß das Löschen der Liste entfällt.

5.4.14 csma_transmit_error()

Die Funktion *csma_transmit_error()* erzeugt nach einer in der Konfiguration ausgewählten Verteilungsfunktion Störungen auf dem Übertragungskanal. Der Zeitpunkt der Störung wird in *csma_stoerzeit* (siehe Abschnitt 5.5 [*Variablen*], Seite 49) vermerkt und in der Funktion *csma_end_transmit()* ausgewertet. Anschließend wird der Zeitpunkt der nächsten Störung ermittelt und für diesen ein Aufruf von *csma_transmit_error()* eingeplant.

In dem simulierten System reagiert der Sender nicht auf die Störung, sondern sendet trotzdem den ganzen `CSMA-FRAME`. Der Empfänger erkennt den Übertragungsfehler und wirft den `CSMA-FRAME` weg.
Dieser Sachverhalt wird simuliert, indem die simuLan-Schicht $N \geq 2$ im Empfänger nach einer Übertragung **nicht** benachrichtigt wird, *(*l2_pdu_received)()* wird nicht aufgerufen.

5.4.15 csma_backoff()

Diese Funktion realisiert das in Abschnitt 5.2, Seite 36, beschriebene Verfahren *truncated binary exponential backoff* zur Bestimmung der Wartezeit nach einer Kollision.
Die Anzahl der bisherigen Übertragungsversuche wird als Parameter übergeben und die Wartezeit (in `ITU`) (siehe Abschnitt 4.2.1 [*ITU*], Seite 15) als Ergebnis an den Aufrufer zurückgeliefert.

5.4.16 Die *calc_laufzeit*-Funktionen des *CSMA*-Moduls

Die Topologie des `CSMA`-Netzwerks wird über die Signallaufzeit zwischen zwei Stationen $\mathbf{S_A}$ und $\mathbf{S_B}$ beschrieben, die durch den Abstand und die Ausbreitungsgeschwindigkeit des Signals im Medium gegeben ist.

In Abb. 5.13 sind drei mögliche Topologien für ein solches Netz angegeben. Die am weitesten verbreitete ist die mit „**1)**" bezeichnete. Diese ist auch die einzige, die sich mit dem im folgenden beschriebenen Ansatz **genau** modellieren läßt (siehe Abschnitt 5.4.16.2 [*csma_linear()*], Seite 57).
Für die **genaue** Modellierung von „**2)**" und „**3)**" müßten neue Funktionen zum Berechnen der Laufzeiten geschrieben werden, die zur Programmlaufzeit eingesetzt werden.

Ein Zeiger auf die aktive Laufzeitfunktion wird in *csma_info.calc_laufzeit* abgespeichert.

Bisher sind folgende Funktionen implementiert:

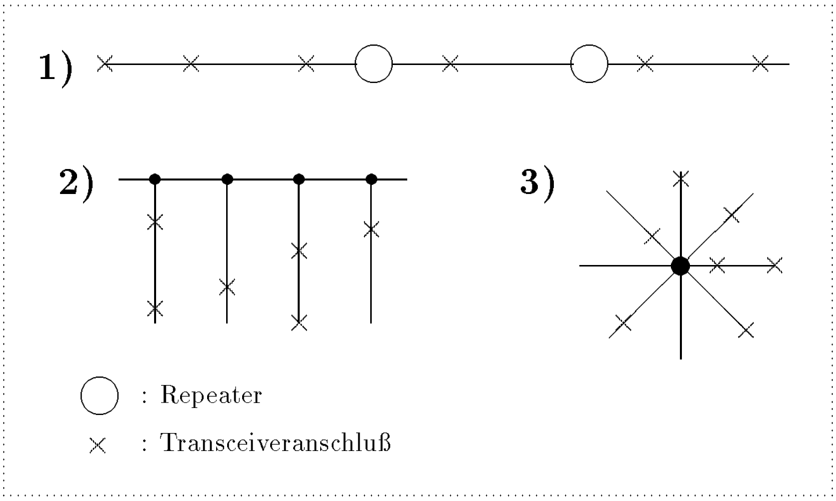


Abbildung 5.13: Topologien für ein CSMA/CD-Netzwerk

csma_konstant_laufzeit(): Diese Funktion liefert für zwei beliebige Stationen im Netz immer die gleiche, über die Konfiguration vorgegebene, konstante Signallaufzeit zurück (s. Abschnitt 5.4.16.1).

csma_genau_laufzeit(): Diese Funktion berechnet die Signallaufzeit aus dem konfigurierten Abstand der Stationen und der Ausbreitungsgeschwindigkeit des Signals im Übertragungsmedium.

5.4.16.1 Die Laufzeitfunktion *konstant*

Die dazugehörigen Funktionen sind *csma_konstant_laufzeit()*, *csma_rc_konstant()* und *csma_init_konstant()*.

Falls über die Konfiguration eine konstante Laufzeit gewählt wurde, wird mit *csma_rc_konstant()* die gewünschte Signallaufzeit eingelesen und der Zeiger *csma_info.calc_laufzeit* auf die Funktion *csma_init_konstant()* gesetzt.
Bei der Initialisierung des `CSMA`-Moduls wird *csma_init_konstant()* über *csma_info.calc_laufzeit* aufgerufen. Diese Funktion trägt in *csma_info.calc_laufzeit* dann einen Zeiger auf *csma_konst_laufzeit()* ein.
Diese Funktion liefert, unabhängig von den übergebenen Parametern, immer den vorgegebenen Wert der Signallaufzeit zurück.

5.4.16.2 Die Laufzeitfunktionen *csma_linear()* und *csma_gleichvert()*

In der Laufzeitfunktion *csma_linear()* soll die Topologie, über die Signallaufzeit, so genau wie möglich dargestellt werden. Dabei ergibt sich das Problem, wie der jeweilige Abstand zwischen zwei beliebigen Stationen im Netz gespeichert werden kann. Für N Stationen sind N^2 Abstandswerte zu speichern. Beim `Ethernet` sind zum Beispiel bis zu 1024 Stationen erlaubt. Eine direkte Speicherung aller Abstandswerte ist deshalb aus Speicherplatzgründen ungünstig.

Für die Topologie Abb. nach 5.14 gibt es jedoch eine einfache Möglichkeit, mit nur einem Abstandswert je Station die Signallaufzeiten zwischen zwei beliebigen Stationen im Netz zu berechnen.
Für dieses Verfahren wird ein Ende des linearen Busses als *Referenzpunkt* $\mathbf{P_R}$ gewählt. Der zu speichernde Abstandswert $\mathbf{D_n}$ für die Station $\mathbf{S_n}$ ist dann der Abstand zwischen $\mathbf{P_R}$ und $\mathbf{P_n}$ (s. Abb. 5.14).
Damit ergibt sich z.B. für $\mathbf{D_2}$ der Wert $\mathbf{a+b}$. Mit diesem Verfahren läßt sich auch ein Repeater (siehe Abschnitt 5.2 [*Ethernet*], Seite 35) als einfaches Verzögerungsglied darstellen. Die durch den Repeater erzeugte Signalverzögerung wird durch ‚*e*' dargestellt. Damit ergibt sich für den Abstand $\mathbf{D_4}$ der Station $\mathbf{S_4}$:

$$\mathbf{D_4 = a + b + c + d + e + f} \ .$$

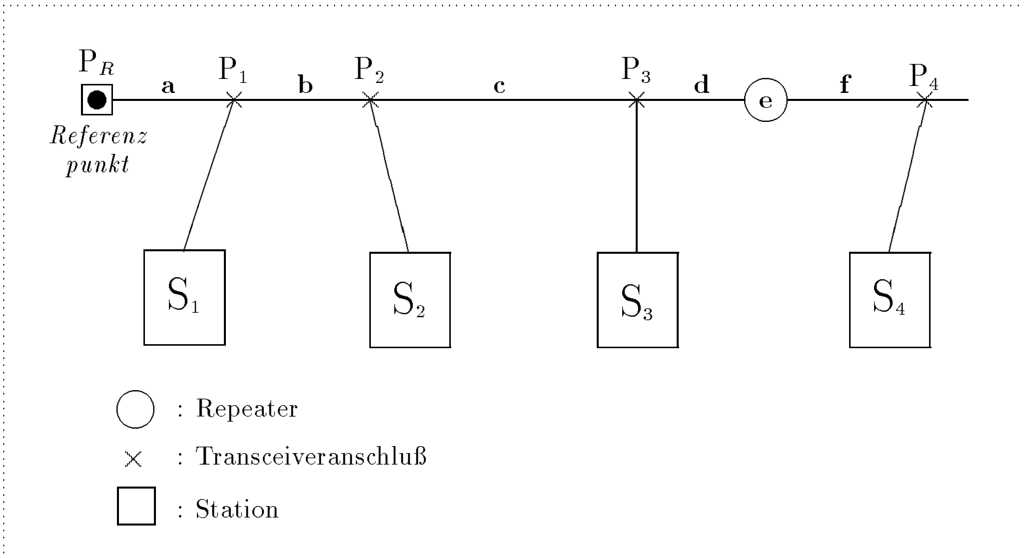


Abbildung 5.14: Die lineare Bustopologie mit Repeater

Der Abstand $\mathbf{D_{AB}}$ zwischen zwei Stationen $\mathbf{S_A}$ und $\mathbf{S_B}$ ergibt sich dann als:
$\mathbf{D_{AB}} = \|\mathbf{D_A} - \mathbf{D_B}\|$.

In dem Array *csma_nodes_abstand[]* ist für jede Station $\mathbf{S_n}$ (bei $\mathbf{P_n}$ angeschlossen) der Abstand zwischen $\mathbf{S_n}$ und dem Referenzpunkt $\mathbf{P_R}$ als Signallaufzeit (in `ITU`) zwischen $\mathbf{P_n}$ und $\mathbf{P_R}$ vermerkt.

Zu der Laufzeitfunktion *csma_linear()* gehören die Funktionen *csma_rc_linear()* und *csma_-init_linear()* .
Bei dieser Funktion wird in der Konfigurationsdatei für jede Station im Netz der Abstand zur vorhergehenden Station angegeben und von der Funktion *csma_rc_linear()* eingelesen. Ein Beispiel für eine solche Konfiguration ist in Abschnitt 5.5.4 angegeben.
Die Einlesefunktion speichert zunächst die Knotennamen mit den entsprechenden Abstandswerten in einer verketteten Liste und trägt dann einen Zeiger auf die Funktion *csma_init_linear()* in *csma_info.calc_laufzeit* ein. Nachdem die komplette Konfiguration eingelesen wurde, wird in *csma_init()* (siehe Abschnitt 5.4.17 [*csma_init()*], Seite 59) über *csma_info.calc_laufzeit* diese Funktion aufgerufen. Mit der Information aus der Liste wird nun das Array *csma_nodes_abstand[]* so initialisiert, daß für Station ‚**i**' in *csma_nodes_abstand[***i***]* die Signallaufzeit (in `ITU`) zwischen $\mathbf{S_i}$ und dem Referenzpunkt steht.
Zum Schluß trägt *csma_init()* noch einen Zeiger auf *csma_genau_laufzeit()* in *csma_info.calc_-laufzeit* ein. Diese Funktion erhält als Parameter die Nummern zweier Stationen und liefert die dazugehörige Signallaufzeit zurück.

Zu der Laufzeitfunktion *csma_gleichvert()* gehören die Funktionen *csma_rc_gleichvert()* und *csma_init_gleichvert()*. Bei dieser Funktion wird angenommen, daß alle Stationen gleichmäßig über das ganze Segment verteilt sind. Die Segmentlänge wird aus der Konfiguration durch die Funktion *csma_rc_gleichvert()* eingelesen. Anschließend wird in *csma_info.calc_laufzeit* ein Zeiger auf *csma_init_gleichvert()* eingetragen.
Diese Funktion wird in *csma_init()* aufgerufen und initialisiert *csma_nodes_abstand[]* entsprechend der gewünschten Gleichverteilung. Dabei wird für Station $\mathbf{S_1}$ der Wert Null und für die letzte Station die sich aus der Segmentlänge ergebende maximale Signalverzögerung eingetragen.
Zum Schluß trägt die Funktion *csma_init()* einen Zeiger auf *csma_genau_laufzeit()* in *csma_info.calc_laufzeit* ein.

5.4.17 Die Rahmenfunktionen des *CSMA*-Moduls

Die Rahmenfunktionen des Netzmoduls `CSMA` sind *csma_rc()*, *csma_init()*, *csma_cleanup()* und *csma_print_frame()*.

Die Aufgaben der Funktion *csma_rc()* sind:

- Falls über die Kommandozeile angegeben, alle variablen Parameter des CSMA-Moduls (siehe Abschnitt 5.5 [*Variablen*], Seite 49) mit einer Erklärung ausgeben.
- Die Datenstruktur *net_params* (siehe Abschnitt 4.2.6 [net-params], Seite 18) mit Defaultwerten vorbelegen.
- Die Funktion *rcreadkeywds()* (s. [Pet91]) mit der Tabelle der definierten Schlüsselwörter für das `CSMA`-Modul aufrufen. Dabei werden gegebenenfalls die Defaultwerte ausgegeben.
- Die eingelesenen Parameter auf Plausibilität überprüfen (siehe Abschnitt 5.4 [*Parameter*], Seite 48).

Die Funktion *csma_init()* hat folgende Aufgaben:

- Die Liste der freien Datenstrukturen vom Typ `CSMA-FRAME` initialisieren (s. auch Kapitel [Speicherverwaltung] in [Pet91]).
- Die lokalen Variablen (siehe Abschnitt 5.5 [*Variablen*], Seite 49) initialisieren.
- Falls die Simulation von Störungen auf dem Übertragungsweg gewünscht ist, das erste Störungsereignis in die Ereignisliste eingefügen.
- Die gewünschte Laufzeitfunktion über die Funktion, auf die *csma_info.calc_laufzeit* zeigt, initialisieren.

Die Funktion *csma_cleanup()* gibt die Anzahl der übertragenen Bits und `CSMA-FRAME`s sowie die Anzahl der Kollisionen und Exceeded-Kollisionen aus. Falls die Simulation von Übertragungsstörungen aktiviert ist, wird zusätzlich die Anzahl durch Übertragungsfehler verlorengegengener `CSMA-FRAME`s ausgegeben.

Die Funktion *csma_print_frame()* ermöglicht die Ausgabe des Inhalts eines `CSMA-FRAME` und der angehängten Datenstrukturen. Diese Funktion ist für die Fehlersuche notwendig.

5.4.18 Der Meßpunkt *csma_probept_end_transmit*

Der Meßpunkt *csma_probept_end_transmit* ist in der Funktion *csma_end_transmit()* eingebaut. Alle an diesem Meßpunkt aktivierten Meßfunktionen werden jeweils nach erfolgreicher Übertragung eines `CSMA-FRAME`s aufgerufen.
Die bisher implementierten Meßfunktionen sind *csma_utilisation()* und *csma_frame_delay()*, deren Arbeitsweise unten näher erläutert wird.

Die zu diesem Meßpunkt gehörende Funktion *csma_probept_end_transmit_printf()* wird, falls es in der Konfigurationsdatei angegeben ist, für **jeden** Meßwert der entsprechenden Meßfunktion aufgerufen oder nur am Ende eines Simulationslaufes. Sie ermöglicht die Ausgabe der verschiedensten Variablenwerte, wie z.B. den aktuellen Meßwert, Sender- bzw. Empfängerstation des `CSMA-FRAME`s, simul_time, ... Alle definierten Ausgabemöglichkeiten werden von simuLan bei Aufruf mit der Option „**-x**" ausgegeben.

Genauere Informationen zu Meßpunkten sind in Abschnitt 4.2.7, Seite 19 und besonders in [Pet91] zu finden.

5.4.18.1 Die Meßfunktion *csma_utilisation()*

Zu dieser Meßfunktion gehören die Funktionen *csma_rc_utilisation()*, *csma_utilisation()* und *csma_utilisation_erg()*.

csma_rc_utilisation() aktiviert die eigentliche Meßfunktion *csma_utilisation()* und die dazugehörige Funktion *csma_utilisation_erg*, die nach jedem Simulationslauf die erhaltenen Meßwerte ausgibt.

In der Meßfunktion *csma_utilisation()* wird der Durchsatz des `CSMA`-Netzes nach folgender Gleichung bestimmt:

$$\frac{\textit{bisherige Zeit, während der } \texttt{CSMA}-\texttt{FRAME}\textit{s über das Netz übertragen wurden } (\texttt{ITU})}{\textit{bisherige Simulationszeit } (\texttt{ITU})}$$

Dabei wird die Zeitdauer *inter_frame_gap*, die nach jeder erfolgreichen Übertragung gewartet werden muß, zu der reinen Übertragungszeit für den `CSMA-FRAME` hinzugezählt. Es wird nur die Zeit für erfolgreich übertragene `CSMA-FRAME`s gezählt.

5.4.18.2 Die Meßfunktion *csma_frame_delay()*

Zu dieser Meßfunktion gehören die Funktionen *csma_rc_frame_delay()*, *csma_frame_delay()* und *print_mean_csma_frame_delay()*.

csma_rc_frame_delay() aktiviert die eigentliche Meßfunktion *csma_frame_delay()* und die dazugehörige Funktion *print_mean_csma_frame_delay()*, die nach jedem Simulationslauf die erhaltenen Meßwerte ausgibt.

In der Meßfunktion *csma_frame_delay()* wird die Zeit für die Übertragung des `CSMA-FRAME`s berechnet. Diese ergibt sich aus der Differenz der aktuellen Simulationszeit und dem Zeitpunkt, an dem der `CSMA-FRAME` in das Netzmodul eingetreten ist (csma_data_encapsulation).

5.5 Validierung des *CSMA*-Moduls

5.5.1 Einführung

In diesem Abschnitt wird die Validierung des `CSMA`-Modells und ein Vergleich des genauen und des optimierten Modells durchgeführt (siehe Abschnitt 5.3.2 [*Das genaue Simulationsmodell*], Seite 39 und Abschnitt 5.3.3 [*Das laufzeitoptimierte Modell*], Seite 44).

Die Validierung erfolgt durch den Vergleich der Simulationsergebnisse mit den in [BMK88] angegebenen Meßwerten. In diesem 1988 veröffentlichten Artikel „*Measured Capacity of an Ethernet: Myths and Reality*" stellen Boggs, Mogul und Kent die an einem **realen** System gewonnen Ergebnisse ausführlich anhand von Graphiken dar. Im Gegensatz zu vielen anderen Veröffentlichungen werden dort auch das System, die Erzeugung der Last und die Gewinnung der Meßwerte genau beschrieben.

Der folgende Unterabschnitt liefert eine genaue Beschreibung dieses Systems und der Meßumgebung, so wie es in dem sich anschließenden Abschnitt *Simulationsmodell* modelliert wird. Nachfolgend wird die benutzte Konfigurationsdatei erläutert. In Abschnitt 5.5.5 *Simulationsergebnisse* werden die Ergebnisse des genauen und des optimierten `CSMA`-Modells angegeben, verglichen und damit die spätere, ausschließliche Benutzung des optimierten Modells begründet. In dem nachfolgenden Unterabschnitt werden verschiedene Simulationsergebnisse für das `Ethernet` angegeben und im letzten Abschnitt *Auswertung und Beurteilung* mit denen aus [BMK88] verglichen.

5.5.2 Systembeschreibung

Das reale System, an dem die Messungen durchgeführt wurden, ist in Abb. 5.15 skizziert.

Die verwendeten *Multiportrepeater* bieten die Möglichkeit, an einen `Transceiver` mehrere Stationen anzuschließen, und verhalten sich ansonsten wie einfache `Repeater` (siehe Abschnitt 5.2 [*Das Vorbild:* Ethernet], Seite 35).
Der Abstand zwischen zwei benachbarten `Transceivern` betrug 1000 Fuß ($\approx$ 303 Meter).
Die bei den Messungen verwendeten Rechner waren `Titan`-Rechner, Prototypen, die von DEC's (Digital Equipment Corporation) Forschungslabor entwickelt wurden. Die Abb. 5.16 aus [BMK88] beschreibt das auf diesen Maschinen ablaufende lasterzeugende Programm.

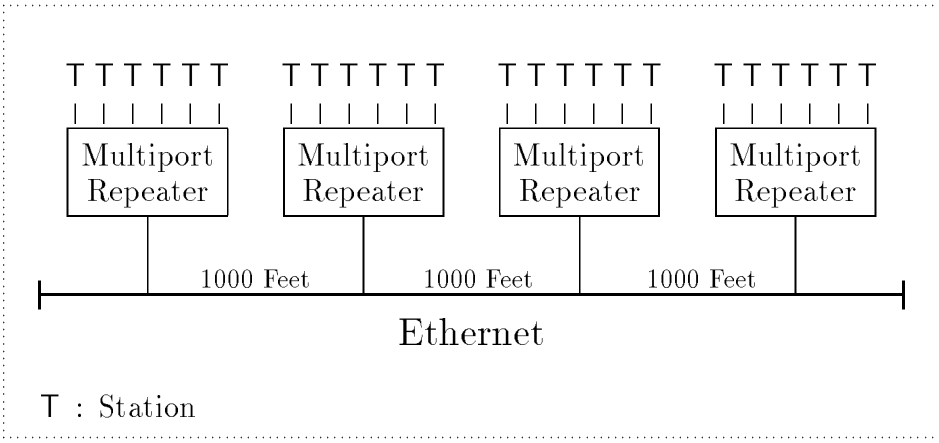


Abbildung 5.15: Das gemessene System (aus [BMK88])

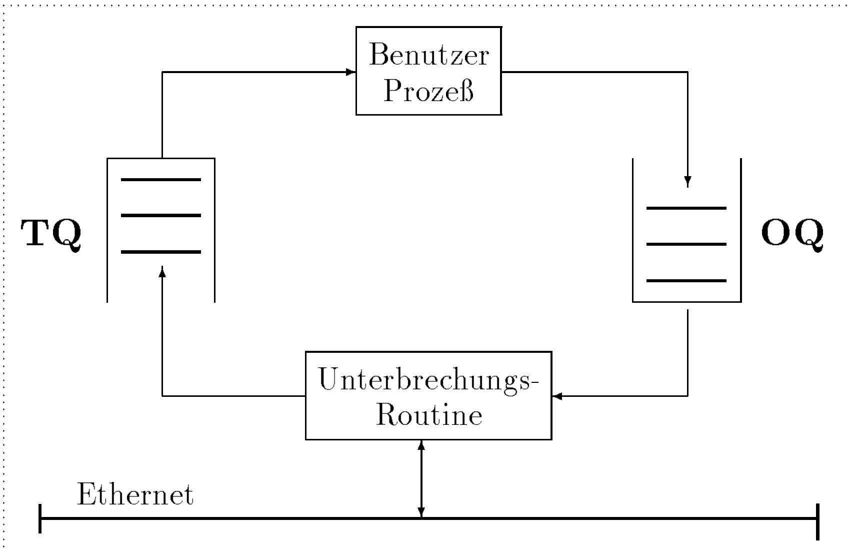


Abbildung 5.16: Das lasterzeugende Programm (aus [BMK88])

Vor Programmbeginn werden 15 leere `Nachricht`en in die Warteschlange (*Transmitted Queue*, *TQ*) mit gesendeten `Nachricht`en eingefügt. Zur Programmlaufzeit überprüft der Benutzerprozeß, ob eine `Nachricht` in der *TQ* steht. Ist dieses der Fall, wird in der `Nachricht` die Nachrichtengröße gesetzt und diese anschließend in die Warteschlange (*Output Queue*, *OQ*) zu sendender `Nachricht`en eingehängt. Bei leerer *TQ* wartet der Benutzerprozeß.
Falls diese Station gerade keine `Nachricht` sendet, wird die Übertragung der ersten `Nachricht` aus *OQ* angestoßen. Andernfalls wird auf die nächste Unterbrechung gewartet. Das Ende einer erfolgreichen Übertragung erzeugt diese Unterbrechung und aktiviert die dazugehörige Unterbrechungsroutine. Diese Routine entfernt zunächst die übertragene `Nachricht` aus *OQ* und hängt sie in die Warteschlange *TQ* ein. Falls noch eine `Nachricht` in *OQ* wartet, wird die Übertragung derselben angestoßen.

Der wichtigste Parameter dieses Programms ist die für die Unterbrechungsbehandlung benötigte Zeit. Für die verwendeten `Titan` wurde diese Zeit mit $100\mu s$ angegeben. Wenn die für das Senden einer `Nachricht` benötigte Zeit größer als $100\mu s$ ist, wird die Warteschlange *TQ* normalerweise voll und *OQ* normalerweise leer sein. Wenn die Sendezeit kürzer ist, wird dagegen normalerweise die Warteschlange *TQ* leer und *OQ* voll sein.
Zusätzliche durch den Kontroller verursachte Verzögerungszeiten treten nicht auf, da der Sender seine `Nachricht` an sich selber adressiert und gleichzeitig seinen Empfänger abschaltet.

Die Größe einer `Nachricht` wurde in Byte angegeben. Diese beinhaltete die Felder `Destination Address`, `Source Address`, `Length` und `LLC Data`.

Für die Bestimmung des `Ethernet`durchsatzes sowie der Übertragungszeiten, wurden zu jeder `Nachricht` noch 24 Byte hinzugezählt. Diese setzten sich aus 12 Byte `Inter Frame Gap`, 8 Byte `Preamble + SFD` und 4 Byte `FCS` zusammen (siehe Abschnitt 5.2 [*Das Vorbild:* Ethernet], Seite 37).
Die Übertragungszeit ergibt sich aus der Differenz der Zeit, zu der die Übertragung der `Nachricht` angestoßen wird, und dem Zeitpunkt der Unterbrechung, die durch das Übertragungsende erzeugt wird.
Falls eine `Nachricht` nach der maximalen Anzahl von erlaubten Versuchen nicht gesendet werden konnte (*exceeded collisions*), wurde der entsprechende Zähler zurückgesetzt und **sofort** wieder versucht, diesen `Nachricht` zu senden. Es erfolgte **keine** Unterbrechung!

Das `Ethernet` wurde für die Messung als störungsfrei angenommen.

Die Validierung des `CSMA`-Modells erfolgt anhand der angegebenen Meßwerte (Mittelwerte) für den Durchsatz (Mbit/s), für Pakete pro Sekunde und der Werte für die mittlere Übertragungszeit einer `Nachricht`.
Für den Durchsatz wurde zu der eigentlichen `Nachricht`engröße der Overhead und die vorhergehende Dauer des `Inter Frame Gap` hinzugezählt (siehe Abschnitt 5.2 [*Das Vorbild:* Ethernet], Seite 37). Damit ist der maximale erreichbare Durchsatz gleich der nominellen Übertragungsrate des Netzwerkes.
Die Übertragungszeit wurde, wie im vorhergehenden Absatz beschrieben, bestimmt.

Die Messungen wurden für `Nachricht`engrößen zwischen 64 Byte und 4000 Byte durchgeführt. Jeder Meßwert wurde für 1 bis 24 Stationen ermittelt, wobei diese **gleichmäßig** auf die `Multiportrepeater` verteilt waren.

5.5.3 Simulationsmodell

In Abb. 5.15 ist die Topologie des realen Systems angegeben. Das dazugehörige modellierte System ist aus Abbildung 5.17 zu ersehen.

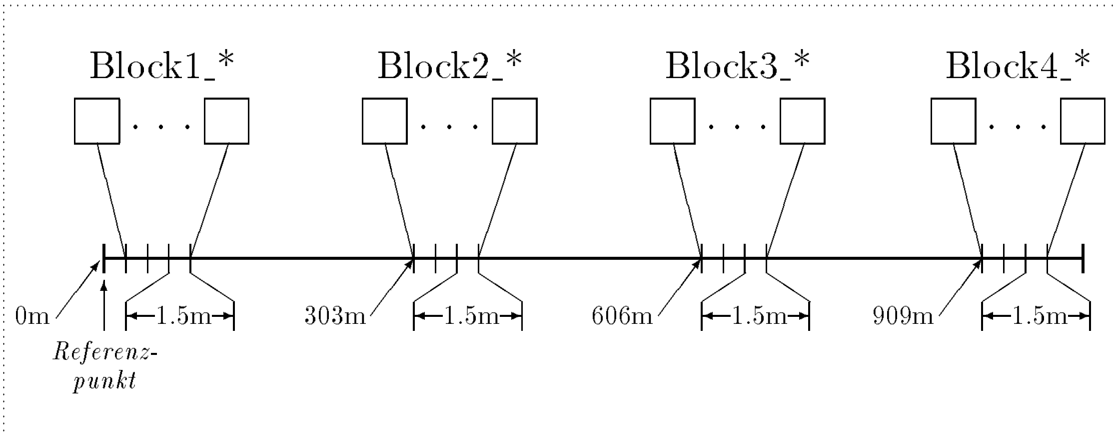


Abbildung 5.17: Topologie des Netzmodells

Da ein `Multiportrepeater` nur die Aufgabe hat, einen `Transceiver` mehreren Stationen zugänglich zu machen, aber sonst keinen signifikanten Einfluß auf das Zugriffsverfahren und die zu messenden Netzparameter hat, wird er nicht explizit modelliert. Stattdessen wird für jede Station ein eigener Anschlußpunkt (`Transceiver`) an das Netz modelliert. Der Abstand zwischen zwei Anschlußpunkten in einem Block wird zu 1.5 Meter angenommen. Dieses ist der minimale nach dem Standard zuläßige Abstand und entspricht einer Signallaufzeit von ca. $6ns$. Die durch den `Multiportrepeater` erzeugten Signalverzögerungen liegen in einer ähnlichen Größenordnung. Die übrigen Netzparameter entsprechen den in `IEEE802.3` angegebenen Standardwerten und sind in der Konfigurationsdatei angegeben.

Die zu modellierende Funktionalität einer `Titan` ist die Erzeugung von `Nachricht`en der richtigen Größe und das anschließende Senden derselben. Dabei muß berücksichtigt werden, daß

zwischen dem Ende einer Übertragung und dem Beginn der nächsten eine Bearbeitungszeit von $100\mu s$ benötigt wird.
Für die Modellierung der `Titan` wird das in Abschnitt 4.4.1 [*Das Knotenmodul* K0], Seite 25, beschriebene Modell `K0` verwendet.
Die Bearbeitungszeit wird über den Parameter *zw_ank_distr* auf konstant $100\mu s$ eingestellt.
Über die Belegung des Parameters *calc_empfaenger* mit dem Wert *self_empf* wird erreicht, daß alle Stationen ihre `Nachricht`en an sich selber adressieren. Bei Empfang einer `Nachricht` wird in dem `K0`-Modell die Anzahl empfangener `Nachricht`en erhöht, und die Meßfunktionen werden aufgerufen. Es wird **keine** Simulationszeit verbraucht. Dieses Verhalten entspricht dem der zu modellierenden Stationen.

Die `Nachricht`engröße wird folgendermaßen bestimmt:
Die `Titans` erzeugen `Nachricht`en, die 64 Byte Daten enthalten. Zu diesen 64 Byte werden in [BMK88] noch 8 Byte `Preamble + SFD` und 4 Byte `SCF` hinzugezählt. Die 64 Byte umfassen somit auch die Felder `Destination Address, Source Address` und `Length` eines `Ethernet-Frames` (siehe Abschnitt 5.2 [*ETHER-FRAME*], Seite 37). Es werden also auf dem Netz pro `Frame` 76 Byte übertragen.
In dem Modell ergibt sich die `Frame`größe aus der Summe der Anzahl zu übertragender Nutzdaten (`Nachricht`engröße), die im `K0`-Modul berechnet wird, und dem über die Konfiguration vorgegebenen Overhead, der im `CSMA`-Netzmodul zu jeder `Nachricht` beim Aufbau eines `Frame`s hinzugefügt wird.
Für eine äquivalente `Nachricht`engröße muß eine durch ein `K0`-Modul erzeugte `Nachricht` eine konstante Größe von **64** Byte besitzen. Der Overhead muß mit **12** Byte in der Konfiguration für das `CSMA`-Modul vorgegeben werden.

5.5.4 Die Konfigurationsdatei

Der Vorspann in der Konfigurationsdatei enthält folgende Zeilen:

```
#  ---- Vorspann ----
##      predefined are :
##      FILE NIL <NULL>         # no file at all, not even /dev/null !

# Für die Ergebnisse aller Statistiken
FILE statistik_ausg ">> 16_node_stat_erg"

# Für die Ergebnisse der Meßfunktion ,,csma_utilisation''
FILE utilisation_ausg ">> 16_node_util_erg"
# Für die Ergebnisse der Meßfunktion ,,frame_delay''
FILE fr_delay_ausg ">> 16_node_fr_delay_ausg"
# Für die Ergebnisse der Meßfunktion ,,msg_delay'' des Blocks ,1_b'
FILE msg_del_b1 ">> 16_node_msg_delay_b1"
# Für die Ergebnisse der Meßfunktion ,,msg_delay'' des Blocks ,2_b'
FILE msg_del_b2 ">> 16_node_msg_delay_b2"
# Für die Ergebnisse der Meßfunktion ,,msg_delay'' des Blocks ,3_b'
FILE msg_del_b3 ">> 16_node_msg_delay_b3"
# Für die Ergebnisse der Meßfunktion ,,msg_delay'' des Blocks ,4_b'
FILE msg_del_b4 ">> 16_node_msg_delay_b4"

# Laufvariable für die Größe der zu sendenden Nachricht eines k0-Knotens
FOR double m FROM 64 TO 2048 STEP 64

# ,Laufvariablen' für die Anzahl Knoten in jedem der vier Blöcke.
FOR int 1_b FROM 4 TO 4
FOR int 2_b FROM 4 TO 4
FOR int 3_b FROM 4 TO 4
FOR int 4_b FROM 4 TO 4
```

Im folgenden werden die definierten symbolischen Dateinamen verwendet. Die Zuordnung des symbolischen Namens zum Namen der Datei, die von simuLan erzeugt wird, und der Inhalt dieser Datei nach einem Simulatonslauf sind in der Konfiguration angegeben.

Die Laufvariable m enthält in jedem Simulationslauf die aktuelle Größe der von einem Knoten `K0` erzeugten `Nachricht`. Der Startwert ist 64 Byte. Nach jedem Simulationslauf wird für den nächsten Lauf die `Nachricht`engröße um 64 Byte erhöht. Die letzte simulierte `Nachricht`engröße ist 2048 Byte und damit schon größer als die nach dem Standard `IEEE 802.3` maximal zuläßige (siehe Abschnitt 5.2 [*Das Vorbild:* Ethernet], Seite 35).

Die Laufvariablen *1_b*, *2_b*, *3_b* und *4_b* geben die Anzahl der simulierten `Titans` in den Blöcken *Block1_** bis *Block4_** (s. Abbildung 5.17, Seite 62) an. Diese Laufvariablen haben für alle Simulationsläufe, die durch diese Konfiguration beschrieben werden, den gleichen konstanten Wert *4*.

```
## --- Body of Configuration ---
BEGINconf

ITU             1e9             # indivisible time-units per Second
max_events      1e8             # Max. Number of Events to simulate
max_simul_time  2e15            # max time to simulate
max_cpu_time    200000          # max cycles to succ
```

Die Auflösung der Simulationszeit wird auf *1ns* gesetzt. Diese Genauigkeit ist notwendig, da die Signallaufzeiten zwischen zwei Stationen in dieser Größenordnung liegen.
Die maximale Anzahl `EVENT`s (*max_events*), die maximal zu simulierende Simulationszeit (*max_simul_time*) und die maximale erlaubte reale Rechenzeit (*max_cpu_time*) werden mit so großen Werten belegt, daß sie sicher nicht erreicht werden. Das `Ethernet` wird über folgende Zeilen in der Konfiguration definiert:

```
NET                             # read the Net-Config
  CSMA                          # CSMA/CD
        bits_per_second 10e6    # Net Speed
        overhead        96 bit
        max_data_len    20e3
        itu_pro_meter   4.2

#     --- Defaults ---
        inter_frame_gap 96 bit
        jam_time        32 bit
        min_send_bits   64 bit
        min_data_len    368 bit
        slot_time       512 bit
        attempt_limit   16
        backoff_limit   10
#     ----------------
        calc_backoff            # Function to compute backoff-times
                csma_backoff    # exp Backoff a la 802.3
        error_distr             # ditribution of transmission errors
                konstant 1e50   # Never !
        calc_laufzeit
                linear
                Distance "Block1_*" 1.5 m
                distance "FIXPOINT" 303 m
                distance "Block2_*" 1.5 m
                distance "FIXPOINT" 606 m
```

```
                distance "Block3_*" 1.5 m
                distance "FIXPOINT" 909 m
                distance "Block4_*" 1.5 m

        end_transmit_probept
           MEASURE NIL csma_utilisation
                STATISTIC WAIT statistik_ausg

# Ausgabe folgendes/r Textes [Variablenwerte] nach jedem Simulationslauf:
# [Anzahl Knoten] [Nachrichtengröße] [Durchsatz in Mbit/s (inkl. overhead)]
# [Anzahl Pakete/s]
                        FORMAT utilisation_ausg "%c%%$m%%e%%p% \n"
                        bmva delete_count           4000
                ENDbmva

           MEASURE NIL frame_delay
                STATISTIC WAIT statistik_ausg

# Ausgabe folgendes/r Textes [Variablenwerte] nach jedem Simulationslauf:
# [Anzahl Knoten] [Nachrichtengröße] [Mittlere Verzögerung eines FRAMEs]
                        FORMAT fr_delay_ausg "%c%%$m%%S% \n"
                        bmva delete_count           4000
                ENDbmva

    ENDcsma                     # end of ethernet-description
ENDnet
```

Alle im folgenden nicht erwähnten Parameter des `Ethernet` haben die in dem Standard `IEEE 802.3` angegebenen Werte und stehen in der Konfiguration unter der Überschrift *Defaults*.
Die nominelle Übertagungsgeschwindigkeit wird auf 10 Mbit/s gesetzt, und der von dem Netzmodul bei Bildung eines `Frames` zu einer `Nachricht` hinzugefügte Overhead wird mit dem im vorherigen Abschnitt ermittelten Wert von 96 Bit (12 Byte) initialisiert.
Im Knotenmodul `K0` wird anhand von *max_data_len* die maximale `Nachrichtengröße` ermittelt. Falls die berechnete Größe diesen Wert überschreitet, wird die `Nachricht` entsprechend verkleinert. Die maximale simulierte Größe wird durch die Laufvariable m mit 2048 Byte (16384 Bit) vorgegeben. Der verwendete Wert von $20 \cdot 10^3$ Bit für *max_data_len* stellt sicher, daß `Nachrichten` durch das Knotenmodul `K0` nicht verkleinert werden.

Für die Bestimmung der Signallaufzeit zwischen zwei Stationen wird die Signalausbreitungsgeschwindigkeit im Netz benötigt. Der Standard `IEEE 802.3` gibt diese mit ca. 80% Lichtgeschwindigkeit an. Für die Verwendung in der Konfiguration muß dieser Wert in die Einheit ‚`ITU` pro Meter' umgerechnet werden.
80% Lichtgeschwindigkeit entsprechen ca. 0.238 m/ns. Da ein `ITU` einer Nanosekunde entspricht, ergibt sich der Wert für den Parameter *itu_pro_meter* direkt aus 0.238^{-1} m/ns zu 4.2 ITU/m.

Die Backoffwartezeiten werden in der Simulation mit der Funktion *csma_backoff()* bestimmt, die den Backoffalgorithmus nach `IEEE 802.3` nachbildet.
Da keine Übertragungsfehler während der Simulaton auftreten sollen, wird die Zeit des ersten Fehlers auf $1 \cdot 10^{50}$ `ITU` gesetzt und ist damit größer als die vorgegebene maximale zu simulierende Zeit.

Die Signallaufzeiten werden über die Funktion *linear* bestimmt. Die mit `distance` beginnenden Zeilen beschreiben die in Abb. 5.17 angegebene Netztopologie (siehe Abschnitt 5.4.16.2 [*csma_linear()*], Seite 57). Dabei sind die Anschlußpunkte der Stationen innerhalb eines Blocks jeweils 1.5 m voneinander entfernt. Der Abstand zwischen zwei benachbarten Blöcken beträgt 303 m.
Mit `FIXPOINT` kann ein absoluter Abstand zum *Referenzpunkt* vorgegeben werden. Alle anderen Entfernungsangaben sind jeweils *relativ* zu dem letzten angegebenen Anschlußpunkt oder

FIXPOINT.

Für den Meßpunkt *end_transmit_probept* werden zwei Meßfunktionen definiert. Es werden jeweils die Mittelwerte der Meßspuren ermittelt.
Die erste Funktion (*csma_utilisation*) bestimmt den mittleren Durchsatz des **Ethernet** in Mbit/s für jeden Simulationslauf und gibt diesen in die Datei '**utilisation_ausg**' zusammen mit der Anzahl simulierter Knoten, der aktuellen **Nachricht**engröße und der Anzahl Pakete pro Sekunde aus. Die zweite Meßfunktion (frame_delay) bestimmt die mittlere für die Übertragung einer **Nachricht** benötigte Zeit und gibt diese in die Datei '**fr_delay_ausg**', zusammen mit der Anzahl simulierter Knoten und der aktuellen **Nachricht**engröße, aus.
Für die statistische Absicherung wird in beiden Fällen das von simuLan bereitgestellte Absicherungs-Modul **bmva** mit den vorgegebenen Defaultwerten verwendet (s. [Pet91]). Die ersten 4000 Meßwerte bleiben jeweils bei der Bestimmung der gesuchten Mittelwerte unberücksichtigt, um eine Verfälschung der Simulationsergebnisse durch die Einschwingphase des Netzwerks zu verhindern.

Die zu den **Titans** gehörenden K0-Knotenmodelle werden mit den folgenden Zeilen in der Konfiguration initialisiert. Es wird nur die Definition des Blocks *Block1_** angegeben. Die entsprechenden Zeilen für die Blöcke *Block2_**, *Block3_** und *Block4_** ergeben sich analog.

```
NODES                                  # read the Node-Configs

    NAME "Block1_%d" COUNT $1_b
    LAYER5 k0
## Parameters for k0-node
    zw_ank_distr                       # prob. distr. for messages'
                                       # interarrival-times
       konstant 100 us
    msg_size_distr                     # prob. distr. for message sizes
        konstant $m byte
    calc_empfaenger                    # Function to determine the
                                       # receiver-nodenumber
        self_empf                      # Receiver is the Sender

    send_probept
        MEASURE NIL msg_delay
                STATISTIC WAIT COMMON statistik_ausg

# Ausgabe folgendes/r Textes [Variablenwerte] nach jedem Simulationslauf:
# b1 [Anzahl Netzknoten] [Aktuelle Nachrichtengröße] [mittlere Verzögerung
# einer Nachricht]
                FORMAT msg_del_b1 "%c%%$m%%s% \n"
                  bmva
                       delete_count           4000
                ENDbmva

    ENDk0   #          <<end block>>   # end of k0-descr.
```

Es werden *$1_b* identische Knoten initialisiert, die die Namen `Block1_0`, `Block1_1`, ..., `Block1_($1_b-1)` erhalten. Die Bestimmung der Werte für die Parameter *zw_ank_distr*, *msg_size_distr* und *calc_empfaenger* ist im Abschnitt 5.5.3 ausführlich beschrieben worden.
Für den Meßpunkt *send_probept* wird die Meßfunktion *msg_delay* aktiviert. Diese bestimmt die Zeit, die zwischen dem ersten Sendeversuch und dem Ende der erfolgreichen Übertragung der **Nachricht** vergeht. Für die statistische Absicherung wird wieder das Modul **bmva** verwendet. Alle Meßfunktionen eines Blocks geben ihre Meßwerte an die gleiche Absicherungsfunktion. Der Ergebniswert für die Übertragungszeit ist also ein Mittelwert über **alle** in diesem einen Block ermittelten Übertragungszeiten. Dieser Wert wird zusammen mit der Gesamtanzahl simulierter Knoten und der aktuellen **Nachricht**engröße in die für '**msg_del_b1**' definierte Datei geschrieben.

5.5.5 Simulationsergebnisse

Dieser Abschnitt vergleicht zunächst die Simulationsergebnisse für das genaue und für das optimierte Modell, die sich aus der im vorhergehenden Abschnitt beschriebenen Konfiguration ergeben.
Anschließend werden die für die Validierung verwendeten Simulationsergebnisse angegeben und mit den in [BMK88] aufgeführten Meßwerten verglichen. Alle für die Validierung verwendeten Simulationergebnisse wurden unter Verwendung des **optimierten** Netzmoduls `CSMA` ermittelt, da dieses eine geringere Laufzeit benötigt und dessen Ergebnisse praktisch identisch mit denen des **genauen** Modells sind (siehe nächster Unterabschnitt).

Die im folgenden dargestellten Graphiken geben die Simulationsergbnisse in Abhängigkeit von der Größe der `Nachricht` an. Diese Größe wird in der Konfiguration direkt über den jeweiligen Wert der Laufvariablen m vorgegeben.
Für die Berechnung des `Ethernet`-Durchsatzes wurde zu dieser Größe der Netzoverhead von 12 Byte sowie der `Inter Frame Gap` (12 Byte) hinzugezählt. Damit ergibt sich der maximale erreichbare Wert für den Durchsatz zu 10 Mbit/s.
Die Übertragungszeit einer `Nachricht` repräsentiert die Zeit vom ersten Sendeversuch der `Nachricht` bis zum Ende der erfolgreichen Übertragung.
Die Anzahl Kollisionen bzw. Exceeded Kollisionen wird relativ zu **allen** über das Netz erfolgreich übertragenen `Frame`s angegeben. Durch die Exceeded Kollisionen verursacht, kann zu einer `Nachricht` mehr als ein erzeugter `Frame` gehören. Die Anzahl Kollisionen pro `Nachricht` kann somit größer als *attempt_limit* sein.

5.5.5.1 Vergleich: genaues – optimiertes *CSMA*-Modell

Die Abb. 5.18, Seite 68 zeigt den mittleren Durchsatz des `Ethernet` für das optimierte und das genaue Modell. In Abb. 5.20 und 5.21, Seite 69 ist die dazugehörige Anzahl Kollisionen bzw. Exceeded Kollisionen angegeben. Die Abb. 5.23, Seite 70 zeigt die mittlere Übertragungszeit einer `Nachricht`, die durch einen Knoten `K0` erzeugt wurde, und Abb. 5.22 die mittlere Übertragungszeit eines `CSMA-FRAME`s. Die Kurven zeigen jeweils eine sehr gute Überdeckung. Einzige Ausnahme ist der für eine `Nachricht`engröße von 64 Byte ermittelte Durchsatz des `Ethernet`. Dabei liefert das genaue Modell einen um ca. 0.2 Mbit/s niedrigeren Wert als das optimierte.
Abb. 5.20 zeigt, daß bei dieser `Nachricht`engröße eine große mittlere Anzahl von Kollisionen für jeden dazugehörigen `Frame` auftritt. Zusätzlich ist die Anzahl Pakete (übertragene `Frames`) pro Sekunde (s. Abb. 5.19), die sich aus dem Durchsatz berechnen läßt, maximal. Bei dieser `Frame`größe wird damit die größte Anzahl Kollisionen während eines Simulationslaufes simuliert. Die im optimierten Modell vorgenommene Vereinfachung bei der Simulation der Kollisionsbehandlung wirkt sich deshalb hier am stärksten aus.
Bei der Bewertung der Abweichung ist zu berücksichtigen, daß 0.2 Mbit/s einer Abweichung von nur **2.3** Prozent gegenüber dem Wert von 8.73 Mbit/s entspricht.
Aus diesem Grund ist die Abweichung nicht als signifikant anzusehen.

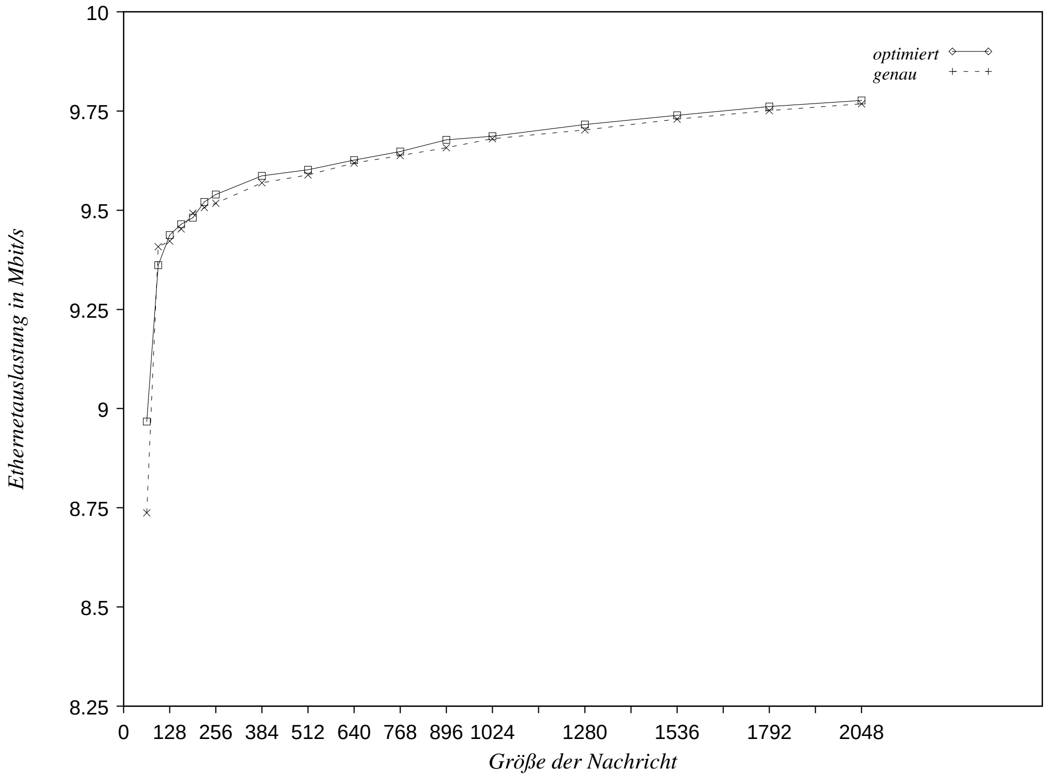


Abbildung 5.18: Ethernetdurchsatz in Abhängigkeit von der `Nachricht`engröße

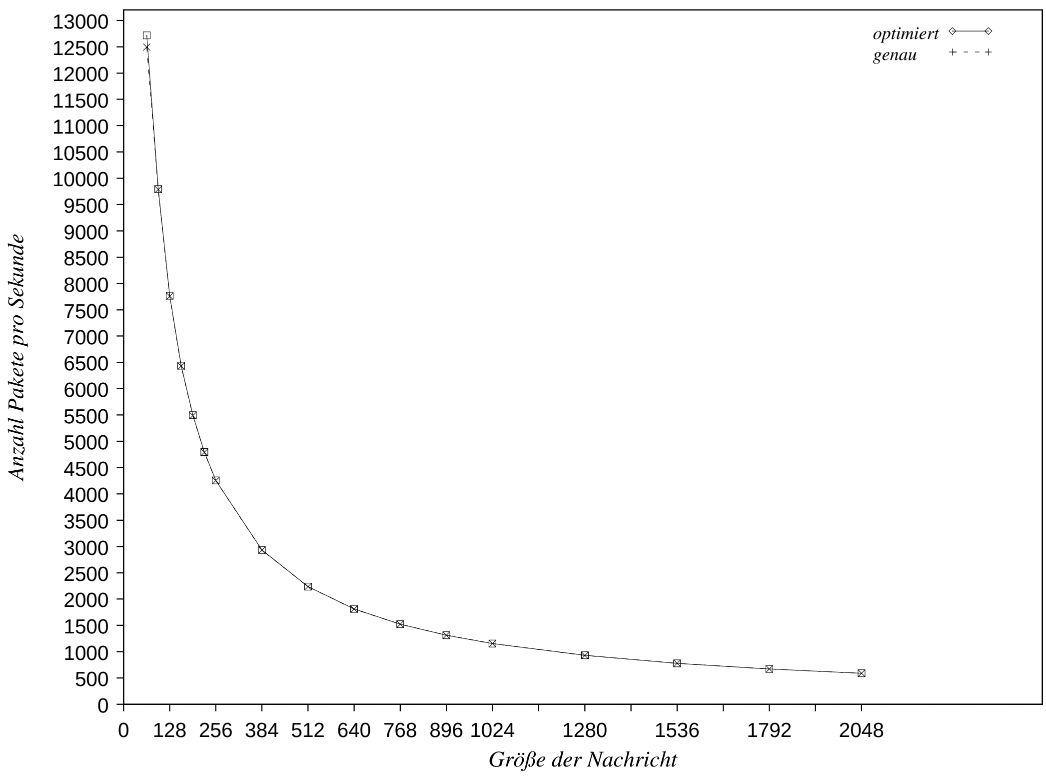


Abbildung 5.19: Anzahl Pakete pro Sekunde in Abhängigkeit von der `Nachricht`engröße

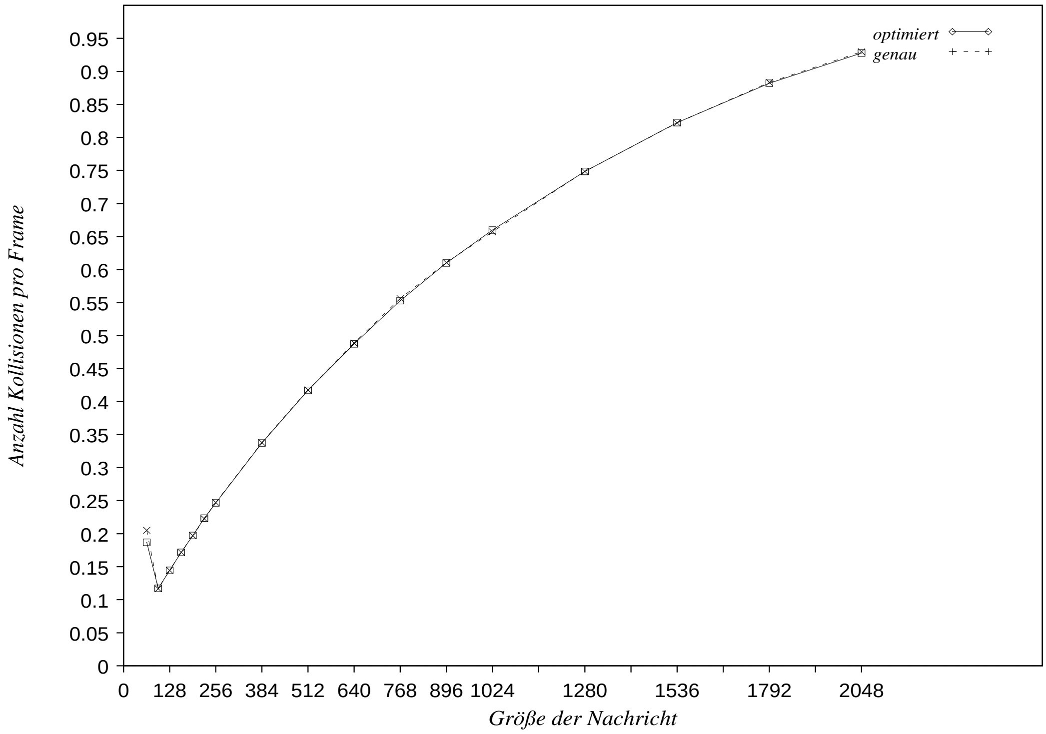


Abbildung 5.20: Anzahl Kollisionen in Abhängigkeit von der `Nachricht`engröße

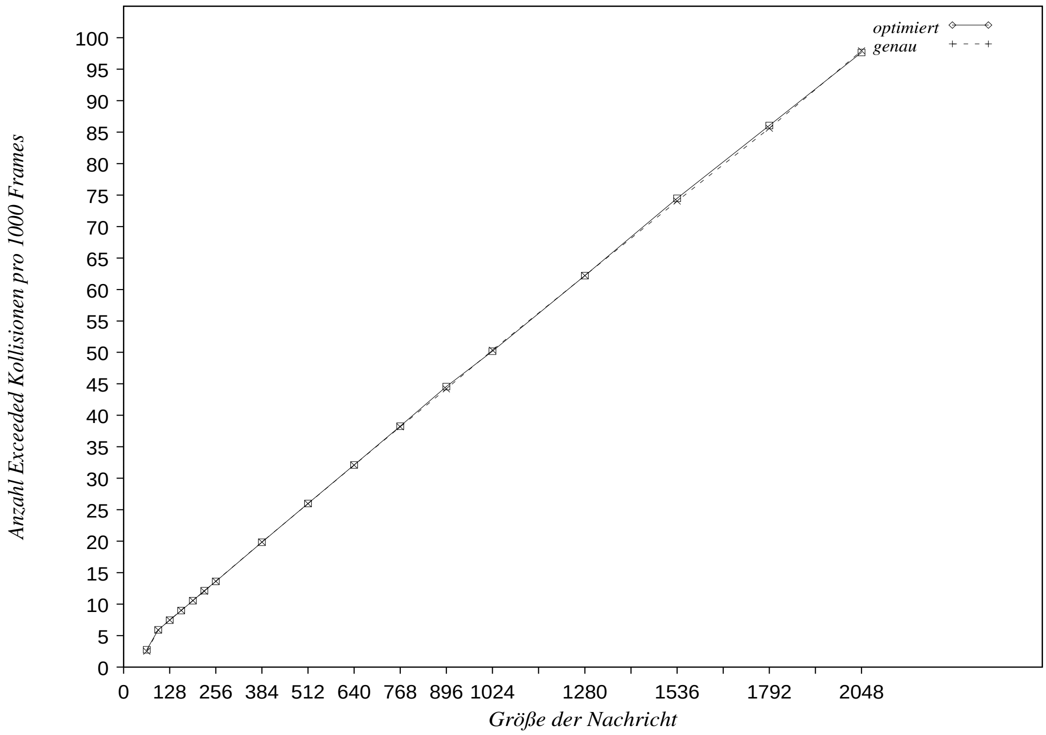


Abbildung 5.21: Anzahl Exceeded Kollisionen in Abhängigkeit von der `Nachricht`engröße

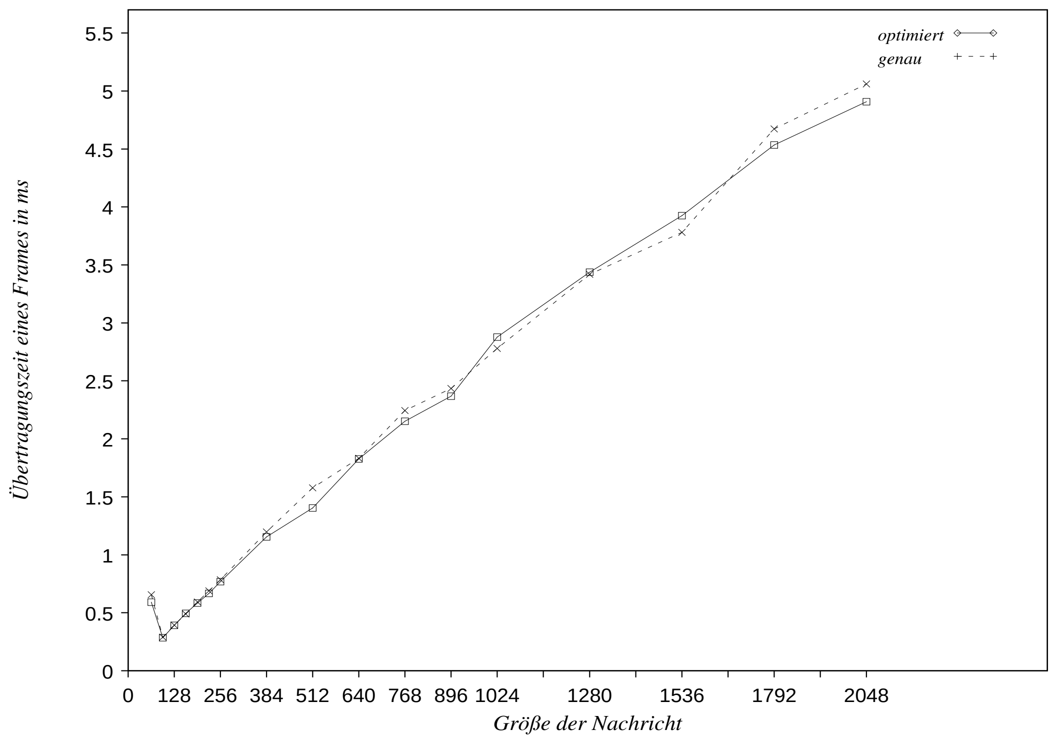


Abbildung 5.22: Übertragungszeit eines Frames (64 - 2048 Byte `Nachricht`engröße)

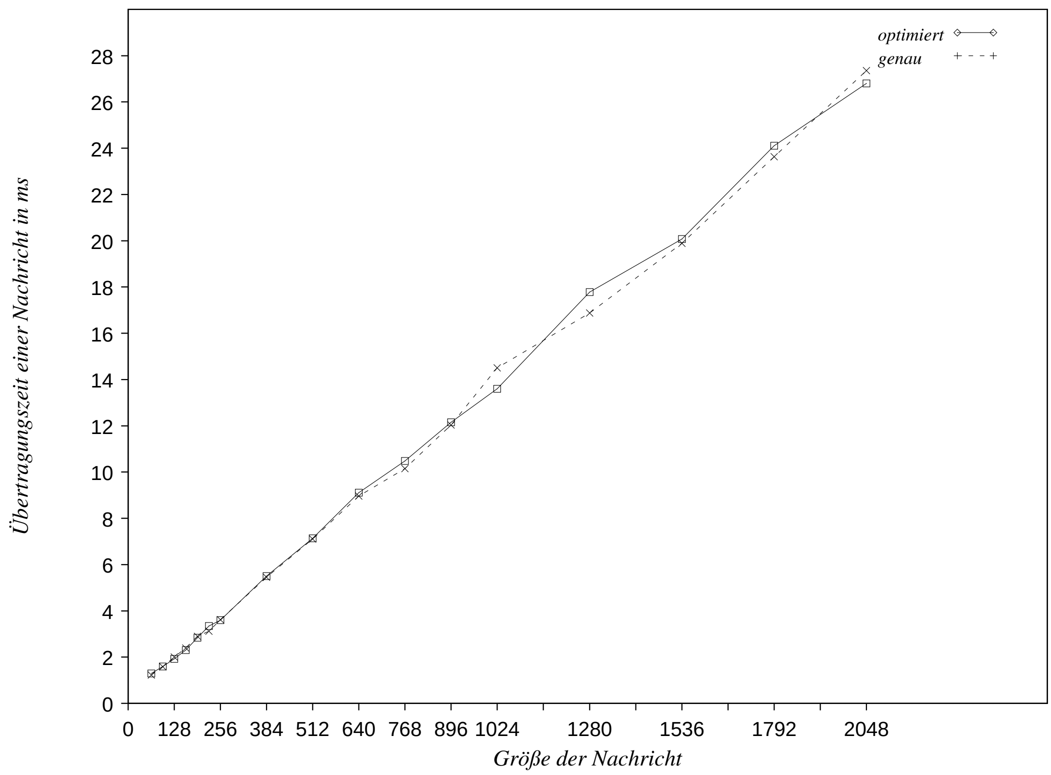


Abbildung 5.23: Übertragungszeit einer Nachricht in Abhängigkeit von der `Nachricht`engröße

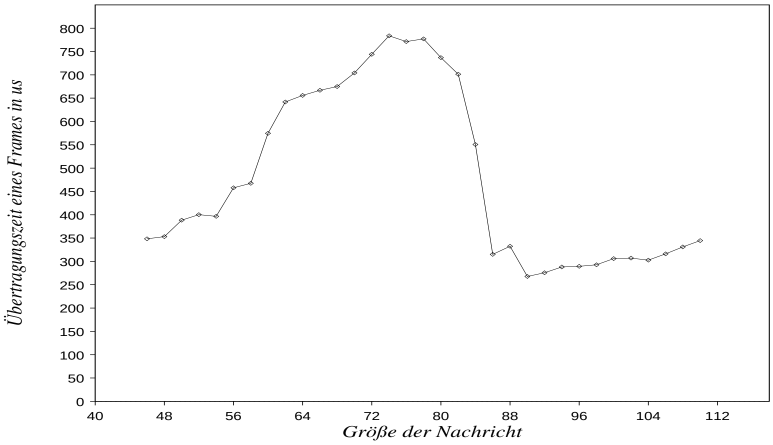


Abbildung 5.24: Übertragungszeit eines Frames (46 - 110 Byte `Nachricht`engröße)

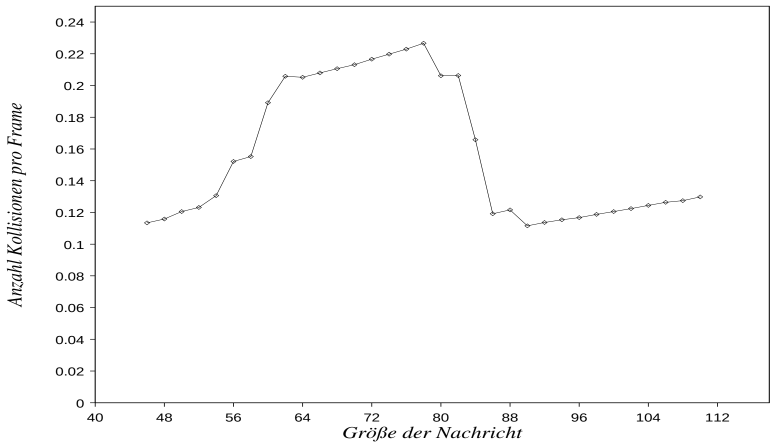


Abbildung 5.25: Anzahl Kollisionen (46 - 110 Byte `Nachricht`engröße)

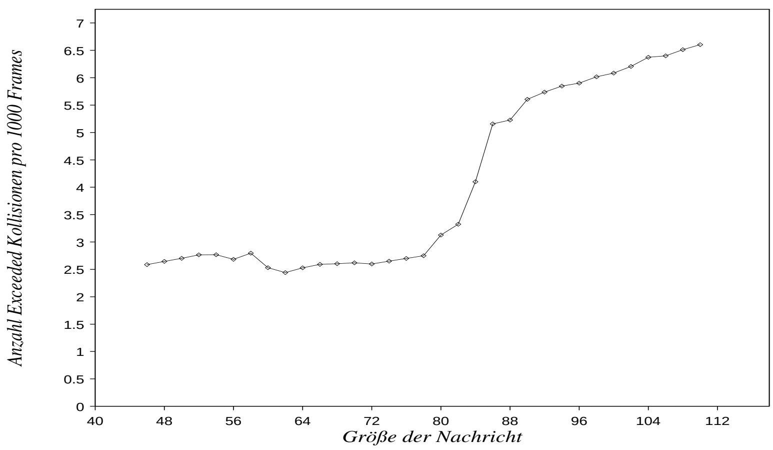


Abbildung 5.26: Anzahl Exceeded Kollisionen (46 - 110 Byte `Nachricht`engröße)

Die Abb. 5.22, Seite 70, zeigt die mittlere Übertragungszeit für einen `Frame` über das Netz . Auffällig an diesem Graphen ist, daß die Übertragungszeit zunächst abnimmt, um dann ab einer `Nachricht`engröße von ca. 90 Byte wieder anzusteigen. In Abb. 5.24 ist dieser Graph für `Nachricht`engrößen von 46 bis 110 Byte noch einmal dargestellt. Die Abb. 5.25 und 5.26 zeigen die dazugehörige mittlere Anzahl Kollisionen bzw. Exceeded Kollisionen pro `Frame`. Diese Graphen zeigen, daß die Anzahl Kollisionen in diesem Bereich stark ab- und die Anzahl Exceeded Kollisionen stark zunimmt. Dieses Verhalten beruht auf der Tatsache, daß eine Station, die eine `Nachricht` erfolgreich übertragen konnte, für die Erzeugung des nächsten Sendewunsches 100 μs („Totzeit") benötigt und während dieser Zeit das Netz nicht belastet. Wenn das Netz nur zwei Stationen besäße, würde es nach der ersten Kollision keine weiteren geben, da die beiden Stationen sich sysnchronisieren und ihre `Nachricht` jeweils während der „Totzeit" der anderen Station senden würden. Dieses funktioniert so lange, wie die zu sendende `Nachricht` + Overhead + `Inter Frame Gap` kleiner als diese „Totzeit" ist. Wenn die `Nachricht` größer ist, folgen weitere Kollisionen und eine Station kann erst senden, wenn die andere so lange im Backoff verweilt, daß sie beim nächsten Sendeversuch den Kanal als belegt erkennt und wartet. Eine „Totzeit" von 100 μs entspricht 125 Byte-Zeiten. Die entsprechende maximale Größe der `Nachricht`, die in diese Zeit paßt, beträgt 101 Byte = 125 Byte - 24 Byte (Overhead + `Inter Frame Gap`). Man erwartet also eine relativ geringe Anzahl Exceeded Kollisionen für `Nachricht`engrößen kleiner als 101 Byte und dann einen deutlichen Sprung. Tatsächlich tritt dieser Sprung schon bei Größen um ca. 86 Byte auf. Die restliche Zeit von 15 Byte-Zeiten wurde für die Auflösung der auftretenden Kollisionen benötigt.
Die Abnahme der **mittleren** Übertragungszeit eines `Frame`s beruht darauf, daß bei zunehmender `Frame`größe, und damit zunehmender erzeugter Last eine zunehmende Anzahl `Frames` mehr als *attempt_limit* Kollisionen erfährt. Die Station sendet nach diesem Ereignis (Exceeded Kollisionen) die `Nachricht` **sofort** noch einmal. Der neu erzeugte `Frame` ist gegenüber anderen im Vorteil, da seine Wartezeit nach einer eventuellen weiteren Kollision, bestimmt nach dem Verfahren des *truncated binary exponential backoff*, deutlich kleiner als die der „älteren" `Frames` ist. Dieser `Frame` wird also nach sehr wenigen Kollisionen erfolgreich übertragen und somit den **Mittelwert** der Übertragungszeit eines `Frame`s verkleinern.
Dieser Effekt wird immer geringer, da mit zunehmender `Nachricht`größe eine immer größere Anzahl `Frame`s mehr als *attempt_limit* Kollisionen erfährt und dann erneut gesendet wird.

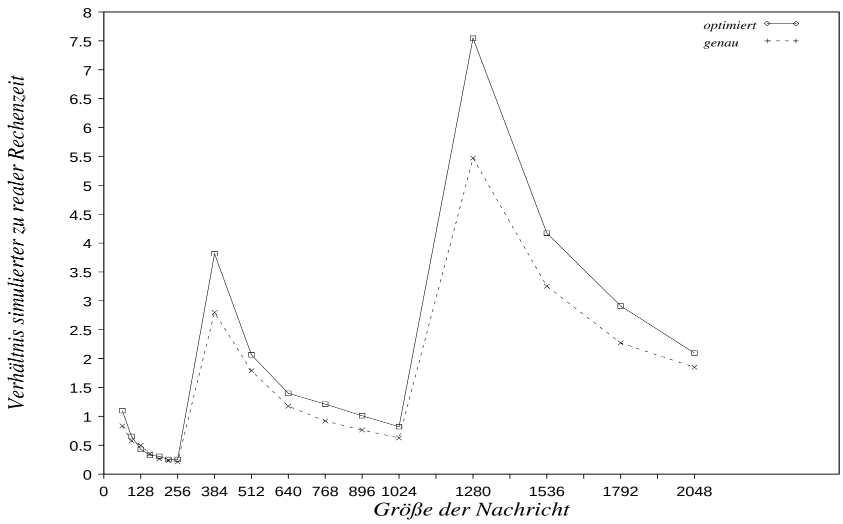


Abbildung 5.27: Verhältnis simulierter zur realer Rechenzeit

In der Abb. 5.27, Seite 72 ist das Verhältnis der simulierten Zeit zu real benötigten Rechenzeit angegeben. Dieses Verhältnis ist für das optimierte Modell, besonders bei vielen zu simulierenden Kollisionen, deutlich besser als das des genauen Modells.
Alle dargestellten Meßwerte, bis auf die der Abb. 5.23, 5.25 und 5.26, wurden auf einem Rechner vom Typ SUN Sparc 2 in einem durchgehenden Simulationslauf ermittelt. Dabei benötigte das genaue Modell eine Rechenzeit von 110 Minuten und das optimierte Modell 91 Minuten. Dieses entspricht einer Laufzeiteinsparung von 17 Prozent.

Pro Simulationslauf wurde die Übertragung von im Mittel 900000 (min: 179325, max: 3061246) `Frames` simuliert. Diese große Anzahl wurde benötigt, um die Übertragungszeit für eine `Nachricht` statistisch absichern zu können.

Da die Simulationsergebnisse für das genaue und das optimierte Modell weitgehend identisch sind und das optimierte Modell eine deutliche Reduzierung der benötigten Laufzeit ermöglicht, wird für die Validierung ausschließlich das optimierte Modell verwendet.

5.5.5.2 Ergebnisse der Validierungsläufe

Die in diesem Abschnitt wiedergegebenen Graphiken entstanden mit der in Abschnitt 5.5.4, Seite 63 beschriebenen Konfiguration. Dabei wurde mit einer Station (Laufvariable $1\_b = 1$) bei ,,Block1_*" (siehe Abschnitt 5.17 [*Simulationsmodell*], Seite 62) begonnen und die Meßwerte für alle `Nachricht`engrößen in einem Programmlauf bestimmt. Anschließend wurde eine zweite Station bei ,,Block2_*" eingefügt (Laufvariable $2\_b = 1$) und ein neuer Lauf gestartet. Die dritte Station wurde bei ,,Block3_*" und die vierte bei ,,Block4_*" angeschlossen. Die fünfte Station wurde dann wieder ,,Block1_*" zugeteilt usw. .

Bei den folgenden Graphiken ist auf der X-Achse immer die Anzahl simulierter Stationen und auf der Y-Achse der Meßwert aufgetragen (Mittelwert der Meßreihe).
Hinter jedem Graphen ist die dazugehörige `Nachricht`engröße, wie sie in der Konfiguration durch die Laufvariable m vorgegeben wurde, angegeben.
Der Durchsatz des `Ethernet` und die Übertragungszeiten für `Frames` sowie `Nachricht`en wurden, wie im vorhergehenden Abschnitt beschrieben, ermittelt.

Die Abb. 5.28, Seite 73 zeigt den ermittelten Durchsatz und Abb. 5.29 die sich daraus ergebende Anzahl Pakete (erfolgreich übertragene `Nachricht`en) pro Sekunde.
In Abb. 5.31, Seite 75 wird die für die Übertragung eines `Frames` und in Abb. 5.30 die für die Übertragung einer `Nachricht` benötigte Zeit dargestellt.
Als letzte Graphik ist in Abb. 5.32, Seite 76 die Übertragungszeit einer `Nachricht` bei bis zu 200 simulierten Stationen am Netz dargestellt. Es wurden die Stationsanzahlen 1-24, 50, 75, 100, 150 und 200 simuliert. Die Stationen wurden wie vorher gleichmäßig auf die vier Blöcke verteilt.

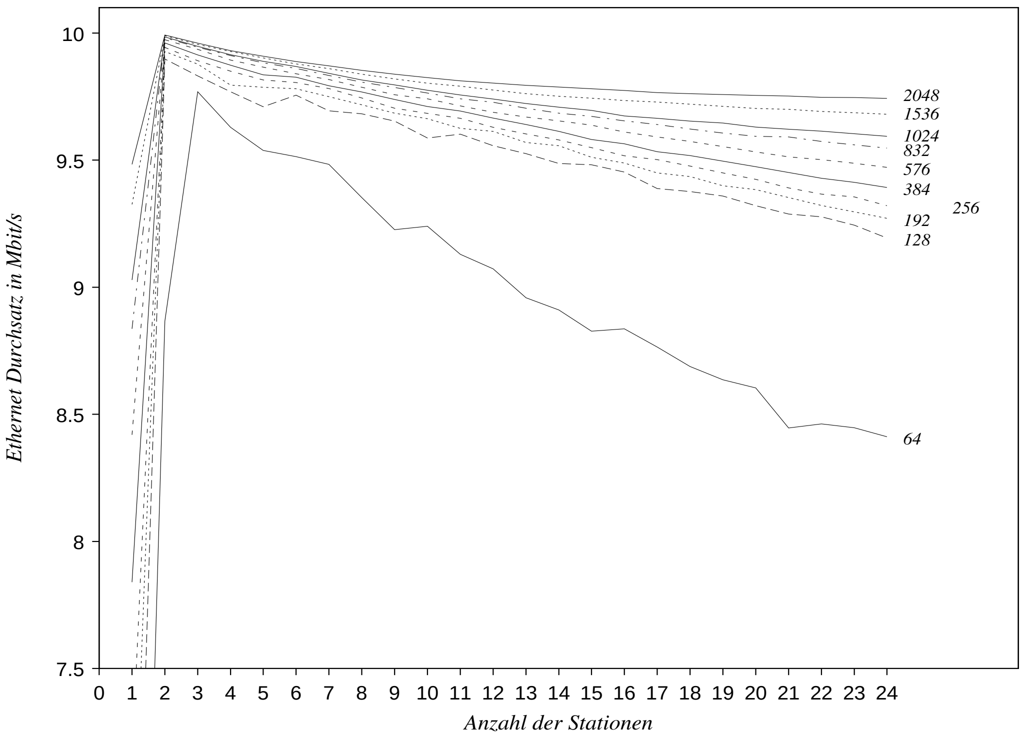


Abbildung 5.28: Durchsatz des Ethernet in Abhängigkeit von der Anzahl Stationen

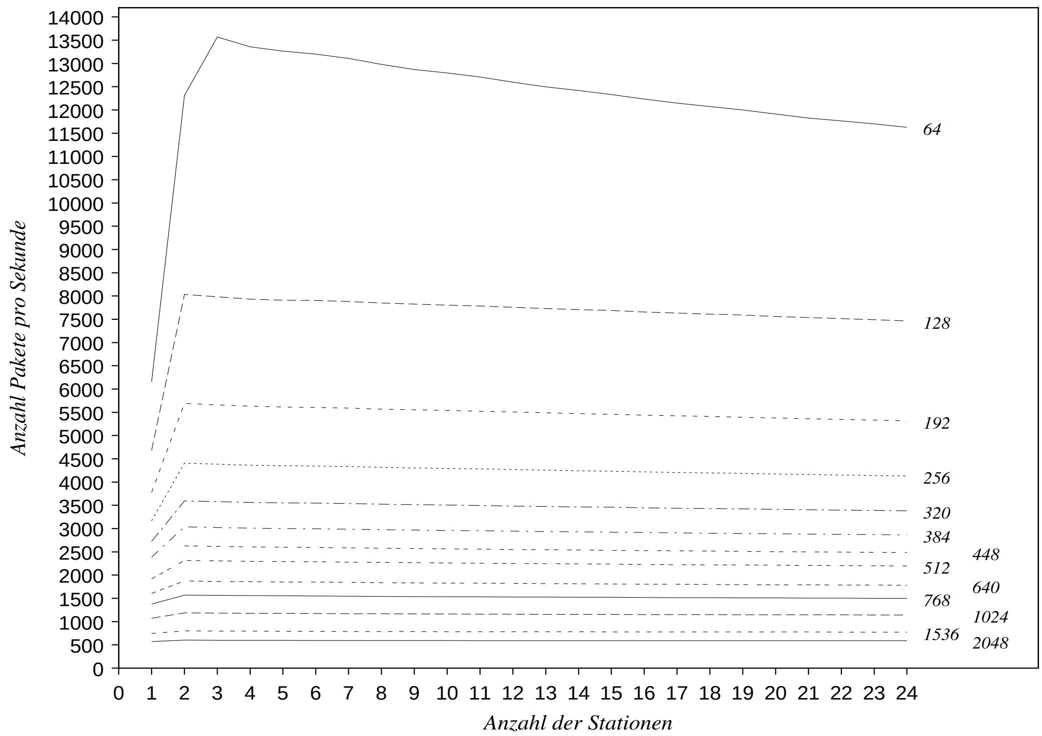


Abbildung 5.29: Anzahl Pakete pro Sekunde in Abhängigkeit von der Anzahl Stationen

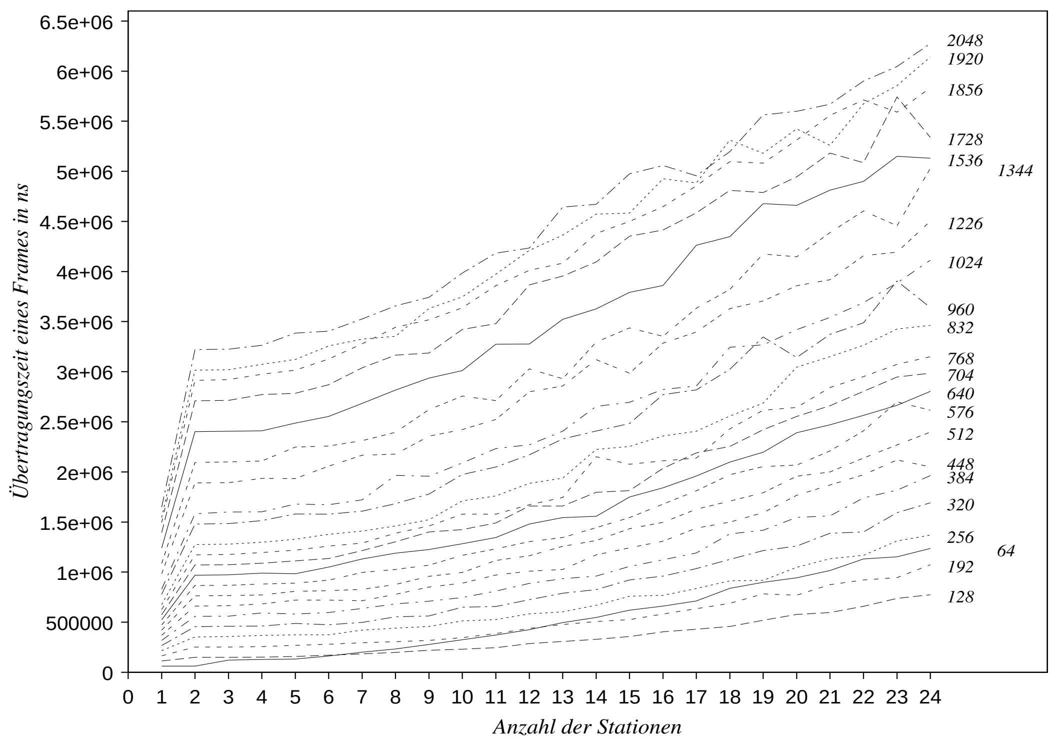


Abbildung 5.30: Übertragungszeit eines Frames in Abhängigkeit von der Anzahl Stationen

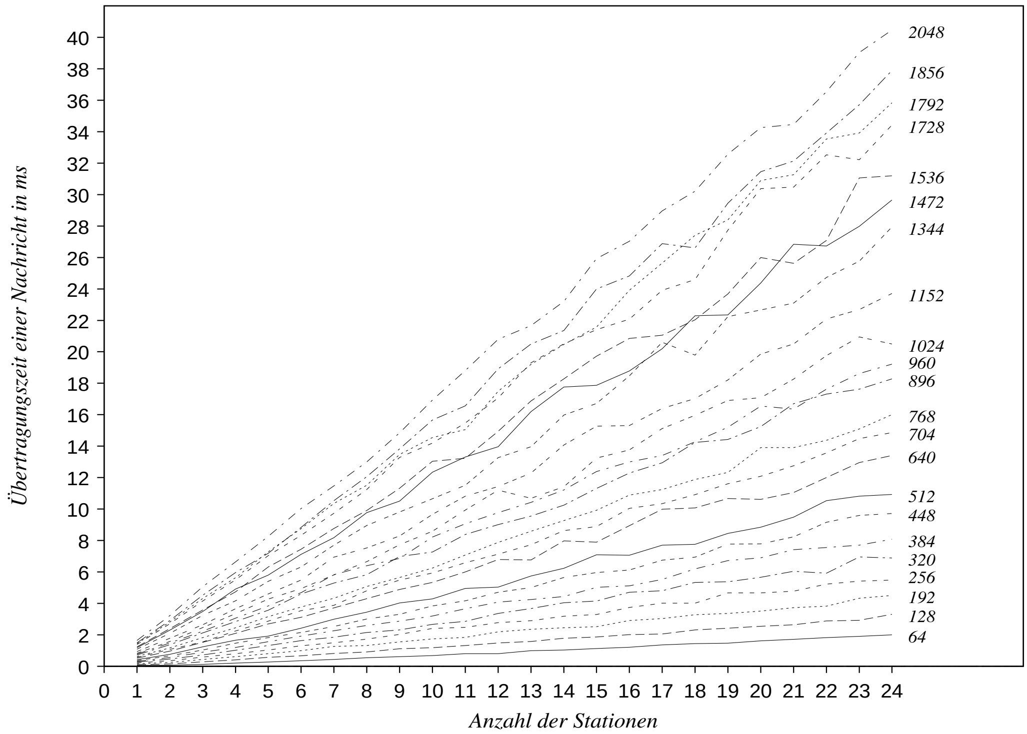


Abbildung 5.31: Übertragungszeit einer Nachricht (1 - 24 Stationen)

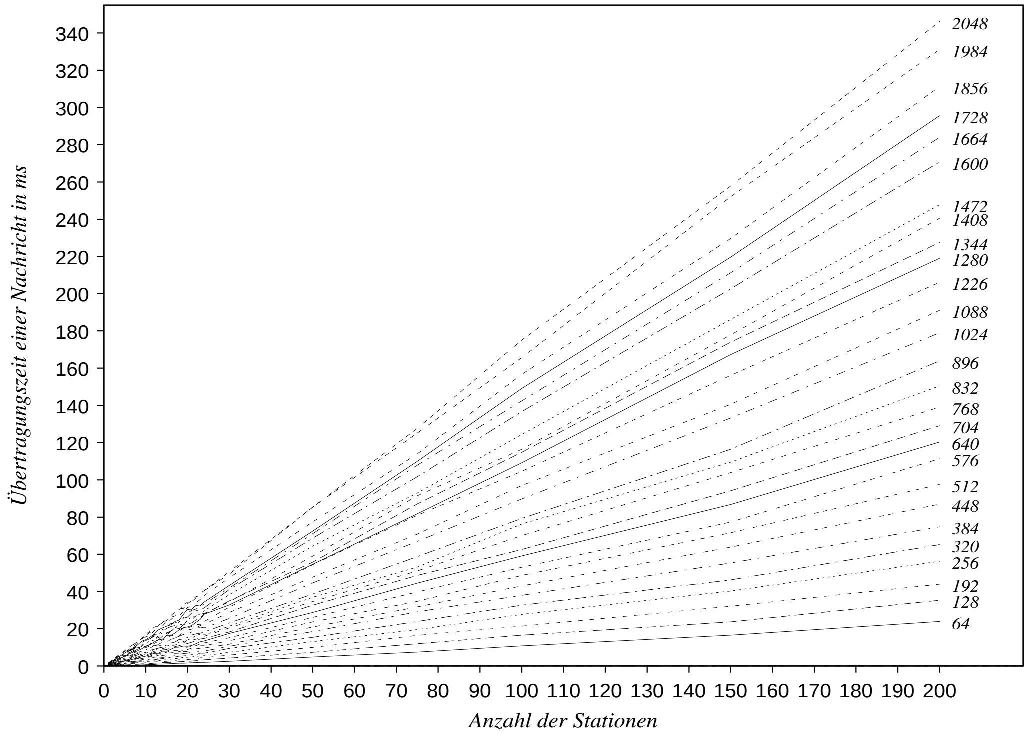


Abbildung 5.32: Übertragungszeit einer Nachricht (1 - 200 Stationen)

5.5.6 Auswertung und Beurteilung

Ein Vergleich der verschiedenen Simulatonsergebnisse mit den in [BMK88] graphisch dargestellten Meßwerten eines realen Systems ergibt, bis auf eine Ausnahme, eine ausgezeichnete Übereinstimmung. Die Ausnahme wird durch die Ergebniswerte der Simulation bei einer `Nachricht`engröße von 64 Byte und durch den ermittelten Wert für eine Station bei einer `Nachricht`engröße von 128 Byte gebildet.
Dabei ist der in der Simulation ermittelte Wert für eine Station bei einer `Nachricht`engröße von 64 Byte mit einem Durchsatz von 4.22 Mbit/s um 0.56 Mbit/s und der Wert für zwei Stationen mit 8.73 Mbit/s um 1.4 Mbit/s zu hoch. Die Durchsatzwerte für größere `Nachricht`en bei einer sendenden Station weichen mit zunehmender Größe immer weniger von den in [BMK88] angegebenen ab.
Die erhöhten Durchsatzwerte deuten daraufhin, daß im realen System die Zeit zwischen zwei Sendewünschen einer Station etwas höher als die angegebenen 100 μs ist. Dieser Unterschied wirkt sich bei kleinen `Nachricht`en stärker aus als bei größeren.
Bei einer Übertragungszeit für ein `Frame` von 60.8 μs (64 Byte `Nachricht` + 12 Byte Overhead) und einer anschließenden ,,Totzeit" von 100 μs kann alle 160.8 μs eine vollständige `Nachricht` gesendet werden. Dies entspricht 6218 `Nachricht`en pro Sekunde. Aus der in [BMK88] angegebenen Graphik ergibt sich jedoch nur ein Wert von 5200 `Nachricht`en pro Sekunde. Dieser Wert entspricht einer übertragenen `Nachricht` alle 192 μs. Die sich daraus ergebende ,,Totzeit" ist 131.2 μs.
Die erneute Simulation mit einer ,,Totzeit" von 131.2 μs ergab Werte, die identisch mit den in [BMK88] angegebenen sind.

Der Durchsatz des `Ethernet` steigt mit zunehmender `Nachricht`engröße, weil größere `Nachrichten` eine kleinere Anzahl `Nachricht`en pro Sekunde bedeuten und somit weniger Zeit für das Auflösen von Kollisionen verbraucht wird. Mit zunehmender Anzahl Stationen und konstanter `Nachricht`engröße sinkt dagegen der Durchsatz, da mehr Kollisionen pro `Nachricht` auftreten.

Auch die ermittelte mittlere Übertragungszeit eines Pakets (`Nachricht`) stimmt mit der in [BMK88] angegebenen sehr gut überein. Allerdings ergeben die simulierten Meßwerte teilweise keine Gerade, sondern sind mal größer und mal kleiner als die an dem realen System ermittelten

Werte. Mit Abweichungen im Bereich von ca. ± 2.5 Prozent liegen die Meßwerte allerdings immer noch innerhalb der Fehlergrenzen, die für die statistische Absicherung vorgegeben wurden. Diese Varianz in den Mittelwerten ist durch die hohe Varianz in den einzelnen Übertragungszeiten einer `Nachricht` begründet. Diese Zeit liegt zwischen der reinen Übertragungszeit für die `Nachricht` (keine Kollision) und der sich nach vielen Kollisionen ergebenden Zeit, die um bis zu drei Größenordnungen größer sein kann.

Die Abb. 5.32 zeigt, daß die Übertragungszeiten bei bis zu 200 Stationen nur **linear** wachsen. Dieses widerspricht der weit verbreiteten Annahme, daß die Übertragungszeit bei einer Last von mehr als $100 \cdot 1/e$ (37) Prozent überproportional anwächst [BMK88].

Die Vergleiche in diesem Abschnitt haben gezeigt, daß das entwickelte Netzmodul alle Aspekte des Verfahrens `CSMA/CD` realistisch darzustellen vermag. Damit bietet dieses Modell eine gute Alternative für die Analyse des Einflusses verschiedenster Umgebungsparameter auf Netze, die nach diesem Verfahren arbeiten. Zusätzlich bietet es die Möglichkeit, die Auswirkungen von Änderungen, wie einer höheren Übertragungsrate (s. Abschnitt 5.6) oder modifizierter Backoffalgorithmen auf die Performanz des Netzwerks, zu untersuchen.

5.6 Das CSMA/CD - Netz mit 20 Megabit pro Sekunde

In diesem Abschnitt wird ein modifiziertes `Ethernet` mit einer nominellen Übertragungsgeschwindigkeit von 20 Mbit/s simuliert.

Die notwendigen Änderungen in der Konfigurationsdatei werden im nächsten Abschnitt erläutert und dann die Simulationsergebnisse in Form von Graphiken wiedergegeben. Der letzte Abschnitt enthält eine kurze Auswertung der erhaltenen Ergebnisse und einen Vergleich mit dem Standard-`Ethernet`.

5.6.1 Konfigurationsdatei

Die Konfigurationsdatei ist bis auf wenige Zeilen mit der in Abschnitt 5.5.4 [*Konfiguration*], Seite 63 angegebenen Konfiguration identisch. Der Unterschied wird durch folgende Zeilen, die die Netzkonfiguration betreffen, gebildet:

```
bits_per_second      20e6           # Net Speed old 10e6
inter_frame_gap      192            # old 96.0
jam_time             64             # old 32.0
slot_time            1024           # old 512
min_data_len         880            # old 368
```

Zunächst wird über *bits_per_second* die nominelle Übertagungsgeschwindigkeit des Netzes auf 20 Mbit/s gesetzt.

Für die Bestimmung der restlichen Parameterwerte müssen einige Punkte berücksichtigt werden. Die meisten Werte werden in Bit-Zeiten angegeben. Bei 20Mbit/s ist ein Bit jedoch nur noch 50 *ns* lang. Damit das `CSMA/CD`-Verfahren korrekt arbeitet, müssen noch andere Parameter angepaßt werden, da sich die Signalausbreitungsgeschwindigkeit **nicht** verändert. Die Anpaßung erfolgt auf die Art und Weise, daß die entprechenden Parameter die gleichen Zeitwerte wie beim Standard-`Ethernet` nach `IEEE 802.3` erhalten.

Unter Berücksichtung der halbierten Bit-Zeit ergibt sich für *inter_frame_gap* ein Wert von 192 Bit (2 * 96 Bit), für *slot_time* ein Wert von 1024 Bit (2 * 512 Bit) und für *jam_time* ein Wert von 64 Bit (2 * 32 Bit).
Der Parameter *min_data_len* beschreibt die minimale Größe eines `Frames` auf dem Netz. Eine bestimmte Größe des `Frames` darf für die Funktionsfähigkeit des `CSMA/CD`-Verfahrens nicht unterschritten werden. `Frames`, die kleiner als diese minimale Größe sind, können durch Signallaufzeiten zu unerkannten Kollisionen führen.
Die minimale Größe eine `Frames` setzt sich zusammen aus der minimalen `Nachricht`engröße von 46 Byte und dem Overhead von 18 Byte (6 Byte Destination-, 6 Byte Source Address, 2

Byte Length, 4 Byte CRC) (siehe Abschnitt 5.2 [*ETHER-FRAME*], Seite 37). Dies entspricht bei 10 Mbit/s einer Zeit von 51.2 μs. Der Overhead ist bei dem 20 Mbit-Netz mit 18 Byte identisch, aber die Zeitdauer zum Senden ist nur 7.2 μs im Gegensatz zu 14.4 μs im 10 Mbit-Netz. Damit ergibt sich die Zeit für ein minimales `Frame` ohne Overhead zu 44 (52.1 - 7.2) μs. Diese Zeit entspricht bei 20 Mbit/s 880 Bit. Dieser Wert wird in der Konfiguration für den Parameter *min_data_len* verwendet. Somit ist sichergestellt, daß `Frame`s den Kanal für die gleiche Minimalzeit belegen wie beim Standard-`Ethernet`.

5.6.2 Simulationsergebnisse

Bei den folgenden Graphiken ist auf der X-Achse immer die Anzahl simulierter Stationen und auf der Y-Achse der Meßwert aufgetragen (Mittelwert der Meßspur).
Hinter jedem Graphen ist die dazugehörige `Nachricht`engröße, wie sie in der Konfiguration durch die Laufvariable m vorgegeben wurde, angegeben.

Die Abb. 5.33, Seite 78 zeigt den Durchsatz des modifizierten `Ethernet` mit ,,Nutzdaten". Für die Bestimmung dieses Wertes wird wieder der Overhead und der `Inter Frame Gap` zu der reinen `Nachricht`engröße hinzugezählt. Der maximale erreichbare Durchsatz ist damit 20Mbit/s. Die sich aus dem Durchsatz ergebende Anzahl Pakete (`Nachricht`en) pro Sekunde ist Abb. 5.34 zu entnehmen.
Die Abbildungen 5.35 und 5.36 (Seite 80) zeigen die Simulationsergebnisse für die Übertragungszeit eines `Frame`s bei 1–24 Stationen bzw. 1–200 Stationen. Für 1–24, 50, 75 100, 150 und 200 Stationen wurde jeweils ein Programmlauf durchgeführt. Die Stationen waren gleichmäßig auf die vier Blöcke verteilt (siehe Abschnitt 5.17 [*Simulationsmodell*], Seite 62).
Die letzten beiden Graphiken 5.37 und 5.38 geben die ermittelten Übertragungszeiten für eine `Nachricht` für eine bis zweihundert Stationen am Netz wieder. Es wurden ebenfalls für jeweils 1–24, 50, 75, 100, 150 und 200 Stationen Programmläufe durchgeführt.

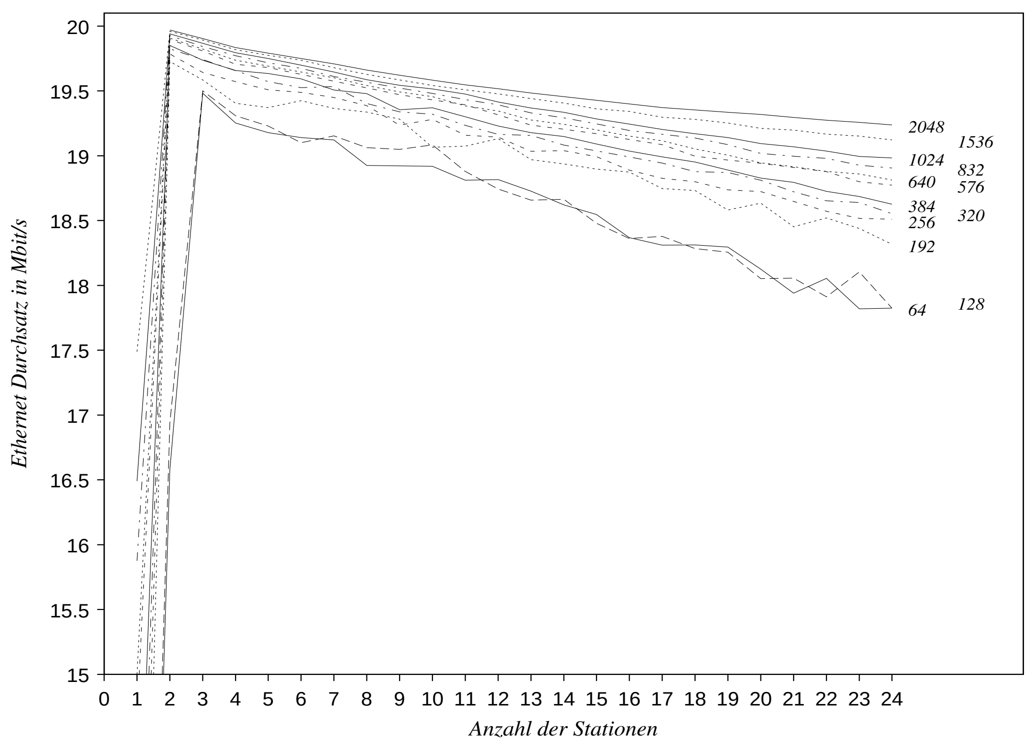


Abbildung 5.33: Durchsatz des Ethernet in Abhängigkeit von der Anzahl Stationen

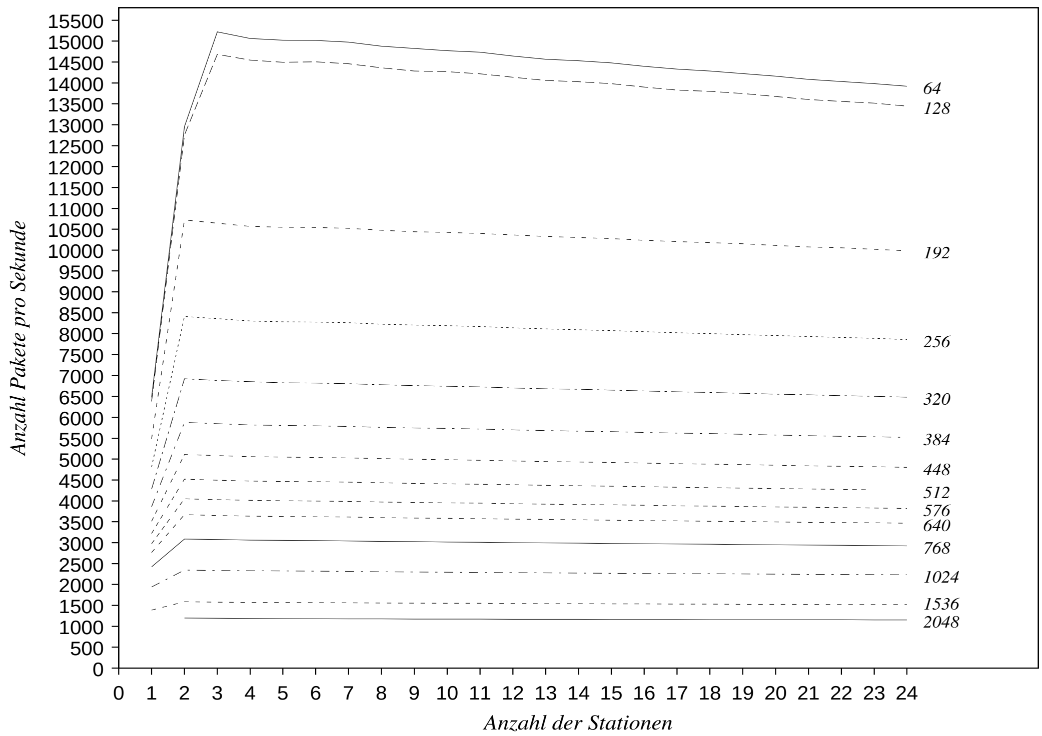


Abbildung 5.34: Anzahl Pakete pro Sekunde in Abhängigkeit von der Anzahl Stationen

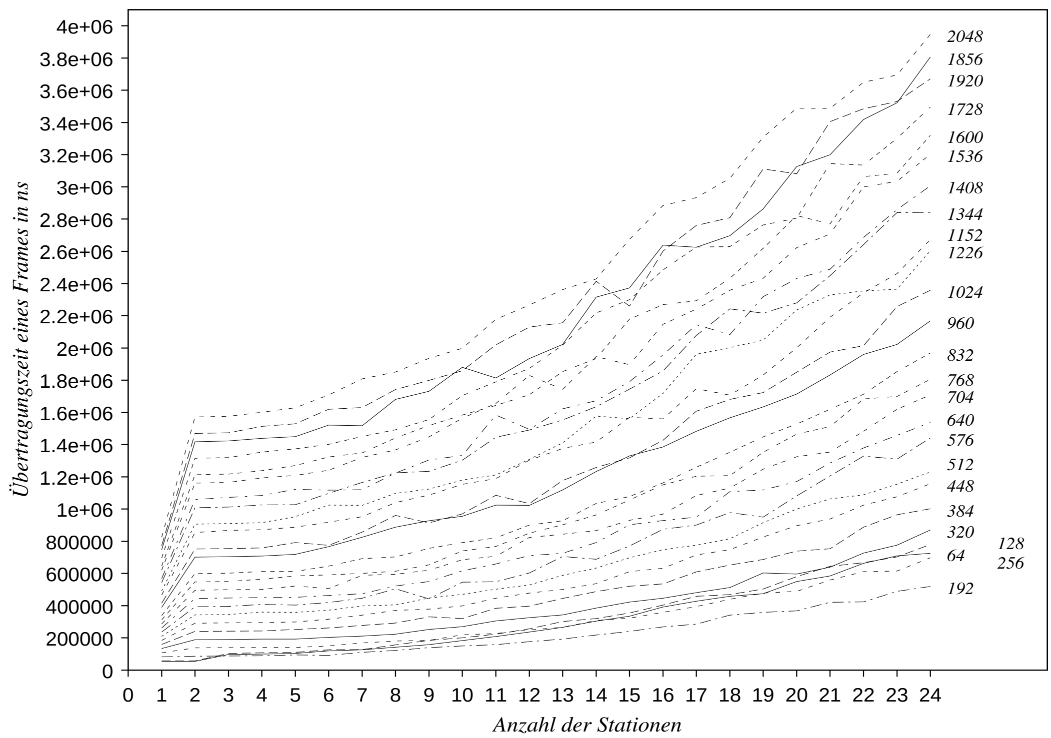


Abbildung 5.35: Übertragungszeit eines FRAMEs (1 - 24 Stationen)

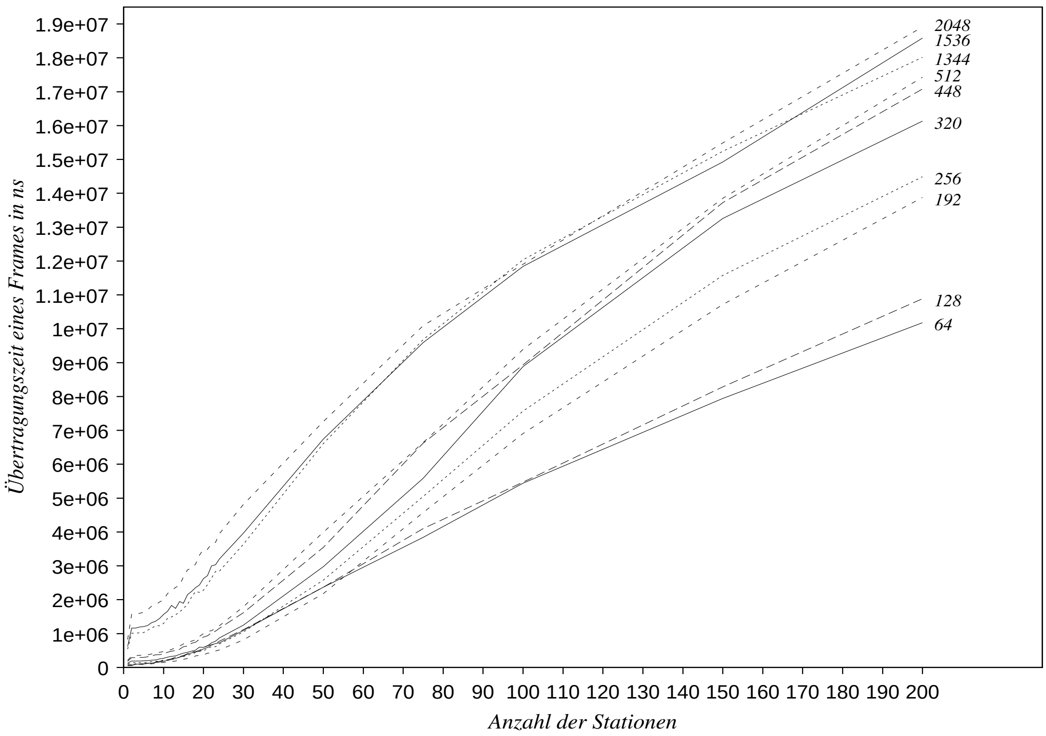


Abbildung 5.36: Übertragungszeit eines Frames (1 - 200 Stationen)

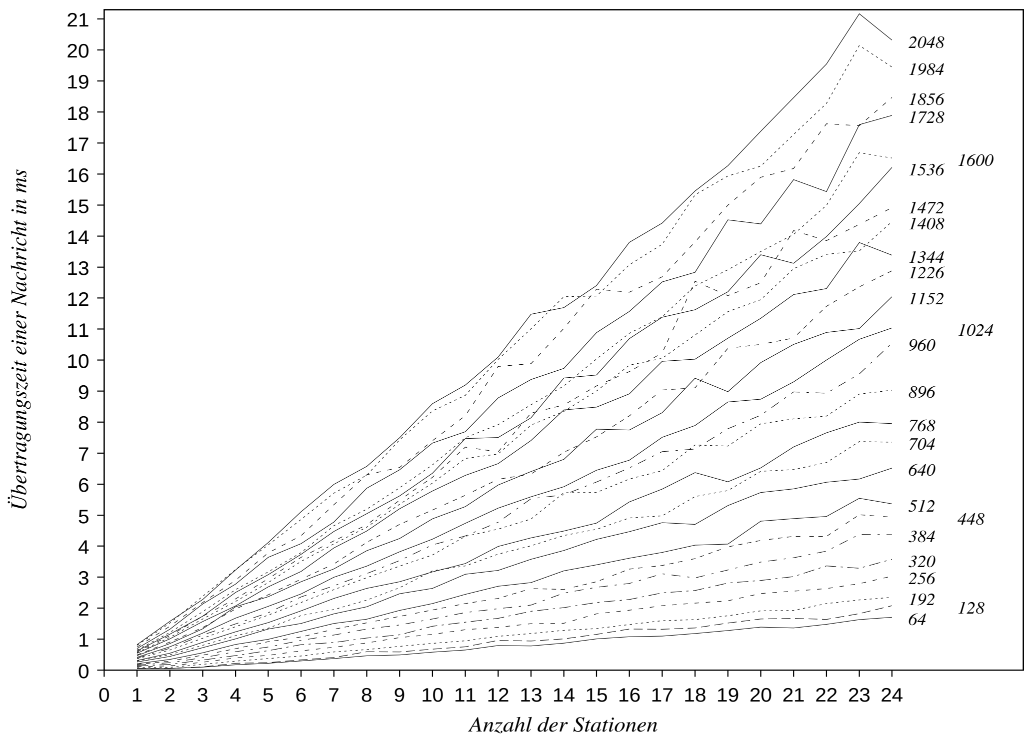


Abbildung 5.37: Übertragungszeit einer Nachricht (1 - 24 Stationen)

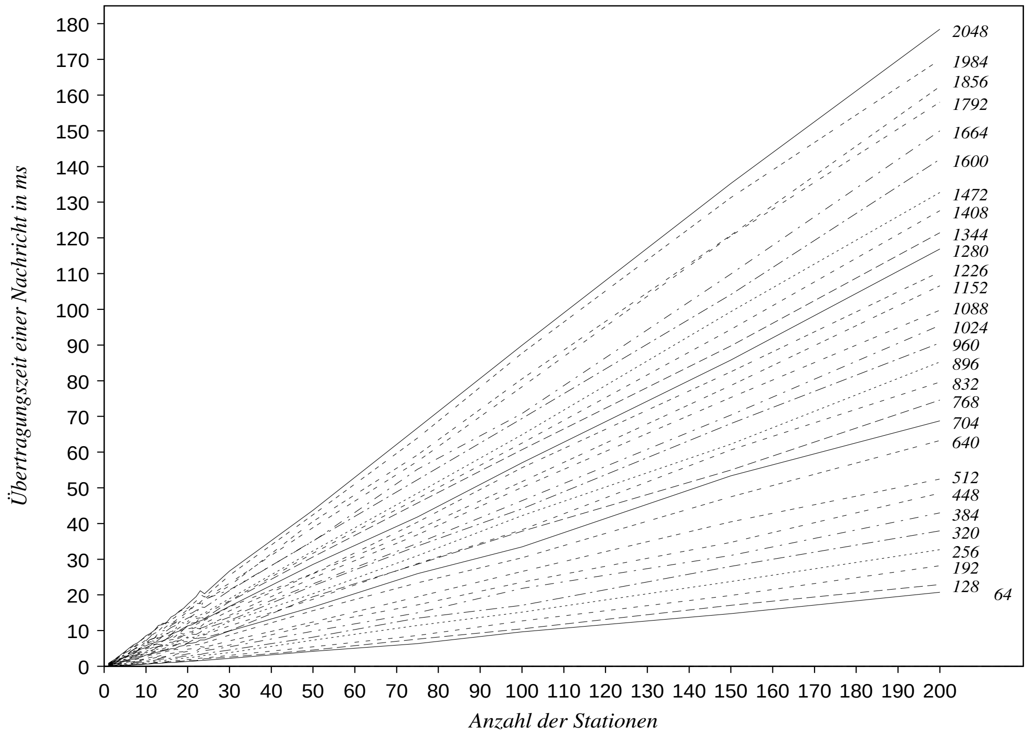


Abbildung 5.38: Übertragungszeit einer Nachricht (1 - 200 Stationen)

5.6.3 Auswertung und Folgerungen

Aus der Abb. 5.33 ist ersichtlich, daß die doppelte Bandbreite zum Großteil für die Übertragung von `Nachricht`en zur Verfügung steht.
Für kleine `Frame`s ist die Übertragungszeit des modifizierten `Ethernet`, wie aus Abb. 5.35 zu sehen, etwas größer als die des Standard-`Ethernet`. Mit zunehmender `Frame`größe verändert sich das Verhältnis jedoch auf ca. 0.45 zu 1. Das heißt, daß das modifizierte `Ethernet` nur ca. die halbe Übertragungszeit für einen identischen `Frame` benötigt. Dieses Verhalten ist zu erwarten, da größere `Frame`s weniger Kollisionen bewirken und damit auch weniger Zeit für die Auflösung derselben benötigt wird. Damit steht mehr Bandbreite für ,,Nutzdaten" zur Verfügung.
Das Verhalten der Übertragungszeiten von `Nachricht`en ist mit dem gerade beschriebenen Verhalten identisch. Bei kleinen `Nachricht`en wird bei beiden Verfahren ungefähr die gleiche Übertragungszeit benötigt. Erst bei größeren `Nachricht`en wirkt sich die im modifizierten `Ethernet` zur Verfügung stehende doppelte Kanalbandbreite positiv aus. Bei einer `Nachricht`engröße von 2048 Byte und 200 sendenden Stationen wird z.B. 47 Prozent weniger Übertragungszeit pro `Nachricht` benötigt.

Abschließend läßt sich feststellen, daß das `CSMA/CD`-Verfahren eine vergrößerte Kanalbandbreite unter großer Last nur bei großen `Nachricht`en ausnutzt. In einem Netz mit sehr vielen kurzen `Nachricht`en und nur relativ wenigen großen, wie es bei den meisten lokalen Netzen der Fall ist, ist demnach eine Verdoppelung der nominellen Übertragungsgeschwindigkeit unter Beibehaltung des Zugriffsverfahrens `CSMA/CD` sicher keine Möglichkeit, um unter hoher Last niedrigere Antwortzeiten und einen größeren Durchsatz zu erhalten.

Kapitel 6

Das Schichtmodell *l2simple*

6.1 Einleitung

In diesem Kapitel wird ein Schichtmodell entwickelt, das einen sehr einfachen Netzwerk-Controller nachbildet. Die modellierten Aufgaben eines solchen Controllers sind in der simuLan-Schicht *2* anzusiedeln, die der ISO/OSI-Schicht *2b* (`LLC`) entspricht (siehe Abschnitt 3.5 [*Normen*], Seite 9 und Abschnitt 4.2.5 [*Das Schicht-Modul*], Seite 16).

Im nächsten Abschnitt wird das Modell beschrieben und dann auf die Implementation eingegangen. Dort werden die Modellparameter, die Schnittstellen und die einzelnen Funktionen beschrieben.

6.2 Modellbeschreibung

Das Schichtmodell `l2simple` modelliert einen sehr einfachen Netzwerk-Controller. Die Hauptaufgabe eines solchen Controllers ist es, sicherzustellen, daß zu einem Zeitpunkt von einer Station nur genau *ein* `Paket` gesendet wird. Falls weitere Sendewünsche eintreffen, während gerade ein `Paket` übertragen wird, müssen diese in eine Warteschlange eingereiht werden, um sie dann nach Ende der Übertragung der Reihe nach zu senden.
Nach dem Senden bzw. Empfangen eines `Paket`s benötigt der Controller Zeit, um die anfallenden Verwaltungsaufgaben zu erledigen. Diese bestehen darin, ein neues `Paket` zu erzeugen bzw. ein empfangenes `Paket` der übergeordneten Protokollschicht zu übergeben.

Das implementierte Schichtmodell `l2simple` modelliert diese beiden Aspekte eines Controllers.

6.3 Realisierung

In den folgenden Funktionsbeschreibungen wird das in Kapitel 4 vermittelte Grundlagenwissen über simuLan vorausgesetzt und nur noch die für die Modellierungsmethode spezifischen Sachverhalte beschrieben.

Alle im Schichtmodul `l2simple` verwendeten Warteschlangen und ihre Funktionalitäten werden von simuLan bereitgestellt. Eine kurze Einführung in dieses Thema ist in Abschnitt 4.2.8 [*Warteschlangen*], Seite 19 zu finden. In [Pet91] Anhang A [Programmierhandbuch] wird detailliert auf diese Warteschlangen und ihre Möglichkeiten eingegangen.

Ein `Paket` an eine Funktion „übergeben" bedeutet im folgenden, einen Zeiger auf die dazugehörige Datenstruktur `EVENT` als Parameter an diese Funktion zu übergeben.
Die Abb. 6.1 zeigt die zu einem `Paket` gehörenden Datenstrukturen.

In Anhang D ist das vollständige Programmlisting zu finden.

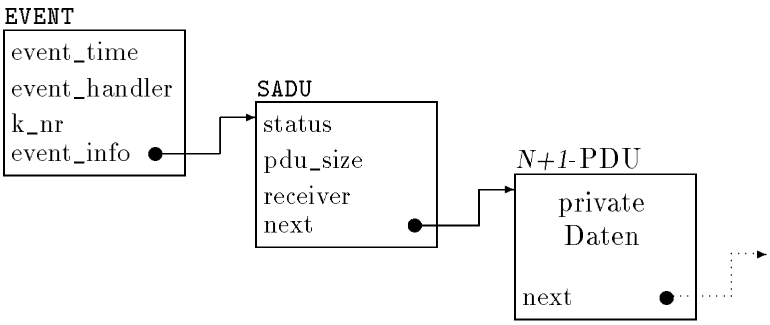


Abbildung 6.1: Die Datenstrukturen eines `Paket`s

6.3.1 Modellparameter

Dieses Modell besitzt zwei Parameter, deren Werte über die Konfiguration vorgegeben werden können.

Der erste Parameter ist *finish_time_needed*. Er beschreibt die von dem Controller für Verwaltungsaufgaben benötigte Zeit nach dem Senden bzw. Empfangen eines `Paket`s. Während dieser Zeit kann kein `Paket` gesendet und empfangen werden.

Der zweite Parameter ist *l2simple_leave_probept*. Über diesen Parameter können für den dazugehörigen Meßpunkt Meßfunktionen angegeben werden (siehe Abschnitt 6.3.7 [*Der Meßpunkt l2simple_leave_probept*], Seite 87). Bisher sind allerdings keine Meßfunktionen implementiert.

6.3.2 Schnittstellen

Aus Abb. 6.2 sind die Schnittstellen zu den Protokollschichten *1* und *3* ersichtlich. Die Schreibweise **(*...)()** besagt, daß die entsprechende Schnittstellenfunktion der Nachbarschicht über einen Funktionszeiger aufgerufen wird.

Die Protokollschicht *3* im Sender erzeugt ein `Paket` und übergibt dieses zum Senden an die Funktion *l2simple_sap()*, die der in Abschnitt 4.2.5 beschriebenen Prozedur `XLayer_sap()` entspricht.
Ein `Paket` besteht dabei zumindest aus dem `EVENT` und einer daran angehängten `SADU` (s. Abschnitt 4.2.2). Für den Informationsaustausch zwischen höheren Schichten ($N \geq 3$) können an dem Zeiger `SADU`→*next* Datenstrukturen angehängt sein (s. Abb. 6.1).
Diese Funktion überprüft, ob nicht schon ein `Paket` übertragen wird. Ist dieses der Fall, wird der Sendewunsch in eine Warteschlange gestellt. Falls kein `Paket` übertragen wird, wird geprüft, ob der Kontoller bereit ist. Wenn seit der letzten Aktion, Senden bzw. Empfangen eines `Paket`s, genug Zeit für die Verwaltungsaufgaben verstrichen ist, wird das `Paket` sofort gesendet und sonst noch eine entsprechende Zeit gewartet, bevor mit der Übertragung begonnen wird.

Über die Funktion *l2simple_pdu_received()* wird die Ankunft eines `Paket`s in der Protokollschicht *2* des Empfängers simuliert. Diese Funktion wird vom verwendeten Netzmodul am Ende einer Übertragung aufgerufen.

Die erfolgreiche Übertragung eines `Paket`s wird der Protokollschicht *2* des Senders, über den Aufruf der Funktion *l2simple_pdu_sended()* mitgeteilt. Diese Funktion übermittelt zunächst der übergeordneten Schicht *3* eine Sendebestätigung und sendet anschließend das nächste `Paket` dieser Station.

6.3.3 l2simple_sap()

Diese Funktion erhält als Parameter ein `Paket` übergeben und simuliert die Ankunft eines zu sendenden `Paket`s in der simuLan-Schicht *2*.

Die Abb. 6.3 zeigt den Kontollfluß in dieser Funktion.

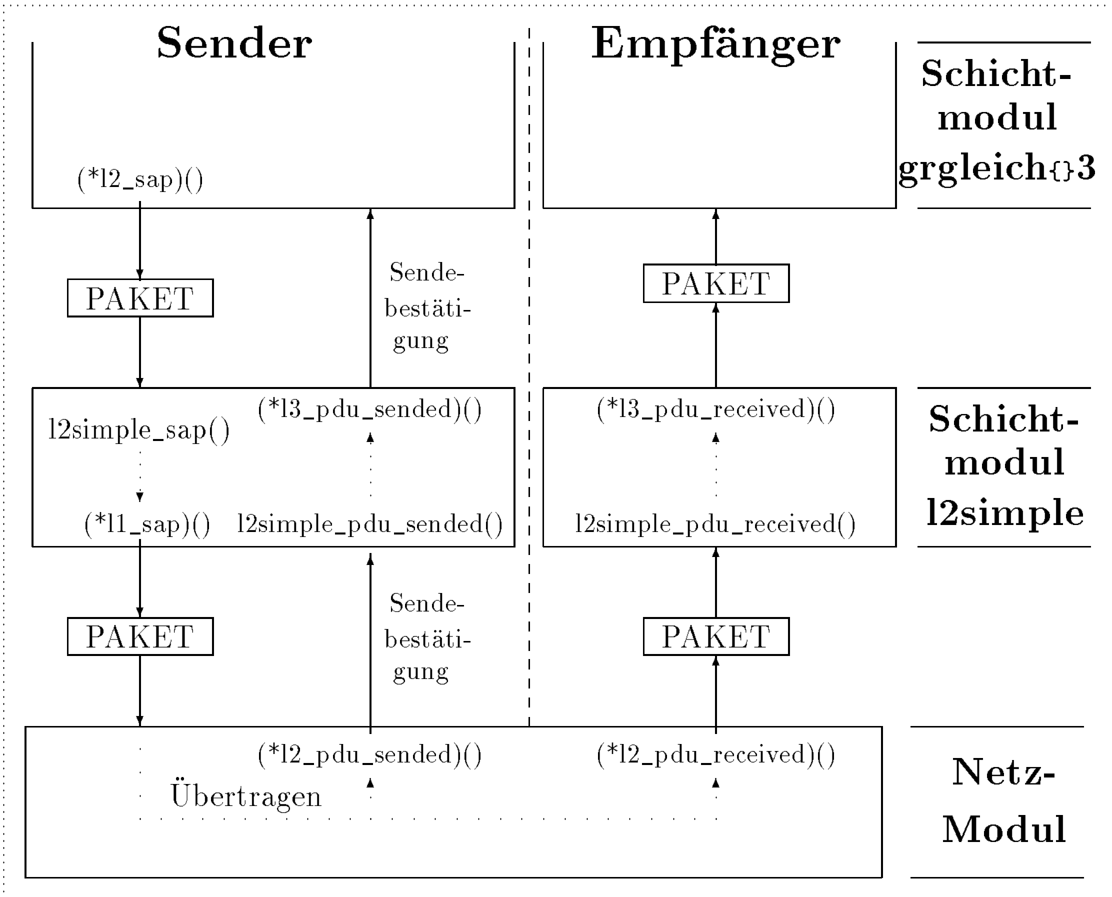


Abbildung 6.2: Schnittstellen und Datenfluß im Schichtmodul-`l2simple`

l2simple_sap() —

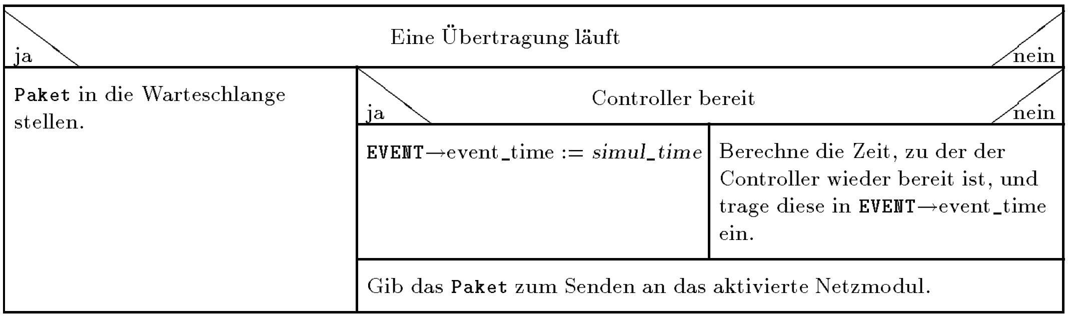


Abbildung 6.3: Kontrollfluß in der Funktion *l2simple_sap()*

Zunächst wird geprüft, ob gerade ein `Paket` gesendet wird. Im Fall einer laufenden Übertragung wird das `Paket` in die Warteschlange für noch zu sendende `Pakete` dieser Station gestellt. Die Bedienstrategie der Warteschlange ist *First In First Out*. Ein Überschreiten der maximalen Warteschlangenlänge führt zum Abbruch des aktuellen Simulationslaufes (siehe Abschnitt 4.2.8 [*Warteschlangen*], Seite 19).

Falls zur Zeit keine Übertragung läuft, wird geprüft, ob der Controller schon bereit ist, ein neues `Paket` zu senden. Dieser Zeitpunkt wird durch T_{Ready} festgelegt. T_{Ready} ist die Summe aus dem Zeitpunkt der letzten Aktivität des Controllers und der Zeit T_{work}, die für simulierte Verwaltungsaufgaben benötigt wird. T_{work} wird über den Konfigurationsparameter *finish_time_needed*

vorgegeben.
Wenn *simul_time* kleiner als T_{ready} ist, ergibt sich `EVENT`→*event_time* zu T_{ready} und sonst zu *simul_time*. Falls für den Controller noch ein zusätzlicher Zeitverbrauch simuliert werden soll, muß der berechnete Zeitpunkt entsprechend nach hinten verschoben werden.
Anschließend wird das `Paket` an das Netzmodul zum Senden übergeben. Dieses muß sicherstellen, daß die eigentliche Übertragung frühstens zu dem in *event_time* eingetragenen Zeitpunkt beginnt.

6.3.4 l2simple_pdu_sended()

Diese Funktion erhält als Parameter ein `Paket` übergeben und wird vom aktivierten Netzmodul, nachdem dieses das `Paket` übertragen hat, aufgerufen.

Als erstes wird die simuLan-Schicht *3* von dem Ende der Übertragung in Kenntnis gesetzt und die Anzahl erfolgreich gesendeter `Paket` erhöht. Danach wird das erste `Paket` aus der Sendewarteschlange entnommen, um dieses zu senden. Der Sendezeitpunkt `EVENT`→*event_time* ergibt sich aus der Summe von *simul_time* und der Zeit *finish_time_needed*, die über die Konfiguration vorgegeben wird (siehe Abschnitt 6.3.1 [*Modellparameter*], Seite 84). Falls für den Controller noch ein zusätzlicher Zeitverbrauch simuliert werden soll, muß der berechnete Zeitpunkt entsprechend nach hinten verschoben werden.
Anschließend wird das `Paket` an das Netzmodul zum Senden übergeben. Dieses muß sicherstellen, daß die eigentliche Übertragung frühstens zu dem in *event_time* eingetragenen Zeitpunkt beginnt.

6.3.5 l2simple_pdu_received()

Diese Funktion wird von der simuLan-Schicht *1* auf der Empfängerseite aufgerufen und entspricht der in Abschnitt 4.2.5 [*Das Schicht-Modul*], Seite 16 beschriebenen Funktion *XLayer_pdu_received()*. Sie erhält als Parameter das `Paket` übergeben.

Falls der Controller noch nicht bereit war, das `Paket` zu empfangen, wird dieses nicht beachtet. Andernfalls wird es unverändert an die simuLan-Schicht *3* weitergegeben und die Anzahl empfangener `Paket`e erhöht.
Anschließend wird der Zeitpunkt berechnet, an dem der Controller wieder bereit ist, ein neues `Paket` zu senden oder zu empfangen (siehe Abschnitt 6.3.3 [*l2simple_sap()*], Seite 84).

6.3.6 Rahmenfunktionen

Die Rahmenfunktionen des Schichtmoduls `l2simple` sind *l2simple_rc()* und *l2simple_print_info()*.

Die Prozedur *l2simple_rc()* initialisiert, entsprechend dem übergebenen Parameter *count*, *count* **identische** `Schichtmodule` vom Typ `l2simple`.
Dabei fallen folgende Aufgaben an:

- Falls über die Kommandozeile angegeben, werden alle variablen Parameter eines Schichtmoduls `l2simple` mit einer Erklärung ausgegeben.
- Die Datenstruktur *l2simple_rc_info*, in der die Werte aus der Konfiguration zwischengespeichert werden, wird mit den Defaultwerten vorbelegt.
- Die Funktion *rcreadkeywords()* wird mit der Tabelle der definierten Schlüsselwörter für das `l2simple`-Schichtmodul aufgerufen. Dabei werden gegebenenfalls die Defaultwerte ausgegeben.
- Die eingelesenen Parameter werden auf Plausibilität geprüft.
- Die Datenstruktur `L2SIMPLE_INFO` wird entsprechend dem Inhalt von *l2simple_rc_info* initialisiert.

Die Funktion *l2simple_print_info()* gibt die in der Sendewarteschlange gesammelten Daten und die Anzahl gesendeter und empfangener `Paket`e aus.

6.3.7 Der Meßpunkt l2simple_leave_probept

Der Meßpunkt *l2simple_leave_probept* ist in der Funktion *l2simple_pdu_sended()* eingebaut. Alle an diesem Meßpunkt aktivierten Funktionen werden jeweils nach Ende der Übertragung eines `Paket`s aufgerufen.
Bisher sind keine Meßfunktionen für diesen Meßpunkt implementiert worden.

Genauere Informationen zu Meßpunkten sind in Abschnitt 4.2.7, Seite 19 und besonders in [Pet91] zu finden.

6.3.8 Verifikation

Die Verfikation dieses Schichtmoduls wurde mit der von simuLan bereitgestellten Möglichkeit des Verfolgens von einzelnen Ereignissen in den verschiedenen Protokollschichten durchgeführt.

Kapitel 7

Das Knotenmodell *K1*

7.1 Einführung

Dieses Kapitel beschreibt ein Rechnermodell, mit dem das Verhalten verschiedener Rechnertypen nachgebildet werden kann. Bei der Entwicklung wurde Wert darauf gelegt, die meisten Systemparameter über die Konfiguration festlegen zu können. Zu diesen Parametern gehören z.B. die Anzahl der Prozessoren (*CPU*, *Central Processing Unit*), die Anzahl der Platten und die dazugehörigen Verteilungsfunktionen für Rechenzeit und Plattenparameter.
Das entwickelte Modell bildet allgemein die wichtigsten Funktionalitäten eines Rechnersystems nach. Es ist als Basismodell gedacht, auf dem aufbauend Modelle entwickelt werden können, die spezifische Sachverhalte eines realen Systems, z.B. Plattenwartezeiten in einem ‚Fileserver', nachbilden.

Im nächsten Abschnitt wird das Modell erläutert und in Abschnitt 7.3 die Implementierung des dazugehörigen Moduls näher beschrieben. In dem sich anschließenden Abschnitt wird die Verifikation des `K1`-Knotenmoduls durchgefürt. Der letzte Abschnitt gibt dann noch einige Beispiele für mögliche Erweiterungen dieses Modells an.

7.2 Modellbeschreibung

In Abb. 7.1, Seite 89 ist die mit dem `K1`-Knotenmodell modellierbare Menge von Systemen dargestellt. Die Ereignisse, die zu Zustandsänderungen des Systems führen, sind in der Abbildung an den entsprechenden Stellen angegeben.

Ein Benutzer sitzt an einem Terminal seines Rechners und gibt Befehle ein. Jeder Befehl führt intern zur Erzeugung eines *JOB*s (siehe Abschnitt D [*Begriffsdefinitionen*], Seite 122), der durch den Rechner abgearbeitet wird. In dem Modell werden alle `JOB`s über **ein** Terminal erzeugt. Dieses stellt keine Einschränkung dar, da eine beliebige Anzahl Terminals durch eine entsprechende Verteilungsfunktion für die Zeiten zwischen zwei Befehlen nachgebildet werden kann. Bei der Erzeugung des `JOB`s wird festgelegt, wieviel Speicher und `CPU`-Zeit er insgesamt für die Abarbeitung des Befehls benötigen wird.

Nachdem der `JOB` erzeugt wurde, fordert er vom System den gesamten benötigten Speicher auf einmal an. Falls das System nicht genügend Freispeicher hat, wird der `JOB` in die Speicherwarteschlange eingehängt. Diese arbeitet nach dem Prinzip *First In First Out* (*FIFO*).
Wenn die Speicheranforderung erfüllt werden konnte, wird als nächstes eine `CPU` angefordert. Falls gerade alle `CPU`s belegt sind, wird der `JOB` in die zugehörige Warteschlange eingefügt. Es gibt für alle verfügbaren `CPU`s nur eine *gemeinsame* Warteschlange (`FIFO`), da die `CPU`s identisch sind. Die Vergabe einer `CPU` erfolgt nach dem *Round Robin*-Verfahren. Jeder `JOB` erhält die `CPU` nur für eine bestimmte maximale Zeit zugeteilt. Nach Ablauf dieser Zeit muß die `CPU` freigegeben werden.
Nachdem eine gewisse Zeit ‚gerechnet' wurde, fordert der `JOB` entweder eine Platte an oder sendet eine `Nachricht` oder fordert erneut eine `CPU` an.

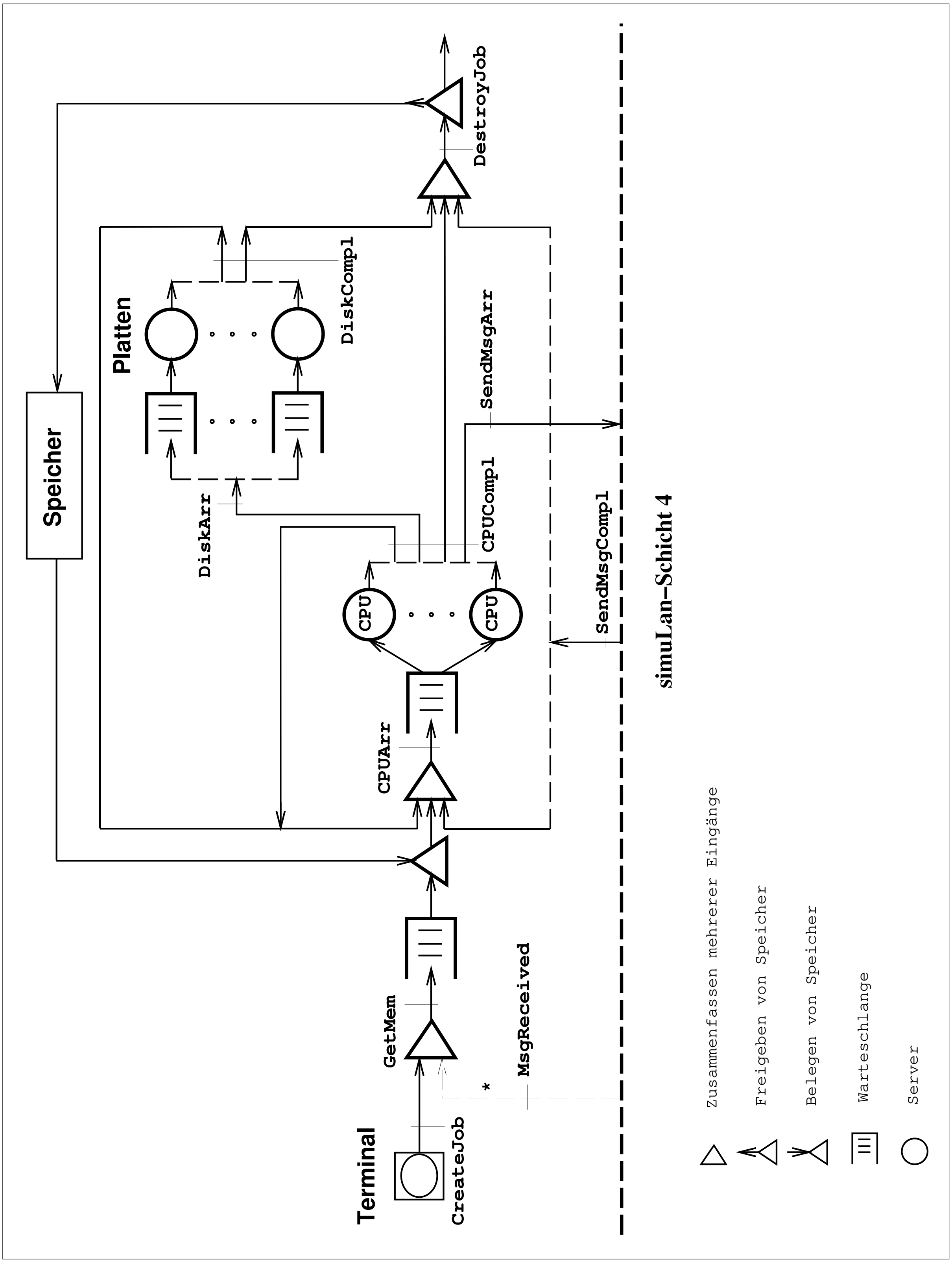


Abbildung 7.1: Das Knotenmodell K1

Die Anforderung der `CPU` wird wie oben beschrieben behandelt.
Falls der `JOB` auf eine Platte zugreifen möchte, wird zunächst die entsprechende Plattennummer und die für den Plattenzugriff benötigte Zeit ermittelt. In dem Modell ist eine beliebige Anzahl (max. 512) Platten mit unterschiedlichen Verteilungsfunktionen für die Plattenzugriffszeiten konfigurierbar. Für jede Platte existiert eine eigene Warteschlange (`FIFO`). Wenn gerade ein anderer `JOB` die Platte belegt, wird der aktuelle `JOB` in die dazugehörige Warteschlange

eingehängt.

Nach Beendigung eines Plattenzugriffes fordert der JOB, falls er noch Rechenzeit benötigt, erneut eine CPU an. Ist dieses nicht der Fall, wird der belegte Speicher freigegeben und der JOB vernichtet.

In dem Fall, daß der JOB eine Nachricht senden will, wird die Größe der Nachricht und der Empfänger bestimmt. Anschließend wird die Nachricht zum Senden an die simuLan-Schicht 4 gegeben und der JOB so lange angehalten, bis die Nachricht übertragen werden konnte. Falls der JOB anschließend noch Rechenzeit benötigt, fordert er wieder eine CPU an. Andernfalls wird der belegte Speicher freigegeben und der JOB beendet.

Das Freigeben von Speicher aktiviert einen JOB aus der Speicherwarteschlange, falls nun dessen Speicheranforderung vollständig erfüllt werden kann.
Das Freigeben einer CPU oder einer Platte führt zur Aktivierung des ersten JOBs in der dazugehörigen Warteschlange.

Die gesendeten Nachrichten führen im Empfänger in der bisherigen Implementation zu keinen weiteren Aktivitäten. Eine Erweiterung des Modells, welche die Erzeugung von JOBs im Empfänger (z.B. Fileserver) ermöglicht, ist jedoch einfach zu implementieren und in Abb. 7.1 durch die mit $\star$ gekennzeichnete, gestrichelte Linie angedeutet.

Die Ereignisse, welche die beschriebenen Aktionen auslösen, sind in Tabelle 7.1 mit den dazugehörigen Funktionen, die diese Aktionen simulieren, aufgeführt. Die Implementierung dieser Funktionen wird im nächsten Abschnitt beschrieben.

Ereignis	,C'-Funktion	Aktionen
CreateJob	k1_create_job()	Erzeugen eines JOBs
GetMem	k1_get_mem()	Stellen der Speicheranforderung
CPUArr	k1_cpu_cust_arr()	Anforderung einer CPU
CPUCompl	k1_cpu_serv_compl()	Freigeben einer CPU
DiskArr	k1_disk_cust_arr()	Anforderung einer Platte
DiskCompl	k1_disk_serv_compl()	Freigeben einer Platte
SendMsgArr	k1_appl_send_arr()	Senden einer Nachricht
SendMsgCompl	k1_message_sended()	Nachricht komplett gesendet
MsgReceived	k1_message_rec()	Nachricht empfangen
DestroyJob	k1_destroy_job()	Speicher freigeben und den JOB beenden.

Tabelle 7.1: Ereignisse des Knotenmoduls K1

7.3 Realisierung

In den folgenden Funktionsbeschreibungen wird das in Kapitel 4 vermittelte Grundlagenwissen über simuLan vorausgesetzt und nur noch die für die Modellierung spezifischen Sachverhalte beschrieben.

Alle in Abb. 7.1, Seite 90 aufgeführten Funktionen sind als Ereignisfunktionen programmiert. Das heißt, daß jede Funktion über ein bestimmtes Ereignis aktiviert werden kann.
Um die Programmlaufzeiten der Simulationen zu verkürzen, werden die Funktionen *k1_get_mem()* und *k1_destroy_job()* direkt augerufen und nicht über den Ereignismechanismus aktiviert. Daraus folgt, daß die Aktion, einen JOB zu erzeugen bzw. zu beenden, keine Zeit verbrauchen kann.

Einen JOB an eine Funktion „übergeben" bedeutet im folgenden, einen Zeiger auf die dazugehörige Datenstruktur EVENT als Parameter an diese Funktion zu übergeben (siehe Abschnitt 7.3 [*k1_create_job()*], Seite 92).

Alle im K1-Knotenmodul verwendeten Warteschlangen und ihre Funktionalitäten werden von simuLan bereitgestellt. Eine kurze Einführung in dieses Thema ist in Abschnitt 4.2.8 [*Warteschlangen*], Seite 19 zu finden. In [Pet91] Anhang A [Programmierhandbuch] wird detailliert auf diese Warteschlangen und ihre Möglichkeiten eingegangen.

In Anhang D ist das vollständige Programmlisting zu finden.

7.3.1 Schnittstellen

Aus der Abb. 7.2 sind die Schnittstellen zur Protokollschicht *4* sowie der Datenfluß innerhalb des `K1-Moduls` ersichtlich. Die Schreibweise (*...)() besagt, daß die entsprechende Schnittstellenfunktion der Nachbarschicht über einen Funktionszeiger aufgerufen wird.

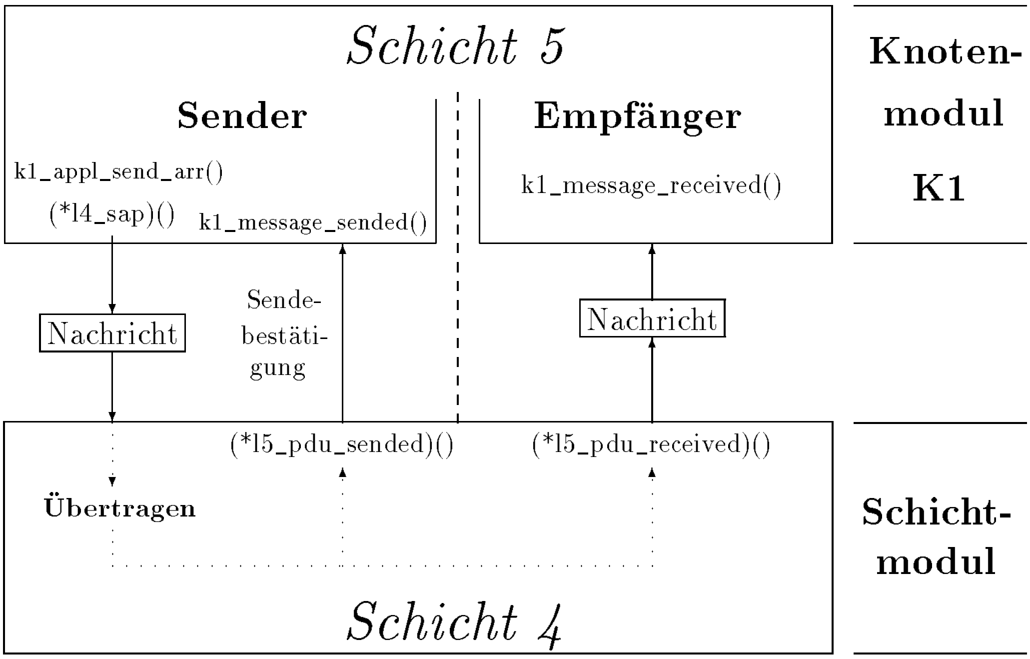


Abbildung 7.2: Schnittstellen und Datenfluß im K1-Knotenmodul

Die durch das `K1`-Knotenmodul modellierte simuLan-Schicht *5* erzeugt in *k1_appl_send_arr()* eine `Nachricht` und sendet diese. Dazu wird die Zugangsfunktion l*4*_sap() zur Schicht *4* mit dem die `Nachricht` beschreibenden `EVENT` als Parameter aufgerufen (siehe Abschnitt 4.2.2 [*EVENT*], Seite 15 und Abschnitt 7.6 [*k1_appl_send_arr()*], Seite 95).
Nach dem Senden ruft die Protokollschicht *4* die Senderfunktion *k1_message_sended()* (über l5_pdu_sended()) auf, um den Sender von dem Ende der Übertragung in Kenntnis zu setzen und die Reaktivierung des dazugehörigen `JOB`s anzustoßen.
Die Prozedur *k1_message_received()* (über l5_pdu_received()) wird von der Schicht *4* genau dann aufgerufen, wenn sie eine empfangene `Nachricht` an den Empfänger abliefern will.

7.3.2 Modellparameter

In der Tabelle 7.2 sind die Parameter des Knotenmodells `K1` sowie jeweils eine kurze Erläuterung dazu angegeben. Die aktuellen Werte für einen Simulationslauf werden über die Konfigurationsdatei (siehe Abschnitt 4.2.9 [*Konfiguration*], Seite 20) festgelegt. Diese Parameter werden in einer Datenstruktur vom Typ *K1_INFO* gespeichert.
Die Parameter und ihre Einheiten gibt simuLan beim Aufruf mit der Option „**-x**" aus.

7.3.3 k1_create_job()

Diese Funktion wird über das Ereignis `CreateJob` aktiviert und erzeugt einen neuen `JOB`.

Als erstes wird der Zeitpunkt für das nächste Ereignis `CreateJob` bestimmt und für diesen Zeitpunkt ein Ereignis `CreateJob` eingeplant.
Anschließend wird ein vollständiger `JOB` neu erzeugt. Die Abb. 7.3, Seite 92 stellt die Datenstrukturen, die einen `JOB` im `K1`-Knotenmodell beschreiben, dar.

Parameter	Erläuterung
term_ank_distr	Verteilungsfunktion für die Zeit zwischen der Erzeugung zweier JOBs
DISKS	Leitet die Plattenparameter ein. Als nächstes muß die Anzahl der Platten angegeben werden und dann für jede dieser Platten eine Verteilungsfunktion, die die Plattenzugriffszeiten beschreibt.
disk_statistic	Flag (1 oder 0), das die in der entsprechenden Warteschlange geführte Statistik aktiviert oder nicht.
cpu_anzahl	Anzahl CPUs in diesem K1-Knoten.
cpu_statistic	Flag (1 oder 0), das die in der entsprechenden Warteschlange geführte Statistik aktiviert oder nicht (siehe Abschnitt 4.2.8 [*Warteschlangen*], Seite 19).
cpu_req_distr	Verteilungsfunktion, nach der die von einem JOB benötigte Rechenzeit bestimmt wird.
cpu_slot_time	Gibt die Größe einer CPU-Zeitscheibe für das Round Robin-Verfahren an.
p_cpu_disk	Gibt die Wahrscheinlichkeit an, daß ein JOB nach Verlassen der CPU auf eine Platte zugreift (≤ 1).
p_cpu_cntr	Gibt die Wahrscheinlichkeit an, daß ein JOB nach Verlassen der CPU eine Nachricht sendet. (≤ 1).
msg_size_distr	Verteilungsfunktion, nach der die Größe der zu sendenden Nachricht bestimmt wird.
calc_empfaenger	Definiert die Funktion, nach der der Empfänger einer Nachricht bestimmt wird.
mem_pag	Anzahl der zur Verfügung stehenden Speichereinheiten.
mem_statistic	Flag (1 oder 0), das die in der entsprechenden Warteschlange geführte Statistik aktiviert oder nicht.
mem_req_distr	Verteilungsfunktion nach der die Anzahl benötigter Speichereinheiten für einen JOB bestimmt wird.
job_leave_probept	Meßpunkt in der Funktion *k1_destroy_job()* (siehe Abschnitt 7.3.14 [*Meßpunkte*], Seite 97).

Tabelle 7.2: Die Paramteter des Knotenmoduls K1

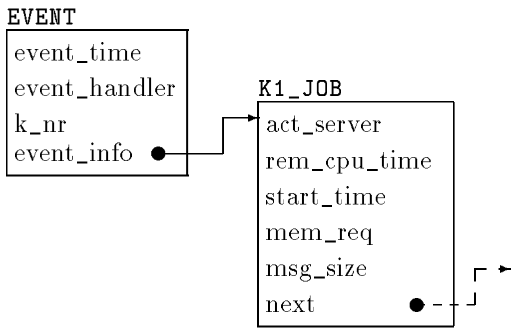


Abbildung 7.3: Datenstruktur, die einen JOB im K1-Modul beschreibt

Nachdem die einzelnen Datenstrukturen erzeugt und verkettet worden sind, werden die von diesem JOB benötigten Gesamtanforderungen an Speicher und CPU-Zeit über die in der Konfiguration angegebenen Verteilungsfunktionen *cpu_req_distr* bzw. *mem_req_distr* bestimmt und im JOB vermerkt.
Nachfolgend wird **direkt** die Funktion *k1_get_mem()* mit dem JOB als Parameter aufgerufen. Die Erzeugung eines JOBs verbraucht **keine** Simulationszeit.

7.3.4 k1_get_mem()

Diese Funktion wird **direkt** von *k1_create_job()* mit einem `JOB` als Parameter aufgerufen und simuliert die Speicherreservierung.

Wenn noch genügend freie Speichereinheiten vorhanden sind, um die Anforderung des `JOB`s zu erfüllen, wird diesem die angeforderte Anzahl Einheiten zugewiesen. Anschließend wird die Zeit für das Ereignis CPUArr bestimmt, die sich in diesem Modell aus der aktuellen Simulationszeit ergibt. Falls der Zeitverbrauch der Speicherverwaltung simuliert werden soll, muß der Zeitpunkt für CPUArr an dieser Stelle entsprechend nach hinten verschoben werden. Zum Schluß wird das Ereignis CPUArr eingeplant.

Falls die Anzahl freier Speichereinheiten kleiner als die vom `JOB` angeforderte Anzahl Einheiten ist, wird dieser in die Speicherwarteschlange eingefügt. Die implementierte Bedienstrategie dieser Warteschlange ist `FIFO`. Ein Überschreiten der maximalen Warteschlangenlänge führt zum Abbruch des aktuellen Simulationlaufes (siehe Abschnitt 4.2.8 [*Warteschlangen*], Seite 19).

7.3.5 k1_cpu_cust_arr()

Diese Funktion wird durch das Ereignis CPUArr aktiviert und erhält als Parameter einen `JOB` übergeben. Sie simuliert die Ankunft eines `JOB`s an der `CPU`.

Falls eine `CPU` frei ist, wird diese dem `JOB` für eine gewisse Zeit, maximal aber für die Zeitdauer *cpu_slot_time* (T_{Slot}) zugeteilt.
Das Ereignis CPUCompl löst das Freigeben der `CPU` aus. Der Zeitpunkt T_{Finish} dieses Ereignisses ergibt sich folgendermaßen:

$$T_{Finish} = \min(T_{Rest}, T_{Slot}) + \mathit{simul\_time}.$$

Nachdem die Restrechenzeit T_{Rest} des `JOB`s um die zugewiesene Rechenzeit verkleinert und gespeichert wurde, wird für den `JOB` ein Ereignis CPUCompl zum Zeitpunkt T_{Finish} eingeplant.

In dem Fall, daß alle `CPU`s belegt waren, wird der `JOB` in die `CPU`-Warteschlange eingehängt.
Die implementierte Bedienstrategie dieser für alle `CPU`s gemeinsamen Warteschlange ist `FIFO`.
Ein Überschreiten der maximalen Warteschlangenlänge führt zum Abbruch des aktuellen Simulationlaufes (siehe Abschnitt 4.2.8 [*Warteschlangen*], Seite 19).
Die `CPU`s werden nach der `Round Robin`-Strategie vergeben.

7.3.6 k1_cpu_serv_compl()

Diese Funktion wird durch das Ereignis CPUCompl aktiviert und erhält als Parameter einen `JOB` übergeben. Sie simuliert das Freigeben einer `CPU`.

In Abb. 7.4 ist der Kontrollfluß in dieser Funktion dargestellt.

Über die in der Konfiguration angegebenen Wahrscheinlichkeiten *p_cpu_disk*, *p_cpu_cntr* wird die nächste Aktivität dieses `JOB`s bestimmt. Die Wahrscheinlichkeit, sofort wieder eine `CPU` anzufordern, ergibt sich aus $1 - \mathit{p\_cpu\_disk} - \mathit{p\_cpu\_cntr}$.

Falls der `JOB` auf eine Platte zugreifen will, wird die Plattennummer zufällig aus allen installierten Platten ausgewählt und für den `JOB` ein Ereignis DiskArr zum Zeitpunkt *simul_time* eingeplant.

In dem Fall, daß der `JOB` eine `Nachricht` senden möchte, wird für den `JOB` ein Ereignis SendMsgArr zum Zeitpunkt *simul_time* eingeplant.

Die letzte Möglichkeit für einen `JOB` ist, falls er noch Rechenzeit benötigt, erneut eine `CPU` anzufordern. In diesem Fall wird für den `JOB` ein Ereignis CPUArr zum Zeitpunkt *simul_time* eingeplant. Falls der `JOB` keine Rechenzeit mehr benötigt, wird die Funktion *k1_destroy_job* direkt mit diesem `JOB` als Parameter aufgerufen.

Nach Abschluß der beschriebenen Aktionen wird geprüft, ob die `CPU`-Warteschlange leer ist. Ist dieses nicht der Fall, wird der erste `JOB` aus der Warteschlange entnommen und für diesen *k1_cpu_serv_compl()* — *Ereignisfunktion des Ereignisses* CPUCompl

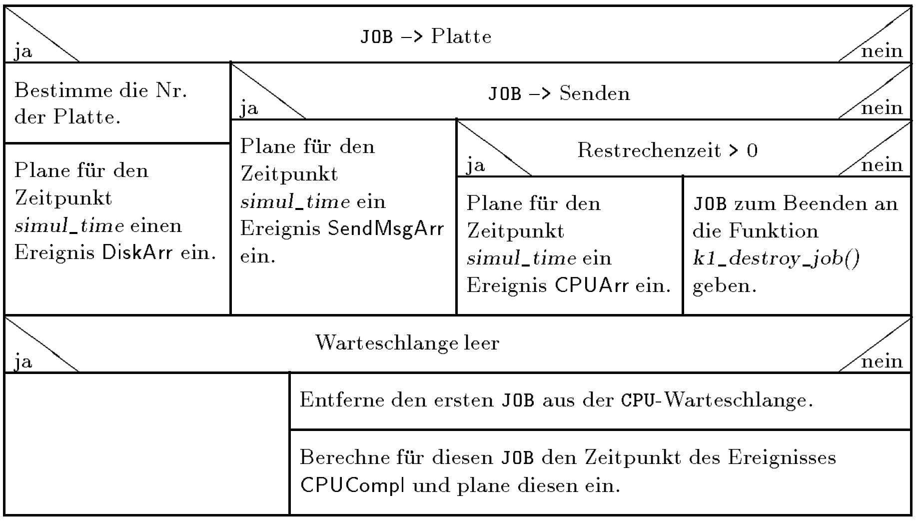


Abbildung 7.4: Kontrollfluß in der Funktion *k1_cpu_serv_compl()*

ein Ereignis CPUCompl eingeplant. Der Zeitpunkt dieses Ereignisses wird analog zu der in Abschnitt 7.3.5 [*k1_cpu_cust_arr()*], Seite 93 beschriebenen Vorgehensweise berechnet.

Falls die für die Aktionen benötigte Systemzeit simuliert werden soll, müßen die Ereignisse statt zum Zeitpunkt *simul_time* zu einem entsprechend späteren Zeitpunkt eingeplant werden.

7.3.7 k1_disk_cust_arr()

Diese Funktion wird durch das Ereignis DiskArr aktiviert und erhält als Parameter einen `JOB` übergeben. Sie simuliert die Ankunft eines `JOB`s an einer Platte.

Falls die gewünschte Platte frei ist, wird diese dem `JOB` zugeteilt. Das Ende des Plattenzugriffes wird durch das Ereignis DiskCompl festgelegt. Der Zeitpunkt T_{Finish} für dieses Ereignis ergibt sich aus der Summe von *simul_time* und der Plattenzugriffszeit, die nach der in der Konfiguration für diese Platte vorgegebenen Verteilungsfunktion bestimmt wird. Für diese Zeit T_{Finish} wird dann das Ereignis DiskCompl eingeplant.

Im Fall einer belegten Platte wird der `JOB` in die entsprechende Warteschlange, die nach dem `FIFO`-Verfahren arbeitet, eingehängt. Ein Überschreiten der maximalen Warteschlangenlänge führt zum Abbruch des aktuellen Simulationlaufes (siehe Abschnitt 4.2.8 [*Warteschlangen*], Seite 19).

7.3.8 k1_disk_serv_compl()

Diese Funktion wird durch das Ereignis DiskCompl aktiviert und erhält als Parameter einen `JOB` übergeben. Sie simuliert das Ende eines Plattenzugriffes.

In Abb. 7.5 ist der Kontrollfluß in dieser Funktion dargestellt.

Falls dieser `JOB` noch Rechenzeit benötigt, wird für ihn ein Ereignis CPUArr zur Zeit *simul_time* eingeplant. Falls die Systemzeiten, die bei den Abschlußarbeiten nach einem Plattenzugriff anfallen, simuliert werden sollen, ergibt sich die Zeit für das Ereignis CPUArr aus der Summe dieser Systemzeit und *simul_time*.
Wenn der `JOB` keine Rechenzeit mehr benötigt, wird die Funktion *k1_destroy_job* direkt mit diesem `JOB` als Parameter aufgerufen.

k1_disk_serv_compl() — Ereignisfunktion des Ereignisses DiskCompl

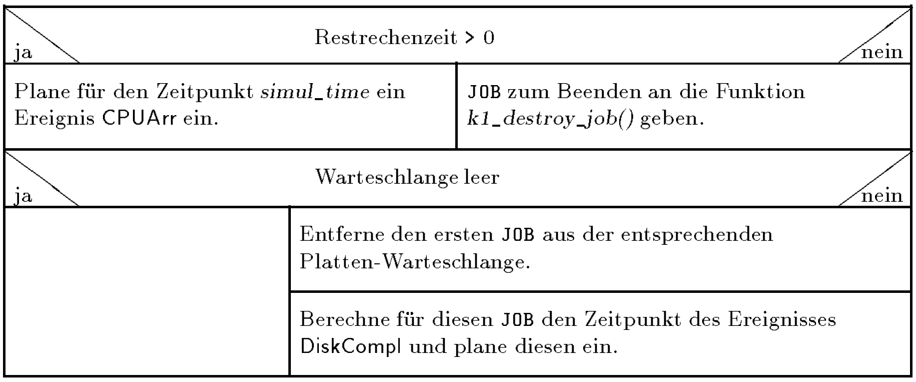


Abbildung 7.5: Kontrollfluß in der Funktion *k1_disk_serv_compl()*

Nach Abschluß der beschriebenen Aktionen wird geprüft, ob die Warteschlange der freigewordenen Platte leer ist. Ist dieses nicht der Fall, wird der erste `JOB` aus der Warteschlange entnommen und für diesen ein Ereignis DiskCompl eingeplant. Der Zeitpunkt dieses Ereignisses wird analog zu der in Abschnitt 7.3.7 [*k1_disk_cust_arr()*], Seite 94 beschriebenen Vorgehensweise berechnet.

7.3.9 k1_appl_send_arr()

Diese Funktion wird durch das Ereignis SendMsgArr aktiviert und erhält als Parameter einen `JOB` übergeben. Sie simuliert den Sendewunsch eines `JOB`s.

Zunächst wird nach der vorgegebenen Verteilungsfunktion *msg_size_distr* die Größe der zu sendenden `Nachricht` ermittelt und dann eine neue `Nachricht` erzeugt. Anschließend wird über die in der Konfiguration angegebene Funktion *calc_empfaenger* der Empfänger dieser `Nachricht` bestimmt.
Die Abb. 7.6 beschreibt die verwendete interne Darstellung einer solchen `Nachricht`.

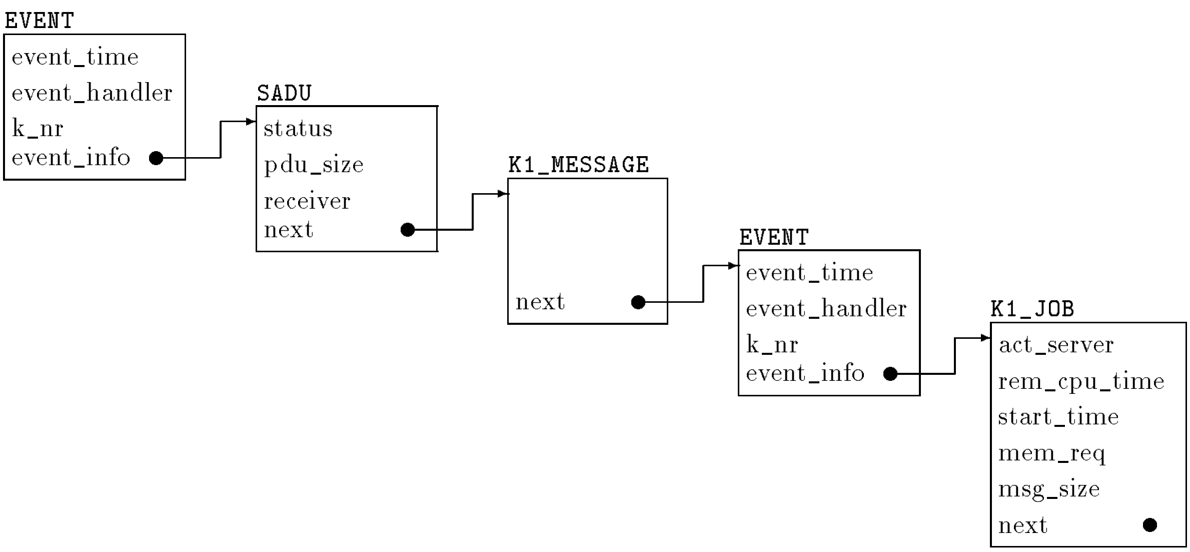


Abbildung 7.6: Datenstruktur, die eine `Nachricht` im `K1`-Modul beschreibt

Der `JOB` wird während des Sendens angehalten. Die beschreibende Datenstruktur (siehe Abschnitt 7.3 [*k1_create_job()*], Seite 92) muß währendessen so gespeichert werden, daß der dazugehörige `JOB` nach Übertragungsende wieder reaktiviert werden kann. Dieses Problem wird

gelöst, indem ein Zeiger auf den JOB in *K1_MESSAGE→next* vermerkt wird, über den der JOB in der Funktion *k1_message_sended()* wieder gestartet werden kann.
Diese Vorgehensweise erfordert allerdings, daß die Datenstrukturen von den für die Übertragung benutzten niedrigeren simuLan-Schichten nicht verändert wird.

Nachdem die Datenstrukturen vollständig initialisiert worden sind, wird die **Nachricht** an die Funktion *(*nodearray[k_nr].l4_sap)()* gegeben, die für die weitere Übertragung verantwortlich ist (siehe Abschnitt 4.2.6 [*NODE*], Seite 18). *K_nr* ist die Nummer des K1-Knotens, in dem der aktuelle JOB abläuft (siehe Abschnitt 4.2.2 [*EVENT*], Seite 15).

Die Datenstruktur *K1_MESSAGE* wird im Augenblick, mit Ausnahme des enthaltenen Zeigers *next*, nicht weiter genutzt. Diese Datenstruktur bietet, bei entsprechender Erweiterung jedoch die Möglichkeit, eine vollständige JOB-Beschreibung an den Empfänger der **Nachricht** zu senden, der dann diesen JOB ausführt und das Ergebnis zurücksendet. Auf diese Weise ist z.B. das Verhalten eines *Fileservers* mit mehreren angeschlossenen Rechnern modellierbar.

7.3.10 k1_message_sended()

Diese Funktion wird von der simuLan-Schicht *4* nach einer Übertragung aufgerufen und erhält als Parameter einen JOB übergeben. Sie simuliert das Ende der Übertragung in der simuLan-Schicht *5* und die Reaktivierung des JOBs, für den die **Nachricht** gesendet wurde.

Falls dieser JOB noch Rechenzeit benötigt, wird für ihn ein Ereignis CPUArr zur Zeit *simul_time* eingeplant. Falls die Systemzeiten, die bei den Abschlußarbeiten nach dem Senden einer **Nachricht** anfallen, simuliert werden sollen, ergibt sich die Zeit für das Ereignis CPUArr aus der Summe dieser Systemzeit und *simul_time*.
Wenn der JOB keine Rechenzeit mehr benötigt, wird die Funktion *k1_destroy_job* direkt mit diesem JOB als Parameter aufgerufen.

Zum Schluß werden die in der Funktion *k1_appl_send_arr()* erzeugten Datenstrukturen, die eine **Nachricht** beschreiben, wieder an die Speicherverwaltung von simuLan zurückgegeben.

7.3.11 k1_message_rec()

Diese Funktion wird von der simuLan-Schicht *4* auf der Empfängerseite aufgerufen und entspricht der in Abschnitt 4.2.5 [*Das Schicht-Modul*], Seite 16 beschriebenen Funktion *XLayer_pdu_received()*.

Die einzige Aktion in der augenblicklichen Implementation besteht darin, die Anzahl empfangener **Nachricht**en zu zählen.

Spätere Modelle könnten in dieser Funktion die empfangenen **Nachricht**en auswerten und so zum Beispiel einen am *Fileserver* ankommenden Auftrag simulieren.

7.3.12 k1_destroy_job()

Diese Funktion wird von den Funktionen *k1_cpu_serv_compl()*, *k1_disk_serv_compl()* und *k1_message_sended()* aufgerufen, wenn ein JOB keine Rechenzeit mehr benötigt. Dieser wird als Parameter an *k1_destroy_job()* übergeben. Dort wird das Ende dieses JOBs simuliert.

Als erstes werden die vom JOB belegten Speichereinheiten an die simulierte Speicherverwaltung zurückgegeben und anschließend die den JOB beschreibenden Datenstrukturen, die in der Funktion *k1_create_job* erzeugt wurden, wieder freigegeben (siehe Abschnitt 7.3 [*k1_create_job()*], Seite 92).

Zum Schluß wird geprüft, ob jetzt die Speicheranforderung des ersten JOBs in der Speicherwarteschlange befriedigt werden kann. Falls dieses zutrifft, wird der JOB aus der Warteschlange entnommen und erhält die angeforderten Speichereinheiten zugeteilt. Dann wird für ihn ein Ereignis CPUArr eingeplant. Anschließend wird dieser Test für den nächsten JOB in der Warteschlange durchgeführt. Dieser Vorgang wird solange wiederholt, bis entweder die Speicherwarteschlange leer ist oder für den ersten JOB in der Warteschlange nicht mehr genügend freie Speichereinheiten zur Verfügung stehen.

7.3.13 Rahmenfunktionen

Die Rahmenfunktionen des Knotenmoduls `K1` sind *k1_rc()*, *k1_rc_disks()*, *k1_init()*, *k1_cleanup()*, *k1_print_info()*, *k1_print_job()* und *k1_print_message()*.

Die Prozedur *k1_rc()* initialisiert entsprechend dem übergebenen Parameter *count*, *count* **identische** `Knotenmodule` vom Typ `K1`.
Dabei fallen folgende Aufgaben an:

- Falls über die Kommandozeile angegeben, werden alle variablen Parameter eines `K1-Moduls` mit einer Erklärung ausgegeben.
- Die Datenstruktur *k1_rc_info*, in der die Werte aus der Konfiguration zwischengespeichert werden, entspricht *k_info* (siehe Abschnitt 4.2.6 [*NODE*], Seite 18), mit den Defaultwerten vorbelegen.
- Die Funktion *rcreadkeywords()* wird mit der Tabelle der definierten Schlüsselwörter für das `K1`-Modul aufgerufen. Dabei werden gegebenenfalls die Defaultwerte ausgegeben.
- Die eingelesenen Parameter werden auf Plausibilität geprüft.
- Die Datenstruktur `NODE` in *nodearray[]* (siehe Abschnitt 4.2.6 [*NODE*], Seite 18) und den angehängten *K1_INFO* der *count* `K1`-Knoten entsprechend dem Inhalt von *k1_rc_info* initialisieren.

Die Funktion *k1_rc_disks()* wird von *rcreadkeywords()* aufgerufen, wenn in der Konfiguration das Schlüsselwort `DISKS` gelesen wird. Diese Funktion liest zunächst die gewünschte Anzahl Platten ein und erwartet dann für jede Platte die dazugehörige Verteilungsfunktion, die die Verteilung der Plattenzugriffszeiten beschreibt.

Die Aufgabe der Funktion *k1_init()* ist es, für den entsprechenden `K1`-Knoten ein Ereignis CreateJob zu erzeugen und einzuplanen. Der Zeitpunkt für dieses erste Ereignis wird nach der Verteilungsfunktion *term_ank_distr* aus der Konfiguration bestimmt. Die späteren Ereignisse vom Typ CreateJob werden in der entsprechenden Ereignisfunktion *k1_create_job()* erzeugt (siehe Abschnitt 7.3.3 [*k1_create_job()*], Seite 91).

Die Funktion *k1_cleanup()* gibt nach Beendigung eines Simulationslaufes den Knotennamen aus und ruft anschließend die Funktion *k1_print_info()* auf.

Die Funktion *k1_print_info()* gibt die in den Warteschlangen gesammelten Daten und die Anzahl gesendeter und empfangener `Nachricht`en aus.

Die Funktionen *k1_print_job()* und *k1_print_message()* geben den Inhalt der für dieses Modul spezifischen Datenstrukturen *K1_JOB* und *K1_MESSAGE* aus. Diese Funktionen werden für eine Fehlersuche benötigt und sollen in einer späteren Version von simuLan das Verfolgen von `JOB`s und `Nachricht`en im ganzen modellierten System ermöglichen.

7.3.14 Der Meßpunkt k1_job_destroy_probept

Der Meßpunkt *k1_job_destoy_probept* ist in der Funktion *k1_destroy_job()* eingebaut. Alle an diesem Meßpunkt aktivierten Funktionen werden jeweils bei Ende eines `JOB`s aufgerufen.
Die einzige bisher implementierte Meßfunktion ist *k1_job_livetime()*, deren Arbeitsweise unten kurz erläutert wird.

Genauere Informationen zu Meßpunkten sind in Abschnitt 4.2.7, Seite 19 und besonders in [Pet91] zu finden.

7.3.14.1 Die Meßfunktion *k1_job_lifetime*

Zu dieser Meßfunktion gehören die Funktionen *k1_rc_job_lifetime()*, *k1_job_lifetime()* und *k1_job_lifetime_erg()*.

k1_rc_job_lifetime() aktiviert die eigentliche Meßfunktion *k1_job_lifetime()* und die dazugehörige Funktion *k1_job_lifetime_erg()*, die nach jedem Simulationslauf die erhaltenen Meßwerte ausgibt.
Die Meßfunktion *k1_job_lifetime()* ermittelt die Laufzeit eines `JOB`s von dem Zeitpunkt des Eintretens in das System über die Funktion *k1_create_job()* bis zum Verlassen über die Funktion *k1_destroy_job()*. Die Funktion *k1_create_job()* vermerkt in `K1_JOB`→start_time die Erzeugungszeit des `JOB`s. Damit ergibt sich die Laufzeit aus :
simul_time - `K1_JOB`→start_time.

7.4 Verifikation

In diesem Abschnitt wird die Verfikation des `K1`-Knotenmoduls beschrieben.

Der erste Schritt der Verifikation wurde durchgeführt, indem einzelne `JOB`s von der Entstehung bis zum Ende verfolgt wurden. Dabei wurden die Möglichkeiten von simuLan zum Verfolgen von Ereignissen verwendet.
Auf diese Weise ist sichergestellt worden, daß die gewünschte Abfolge von Stationen, die der `JOB` befolgen soll, auch wirklich eingehalten wird.

Im zweiten Schritt wurde ein einfaches System modelliert und simuliert. Anschließend wurden die Kennwerte des Systems mittels *operationaler Analyse* bestimmt und mit den Ergebnissen der Simulation verglichen [LZGS84].

Das modellierte System ist in Abb. 7.7 dargestellt.

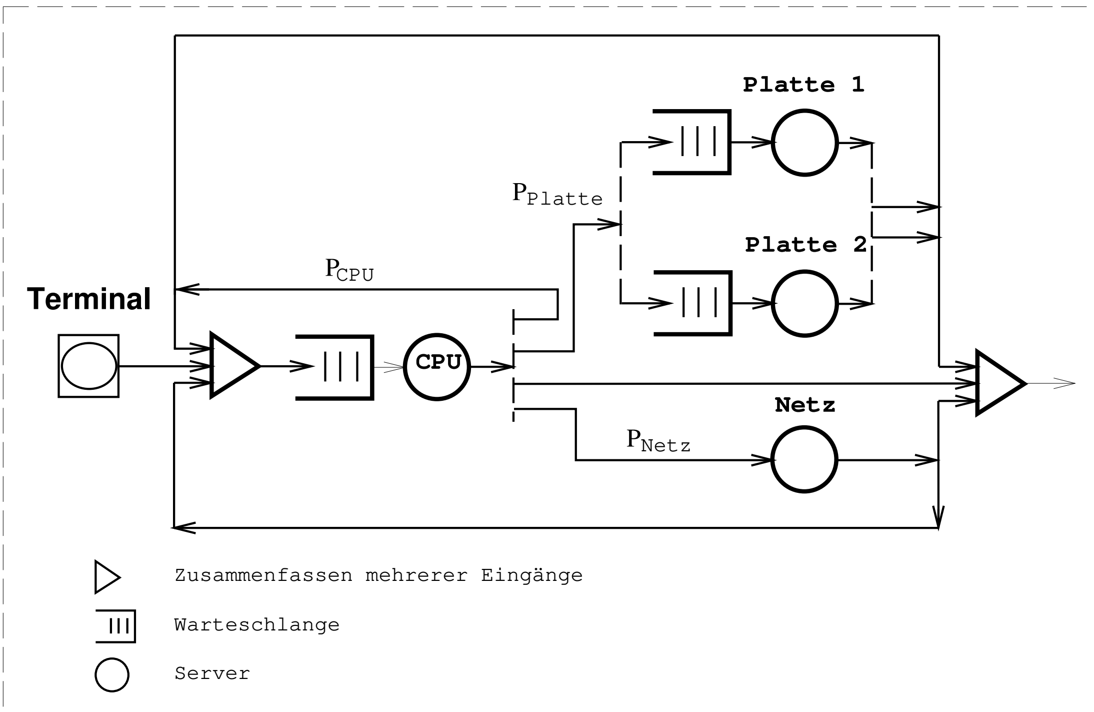


Abbildung 7.7: Das System für die Verifikation

Das Netz wird durch das Netzmodul `PTP` realisiert (siehe Abschnitt 4.3 [*Das Netzmodell* PTP], Seite 21).
Dieses Netzmodul verzögert jeden `JOB` um die Zeit, die der reinen Übertragungszeit der gesendeten `Nachricht` entspricht. Beliebig viele `JOB`s einer Station können gleichzeitig senden, da in dem Netzmodul `PTP` keine Überprüfung stattfindet, ob schon ein `JOB` dieser Station sendet.
Damit reduziert sich das Netz zu einer reinen Verzögerungsstufe, die sich leicht auch analytisch behandeln läßt.

Die Parameter des Systems und ihre Werte sind in dem folgenden Ausschnitt aus der Konfigurationsdatei angegeben.

```
NET PTP
        error_distr      konstant 1e50
        bits_per_second  1e3
        overhead         0
        min_data_len     0
        max_data_len     100000
    ENDptp
ENDnet

NODES
    NAME Hugo
    LAYER5 "k1"
                term_ank_distr  expntl 0.5 s
                DISKS
##                    Anzahl <distr> <parms..> [ <distr> <parms..>]..
                        2
                        expntl  100 ms
                        expntl  200 ms

                disk_statistic  1

                calc_empfaenger self_empf

                cpu_anzahl      1
                cpu_req_distr   expntl 50 ms
                cpu_slot_time   5 ms
                cpu_statistic   1

                p_cpu_disk      0.4
                p_cpu_cntr      0.2

                msg_size_distr  expntl 150 Bit

                mem_pag         2000
                mem_req_distr   konstant 1
   ENDk1
```

Es wurden nur Exponentialfunktionen für die Verteilungsfunktionen eingesetzt, damit das System auch analytisch lösbar ist.

Das Netz besitzt eine Übertragungsrate von 1000 Bit/s. Der zu einer `Nachricht` hinzugefügte Overhead und die Mindestgröße eines `Frame`s ist gleich Null. Damit wird erreicht, daß die Größe der `Nachricht` direkt der erzeugten Verzögerung durch das Netz entspricht.
Im Mittel kommt alle 0.5 Sekunden ein neuer `JOB` an der einen modellierten `CPU` an. Jeder dieser `JOB`s benötigt im Mittel 50 ms Rechenzeit und eine Speichereinheit. Die Größe einer `CPU`-Zeitscheibe ist 5 ms.
Da die Modellierung der Speicherverwaltung auf analytischem Wege zu komplex gewesen wäre, um sie in der zur Verfügung stehenden Zeit durchzuführen, wird die Anzahl der verfügbaren Speichereinheiten so groß gewählt, daß die Speicherverwaltung keinen begrenzenden Einfluß auf den Durchsatz des Systems hat.
Nach Verlassen der `CPU` fordert ein `JOB` mit 40 Prozent Wahrscheinlichkeit (P_{Platte}) eine Disk an. Mit 20 Prozent Wahrscheinlichkeit (P_{Netz} wird er eine `Nachricht` senden und mit einer Wahrscheinlichkeit (P_{CPU}) von $1 - p\_cpu\_disk - p\_cpu\_cntr = 40$ Prozent erneut die `CPU` anfordern.
Die mittlere Plattenzugriffszeit beträgt für `Platte 1` 100 ms und für `Platte 2` 200 ms. Die mittlere Größe einer `Nachricht` ist 150 Bit. Die daraus resultierende Verzögerungszeit einer `Nachricht` durch den Server `Netz` ist 150 ms.

Die Meßwerte wurden mit den von simuLan bereitgestellten Warteschlangenfunktionen für den

jeweiligen Server ermittelt (siehe Abschnitt 4.2.8 [*Warteschlangen*], Seite 19).
Es wurden ca. 100000 JOBs simuliert.

Die Tabelle 7.3 vergleicht die über die Simulation ermittelten Werte mit den aus der analytischen Methode gewonnenen Werten.

Bedeutung	Simulation	Analytisch
Mittlere Auslastung der CPU	9.968 %	10 %
Mittlerer Durchsatz der CPU	20.95	20
Mittlere Auslastung der Platte 1	41.93 %	40.0 %
Mittlerer Durchsatz der Platte 1	4.195	4
Mittlere Auslastung der Platte 2	83.52 %	80.0 %
Mittlerer Durchsatz der Platte 2	4.19	4.0

Tabelle 7.3: Vergleich: Simulation – Analytische Modellierung

Die ermittelten Werte der analytischen und der simulativen Modellierung stimmen gut überein.

Die Verifikation ist damit abgeschlossen. Eigentlich müßte nun eine Validierung des K1-Knotenmodells erfolgen. Diese war nicht möglich, da keine Meßdaten realer Systeme zur Verfügung standen.

7.5 Erweiterungsmöglichkeiten

Dieser Abschnitt stellt einige Erweiterungsmöglichkeiten des K1-Knotenmoduls vor, die aus Zeitgründen in dieser Diplomarbeit nicht mehr realisiert werden konnten.

Bei dem Aufbau oder der Analyse eines Netzwerkes stellt sich oft die Frage, welchen Einfluß der Fileserver auf den Durchsatz und die Antwortzeiten hat.

Für die Beantwortung dieser Frage mit simuLan ist es notwendig, das Senden von Aufträgen an den Fileserver, die Bearbeitung derselben und das Senden der Antwort an den Auftragsteller simulieren zu können. Die für diese Möglichkeit notwendigen Erweiterungen im K1-Knotenmodul sind in den entsprechenden Abschnitten schon angedeutet worden und werden hier noch einmal kurz zusammengefaßt.

In die Datenstruktur K1_MESSAGE muß die notwendige Information für die Erzeugung eines JOBs eingefügt werden. Um die Aufträge variabel zu gestalten, muß die Struktur K1_INFO um Platz für die notwendigen Parameter erweitert werden.
In der Funktion *k1_appl_send_arr()* muß die Datenstruktur K1_MESSAGE entsprechend ausgefüllt werden, damit auf der Empfängerseite ein dem Auftrag entsprechender JOB erzeugt und von dem simulierten System bearbeitet werden kann. Ein so erzeugter JOB muß von ,,normalen", stationseigenen JOBs unterschieden werden können, damit bei seiner Beendigung eine Nachricht an den Auftraggeber zurückgesendet werden kann.

Andere Erweiterungen betreffen die Repräsentation des simulierten Systems.
Es ist zum Beispiel wünschenswert, die Zugriffe zu den Platten, die bisher sehr einfach dargestellt werden, sehr viel genauer zu modellieren. Auch sollte es möglich sein, verschiedene Bedienstrategien für die Server (CPU, Platte, Netz) über die Konfiguration auswählen zu können.

Eine letzte Erweiterung ist die Simulation von JOBs mit verschiedenen Charakteristiken in **einem** K1-Knotenmodell. Beispiele für solche JOBs sind Editor, Übersetzer und kleine Dienstprogramme, die alle sehr unterschiedliche Anforderungen an Speicher, CPU-Zeit und Plattenzeit stellen.

Kapitel 8

Zusammenfassung und Ausblick

In den vorhergehenden Kapiteln wurde die Modellierung und Simulation verschiedener Protokollschichten in einem lokalen Netzwerk unter Verwendung des Simulationswerkzeugs simuLan beschrieben. Dabei hat sich gezeigt, daß simuLan eine Umgebung bereitstellt, in der verschiedenste Systeme, real oder fiktiv, sehr flexibel modelliert und simuliert werden können.
Die Veränderbarkeit der Modellparameter über die Konfigurationsdatei, die verfügbaren Funktionen für Warteschlangen, Eventliste, statistische Absicherung und besonders die kurzen Programmlaufzeiten ermöglichen die detailgetreue Simulation komplexer Abläufe mit einem, im Gegensatz zu reinen Simulationssprachen, sehr geringen Zeitaufwand.

Der Aufwand und die Komplexität der Entwicklung sowie der Implementation eines neuen Schichtmodells ist allerdings relativ hoch. Dieses gilt besonders dann, wenn man alle Möglichkeiten, die simuLan zum Verfolgen von Ereignissen und zum Ermitteln von Meßwerten bietet, ausnutzen möchte.

Für einen Einstieg in die Modellierung und Simulation mit simuLan wird empfohlen, das Kapitel 4 in dieser Arbeit genau zu lesen und eigene Modellierungsprojekte auf den einfachen, dort beschrieben Schichtmodellen aufzubauen.
Insbesondere die von simuLan vorausgesetzten Schnittstellen eines Moduls müssen genau eingehalten werden, um die Austauschbarkeit verschiedener Schichtmodule für eine Schicht untereinander zu gewährleisten. Gerade diese Austauschbarkeit ermöglicht die mit zunehmender Anzahl implementierter Modelle immer einfachere Modellierung und Simulation der verschiedensten Systeme mit simuLan.

Mit den in dieser Arbeit entwickelten und implementierten Modellen lassen sich gute Vorhersagen über die Leistungswerte verschiedener Netzwerke, die auf dem `CSMA/CD`-Verfahren basieren, machen.
Die Entwicklung weiterer Modelle, insbesondere eines Modells für Netze, die nach dem `TOKEN-RING`-Verfahren arbeiten, ist sicherlich wünschenswert.
Auch die Erweiterung des Rechnermodells `K1`, wie sie im Kapitel 7.5 erläutert wurde, ist anzustreben, um den Einfluß der am Netz angeschlossenen Stationen besser darstellen zu können. Letztlich fehlen noch Modelle für Protokolle, wie z.B. `TCP` und `IP`. Ein Modell für das `IP`-Protokoll wurde teilweise implementiert, konnte aber aus Zeitmangel nicht vollendet werden. Die zugehörigen Quellen sind in Anhang D enthalten.

Mit diesen Erweiterungen sollte die realitätsgetreue Darstellung und Simulation der Abläufe in einem lokalen Netzwerk möglich sein und zuverlässige Vorhersagen über die Leistungswerte des modellierten Systems erlauben.

Abbildungsverzeichnis

Tabellenverzeichnis

Anhang A

Liste der Quelldateien

Dieser Anhang führt alle in dieser Arbeit entwickelten „**C**"-Quelldateien mit Angabe der enthaltenen Funktionen und Datenstrukturen auf.
Dateien des Netzmoduls `PTP`:

Dateiname	Funktionen
ptp.c	ptp_cleanup()
	ptp_data_encapsulation()
	ptp_end_transmit()
	ptp_init()
	ptp_mean_throughput()
	ptp_print_frame()
	ptp_rc()
	ptp_rc_throughput()
	ptp_throughput()
	ptp_transmit_error()
	PTP_INFO
	PTP_FRAME
ptp.h	Exportierte Funktionen

Dateien des Knotenmoduls `K0`:

Dateiname	Beschreibung
k0.c	k0_cleanup()
	k0_init_knoten()
	k0_msg_delay()
	k0_msg_delay_erg()
	k0_pdu_received()
	k0_pdu_sended()
	k0_print_info()
	k0_rc()
	k0_rc_msg_delay()
	k0_send_probept_printf()
	print_k0_message()
	K0_INFO
	K0_MESSAGE
k0.h	Exportierte Funktionen

Dateien des Netzmoduls `CSMA`:

Dateiname	Beschreibung
csma.c	csma_backoff()
	csma_cleanup()
	csma_data_encapsulation()
	csma_init()
	csma_transmit_error()
	CSMA_FRAME
csma_opt.c	csma_deassert()
	csma_defer()
	csma_end_transmit()
	csma_start_backoff()
	csma_start_transmit()
csma_genau.c	csma_deassert() (optimiert)
	csma_defer() (optimiert)
	csma_end_transmit() (optimiert)
	csma_start_backoff() (optimiert)
	csma_start_transmit() (optimiert)
	ACT_SENDER
csma_rc.c	csma_frame_delay()
	csma_genau_laufzeit()
	csma_init_gleichvert()
	csma_init_konstant()
	csma_init_linear()
	csma_konstant_laufzeit()
	csma_probept_end_transmit_printf()
	csma_rc()
	csma_rc_frame_delay()
	csma_rc_gleichvert()
	csma_rc_konstant()
	csma_rc_linear()
	csma_rc_utilisation()
	csma_utilisation()
	csma_utilisation_erg()
	print_csma_frame()
	print_mean_csma_frame_delay()
	CSMA_INFO
csma.c	Exportierte Funktionen

Dateien des Schichtmoduls `l2simple`:

Dateiname	Beschreibung
l2simple.c	l2simple_pdu_received()
	l2simple_pdu_sended()
	l2simple_print_info()
	l2simple_rc()
	l2simple_sap()
	L2SIMPLE_INFO
l2simple.h	Exportierte Funktionen

Dateien des Knotenmoduls `K1`:

Dateiname	Beschreibung
k1.c	k1_appl_send_arr()
	k1_cleanup()
	k1_cntr_cust_arr()
	k1_cntr_serv_compl()
	k1_cpu_cust_arr()
	k1_cpu_serv_compl()
	k1_create_job()
	k1_destroy_job()
	k1_disk_cust_arr()
	k1_disk_serv_compl()
	k1_get_mem()
	k1_init()
	k1_l2_pdu_rec()
	k1_message_rec()
	k1_message_sended()
	print_k1_message()
	K1_INFO
	K1_JOB
	K1_MESSAGE
k1.h	Exportierte Funktionen
k1_rc.c	k1_job_lifetime()
	k1_job_lifetime_erg()
	k1_print_info()
	k1_print_job()
	k1_rc()
	k1_rc_disks()
	k1_rc_job_lifetime()
k1_rc.h	Exportierte Funktionen

Anhang B

Debugflags

Über die Kommandozeile wird mit '`-D<modulname>=<int>`' der Wert des *,Debugflags'* für das Modul '`modulname`' vorgegeben.
'`int`' ist eine 32-Bit-Zahl. Die Zählung der Bits beginnt mit dem niederwertigsten Bit als Nr. *0*.
Die Bits *0-8* wählen die Station aus, für die in dem Modul Information ausgegeben wird. Damit kann genau eine Station aus 2^9 möglichen Stationen ausgewählt werden.
Das Bit Nr. *9* entscheidet darüber, ob die Information der Bits *0-8* beachtet wird (= 0) oder ob die Informationen für alle Stationen ausgegeben werden (= 1).
Die Bits mit Nummern größer als *9* aktivieren die Ausgabe der Zusatzinformation in Funktionen, die für das Modul '`modulname`' spezifisch sind.
In den Tabellen wird in der ersten Spalte die Nr. des Bits angegeben. In der zweiten Spalte ist angegeben in welcher Funktion das Setzen dieses Bits die Ausgabe von Zusatzinformation bewirkt.

Netzmodul *PTP* **:**
Zugehörige Dateien: ptp.c, ptp.h.

Bit-Nr.	Funktion:
0-8	Knotennummer fuer die Debugginginfo ausgegeben werden soll.
9	== 0 -> Fuer den in Bits 0-8 kodierten Knoten Info ausgeben.
	== 1 -> Fuer alle Knotennummern Info ausgeben.
10	ptp_data_encapsulation()
11	ptp_end_transmit()
12	ptp_transmit_error()
13	ptp_throughput()
14 -	... nicht benutzt

Netzmodul *l2simple* **:**
Zugehörige Dateien: l2simple.c, l2simple.h.

Bit-Nr.	Funktion:
0-8	Knotennummer fuer die Debugginginfo ausgegeben werden soll.
9	== 0 -> Fuer den in Bits 0-8 kodierten Knoten Info ausgeben.
	== 1 -> Fuer alle Knotennummern Info ausgeben.
10	l2simple_sap()
11	l2simple_pdu_sended()
12	l2simple_pdu_received()
13 -	... nicht benutzt

Netzmodul *CSMA* **:**
Zugehörige Dateien: csma.c, csma_rc.c, csma_gen.c, csma_opt.c, csma.h.

Bit-Nr.	Funktion: (In Datei: '`csma.c`')
0-8	Knotennummer fuer die Debugginginfo ausgegeben werden soll.
9	== 0 -> Fuer den in Bits 0-8 kodierten Knoten Info ausgeben.
	== 1 -> Fuer alle Knotennummern Info ausgeben.
10	data_encapsulation()
11	defer()
12	start_transmit()
13	deassert()
14	end_transmit()
15	start_backoff()
16 -	... nicht benutzt

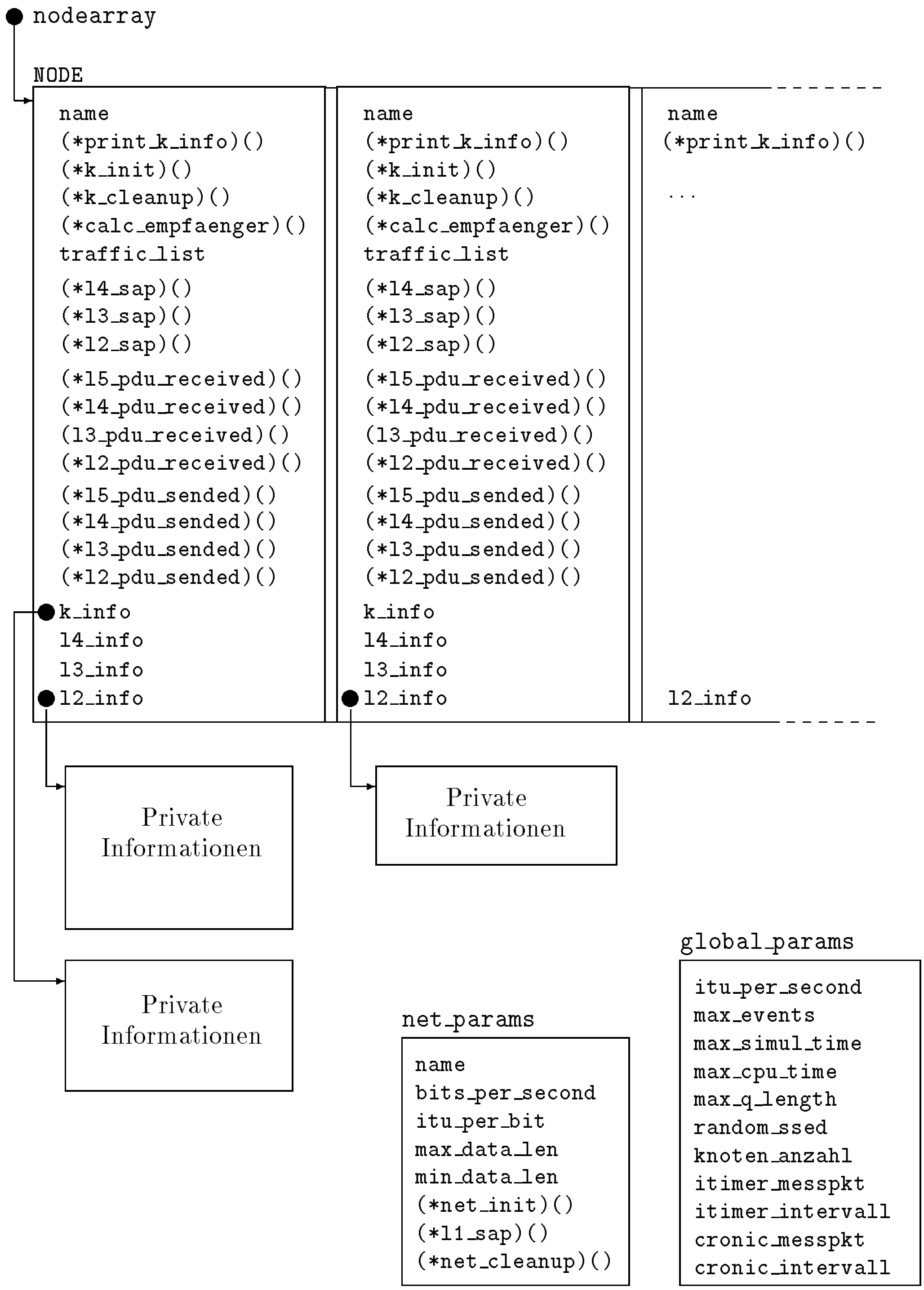
Bit-Nr.	Funktion: (In Datei: '`csma_rc.c`')
0-8	Knotennummer fuer die Debugginginfo ausgegeben werden soll.
9	== 0 -> Fuer den in Bits 0-8 kodierten Knoten Info ausgeben.
	== 1 -> Fuer alle Knotennummern Info ausgeben.
10	csma_frame_delay()
11	csma_utilisation()
12	calc_laufzeit()
13 -	... nicht benutzt

Knotenmodul *K1* **:**
Zugehörige Dateien: k1.c, k1_rc.c, k1.h.

Bit-Nr.	Funktion:
0-8	Knotennummer fuer die Debugginginfo ausgegeben werden soll.
9	== 0 -> Fuer den in Bits 0-8 kodierten Knoten Info ausgeben.
	== 1 -> Fuer alle Knotennummern Info ausgeben.
10	k1_create_job()
11	k1_get_mem()
12	k1_disk_cust_arr()
13	k1_disk_serv_compl()
14	k1_cpu_cust_arr()
15	k1_cpu_serv_compl()
16	k1_appl_send_arr()
17	k1_message_sended()
18	k1_destroy_job()
19	k1_message_rec()
23 -	... nicht benutzt

Anhang C

Globale Datenstrukturen



Anhang D

Programmquellen

Die beiliegende Diskette enthält den kompletten Quellcode für simuLan. Dieser umfaßt die Grundbibliothek, die bisher implementierten Protokollschichtmodule sowie die gesamte Dokumentation.

Die Diskette ist auf 720 Kbyte formatiert und mit `tar(1)` beschrieben. Das `tar`-Archiv enthält ein mit `freeze(1)` kompaktiertes `tar`-Archiv des simuLan-Dateibaumes, und die Quellen für das `freeze`-Programm, so wie sie vom Author Kent Landfield in den USENET-News verbreitet wurden (`freeze` erzielt eine deutlich bessere Kompaktierung als das übliche `compress(1)`, ist aber noch nicht so weit verbreitet).

Der Inhalt der Diskette kann mit `tar(1)` eingelesen werden. Dann können die simuLan-Quellen mit `melt(1)` oder `fcat(1)` (den Gegenstücken zu `freeze`) "aufgetaut" und wiederum mit `tar` ausgepackt werden.

```
% tar xvf /dev/rfd0 simuLan.tar.F
% fcat simuLan.tar.F | tar xvof -
```

Im simuLan-Dateibaum liegt ein '`README`', welches eine Anleitung zum Übersetzen und Installieren des Programms und der Dokumentation enthält.

Literaturverzeichnis

Nicht alle hier aufgeführten Veröffentlichungen hatten Einfluß auf die Entwicklung von simuLan. Wichtig sind die Quellen, auf die im Text verwiesen wird, sowie die mit Anmerkungen versehenen. Die anderen Verweise wurden aufgeführt, um einen Überblick über die Literatur zu diesem Thema zu geben.

[AB88] Ian F. Akyildiz und Hunter Bolch. Mean Value Analysis Approximation for Multiple Server Queueing Networks. *Performance Evaluation*, 8:77–91, 1988.

[AC88] C.C. Ang und E.J.P. County. An experiment in modelling communication protocols using NPNs. *Aust. Comput. J. (Australia)*, 20(2):73–8, Mai 1988.

[Ada83] N.R. Adam. Achieving a confidence interval for parameters estimated by simulation. *Management Science*, 29(7):856–866, Juli 1983.

Vergleich dreier Varianten des BMVA-Verfahrens. Die vorgeschlagene Methode B wurde in simuLan implementiert.

[Agr85] S.C. Agrawal. *Metamodelling. A Study of Approximations in Queueing Models.* MIT, 1985.

[Aha87] M. Ahamad. An Efficient Algorithm for Name Resolution in Computer Networks. *Computer Networks and ISDN Systems*, 13:301–311, 1987.

[Akh88] S. Akhtar. Performance evaluation of CSMA/CD networks using Markovian approach. *GLOBECOM '88. IEEE Global Telecommunications Conference and Exhibition - Communications for the Information Age*, 2:1108–12, 1988. Cat. Nr. 88CH2535-3.

[AKK+87] P.D. Amer, R.N. Kumar, R. Kao, T. Phillips, und L.N. Cassel. Local Area Broadcast Network Measurement: Traffic Characterization. *Proceedings of COMCON Spring 87,San Francisco*, Seite 64–70, Februar 1987.

[AL79] G.T. Almes und E.D. Lazowska. The Behavior of Ethernet-like Computer Communications Networks . *Proceedings 7th Symposium on Operating Systems Principles*, Seite 66–81, Dezember 1979.

[Azm88] M. Azmoodeh. *Abstract Data Types and Algorithms.* Macmillian Education LTD, 1988.

[BBC77] R.M. Brown, J.C. Browne, und K.M. Chandy. Memory Management and Response Time. *Communications of the ACM*, 20(3):153–165, März 1977.

[BC83] R. Berry und K.M. Chandy. Performance models of token ring local area networks. *Proceedings 1983 ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, Seite 266–274, August 1983.

[BC88a] S.L. Beuerman und E.J. Coyle. The Delay Characteristics of CSMA/CD Networks. *IEEE Trans. Commun.*, 36(5):553–563, 1988.

[BC88b] Andreé B. Bondi und Yie-Min Chuang. A new mva-based approximation for closed queueing networks with a preemptive priority server. *Performance Evaluation*, 8:195–221, 1988.

[BDMH87] M. Basu, G. Dutta, T.W. Morgan, und D.P. Heyman. A comparison study of the Queueing Network Analyzer with a CSMA/CD simulation. *Sixth Annual International Phoenix Conference on Computers and Communications. 1987 Conference Proceedings*, Seite 25–27, 1987. Cat. No. 87TH0179-2.

[Ber89] B.R. Bertan. Simulation of MAC layer queueing and priority strategies of CEBus. *IEEE Trans. on Consumer Electronics*, 35:557–563, August 1989.

[BM87] O.J. Boxma und B.W. Meister. Waiting-Time Approximations in Multi-Queue Systems with Cyclic Service. *Performance Evaluation*, 7:59–70, 1987.

[BMI87] S. Bhatia, H.T. Mouftah, und M. Ilyas. Hybrid modeling techniques for complex computer networks. *IEEE International Conference on Communications '87: Communications - Sound to Light. Proceedings.*, 3:1311–15, 1987. Cat. No. 87CH2424-0.

[BMK] D.R. Boggs, J.C. Mogul, und C.A. Kent. Errata for "Measured capacity of an Ethernet: myths and reality".

[BMK88] D.R. Boggs, J.C. Mogul, und C.A. Kent. Measured capacity of an Ethernet: myths and reality. In *Proc. SIGCOMM '88 Symposium on Communications Architectures and Protocols*, Band 18, Seite 222–34, August 1988.

Based on measurements of an actual implementation, the authors show that for a wide class of applications, Ethernet is capable of carrying it's nominal bandwitdth of useful traffic, and allocates bandwidth fairly.
Hauptsächliche Vorlage für die Verifikation des Ethernetmodells.

[Bor88] Petra Borowka. Hierarchische Modellierung eines SUN-Clusters mit dem Modellierungs- und Analysetool HIT. Diplomarbeit, Uni Dortmund, Informatik IV, Februar 1988.

[Bra88] P. Bratley. *Algorithmics: Theory and practice.* 1988.

[Bro87] R.G. Brookshire. Rating the IBM compatibles. *BYTE*, 12(12):193–6,198, 1987.

[BS87] H. Beilner und F.-J. Stewing. Concepts and techniques of the performance modelling tool, HIT. *Discrete Event Simulation and Simulation and Operations Research. Proceedings of the European Simulation Mulitconference*, Seite 38–43, 1987.

[Bul82] W.G. Bulgren. *Discrete System Simulation*, Kapitel 5,6,7, Seite 119ff. Publisher Hall, 1982.

Simulation, Experimentation and Verification. Generating Pseudo-Random Numbers.

[Bux87] W. Bux. Modeling token ring networks – a survey. *Messung, Modellierung und Bewertung von Rechensystemen: 4. GI/ITG Fachtagung, Erlangen, West Germany, 29. Sept. - 1.Okt 1987*, Seite 192–221, 1987.

[Bux89] W. Bux. Modeling token ring networks – a survey. *Informatik - Fachberichte 205. Kommunikation in verteilten Systemen. Grundlagen, Anwendungen, Betrieb.*, Seite 192–221, 1989.

[Buz76] J.P. Buzen. Fundamental Operational Laws of Computer System Performance. *Acta Informatica*, 7(2):167–182, 1976.

[BW75] J.W. Boyse und D.R. Warn. A Straight forward Model for Computer Performance Prediction. *Computing Surveys*, 7(2), 1975.

[CB89] P. Callahan und B. Bradley. New Token Ring versus Ethernet: Counterpoint. *Data Communications*, 18(1):127–34, Januar 1989.

[CL83] E.J. Coyle und B. Liu. A matrix representation of CSMA/CD networks. *IEEE International Conference on Communications: Integrating Communication for World Progress*, 2:745–8, Juni 1983.

[Com88] Douglas Comer. *Internetworking with TCP/IP*. Prentice Hall, 1988.

Principles, Protocols, and Architecture. Überblick über die wichtigsten RFCs. Sehr gute Einführung in die Internet-Architektur.

[Con88] A.E. Conway. Performance modeling of multi-layered OSI communication architectures. *Submitted to a Conference / ICC 89*, 1988.

Warteschlangenmodelle für Protokollschichten.

[DG88] W. Dobosiewicz und P. Gburzyński. Ethernet with segmented carrier. In *Proceedings of IEEE Computer Networking Symposium*, Seite 72–78, Washington, DC, April 1988.

[DG90] W. Dobosiewicz und P. Gburzyński. Performance of Piggyback Ethernet. In *IEEE IPCCC'90*, Seite 516–522, Scottsdale, AZ, März 1990.

[DGR90] W. Dobosiewicz, P. Gburzyński, und P. Rudnicki. An Ethernet-like CSMA/CD protocol for high speed bus LANs. In *IEEE INFOCOM'90*, Seite 238–245, 1990.

[dM89] G. de Montmollin. The trend is going towards distributed PC networks. *Sysdata (Schweiz)*, 20(3):67,69–70, März 1989.

[DMM88] S. Dravida, M.J. Master, und C.H. Morton. A Method to Analyze Performance of Digital Connections. *IEEE Trans. Commun.*, 36(3):298–305, 1988.

[dSeSM87] Edmundo de Souza e Silva und Richard R. Muntz. Approximate Solutions for a Class of Non-Product Form Queueing Network Models. *Performance Evaluation*, 7:221–242, 1987.

[Dus88] Klaudia Maria Dussa. Messen, Auswerten und Vergleichen von Meßwerten verteilter Anwendungen. Diplomarbeit, Uni Erlangen-Nürnberg, Informatik IV, 1988.

Vergleich von Munix-RPC und TCP.

[ea79] J. Moura et al. Evaluation of collision control algorithms in Ethernets. *Proceedings of the 6th Data Communications Symposium*, 1979. Cat. Nr. 79CH1405-0.

[EDF87] P. Ein-Dor und J. Feldmesser. Attributes of the performance of central processing units: a relative performance prediction model. *Communications of the ACM*, 30(4):308–317, April 1987.

[ET88] C. Evequoz und C. Tropper. On Reassembly Delay in Packet Switching Networks. *Computer Networks and ISDN Systems*, 15:1–25, 1988.

[Far88] B.N. Farah. Local area networks - is there a standard ? *J. Comput. Inf. Syst.*, 28(4):20–3, 1988.

[Fer78] Domenico Ferrari. *Computer Systems Performance Evaluation*. Prentice Hall, 1978.

Model Calibration and Validation.

[Fis73] G.S. Fishman. *Concepts and Methods in Discrete Event Digital Simulation*. John Wiley & Sons, 1973.

[Fis78] G.S. Fishman. *Principles of Discrete Event Simulation*. John Wiley & Sons, 1978. (FH 8495).

Autoregressive Method.

[FKLA87] V.S. Frost, M.S. Kashefipour, W. Laure, und R. Ananthanpilla. A combined LAP-D/CSMA-CD simulation model for the study of local area networks. *IEEE International Conference on Communications '87: Communications - Sound to Light. Proceedings.*, 3:1305–10, 1987. Cat. No. 87CH2424-0.

[FM77] W.R. Franata und K. Maly. An Efficient Data Structure for the Simulation Event Set. *Communications of the ACM*, 20(8):596–602, August 1977.

[Fra87] Klaus Franke. Zum Verhalten lokaler Netze vom CSMA/CD-Typ bei großem Verkehrsaufkommen. *Wissenschaftliche Zeitschrift der TU Karl-Marx-Stadt*, 29(4):498–503, 1987.

[FWLS88] V.S. Frost, Jr. W.W. LaRue, und K.S. Shanmugan. Efficient techniques for the simulation of computer communications networks. *IEEE Journal on selected Areas in Communications*, 6(1):130–45, Januar 1988.

[GKS+85] K. Görgen, H. Koch, G. Schulze, B. Struif, und K. Truöl. *Grundlagen der Kommunikationstechnologie: ISO-Architektur offener Kommunikationssysteme.* Springer, Berlin, Heidelberg, 1985.

[Gla89] B. Glass. The token ring. *BYTE*, 14(1):363–8,370,372,374,376, Januar 1989.

[GM80] P. Gburzyński und J. Maitan. An object-oriented configurable simulator for low-level communication protocols. Technical report tr 90-8, University of Alberta, Edmonton, Alberta, Canada, März 1980.

[Gon85] T.A. Gonsalves. Performance Characteristics of 2 Ethernets: an Experimental Study. *Proceedings of the ACM SIGMETRICS Conf. on Modeling and Measurement*, Seite 78–86, 1985.

[GR87] P. Gburzyński und P. Rudnicki. A better-than-Token protocol with bounded packet delay time for ethernet-type LAN's. In *Symposium on the Simulation of Computer Networks*, Seite 110–117, Colorado Springs, Co., August 1987. IEEE.

[GR88a] P. Gburzyński und P. Rudnicki. A Simulation Model of a CSMA/CD Communication Channel. *Submitted to INFOCOM '89*, Seite 1–14, Juli 1988.

[GR88b] P. Gburzyński und P. Rudnicki. Modeling low-level communication protocols: A modular approach. In *Proceedings of the 4th International Conference on Modeling Techniques and Tools for Computer Performance Evaluation*, Seite 89–108, Palma de Mallorca, Spain, 1988. Universitat de les Illes Balears.

[GR89a] P. Gburzyński und P. Rudnicki. LANSF—a modular system for modelling low-level communication protocols. In R. Puigjaner und D. Potier, Herausgeber, *Modeling Techniques and Tools for Computer Performance Evaluation*, Seite 77–93. Plenum Publishing Company, 1989.

[GR89b] P. Gburzyński und P. Rudnicki. *The LANSF Protocol Modeling Environment, version 2.0.* The University of Alberta, Department of Computing Science, TR 89-19, Edmonton, 1989.

[GR89c] P. Gburzyński und P. Rudnicki. A note on the performance of ENET II. *IEEE Journal on Selected Areas in Communications*, 7:424–427, April 1989.

[GR89d] P. Gburzyński und P. Rudnicki. On formal modelling of communication channels. In *IEEE INFOCOM'89*, Seite 143–151, 1989.

[GR89e] P. Gburzyński und P. Rudnicki. A virtual token protocol for bus networks: Correctness and performance. *INFOR*, 27:183–205, 1989.

[GR90a] P. Gburzyński und P. Rudnicki. Modelling of reactive systems in SMURPH. In *Proceedings of the European Simulation Multiconference*, Seite 661–667, Erlangen-Nuremberg, W-Germany, Juni 1990.

[GR90b] P. Gburzyński und P. Rudnicki. On executable specifications, validation, and testing of MAC-level protocols. In E. Brinksma, G. Scollo, und Vissers C. A., Herausgeber, *Proceedings of the 9th IFIP WG 6.1 Int. Symposium on Protocol Specification, Testing, and Verification, June 1989, Enschede, The Netherlands*, Seite 261–273. North-Holland, 1990.

[GR91a] P. Gburzyński und P. Rudnicki. LANSF: a protocol modelling environment and its implementation. *Software Practice and Experience*, 21(1):51–76, Januar 1991.

[GR91b] P. Gburzyński und P. Rudnicki. Object-oriented simulation is SMURPH: A case study of dqdb protocol. In *Proceedings of 1991 Western Multi Conference on Object-Oriented Simulation*, Seite 12–21, Anaheim, California, Februar 1991.

[GT88] T.A. Gonsalves und F.A. Tobagi. On the performance effects of station locations and access protocol parameters in Ethernet networks. *IEEE Trans. Commun.*, 36(4):441–9, April 1988.

[Gus87] R. Gusella. The Analysis of Diskless Workstation Traffic on an Ethernet. Bericht UCB/CSD 87/379, Computer Science Division, Uni. California, Berkeley, 1987.

[GW89] O. Gihr und M. Weixler. Messung der Datenverkehrsprofile in lokalen Netzen. *Informatik - Fachberichte 205. Kommunikation in verteilten Systemen. Grundlagen, Anwendungen, Betrieb.*, Seite 861–877, 1989.

[Hau86] B.M. Hauzeur. A model for naming, Addressing, and Routing. *ACM Transactions on Office Information Systems*, 4(4):293–311, Oktober 1986.

[HC87] S.E. Hauser und M.A. Crocker. A performance study of an Ethernet-based image transmission protocol. *Proceedings of the 12th Conference on Local Computer Networks*, Seite 86–94, 1987. Cat. No. 87CH2445-5.

[Hen77] James O. Henriksen. An Improved Events List Algorithm. *Proceedings of the 1977 Winter Simulation Conference*, Seite 547–557, 1977.

An Adaptive version of an indexed list algorithm.

[Hen83] James O. Henriksen. Event List Management - A Tutorial. *Proceedings of the 1983 Winter Simulation Conference*, Seite 543–551, 1983.

[Hic88] P.R. Hicks. Throughput and delay performance measurement of packet switched data networks. *Aust. Telecommun. Res. (Australia)*, 22(2):23–37, 1988.

[HKW88] M.B. Hidalgo, W.S. Kelly, und T.N. Washburn. Verification / validation of performance simulation using ANSI X3.102 measurements. *IEEE Netw.*, 2(6):15–16,23–28, November 1988.

[HL82] H.D. Huges und L. Li. Simulation model of an Ethernet. *Computer Performance (GB)*, 3(4):210–17, Dezember 1982.

[HL83] H.D. Huges und L. Li. A performance study of ethernet. *Proceedings of the CMG XIII International Conference, San Diego, CA, USA, 14-17 Dec. 1982*, Seite 79–85, 1983.

Simple discrete event simulation of Ethernet. Different network loads and backoff schemes.

[HL87] Ching-Tarng Hsieh und Simon S. Lam. Two Classes of Performance Bounds for Closed Queueing Networks. *Performance Evaluation*, 7:3–30, 1987.

[HL88] E. Heck und R. Lange. An aggregated model of CSMA/CD as an example for runtime reduction in LAN-simulation. *Proceedings: Workshop on the Future Trends of Distributed Computing Systems in the 1990s*, Seite 51–6, 1988. Cat. No. 88TH0228-7.

[HM82] B. Hawe und M. Marathe. Performance of a simulated Ethernet programming enviroment. *Electro/82 Conference Record*, Seite 21–4/1–8, Mai 1982.

[IEE85] Local Area Networks: Token-Passing Bus, Std 802.4. Bericht, ANSI/IEEE, 1985.

[IEE88a] Local Area Networks: Introduction, Std 802.1. Bericht, ANSI/IEEE, 1988.

[IEE88b] Local Area Networks: Logical Link Control, Std 802.2. Bericht, ANSI/IEEE, 1988.

[IEE88c] Local Area Networks: CSMA/CD, Std 802.3. Bericht, ANSI/IEEE, 1988.

Der Ethernet-Standard.

[IEE89] Local Area Networks: Token Ring Access Method and physical Layer Specifications, Std 802.5. Bericht, ANSI/IEEE, 1989.

[IM88] M. Ilyas und H.T. Mouftah. Simulation tools for computer communication networks. *GLOBECOM '88. IEEE Global Telecommunications Conference and Exhibition - Communications for the Information Age*, 3:1702–6, 1988.

[ISO88] ISO7498: Basic Reference Model for Open Systems Interconnections (CCITT Recommendation X.200). Bericht, ISO/CCITT, 1988.

[ISO89] Status of OSI (and related) Standards 6/1/89. *Computer Communication Reviews*, 19(3), Juli 1989.

[JM88] J. Jackman und D.J. Medeiros. A graphical methodology for simulating communication networks. *IEEE Trans. Commun.*, 36(4):459–64, April 1988.

[Jon86] D.W. Jones. An Empirical Comparison of Priority-Queue and Event-Set Implementations. *Communications of the ACM*, 29(4):300–311, April 1986.

Vergleich von Linearer Liste, Implicit Heap, Leftist Tree, Two-List Event-Set, Henriksen's Event Set, Binomial Queue, Pagode, Skew Heap, Spaly Tree, Pairing Heap.
Es gibt keinen universell optimalen Event-Set Algorithmus; Bottom Up Skew Heaps sind unter den meisten Voraussetzungen gut einsetzbar.

[KB88] R. Kuruppillai und N. Bengston. Performance analysis in local area networks of interconnected token rings. *Computer Communications*, 11(2):59–64, April 1988.

[Kel86] K.H. Kellermayr. *Lokale Computernetzwerke - LAN*. Springer, Wien, New York, 1986.

[Khe88] Naim A. Kheir. *Systems Modelling and Computer Simulation*. Marcel Dekker Inc, 1988.

[Kin89] B. Kindel. How fast is fast ? *BYTE*, 14(2):251–4, Februar 1989.

Measuring a computer's performance is not as simple as users would like.

[Kle76] Kleinrock. *Queuing Systems*, Band I + II. Wiley, 1976.

Ausführliche Darstellung der Warteschlangentheorie.

[Kle87] J.P.C. Kleijnen. *Statistical Tools for Simulation Practioners*. Marcel Dekker Inc, 1987.

[Knu73] D.E. Knuth. *The Art of Computer Programming*, Band 3, Sorting and Searching. Addison-Wesley, 1973.

[Kob78] H. Kobayashi. *Modeling and analysis. An introduction to system performance evaluation methodology*. Addison-Wesley, 1978. Aus der Lehrbuch-Sammlung.

[KP86] R.M. Keller und P. Panangaden. Semantics of digital networks containing intermediate modules. *Distributed Computing*, 1:235–245, 1986.

[Lan83] H. Langendörfer. *Rechnerverbund*, Band 5 der Reihe *Informatik-Skripten*. Technische Universität Braunschweig, 1983.

[Lan86] H. Langendörfer. *Messen, Modellieren, Simulation*, Band 16 der Reihe *Informatik-Skripten*. Technische Universität Braunschweig, 1986.

[Lan88] Rainer Lange. Ein Werkzeug zur LAN-Modellierung. Diplomarbeit, Uni Dortmund, Informatik IV, Februar 1988.

Verschiedene Betriebssystem- und Benutzermodelle, Aggregierung von Zugriffsverfahren und Übertragungsmedien (in HIT).

[Lav83] Stephen S. Lavenberg, Herausgeber. *Computer Performance Modeling Handbook*. Notes and Reports in Computer Sience and Applied Mathematics. Academic Press, 1983.

[LB87] C.E. LaBarre und P.J. Brusil. Architecture for evaluating LAN traffic and performance. *Proceedings of the 12th Conference on Local Computer Networks*, Seite 109–16, 1987. Cat. No. 87CH2445-5.

[LC79] A.M. Law und J.S. Carson. A Sequential Procedure for Determining the Length of a Steady-State Simulation. *Operations Research*, 27(5):1011–1025, 1979.

[LG87] U. Lübbe und O. Gihr. Messung von Verkehrsprofilen in Lokalen Netzen. *DATACOM*, unknown volume(11-12):88–94, 1987.

[LH88] M.S. Leuner und J.L Hammond. A monitoring system for an Ethernet installation. *Proceedings of the Twentieth Southeastern Symposium on System Theory*, Seite 160–4, 1988. Cat. No. 88CH2553-6.

[Lit85] D. Litzba. *Auswertung von Simulationsdaten mittels autoregressiver Modelle*, Band 203. Uni Dortmund, 1985.

[LK82] A.M. Law und W.D. Kelton. *Simulation Modelling and Analysis*. McGraw - Hill, 1982.

[LMB88] J.B. Lewoc, L.Z. Misiura, und L. Blach. Performance evalution for computer networks. *Adv. Model. Simul.*, 14(2):37–43, 1988.

[LO89] P.A.W. Lewis und E.J. Orav. *Simulation Methodology for Statisticans, Operations Analysts, and Engeneers (Volume 1)*, Band 1 der Reihe *Statistic/Probability*. Wadsworth & Brooks/Cole, 1989. (FN 9724 (1)).

[LS88] P. Liggesmeyer und L. Sturm. Augmented petri nets - a specification tool for complex applications. *Angewandte Informatik*, 30(6):252–8, Juni 1988.

[LSS89a] D. Litzba, M. Sczittnick, und F.-J. Stewing. *Eine Online-Update-Auswertungskomponente auf Basis autoregressiver Modelle*. Uni Dortmund, Informatik IV, April 1989.

Auswertung von Ausgaben von BORIS und HIT.

[LSS89b] D. Litzba, M. Sczittnick, und F.-J. Stewing. Yet another simulation output analysis algorithm: the autoregressive, online-update evaluation technique of the modelling tool, HIT. *(to be) presented at the 3rd European Simulation Conference, Edinburgh*, September 1989.

Simulation Run Length Control.

[LZ86] E.D. Lazowska und J. Zahorjan. File Access Performance of Diskless Workstations. *ACM Transactions on Computer Systems*, 4(3):238–268, August 1986.

[LZGS84] E.D. Lazowska, J. Zahorjan, G.S. Graham, und K.C. Sevcik. *Quantitative System Performance. Computer Systems Analysis Using Queueing Network Models*. Prentice Hall, Inc., 1984.

[Mac87] M.H. MacDougall. *Simulating Computer Systems. Techniques and Tools*. Computer Systems Series. MIT, 1987.

[MC90] B. Müller-Clostermann. Modelle von Rechensystemen - Zuverlässigkeit, Leistung, Funktion. Skript einer Vorlesung an der TU Braunschweig, 1990.

Zuverlässigkeitsnetze, Markov-Ketten, Petri-Netze.

[MCF87] M.A. Marsan, G. Chiola, und A. Fumagalli. Timed Petri net model for the accurate performance analysis of CSMA/CD bus LANs. *Computer Communications (UK)*, 10(6):304–12, Dezember 1987.

[Mee86] F. Meelamkavil. *Computer Simulation and Modelling*. John Wiley & Sons, 1986.

[Mel87] A.S. Melamed. Performance of Unix-based network file systems. *IEEE Micro (USA)*, 7(1):25–38, Februar 1987.

[MFW79] J.A.B. Moura, J.A. Field, und J.W. Wong. Evaluation of Collision Control Algorithms in Ethernets. *Proceedings of the 6th Data Communications Symposium*, 1979. CH1405-0/79/0000-0082.

[MHO89] J.W. Münch, L. Hofman, und R. Ostermann. Durchsatzmessungen auf einem Breitbandnetz. *Eingereicht zur Veröffentlichung für die Proceedings des 8. GI-Fachgesprächs über Rechenzentren am 02. / 03. März 1989 in Düsseldorf*, 1989.

[ML84] M. Melman und M. Livny. Models of the Ethernet protocol using DISS language. *4th Jerusalem Conference on Information Technology (JCIT)*, Seite 89–98, Mai 1984. Cat. No. 84CH2022-2.

[Mol85] Michael K. Molloy. Collision Resolution on the CSMA/CD Bus. *Computer Networks and ISDN Systems*, 85(?)(9):209–214, 1985.

[MS81] W.M. McCormack und R.G Sargent. Analysis of Future Event Set Algorithms for Discrete Event Simulation. *Communications of the ACM*, 24(12):801–812, Dezember 1981.

[MT88] M. Murata und H. Takagi. Two-layer modeling for local area networks. *IEEE Trans. Commun.*, 36(9):1022–34, September 1988.

[OHdNB87] S. Ong, E.C. Hu, A. Martin de Nicolas, und R.A. Baluch. A performance evaluation of microprocessors. *1987 SID International Symposium. Digest of Technical Papers*, Seite 408–11, 1987.

[OJH84] P.J.P. O'Reilly und Jr. J.L Hammond. An efficient simulation technique for performance studies of CSMA/CD local networks. *IEEE J. Sel. Areas Commun. (USA)*, SAC-2(1):238–49, Januar 1984.

[OlWmK88] M.T. Ozsu, Kok lung Wong, und Tse men Koon. System modeling and analysis using Petri nets. *Syst. Anal.-Model.-Simul. (East Germany)*, 5(1):3–25, 1988.

[O'R87] P. O'Reilly. Burst and Fast Packet Switching: Performance Comparisons. *Computer Networks and ISDN Systems*, 13:21–32, 1987.

[Par85] P. Parizek. Experience and results in simulation modeling of computer networks. *Systems Analysis and Simulation 1985. Proceedings of the 2nd International Symposium*, 1:441–4, August 1985.

[Pet91] Stefan Petri. simuLan – Ein Werkzeug zur Simulation lokaler Netze. Diplomarbeit, TU Braunschweig, 1991.

[Pid89] Michael Pidd. *Computer Modelling for Discrete Simulation.* John Wiley & Sons, 1989.

[PM90] S.R. Pakkam und C.N. Manikopoulos. Performance evaluation of the Consumer Electronic Bus. *IEEE Transactions on Computer Electronics*, 1990. (In press.).

[PP88] A. Pattavina und G. Pacifici. Performance Evaluation of Ring Networks Supporting a Packet Voice Service. *Computer Networks and ISDN Systems*, 15:31–45, 1988.

[Pre88] L. Press. Benchmarks for LAN performance evaluation. *Communications of the ACM*, 31(8):1014–17, August 1988.

[PS87] K. Prasad und A. Singhal. Simulation of ethernet performance based on single server and single queue model. *Proceedings of the 12th Conference on Local Computer Networks*, Seite 74–85, 1987. Cat. No. 87CH2445-5.

[PsT88] Gee-Swee Poo und Tiow seng Tan. Performance comparison of PC LAN file servers - an experimental study. *Computer Communications*, 11(2):71–9, April 1988.

[Ram87] K. Ramamritham. Channel Characteristics in Local-Area Hard Real-Time Systems. *Computer Networks and ISDN Systems*, 13:3–13, 1987.

[Ram88] G. Ramamurthy. An analytical model for UNIX systems. *AT&T Technical Journal*, Seite 86–99, September 1988.

[RFC81] Transmission control protocol. Bericht, for Defense Advanced Research Project Agency by Information Sciences Institute, University of Southern California, September 1981.

[RS88] K.M. Rege und B. Sengupta. Response Time Distribution in a Multiterminal Computer with Terminal Traffic. *Performance Evaluation*, 8:41–50, 1988.

[SA85] B.W. Stuck und E. Arthurs. *A computer and communications network analysis primer*. Prentice Hall Software Series. Prentice Hall, Inc., 1985.

Nette Einführung in die analytische Modellierung einfacher Systeme, einschließlich einfacher Modelle von LANs - aber mehr auch nicht.

[Sar87] R.G. Sargent. An overview of verification and validation of simulation models. *1987 Winter Simulation Conference Proceedings*, Seite 33–9, 1987.

[SC81] C.H. Sauer und K.M. Chandy. *Computer Systems Performance Modelling*. Prentice Hall, 1981.

[Sch82a] B. Schmeiser. Batch size effects in the analysis of simulation output. *Operations Research*, 30(3):556–568, 1982.

[Sch82b] B. Schmidt. Computer science and the general theory of models - an introduction. *Angewandte Informatik*, 24(1):35–42, Januar 1982.

[Sch82c] B. Schmidt. Die Bestimmung von Konfidenzintervallen in der Simulation stochastischer zeitdiskreter Systeme. *Elektronische Rechenanlagen*, 24(3):118–124, 1982.

Vergleich verschiedener Methoden: Independent Replication, Batch Means, Regenerative, Spectrum, Autoregression.

[Sch85] B. Schwarmer. Application of Computer aided tools in system analysis using modelling techniques. *Angewandte Informatik*, 27(6):225–31, Juni 1985.

Über die Tools COPE und USPE.

[SD91] Ralf Schmidt-Dannert. simuLan : Modellierung und Simulation lokaler Netzewerke. Diplomarbeit, TU Braunschweig, 1991.

[Sed88] Robert Sedgewick. *Algorithms*, Kapitel 11 - Priority Queues, Seite 145–162. Addison-Wesley Series in Computer Sience. Addison-Wesley Publishing Company, Inc., second edition Auflage, 1988.

Das Kapitel "Priority Queues" diente als Vorlage für die Implicit Heap - Implementation der Eventliste.

[SH80] J.F. Shoch und J.A. Hupp. Measured Performace of an Ethernet Local Network. *Communications of the ACM*, 23(12):711–721, Dezember 1980.

[Sha88] J.G. Shanthikumar. On some queueing models of time division multiple access communication system. *Performance Evaluation*, 8(4):289–304, August 1988.

[SM83] C.H. Sauer und E.A. MacNair. *Simulation of Computer Communication Systems*. Prentice Hall, 1983.

[Spa88] S. Spanier. Design and implementation of a LAN monitoring tool. *Computer Communications*, 11(2):85–9, April 1988.

[Sri88] M.M. Srinivasan. An approximation for mean waiting times in cyclic server systems with nonexhaustive service. *Performance Evaluation*, 9(1):17–33, November 1988.

[SSSB86] T. Simon, O. Spaniol, und J. Suppan-Borowka. Performance measurements of different protocol layers in a LAN enviroment. *EFOC/LAN 86 Proceedings.*, Seite 198–204, 1986.

[ST83] Daniel Dominic Sleator und Robert Endre Tarjan. Self-Adjusting Binary Trees. *Proceedings of the ACM SIGCAT Symposium on Theory of Coumputing*, Seite 235–245, 1983.

Die Vorlage für die BUSH-Implementation der Eventliste.

[ST86] Daniel Dominic Sleator und Robert Endre Tarjan. Self-Adjusting Heaps. *SIAM Journal on Computing*, 15(1):52–69, Februar 1986.

[Str89] N. Strole. Inside Token Ring Version II, according to Big Blue. *Data Communication*, 18(1):117–25, Januar 1989.

[Swe78] H.D. Swetman. Hybrid Simulation Models of Computer Systems. *Communications of the ACM*, 21(9):718–723, September 1978.

[Tan87] A.S. Tanenbaum. *Operating Systems*. Prentice Hall, 1987.

[Tan89] A.S. Tanenbaum. *Computer Networks*. Prentice Hall, 2. Auflage, 1989.

[TB84] A. Thomasian und P. Bay. Analytic solution of an integrated performance model of a computer communication network with window flow control. *Computer Communication Reviews*, 14(2):225–32, 1984.

[TGR88] P. Tran-Gia und T. Raith. Performance analysis of finite capacity polling systems with nonexhaustive service. *Performance Evaluation*, 9(1):1–16, November 1988.

[TH79] T.A. Tobagi und V.B. Hunt. A performance analysis of carrier sense multiple access with collision detection. *Proceedings of the Loacal Area Communication Network Symposium*, 1979.

[TM87] H. Takagi und M. Murata. Output processes of slotted persistent CSMA and CSMA/CD systems. *Computer Network ISDN Systems*, 13(4-5):333–46, 1987.

[TTH88] T. Takine, Y. Takahashi, und T. Hasegawa. An approximate analysis of a buffered CSMA/CD. *IEEE Trans. Commun.*, 36(8):932–41, August 1988.

[Ulf86] H.E. Ulfers. PACKSIM – a personal computer implementation of a packet switched network simulation model. *Proceedings of the 1986 Winter Simulation Conference*, Seite 769–775, 1986.

[Vid88] V.P. Vidomenko. Criterial evaluation of network topologies. *Autom. Control Comput. Science*, 22(4):27–33, 1988.

[Wil77] N.C. Wilhelm. A General Model for the Performance of Disk Systems. *Journal of the Association for Computing Machinery*, 24(1):14–31, Januar 1977.

[WMF80] J.W. Wong, J.A.B. Moura, und J.A. Field. Hierarchical modelling of local area computer networks. *NTC '80. IEEE 1980 National Telecommunications Conference, Houston*, Seite 37.1/1–7, 1980.

[WS88] A.C. Weaver und C.F. Summers. The IEEE token bus - A performance bound on GM MAP. *IEEE Trans. Ind. Electron.*, 35(1):13–17, Februar 1988.

Glossar

Ein *Rechnernetz* besteht aus einzelnen *Netzknoten*, die durch ein *Netz* verbunden sind.

Ein Netzknoten wird durch eine *Knotenbeschreibung* beschrieben, das Netz durch eine *Netzbeschreibung*.

Ein Netzknoten umfaßt mehrere (nämlich vier) *Protokollschichten* (Nr. 2 - 5), das Netz besteht aus einer Protokollschicht (Nr. 1).

Ein *Protokollschichtmodell* modelliert eine Protokollschicht, es wird durch ein *Protokollschichtmodul* realisiert (im C–Programm).

Die Protokollschichtmodelle für Protokollschicht Nr. 1 werden auch *Netzmodell* genannt, das Protokollschichtmodul dazu *Netzmodul*. Die Protokollschichtmodelle für Protokollschicht Nr. 5 werden auch als *Knotenmodell* bezeichnet, die Protokollschichtmodule dazu als *Knotenmodul*e (diese Schicht modelliert die Vorgänge "im Inneren des Knotens").

Das Rechnernetz insgesamt wird durch eine *Modellbeschreibung* beschrieben.

Die *Konfiguration* einer Simulation enthält neben der Modellbeschreibung auch Angaben über die Simulationsumgebung (Zuordnung von Ausgabekanälen, Variation von Parametern der Modellbeschreibung, Beschränkung von Resourcen).

Ein *Programmlauf* beginnt mit dem Aufruf von simuLan, und endet wenn simuLan terminiert.

Während eines Programmlaufes werden kein, ein oder mehrere *Simulationsläufe* durchgeführt, abhängig u.a. davon, was in der Konfiguration über die Variation von Parametern angegeben wird.

Die Peer-to-Peer- Kommunikation erfolgt in Protokollschicht 1 mittels *Frame*s, in Protokollschicht 2 mittels *Paket*en, in Protokollschicht 3 mittels *Datagramm*en und in Protokollschicht 5 mittels *Nachricht*en.

Frame, Paket, Datagramm, Nachricht werden allgemein als *Protocol Data Unit* (*PDU*) bezeichnet.

Ein *JOB* beschreibt im *K1*-Knotenmodell einen Prozeß, der durch einen Benutzerbefehl erzeugt und aktiviert wird. Er wird durch eine Struktur *K1_JOB* beschrieben, die an einem `EVENT` angehängt ist. In dieser Struktur ist zu jedem Zeitpunkt vermerkt, welche Betriebsmittel der `JOB` noch benötigt.

Gesamtindex

Typenindex

Variablenindex

N

O

P

R

S

T

Z

Funktionenindex