INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Diplomarbeit

Modulare Objektstruktur für eine Rechnernetz-Simulation

Florian Schmidt

Aufgabensteller:	Prof. Dr. Claudia Linnhoff-Popien
Betreuer:	Mang Li Markus Garschhammer

Abgabetermin: 24. Oktober 2002

INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Diplomarbeit

Modulare Objektstruktur für eine Rechnernetz-Simulation

Florian Schmidt

Aufgabensteller:	Prof. Dr. Claudia Linnhoff-Popien
Betreuer:	Mang Li Markus Garschhammer

Abgabetermin:	24. Oktober 2002

Hiermit versichere ich, dass ich die vorliegende Diplomarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 24. Oktober 2002

. . . .

(Unterschrift des Kandidaten)

Zusammenfassung

Diese Diplomarbeit ist Teil des BMBF Projektes "VIP - Virtuelles Informatik Praktikum", welches in Kooperation des Lehrstuhls Kommunikationssysteme und Systemprogrammierung v. Prof. Dr. Claudia Linnhoff-Popien der Ludwig-Maximilians-Universität München, des Lehrstuhls für Informatik IV v. Prof. Spaniol der RWTH Aachen, des Lehrstuhls für Berufs- und Wirtschaftspädagogik v. Prof. Michelsen der RWTH Aachen und der Firma Cisco Systems durchgeführt wird.

Ziel dieses Projektes ist es, StudentenInnen eine interaktive, verteilte Lernumgebung zur Verfügung zu stellen, um kooperatives Arbeiten zwischen StudentenInnen bzw. zwischen StudentenInnen und BetreuernInnen zu fördern. Um die Funktionalität dieser Kommunikations- bzw. Lernplattform zu erweitern, sollen typische Aufgaben eines Informatikpraktikums nachgebildet und in virtueller Form den StudentenInnen angeboten werden. Für diesen Zweck müssen real existierende Objekte und Abläufe in einem Simulator nachgebildet werden. Diese beiden Komponenten - Kommunikations- und Simulationsumgebung - bilden den Kern des Projektes.

Diese Diplomarbeit beschäftigt sich nun speziell mit dem Aufbau einer objektorientierten Basisstruktur für eine Rechnernetzsimulation, basierend auf dem DaSSF-Simulationskern. DaSSF ist eine C++ Implementierung der Scalable Simulation Framework API, welche in einem DARPA Projekt entstanden ist. Für die Modularität und Erweiterbarkeit der Rechnernetzsimulation werden objektorientierte Technologien beim Aufbau der Basisstruktur angewandt. Die Grundkonzepte von Rechnernetzen müssen ebenso berücksichtigt werden. In der Diplomarbeit ist die Implementierung der Funktionalitäten der Basisobjekte eingeschlossen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einführung

1.1 Motivation

Schlagwörter und Konzepte wie “lebenslanges Lernen“, “lernende Organisation“ oder “virtuelles Lernen“ sind In-Worte neuer Kommunikations- und Informationssysteme. Die alleinige Weiterbildung von Lernenden in Seminaren, Vorlesungen oder Praktika kann den Ansprüchen einer sich kontinuierlich verändernden “lernenden Organisation“ allerdings nicht genügen. Eine Unterstützung der herkömmlichen Methoden, Lerninhalte zu vermitteln, kann durch virtuelles Lernen erfolgen.

Mittels moderner, vernetzter Computertechnik hat man die Möglichkeit, vorliegendes Wissen und Lerninhalte flexibel zeit- und bedarfsgerecht zur Verfügung zu stellen. Bereits seit Beginn der Entwicklung von Computertechnologien wird die Symbiose von Informations- und Kommunikationstechnologie mit Lernprozessen diskutiert und erforscht. Telelearning, als eine innovative und gegenwärtig stetig zunehmende Lern- und Weiterbildungsmethode, stellt besonders für Betriebe, die kein eigenes Personal zur Personalentwicklung vorhalten, eine interessante Möglichkeit dar, ihre MitarbeiterInnen unabhängig von aufwändig zu organisierenden Weiterbildungsseminaren am Arbeitsplatz oder in vernetzten Lernzentren weiterzubilden. Hochschulen haben zunehmend das vernetzte Lernen als Chance erkannt, um unabhängig von Ort und Zeit spezifische Lerninhalte einer breiten Zielgruppe zur Verfügung stellen zu können.

Besonders Studiengänge, basierend auf Kommunikations- und Informationssystemen, wie z.B. Informatik, die mit sich schnell verändernden Lerninhalten zu tun haben, kann virtuelles Lernen von Vorteil sein. Lernplattformen mittels Videoconferencing, Online-Chat, Wissensbibliotheken oder Simulationen wären Realisierungen, die virtuelles Lernen möglich machen würden.

Das **Virtuelle Informatik Praktikum** (**VIP**) ist ein Ansatz zur Realisierung eines solchen Systems. Teil dieser sich in der Entwicklung befindenden Simulations- und Lernumgebung ist ein virtuelles Rechnernetzpraktikum, welches zu bisherigen Praktika, basierend auf herkömmlichen Lernmethoden, eine Ergänzung sein soll.

1.2 Aufgabenstellung

Diese Diplomarbeit beschäftigt sich mit der Erstellung einer Objektstruktur für eine Simulationsanwendung. Im Fachbereich Informatik wird derzeit in einem Gemeinschaftsprojekt von zwei Universitäten eine multimediale, interaktive Lernumgebung realisiert. Dieses Projekt trägt den Titel "Virtuelles Informatik Praktikum (VIP)". Den StudentenInnen soll damit die Möglichkeit geboten werden, Praktika an realen Geräten und Komponenten in einer Netzwerkumgebung durch ein virtuelles Informatikpraktikum zu ersetzen. Darüber hinaus soll im Rahmen dieses Projektes auch die Form des kooperativen Arbeitens, sowohl zwischen StudentenInnen als auch zwischen StudentenInnen und BetreuernInnen, gefördert werden.

Diese Diplomarbeit beschäftigt sich mit einer C++-Implementierung des **Scalable Simulation Frameworks** (**SSF**) entwickelt am Dartmouth College. Dieser generische Simulationskern, als **Dartmouth Scalable Simulation Frameworks** (**DaSSF**) bezeichnet, ist speziell darauf ausgelegt, sehr große Netze durch Nutzung von Shared-Memory-Multiprozessorumgebungen und Clustersystemen mit hoher Simulationsgeschwindigkeit nachzubilden. Hauptaufgabe dieser Diplomarbeit ist der Aufbau einer objektorientierten Simulationsbibliothek für eine Rechnernetzsimulation, der sog. VIP-Library, die auf dem DaSSF-Framework aufsetzt. Diese VIP-Library soll dem Anwender die Möglichkeit bieten, selbst Simulationen definieren zu können und deren Konfiguration und Ablaufdynamik festlegen zu können. Die Steuerung und Konfiguration der Simulationen soll dem Anwender durch ein Simulationsinterface wesentlich erleichtert werden. Beim Benutzer sollte zur Nutzung der Simulation keinerlei Wissen über den darunter liegenden DaSSF-Kern vorausgesetzt werden, weder was dessen interne Struktur noch dessen Arbeitsweise betrifft.

Die Simulation sollte alle Topologien in Rechnernetzen mit den unterschiedlichsten Strukturen, wie z.B. Bus oder Stern, darstellen können. Komponenten wie Hubs, Switches oder Router sollen modellierbar sein. Darüber hinaus müssen auch die Standardmechanismen, z.B. Protokollkomunikation des OSI-Schichtenmodelles, abbildbar sein. Durch einen modularen Aufbau soll die leichte Erweiterbarkeit der VIP-Library sichergestellt werden.
In einem zweiten Schritt soll anschließend unter Verwendung dieser im ersten Teil der Diplomarbeit

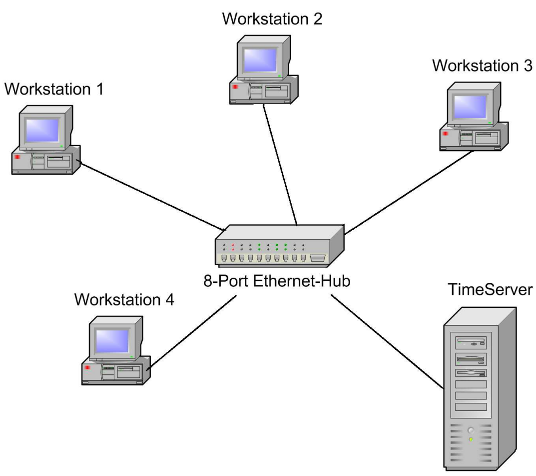

Abbildung 1.1: VIP_TIME Beispielsimulationsszenario

realisierten VIP-Library eine einfache Simulation erstellt werden, die deren Grundmechanismen veranschaulichen soll (Abschnitt 1.1). Dazu wird in einem einfachen Simulationsszenario, bestehend aus mehreren Endsystemen gekoppelt über eine Komponente, eine Clientanwendung mit einer Serveranwendung kommunizieren.

1.3 Vorgehensweise und Gliederung

Zu Beginn dieser Diplomarbeit wird in Kapitel 2 das Gesamtprojekt "Virtuelles Informatik Praktikum" eingeführt. Dabei wird der Zusammenhang zwischen dem VIP und der Diplomarbeit hergestellt und genauer erläutert.
In Kapitel 3 werden die für das Verständnis relevanten Grundlagen geschaffen. Es werden die wichtigsten Konzepte und Mechanismen aus dem Bereich Rechnernetzkommunikation erklärt, da Begriffe wie Schichtenmodelle, Protokoll oder ISO/OSI-Referenzmodell einen zentralen Punkt dieser Diplomarbeit bilden. Anschließend wird der DaSSF-Simulationskern, auf dem die zu erstellende VIP-Library basiert, beschrieben.
Nach diesen einführenden Erläuterungen beginnt mit Kapitel 4 der Hauptteil dieser Diplomarbeit. Es wird der komplette Entwicklungsprozess der Objektstruktur, beginnend mit der Anforderungsanalyse (Abschnitt 4.1), über die Designphase (Abschnitt 4.2) und Implementierungsphase (Abschnitt 4.3), bis hin zum Testen (Abschnitt 4.4) beschrieben.
In Kapitel 5 wird beschrieben, wie die im Laufe der Diplomarbeit entwickelte Simulationsumgebung auf einem Rechnersystem einzurichten ist.
Die Ergebnisse der Diplomarbeit werden dann in Kapitel 6 zusammengefasst und durch einen kleinen Ausblick auf die weiter vorstellbaren Entwicklungen und Möglichkeiten abgeschlossen.

Kapitel 2

VIP - Virtuelles Informatik Praktikum

Die normale Lehrtätigkeit an Hochschulen findet heute im wesentlichen in Form von Vorlesungen, Seminaren und Praktika statt. Der Schwerpunkt liegt hierbei auf der Informations- und Wissensvermittlung. Die Lerninhalte werden hierbei in der Regel textbasiert angeboten, allenfalls um statische Bilder und Grafiken ergänzt. Darüber hinaus sind mit der Zunahme von Bildungseinrichtungen, wie der Fernuniversität Hagen, zukünftig verstärkt neue Modelle für die Wissensvermittlung zu entwickeln, zu erproben und in die Praxis umzusetzen.

Instrumente wie

- *Teletutoring*
- *Gruppenorientiertes Teleconferencing*
- *Moderierte Foren*
- *Virtuelle Praktika*

werden hier verstärkt Bedeutung erlangen, da sich sowohl die Lehrenden als auch die Lernenden nicht mehr wie bei einem klassischen Studium am gleichen Ort befinden. Stattdessen sind die StudentenInnen teilweise über große räumliche Entfernungen von ihrer Universität getrennt. Der Zugang zu den Lehrangeboten und Lerninhalten erfolgt über die heute verfügbaren Kommunikationsinfrastrukturen wie **Integrated Services Digital Network** (**ISDN**) bzw. **Digital Subscriber Line** (**DSL**). Diese Entwicklungen ermöglichen auch neue Lehrangebote und neue Formen von Praktika seitens der Universitäten.

Nachdem auch die Komplexität von Tele- und Datenkommunikationssystemen selbst immer mehr zunimmt, stellen sich dadurch bedingt neue Herausforderungen für die im Studiengang Informatik angebotenen Lehrveranstaltungen. Lassen sich die theoretischen Grundlagen noch sehr gut mit konventionellen Lehrmethoden vermitteln, so gestaltet sich die Vertiefung des Wissens aus den Vorlesungen in Praktika zunehmend schwieriger. Dies liegt einerseits an der zunehmenden Komplexität und den schnellen Technologizyklen der beteiligten Kommunikationssysteme wie auch an der Verfügbarkeit entsprechend moderner Systeme in ausreichender Zahl für den Übungsbetrieb.
Selbst für ein Grundpraktikum im Netzwerkbereich ist heute bereits ein Netzwerkanalysegerät Voraussetzung, um Informationen über die verschiedenen, im Netzwerk transportierten Informationen erhalten zu können. Der Einsatz eines derartigen Gerätes kann jedoch in diesem Ausbildungsstadium noch nicht in Betracht gezogen werden. Da es also aus den vorgenannten Gründen nicht möglich ist, StudentenInnen an realen Systemen dieser Größenordnung auszubilden, müssen diese dafür ersatz-

weise in einer virtualisierten Form zur Verfügung gestellt werden.
Diese Simulation der Systeme muss dem Benutzer einen explorativen Umgang mit der simulierten Umgebung ermöglichen. Simulationsumgebungen heutigen Umfangs leisten dies derzeit jedoch nicht. Aus dem Funktionsumfang der Simulationsumgebung sollen jeweils nur diejenigen Funktionen dem Praktikanten angeboten werden, die dem Wissensstand angemessen und zum Erreichen des Lernzieles notwendig sind.
Auf diese Weise ist mit einer schrittweisen Steigerung der Komplexität, der Nutzung, der selbständigen Steuerung und der entsprechenden Korrekturen im Rahmen der Simulationsumgebung die gezielte Heranführung des Lernenden an das Verständnis des Gesamtsystems didaktisch sehr effizient möglich.

Das **Bundesministerium für Bildung und Forschung** (**BMBF**) hat deshalb das "Projekt Virtuelles Informatik Praktikum (VIP)" ins Leben gerufen, um eines dieser vorgenannten neuen Lernelemente durch Zusammenarbeit von Universitäten und Herstellern zu realisieren.
Nachstehend nun eine kurze Darstellung der Beteiligten und des Projektumfanges.

2.1 Beteiligte Institute und Organisationen am VIP

Prof. Dr. Otto Spaniol
Lehrstuhl für Informatik IV
Prof. Dr. Uwe A. Michelsen
Lehrstuhl für Berufs- und Wirtschaftspädagogik
Rheinisch-Westfälischen Technischen Hochschule (**RWTH**) Aachen

Prof. Dr. Claudia Linnhoff-Popien
Institut für Informatik
Ludwig-Maximilians-Universität (**LMU**) München

Michael R. Kunzi
Sales Manager
Cisco Systems GmbH

2.2 Zielsetzungen des VIP

Ziel dieses Projektes ist es, den StudentenInnen durch eine multimediale, interaktive Simulations- und Lernumgebung einen experimentellen und explorativen Umgang mit komplexen technischen Systemen zu ermöglichen. In diesem konkreten Fall werden Datenkommunikationssysteme betrachtet. Im Rahmen dieses Projektes werden nun Konzepte und Werkzeuge entwickelt, um die typischen Aufgabenstellungen in einem Informatikpraktikum, wie Versuche mit IP-Infrastrukturen, in einer virtualisierten Form zur Verfügung zu stellen. Dazu werden die realen Systeme und Komponenten in einem

Simulator nachgebildet.

2.3 Einordnung der Diplomarbeit in das VIP

Diese Diplomarbeit beschäftigt sich im Rahmen des vorgenannten Projektes speziell mit dem Aufbau der Basisstruktur für eine Rechnernetzsimulation, die ursächlich auf dem Simulationskern DaSSF (Abschnitt 3.2.2), entwickelt am Dartmouth College (USA), basiert. Wie in den vorangestellten Abschnitten beschrieben, besteht das VIP aus einer interaktiven Lernumgebung als Frontend, gekoppelt mit einer Simulationsumgebung im Backend. Diese logische Trennung in Front- bzw. Backend ist in Abbildung 2.1 schematisch dargestellt.

Aktuelle Projekte an der RWTH Aachen befassen sich momentan mit der Entwicklung des Frontends. Bezogen auf den simulationsrelevanten Teil sollte dieses Frontend dem Anwender eine einfache Möglichkeit bieten, steuernd bzw. konfigurierend auf das Simulationsbackend zuzugreifen. Das Design sowie die Umsetzung dieser Schnittstelle zu den Anwendern, dem sog. Simulationsinterface (Abbildungen 2.1 und 4.1), ist Teil dieser Diplomarbeit.

Neben der Konzeption des Simulationsinterfaces ist die Erstellung einer Objektstruktur für das Simulationsbackend die Hauptaufgabe dieser Diplomarbeit. Alle Anwendungsobjekte (z.B. Hosts, Router, Protokolle, usw.) benutzerdefinierter Simulationen müssen aus dieser Objektstruktur aufgebaut werden können. Basis dieser Struktur ist der DaSSF-Simulationskern, welcher in Abschnitt 3.2 genauer erläutert wird. Im weiteren Verlauf dieser Diplomarbeit wird diese Objektstruktur als VIP-Library bezeichnet.

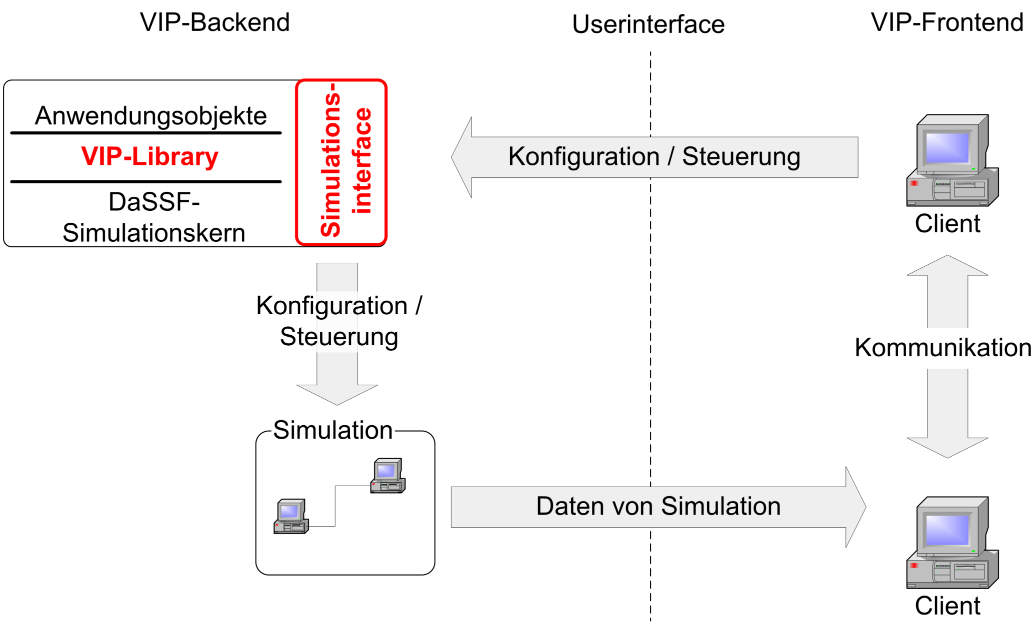

Abbildung 2.1: Einordnung der Diplomarbeit in das VIP

Kapitel 3

Grundlagen

Für den Entwurf und die Implementierung einer Klassenbibliothek für eine Rechnernetzsimulation werden verschiedene Grundlagen benötigt. Diese Grundlagen werden in den nächsten zwei Abschnitten geschaffen. Zunächst wird im ersten Abschnitt eine Einführung in die wichtigsten Aspekte einer Rechnernetzkommunikation, deren Mechanismen und Gemeinsamkeiten gegeben. Neben dieser theoretischen Einführung wird aber ein besonderes Augenmerk auf die in der realen Welt tatsächlich implementierten Rechnernetze gelegt.
Der zweite Abschnitt behandelt die DaSSF-Simulationsumgebung mit ihren Klassenbibliotheken. Es werden die fünf Simulationsklassen (Entity, inChannel, outChannel, Event und Process) detailliert behandelt und deren Verbindungen und Kommunikationsmechanismen erklärt, da diese den Kern der Simulationsbibliothek bilden.

3.1 Rechnernetzkommunikation

Rechnernetze bestehen aus autonomen, heterogenen und voneinander getrennten Endsystemen [HAN 99], die untereinander Informationen austauschen können. Rechnernetze besitzen eine physische und eine logische Struktur. Die physische Struktur definiert die Endsysteme, die Vermittlungeinrichtungen, den topologischen Aufbau und die Verbindungsmedien. Im Gegensatz dazu beschreibt die logische Struktur die Kommunikationsmechanismen. Diese strukturelle Basis wird auch Architekturmodell genannt. Moderne Rechnernetzarchitekturen basieren auf dem Strukturierungsprinzip der Schnittbildung.
Ausgehend vom ISO/OSI-Referenzmodell, werden die Grundkonzepte und Eigenschaften der logischen und physischen Struktur moderner Rechnernetze kurz erläutert.

3.1.1 Struktur des OSI-Modells

3.1.1.1 Instanzen

OSI zerlegt ein Kommunikationssystem in sieben Schichten. In jeder Schicht existieren Arbeitseinheiten, sog. Instanzen. Diese Instanzen stellen logische, parallele Prozesse in einer Schicht dar, welche

die Aufgaben der jeweiligen Schicht durchführen können (Abbildung 3.1). Eine Schicht eines Endsystems kann mehrere Instanzen enthalten. Die Implementierung einer Instanz kann in Hard- oder Software erfolgen.
Korrespondierende Instanzen derselben Schicht in verschiedenen Systemen werden Partnerinstanzen (Peer Entities) genannt. Aus dieser Struktur ergeben sich zwei Beziehungen zwischen Instanzen:

- Vertikale Beziehung
- Horizontale Beziehung

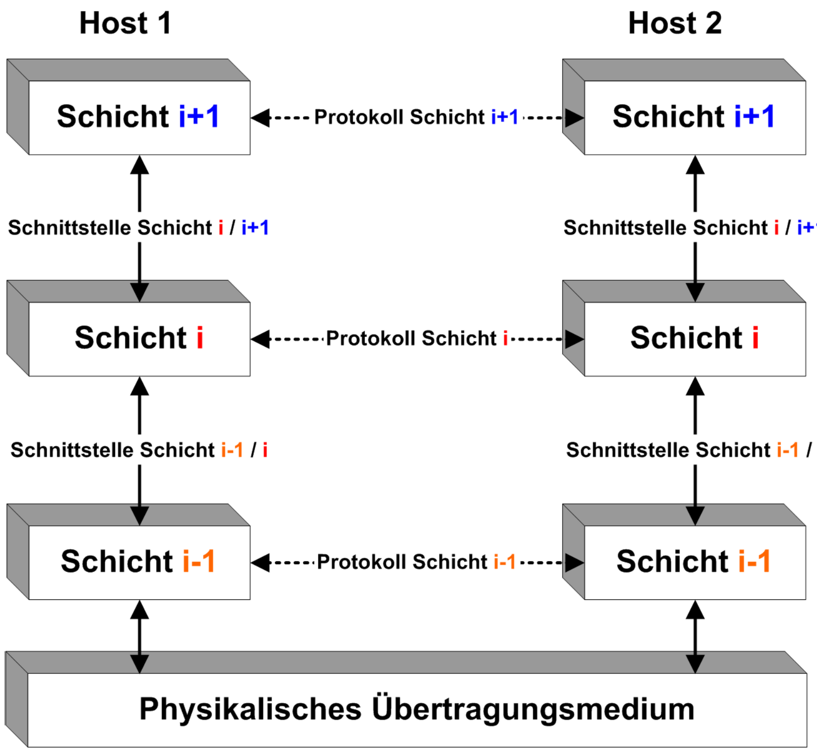

Abbildung 3.1: Protokollhierarchie - Schichtenmodell

3.1.1.2 Dienste (Vertikale Beziehungen)

Eine Schicht N liefert Instanzen der darüberliegenden Schicht N+1 eigene Leistung. Diese Leistung nennt OSI den Dienst (Service) der Schicht. Der Dienst wird über einen Dienstzugangspunkt, den sog. **Service Access Point** (**SAP**), der darüberliegenden Schicht angeboten [Tann 00]. Die Kommandos zur Inanspruchnahme von Diensten und die Ergebnismeldungen heißen Dienstelemente (Service primitives). Ein Dienst stellt hierbei einen Satz von Funktionen dar, die einem Benutzer von einem Anbieter (Provider) zur Verfügung gestellt werden. Eine Instanz der Schicht N nimmt also die Dienste der Schicht N-1 in Anspruch, ohne zu wissen, wie diese ihre Dienstleistungen erbringt (Prinzip des Information Hiding).
Grundsätzlich können Dienste in zwei Kategorien unterteilt werden:

Verbindungslose Dienste Der verbindungslose Dienst ist gekennzeichnet durch die Übertragung beziehungsloser Dateneinheiten, sog. Datagrammen. Da keine Verbindung aufgebaut wird, müssen alle gesendeten Nachrichten die volle Adresse der Partnerinstanz enthalten. Datagramme können unterschiedliche Wege zu ihrem Ziel zurücklegen. Aufgrund dessen ist die Auslieferung in richtiger Reihenfolge nicht gesichert.

Verbindungsorientierte Dienste Eine verbindungsorientierte Datenübertragung ist durch folgende Merkmale gekennzeichnet:

- Verbindungsaufbau, Datentransfer, Verbindungsabbau
- Eine N-Verbindungsendpunktkennung wird auf beiden Seiten eines Dienstzugangspunkts (SAP) zur Identifikation der N-Verbindung benutzt.

Verbindungsorientierte Protokolle haben Mechanismen zur Erkennung fehlender Blöcke (Vollständigkeit nicht gewahrt), duplizierter Blöcke (Eindeutigkeit nicht gewahrt) und außerhalb der Sequenz eintreffenden Informationsblöcke (Sequenz nicht gewahrt) sowie zur Beseitigung der entsprechenden Fehlerzustände. Sie gewährleisten also Vollständigkeit, Eindeutigkeit und Sequenz.

3.1.1.3 Protokolle (Horizontale Beziehungen)

Die zweite Art von Beziehungen sind die horizontalen Beziehungen zwischen Partnerinstanzen. Protokolle sind Regeln zur Kommunikation zwischen diesen Partnerinstanzen [Stev 98]. Typische Funktionen von Protokollen sind in Tabelle 3.1 zusammengefaßt.

Verbindungsmanagement	Aufbau, Abbau, Multiplexing
Datentransfer	Segmentierung, Wegewahl, Flußsteuerung
Fehlerbehandlung	Quittung, Fehlererkennung, Reihenfolgesicherung

Tabelle 3.1: Aufgaben von Protokollen

3.1.1.4 Datentransport

Ein Datenpaket durchläuft beim Absender nacheinander alle darunterliegenden Schichten bis es auf dem Übertragungsmedium transportiert werden kann. Dabei erhält es in jeder Schicht Protokollinformationen (Header, Trailer), die in der jeweils tieferen Schicht als Nutzdaten betrachtet werden und somit nicht verändert werden können. Auf Empfängerseite werden die Daten in jeder Schicht wieder von den jeweiligen Protokolldaten befreit, und an die darüberliegende Schicht weitergegeben (Abbildung 3.2).
Typische Protokollinformationen sind z.B. Absender- und Empfängeradresse, Dienstarten und viele andere, die für eine erfolgreiche Kommunikation unerlässlich sind.

3.1.1.5 Beschreibung der Schichten

Nachstehend werden kurz die einzelnen Schichten des OSI-Modells mit ihren wesentlichen Funktionalitäten dargestellt, da sie für ein Verständnis der Designentscheidungen der Diplomarbeit hilfreich sind.

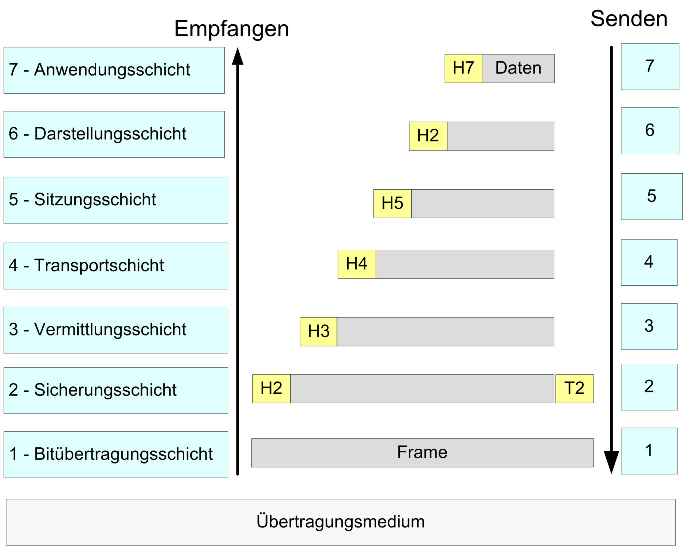

Abbildung 3.2: Datentransport im Schichtenmodell

Bitübertragungsschicht (Physical Layer)

Hauptfunktionalität: Transparente Übertragung von Bit-Sequenzen.

Dies ist die unterste Ebene des OSI-Modells. Sie stellt die Grundlage bzw. Voraussetzung für die Kommunikation dar. Vorwiegend geht es hier um Fragen der elektrischen Anpassung von Endeinrichtungen an Übertragungseinrichtungen und an das Übertragungsmedium. Sie ist für eine transparente Übertragung von Bit-Sequenzen zuständig. Vor einer Übertragung wird die Übertragungsart (Analog/Digital, Simplex/Duplex) festgelegt [Tann 00].

Sie beschreibt im Falle eines Rechnernetzes die Rechnernetzhardware einer Komponenten, bestehend aus:

- Physikalischem Übertragungsmedium: elektrisch, akustisch oder optisch
- Mechanischen Kenndaten: Steckerform, Pinanordnungen, ...
- Funktionalen und prozeduralen Parametern für die physikalische Verbindung: Bedeutung der Pins, ...

Sicherungsschicht (Link Layer)

Hauptfunktionalität: Festlegung einer gesicherten Übertragung auf einer Teilstrecke des Übertragungsweges.

Die Sicherungsschicht umfaßt die Aufgabe, in der physikalischen Schicht entstandene Übertragungsfehler zu erkennen und nach Möglichkeit zu korrigieren. Bekanntlich wird jeder physikalische Übertragungskanal statisch und systematisch mehr oder weniger stark gestört. Der Empfänger sollte folglich die Möglichkeit haben, jede Nachricht auf ihre formale Richtigkeit hin zu überprüfen.

Vermittlungsschicht (Network Layer)

Hauptfunktionalität: Festlegung des gesamten physischen Übertragungsweges der Netzverbindung.

Nachdem Schicht 1 die Existenz verwertbarer Nachrichten physikalisch sicherstellt und Schicht 2 für fehlerfreien Datenfluß zwischen zwei Punkten sorgt, wird in der dritten Schicht, der Vermittlungsschicht, die Art der Wegfindung einer Nachricht durch ein ganzes Rechnernetzwerk definiert, da es oft nicht genügt, Nachrichten lediglich zwischen zwei Punkten zu übertragen. Die Leistung der Netzwerkschicht besteht folglich in der Kombination der von den unteren Schichten bereitgestellten, gesicherten Übertragungsabschnitten zu einem optimalen Weg für die Daten.

Transportschicht (Transport Layer)

Hauptfunktionalität: Netzunabhängiger Transport von Nachrichten.

Schicht 4 ist für den netzunabhängigen Transport von Nachrichten zwischen zwei Endsystemen zuständig.
Funktionen der Transportschicht:

- Aufbau und Abbau von Ende-zu-Ende-Verbindungen
- Ende-zu-Ende-Flußsteuerung (Flow-Control)
- Ende-zu-Ende-Fehlererkennung und -behandlung

Sitzungsschicht (Session Layer)

Hauptfunktionalität: Festlegung der Sitzungssteuerung.

Wurden durch die bisher beschriebenen Schichten Transportfunktionen ausgeübt, so beschreiben die nächsthöheren Schichten Leistungen, die die Endteilnehmer einer Kommunikation selbst erbringen müssen, z.B. Auf- und Abbau der Sitzungen, Festlegung der Form des Datenaustausches, Dialogverwaltung usw. .

Darstellungsschicht (Presentation Layer)

Die nächst höhere Ebene des Schichtenmodells ist die Darstellungsschicht. Hier müssen die Datenformatierung, die Darstellung von Festkomma- und Gleitkommazahlen, die Festlegung von Steuerzeichen und die Festlegung der Datencodierung vorgenommen werden. Die Dateneingabe wird überwacht und Code- sowie Datenformatumwandlung durchgeführt.
Als standardisierte Dienste auf dieser Ebene sind das "Virtuelle Terminal" und die "Virtuelle Datei" repräsentative Beispiele. Hierbei werden Nachrichten in eine Form gebracht bzw. in einer Art gehandhabt, dass sie für Kommunikationspartner universell nutzbar sind.

Anwendungsschicht (Application Layer)

Die Anwendungsschicht, die höchste Schicht des Referenzmodells, stellt die Schnittstelle der Anwendungsprozesse zum Kommunikationssystem dar. Es handelt sich hierbei um allgemeine Hilfsdienste für die Kommunikation oder aber spezielle Kommunikationsdienste wie File Transfer, **Message Handling System** (**MHS**), etc. Der Anwendungsprozess hat ausschließlich über die Schicht 7 Zugang zum OSI-Kommunikationssystem.

3.1.2 Vermittlungskomponenten

Bisher sind in den Ausführungen nur Endsysteme betrachtet worden, die direkt miteinander gekoppelt waren. Umfangreichere Rechnernetze benötigen jedoch darüber hinaus Verbindungskomponenten, um es strukturieren zu können. Außerdem können mit Verbindungskomponenten Längenrestriktionen in LAN´s überwunden werden.

3.1.2.1 Repeater

Repeater sind Verstärkereinrichtungen, die auf OSI-Schicht 1 anzusiedeln sind. Sie koppeln zwei Segmente gleichen LAN-Types [HAN 99], um deren Längenrestriktionen aufzuheben. Elektrische Signale werden vom Repeater auf dem Eingang empfangen, regeneriert und zum Ausgang weitergeleitet. Repeater haben weder eine Filter- noch eine Wegefindungsfunktion, sodass sie daher nicht zur Strukturierung von Netzen eingesetzt werden können.

3.1.2.2 Hub

Hubs arbeiten auf der Schicht 1 des OSI-Modells. Sie haben reine Verteilfunktionalität und können als Multiport-Repeater angesehen werden. Ein Hub sendet empfangene Datenpakete an alle Ports weiter. Alle Stationen, die an einem Hub angeschlossen sind, teilen sich die gesamte Bandbreite mit der der Hub an ein Netzwerk angeschlossen ist. Durch die Verbindung (Kaskadierung) mehrerer Hubs lassen sich die Anzahl der möglichen Stationen erhöhen. Darüber hinaus sind heute aber auch leistungsfähigere Geräte verfügbar. U.a. gibt es heute auch sog. Sicherheits-Hubs, die sicherstellen, dass die Informationen an die Ziel-MAC-Adresse (**Medium Access Control** (**MAC**)) nur auf dem Port gesendet werden, an dem das entsprechende Gerät auch tatsächlich angeschlossen ist. Für alle anderen Stationen wird der Datenteil im Ethernet-Frame gelöscht.

3.1.2.3 Bridge

Eine Bridge ist ein Netzknoten mit deutlich mehr Intelligenz als ein Hub. Sie kann zwei oder mehrere physische Teilnetze miteinander verbinden. Sie arbeitet auf der OSI-Schicht 2 des OSI-Modells. Die dabei verwendete Technik ist etwas aufwändiger als bei Repeatern und erfordert in der Regel auch Software innerhalb der Bridge. Da eine Bridge auf der OSI-Schicht 2 arbeitet, sind über dieser Schicht liegende Protokolle für sie transparent. Das heißt, dass es für eine Bridge völlig gleichgültig ist, ob z.B. TCP/IP- oder NetBIOS-Protokolle eingesetzt werden. Die verschiedenen Protokolle können alle mit der gleichen Bridge übertragen werden [HAN 99].
Innerhalb dieser Geräteklasse gibt es zwei unterschiedliche Typen:

- Die Standard-MAC-Layer-Bridge, die in der unteren Teilschicht der zweiten OSI-Schicht arbeitet. Damit diese Bridge verwendet werden kann, muss das Zugriffsverfahren der beiden Teilnetze identisch sein. Daher wäre eine Verbindung eines Ethernet-LAN´s mit einem Token-Ring-Netzwerk mit dieser Art von Bridge nicht möglich.
- Die LLC-Bridge, in der die MAC-Adressen auf das zweite Subnetz umgesetzt, sowie die Datenpakete in das neue Format umgewandelt werden.

Eine Bridge teilt das Gesamtnetzwerk in einzelne Kollisionsdomänen auf. Dementsprechend unterliegen damit Bridges, im Gegensatz zu Repeatern, keinerlei Beschränkungen in Bezug auf die Kaskadierung. Jedes Netzsegment ist demgemäß ein eigenständiges Subnetzwerk. Eine Bridge arbeitet nach dem sog. "Store-und Forward"-Mechanismus. Gemäß diesem Mechanismus hört eine Bridge alle Datenpakete des einen Netzsegmentes mit, hinterlegt sie in ihrem Zwischenspeicher und wertet die Ziel-MAC-Adresse aus. Die Ziel-MAC-Adresse wird nun in der Adresstabelle des entsprechenden LAN-Segmentes gesucht. Ist sie dort eingetragen, wird das Datenpaket verworfen, da das Ziel im gleichen Netzsegment liegt. Ist die Zieladresse dort nicht vertreten, werden die Tabellen der anderen Netzsegmente mit der Zieladresse verglichen und das Datenpaket an den zuständigen Port weitergeleitet (Forwarding). Eine Bridge kann darüber hinaus das komplette Datenpaket mit der im Header enthaltenen Prüfsumme überprüfen und all die Datenpakete, die aus irgendwelchen Gründen fehlerhaft sind, verwerfen. Diese Funktionalität verringert zwar den Durchsatz, belastet aber dafür andere Netzsegmente nicht mehr mit fehlerhaften Paketen.

3.1.2.4 Switch

Switches sind Netzkomponenten, die je nach Funktionalität ("und Preis") ab der OSI-Schicht 2 bis hinauf zur Schicht 7 arbeiten können. Klassisch im Einsatz sind aber heute i.d.R. die sog. Layer-2-Switches.

Ein Layer-2-Switch arbeitet auf der Schicht 2 des OSI-Modells. Ein Switch schaltet direkte Verbindungen zwischen den an seinen Ports angeschlossenen Geräten bzw. Netzsegmenten. Auf dem so geschalteten Kommunikationsweg steht die gesamte Bandbreite des Netzwerkes zur Verfügung. Empfängt ein Switch ein Datenpaket, so sucht er in seinem Speicher nach der Zieladresse (MAC) und schickt dann das Datenpaket nur an diesen Port. Parallel zu dieser Verbindung können zwei andere Geräte, die an andere Switch-Ports angeschlossen sind, gleichzeitig mit voller Netzwerkgeschwindigkeit miteinander kommunizieren. Im Idealfall kann ein n-Port-Switch n/2 Datenpakete gleichzeitig vermitteln, wenn das seine interne Vermittlungskapazität nicht übersteigt. Die MAC-Adressen lernt ein Switch während des Betriebes. Bei den Switches unterscheidet man grundsätzlich zwei Typen, die durch unterschiedliche Leistungsfähigkeit gekennzeichnet sind:

- Store-and-Forward-Switches:
  Diese Switches können als eine Mischform zwischen Switch und Bridge bezeichnet werden. Sie bieten jedoch eine erheblich höhere Bandbreite als eine normale Bridge. Durch den Store-and-Forward-Prozeß tritt eine relativ hohe Latenzzeit auf, da das ganze Datenpaket zwischengespeichert und überprüft werden muss, bevor es weitergeleitet wird. Diese Art von Switch kann auch als sog. "High-Performance-Multiport-Bridge" bezeichnet werden. Der Vorteil dieser Methode liegt in der Aussonderung von defekten Datenpaketen, die zur Verminderung der Netzlast vom Switch verworfen werden. Der Nachteil der relativ hohen Latenzzeit wird hierdurch etwas ausgeglichen.
- Cut-Through-Switches:
  Bei einem Cut-Through-Switch wird im Gegensatz zum "Store-and-Forward-Switch" der Weiterleitungsvorgang bereits gestartet, wenn die sechs Byte lange Destination-Adresse vom Switch-Controller ausgewertet ist. Das restliche Datenpaket wird nicht mehr analysiert. Die Verzögerungszeit reduziert sich bei diesem Prinzip natürlich erheblich.

3.1.2.5 Router

Große Netzwerke wie das Internet bestehen aus unzähligen kleineren Teilnetzwerken die miteinander verbunden sind. Die Verbindung dieser verschiedenen Netze wird durch spezielle Systeme, den sog. Routern, hergestellt. Diese Systeme haben vor allem die Aufgabe, Daten zwischen Rechnern in verschiedenen Netzen auf möglichst günstigen Wegen weiterzuleiten. Ein Router ermöglicht es, Netzwerke unterschiedlicher Protokolle und Architekturen miteinander zu verbinden. Erst mit diesen Komponenten ist es möglich, sehr große und vermaschte Netzwerke wie das Internet aufzubauen.

Router arbeiten, anders als Bridges, auf der OSI-Schicht 3. Über sog. Routingtabellen, die zur schnellen Datenweiterleitung im Speicher gehalten werden, entscheidet ein Router, wie ein Datenpaket weitergeleitet werden soll. Diese Routingtabellen enthalten als wesentliche Information die bekannten Rechnernetzadressen, die Verbindungsarten in andere Rechnernetze, die Weginformationen zu anderen Routern und die Verbindungskosten dorthin. Routingtabellen halten jedoch nicht den gesamte Weg bis zum Zielsystem. Vielmehr kennt der einzelne Router nur die nächste Zwischenstation (engl. next hop) auf dem Weg zum Ziel. Das kann ein weiterer Router oder der Zielrechner sein. Unter Zuhilfenahme dieser Informationen entscheidet der Router im Fall eines jeden Datenpaketes über den weiteren Weg, den dieses Datenpaket nehmen soll.
Bevor jedoch ein Router ein Paket weiterleiten kann, muss er zunächst den weiteren Weg durch das Netz zum Zielnetz bestimmen. Das geschieht mit Hilfe spezieller Routingprotokolle, wie z.B. RIP oder OSPF.

3.1.2.6 Gateway

Ein Gateway kann im Gegensatz zu Routern zwei Netzwerke miteinander verbinden, die zueinander inkompatibel sind. Sie stellen einen gemeinsamen (virtuellen) Knoten dar, der zu beiden Netzen gehört und den netzübergreifenden Datenverkehr abwickelt. Ein Gateway muss dazu die Protokolle und die Adressierung in das jeweilige Zielrechnernetz umsetzen. Gateways behandeln auf beiden Seiten unterschiedliche Protokolle bis zur Schicht 7.

3.1.3 Übertragungsmedien

Übertragungsmedien dienen zur Verbindung der Endsysteme und der Vermittlungskomponenten. Sie sind in zwei Kategorien aufgeteilt [HAN 99]:

- **Leitergebundene Übertragungsmedien**
  Diese Kategorie umfaßt alle Kabeltypen. Hierzu gehören Lichtwellenleiter und elektrische Leiter.
- **Leiterungebundene Übertragungsmedien**
  Diese Kategorie umfaßt alle Übertragungsmedien, welche nicht an einen Leiter gebunden sind. Hierzu gehören Techniken wie Funk-, Infrarot- oder Laserübertragung.

Diese Übertragungsmedien haben die Aufgabe Nachrichten in Form von Signalen zwischen zwei logisch verbundenen Systemen zu übertragen. Dabei können Nachrichten verfälscht, z.B. Nebensprechen innerhalb eines Leitungsbündels durch Induktion ("Crosstalk"), oder sogar verloren gehen. Übertragungsmedien unterscheiden sich stark in ihrer Anfälligkeit auf Störeinflüsse zu reagieren.

3.1.4 Netztopologien

Die Struktur des Zusammenschlusses mehrerer Systeme zu einem Netz wird als Topologie des Netzes bezeichnet.

Aus der Netztopologie selbst lassen sich bereits einige Leistungs- und Stabilitätsaussagen des Netzes ableiten:

- Möglichkeiten und Verhalten des Netzes hinsichtlich seiner Skalierungsfähigkeit
- Reaktion des Netzwerkes bei Ausfall einer Station oder einer Verbindungsleitung
- Anzahl der Verbindungsleitungen die ausfallen dürfen, ohne dass eine Station von der Kommunikation abgeschnitten wird
- Verwendbare Methoden zur Wegeermittlung (Routing-Algorithmen)
- Notwendiger Protokolloverhead für eine fehlerfreie Kommunikation

3.1.4.1 Bus-Topologie

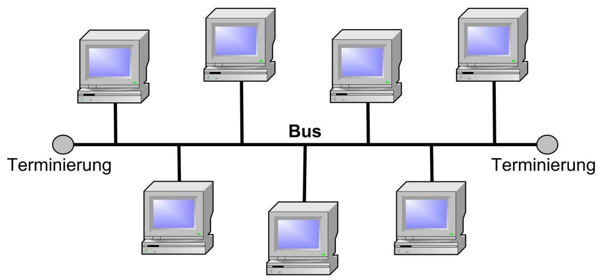

Abbildung 3.3: Bus-Topologie

Die Bus-Topologie (Abbildung 3.3) ist eine heute kaum mehr eingesetzte Methode für eine LAN-Vernetzung. Alle Stationen sind an ein gemeinsames Medium (shared LAN) angeschlossen und haben somit zu jeder Nachricht auf dem Bus direkten Zugriff. Dies war die klassische Topologie zu Beginn des LAN-Einsatzes (10base5 und 10base2). Die maximale Ausdehnung des Busses, sowie die Erweiterbarkeit um weitere Stationen, wird durch die verwendeten Zugriffsprotokolle und Kabel limitiert. Die Erweiterung des Busses bzw. die Hinzunahme weiterer Stationen ist mit einem kurzzeitigen Ausfall des Netzes verbunden. Der Ausfall einer einzelnen Station beeinträchtigt die Funktion des Netzes nicht. Eine Beschädigung des Busses oder der Ausfall der Terminierung an einem Ende bedingt jedoch zwangsläufig immer den Ausfall der gesamten Kommunikation unter den angeschlossenen Stationen.
Die beiden typischen Vertreter für eine Bus-Verkabelung ist das Ethernet in seiner Ausprägung als 10base5 und 10base2.

3.1.4.2 Ring-Topologie

Bei der Ring-Topologie (Abbildung 3.4) ist jedes System mit den beiden jeweiligen Nachbarsystemen verbunden. Ein System empfängt die im Ring übertragenen Nachrichten, prüft ob diese für sie bestimmt sind und leitet sie dann an den Nachbarn weiter. Da der Ausfall einer Station damit den Ausfall des gesamten Netzes bedeutet, kann dem entgegengewirkt werden. Z.B. im Falle der Token-Ring-Topologie durch Ausgestaltung des Ringes als Sternstruktur über ein Wire-Center mit aktiven Komponenten zur automatischen Deaktivierung und Überbrückung ausgefallener Geräte bzw. Leitungen.

3.1.4.3 Stern-Topologie

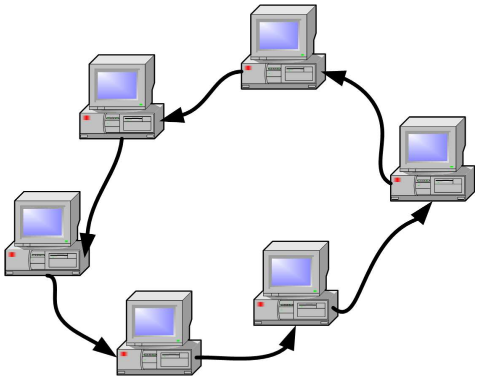

Abbildung 3.4: Ring-Topologie

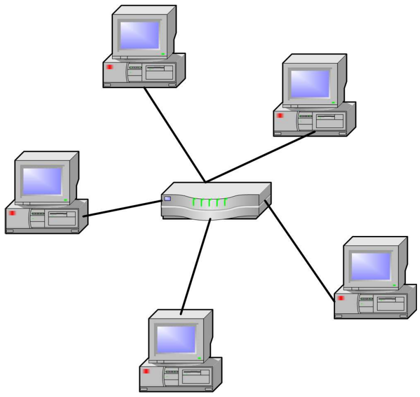

Abbildung 3.5: Stern-Topologie

Die Stern-Topologie (Abbildung 3.5) ist, wie aus dem Namen bereits entnehmbar, durch einen zentralen Knoten gekennzeichnet. Jede Station ist einzeln mit einer eigenen Verbindungsleitung mit diesem verbunden, sodass jegliche Kommunikation über diesen Knoten abgewickelt werden muss. Durch unterschiedliche Ausführung des zentralen Knotens können unterschiedliche Topologien nachgebildet werden. Wie vorher in Abschnitt 3.1.4.2 bereits ausgeführt, wird bei der Token-Ring-Infrastruktur der Ring in eine sternförmige Struktur überführt. Die Hauptvorteile dieser Struktur liegen in der einfachen Erweiterbarkeit des LANs, sowie seiner Stabilität in Hinblick auf den Ausfall einzelner Segmente. So wird bei Beschädigung einer Zuleitung zu einer Station nur diese darüber angeschlossene Station gestört, nicht jedoch das übrige LAN.

Der Hauptvorteil der Stern-Topologie ist jedoch systembedingt gleichzeitig auch ihr größter Nachteil. Ein Ausfall des zentralen Knotens bedeutet den Zusammenbruch der Kommunikation auf dem gesamten LAN. Deshalb ist dem Gesichtspunkt der redundanten Auslegung dieses Knotens besonderes Gewicht beizumessen. Als Beispiel für die aktuelle Stern-Topologie kann hier Ethernet, in seinen Ausprägungen 10baseT oder 100baseT, genannt werden.

3.1.4.4 Baum-Topologie

Die Ausdehnung von LAN-Infrastrukturen bei größeren Institutionen, wie z.B. einer Universität oder einem größeren Unternehmen, führt aus physikalischen Restriktionen, baulichen Beschränkungen (Kabelkanäle) und Kostengründen für die Installation der Infrastruktur zwangsläufig auf eine hierarchische Verkabelung. Diese Struktur hat die Form einer Baum-Topologie (Abbildung 3.6). Bedingt durch die Zunahme des Datenverkehrs zur Baumwurzel hin, ist der Einsatz unterschiedlicher physikalischer Übertragungstechnologien innerhalb der verschiedenen Bereiche des Netzes notwendig.

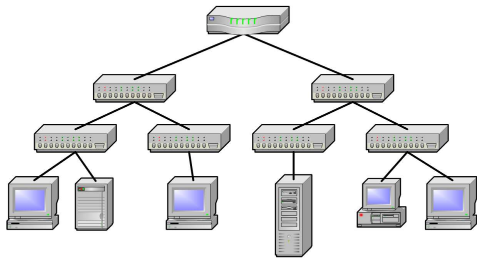

Abbildung 3.6: Baum-Topologie

3.2 Scalable Simulation Framework

Kommunikationssysteme sind in alle wichtigen Bereiche unseres Lebens vorgedrungen und bestimmen somit einen weiten Teil unseres Handelns. Ob Heimnetz oder globales Internet, wir sind stark abhängig von diesen Technologien. Die rasante Entwicklung und extreme Komplexität dieser Kommunikationssysteme erschwert das Verständnis der Zusammenhänge jener sehr stark.

Abschätzungen und Prognosen über das Verhalten eines komplexen globalen Rechnernetzes, wie das Internet bei Konfigurationsänderungen (z.B. Umstellung auf IPv6), sind kaum machbar. Aufgrund dessen wurde vom S3 Konsortium ein Projekt gegründet, dessen Ziel es war, eine Simulationsumgebungen zu schaffen, die vom Leistungspotential her fähig sein sollte, das Internet mit seiner ganzen Komplexität und Heterogenität zu modellieren und dynamisch zu simulieren. Millionen von Rechnern, Routern und anderen Rechnernetzkomponenten des Internets, die über eine Vielzahl von Protokollen miteinander kommunizieren, sollten modellierbar sein. Damit sollte ein genaues Abbild der realen Gegebenheiten ermöglicht werden, um Kommunikationssysteme besser zu verstehen und zu erforschen.

Aus diesen Bemühungen heraus entstand das SSF [Cowi 99], welches eine einheitliche Schnittstelle für parallele, diskrete ereignisorientierte Simulationen bietet. Teil dieses Framework ist die Spezifikation von Klassen und Methoden (SSF API), welche zur Beschreibung von Simulationen großer komplexer Systeme nötig sind. Diese Spezifikationen liegen in Java und C++ (Anhang A) vor.

3.2.1 SSF API Spezifikation

Die SSF API bestícht durch ihre Modularität und ihren kompakten Umfang. Sie spezifiziert die fünf Basisklassen [LuNi 01]:

- Event
- Entity
- inChannel
- outChannel
- Process

Hier ein Überblick[1] über die Grundkonzepte dieser Klassen [LuNi 01]:

[1]Für einen tieferen Einblick in die SSF API [LuNi 01].

Entity Die Hauptklasse der SSF API ist die Klasse *Entity*, welche logische Prozessobjekte in einer Simulation darstellt. Man kann sie als eine Art Behälter für alle an einer Simulation beteiligten Variablen auffassen. Jedes Entityobjekt kann Objekte der SSF-Klassen Process, inChannel und outChannel besitzen. Bezogen auf diese Objekte wird das Entityobjekt als sog. “Owner“ bezeichnet.

inChannel In SSF sind Kanäle die Verbindungsmedien zwischen den kommunizierenden Parteien. Jeder dieser Kanäle besteht aus zwei Endpunkten, einem Eingangskanal (sog. inChannel) und einem Ausgangskanal (sog. outChannel), welche durch unterschiedliche SSF-Objekte dargestellt werden. Da ein Endpunkt zu mehreren Kanälen gehören kann, sind auch Multicast-Verbindungen möglich. Das Zusammenschalten von Kanalendpunkten wird in SSF als Mapping bezeichnet. Dieses Mapping erlaubt nur die Zusammenschaltung von outChanneln zu inChanneln.
InChannels stellen das logische Ende einer Verbindung zwischen Entities dar (Abbildung 3.7). Auf Grund dessen können inChannels nur als Objekte eines Entities auftreten. Der inChannel ist somit ein Eingangskanal der Owner-Entity.
InChannels können mit mehreren outChannels anderer Entities gekoppelt (gemappt) werden.

outChannel Im Gegensatz zu inChannels stellen outChannels den logischen Ausgangspunkt einer Verbindung zwischen Entities dar (Abbildung 3.7). OutChannels können ebenfalls an mehrere inChannels gekoppelt werden. Das Prinzip des Event-Austausches in SSF läuft dann wie folgt ab:
Nachrichten, die auf einen outChannel gelegt werden, kommen anschließend bei allen dazu gemappten inChannels an.

Event Entities tauschen Informationen in Form von Nachrichten über deren gemappte Channels aus. Als Basisklasse dieser Nachrichten dient die Klasse Event.

Process Jedes Entity einer Simulation kann Process-Objekte enthalten, welche den dynamischen Aspekt eines Entities darstellen. Jeder Simulationsprozess wird durch ein Process-Objekt dargestellt. Startzeitpunkte sind unterschiedlich, da es mehrere Arten gibt, wartende Process-Objekte zu starten.

Insgesamt gibt es die nachfolgenden Arten Prozesse zu starten:

- Process-Objekte, die an einem inChannel auf neue Nachrichten warten und dann starten, wenn die Nachricht eingetroffen ist
- Process-Objekte, die für eine festgelegte Zeit warten und dann starten
- Process-Objekte, die für eine festgelegte Zeit warten und dann starten bzw. die auf in der Wartephase eintreffende Nachrichten mit einem Start reagieren

Der Umfang der SSF API ist extrem kompakt und modular gehalten.

3.2.2 Dartmouth Scalable Simulation Framework

Das Dartmouth Scalable Simulation Framework ist eine C++-Implementierung von SSF [LuNi 01]. Im Gegensatz zu anderen Simulationskernen ist DaSSF extrem performant, da das Hauptaugenmerk der Entwickler von DaSSF auf Simulationsgeschwindigkeit und guter Speicherausnutzung lag. Außer-

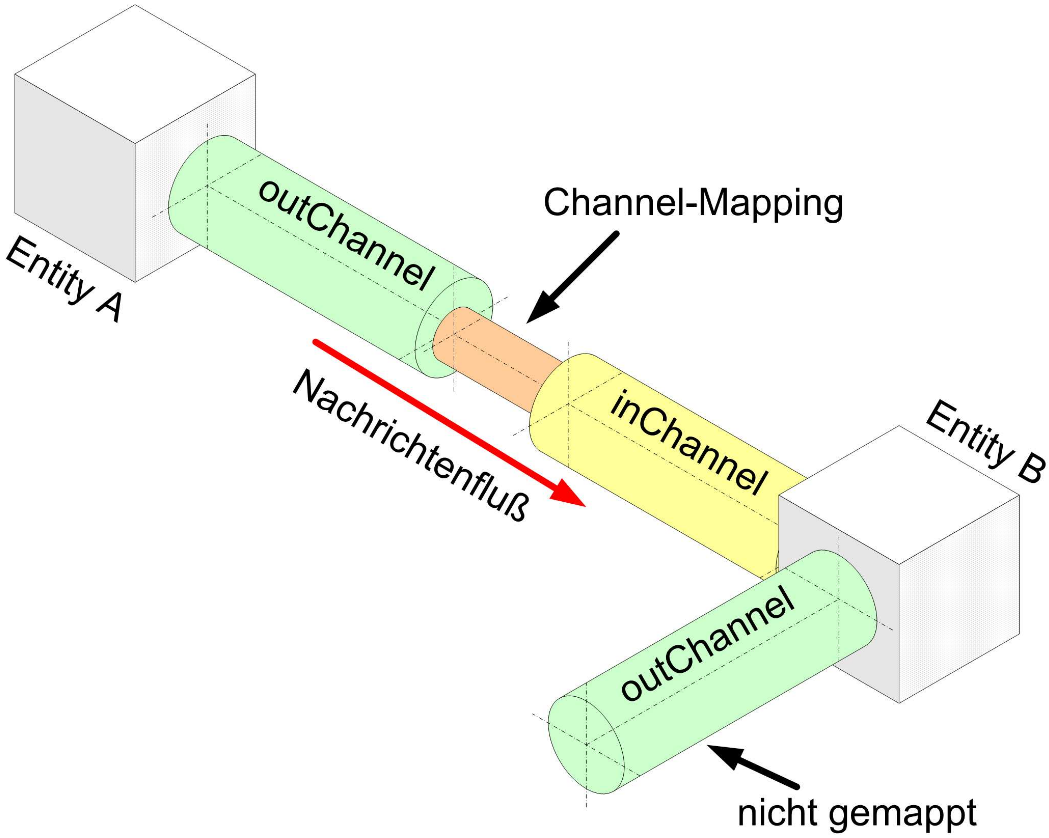

Abbildung 3.7: SSF - Channel-Mapping

dem ist DaSSF in der Lage, Simulationen parallel auf SMP-Umgebungen so wie auf Cluster-Systemen ablaufen zu lassen. Die Synchronisation der beteiligten Komponenten wird von DaSSF vor dem Benutzer verborgen. DaSSF läuft unter den meisten heute gängigen UNIX-Derivaten.

3.2.2.1 Simulationsphasen

Bei DaSSF wird ein Simulationslauf in vier verschiedene Phasen (Abbildung 3.8) unterteilt [LuNi 01]. Das Zusammenspiel dieser Phasen und deren zeitliche Abfolge ist für die Entwicklung DaSSF-basierter Simulationen extrem wichtig, da nur zu ganz bestimmten Zeiten Simulationsobjekte erstellt, bzw. zerstört werden dürfen. Nachstehend werden die Phasen einer DaSSF-basierten Simulation dargestellt:

Phasen einer DaSSF-basierten Simulation:

1. **Vorbereitung (Preparation Phase)**
   Diese Phase eines Simulationslaufes ist optional. Nur wenn der Anwender zu Beginn einer Simulation eine zu parsende DML-File (Abschnitt 3.2.2.2) angibt, um ein Simulationsszenario dynamisch zu generieren, wird diese Phase durchlaufen. Es werden alle durch die DML angegebenen Entities generiert. Während dieses Vorgangs werden außerdem die zu den Entities

gehörigen inChannels, outChannels und Processes erstellt und zueinander gemappt.

2. **Initialisierung (Initialization Phase)**
   Das Durchlaufen der Initialisierungsphase ist nur dann optional, wenn zuvor die Preparation Phase durchlaufen wurde.
   Die Initialisierungsphase wird nur dann durchlaufen, wenn der Anwender in seiner Simulation eine main-Funktion angibt. Diese main-Funktion dient dazu, Entities, inChannels, outChannels und Processes manuell, d.h. nicht mit Hilfe einer DML-File, zu generieren. Sonstige Aufgaben wie das Mappen von Channels oder das Anordnen von Entities zueinander, werden hier durchgeführt.

3. **Ablauf (Running Phase)**
   Diese Phase ist die eigentliche Simulation. Hier wird die vordefinierte Dynamik, d.h. der Austausch von Nachrichten über Kanäle mittels Prozessen durchgeführt. Während dieser Phase dürfen keine neuen Objekte generiert werden, da in der aktuellen Version des DaSSF-Simulationskerns kein dynamisches Generieren und Mappen von Entities und Channels implementiert ist.

4. **Nachbearbeitung (Wrapup Phase)**
   Diese Phase beginnt nach dem Erreichen der gewünschten Simulationsendzeit. Es werden alle aktiven Simulationsprozesse beendet und deren "wrapup"-Methoden aufgerufen. Diese Methoden sind dafür zuständig, statistische Daten über den Simulationsverlauf zu sammeln und sie am Ende der Simulation dem Anwender zur Weiterverarbeitung durchzureichen. Nachdem alle DaSSF-Prozesse beendet worden sind, wird die "Entity::wrapup"-Methode für jedes zuvor erstellte Entity aufgerufen. Danach wird die Simulation mittels eines "Entity::joinAll"-Aufrufes an den Anwender zurückgegeben. Der Anwender hat nun die Möglichkeit, die gewonnenen statistischen Simulationsdaten zu verarbeiten und darzustellen.

3.2.2.2 DML - Domain Modeling Language

In der Beschreibung der unterschiedlichen Phasen einer DaSSF-basierten Simulation kann man erkennen, dass es zwei verschiedene Methoden gibt Simulationsobjekte zu erstellen. Einerseits hat der Anwender die Möglichkeit Objekte direkt innerhalb einer main-Funktion zu erstellen, andererseits gibt es das Konzept der skriptbasierten Objektgenerierung. Als Skriptsprache wird in DaSSF die **Domain Modeling Language** (**DML**) benutzt [LuNi 01].
Zu Beginn einer Simulation werden drei DML-Dateien an einen DML-Interpreter übergeben.

Model-DML Diese Datei beschreibt alle an der Simulation beteiligten Objekte (Entities). Es werden deren Eigenschaften, Beziehungen und Abhängigkeiten definiert, z.B. die zu schaltenden Kanäle (Mapping).

Machine-DML Hier wird die Rechnerumgebung, auf der die Simulation ablaufen soll, beschrieben. Eigenschaften wie Betriebssystem, IP-Adresse oder Prozessoranzahl sind Teil dieser Beschreibung, damit die Simulation auf die beteiligten Rechnersysteme verteilt werden kann.

Runtime-DML Diese Datei gibt das Laufzeitverhalten, z.B. Start- und Endzeit der Simulation an.

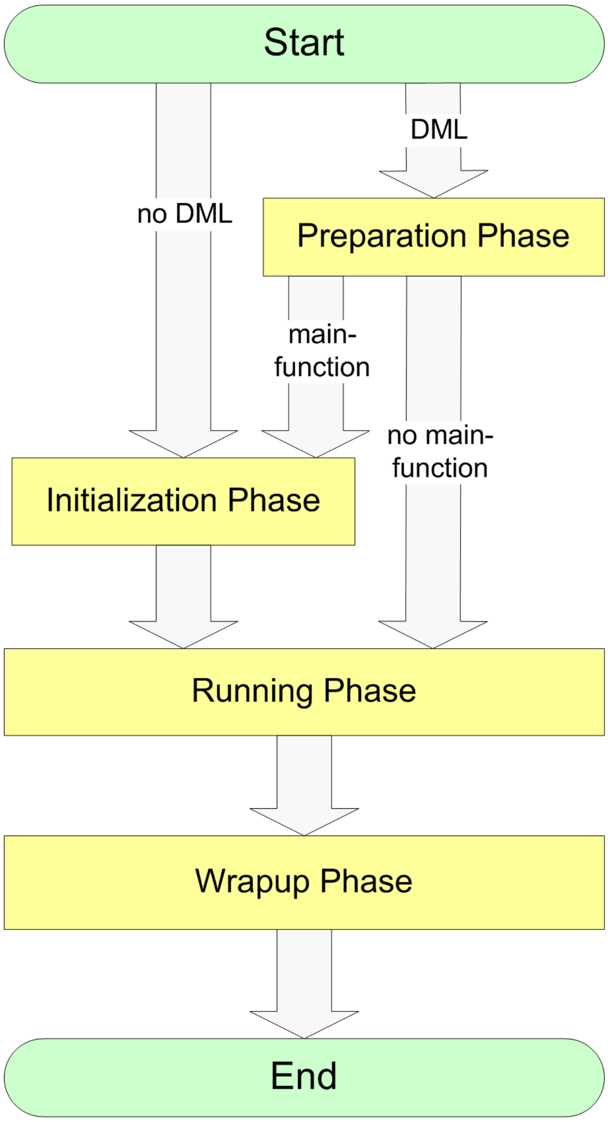

Abbildung 3.8: Überblick der Phasen einer DaSSF-Simulation

Die Aufgabe des Interpreters ist es, das Simulationsmodell, beschrieben durch Model-, Runtime- und Machine-DML, in Teilmodelle (DML-Dateien) aufzuteilen. Diese Teilmodelle werden dann auf die verschiedenen Rechnersysteme verteilt. Diese Rechnersysteme führen ebenfalls eine Analyse mittels DML-Interpreter durch, um das Teilmodell ggf. wiederum auf mehrere Prozessoren zu verteilen. Danach können auf allen Systemen die Simulationsobjekte generiert werden und die Simulation ist bereit gestartet zu werden.
Diese Ablaufsystemathik einer durch DML generierten Simulation wird in Abbildung 3.9 schematisch dargestellt.

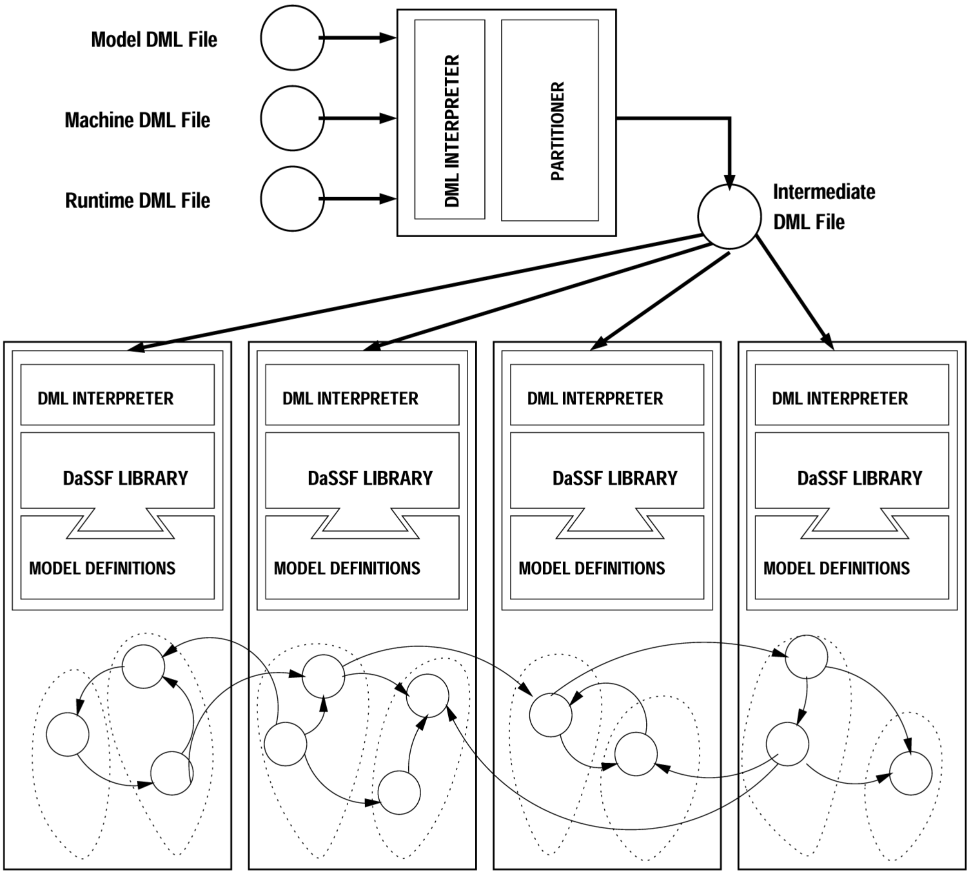

Abbildung 3.9: DML - Generierung einer DaSSF-Simulation (Graphik aus [LuNi 01])

3.2.2.3 Source-to-Source Translation

Um die gewünschte Performance, die Hauptziel bei der Entwicklung des DaSSF-Kerns war, zu erreichen, wurde ein DaSSF-eigener Threading-Mechanismus implementiert. Um diesen komplexen Mechanismus vor DaSSF-Anwendungsentwicklern zu verbergen, wurde das Konzept der "Source-to-Source Translation" erschaffen. Anwender können mittels regulärer C++-Kommentare spezielle Bereiche eines Programms festlegen, für die das DaSSF-eigene Threading angewandt werden soll. Durch Vorschalten eines Perl-Skriptes vor den eigentlichen Kompiliervorgang, werden diese Bereiche dann durch einen speziellen DaSSF-Quellcode ersetzt, welcher den Threading-Mechanismus umsetzt. Dieses Ersetzen wird als "Source-to-Source Translation" bezeichnet. Genauere Informationen zu diesem komplizierten Verfahren finden sie in [LuNi 01].

3.2.2.4 Erweiterungen des DaSSF-Standardsimulationkerns

Die dynamische Generierung von Simulationsobjekten und das damit verbundene Mapping der Kanäle ist ein notwendiges Feature, das der DaSSF-Standardsimulationkern nicht unterstützt. Um aber die Leistungsanforderungen, die eine Rechnernetzsimulation an einen Simulationskern stellt ein-halten zu können, musste der DaSSF-Simulationskern diesbezüglich leicht modifiziert bzw. erweitert werden. Dieser erweiterte Simulationskern wird in dieser Diplomarbeit als **Extended Dartmouth Scalable Simulation Framework** (**extDaSSF**) bezeichnet. Diesen Zusammenhang zwischen VIP-Library und extDaSSF- bzw. DaSSF-Simulationskern wird in Abbildung 4.1 schematisch dargestellt.

Kapitel 4

Entwicklungsprozess der Objektstruktur

Im Rahmen des traditionellen Software-Engineering-Prozesses wird ein Softwareprojekt in unterschiedliche Phasen aufgeteilt. Eine häufig verwendete Aufteilung stellt sich folgendermaßen dar:

- Produktspezifikation
- Design
- Implementierung
- Test

Diese Phasen werden während des Entwicklungsprozesses aufeinanderfolgend durchlaufen. Der Beginn einer neuen Phase ist nur erlaubt, wenn die vorausgegangene Phase abgeschlossen ist. Auf Grund dieser Eigenschaft wird diese Methode der Softwareentwicklung auch "Wasserfall Methode" genannt. Neben dieser Methodik existieren noch weitere Arten der Softwareentwicklung, z.B. "Spiral Methode", bei der ein Projekt ebenfalls in mehrere Phasen aufgeteilt wird [Müll 97]. Im Gegensatz zum ersten sequentiellen Ansatz ist dies ein zyklisches Programmentwicklungsverfahren. Beginnend mit einem Anfangsentwurf, dem sog. Prototypen, werden dann die Phasen so oft durchlaufen, bis durch sukzessives Verfeinern dieses Entwurfes die passende Abschlußform erreicht wird.

In dieser Diplomarbeit wurde der zweite Ansatz gewählt, da er einem natürlichen Entwicklungsprozess näher kommt, bei dem Phasen mehrfach durchlaufen werden können. Man nähert sich dem Ergebnis in mehreren Durchläufen an. Der Entwicklungsprozess dieser Diplomarbeit ist in die vier Projektstufen Anforderungsanalyse, Design, Implementierung und Test aufgeteilt. Das Endergebnis dieser vier Phasen, erreicht über das sog. "Spiralmodell" des Entwicklungsprozesses, wird in den folgenden Abschnitten genauer erläutert.

4.1 Anforderungsanalyse

Aufgabe der Anforderungsanalyse ist es, das zu erstellende Produkt mit seinen Fähigkeiten und Eigenschaften zu beschreiben. Es wird darin funktional festgelegt, **"WAS"** das Produkt können soll [Oest 01b]. In dieser Phase wird aber noch nicht über die Umsetzbarkeit nachgedacht. Das Ergebnis einer Anforderungsanalyse ist ein Pflichtenheft, in dem alle Anforderungen an den Funktionsumfang und das Verhalten einer Software schriftlich niedergelegt sind.

Wie in der Aufgabenstellung (Abschnitt 1.2) beschrieben, besteht der entwickelte Code aus zwei Hauptteilen, der VIP-Objektstruktur und einer Beispielsimulation, kurz als VIP_TIME bezeichnet. Dementsprechend wird die dazugehörige Anforderungsanalyse auch in zwei Bereiche aufgeteilt, die nun in den folgenden Abschnitten näher erläutert werden.

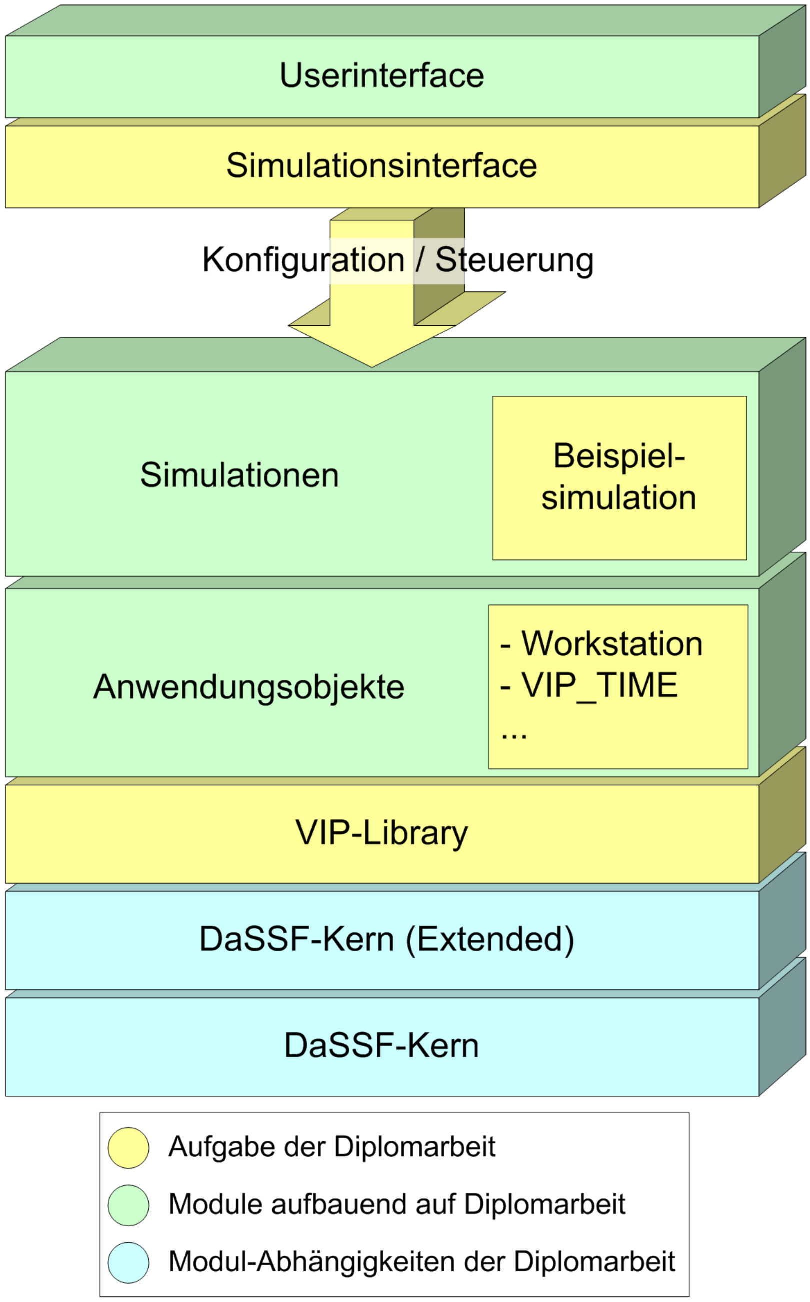

Abbildung 4.1: Schematischer Aufbau des gesamten Simulationsprojektes

4.1.1 Anforderungsanalyse der Objektstruktur

Die Objektstruktur kann aus zwei unterschiedlichen Blickwinkeln betrachtet werden. Aus der Sicht eines Entwicklers, der neue Simulationsobjekte, z.B. neue Protokolle wie IP, auf Basis der VIP-Library implementiert. Diese Sichtweise wird als Entwicklersicht bezeichnet.
Die von den Entwicklern realisierten Simulationsobjekte werden zu einem späteren Zeitpunkt von Anwendern, z.B. StudentenInnen für Simulationen benutzt, um eigene Simulationen ablaufen zu lassen. Das Steuern und Konfigurieren der Simulationen wird dem Anwender über ein Simulationsinterface erleichtert. Die Anforderungen und Wünsche des Anwenders an die Funktionalität dieser Schnittstelle wird hier als Anwendersicht bezeichnet. Wie in Abschnitt 3.1 gezeigt wurde, sind Rechnernetze und deren Kommunikationsmechanismen ein weitgefächertes und komplexes Themengebiet. Die zu erstellende Simulationsbibliothek sollte die wichtigsten Grundkonzepte dieser Kommunikationssysteme unterstützen und es dem Benutzer erleichtern, Rechnernetzsimulationen zu generieren.
Eine weitere wichtige Anforderung ist das Verbergen des DaSSF-Simulationskerns vor den Benutzern der Bibliothek. Ein Programmierer sollte keine Gedanken darauf verwenden müssen, wie der DaSSF-Simulationskern aufgebaut ist, bzw. wie die eigentliche Kommunikation über Kanäle abläuft. Nachstehend werden nun die Anforderungen an das Simulationsinterface und die VIP-Library dargestellt.

4.1.1.1 Anforderungen an das Simulationsinterface – Anwendersicht

- Anwendern soll die Möglichkeit geboten werden, sich aus bestehenden Anwendungsobjekten eigene Rechnernetze mit unterschiedlichen Topologien und Konfigurationen zu erstellen und diese im Simulationskern ablaufen zu lassen.
- Die Schnittstelle zur Simulationskonfiguration bzw. Simulationssteuerung soll einfach und übersichtlich gestaltet sein.
- Neben diesem Konfigurationsaspekt soll das Simulationsinterface (Abbildung 4.1) eine einfache Schnittstelle zur Steuerung der Simulationen bereitstellen. Der Benutzer soll folgende Möglichkeit der Steuerung haben:
  - Neue Simulationen starten
  - Laufende Simulationen stoppen
  - Laufende Simulationen pausieren
  - Pausierende Simulationen wieder fortsetzen

  An dieser Stelle ist jedoch noch nicht absehbar, ob diese Anforderungen letztendlich mit dem Funktionsumfang des DaSSF-Kerns realisierbar sind.
- Die Konfiguration und Steuerung der Simulationen zur Laufzeit soll von dem Simulationsinterface unterstützt werden.
- Das Simulationsinterface soll dem Anwender die Möglichkeit bieten, Haltepunkte, sog. Breakpoints, zu setzen. Diese Breakpoints sollen es dem Anwender ermöglichen, Rechnernetzsimulationen an vordefinierten Stellen bzw. Ereignissen anhalten zu können.
- Der Anwender soll Breakpoints zur Laufzeit definieren können.

- Dem Anwender soll eine Loggingfunktionalität angeboten werden. Analog zum Breakpointmechanismus soll der Anwender an vordefinierten Stellen bzw. Ereignissen die Möglichkeit haben, Simulationsablauf- bzw. Simulationszustandsinformationen aufzuzeichnen, welche nach Beendigung der Simulation weiterverarbeitet werden können.
- Der Anwender soll Loggingpunkte zur Laufzeit definieren können.

In Tabelle 4.1 werden die wesentlichen Anforderungen an das Simulationsinterface aus Anwender- bzw. Entwicklersicht nochmals zusammengefasst dargestellt.

- Benutzerdefinierte Simulationen möglich
- Einfache Konfigurationsschnittstelle
- Einfache Steuerungsschnittstelle mit Funktionalität: Start, Stop, Pause, Resume
- Laufzeitkonfiguration
- Laufzeitsteuerung
- Breakpointfunktionalität
- Loggingfunktionalität

Tabelle 4.1: Zusammenfassung der Anforderungsspezifikation des Simulationsinterfaces

4.1.1.2 Anforderungen an die VIP-Library – Entwicklersicht

Die wesentliche Anforderung an die VIP-Library ist die Modellierung einer adequaten Objektstruktur für eine Rechnernetzsimulation. Die VIP-Library soll offen gehalten werden, damit zukünftige Änderungen im Kommunikationssystem- bzw. Rechnernetzumfeld durch Erweiterungen nachträglich leicht nachvollzogen werden können. Trotz dieser Flexibilität soll den Entwicklern soviel Aufwand wie möglich durch die Programmbibliothek abgenommen werden, indem gemeinsame Aspekte verschiedenster Kommunikationsarchitekturen (z.B. Schichtenmodell - Protokollhierarchie) abstrahiert eingebunden werden. Nachfolgend werden die in Abschnitt 3.1 eingeführten Grundkonzepte moderner Rechnernetzkommunikation in Bezug zu dieser Diplomarbeit gebracht und einer Anforderungsanalyse unterzogen. Das Ergebnis ist die folgende Anforderungsspezifikation an die VIP-Library, welche nach zwei Kriterien, den rechnernetzrelevanten und den programmiertechnischen, klassifiziert werden (Tabelle 4.2).

Rechnernetzrelevante Anforderungsspezifikation

- In Abschnitt 3.1.4 haben wir gesehen, welche grundsätzlichen Rechnernetztopologien existieren. Dementsprechend müssen diese Topologien in der VIP-Library modellierbar sein.
- Moderne Rechnernetze bestehen aus verbundenen Endsystemen, welche untereinander kommunizieren. Diese Endsysteme müssen dementsprechend nachbildbar sein.
- Die Modellierung der Endsysteme soll den Standard Protokollmechanismen des OSI-Referenzmodelles, z.B. Schichtenmodell, SAP-Konzept, nachgebildet werden.
- Wie in realen Rechnernetzen sollen diese Endsysteme mittels Übertragungsmedien und Vermittlungskomponenten verbunden sein und Nachrichten austauschen können. Typische Koppelelemente, wie Hub, Switch, Router oder Ringkonzentrator, sollen durch die VIP-Library modellierbar sein. Zusätzlich sollen die in Abschnitt 3.1.3 beschriebenen Übertragungsmedien,

die für eine Kommunikation zwischen den Endsystemen notwendig sind, von der VIP-Library nachgebildet werden.

- Rechnernetztypen, wie z.B. Ethernet oder Token-Ring mit ihren typischen Protokollmechanismen und Kommunikationsabläufen, sollen umsetzbar sein.

Programmiertechnische Anforderungsspezifikation

- Um dem Entwickler eine einfachere Programmierung zu ermöglichen, soll der DaSSF-Kern mittels der VIP-Library vor dem Entwickler verborgen werden.
- Durch einen modularen, flexiblen Aufbau (Baukastenprinzip) soll die VIP-Library leicht erweiterbar sein.
- Dem Entwickler sollen Logging- bzw. Breakpointmechanismen bereit gestellt werden, welche er bei der Erstellung neuer Anwendungsobjekte einfach einbinden kann.
- Die von der VIP-Library angebotenen Grundobjekte sollen bereits Standard-Logging- und Breakpointfunktionalitäten implementiert haben, die den Entwickler dahingehend entlasten, dass er diese Logging- bzw. Breakpointfunktionalitäten in seinen Objekten nicht mehr eigens vorsehen muss.
- Dem Entwickler sollen später bei der Implementierung von Simulationsobjekten viele Aufgaben abgenommen werden.
- Aus didaktischen Gründen soll die VIP-Library Entwicklern die Möglichkeit bieten, den Anwendern (StudentenInnen) theoretische Grundmechanismen, z.B. 3-Way-Handshake von TCP oder Durchlauf von Nachrichten durch Protokollstacks, in einfacher Weise präsentieren zu können, ohne dabei komplette Implementierungen umsetzen zu müssen.

Anforderungen (rechnernetzrelevant)
- Modellierbarkeit gängiger Rechnernetztopologien - Modellierbarkeit von Endsystemen - Modellierbarkeit von Übertragungsmedien - Modellierbarkeit von Vermittlungskomponenten - Modellierbarkeit der Protokollmechanismen des OSI-Referenzmodells - Modellierbarkeit gängiger Rechnernetztypen
Anforderungen (programmiertechnisch)
- Verbergen des DaSSF-Kerns - Leichte Erweiterbarkeit - Breakpointmechanismus individuell erweiterbar - Loggingmechanismus individuell erweiterbar - Standard-Breakpointmechanismus - Standard-Loggingmechanismus - Modellierbarkeit von Teilaspekten realer Kommunikation (didaktische Gründe)

Tabelle 4.2: Zusammenfassung der Anforderungsspezifikation der VIP-Library

4.1.2 Anforderungsanalyse der Beispielsimulation VIP_Time

Der zweite Teil der Diplomarbeit besteht aus der Generierung einer Beispielsimulation unter Verwendung der VIP-Library. Anhand der Beispielsimulation soll die Grundfunktionalität der Objektstruktur gezeigt werden.

Abbildung 4.2 zeigt den logischen Aufbau des zu simulierenden Rechnernetzes, welches der Beispielsimulation zu Grunde liegen soll. In einem Beispielrechnernetz sind vier Time-Clients und ein Time-Server über einen 8-Port-Hub mittels Twisted Pair Kabel miteinander verbunden. Alle Endsysteme haben den in Abbildung 4.3 dargestellten Protokollstack implementiert. Die Protokolle dieses Stacks werden später genauer betrachtet. Workstation 1 soll mit dem Server mittels der VIP_TIME-Anwendung kommunizieren. VIP_TIME bietet einen einfachen verbindungsorientierten Dienst an. Ein Time-Client kann sich mit einem TIME-Server verbinden und dessen aktuelle Uhrzeit abfordern.

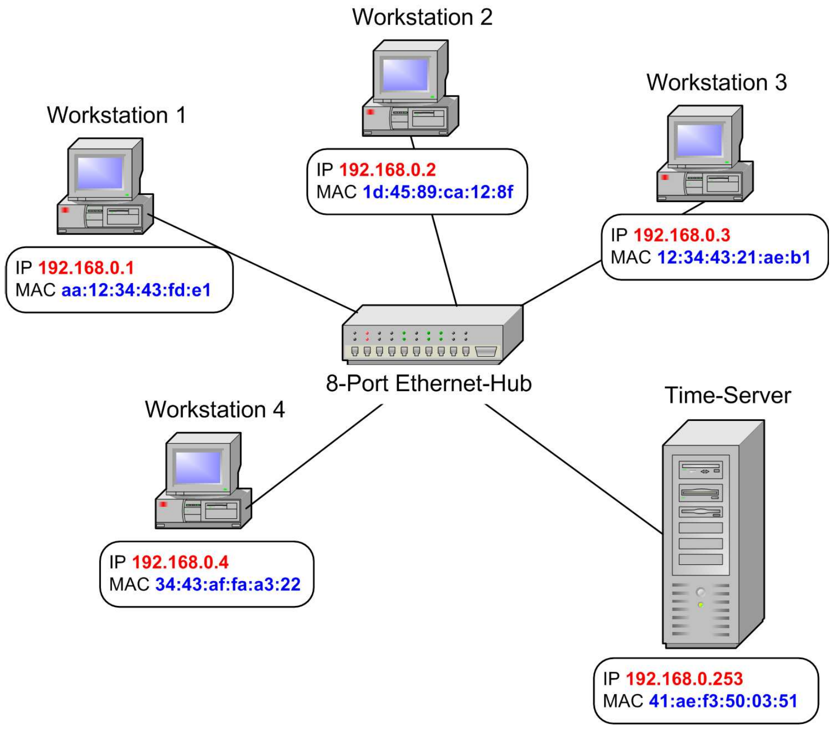

Abbildung 4.2: Rechnernetzaufbau der Beispielsimulation

4.1.2.1 Detaillierte Anforderungsspezifikation: 8-Port-Ethernet-Hub

Einzige Aufgabe dieser Kopplungskomponente ist es, eingehende Nachrichten unverändert auf allen Ports wieder auszugeben.

4.1.2.2 Detaillierte Anforderungsspezifikation: Komponenten

- Alle zu simulierenden Endsysteme haben den in Abbildung 4.3 dargestellten Protokollstack implementiert. Auf einem der Endsysteme, dem Time-Server, läuft die VIP_TIME-Serveranwendung. Auf den restlichen Endsystemen ist die VIP_TIME-Clientanwendung implementiert.
- Die Konfigurationen der einzelnen Endsysteme sind in Abbildung 4.2 dargestellt.

4.1.2.3 Detaillierte Anforderungsspezifikation: Protokollstack und Protokolle

Der Protokollstack aller beteiligten Endsysteme sollte normalerweise auf dem TCP/IP-Referenzmodell aufgebaut sein. Eine Vollimplementierung allein des Protokolls IP, deren Beschreibung im RFC 971 mehr als 150 Seiten umfaßt, würde den Rahmen dieser Diplomarbeit logischerweise völlig überschreiten. Deshalb werden in der Beispielsimulation der Diplomarbeit nur sehr stark vereinfachte Pseudoimplementierungen des TCP/IP-Referenzmodells erstellt.

ETHmin[1]

Einzige Funktion dieses Protokolls ist es, ausgehende Nachrichten mit einem ETHmin-Header zu versehen bzw. von eingehenden Nachrichten diesen Header abzustreifen. Dieser Header enthält als alleinige Informationen die Ziel- und Quell-MAC-Adressen.

IPmin

Einzige Funktion dieses Protokolls ist es, ausgehende Nachrichten mit einem IPmin-Header zu versehen bzw. von eingehenden Nachrichten diesen Header abzustreifen. Dieser Header enthält als alleinige Informationen die Ziel- und Quell-IP-Adressen.

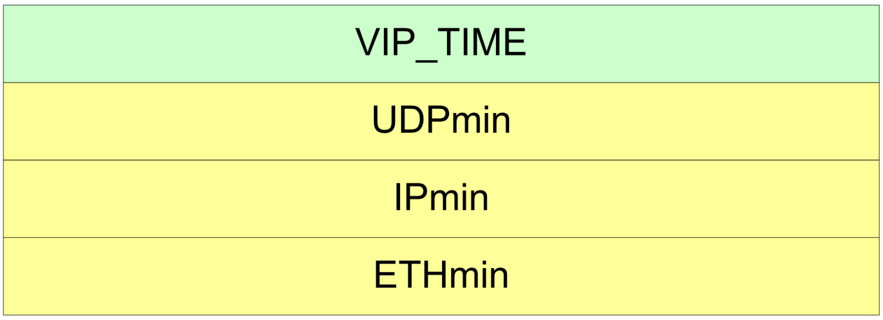

Abbildung 4.3: Protokollstack der Komponenten der Beispielsimulation

UDPmin

Einzige Funktion dieses Protokolls ist es, ausgehende Nachrichten mit einem UDPmin-Header zu versehen bzw. von eingehenden Nachrichten diesen Header abzustreifen. Dieser Header enthält als alleinige Informationen die Ziel- und Quell-Portnummern.

VIP_TIME

VIP_TIME ist eine verbindungsorientierte Client-Server-Anwendung, welche auf dem Protokoll UDPmin aufsetzt. Ein Dienstnutzer dieser Anwendung, ein VIP_TIME-Client, kann sich an einem VIP_TIME-Server anmelden (Abbildung 4.4) und dessen Systemzeit anfordern. Die Übertragung der Zeit soll unbestätigt erfolgen (Abbildung 4.5). Nach erfolgreicher Übertragung kann der Client die Verbindung wieder schließen (Abbildung 4.6).

Unterschiedliche Nachrichtentypen:

[1] Das Postfix min steht für minimale Protokollimplementierung

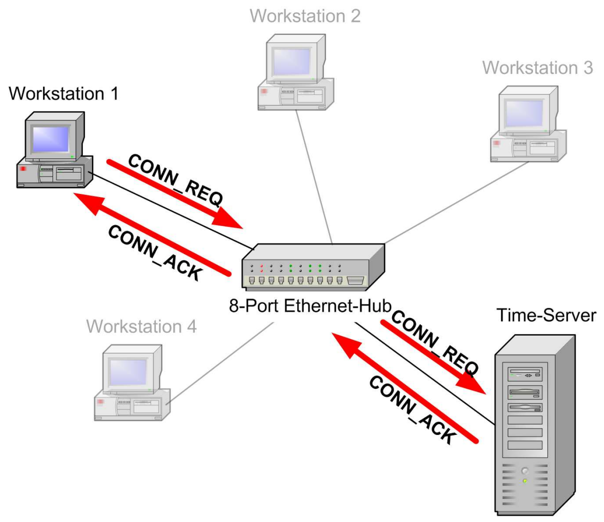

Abbildung 4.4: Verbindungsaufbau der Beispielapplikation VIP_TIME

- **CONN_REQ** — Verbindungsaufbauwunsch (TIME-Client)
- **CONN_ACK** — Verbindungsbestätigung (TIME-Server)
- **TIME_REQ** — Zeitanforderung (TIME-Client)
- **TIME_DATA** — Systemzeitantwort (TIME-Server)
- **DISCONN_REQ** — Verbindungsabbauwunsch (TIME-Client)
- **DISCONN_ACK** — Verbindungsabbaubestätigung (TIME-Server)

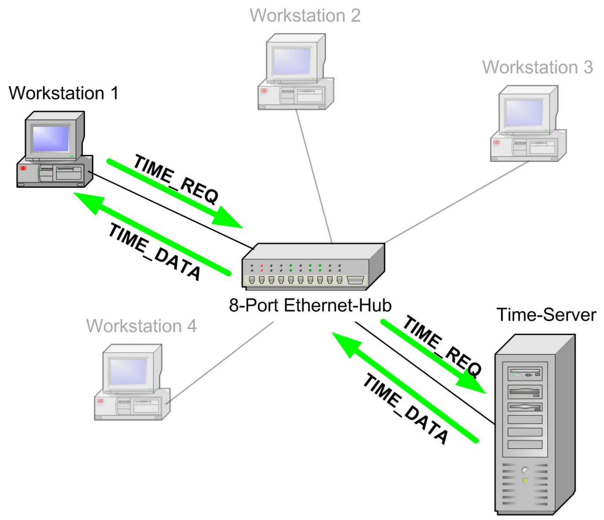

Abbildung 4.5: Datentransfer der Beispielapplikation VIP_TIME

Befehlsschnittstelle (SAP) für Dienstnutzer von VIP_TIME:

- **connect** ***ip*** — Verbindungsaufbauwunsch mit der angegebenen Ziel-IP-Adresse

- **gettime** — Abfrage der aktuellen Systemzeit des TIME-Servers
- **disconnect** — Verbindungsabbauaufforderung

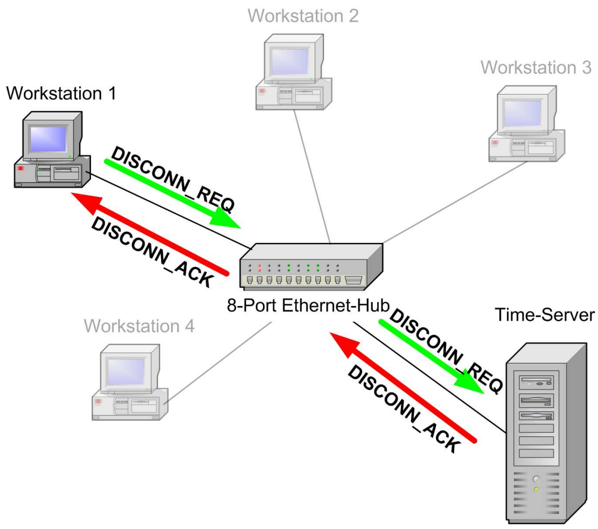

Abbildung 4.6: Verbindungsabbau der Beispielapplikation VIP_TIME

Zustände des VIP_TIME-Protokolls:

- **IDLE** — Initialzustand von TIME-Client bzw. TIME-Server
- **WAITING_OPENACK** — TIME-Client wartet auf die Verbindungsaufbaubestätigung des TIME-Servers durch CONN_ACK-Paket
- **ESTABLISHED** — Zustand von Client bzw. Server, wenn eine Verbindung besteht
- **WAITING_TIME** — Client wartet auf ein TIME_DATA-Paket des Servers
- **WAITING_CLOSE** — Client wartet auf die Verbindungsabbaubestätigung des Servers durch DISCONN_ACK-Paket

4.2 Design

Das Design der VIP-Library für das VIP-Projekt kann grundsätzlich in zwei Hauptphasen unterteilt werden.

Die erste Phase des Entwurfes behandelt die Modellierung der logischen Konzepte und Zusammenhänge von Rechnernetzen. Es werden die allgemeinen Grundobjekte und deren Kommunikationsmechanismen erarbeitet und in schrittweiser Verfeinerung detailliert beschrieben.
Aufbauend auf diesen Ergebnissen wird in der zweiten Phase der DaSSF-Simulationskern zur Entwurfsbetrachtung hinzugezogen. Die Kommunikation zwischen Objekten in einer DaSSF-Simulation beruht nicht auf dem klassischen objektorientierten Nachrichtenaustausch. Bei diesem Nachrichtenaustauschverfahren muss ein Objekt eine Methode eines anderen Objektes aufrufen, um Nachrichten zu diesem zu senden. Im Gegensatz dazu wird die Kommunikation zwischen Objekten in einer DaSSF-Simulation über simulationsinterne Nachrichtenkanäle abgewickelt. Aufgrund dessen müssen

die in Phase 1 gewonnenen Entwurfsergebnisse angepasst bzw. erweitert werden.
Nach Abschluß des Designs der VIP-Library wird das Simulationsinterface entworfen.

4.2.1 Design – VIP-Library (Phase 1)

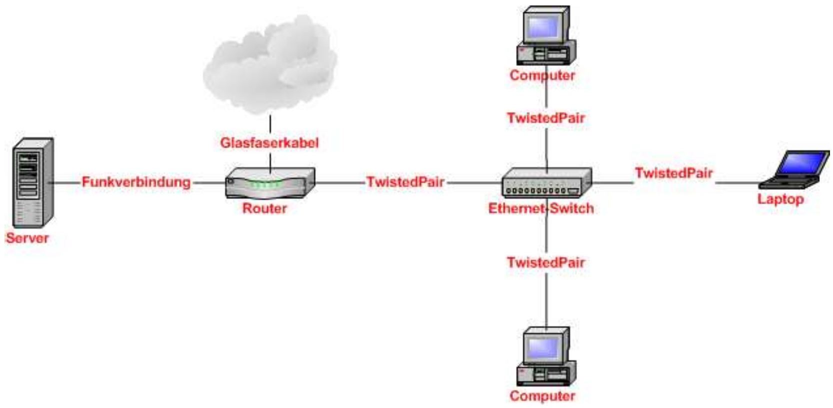

Abbildung 4.7: Beispielhafter Aufbau eines Rechnernetzes

4.2.1.1 Komponenten

Abbildung 4.7 zeigt einen typischen Aufbau eines in der Realität vorkommenden Rechnernetzes. Rechnernetze bestehen aus autonomen, miteinander verbundenen Endsystemen, die untereinander Informationen austauschen können. Diese Endsysteme bilden das erste Grundobjekt der zu modellierenden VIP-Library und werden im weiteren Verlauf der Diplomarbeit als Komponenten bezeichnet. Dieser Begriff der Komponente umfasst im Gegensatz zum normalen Sprachgebrauch nur die Endsysteme eines Rechnernetzes, nicht jedoch die Kopplungselemente (Tabelle 4.3).

Objekt	Beispiel
Komponente	Host, Workstation, ...

Tabelle 4.3: Grundobjekte: Komponente

Betrachtet man die Ausstattung von Komponenten nur unter dem Kommunikationsaspekt, so besteht diese nur aus einer Netzwerkkarte, gekoppelt an ein Betriebssystem (Abbildung 4.8) und den darauf aufbauenden Anwendungen. Andere typische Elemente eines Rechnersystems, z.B. Grafikkarte, Arbeitsspeicher oder Festplatte können vernachlässigt werden, da sie keinen wesentlichen Einfluß auf die Kommunikation haben. Eine spätere Erweiterung des Komponentenbegriffs ist aber durchaus vorstellbar und möglich.
Das Betriebssystem umfaßt die für die Kommunikation wichtigen Mechanismen einer Komponente. Neben dieser logischen Einheit, deren weitere notwendige Verfeinerung in Abschnitt 4.2.1.5 erklärt wird, sind Netzwerkkarten ein wichtiges und zentrales Objekt.
Diese Schnittstellenkarten, die als “Ein- bzw. Ausgangs-Pforte“ der Komponenten für Nachrichten zum Rechnernetz dienen, bilden das nächste Grundobjekt der Library, den Netzwerk-Adapter. Netzwerk-Adapter stellen die Bitübertragungsschicht des ISO/OSI-Referenzmodells dar. Wie in Abschnitt 3.1 beschrieben, gibt es viele verschiedene Ausprägungen von Rechnernetztopologien und

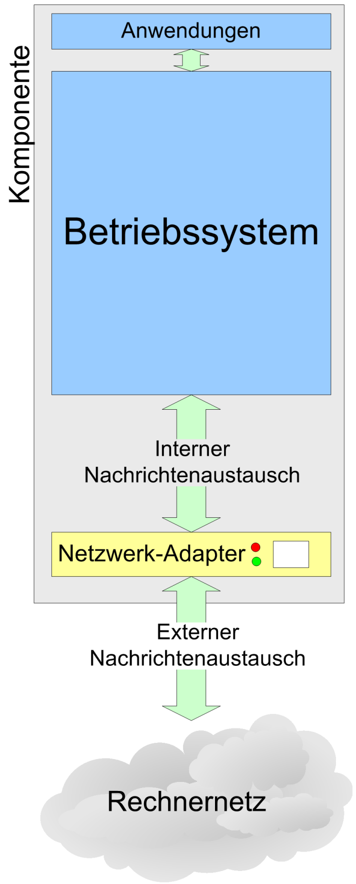

Abbildung 4.8: Logischer Aufbau einer Komponente (Betriebsystem)

dementsprechend auch die unterschiedlichsten Arten von Schnittstellenkarten, z.B. für Ethernet, Token-Ring oder ATM. Einzige gemeinsame Eigenschaft dieser Adapter ist es, Nachrichten vom Be-

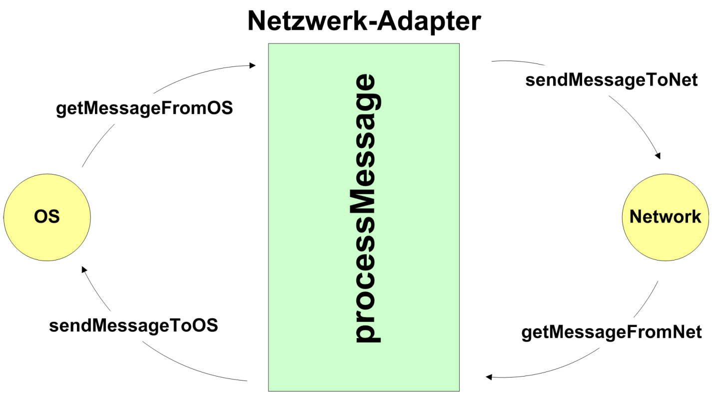

Abbildung 4.9: Ablaufmechanismus eines Netzwerk-Adapters

triebssystem bzw. Rechnernetz entgegen zu nehmen, nach vordefinierten Regeln zu verarbeiten und ggf. wieder weiterzuleiten. Dieser generelle Ablaufmechanismus eines Netzwerk-Adapters wird in Abbildung 4.9 schematisch dargestellt.

In der VIP-Library werden die Netzwerk-Adapter auf reine Sende- und Empfangseinheiten reduziert. Die Logik, welche eine reale Netzwerkkarte normalerweise implementiert, z.B. Zugriffverfahren wie CSMA/CD, wandert aus dem Netzwerk-Adapter-Objekt in das darübergelegene Betriebssystem. Da eine reale Netzwerkkarte nur ein Protokoll implementiert hat, kann somit dementsprechend ein Netzwerk-Adapter nur an genau ein Protokoll, das sog. Interfaceprotokoll, gebunden sein. Diese Trennung der logischen von den physikalischen Eigenschaften eines Netzwerk-Adapter wird deshalb so modelliert, damit die Logik des Netzwerk-Adapters in das darüberliegende Betriebssystem wandert. Aufgrund dieser Reduzierung benötigt die VIP-Library nur noch einen Adaptertypen. Wie die Logik im Betriebssystem umgesetzt wird, ist Thema des Abschnittes 4.2.1.4.

4.2.1.2 Konnektoren

Neben diesen autonomen Komponenten gibt es in einem Rechnernetz Koppelelemente, die zur Verbindung der Endsysteme eingesetzt werden. In Abbildung 4.7 wird beispielsweise ein Ethernetswitch eingesetzt, um die Endsysteme untereinander zu verbinden. In der VIP-Library werden diese Koppelelemente durch sog. Konnektoren modelliert (Tabelle 4.4).
In ihrer Arbeitsweise ähneln Konnektoren Netzwerk-Adaptern. Sie empfangen Nachrichten, verarbeiten sie und leiten sie daraufhin weiter.
Im Gegensatz zu einem Netzwerk-Adapter besitzt ein Konnektor aber mehrere Ein- bzw. Ausgänge, welche als Ports bezeichnet werden, z.B. ein 8-Port-Ethernet-Switch. Die Logik, auf welchen Ports Nachrichten ausgegeben werden, wird in der Verarbeitungsphase bestimmt und ist von der Logik des Konnektors abhängig. Ein Hub leitet beispielsweise eingehende Nachrichten an alle seine verfügbaren Ports weiter. In dieser Phase, in der empfangene Nachrichten vom Konnektor verarbeitet werden, können typspezifische Datenstrukturen, wie z.B. MAC-Port-Tabellen eines Switches, angelegt werden.

Objekt	**Beispiel**
Konnektor	Repeater, Hub, Switch, Router, ...

Tabelle 4.4: Grundobjekt: Konnektor

4.2.1.3 Verbindungsmedium

Komponenten und Konnektoren können mittels unterschiedlichster Verbindungsmedien gekoppelt werden (Tabelle 4.5). Neben den klassischen Medien, wie Twisted-Pair-Kabel und Glasfaserkabel, können die kommunizierenden Objekte auch mittels anderer Verbindungsmedien, z.B. Funk, verbunden werden (Abschnitt 3.1).
Abstrakt betrachtet ist ein Verbindungsmedium ein uni- bzw. bidirektionaler (z.B. Telefonleitung: bidirektional, Satelliten-Verbindung: unidirektional) Nachrichtenkanal, in den Nachrichten eingespeist werden können. Diese Nachrichten werden dann über die Kanallänge transportiert, ggf. verfälscht und auf der anderen Seite an den Kommunikationspartner übergeben. Wie man leicht erkennen kann, ähnelt dieses Verfahren wiederum den zuvor erklärten Mechanismen der Komponenten und Konnektoren (empfange – verarbeite – sende).

Die Konstellation der an ein Verbindungsmedium angeschlossenen Kommunikationspartner kann folgende Formen annehmen:

- Konnektor (Port) — Konnektor (Port)
- Konnektor (Port) — Komponente (Interface)
- Komponente (Interface) — Komponente (Interface)

Um die unterschiedlichen Anschlussmodule (Konnektor: Port, Komponente: Interface) der Kommunikationspartner, bezogen auf das Anschlussmedium, zu vereinheitlichen, wird das Bild der Komponenten bzw. der Netzwerk-Adapter (Abschnitt 4.2.1.1) um den Begriff Port erweitert.
Ein Netzwerk-Adapter kommuniziert nun nicht mehr direkt mit seiner Außenwelt, sondern besitzt nun wie ein Konnektor einen Port, um die Kommunikation durchzuführen. Aufgrund dieser Porterweiterung kann ein Verbindungsmedium nun als ein Nachrichtenkanal zwischen zwei Ports betrachtet werden. Der Typ der Kommunikationspartner (Konnektor, Komponente) kann somit vor dem Verbindungsmedium verborgen werden und erleichtert dadurch das Erweitern der VIP-Library um neue Kommunikationsobjekte.

Objekt	**Beispiel**
Verbindungsmedium	Koaxialkabel, Glasfaserkabel, Twisted-Pair CAT-4, Twisted-Pair CAT-5, Funk, Infra-Rot, ...

Tabelle 4.5: Grundobjekt: Verbindungsmedium

Abbildung 4.10 zeigt einen schematischen Überblick der bis zu diesem Zeitpunkt des Designs gewonnenen Grundobjekte der VIP-Library.

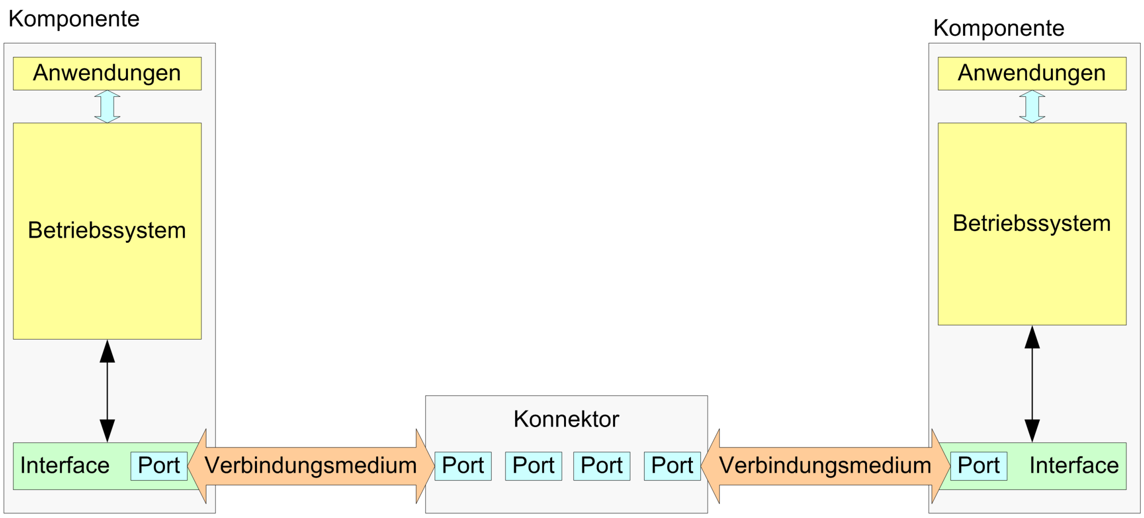

Abbildung 4.10: Überblick der Grundobjekte

4.2.1.4 Betriebssystem

Da das Betriebssystem den Kern einer Komponente (Abschnitt 4.2.1.1) bildet, wird es in den folgenden Abschnitten detaillierter betrachtet. In diesem Zusammenhang wird ausschließlich auf kommunikationsrelevante Eigenschaften eingegangen.

Der Begriff des Betriebssystems umfasst fast alle Funktionen heutiger Rechnersysteme. Arbeitsspeicherverwaltung, Prozessorzuteilung oder Bereitstellung eines Dateisystems sind nur ein kleiner Teil der Aufgaben, die ein modernes Betriebssystem leisten muss. Um die VIP-Library übersichtlich und flexibel zu halten, müssen solche Eigenschaften bzw. Funktionalitäten vernachlässigt werden. Nur für die Rechnerkommunikation wirklich wichtige Aspekte eines Betriebssystems werden nachgebildet. Aus dieser Sichtweise heraus kann das Betriebssystem auf eine Komponente reduziert werden, den Protokollstack, welcher alle für eine Kommunikation relevanten Objekte beinhaltet. Wie in Abschnitt 3.1 bereits gezeigt wurde, definiert ein Protokollstack alle kommunikationsrelevanten Protokolle, deren Schnittstellen und Nachrichtenformate.

4.2.1.5 Protokollgraph

Ein Protokollstack kann abstrakt als Graph dargestellt werden, da ein Protokollstack neben den Protokollen mit all ihren Eigenschaften unter anderem auch deren gegenseitige Abhängigkeiten definiert. Die Mögichkeit, einen Stack als Graphen darzustellen, wird in die VIP-Library übernommen. Eine Komponente besteht nun nicht mehr aus einem Betriebssystem, sondern aus einem Protokollgraphen, welcher wiederum an einen Netzwerk-Adapter gekoppelt ist (Abbildung 4.11). Dieser Protokollgraph bildet ein neues Grundobjekt der VIP-Library. Um eine noch weitergehende Vereinfachung zu erreichen, werden die Anwendungen, die normalerweise auf dem Betriebssystem aufsetzen, in der VIP-Library als Protokolle modelliert. Damit können sie als ein Teil des Protokollgraphen betrachtet werden.

Ein Protokollgraph hält die Information über den Aufbau eines Protokollstacks. Diese Struktur ist deutlich flexibler im Hinblick auf Änderungen in der Protokollschichtung gegenüber einer Verankerung der Abhängigkeitsinformation in den Protokollen selbst. Protokolle können, ohne geändert zu werden, unterschiedlich miteinander verbunden werden. Vebindungen zwischen Protokollobjekten sind allerdings nur dann möglich, wenn die neu zu verbindenden Protokolle logisch zueinander passen. Mittels dieses Mechanismuses kann man schnell neue Protokollstacks aus schon existierenden Protokollen generieren. Beispielsweise kann man einen minimalen TCP/IP-Protokollstack erstellen, der nur Protokolle enthält, die für eine bestimmte Applikation notwendig sind (z.B. wird kein **Transmission Control Protocol** (**TCP**) für den Datentransfer über **Trivial File Transfer Protocol** (**TFTP**) benötigt). Gerade für die Aufbauphase der Anwendungsobjekte des VIP-Projektes sind diese “Plug-And-Play“-Eigenschaften vorteilhaft, da anfangs nur Protokolle implementiert werden müssen, die für eine Anwendung notwendig sind. Dieser minimale Protokollstack kann dann kontinuierlich erweitert werden, ohne dabei die vorhandenen Protokolle modifizieren zu müssen.

Die Aufgabe der VIP-Library sollte es sein, Grundobjekte, die zur Erstellung von Anwendungsobjekten für eine Rechnernetzsimulation notwendig sind, dem Entwickler zur Verfügung zu stellen. Aufbauend auf diese VIP-Library sollte dem Entwickler neuer Anwendungsobjekte soviel Programmieraufwand wie möglich abgenommen werden. Das Finden eines geeigneten Grundobjektes zur abstrakten Darstellung der Eigenschaften und Mechanismen realer Rechnernetzprotokolle ist schwierig. Aus der großen Anzahl verschiedener Protokolle muss dementsprechend eine Gemeinsamkeit in Verhalten und Eigenschaft gefunden werden.

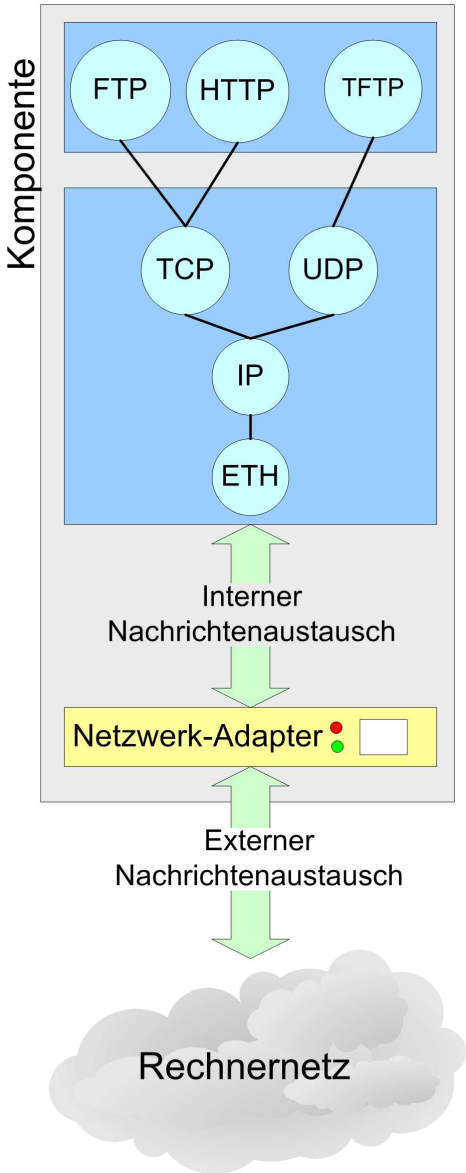

Abbildung 4.11: Logischer Aufbau einer Komponente (Betriebssystem reduziert auf Protokollgraph)

4.2.1.6 Protokollnachricht

Nach intensiver Analyse der Nachrichtenformate realer Protokolle kann eine Protokollnachricht übergreifend modelliert werden. Sie kann aus folgenden Elementen bestehen:

Header Der Header einer Protokollnachricht besteht aus einzelnen Feldern, deren Bedeutung protokollspezifisch ist.

Payload Die Payload ist der Anteil einer Nachricht, welcher die wirklichen Nutzdaten enthält.

Trailer Ähnlich wie der Header hat der Trailer einen protokollspezifischen Inhalt.

Logisch betrachtet können diese drei Bestandteile in zwei Gruppen aufgeteilt werden. Header und Trailer halten die Steuerungsinformationen für die Kommunikation. Darin wird unter anderem die Ziel- und Quelladresse der Nachricht festgelegt. Die tatsächlich zu übertragende Nutzdateninformation ist in der Payload enthalten.
Um nun diese Vielzahl an unterschiedlichen Nachrichtenarten in der VIP-Library auf ein Oberobjekt

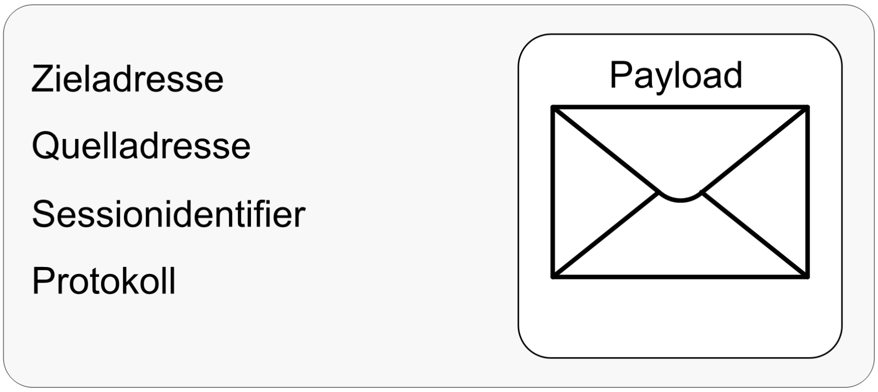

Abbildung 4.12: Abstrahierte Form einer Protokollnachricht

abbilden zu können, müssen deren Grundzüge abstrahiert werden. Abbildung 4.12 ist das Ergebnis dieses Abstraktionsprozesses. Natürlich benötigt eine Protokollnachricht, welche zwischen Kommunikationspartnern ausgetauscht werden soll, entsprechende Quell- und Zielinformationen. Diese Informationen sind Grundbestandteil eines jeden Protokollobjektes der VIP-Library. Zu diesen Informationen werden auch die passenden Zugriffsmechanismen für das Auslesen und Setzen bereitgestellt. Zusätzlich enthält jede Protokollnachricht der VIP-Library einen Session-Identifier, welcher in der Initialisierungsphase auf der Senderseite kreiert wird. Als letzte Steuerungsinformation enthält jedes Nachrichtenobjekt den zugehörigen Protokolltyp. Die Payload ist optionaler Bestandteil und selbst wieder vom Typ Protokollnachricht, da beim Durchlaufen des Protokollgraphen (Abbildung 3.2) sukzessive an Daten bzw. Nachrichten weitere Header bzw. Trailer angefügt werden. Damit werden Nachrichten der Schicht N zu Nutzdaten, d.h. zur Payload der Schicht N-1.

4.2.1.7 Protokoll und Protokollsession

Die beiden Hauptobjekte der VIP-Library bilden Protokoll und Protokollsession. Diese Modellierung ist an die Architektur des x-Kernel, welcher ein Framework zur Erstellung von Protokollen ist, angelehnt [PDB 96].

Die Funktionsweise eines Protokolls kann aus zwei verschiedenen Blickwinkeln betrachtet werden. Ein Protokoll ist Teil eines Protokollstacks, in welchem es über Service Access Points mit anderen Protokollen des gleichen Protokollstacks kommuniziert und Informationen austauscht. Diese Sichtweise wird als “SAP-View“ bezeichnet. Wie in Abschnitt 3.1.1 gezeigt, läuft die logische Kommunikation eines Protokolls mit Partnerinstanzen, sog. Peers anderer Komponenten ab. Diese Betrachtungsweise eines Protokolls, bezogen auf dessen Umfeld, wird als “Peer-to-Peer-View“ bezeichnet.

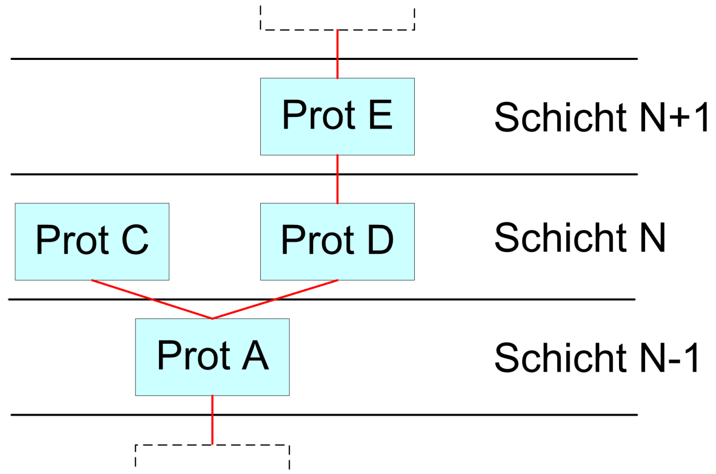

Abbildung 4.13: Schichtendarstellung eines Protokollgraphen

Die hierarchische Anordnung der Protokolle einer Komponente innerhalb eines Protokollgraphen unterliegt einigen Regeln. Protokollgraphen werden in unterschiedliche Schichten aufgeteilt. Diesen Schichten können nun Protokolle zugeordnet werden (Abbildung 4.13). Entsprechend dem Konzept des OSI-Modells, bietet jedes Protokoll über SAP den Protokollinstanzen der darüberliegenden Schicht bestimmte Dienste an. Auf welche Art diese Dienste erbracht werden ist für die Dienstnutzer, also die Protokollinstanzen der höheren Schicht die auf diesem Protokoll aufsetzen, unbedeutend. Die Beziehung zwischen den Nutzern und Anbietern von Diensten ist eindeutig definiert, d.h. Protokolle können nur Dienste von Protokollen der darunterliegenden Schicht nutzen. Diese Grundeigenschaft der Protokollkommunikation des OSI-Modells wird ebenfalls in der VIP-Library umgesetzt. Jegliche Dienstnutzungsbeziehungen innerhalb eines Protokollgraphen entsprechen dieser Dienstschnittstellendefinition.

Protokollinstanzen können während eines Kommunikationsablaufes mit ihrer Partnerinstanz (Peer-to-Peer-View) verschiedene Zustände annehmen, z.B. *ESTABLISHED*, *CLOSE-WAIT* im Falle von TCP. Abhängig von bestimmten Nachrichtentypen können sich diese Zustände verändern. Da zu einem Zeitpunkt ein Protokoll mehrere Instanzen verwalten kann, beispielsweise zwei zeitgleiche TCP-Verbindungen zu unterschiedlichen Ports, muss dieses in der Lage sein auch mehrere Zustände verwalten zu können. Diese mehrfache Zustandsverwaltung wird in der VIP-Library durch die Aufteilung in Protokoll und Protokollsession ermöglicht. Diese beiden Objekte der VIP-Library bilden dementsprechend Protokoll und Protokollinstanz des OSI-Modells nach. Protokollsessions sind Containerobjekte, die protokollspezifische Zustände annehmen können. Die dazugehörigen Protokolle sind die Session-“Owner“, welche die zuständigen Verwaltungsobjekte darstellen. Da wie eben gezeigt ein Protokoll gleichzeitig mehrere Zustände annehmen kann, verwaltet ein Protokoll auch mehrere Protokollsessions. An einer Kommunikation sind immer mindestens zwei Kommunikationspartner beteiligt. Diese Partner besitzen jeweils eigene Zustände und dementsprechend eigene Sessionobjekte. Aufgrund dieser Eigenschaft können Protokollsessions auch als Kommunikationsendpunkte aufgefasst werden (Peer-to-Peer-View). Dementsprechend sind für eine Kommunikation zwischen zwei Peers genau zwei Protokollsessions dieser Protokolle notwendig.

Zu den Aufgaben eines Protokolls gehört neben dem Verwalten auch das Erstellen bzw. Zerstören von Protokollsessions, da diese Funktionalitäten für jede Kommunikation benötigt werden. Diese Funk-

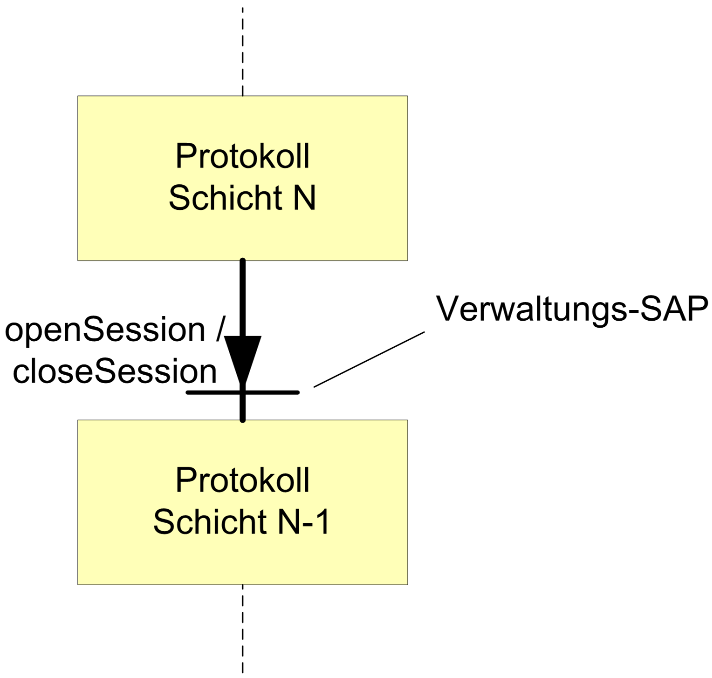

Abbildung 4.14: Verwaltungs-SAP eines Protokolls

tionalitäten werden in der VIP-Library entsprechend durch openSession- und closeSession-Methoden der Protokolle modelliert, welche als Verwaltungs-SAPs Protokollen der darüberliegenden Schicht angeboten werden (Abbildung 4.14). Bevor ein Protokoll der Schicht N eine Protokollsession mittels der openSession-Methode erstellen kann, benötigt es eine passende Protokollsession des darunterliegenden Protokolls N-1, um von dieser eine send-Funktionalität zu erhalten. Diese Protokollsession wird durch einen openSession-Aufruf bei Protokoll N-1 angefordert. Dieses Verfahren setzt sich nun bis zum Interfaceprotokoll fort. Diese auf jeder Schicht getroffenen Zuordnungen von eigenen Protokollsessions zu denen der darunterliegenden müssen nun für eine weitere Nutzung in einer passenden Datenstruktur, der sog. Sessionmap, für jedes Protokoll gehalten werden. Die einer Sessionmap zugrundeliegende Datenstruktur ist eine Tabelle. Eine Sessionmap eines Protokolls A der Schicht N besteht aus folgenden Einträgen:

- Protokollsession des Protokolls A
- dazugehörige Protokollsession aus Schicht N-1
- Sessionidentifier

Der Sessionidentifier wird benötigt, um im Falle eines Nachrichtenempfangs die Nachrichten auf jeder Protokollschicht wieder ordnungsgemäß der richtigen Protokollsession der gleichen Schicht zuordnen zu können. Der Sessionidentifier wird in jeder Schicht in der Initialisierungsphase auf jener Seite der Komponente, welche die Kommunikation initialisiert, in den Protokollen generiert. Dieser Identifier ist in der Simulation eindeutig. Die Kommunikation, d.h. das Senden, Empfangen und Verarbeiten von Nachrichten wird über die Sessionobjekte abgebildet. Sessionobjekte eines Protokolls können als dynamische SAPs angesehen werden, welche Sessionobjekten der darüberliegenden Schicht Dienste zum Versenden von Nachrichten, mittels send-Methode, anbieten (Abbildung 4.15). Der Sendefall, d.h. Nachrichten durchlaufen die Protokollsessionobjekte eines Protokollgraphen von höher- zu tieferliegenden Protokollschichten, ist einfach modellierbar. Der Sendefall ist die Nutzung eines Dienstes, der durch das unmittelbar darunterliegende Sessionobjekt mittels SAP angeboten wird, d.h. Sessionobjekte kommunizieren in diesem Fall direkt mit ihren darunterliegenden Sessionobjekten. Während eine Nachricht die Sessionobjekte durchläuft, werden die jeweils protokollspezifischen Behandlungen der Nachricht vorgenommen. Dies geschieht durch die Protokollsessionmethode processDownMessage. Hierbei wird ein neues Protokollnachrichtenobjekt generiert und die von der darüberliegenden Protokollsession erhaltene Nachricht als Payload verwendet. Zusätzlich wird hierbei der Protokolltyp des "Owner"-Protokolls in den Nachrichtenheader eingetragen.

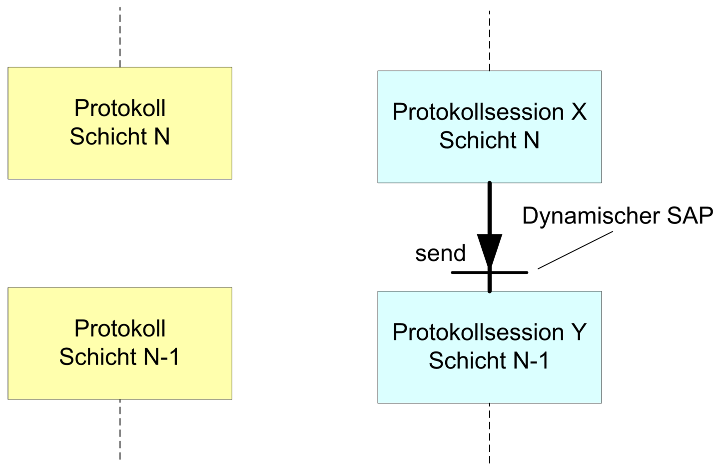

Abbildung 4.15: Dynamische SAP-Eigenschaft einer Protokollsession im Sendefall

Der Empfangsfall ist gegenüber dem Sendefall deutlich komplexer, da wie in Abschnitt 3.1.1 beschrieben, Dienst-Dienstnutzerbeziehungen einer Protokollschichtung nur unidirektional definiert sind. Protokollinstanzen einer Schicht können nur Dienste der darunterliegenden Schicht benutzen, wissen dabei aber nicht, welche Instanz ihre Dienste nutzt. Somit ist eine Nachrichtenweitergabe von Sessionobjekten der Schicht N-1 zu Sessionobjekten der Schicht N (Empfangsfall) auf direktem Weg nicht möglich. Damit dieser Grundmechanismus der Protokollschichtung nicht verletzt wird, werden in der VIP-Library im Empfangsfall die Protokollobjekte herangezogen, um diese Zuordnung von Sessionobjekten herzustellen. Sessionobjekte der Schicht N-1 kennen aufgrund des Protokolltyps in der Payload einer empfangenen Protokollnachricht das darüberliegende zugehörige Protokoll. Mittels dieser Information können Nachrichten an das passende Protokollobjekt weitergeleitet werden. Um diese Funktionalität in der VIP-Library nachzubilden, erhält jedes Protokollobjekt eine demux-Callbackmethode, die Daten der Sessionobjekte der darunterliegenden Schicht entgegennimmt (Abbildung 4.16). Das Protokollobjekt kennt als Verwalter seine Sessionobjekte. Um nun die Nachricht an die dazugehörige Protokollsession weiterleiten zu können, muss diese zuerst identifiziert werden. Für diesen Zweck halten Protokollobjekte der VIP-Library sog. Sessionmaps, welche eine eindeutige Identifizierung des entsprechenden Sessionobjektes ermöglichen. Eine Anfrage an diese Verwaltungsstruktur wird als Sessionmap-Lookup bezeichnet. Um Nachrichten von Protokollobjekten zu den zugehörigen Sessionobjekten weiterleiten zu können, wird in der VIP-Library ein zusätzlicher SAP zwischen Protokoll und zugehörigen Protokollsessionobjekten eingeführt. Die danach an die Sessionobjekte übergebenen Nachrichten werden mittels der proccessUpMessage-Methode bearbeitet und an das übergeordnete Protokoll übergeben.

Um das dynamische Verhalten der Protokoll- und Protokollsessionobjekte besser verstehen zu können, wird dieses anhand eines Beispiels (Abbildung 4.17) dargestellt. Es wird ein einseitiger Informati-

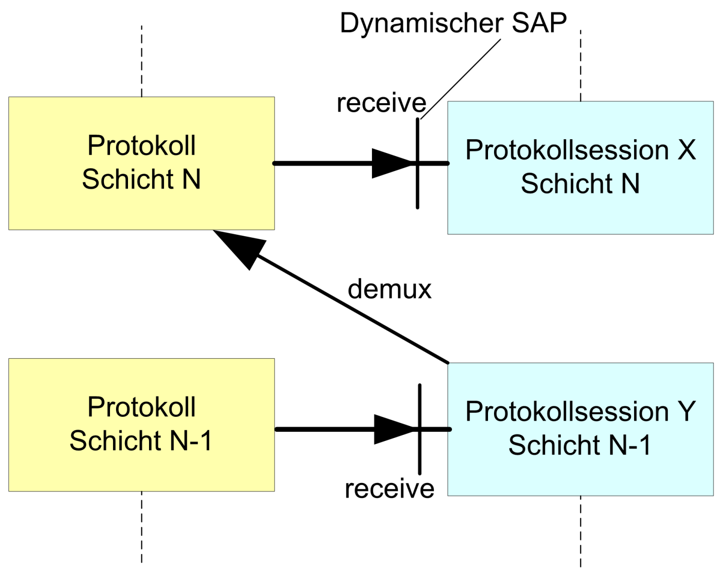

Abbildung 4.16: Dynamische SAP-Eigenschaft einer Protokollsession im Empfangsfall

onsfluß zwischen zwei Endsystemen detailliert beschrieben. Dieses Beispiel ist ausreichend, da jeder komplexe Informationsfluß aus einzelnen Folgen von unidirektionalen Informationsflüssen dargestellt werden kann. Das Beispiel wird in drei Phasen unterteilt, welche nachstehend erläutert werden.

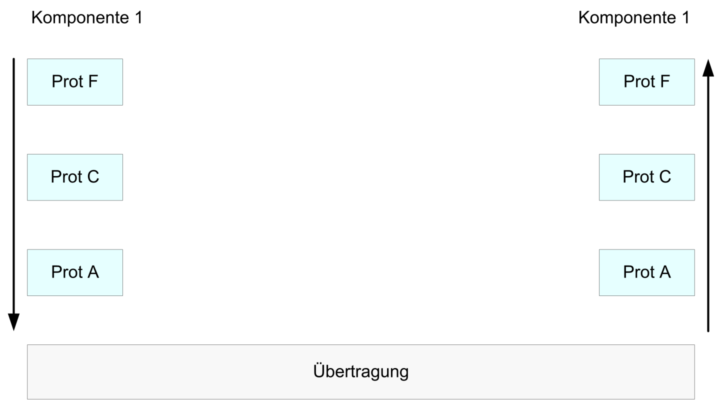

Abbildung 4.17: Dynamisches Verhalten Protokoll- und Protokollsession

1. **Sessioninitialisierungsphase**
   Bevor ein Nachrichtenaustausch zwischen den Partnerinstanzen des Protokolls *Prot F* stattfinden kann, müssen Sessionobjekte in der *Komponente 1* generiert werden. Wie zuvor erklärt, stehen Protokollsessions für dynamische SAPs, welche Sende- bzw. Empfangsfunktionalität besitzen. Um Nachrichten von der *Komponente 1* versenden zu können, müssen zuvor alle Protokollsessionobjekte angelegt werden und ihre Zuordnungen in den jeweiligen Sessionmaps der dazugehörigen Protokollobjekte eingetragen werden, um eine spätere Empfangsfunktion gewährleisten zu können. Die zu erstellenden Sessionobjekte werden durch rekursiven Aufruf der openSession-Methode der Protokollobjekte erzeugt, d.h. ein Protokoll der Schicht N

fordert eine Protokollsession bei dem Protokoll der Schicht N-1 an, welches wiederum entsprechend auf das Protokoll der nächsttieferen Schicht geht (Abbildung 4.18). Die so angeforder-

Abbildung 4.18: Sessioninitialisierung

ten Sessionobjekte werden daraufhin, beginnend bei der untersten Schicht, erzeugt und an die darüberliegenden Protokolle übergeben (Abbildung 4.19). Nach dem alle Sessionobjekte auf der Senderseite erstellt worden sind, können Protokollnachrichten versendet werden.

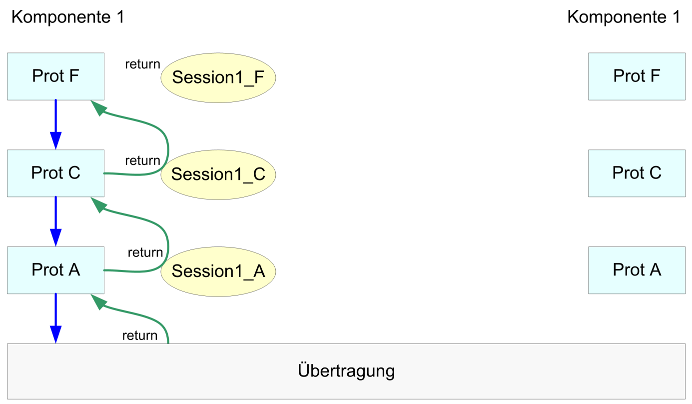

Abbildung 4.19: Sessioninitialisierung - Protokollsession-Rückgabe

2. **Sendephase**
   Am Ende der Initialisierungsphase sind alle für die Kommunikation nötigen Sessionobjekte der *Komponente 1* generiert und können nun für die weitere Kommunikation genutzt werden. Wie zuvor beschrieben, möchte die Protokollschicht F über die Schichten C und A eine Nachricht über das darunterliegende Verbindungsmedium zu ihrer Partnerinstanz auf *Komponente 2* senden.

   Beim Versendevorgang auf *Komponente 1*, bei welchem die Nachricht den Protokollgraphen von oben nach unten durchläuft, hat jedes Sessionobjekt Kenntnis über ihren jeweils direkten Nachfolger, welchen sie zuvor bei ihrer Generierung durch ihr dazugehöriges Protokollobjekt erhalten hat. Dadurch können die Sessionobjekte die Nachricht jeweils direkt, ohne bei ihrem

“Owner“-Protokoll nachzufragen (Sessionmap-Lookup), an die nächste darunterliegende Session weiterleiten. Dies geschieht über die send-Methode. Die Nachricht durchläuft so sukzessiv alle nötigen Protokollsessions (Abbildung 4.20). Abschließend werden sie an den Netzwerk-Adapter übergeben und von dort übertragen. Da der Netzwerk-Adapter in dem Modell der VIP-Library ausschließlich Sende- und Empfangsfunktionen ausführt, ohne Nachrichten zu verarbeiten, ist dieses Element in den Abildungen 4.17 - 4.23 mit dem Übertragungsblock zusammengefasst.

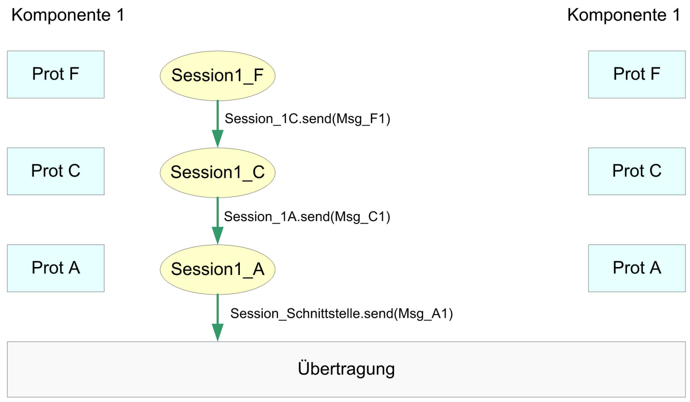

Abbildung 4.20: Sendephase

3. **Empfangsphase**
   Wenn die Nachricht über das Verbindungsmedium den Netzwerk-Adapter der *Komponente 2* erreicht, übergibt dieser die Nachricht an das Protokoll der untersten Schicht, dem Interfaceprotokoll. Dies geschieht durch Aufruf der demux-Methode des Interfaceprotokolls, indem dieser die zu übertragende Nachricht übergeben wird. Der Sonderfall, dass es mehrere Protokolle auf der untersten Schicht gibt und somit Zuordnungsprobleme entstehen können, ist in der VIP-Library nicht möglich. Dies stellt jedoch keine Einschränkung dar, wie in Abschnitt 4.2.1.1 bereits erläutert wurde.
   Abbildung 4.21 zeigt diesen Zusammenhang auf seiten der *Komponente 2*. Nach der Übergabe

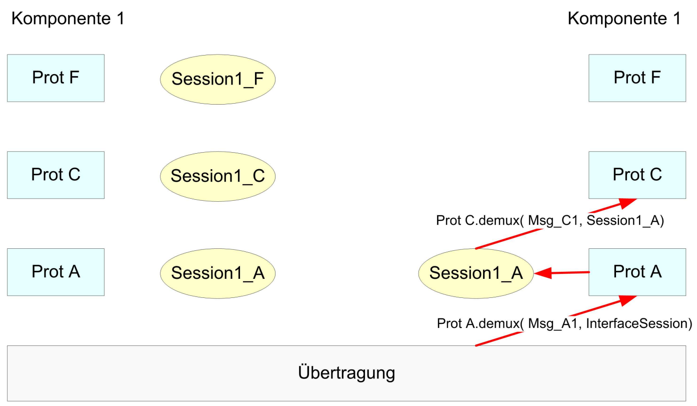

Abbildung 4.21: Empfangsphase

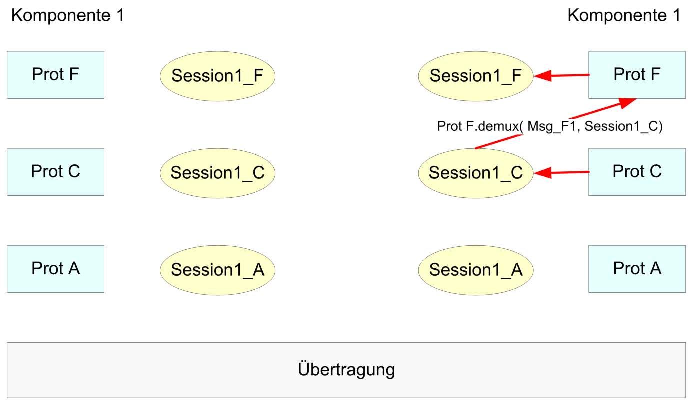

Abbildung 4.22: Empfangsphase

der empfangenen Nachricht an das Protokollobjekt führt dieses einen Sessionmap-Lookup, bezogen auf die Nachricht, durch. Wird beim Lookup festgestellt, dass für diese Nachricht noch kein Sessionobjekt existiert, wird ein Sessionobjekt mittels der openSession-Methode angelegt und in der Sessionmap eingetragen. Danach wird die Nachricht an dieses Sessionobjekt übergeben. Das Zusammenspiel von demux-Aufruf, Sessionmap-Lookup, openSession (falls noch keine Session existiert) und Übergabe der Nachricht an das Sessionobjekt wiederholt sich auf jeder weiteren Protokollschicht (Abbildung 4.22), bis die Nachricht bei dem Zielprotokoll angekommen ist. Nach diesem Durchlauf stehen sowohl auf seiten der *Komponente 1* als auch der *Komponente 2* alle Sessionobjekte zur Verfügung (Abbildung 4.23). Ab diesem Zeitpunkt können Nachrichten in beiden Richtungen über die bestehenden Protokollsessions übertragen werden.

Abbildung 4.23: Endzustand nach beidseitiger Sessioninitialisierung

4.2.2 Design – VIP-Library (Phase 2)

Die zweite Designphase kann als Anpassungsphase betrachtet werden. Die bis zum jetzigen Zeitpunkt entworfenen Grundobjekte sind allgemein gehalten, ohne irgendwelchen Bezug auf die Funktionalität des DaSSF-Simulationskerns zu nehmen. Besonders das Kommunikationsmodell des DaSSF-

Kerns, Nachrichten über geschaltete Kanäle, sog. Channels (Abschnitt 1) zu übermitteln, verändert das Grundmodell des Entwurfes aus der ersten Designphase. Zusätzlich werden in dieser Phase die in der Anforderungsspezifikation geforderten Breakpoint- und Loggingmechanismen modelliert.

Wie in Abschnitt 3.2.1 beschrieben, läuft die Kommunikation in jeder DaSSF-basierten Simulation über *Entity*-Objekte. Möchte man eigene Simulationsobjekte einführen, die aktiv an einer Simulation teilnehmen können, so müssen diese von der Klasse *Entity* abgeleitet werden. Alle nun beschriebenen Objekte der Designphase 2 sind Entity-Unterobjekte.

4.2.2.1 Komponente

Bevor Anpassungen an die Struktur des Komponentenobjektes gemacht werden können ist es notwendig, den Portbegriff in Bezug auf DaSSF zu erweitern. In funktionaler Hinsicht ist ein Port ein Weiterleitungselement, das beidseitig Nachrichten empfangen und versenden kann. Die interne Sei-

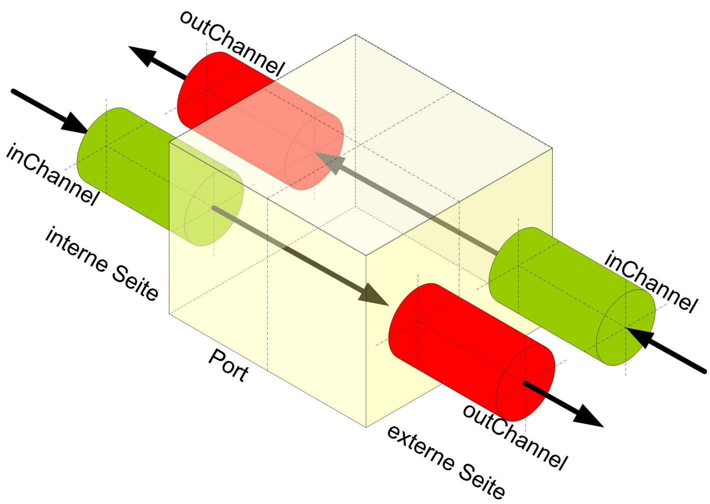

Abbildung 4.24: Logischer Aufbau eines Ports

te eines Ports ist diejenige, welche an Komponenten (Netzwerk-Adapter) bzw. Konnektoren gekoppelt ist. Dagegen ist die externe Seite an ein Verbindungsmedium (z.B. Kabel) angeschlossen. Diese Funktionalität wird nun unter DaSSF auf die Kommunikation über sog. Channels abgebildet. Ein Port besitzt sowohl auf der internen als auch auf der externen Seite je ein Kanalpaar (Abbildung 4.25).

Ein solches Kanalpaar besteht jeweils aus einem inChannel, über den Daten empfangen werden können und einem outChannel, über den Daten gesendet werden können (Abbildung 4.24). An jedem inChannel wartet ein sog. Listener, realisert durch ein DaSSF-Processobjekt, auf eintreffende Nachrichten. Eingetroffene Nachrichten werden vom Listener an den zugehörigen Worker-Process weitergegeben. Dieser leitet die eingetroffene Nachricht unverändert an den outChannel der anderen Seite. Wie eben ausgeführt, ist das Portobjekt das universelle Koppelelement.

Das eben beschriebene Ablaufmodell des Ports zwischen inChannel, Listener, Worker und outChannel ist das Grundprinzip jeder Kommunikationseinheit der VIP-Library. Jedes Objekt der VIP-Library, welches einen Datenfluß in einer bzw. zwei Richtungen zuläßt, folgt diesem Prinzip. An jedem Eingangskanal (inChannel) wartet ein Listener (Process), der eingehende Daten entgegennimmt und an

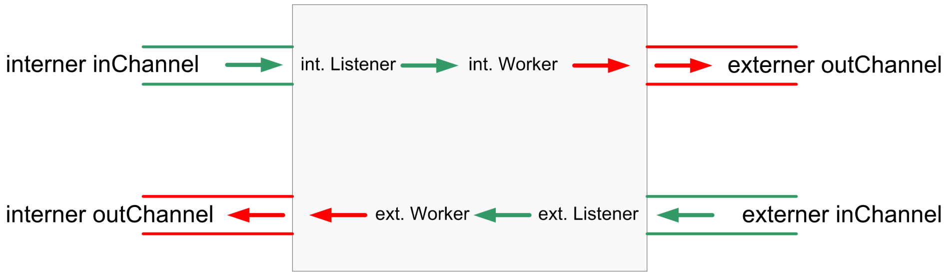

Abbildung 4.25: Logische Funktionsweise eines Ports

einen Worker weiterleitet. Dieser Worker gibt die Daten bearbeitet oder unberührt an den passenden outChannel wieder aus. Im weiteren Verlauf der Diplomarbeit wird immer davon ausgegangen, wenn ein Objekt einen inChannel besitzt, besitzt es grundsätzlich auch einen dazugehörigen Listener.

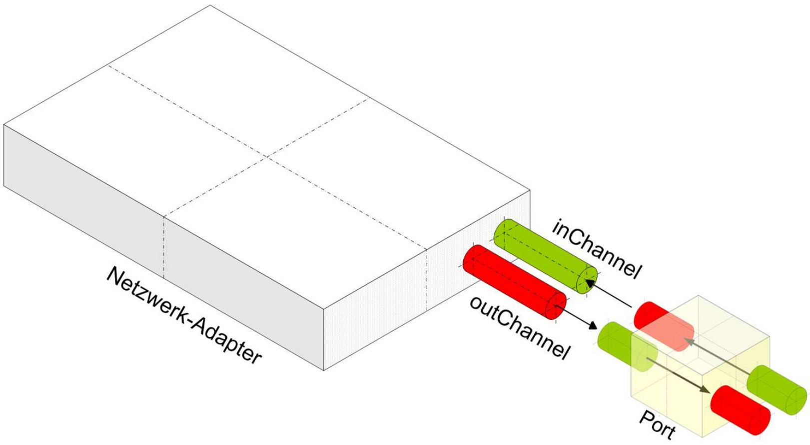

Abbildung 4.26: Logischer Aufbau eines Netzwerk-Adapters

Aufgrund dieser Veränderung des Portobjektes, welches das zentrale Verbindungselement ist, muss der Netzwerk-Adapter ebenfalls angepasst werden. Wie in Abschnitt 4.2.1.1 eingeführt, besitzt ein Netzwerk-Adapter für die Kommunikation mit dem Rechnernetz nach aussen einen Port. Damit nun dieser Port mit seinen Channels an den Netzwerk-Adapter gekoppelt werden kann, muss dieser wiederum ein Kanalpaar, bestehend aus einem in- und einem outChannel, besitzen. Dieses Paar wird (Abbildung 4.26) an die interne Seite des Ports gemappt. Der Adapter kann über den Port Daten nach aussen mit den Kommunikationspartnern austauschen. Die Art und Weise des internen Datenaustausches mit dem implementierten Protokollgraphen bzw. dem Interfaceprotokoll wird in Abschnitt 4.2.2.7 ergänzt, da hierfür die Grundkommunikation zwischen Protokoll und Protokoll-Session bezogen auf den DaSSF-Kern, angepasst werden muss.

4.2.2.2 Konnektor

Wie in Designphase 1 beschrieben, besitzt ein Konnektor mehrere Ports, deren Funtionsweise in Abschnitt 4.2.2.1 bereits erläutert wurde. Wesentliche Eigenschaft eines Konnektors ist das Vorhanden-

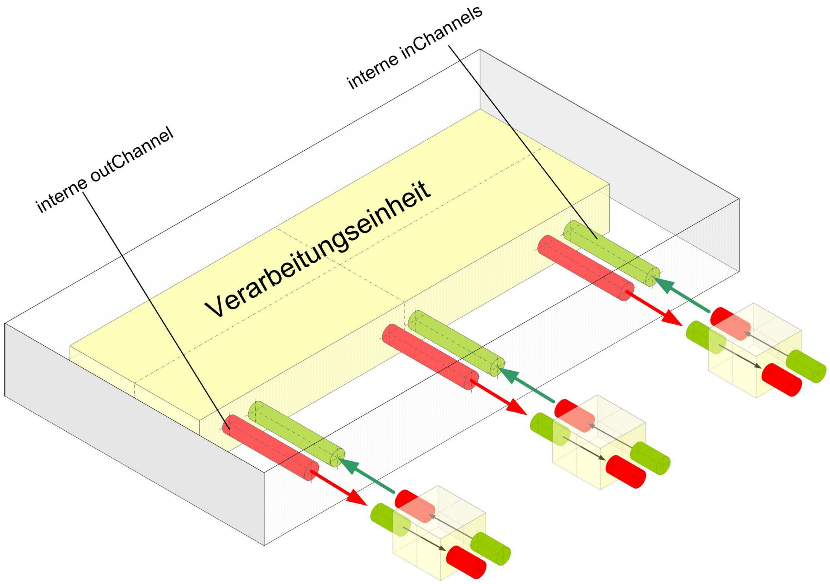

Abbildung 4.27: Logischer Aufbau eines Konnektors

sein mehrerer Ports. Um diese Ports an den Konnektor zu binden, besitzt dieser pro Port je einen internen in- und outChannel. Alle internen outChannels der Ports werden auf die zugehörigen inChannels des Konnektors gemappt. An jedem internen inChannel des Konnektors wartet ein Listener auf die eintreffenden Nachrichten, um diese an eine Verarbeitungseinheit (Worker) mit der Quellportangabe weiter zu leiten. Die Logik dieser Verarbeitungseinheit (Abbildung 4.27) bestimmt die Funktionalität des Konnektors. Leitet die Verarbeitungseinheit die eintreffende Nachricht an alle verfügbaren internen outChannels weiter, so handelt es sich um einen Hub. Durch die Modellierung mit je einem Listener pro internen inChannel ist es möglich eine Zuordnung von Eingangsport und empfangener Nachricht vorzunehmen. Diese Zuordnung ist für die weitere Behandlung der Nachricht in der Verarbeitungseinheit wichtig, da es Kopplungselemente gibt, die diese Information für ihre Funktionsweise benötigen, z.B. für die Port-Mac-Zuordnungstabelle eines Switches.

4.2.2.3 Verbindungsmedium

Die Funktionsweise eines Verbindungsmediums ist mit dem eines Ports vergleichbar. Es besitzt wie dieser an jedem Ende ein Kanalpaar, bestehend aus einem inChannel und einem outChannel. Im Gegensatz zu einem Port, dessen Worker Daten nur weiterleitet, werden Daten die über das Verbindungsmedium transportiert werden, von dessen Worker behandelt (Abbildung 4.28). Hierbei können Störeinflüsse aller Art nachgebildet werden, beginnend von Nachrichtenstörungen (Paketveränderung) bis hin zum Kabelbruch (Paketvernichtung). Ein Verbindungsmedium kann nur zwischen Ports geschaltet werden. Abbildung 4.29 zeigt eine mögliche Zusammenschaltung der vorgenannten Objekte. Für das Verbindungsmedium ist es gleichgültig, welche Objekte damit gekoppelt werden, da ihre Schnittstellenobjekte in allen Fällen immer Ports sind (Interface- oder Konnektorports).

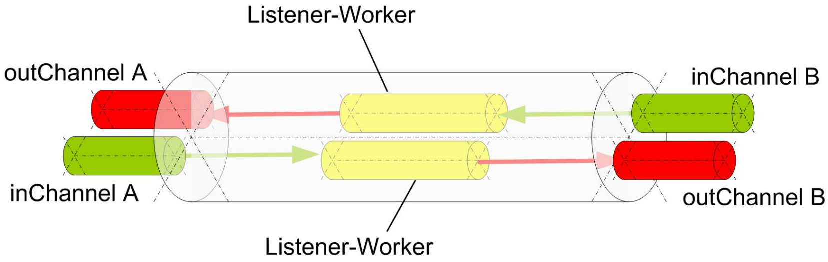

Abbildung 4.28: Logischer Aufbau eines Übertragungsmediums

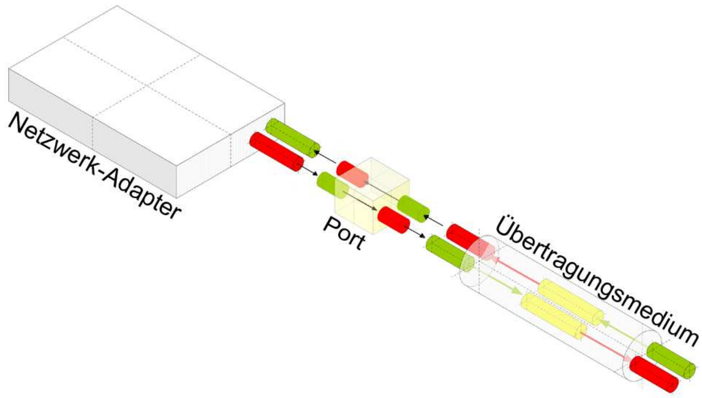

Abbildung 4.29: Zusammenspiel aller Komponenten

4.2.2.4 Protokollgraph

Das Protokollgraphenobjekt bleibt gegenüber der Designphase 1 unverändert. Da der tatsächliche Kommunikationsablauf über die Protokoll- bzw. Protokoll-Sessionobjekte abgewickelt wird und der dazugehörige Protokollgraph davon unberührt ist, muss dieser von der Simulation aus auch nicht erreichbar sein. Aufgrund dessen ist eine Ableitung von dem Entityobjekt nicht notwendig.

4.2.2.5 Protokollnachricht

Das in Designphase 1 entwickelte Modell der Nachrichtentypen kann von ihrem Funktionsumfang bestehen bleiben. Als einzige Änderung, die sich durch die Hinzunahme des DaSSF-Kerns ergibt, ist das Ableiten der Nachrichtenobjekte von der Klasse Event zu nennen, welches als Grundobjekt im Informationsaustausch zwischen Entities über Kanäle benutzt werden muss.

4.2.2.6 Protokoll und Protokollsession

Bei der Designerweiterung um den DaSSF-Kern zeigen sich die grössten Auswirkungen bei den Objekten Protokoll und Protokoll-Session. Wie schon in der Designphase 1 betont wurde, bilden diese beiden Objekte die Hauptfunktionalität der Objektstruktur. Jeder Kommunikationswunsch muss über diese Objekte abgewickelt werden. Die Änderungen wirken sich aber nicht auf den in Designphase 1 beschriebenen Nachrichtenfluß aus, sondern auf die an der Kommunikation beteiligten Objekte. Wir haben es hier mit einem Bedeutungswechsel des Begriffes *Objekt* zu tun. Es handelt sich nicht mehr um klassische Objekte, wie wir sie aus der objektorientierten Programmierung kennen, wo Nachrichtenaustausch mittels Aufruf bestimmter Methoden des Kommunikationspartners durchgeführt wird. Es handelt sich nunmehr um Objekte im Sinne von DaSSF, bei denen der Nachrichtenaustausch zwischen DaSSF-Objekten über Channels abgewickelt wird. Die Eigenheit liegt nun darin, dass im Gegensatz zum klassischen Nachrichtenaustausch, DaSSF-Objekte ihren direkten Kommunikationspartner nicht notwendigerweise kennen müssen. Um Nachrichten versenden zu können, werden diese einfach auf einen passenden outChannel des sendewilligen Entity-Objektes gelegt. Wohin dann die Nachrichten geleitet werden, hängt von dem inChannel ab, welcher aktuell an diesen Ausgang gemappt ist. Dieses Mapping kann natürlich auch unabhängig vom Sender in einem anderen Objekt für diesen transparent geschehen.

In der Anpassungsphase an den DaSSF-Kern stellt sich nun die Frage, welche Teilabschnitte des gesamten Kommunikationsweges durch Channels ersetzt werden sollen.
Servicebeziehungen im ISO/OSI-Modell beruhen immer auf dem Prinzip, dass Schicht N-Protokolle nur den Schicht N+1-Protokollen ihre Dienste anbieten. Diese Vereinbarung, Services immer nur von "unten" nach "oben" anzubieten, war maßgeblich für die Konzeption der Channelverbindungen zwischen Protokoll-Session- und Protokollobjekten der VIP-Library. Kanäle werden nur dort benutzt, wo SAP-Beziehungen vorhanden sind. Es bestehen folgende SAP-Beziehungen zwischen Protokoll- und Protokoll-Sessionobjekten:

1. Protokoll-Sessions bieten ihren Sendedienst Sessions der darüberliegenden Protokoll-Schicht an (Abbildung 4.15)

2. Protokoll-Sessions bieten ihren Empfangsdienst ihrem dazugehörigen Protokoll an (Abbildung 4.16)

Die erste Situation tritt nur im Fall der "Downstream"-Kommunikation auf. Dementsprechend wird der in Abbildung 4.19 beschriebene Ablaufmechanismus angepasst, indem Protokoll-Sessionobjekte jeweils einen in- und outChannel für diese Kommunikationsrichtung nutzen. Abbildung 4.30 zeigt diese Neuerung des Kommunikationsverlaufes. Die Verarbeitung beruht bei diesen beiden Kanälen wieder auf dem Listener-Worker-Prinzip, wobei hier dem Worker eine wichtige Rolle zukommt. Er realisiert die wirkliche protokollspezifische "Downstream"-Logik des dazugehörigen Protokolls. So führt z.B. der Worker der Vermittlungs-Schicht in der Beispielsimulation mit dem synthetischen Protokoll IPmin (Abschnitt 4.1.2.3) als einzige Aktion das Anfügen des protokolleigenen IPmin-Headers durch und legt die so ergänzte Nachricht mittels seiner "send"-Methode auf seinen Session-outChannel.

Im Gegensatz dazu tritt der zweite Fall in der "Upstream"-Kommunikation auf. Damit ist klar, dass in diesem Fall die Protokoll-Sessionobjekte einen "Protokoll"-inChannel erhalten, welche auf einen outChannel des dazugehörigen Protokollobjektes gemappt werden. Entsprechend erhält nach dem Listener-Worker-Designprinzip das Protokoll-Sessionobjekt einen Listener sowie einen dazugehörigen Worker. Aufgabe dieses Workers ist es wiederum protokollspezifische Nachrichtenbehandlungen, wie das Entfernen von Headern, durchzuführen und die Nachricht via Callback an das darüberliegende Protokollobjekt weiterzuleiten. Im Gegensatz dazu benötigt das Protokollobjekt keinen Listener, da ihm die Nachrichten nicht über Kanäle, sondern mittels Callback übergeben werden. Abbildung 4.31 zeigt das Ergebnis dieser Überlegung.

Die eben dargestellten Ablaufkonzepte beschreiben nur den Weg der Nachrichten vom Sender zum Empfänger im Zusammenspiel mit den Protokollen und deren Sessionobjekten. Die Sessiongenerierung mittels der in Abschnitt 4.2.1.7 beschriebenen "open"-Methode bleibt vom Ablaufverhalten un-

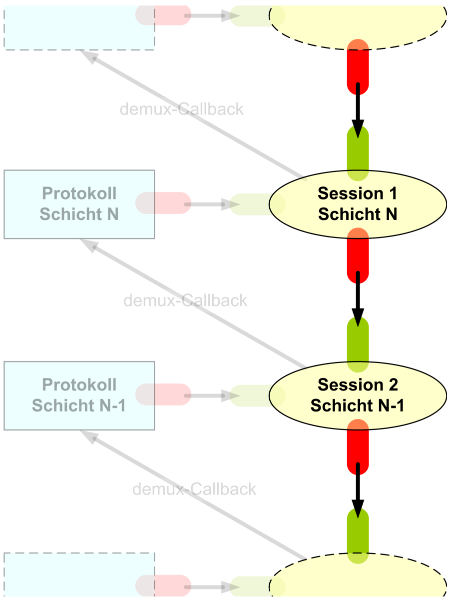

Abbildung 4.30: Downstream-Zusammenspiel Protokoll – Protokoll-Session – DaSSF

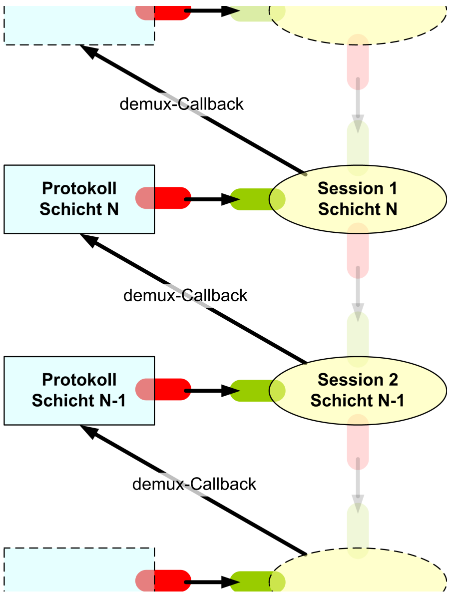

Abbildung 4.31: Upstream-Zusammenspiel Protokoll – Protokoll-Session – DaSSF

verändert. Da aber nun Protokolle bzw. deren Sessionobjekte über Kanäle Daten austauschen, müssen diese in der Initialisierungsphase auf der Sende- bzw. Empfangsseite dynamisch gemappt werden. Dieses Mapping wird nun für den Anwendungsentwickler transparent von der VIP-Library übernommen.

4.2.2.7 Netzwerk-Adapter

In Abschnitt 4.2.2.1 wurde bereits die Schnittstelle des Netzwerk-Adapters zur Außenwelt modelliert. Im Folgenden wird nun die Koppelung an den Protokollgraphen einer Komponente erklärt.
Um in der Systematik der bisherigen Protokollmodellierung zu bleiben und die dort modellierten

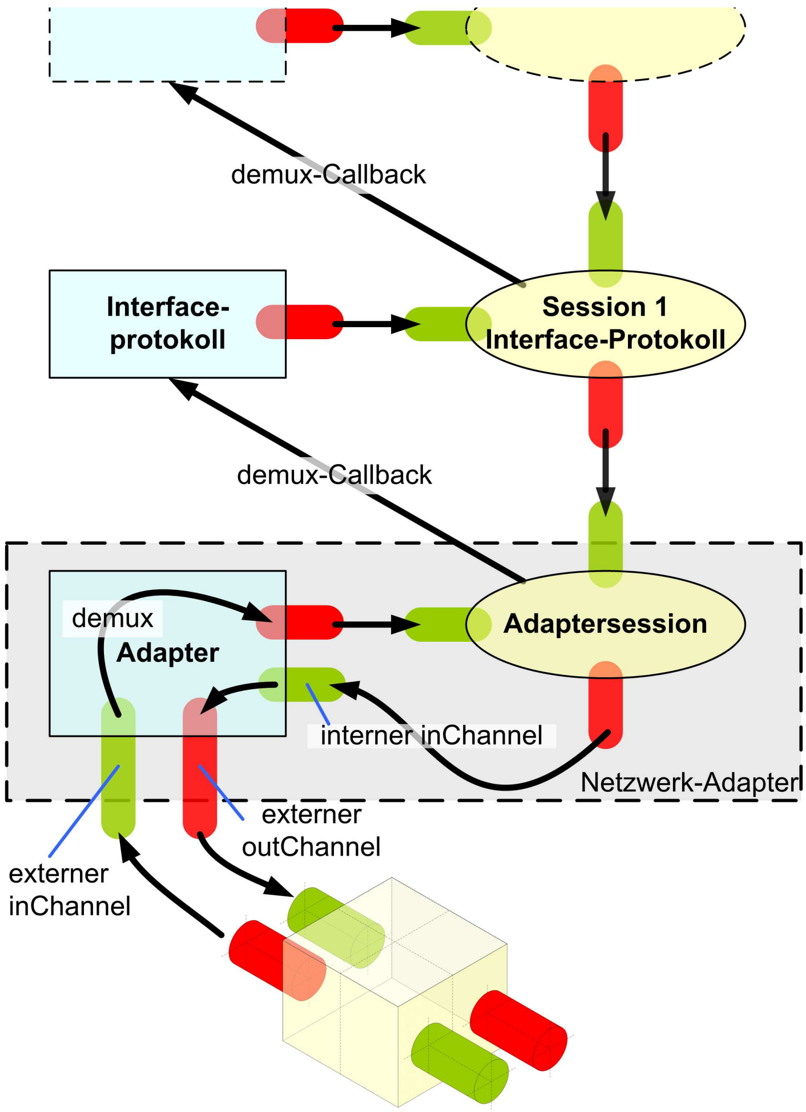

Abbildung 4.32: Netzwerk-Adapter

Schnittstellen weitgehend wiederverwenden zu können, wird der Netzwerk-Adapter ebenfalls in einen Protokoll- und Protokollsessionteil aufgeteilt. Der protokollspezifische Teil wird als Adapter bezeichnet, der protokollsessionspezifische als Adaptersession. Dadurch können nach "oben" die Standardprotokollmechanismen benutzt werden. Das Interfaceprotokoll kann mittels openSession eine Protokollsession bei dem Adapter anfordern. Die Besonderheit hierbei ist, dass ein Adapter im Gegensatz zu einem Protokoll immer nur genau eine Adaptersession verwaltet. Jeglicher Nachrichtenverkehr aus dem darüberliegenden Protokollstack wird über diese immer vorhandene Adaptersession abgewickelt. Die Schnittstelle des Adapters nach außen mit je einem externen in- und outChannel bleibt, wie in Abschnitt 4.2.2.1 beschrieben, unverändert. Eingehende Nachrichten aus dem externen inChannel werden vom Listener an die demuxCallback-Methode des Adapters (Abbildung 4.32) übergeben. Diese leitet dann die Nachricht nach dem Standardprotokollverfahren mittels Sessionmap-Lookup an die Adaptersession weiter. Die Adaptersession übergibt wiederum die Nachricht via demux-Callback an das Protokoll der nächsthöheren Schicht, das Interfaceprotokoll. Ausgehende Nachrichten werden von der Adaptersession an den internen inChannel des Interfaces gegeben. Der Listener übergibt daraufhin die Daten an den externen outChannel, welcher an den Port angeschlossen ist.

4.2.2.8 Anwendungen

Anwendungen verhalten sich genau wie Protokoll- und Protokoll-Sessionobjekte und werden deshalb in der VIP-Library gleich behandelt. Diese Vereinfachung läßt sich leicht am Beispielszenario aus Abbildung 4.17 erklären. Ob das dort auf oberster Schicht liegende Objekte *Prot F* nun als Protokoll oder als Applikation angesehen wird, ist im Sinne des Modells nicht von Bedeutung. Ihr Verhalten in der Kommunikation eines Protokollgraphen ist identisch.
Als einzige Ergänzung benötigt ein Applikationsobjekt eine Möglichkeit, aktiv Kommunikationen anzustoßen. Dieses Verhalten ist aber wiederum applikationsabhängig und somit nicht Teil der Objektstruktur.

4.2.3 Design – Simulationsinterface

Das Aufteilen des Designs des Simulationsinterfaces in zwei Phasen, wie zuvor in der VIP-Library, ist nicht sinnvoll, da eine Betrachtung des Simulationsinterfaces ohne den DaSSF-Simulationskern sinnlos ist. Dieser ist essentiell für den Entwurf dieser Schnittstelle.
Bevor die Konfigurations- und Steuerungsschnittstelle beschrieben werden kann, muß zuvor auf das Logging- und Breakpointhandling eingegangen werden, da diese Funktionalität Teil dieser Schnittstellen sein wird.

4.2.3.1 Breakpoint- und Loggingmechanismus

Breakpoint- und Loggingmechanismen sind von der Grundstruktur her gleich zu betrachten. Sie erlauben einem Benutzer, bestimmte Stellen einer Simulation vorab festzulegen, an denen bestimmte Aktionen ausgeführt werden sollen. Im Falle eines Breakpoints wird die Simulation pausiert und im Falle des Loggings wird eine bestimmte Simulationsinformation gespeichert. Aufgrund dieser Tatsache wird ein Oberobjekt modelliert, welches die Funktionalität des Festlegens einer Stelle in einer Simulation unterstützt, die auszuführende Aktion aber variabel hält. Die makierten Punkte werden als Aktionspunkte bezeichnet.
Die Flexibilität beim Festlegen dieser Haltepunkte ist natürlich nicht vergleichbar mit marktüblichen Debuggingsystemen aus dem Bereich der Softwareentwicklung, bei denen auf Quellcodeebene an beliebigen Stellen Aktionspunkte definiert werden können. In der Simulationsumgebung wird mit kompilierten Objekte gearbeitet, welche eine derartige Funktionalität nicht zulassen.

Um dieses Modell umsetzen zu können, müssen nun für alle Objekte Gemeinsamkeiten gefunden werden, welche sowohl für das Setzen dieser Aktionspunkte interessant als auch programmiertechnisch umsetzbar sind. Aus dieser Betrachtung heraus ergeben sich zwei grundlegende Ansätze:

1. Breakpoint- bzw. Loggingmechanismen auf Zustandsänderungen
2. Breakpoint- bzw. Loggingmechanismen auf DaSSF-Eventbasis

Beide Mechanismen werden im Rahmen der VIP-Library realisiert.

In einer DaSSF-basierten Simulation müssen Objekte, um für den Simulationskern ansprechbar zu sein, von der Klasse Entity abgeleitet sein. Um Logging- und Breakpointmechanismen in Simulationsobjekten anbieten zu können, muß der Simulationskern Zugriff auf diese Objekte haben. Deshalb macht es nur Sinn, in Objekten, welche von Entity abgeleitet sind, diese Mechanismen zu implementieren. Andernfalls hätte der DaSSF-Kern darauf keinen Zugriff. Deshalb ist es sinnvoll, die Objekte nicht mehr von der Klasse Entity direkt abzuleiten, sondern eine Klasse zwischenzuschalten, die Logging- und Breakpointfunktionalitäten implementiert hat. Diese Klasse wird als LoggingEntity bezeichnet. Alle davon abgeleiteten Objekte der VIP-Library haben damit Logging-, Breakpoint- und Entityfunktionalitäten.
Die Tabelle 4.6 zeigt die Funktionalität der LoggingEntity-Klasse aus Sicht eines Entwicklers, Tabelle 4.7 aus Sicht eines Anwenders.

Funktionalität	**Beschreibung**
setEventBreakpoint / setEventLoggingpoint	Festlegen einer Stelle als Aktionspunkt für das DaSSF-eventbasierte Logging- bzw. Breakpointverfahren
setStateBreakpoint / setStateLoggingpoint	Festlegen einer Stelle als Aktionspunkt für das zustandsbasierte Logging- bzw. Breakpointverfahren

Tabelle 4.6: LoggingEntity aus Entwicklersicht

Funktionalität	**Beschreibung**
activateEventLogging / activateEventBreakpoint	Aktivierung des DaSSF-eventbasierten Logging- bzw. Breakpointverfahrens
activateStateLogging / activateStateBreakpoint	Aktivierung des zustandbasierten Logging- bzw. Breakpointverfahrens

Tabelle 4.7: LoggingEntity aus Anwendersicht

4.2.3.2 Konfiguration und Steuerung

Die Konfiguration und Steuerung einer Simulation ist eine Funktionalität, welche Anwendern zur Verfügung gestellt wird. Sie unterstützt Anwender bei der Erstellung von Simulationen. Zur Definition dieser Schnittstelle sollten Datenstrukturen verwendet werden, die für den Anwender leicht verständlich sind, eine einfache Prüfung auf Einhaltung von Regeln erlauben und mit wenig Aufwand umsetzbar sind. Dazu bietet sich an, XML als Beschreibungs- bzw Konfigurationsdefinitionssprache zu verwenden. Definitionen des Anwenders bei der Erstellung einer neuen Simulation können mittels **Document Type Definition** (**DTD**) auf Regeleinhaltung geprüft werden.

Die Schnittstelle wird in zwei Teile aufgeteilt. Auf unterer Ebene liegt die tatsächliche Schnittstelle, welche die Funktionalität zum Konfigurieren bzw. Steuern bereitstellt. Zusätzlich zu diesem "Schnittstellen-Befehlssatz" (Abbildung 4.33) wird ein darauf aufsetzender XML-Parser modelliert, welcher dem Userinterface eine einfache Aktionsschnittstelle zum Simulationsinterface anbietet.

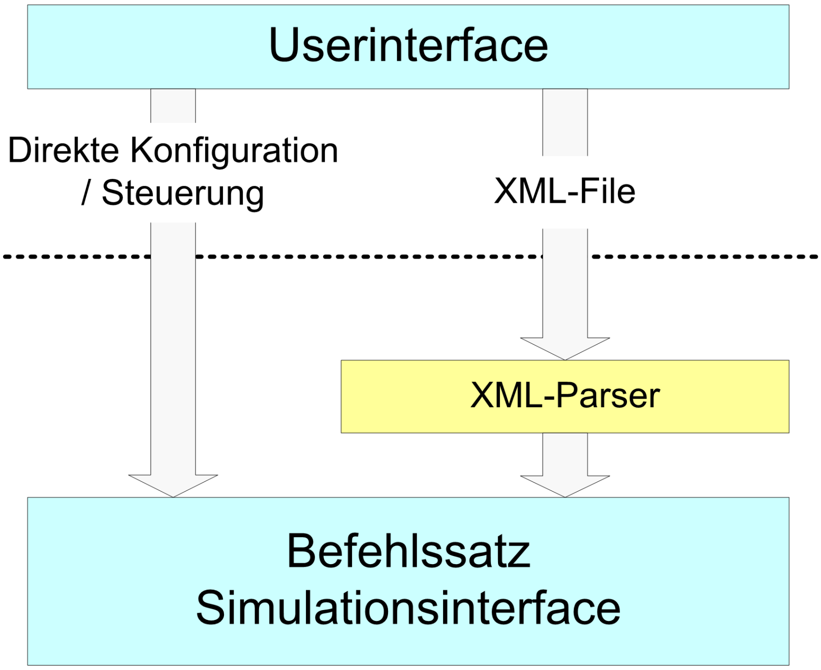

Abbildung 4.33: Schematischer Überblick der XML-Schnittstelle des Simulationsinterfaces

Dieser Parser setzt auf eine Datenstruktur in XML-Format auf, mit der Anwender, ohne direkt mit der Simulationsschnittstelle zu kommunizieren, ihre zu erstellende Simulation formulieren. Dieses Design der Simulationsschnittstelle hat den großen Vorteil, dass zur Prüfung der Benutzereingaben die Validierungstechniken eines XML-Parsers genutzt werden können.

4.2.4 Design – Beispielsimulation VIP_TIME

Eine Designphase ist hier nicht notwendig, da in der Anforderungsspezifikation die zu erstellende Simulation schon ausreichend modelliert wurde.

4.3 Implementierung

In diesem Abschnitt der Diplomarbeit werden die grundlegenden Funktionalitäten der im Design modellierten Elemente an der Implementierung erläutert. Unter Verwendung von UML-Diagrammen [Oest 01a] wird deren Aufbau gezeigt, wobei dabei nur aus dem Diagramm nicht ersichtliche Eigenschaften erläutert werden. Für weitergehende Informationen steht als Anlage die detaillierte Klassendokumentation zur Verfügung.
Der Begriff der VIP-Library aus dem Design wird nun in der Umsetzung als VIPAPI bezeichnet.

4.3.1 Klassenübersicht der VIPAPI

Die Abbildung 4.34 zeigt alle in der VIPAPI enthaltenen Klassen und deren Vererbungshierarchie. Diese Klassen stehen dem Entwickler für die Implementierung neuer Simulationsobjekte zur Verfügung. Um neue Anwendungsobjekte zu erstellen, muss ein Entwickler lediglich die VIPAPI-Headerdatei (vipapi.h) includen, um die Klassen der VIPAPI verwenden zu können.

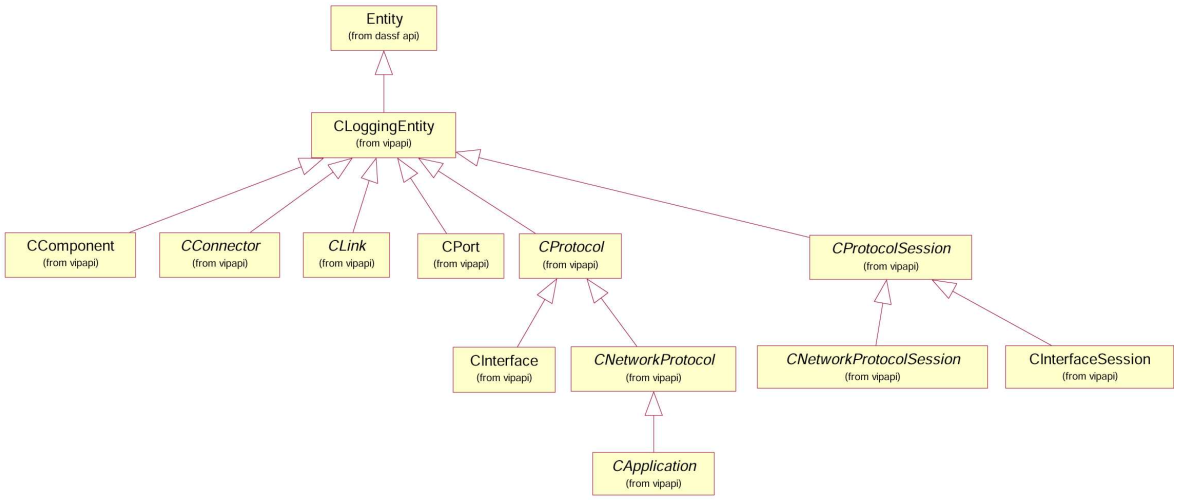

Abbildung 4.34: Übersicht der Klassen der vipapi

4.3.1.1 CComponent

Komponenten werden in der VIPAPI durch die Klasse CComponent umgesetzt (Abbildung 4.35). Wie im Design beschrieben, ist eine Komponente ein reduziertes Objekt, welches nur kommunikationsrelevante Elemente enthält. Da Elemente wie Arbeitsspeicher, Prozessor usw. nicht darin enthalten sind, ist diese Klasse eine reine Containerklasse für Netzwerk-Adapter und Protokollgraph. Davon abgeleitete Klassen müssen keine besonderen Methoden implementieren (keine virtual-Methoden).

4.3.1.2 CConnector

Konnektoren werden in der VIPAPI durch die Klasse CConnector implementiert (Abbildung 4.36). Die Funktion des Workers (Designphase) wird durch die virtuelle Methode `processMessage` umgesetzt. Jeder von dieser Klasse abgeleitete Konnektor muss diese Methode implementieren. Als Paramenter bekommt die Methode die Nummer des Ports der eintreffenden Nachricht und die erhaltene Nachricht selbst. Die processMessage-Methode ist dafür zuständig, nach Verarbeitung der Nachricht diese an die passenden Ports weiterzuleiten. Für dieses Weiterleiten stehen ihr die Methoden `sendToAll` und `sendToPort` standardmäßig zur Verfügung. Mittels dieser beiden Sendefunktionalitäten ist es möglich, jegliche Kombination von Weiterleitungsroutinen in der Verarbeitungseinheit zu realiseren. Weitere wichtige Methoden dieser Klasse sind `connectComponent` bzw. `connectConnector`. Mittels dieser Methoden ist es möglich Komponenten bzw. Konnektoren an einen angegebenen Port durch ein Verbindungsmedium anzuschließen.

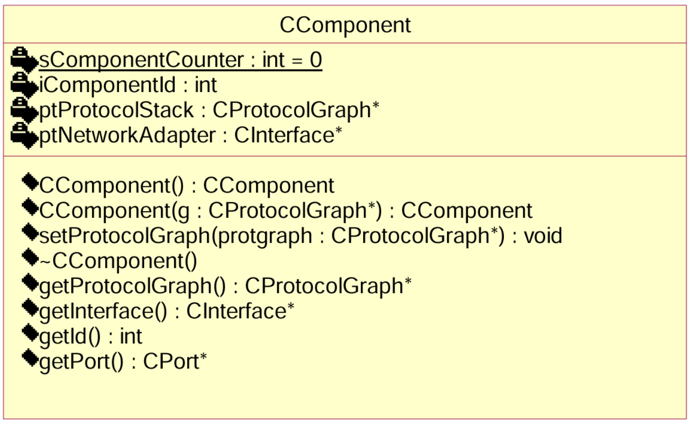

Abbildung 4.35: Übersicht der Klasse CComponent

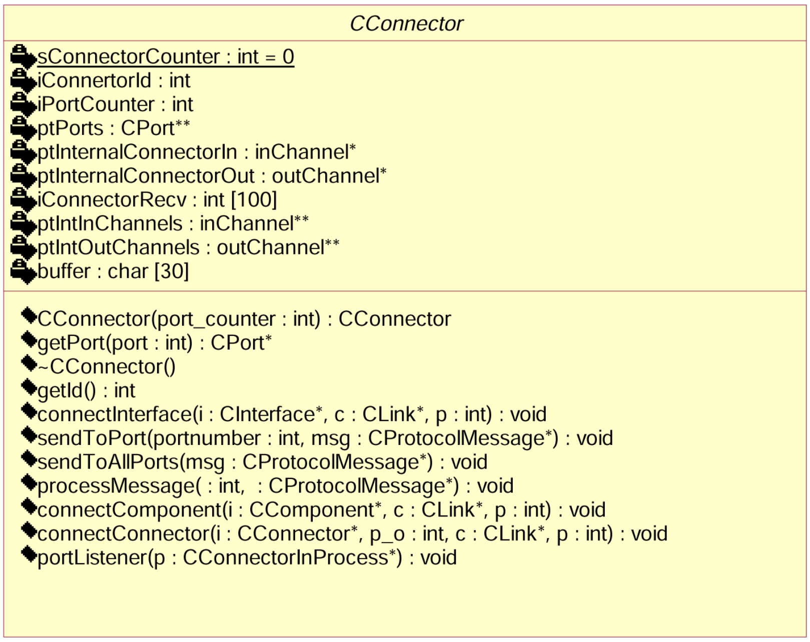

Abbildung 4.36: Übersicht der Klasse CConnector

4.3.1.3 CLink

Verbindungsmedien werden in der VIPAPI durch die Klasse CLink implementiert (Abbildung 4.37). Ähnlich wie bei den Konnektoren besitzt CLink ebenfalls eine virtuelle Methode `processMessage`, welche in abgeleiteten Klassen umgesetzt werden muss. Diese Methode hat aber im Gegensatz zu der der Konnektoren, welche Weiterleitungsfunktionalität besitzt, nur reine Nachrichtenbehandlungsfunktionalität. Dies ist auch verständlich, da wie im Design modelliert, ein Verbindungsmedium jeweils nur einen Ein- und Ausgang besitzt und somit eine unterschiedliche Ausgangsbehandlung (im Gegensatz zum Konnektor) nicht erforderlich ist.

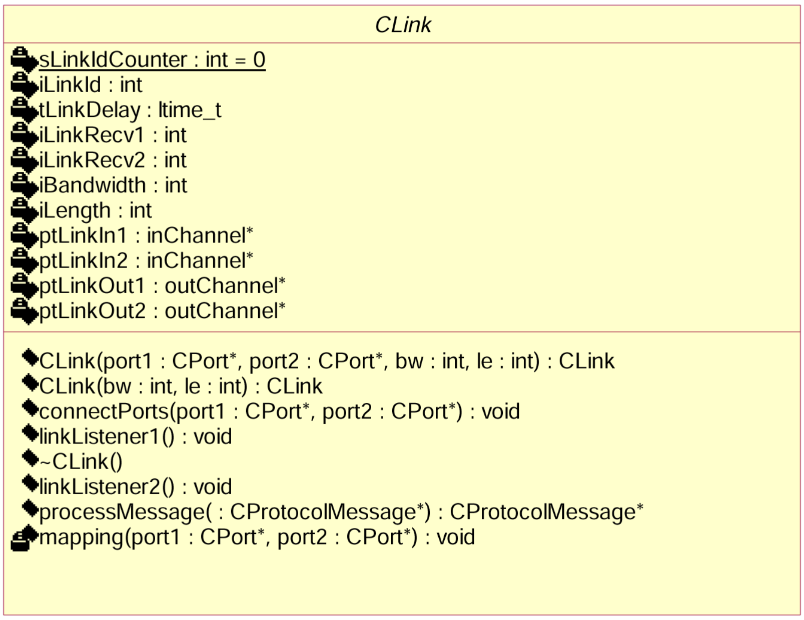

Abbildung 4.37: Übersicht der Klasse CLink

4.3.1.4 CPort

Die in der Designphase eingeführten Ports werden durch die Klasse CPort implementiert (Abbildung 4.38). Da ein Port keine Verarbeitungsfunktionalitäten gegenüber der Nachricht besitzt, ist als einzige wichtige Eigenschaft eine Nachrichtenverzögerungszeit implementiert. Mittels Konstruktor ist es möglich, einem Port ein sog. Delay mitzugeben. Nachrichten, die den Port durchlaufen, werden dementsprechend verzögert weitergeleitet.

4.3.1.5 CProtocol

Als Oberklasse aller Protokoll-, Netzwerk-Adapter- oder Applikationsklassen dient CProtocol (Abbildung 4.39). Sie stellt die Grundprotokollfunktionalität (Abschnitt 4.2.1.7) zur Verfügung.
Ein wichitges Element der Klasse CProtocol ist die Membervariable ptActiveSessionMap, in der die Verwaltungsinformationen zu den momentan vorhandenen Protokollsessionobjekten gehalten werden. Mittels `registerProtocolSession` bzw. `getDemuxChannel` können entweder neue Protokollsessions eingetragen bzw. ein Sessionmap-Lookup durchgeführt werden.
Für die Upstream-Nachrichtenübergabe von Protokollsessions der darunterliegenden Schicht mittels Callback (Abbildung 4.16), setzt CProtocol die Methode demux ein. Einzige Parameter, welche dieser Methode übergeben werden, ist die auslösende Protokollsession und die hochzuschleusende Nachricht.
Wichtige virtuelle Methode der Klasse CProtocol ist openSession, welche für das Anfordern eines Protokollsessionobjektes Protokollen der darüberliegenden Schicht angeboten wird. Da die `openSession`-Methoden für Netzwerk-Adapter und Protokolle von der Schnittstelle her gleich sein müssen, aber von ihrer Logik unterschiedlich, wird in der Klasse CProtocol diese Methode virtuell gesetzt.

Abbildung 4.38: Übersicht der Klasse CPort

4.3.1.6 CProtocolSession

Die in Abschnitt 4.2.1.7 modellierten Protokollsessions werden in der VIPAPI durch CProtocolSession umgesetzt. Wichtigste Funktionalitäten bzw. Eigenschaften dieser Klasse sind (Abbildung 4.40):

- Jedes CProtocolSession-Objekt besitzt eine eindeutige Session-ID, welche in der Membervariable m_sessionid gehalten wird.
- CProtocolSession implementiert das Versenden von Nachrichten zu den Protokollsessionobjekten der darunterliegenden Schicht mittels der `send`-Methoden. Als Parameter wird dieser Methode die zu übertragende Nachricht übergeben. Mit einem optionalen delay-Parameter kann eine Verzögerung im Nachrichtenversand aktiviert werden, der eine Laufzeitverzögerung durch Bearbeitung simuliert.
- Das Klassenattribut m_sessionstate dient dazu, den Zustand des Protokollsessionobjektes zu

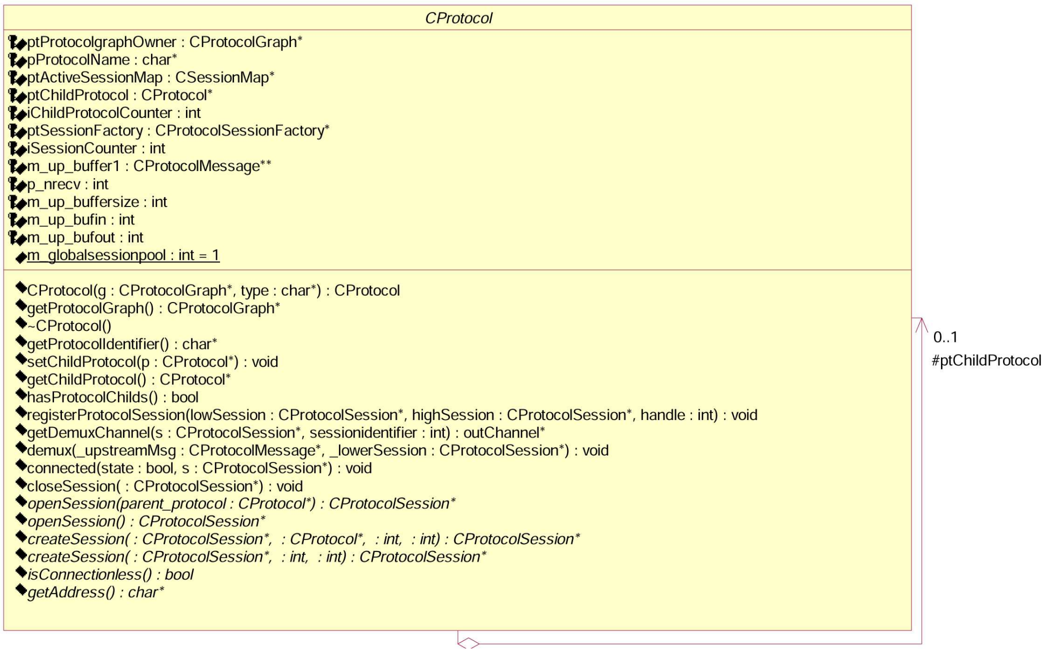

Abbildung 4.39: Übersicht der Klasse CProtocol

halten. Um Zugriff bzw. Änderungen an diesem Attribut vornehmen zu können, stellt CProtocolSession die Methoden `getSessionState` bzw. `setSessionState` zur Verfügung.

- CProtocolSession implementiert alle für die Kommunikation (upstream bzw. downstream) notwendigen in- und outChannels mit den jeweils dazugehörigen Listener- und Workerprozessen. Für den upstream-Nachrichtenempfang bzw. -weiterleitung sind `protocolListener` und `protocolWorker` zuständig. Für den downstream-Nachrichtenempfang bzw. -weiterleitung sind `sessionListener` und `sessionWorker` zuständig.

Die eigentliche Funktionalität eines Protokolls besteht in der Verarbeitung von Nachrichten. Da diese Logik von Protokoll zu Protokoll unterschiedlich ist, kann diese Funktionalität nicht direkt zur Verfügung gestellt werden. Um die Schnittstelle für alle von CProtokollSession abgeleiteten Klassen (CInterface, CNetworkProtocol) zu vereinheitlichen, stellt CProtocolSession die beiden virtuellen Methoden `processUpMessage` und `processDownMessage` zur Verfügung. Im Gegensatz zu CInterfaceSession überlässt CNetworkProtocolSession die Umsetzung der Verarbeitungslogik, bestehend aus `processUp`- und `processDownMessage`, den von ihr abgeleiteten Protokollklassen. Das hat zur Folge, dass beide Methoden virtuell bleiben.

4.3.1.7 CNetworkProtocol und CNetworkProtocolSession

Die Untergliederung der Oberklasse CProtocol in die Unterklassen CNetworkProtocol und CInterface wurde deshalb durchgeführt, weil zwar die Schnittstellen identisch, die Funktionalität jedoch unterschiedlich ist.
CNetworkProtocol erweitert CProtocol um die zwei zusätzlichen Attribute bUnreliable und bConnectionless, welche den von einem Protokoll umgesetzten Dienst (verbindungsorientiert/verbindungslos bzw. zuverlässig/unzuverlässig) beschreibt. Die geforderten `openSession`- und `createSes-`

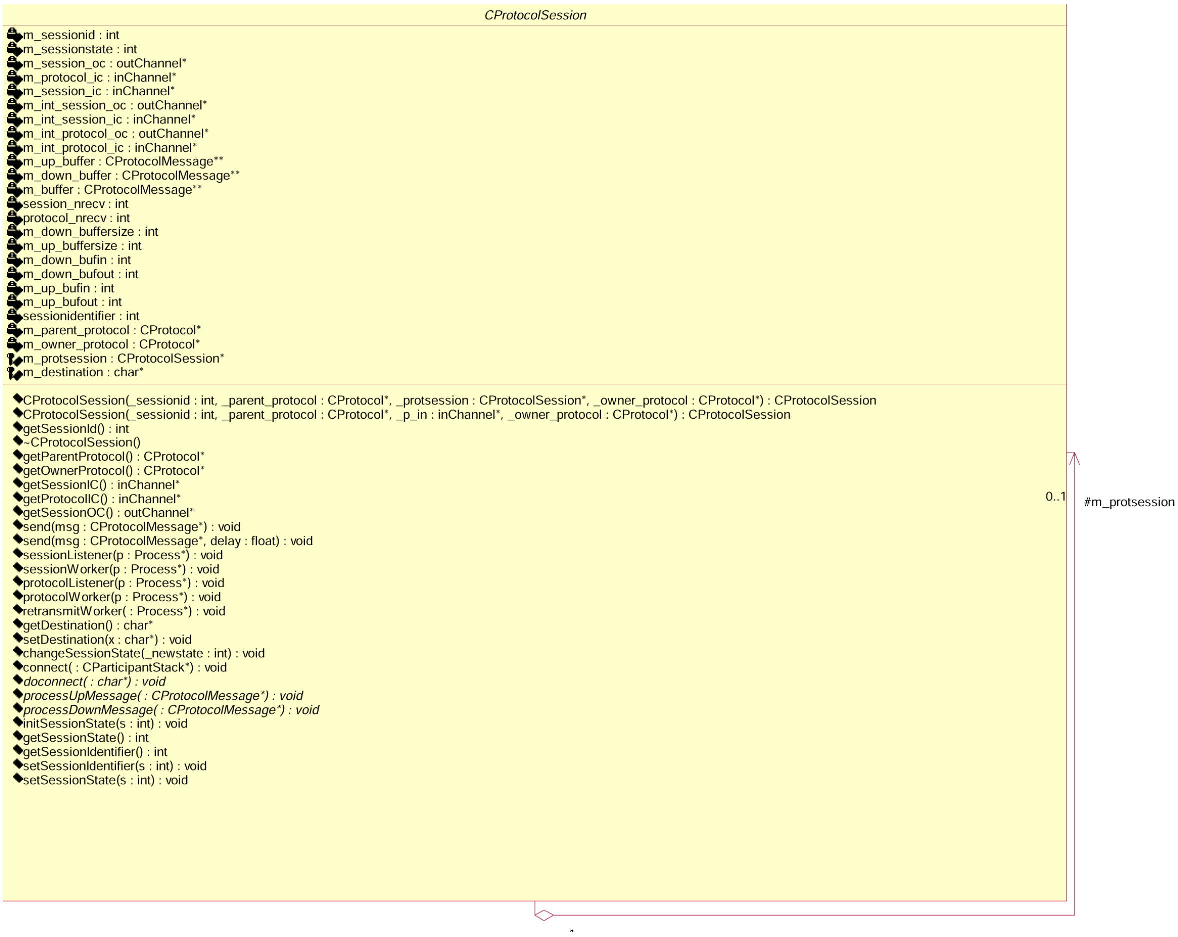

Abbildung 4.40: Übersicht der Klasse CProtocolSession

sion-Methoden der Oberklasse werden umgesetzt. Besonderheit hierbei ist die Verwendung einer Factory als Session-Generator. Weitere Informationen über die Factory stehen in der Klasssendokumentation zur Verfügung.

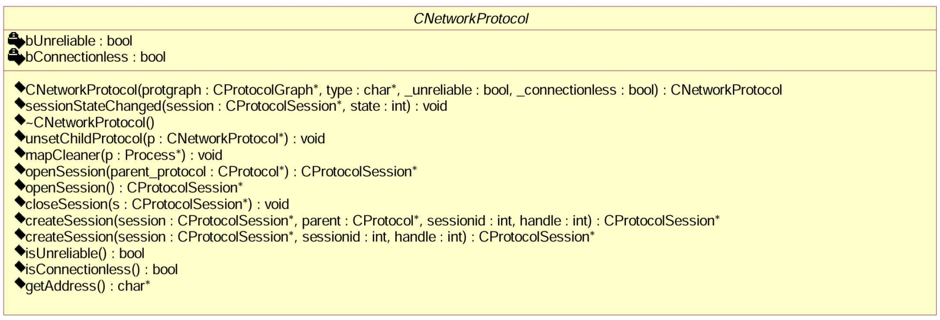

Abbildung 4.41: Übersicht der Klasse CNetworkProtocol

CProtocolSession muss in zwei Unterklassen aufgeteilt werden, CInterfaceSession und CNetwork-

ProtocolSession.

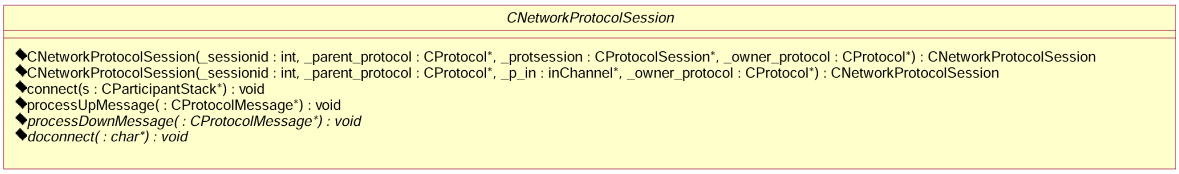

Abbildung 4.42: Übersicht der Klasse CNetworkProtocolSession

4.3.1.8 CInterface und CInterfaceSession

Die beiden Klassen CInterface und CInterfaceSession stellen den aus dem Design modellierten Netzwerk-Adapter (Interface und Interfacesession aus Abschnitt 4.2.2.7) dar.
CInterface erweitert die von CProtocol (Abschnitt 4.3.1.5) vererbten Eigenschaft um das Attribut ptInterfacePort vom Typ CPort* (Port des Adapters) und ptInterfaceSession vom Typ CInterfaceSession (Standardsessionobjekt). Um auf den Port zugreifen zu können, stellt CInterface die zusätzlichen Methoden getPortInChannel und getPortOutChannel bereit.
Da ein Netzwerk-Adapter, wie im Design modelliert, immer einer Standardsession (Member: ptInterfaceSession) zugeordnet ist, wird diese auch bei jeder Sessionanforderung des Interfaceprotokolles durch einen openSession-Aufruf zurückgegeben (Listing 4.1).

Listing 4.1: openSession-Methode der Klasse CInterface

```
CProtocolSession* openSession(CProtocol* parent_protocol)
{
  return ptInterfaceSession;
}
```

Da ein Netzwerk-Adapter keine Verarbeitungslogik besitzt, sondern nur Nachrichten weiterleitet, können die aus der Oberklasse CProtocolSession geforderten Methoden, processUpMessage und processDownMessage, umgesetzt werden (Listing 4.2).

Listing 4.2: Verarbeitungslogik der Klasse CInterfaceSession

```
void processUpMessage(CProtocolMessage* msg)
{
  getParentProtocol()->demux(msg, this);
}

void processDownMessage(CProtocolMessage* msg)
{
  send(msg);
}
```

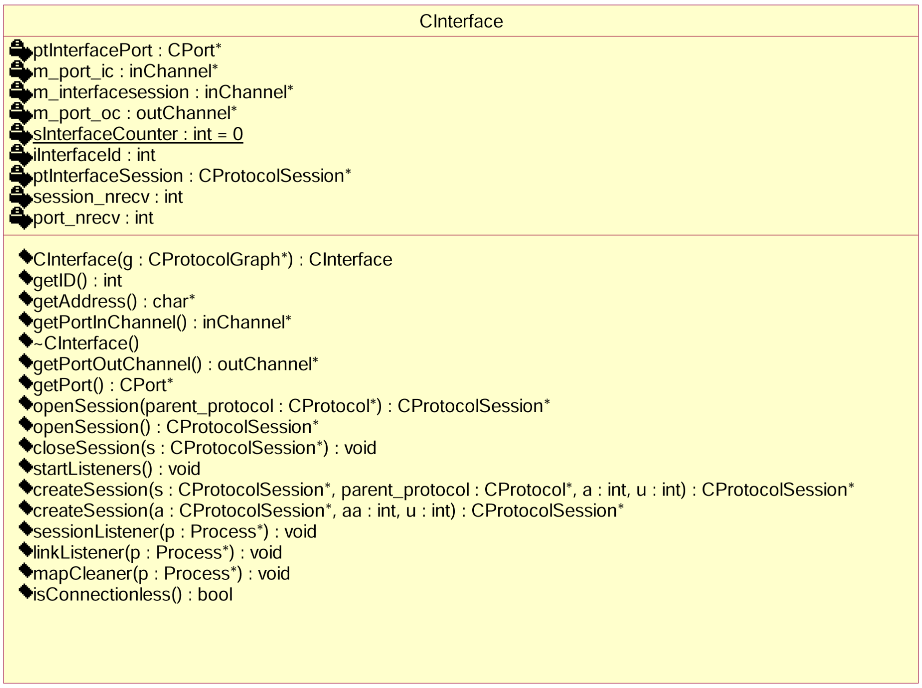

Abbildung 4.43: Übersicht der Klasse CInterface

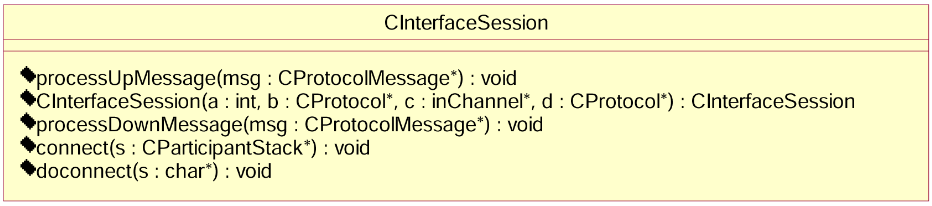

Abbildung 4.44: Übersicht der Klasse CInterfaceSession

4.3.1.9 CProtocolMessage

CProtocolMessage ist die Oberklasse aller Protokollnachrichten. Als einzige Erweiterung gegenüber dem Design besitzt die Klasse CProtocolMessage eine Hashtable (Member: header_fields) [Stro 97]. Darin können zusätzliche Headerinformationen, welche bei der Implementierung neuer Protokollnachrichten erforderlich sind, abgelegt werden. Hierbei wird jeweils ein Wertepaar, bestehend aus einem Integer-Wert (Header-Feldtyp) gekoppelt mit einem String-Wert (Header-Feldinhalt), abgespeichert. Diese Erweiterung ist deshalb nötig, damit bei der Implementierung neuer Protokollnachrichten die von der VIPAPI bereitgestellte clone-Methode benutzt werden kann, ohne diese verändern zu müssen. Für alle Membervariablen stehen die entsprechenden Zugriffsmethoden wie getSourceAddress und setSourceAddress zur Verfügung.

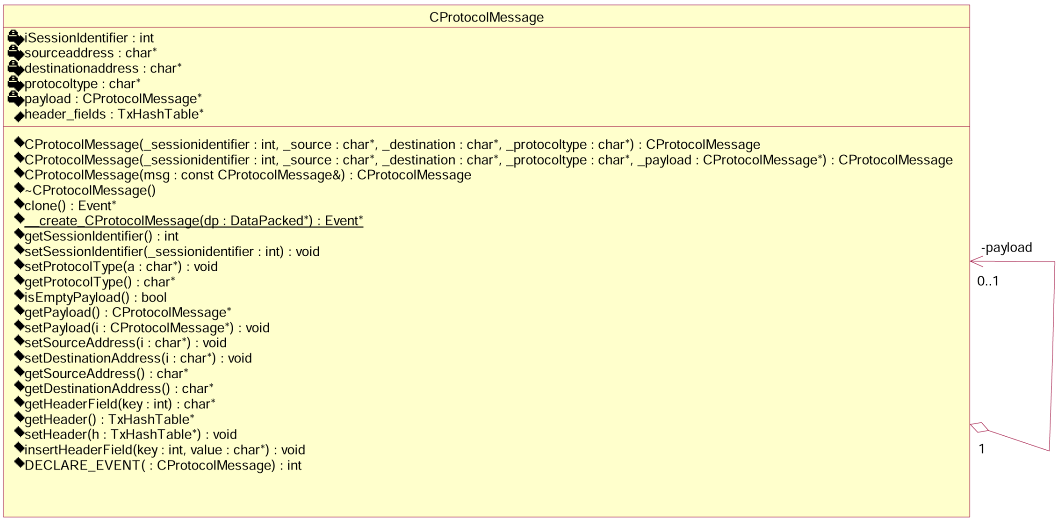

Abbildung 4.45: Übersicht der Klasse CProtocolMessage

4.3.1.10 CApplication

Anwendungen werden in der VIPAPI durch die Klasse CApplication umgesetzt. Einziger Unterschied zu der Oberklasse CNetworkProtocol ist die virtuelle Methode appExecuter. Diese ist als Schnittstelle zum Starten von Prozessen vorgesehen.

4.3.1.11 CLoggingEntity

Die Oberklasse aller logging- und -breakpointfähigen Klassen ist CLoggingEntity. Diese Klasse besitzt vier Membervariablen, welche den Status bezüglich Logging und Breakpoint beinhalten (Eventlogging/Eventbreakpoint, Statuslogging/Statusbreakpoint). Um auf diese Statusflags zugreifen zu können bietet CLoggingEntity Aktivierungs- bzw. Deaktivierungsmethoden an. Diese können von dem Simulationsinterface benutzt werden.
Zusätzlich stellt CLoggingEntity Entwicklern Methoden zur Verfügung, um in neuen Anwendungsobjekten, welche von CLoggingEntity abgeleitet sind, Loggingstellen bzw. Breakpoints definieren zu können. Listing 4.3 zeigt eine dieser Markierungsmethoden.

Listing 4.3: setEventLoggingpoint-Methode der Klasse CLoggingEntity

```
void CLoggingEntity::setEventLoggingpoint(CProtocolMessage* msg)
{
  if(isactiveEventLogging())
    {
      printf("[%g] MsgType: %s\n", now(), msg->getSourceAddress());
    }
}
```

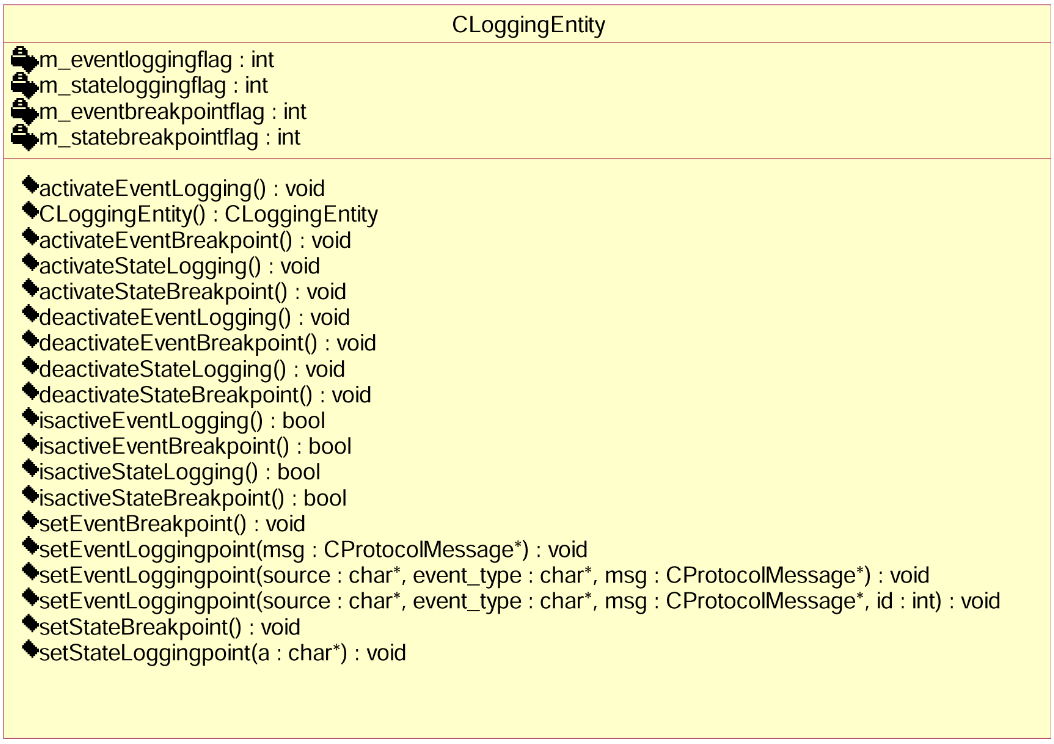

Abbildung 4.46: Übersicht der Klasse CLoggingEntity

4.3.1.12 CSimulation – Das Simulationsinterface

Das Simulationsinterface wird mittels der Klasse CSimulation umgesetzt. Diese Klasse beinhaltet die Steuerungs- bzw. Konfigurationsmechanismen für den direkten Zugriff auf Simulationen. Zusätzlich bietet aber auch eine XML-Parserfunktionanlität an, mit der indirekt das Simulationsinterface benutzt werden kann.
Für jeden zu verwaltenden Objekttyp, z.B. CComponent oder CConnector, wird eine eigene Hashtabelle angelegt, die dazu dient, die Simulationsinformationen zentral zu halten, um zu späteren Zeitpunkten (Laufzeit) wieder darauf zugreifen zu können.
Abbildung 4.47 zeigt einen Überblick der in dieser Klasse zur Verfügung gestellten Methoden. Die Funktionalitäten einzelner Methoden können aus der Klassendokumentation entnommen werden, da die Methodennamen selbsterklärend sind.
Nunmehr wir nachfolgend im Detail auf die XML-Struktur eingegangen, in der benutzerdefinierte Simulationen erstellt werden können. Als Parser für diese Konfigurationsdateien dient die Methode `parseXMLConfig` der Klasse CSimulation. Diese Methode implementiert einen auf Xerces-C++-basierenden Parser. Xerces-C++ ist ein validierender XML-Parser, der Teil des Apache XML Projects ist. Die XML-Dateien, welche der Parser der Klasse CSimulation verarbeitet, müssen der in Anhang C entsprechen.

4.3.2 Schrittweise Entwicklung der Beispielsimulation VIP_TIME

Die Erstellung der VIP_TIME Beispielsimulation erfordert folgende Anwendungsobjekte:

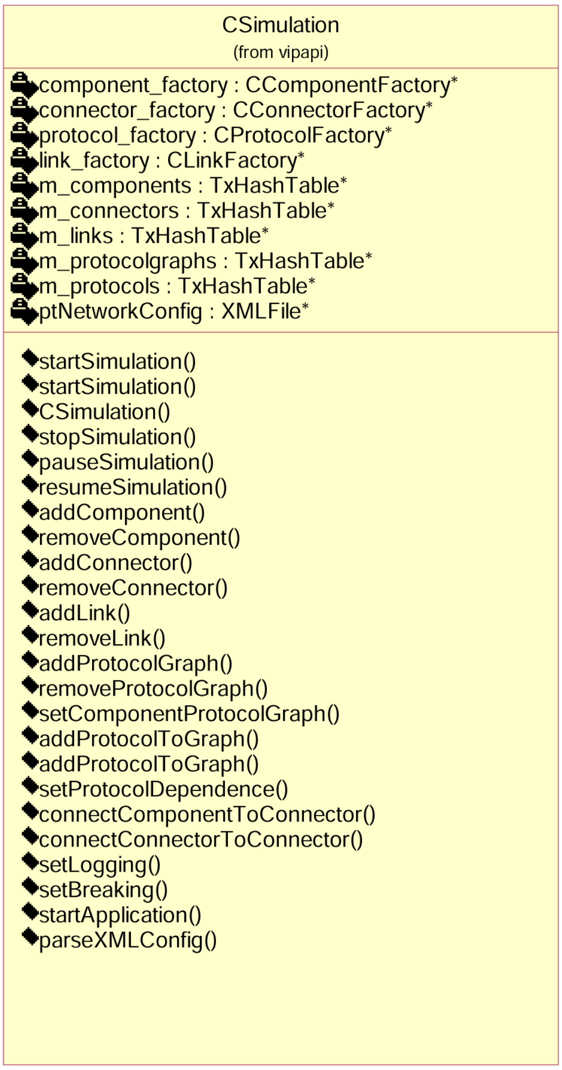

Abbildung 4.47: Übersicht der Klasse CSimulation

- Komponente: **Workstation**
- Konnektor: **8-Port-Hub**
- Verbindungsmedium: **Kabel**
- Protokolle: **ETHmin, IPmin und UDPmin**
- Anwendung: **VIP_TIME**

Im folgenden werden die Schritte erläutert, die notwendig sind, um diese Objekte zu erstellen und daraus eine Simulation zu generieren.

4.3.2.1 Komponente: Workstation

Um eine Komponente zu erstellen, reicht es aus, die gewünschte Klasse von CComponent abzuleiten. Die Headerdatei der Klasse CWorkstation, dargestellt in Listing 4.4, zeigt diesen Zusammenhang.

Listing 4.4: Klasse CWorkstation

```
class CWorkstation : public CComponent
  {
  public:
    CWorkstation();
    CWorkstation(CProtocolGraph*);
  };
```

4.3.2.2 Konnektor: 8-Port-Hub

Als nächstes Anwendungsobjekt muss ein 8-Port-Hub erstellt werden. Hierzu wird eine Klasse CHub_8 erstellt, welche von CConnector abzuleiten ist. Zusätzlich zu dieser Ableitung muss der Entwickler die virtuelle Methode processMessage implementieren. Dazu muss nun in dieser Methode, um die Funktionalität eines Hubs nachzubilden, die Logik umgesetzt werden, dass die an einem Port ankommende Nachricht an alle Ports wieder ausgegeben wird. Listing 4.5 und 4.6 zeigen zwei unterschiedliche Umsetzungen unter Verwendung der unterschiedlichen `send`-Methoden.

Listing 4.5: Variante 1: processMessage-Methode der Klasse CHub_8

```
void CHub_8::processMessage(int portnumber, CProtocolMessage* msg)
{
  sendToAllPorts(msg);
}
```

Listing 4.6: Variante 2: processMessage-Methode der Klasse CHub_8

```
void CHub_8::processMessage(int portnumber, CProtocolMessage* msg)
{
 for (int i=0; i<getPortCount() ; i++)
 {
  sendToPort(i+1, msg);
 }
}
```

4.3.2.3 Verbindungsmedium: Kabel

Um ein Kabelobjekt CCable zu erhalten, muss diese zunächst von CLink abgeleitet werden (Abschnitt 4.3.1.3). Danach muss die Methode `processMessage` implementiert werden. In dieser Methode kann man nun die üblichen Störeinflüsse eines Verbindungsmediums abbilden. In der Beispielsimulation wird dagegen von einer ungestörten Nachrichtenübertragung im Kabel ausgegangen. Listing 4.7 zeigt die `processMessage`-Methode für diesen Fall.

Listing 4.7: processMessage-Methode der Klasse CCable

```
CProtocolMessage* CCable::processMessage(CProtocolMessage* msg)
{
```

```
  /*
   * Mögliche Nachrichtenbearbeitung zur Simulation von
   * Übertragungsstörungen im Kabel, z.B. CRC-Fehler
   */

  return msg;
}
```

4.3.2.4 Protokolle: ETHmin, IPmin und UDPmin

Um Protokolle zu implementieren, müssen je Protokoll drei Klassen angelegt werden:

- Protokollklasse, abgeleitet von CNetworkProtocol
- Protokollsessionklasse, abgeleitet von CNetworkProtocolSession
- Protokollnachrichtenklasse, abgeleitet von CProtocolMessage

Da die zu erstellenden Protokolle in der Beispielsimulation VIP_TIME alle die gleiche Funktionalität besitzen, reicht es aus, diesen Prozess anhand eines der drei Protokolle, ETHmin, zu erläutern. Der Klassenname für das in der Anforderungsspezifikation beschriebene Protokoll ETHmin lautet CEthMin. Dementsprechend heißt die dazugehörige Protokollsession- bzw. Protokollnachrichtenklasse CEthMinSession bzw. CEthMinMessage.

Für die volle Implementierung von CEthMin muss neben der Ableitung von CNetworkProtocol deren virtuelle Methode `getAddress` umgesetzt werden. Diese Methode gibt die Konfigurationsadresse eines Protokolls, z.B. aktuelle MAC-Adresse bei Ethernet, zurück. Listing 4.8 zeigt die Umsetzung bei CEthMin. Hier wird über den Konstruktor eine einfache Adresse in einem char* abgelegt, welche dann über die definierte Methode `getAddress` zurückgegeben wird. Dies reicht aus, um eine minimale `getAddress`-Methode zu implementieren. Komplexere Protokolle wie TCP erfordern hier natürlich einen komplexeren Algorithmus.

Listing 4.8: Implementierung der Klasse CEthMin

```
CEthMin(CProtocolGraph* protgraph, char* type, char* _address)
               : CNetworkProtocol(protgraph, type, true, true)
{
  address=_address;
}

char* getAddress()
{
  return address;
}
```

Die Erstellung der passenden Protokollsessionklasse ist etwas aufwändiger, da hier die wirkliche Logik eines Protokolls abgebildet werden muss. Die Logik von CEthMinSession besteht darin, vor downstream-Nachrichten einen minimalen Header, bestehend aus Zieladdresse, Quelladresse, Protokolltyp und den Sessionidentifier zu setzen. Dies wird in Listing 4.9 dargestellt. Hierbei wird eine neue CEthMinMessage erstellt, welche als Parameter den Sessionidentifier (`getSessionIdentifier()`), die eigene Adresse (`getOwnerProtocol-`

>getAddress()) und die Ziel-Adresse (getdestination())enthält. Dieser leeren Nachricht wird danach als Payload die erhaltene Nachricht der Protokollsession (CIpMinSession) der darüberliegenden Schicht mittels setPayload angefügt. Danach wird die Nachricht mit der Methode send an die Protokollsession der darunterliegenden Schicht (Interfacesession) gesandt.

Listing 4.9: processDownMessage-Methode der Klasse CEthMinSession

```
void CEthMinSession::processDownMessage(CProtocolMessage* msg)
{
  CEthMinMessage* d_msg = new CEthMinMessage(getSessionIdentifier(),
                          getOwnerProtocol()->getAddress(), getDestination());
  d_msg->setPayload(msg);
  send(d_msg);
}
```

Upstream-Nachrichten, welche bei CEthMinSession eintreffen, werden mittels processUpMessage bearbeitet. Hierbei muss zunächst überprüft werden, ob die erhaltene Nachricht tatsächlich für diese Session bestimmt ist. Dazu wird die Zieladresse extrahiert und mit der eigenen Adresse verglichen. Sollten die Adressen nicht übereinstimmen, so wird die Nachricht verworfen. Im Fall der Adressübereinstimmung muss die Payload (CIpMinMessage) extrahiert werden. Anschließend wird über Aufruf der Callback-Methode demux des darüberliegenden Protokolls (CIpMin) die Nachricht an diese Schicht weitergegeben (Listing 4.10).

Listing 4.10: processUpMessage-Methode der Klasse CEthMinSession

```
void CEthMinSession::processUpMessage(CProtocolMessage* msg)
{
  CProtocolMessage* s;

  if(strcmp(msg->getDestinationAddress(), getOwnerProtocol()->getAddress())==0)
    {
      s = msg->getPayload();
      getParentProtocol()->demux(s,this);
    }
  else
    {
       // Discard message
    }
}
```

Die Nachricht CEthMinMessage besitzt keine Erweiterungen zu CProtocolMessage. Einzige Besonderheit ist der Standard-Protokolltyp *ETHmin* im Konstruktor (Listing 4.11). Dies ist der eindeutige Identifier (char*) für einen Protokolltyp. Dieser Typ entspricht dem Parameter type im Konstruktor der Klasse CEthMin. Ein komplexeres Beispiel mit Erweiterungen wird in dem VIP_TIME-Nachrichtenformat erklärt.

Listing 4.11: Konstruktor der Klasse CEthMinMessage

```
CEthMinMessage(int sid, char* _source, char* _destination)
        : CProtocolMessage (sid, _source, _destination, "ETHmin")
{

}
```

Diese am Beispiel von ETHmin geschilderte Vorgehensweise ist identisch mit der von IPmin und UDPmin. Natürlich kann sie auch als Leitfaden bei der Implementierung weiterer Protokolle herangezogen werden.

4.3.2.5 Anwendung: VIP_TIME

Da bis auf die Methode `appExecuter` Anwendungen und Protokolle in der VIPAPI gleiche Funktionalität haben, läuft die Entwicklung einer Anwendung genauso ab wie die eines Protokolls. Deshalb wird in diesem Abschnitt nur auf Unterschiede und Besonderheiten des Entwicklungsprozesses eingegangen.
Der Protokollteil der VIP_TIME Anwendung wird in der Klasse CVipTime implementiert. Dementspechend werden die dazugehörigen Session- bzw. Nachrichtenklassen als CVipTimeSession bzw. CVipTimeMessage bezeichnet.

Unterschiedlich zum normalen Entwicklungsprozess eines Protokolls ist die vorgeschriebene Anwendungsmethode `appExecuter` (Listing 4.12), welche umgesetzt werden muss. Diese Methode dient dazu, Userinteraktionen nach Start einer Simulation zu simulieren. In Bezug auf VIP_TIME wird ein SSF Process gestartet, welcher ein Sessionobjekt bei CVipTime anfordert und auf dem enthaltenen CVipTimeSession-Objekt die `connect`-Methode aufruft (Listing 4.13). Da solche Mechanismen wie **Address Resolution Protocol** (**ARP**) nicht implementiert sind, werden die Ziel-Adressen in einem separaten Adressstack mitgegeben. Jede Session nimmt sich nun beim Durchlauf des `connect`-Aufrufes ihre jeweiligen Zieladressen vom Stapel. Diese Zieladressen werden dann anschließend in den Sessionobjekten zwischengespeichert.

Listing 4.12: appExecuter-Methode der Klasse CVipTime

```
void appExecuter()
{
  new AppExecuter((CNetworkProtocol*)this);
}
```

Listing 4.13: action-Methode der Prozess-Klasse AppExecuter zum Starten eines Prozesses

```
//! SSF PROCEDURE
void AppExecuter::action()
{
  CProtocolSession* x = o->openSession();

  s->push("41:ae:f3:50:03:51");
  s->push("192.168.0.253");
  s->push("124");
  s->push("");

  x->connect(s);
  waitForever();
}
```

Da VIP_TIME im Gegensatz zu den zuvor behandelten Protokollen verbindungsorientiert ist, kann CVipTimeSession unterschiedliche Zustände annehmen. Dementsprechend muss im Konstruktor mittels der Methode `initSessionState` ein Initialzustand gesetzt werden (Listing: 4.14), welcher

im weiteren Verlauf der Simulation dann die in Abschnitt 4.1.2.3 beschriebenen Zustände annehmen kann.
Wie im vorherigen Abschnitt beschrieben, wird die Logik eines Protokolls durch die `processUpMessage`- und `processDownMessage`-Methode implementiert. Eine Implementierung der `processDownMessage`-Methode ist natürlich nicht notwendig, da die VIP_TIME Anwendung keine Nachrichten von einer höheren Protokollschicht erhält. Jegliche Funktionalität wandert in die `processDownMessage`-Methode. Der Grundalgorithmus dieser Methode ist folgendermaßen definiert:

1. Extrahiere den Nachrichtentyp aus dem Header-Feld `MSG_TYPE_FIELD`.
2. Ermittle den aktuellen Zustand der Session mittels der `getSessionState`-Methode.
3. Überprüfe, ob der erhaltene Nachrichtentyp mit dem aktuellen Status der Session verträglich ist. Beipielsweise wird in der VIP_TIME Anwendung bei einem Session-State `WAITING_CLOSE` eine Nachricht vom Typ `DISCONN_ACK` erwartet.
4. Sollten die beiden Werte nicht miteinander vereinbar sein, so wird die Nachricht verworfen. Andernfalls wird mittels der `send`-Methode eine passende Antwortnachricht zurückgesandt. Bezogen auf diesen Nachrichtenversand muss ggf. der Zustand verändert werden.

Um zu wissen welche initiale Aktion bei einem Verbindungsaufbauwunsch auszuführen ist, muss mittels der `doconnect`-Methode angegeben werden, welcher Nachrichtentyp als Initialnachricht versandt werden soll (Listing 4.15).

Listing 4.14: Klasse CVipTimeSession

```
CVipTimeSession(int a, CProtocol* b, CProtocolSession* c, CProtocol*d)
                                   : CNetworkProtocolSession(a,b,c,d)
{
  initSessionState(IDLE);
}
```

Listing 4.15: Klasse CVipTimeSession

```
void doconnect(char* s)
{
  CVipTimeMessage* msg =  new CVipTimeMessage( getSessionIdentifier(), getDestination(
  setSessionState(WAITING_OPENACK);
  send(msg);
}
```

Besonderheit bei der Implementierung der Klasse CVipTimeMessage ist die zusätzliche Information des Nachrichtentyps. Dieses Feld muss in CVipTimeMessage mittels der Hashtable realisiert werden, damit während eines Simulationslaufes die `clone`-Methode von CProtocolMessage genutzt werden kann. Das bedeutet, dass dieses Feld im Konstruktor mittels der `insertHeaderField`-Methode hinzugefügt werden muss (Listing 4.16). In späteren Verarbeitungsphasen kann mittels der Methode `getHeaderField` abgefragt werden.

Listing 4.16: Konstruktor der Klasse CVipTimeMessage

```
CVipTimeMessage(int sid, char* _source, char* _destination, char* _type)
        : CProtocolMessage (sid, _source, _destination, "VIP_TIME")
{
  insertHeaderField(MSG_TYPE_FIELD, _type);
}
```

4.3.2.6 Simulationserstellung

Zr Erstellung der Beipielsimulation muss eine passende XML-Datei erstellt werden. Damit der Parser nach Erstellung neuer Anwendungsobjekte nicht verändert werden muss, werden in der VIPAPI sog. Factory-Klassen benutzt. Nach der Implementierung neuer Anwendungsobjekte müssen diese in der entsprechenden Factory-Klasse registriert werden, z.B. CCable in der Klasse CLinkFactory welche für die Generierung von CLink-Objekte zuständig ist. Zusätzlich muß dementsprechend die DTD, um den in der Factory-Klasse erweiterten Typen, ergänzt werden. Nach diesen Änderungen können die neuen Anwendungsobjekte über die XML-Schnittstelle benutzt werden.
Die in Anhang D abgebildete XML-Datei ist die Konfigurationsumsetzung der Beispielsimulation VIP_TIME.
Die XML-Parsing-Funktionalität des Simulationsinterfaces wurde umgesetzt, jedoch traten in Zusammenhang mit dem Parsen zur Laufzeit Probleme auf, die wegen der Kürze der verfügbaren Zeit nicht mehr behoben werden konnten. (Anhand der Beispielsimulation `test/beispielsimulation_simulationsinterface_xml.cpp` ist dieser Fehler dokumentiert.) Wenn das Simulationsinterface nicht über den XML-Parser versorgt wird, sondern direkt benutzt wird, tritt dieser Fehler nicht auf. Es scheint sich also um ein Problem in der Umsetzung des Parsers zu handeln, das in der verfügbaren Zeit nicht mehr behoben werden konnte. Die Beispielsimulation `test/beispielsimulation_simulationsinterface.cpp` zeigt die fehlerfreie Funktionalität des Simulationsinterfaces für diesen Fall. Ein detailliertes Ablaufbeispiel (mit eingeschalteter Debug-Funktionalität) ist auf der beiliegenden CD enthalten.

4.4 Test

Das Testen einer zu erstellenden Applikation ist ein sehr wichtiger und zeitaufwändiger Teil im Entwicklungsprozess. Das richtige Eingrenzen in eine Phase ist aber sehr schwierig, da natürlich permanent während der Implementierung Tests parallel laufen. Aufgrund dessen wird im folgenden Abschnitt nur Bezug auf die abschließenden Performance-Tests der Beispielsimulation genommen.

4.4.1 Test- und Simulationsumfeld

Die abschließenden Performance-Tests, bezogen auf Richtigkeit und Performance der erstellten Simulation, wurden auf der in in Tabellen 4.8 beschriebenen Testumgebung durchgeführt.

Rechnersystem	AMD Athlon 900 640 MByte Arbeitsspeicher
Betriebssystem	SuSE Linux 7.1 Professional
DaSSF Simulationskern	Release 3.1.5 + Erweiterungen für dynamisches Generieren von Simulationsobjekten

Tabelle 4.8: Übersicht der Testumgebung

4.4.2 Performance-Report

Für die Performancetest wurde die Beispielsimulation verwendet. Durch sukzessives Erhöhen der an der Simulation beteiligten Komponenten konnte das Verhalten der VIPAPI in Bezug auf Speicherauslastung in Grenzbereichen ermittelt werden. Die Tabelle 4.9 zeigt das Ergebnis dieser Tests. Wie ersichtlich ist, ist eine Simulation großer Netze möglich.

Komponentenanzahl	**Speicherbelastung (Mbyte)**
5	1,2
10	1,2
100	1,3
1000	1,5
10000	6,9
100000	62,3
200000	112,1

Tabelle 4.9: Übersicht der durchschnittlichen Speicherauslastung bezogen auf die beteiligten Komponenten

Kapitel 5

Installation der Simulationsumgebung

5.1 Systemvoraussetzungen

5.1.1 Rechnersysteme

Die Lauffähigkeit der VIP-Simulationsumgebung wird durch den benutzten DaSSF-Simulationskern begrenzt. Dieser ist auf fast allen UNIX-basierenden Rechnersystemen lauffähig, z.B. SUN/Solaris, i386/Linux.

Ein genauer Überblick der unterstützten Rechnersysteme wird in [Cowi 99] gegeben.

5.1.2 Softwarevoraussetzungen

Um auf einem Rechnersystem die VIP-Simulationsumgebung einrichten zu können, sollten folgende Tools vorinstalliert sein (* optional):

GNU-make	http://www.gnu.org/directory/GNU/make.html
GNU C++ Compiler	http://www.gnu.org/directory/GNU/gcc.html
flex	ftp://ftp.gnu.org/non-gnu/flex/
GNU-bison	http://www.gnu.org/directory/GNU/bison.html
Perl	http://www.perl.com/CPAN/src/
GNU-binutils	http://www.gnu.org/directory/GNU/binutils.html
astyle*	http://www.arrizza.com/downloads/astyle/_files.html
doxygen*	ftp://ftp.stack.nl/pub/users/dimitri/doxygen-1.2.18.src.tar.gz
extDaSSF	siehe Tabelle 5.1
xcerces*	http://xml.apache.org/dist/xerces-c/stable/

5.2 Verzeichnisstruktur

Die VIP-Library ist in einer strukturierten Verzeichnishierarchie abgelegt. In Tabelle 5.1 wird ein Überblick dieser Struktur gegeben.

5.3 Installationsreihenfolge

1. Installation der in Abschnitt 5.1.2 angegebenen Software.
2. Kopieren des kompletten *vip_d*-Verzeichnisses (Tabelle 5.1) an eine beliebige Stelle.
3. Anpassen der VIP-Makefile (Tabelle 5.1):
   - Der Wert **D2DIR** der Makefile muss auf das installierte extDaSSF-Wurzelverzeichnis gestellt werden (Anhang B).
   - Der Wert **USERLDFLAGS** muss mittels der *-L*-Option um den Xcerces-Library-Pfad erweitert werden.
4. Nach dem Aufruf von: *make clean; make depend; make all* ist die Simulationsumgebung fertig eingerichtet und kann benutzt werden.

5.4 Nützliche Tools

make_doc.sh Dieses Shell-Skript erstellt eine sehr anschauliche Klassendokumentation, welche in das *vip/doc*-Verzeichnis abgelegt wird. Über die Konfigurationsdateidoxygen.conf hat man die Möglichkeit Dokumentationseigenschaften zu steuern. Die Klassendokumentation wird in folgenden Formaten erstellt:

- **Hypertext Markup Language** (**HTML**)
- **Portable Document Format** (**PDF**)
- **DeVice Independent** (**DVI**)
- **Extensible Markup Language** (**XML**)
- Postscript
- Man-Page

style.sh Dieses Shell-Skript erleichtert dem Programmierer das Formatieren neu erstellter C++-Dateien, indem es alle Code-Dateien des VIP-Projektes auf ein einheitliches Codebild konvertiert.

Verzeichnis			Inhalt
vip_d			Hauptverzeichnis enthält die VIP-Makefile
	— ***bin***		Simulationsbinaries der Beispielsimulation
	— ***doc***		Dokumentation der VIP-Library (HTML-Version, Latex-Version, Postscript-Version) erstellt durch das Shell-Skript *../src/make_doc.sh*, basierend auf dem Tool doxygen.
	— ***include***		Header-Dateien
		— ***applications***	Header-Dateien der fertigen Anwendungsobjekte: Applikation
		— ***components***	Header-Dateien der fertigen Anwendungsobjekte: Komponenten
		— ***connectors***	Header-Dateien der fertigen Anwendungsobjekte: Konnektoren
		— ***links***	Header-Dateien der fertigen Anwendungsobjekte: Verbindungsmedium
		— ***protocols***	Header-Dateien der fertigen Anwendungsobjekte: Protokolle und Sessions
		— ***vipapi***	Header-Dateien der VIP-Library und des Simulations-Interfaces
		— ***vipfactory***	Header-Dateien der Klassen-Factories für Simulations-Interface
	— ***scripts***		Nützliche Skripts zur Erstellung der Dokumentation und zur Aufbereitung des Layoutes der Sourcen
	— ***src***		Implementierungs-Dateien
		— ***applications***	Implementierungs-Dateien der fertigen Anwendungsobjekte: Applikation
		— ***components***	Implementierungs-Dateien der fertigen Anwendungsobjekte: Komponenten
		— ***links***	Implementierungs-Dateien der fertigen Anwendungsobjekte: Verbindungsmedium
		— ***connectors***	Implementierungs-Dateien der fertigen Anwendungsobjekte: Konnektoren
		— ***protocols***	Implementierungs-Dateien der fertigen Anwendungsobjekte: Protokolle und Sessions
		— ***vipapi***	Implementierungs-Dateien der VIP-Library-Interfaces
		— ***vipfactory***	Implementierungs-Dateien der Klassen-Factories für Simulations-Interface
	— ***test***		Quellcode der Beispielsimulation
	— ***log***		Ablauf-Log der Beispielsimulation
	— ***extDaSSF***		Quellcode des extended DaSSF-Simulationskerns

Tabelle 5.1: Übersicht der Verzeichnisstruktur des VIP

Kapitel 6

Zusammenfassung und Ausblicke

Im Rahmen der Diplomarbeit wurde eine modulare Objektstruktur (VIP-Library) für eine Rechnernetz-Simulation erstellt. Dabei wurde versucht, die Basismechanismen des OSI-Modells nachzubilden und in der VIP-Library zu implementieren. Anhand der erstellten Beispielsimulation wurde die Grundfunktionalität der VIP-Library demonstriert. Außerdem wurde die Vorgehensweise bei der Entwicklung neuer Anwendungsobjekte schrittweise erläutert.
Zusätzlich zur VIP-Library wurde ein Simulationsinterface erstellt. Die geforderten Logging- und Breakpointmechanismen konnten, bezogen auf Anwender- bzw. Entwicklersicht umgesetzt werden. Die Steuerungs- bzw. Konfigurationsfunktionalität konnte ebenfalls umgesetzt werden. Um die Versorgung der Konfigurationsschnittstelle zu vereinfachen, wurde diese um einen auf XML-basierenden Konfigurationsteil erweitert. Simulationen können vom Anwender nun in Form von XML-Dateien, welche einer vordefinierten DTD entsprechen müssen, in einfacher Weise definiert werden. Da die Objektstruktur von dem Funktionsumfang des verwendeten DaSSF-Kerns abhängig ist, dessen Funktionalität aber derzeit bestimmte simulationssteuernde Aktionen nicht zulässt, wie beispielsweise das Anhalten bzw. Fortsetzen von Simulationen, konnten diese nur im Design der Steuerungsschnittstelle berücksichtigt, aber nicht implementiert werden.
Besonderer Wert wurde beim Design auf die leichte Erweiterbarkeit der VIP-Library gelegt. Die Bewertung des vollen Leistungspotentials der erstellten Objektstruktur konnte natürlich im Rahmen der Diplomarbeit aus nachvollziehbaren Gründen nicht durchgeführt werden. Um eine Bewertung dieser Art durchführen zu können, müssen nicht nur weitere Anwendungsobjekte realisiert, sondern auch die implementierten minimalen Pseudoprotokolle durch Implementierungen realer Protokolle erweitert werden. Diese Erweiterungen könnten Themen weiterer Diplomarbeiten im Rahmen des “Virtuellen Informatik Praktikums“ sein.

Wie in der Diplomarbeit gezeigt werden konnte, ist die Realisierung eines virtuellen Informatikpraktikums durchaus möglich. Bis zum eigentlichen Ziel, der Simulation komplexer Netzwerke mit vielen Komponenten, ist es jedoch sicherlich noch ein weiter Weg. Die volle Nachbildung der Funktionalität von aktiven Netzelementen wie Hubs, Switches und Routern bis hin zu solchen Details wie MIB-Daten und SNMP-Funktionalitäten oder der Ressource-Reservation-Protokolle für VOIP-Übertragungen ist schwierig, aber im Rahmen weiterer Diplomarbeiten sicher bis zu einem gewissen Grad machbar. Grundlegende rechnernetztypische Zusammenhänge und Ablaufmechanismen, wie das wichtige Prinzip der Protokollkommunikation, die für ein theoretisches Grundverständnis moderner Kommunikationssysteme wichtig sind, konnten in dieser Diplomarbeit umgesetzt werden.

Es ist für StudentenInnen sicherlich didaktisch deutlich einfacher, Aufgaben in einem virtuellen Praktikum zu lösen, bei dem die Benutzerinterfaces alle nach einem einheitlichen Styleguide realisiert sind, als sich mit unterschiedlichen graphischen und command-line-artigen Interfaces von Netzwerkkomponenten und LAN- und WAN-Analyzern verschiedenster Hersteller auseinander zu setzen, bevor mit dem eigentlichen Praktikumsbetrieb begonnen werden kann.

Insgesamt ist festzustellen, dass im Bereich der Universitäten die weiteren Potenziale für eine Virtualisierung hoch sind. Es ist durchaus vorstellbar, dass die dem Virtuellen Informatik Praktikum zugrunde liegende Idee zu einer virtuellen Informatikvorlesung (VIV) und letztendlich einem virtuellen Informatikstudium (VIS) weitergeführt werden könnte. Hierbei könnten die Vorlesungen mit multimedialen Mitteln über die in Deutschland in Bälde flächendeckend verfügbare DSL-Technologie zu den StudentenInnen gebracht werden. Diese müssten über lange Zeiträume ihres Studiums nun nicht mehr am Ort der Universität ansässig sein, sondern könnten an ihrem ursprünglichen Wohnort bleiben. Nachdem die Lebenshaltungskosten am Studienort (i.d.R. in Ballungsgebieten), insbesondere die Finanzierung einer Wohnung, viele Studenten zwingen, Nebentätigkeitsjobs zu suchen, wirkt sich das heute verständlicherweise meist negativ auf die Studiendauer aus bzw. führt in diversen Fällen zum Nichtabschluss des Studiums.

Virtuelle Studienangebote könnten also durchaus auch ein Instrument darstellen, sozial schwächeren Schichten eine Studienmöglichkeit zu eröffnen, da wegen des nicht notwendigen Wohnortswechsels deutlich geringere Kosten anfallen würden. Die so weitergeführte Idee des VIP könnte also über den heutigen technischen und didaktischen Ansatz hinaus am Ende einen wesentlichen Beitrag zur Beseitigung sozialer Benachteiligungen leisten.

Anhang A

SSF API C++-Bindings

```
#define Object void

class Entity {
public:
  virtual void init()=0;
  ltime_t now();

  void startAll(ltime_t t);
  void startAll(ltime_t t_start, ltime_t t_end);
  ltime_t pauseAll();
  void resumeAll();
  void joinAll();

  Object* alignment();
  ltime_t alignto(Entity* entity);
  Entity** coalignedEnities();
  Process** processes();
  inChannel** inChannels();
  outChannel** outChannels();
};

class inChannel {
public:
  Entity* owner();
  Event** activeEvents();
  outChannel** mappedto();
};

class outChannel {
public:
  Entity* owner();
  inChannel** mappedto();
  void write(Event* e, ltime_t delay=0);
  ltime_t mapto(inChannel* tgt, ltime_t mapping_delay=0);
  ltime_t unmap(inChannel* tgt);
};
```

```
class Process {
public:
  process(Entity* owner, int simple=0);
  process(Entity* owner, void(Entity::*body)(process*), int simple=0);

  virtual void action();
  virtual void init();
  virtual boolean isSimple();
  Entity* owner();

  void waitOn(inChannel** waitchannels);
  void waitOn(inChannel* waitchannel);
  void waitForever();
  void waitFor(ltime_t waitinterval);
};

class Event Event {
public:
  Event* save();
  void release();
  boolean aliased();
};
```

Anhang B

VIP-Makefile

```
#
# Makefile
#
# Get a template of the Makefile from $(D2DIR)/config/Makefile.template.
#
# Change this to reach D2 root directory.
D2DIR       =   /usr/local/DaSSF_dynamic/dassf

# The following lines introduce definitions needed, don't touch.
include $(D2DIR)/config/Makefile.common
all:    target

# Comment out this line if you want detailed output from compilation.
.SILENT:

# Put the name of the ultimate executable in EXECTGT
EXECTGT         = bin/vip_time_simulation

# Put the name of the ultimate library in LIBTGT.
#LIBTGT         = -I/usr/local/xerces-c1_6_0-linux/lib
# List all C++ header files here.
HDRFILES        = include/applications/viptime.h \
                include/applications/viptimemessage.h \
                include/applications/viptimesession.h \
                include/vipapi/vip_protocol.h \
                include/vipapi/vip_simulation.h \
                include/vipapi/vip_application.h \
                include/vipapi/vip_participants.h \
                include/vipfactory/vip_protocolsessionfactory.h \
                include/vipfactory/vip_connectorfactory.h \
                include/connectors/vip_hub_8.h \
                include/components/workstation.h \
                include/vipfactory/vip_componentfactory.h \
                include/vipapi/vip_protocolgraph.h \
                include/vipapi/vip_connector.h \
                include/vipapi/vip_interfacesession.h \
```

```
                include/vipapi/vip_interface.h \
                include/vipapi/vip_link.h \
                include/vipapi/vip_port.h \
                include/vipapi/vip_protocolmessage.h \
                include/vipapi/vip_loggingentity.h \
                include/vipapi/vip_networkprotocol.h \
                include/vipapi/vip_protocolsession.h \
                include/vipapi/vip_networkprotocolsession.h \
                include/vipapi/vip_sessionmap.h \
                include/vipapi/vip_sessionmapentry.h \
                include/protocols/ipminsession.h \
                include/protocols/ipmin.h \
                include/protocols/ipminmessage.h \
                include/protocols/ethminsession.h \
                include/protocols/ethmin.h \
                include/protocols/ethminmessage.h \
                include/protocols/udpminsession.h \
                include/protocols/udpmin.h \
                include/protocols/udpminmessage.h \
                include/vipapi/vip_component.h \
                include/links/cable.h \
                include/links/funk.h \
                include/vipfactory/vip_linkfactory.h \
                include/vipfactory/vip_protocolfactory.h \
                include/vipapi.h

# List all C++ source files here.
SRCFILES        = src/applications/viptime.cpp \
                src/applications/viptimemessage.cpp \
                src/vipapi/vip_protocol.cpp \
                src/vipapi/vip_application.cpp \
                src/vipapi/vip_participants.cpp \
                src/vipfactory/vip_protocolsessionfactory.cpp \
                src/vipfactory/vip_connectorfactory.cpp \
                src/connectors/vip_hub_8.cpp \
                src/components/workstation.cpp \
                src/vipfactory/vip_componentfactory.cpp \
                src/vipapi/vip_connector.cpp \
                src/vipapi/vip_simulation.cpp \
                src/vipapi/vip_interface.cpp \
                src/vipapi/vip_interfacesession.cpp \
                src/vipapi/vip_link.cpp \
                src/vipapi/vip_port.cpp \
                src/vipapi/vip_protocolmessage.cpp \
                src/vipapi/vip_loggingentity.cpp \
                src/vipapi/vip_networkprotocol.cpp \
                src/vipapi/vip_protocolgraph.cpp \
                src/vipapi/vip_sessionmap.cpp \
                src/vipapi/vip_protocolsession.cpp \
                src/vipapi/vip_networkprotocolsession.cpp \
                src/vipapi/vip_sessionmapentry.cpp \
                src/protocols/udpmin.cpp \
                src/protocols/udpminmessage.cpp \
```

```
                src/protocols/udpminsession.cpp \
                src/protocols/ipmin.cpp \
                src/protocols/ipminmessage.cpp \
                src/protocols/ipminsession.cpp \
                src/protocols/ethmin.cpp \
                src/protocols/ethminmessage.cpp \
                src/protocols/ethminsession.cpp \
                src/vipapi/vip_component.cpp \
                src/links/cable.cpp \
                src/applications/viptimesession.cpp \
                src/links/funk.cpp \
                src/vipfactory/vip_linkfactory.cpp \
                src/vipfactory/vip_protocolfactory.cpp


# List C language header files here.
C_HEADERS       =

# List C language source files here.
C_SOURCES       =

# You can add flags to be appended to each compilation.
USERCFLAGS      = -Iinclude

# You can add flags to be appended to the link step.
USERLDFLAGS     = -L/usr/local/xerces-c-src2_0_0/lib -lxerces-c

# Add your own libraries here for linkage.
USERLIBS        =

# List all DaSSF header files to be used with the libraries above.
USERLIBHDRS     =

# Put the name of the DML file that specifies run-time information
RUNTIME         =

# Put the name of the DML file that specifies the model
MODEL           =

# Put the name of the DML file that specifies hardware profile
MACHINE         =

# Put files that need to be cleaned up
USER_CLEANUP    =

# Add your rules here

# You're done by now, don't touch the following lines.
include $(D2DIR)/config/Makefile.application
```

Anhang C

Simulationsinterface - DTD

```
<!-- Simulation (1.0)
     Document Type Definition
-->

<!ELEMENT simulation   (modelsection , runtimesection)>
<!ELEMENT modelsection (component | connector | link | connection)*>
<!ELEMENT runtimesection (executer*)>

<!ELEMENT executer EMPTY>
<!ATTLIST executer executer_id ID #REQUIRED>
<!ATTLIST executer application_idref IDREF #REQUIRED>


<!ELEMENT connector EMPTY>
<!ATTLIST connector connector_id ID #REQUIRED>
<!ATTLIST connector type (8Hub|4Hub) #REQUIRED>
<!ATTLIST connector statelogging (yes|no) "no">
<!ATTLIST connector eventlogging (yes|no) "no">
<!ATTLIST connector statebrekpoint (yes|no) "no">
<!ATTLIST connector eventbreakpoint (yes|no) "no">


<!ELEMENT link EMPTY>
<!ATTLIST link link_id ID #REQUIRED>
<!ATTLIST link type (Kabel|Funk) #REQUIRED>
<!ATTLIST link length CDATA #IMPLIED>
<!ATTLIST link bandwidth CDATA #IMPLIED>
<!ATTLIST link statelogging (yes|no) "no">
<!ATTLIST link eventlogging (yes|no) "no">
<!ATTLIST link statebrekpoint (yes|no) "no">
<!ATTLIST link eventbreakpoint (yes|no) "no">

<!ELEMENT component (protocolgraph)>
<!ATTLIST component type (Workstation) #REQUIRED>
<!ATTLIST component component_id ID #REQUIRED>
<!ATTLIST component statelogging (yes|no) "no">
```

```
<!ATTLIST component eventlogging (yes|no) "no">
<!ATTLIST component statebrekpoint (yes|no) "no">
<!ATTLIST component eventbreakpoint (yes|no) "no">


<!ELEMENT protocolgraph (protocol*)>
<!ATTLIST protocolgraph protocolgraph_id ID #REQUIRED>
<!ELEMENT protocol EMPTY>
<!ATTLIST protocol protocol_id ID #REQUIRED>
<!ATTLIST protocol protocol_idref IDREF #IMPLIED>
<!ATTLIST protocol type (TCPmin|UDPmin|IPmin|ETHmin) #REQUIRED>
<!ATTLIST protocol address CDATA "0">
<!ATTLIST protocol application (no|yes) "no">
<!ATTLIST protocol statelogging (yes|no) "no">
<!ATTLIST protocol eventlogging (yes|no) "no">
<!ATTLIST protocol statebrekpoint (yes|no) "no">
<!ATTLIST protocol eventbreakpoint (yes|no) "no">

<!ELEMENT connection EMPTY>
<!ATTLIST connection link_idref IDREF #REQUIRED>
<!ATTLIST connection port CDATA "1">
<!ATTLIST connection component_refid IDREF #REQUIRED>
<!ATTLIST connection connector_refid IDREF #REQUIRED>
```

Anhang D

Beispielsimulation - Konfiguration

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE simulation SYSTEM "simulation-1.0.dtd">

<simulation>
<modelsection>

<connector connector_id="1" type="8Hub"/>

<link link_id="1" type="Kabel" length="100" bandwidth="10"/>
<link link_id="2" type="Kabel" length="100" bandwidth="10"/>
<link link_id="3" type="Kabel" length="100" bandwidth="10"/>
<link link_id="4" type="Kabel" length="100" bandwidth="10"/>
<link link_id="5" type="Kabel" length="100" bandwidth="10"/>

<component component_id="1" type="Workstation">
<protocolgraph protocolgraph_id="1">
<protocol protocol_id="1" protocol_idref="2" type="VIP_TIME"/>
<protocol protocol_id="2" protocol_idref="3" type="UDPmin" address="12"/>
<protocol protocol_id="3" protocol_idref="4" type="IPmin" address="192.168.0.1"/>
<protocol protocol_id="4" type="ETHmin" address="aa:12:34:43:fd:e1"/>
</protocolgraph>
</component>

<component component_id="2" type="Workstation">
<protocolgraph protocolgraph_id="2">
<protocol protocol_id="5" protocol_idref="6" type="VIP_TIME"/>
<protocol protocol_id="6" protocol_idref="7" type="UDPmin" address="12"/>
<protocol protocol_id="7" protocol_idref="8" type="IPmin" address="192.168.0.2"/>
<protocol protocol_id="8" type="ETHmin" address="1d:45:89:ca:12:8f"/>
</protocolgraph>
</component>

<component component_id="3" type="Workstation">
<protocolgraph protocolgraph_id="3">
<protocol protocol_id="9" protocol_idref="10" type="VIP_TIME"/>
<protocol protocol_id="10" protocol_idref="11" type="UDPmin" address="12"/>
```

```
<protocol protocol_id="11" protocol_idref="12" type="IPmin" address="192.168.0.3"/>
<protocol protocol_id="12" type="ETHmin" address="12:34:43:21:ae:b1"/>
</protocolgraph>
</component>

<component component_id="4" type="Workstation">
<protocolgraph protocolgraph_id="4">
<protocol protocol_id="13" protocol_idref="14" type="VIP_TIME"/>
<protocol protocol_id="14" protocol_idref="15" type="UDPmin" address="12"/>
<protocol protocol_id="15" protocol_idref="16" type="IPmin" address="192.168.0.4"/>
<protocol protocol_id="16" type="ETHmin" address="34:43:af:fa:a3:22"/>
</protocolgraph>
</component>

<component component_id="5" type="Workstation">
<protocolgraph protocolgraph_id="5">
<protocol protocol_id="17" protocol_idref="18" type="VIP_TIME"/>
<protocol protocol_id="18" protocol_idref="19" type="UDPmin" address="1212"/>
<protocol protocol_id="19" protocol_idref="20" type="IPmin" address="192.168.0.253"/>
<protocol protocol_id="20" type="ETHmin" address="41:ae:f3:50:03:51"/>
</protocolgraph>
</component>

<connection link_idref="1" component_refid="1" connector_refid="1" port="1"/>
<connection link_idref="2" component_refid="2" connector_refid="1" port="2"/>
<connection link_idref="3" component_refid="3" connector_refid="1" port="3"/>
<connection link_idref="4" component_refid="4" connector_refid="1" port="4"/>
<connection link_idref="5" component_refid="5" connector_refid="1" port="5"/>

</modelsection>
<runtimesection>
<executer executer_id="executer1" application_idref="1"/>>
</runtimesection>
</simulation>
```

Abkürzungsverzeichnis

API Application Programming Interface

ARP Address Resolution Protocol

ATM Asynchronous Transfer Mode

BMBF Bundesministerium für Bildung und Forschung

CSMA/CD Carrier Sense Multiple Access with Collision Detection

DaSSF Dartmouth Scalable Simulation Framework

DML Domain Modeling Language

DSL Digital Subscriber Line

DTD Document Type Definition

DVI DeVice Independent

extDaSSF Extended Dartmouth Scalable Simulation Framework

FTP File Transfer Protocol

HTML Hypertext Markup Language

IP Internet Protocol

ISDN Integrated Services Digital Network

LAN Local Area Network

LLC Logical Link Control

LMU Ludwig-Maximilians-Universität

MAC Medium Access Control

MHS Message Handling System

OSPF Open Shortest Path First

PDF Portable Document Format

RFC Request For Comment

RIP Routing Information Protocol

RWTH Rheinisch-Westfälischen Technischen Hochschule

SAP Service Access Point

SMP Shared Memory Processing

SSF Scalable Simulation Framework

TCP Transmission Control Protocol

TFTP Trivial File Transfer Protocol

TPDU Transport Protocol Data Unit

UDP User Datagram Protocol

VIP Virtuelles Informatik Praktikum

WAN Wide-Area Network

XML Extensible Markup Language

Literaturverzeichnis

[Bagr 94] BAGRODIA, RAJIVE: *Language support for parallel discrete-event simulations.* In: *Winter Simulation Conference*, Seiten 1324–1331, 1994, citeseer.nj.nec.com/bagrodia94language.html .

[Cowi 99] COWIE, JAMES H.: *Scalable Simulation Framework API Reference Manual Version 1.0.* Technischer Bericht, 1999.

[FeTr 94] FERSCHA, ALOIS und SATISH K. TRIPATHI: *Parallel and Distributed Simulation of Discrete Event Systems.* Technischer Bericht CS-TR-3336, 1994, citeseer.nj.nec.com/ferscha95parallel.html .

[HAN 99] HEGERING, HEINZ-GERD, SEBASTIAN ABECK und BERNHARD NEUMAIR: *Integriertes Management vernetzter Systeme – Konzepte, Architecturen und deren betrieblicher Einsatz.* dpunkt Verlag, ISBN 3-932588-16-9, 1999. 607 p.

[LOZ 01] LIU, C., M. A. ORGUN und K. ZHANG: *A parallel execution model for Chronolog.* International Journal of Computer Systems Science and Engineering, 16(4), 2001, citeseer.nj.nec.com/liu01parallel.html .

[LuNi 01] LUI, JASON und DAVID M. NICOL: *Dartmouth Scalable Simulation Framework (Dartmouth SSF or DaSSF) User's Manual Version 3.1*, 2001, http://www.cs.dartmouth.edu/research/DaSSF/docs.html .

[Müll 97] MÜLLER, UDO: *C++-Implementierungstechniken: Idiome, Entwurfsmuster, Containerbibliotheken.* International Thomson Publishing, ISBN 3-8266-0254-4, 1997. 409 p.

[Oest 01a] OESTEREICH, BERND: *Die UML - Kurzreferenz für die Praxis.* Oldenbourg Wissenschaftsverlag GmbH, ISBN 3-486-25637-8, 2001, http://www.oldenbourg-verlag.de . 148 p.

[Oest 01b] OESTEREICH, BERND: *Objektorientierte Softwareentwicklung : Analyse und Design mit der Unified modeling language.* Oldenbourg Wissenschaftsverlag GmbH, ISBN 3-486-25573-8, 2001, http://www.oldenbourg-verlag.de . 355 p.

[PDB 96] PETERSON, LARR L., BRUCE S. DAVIE und ANDREW C BAVIER: *x-kernel Tutorial*, 1996.

[PNL+ 01] PERRONE, F., D. NICOL, J. LIU, C. ELLIOT und D. PEARSON: *Simulation modeling of large-scale ad-hoc sensor networks*, 2001, citeseer.nj.nec.com/perrone01simulation.html .

[Stev 98] STEVENS, W. RICHARD: *UNIX Network Programming Volume 1 – Network APIs: Socket and XTI*. Prentice HALL PTR, ISBN 0-13-490012-X, 1998. 1009 p.

[Stro 97] STROUSTRUP, BJARNE: *Die C++-Programmiersprache*. Prentice Hall Verlag GmbH, ISBN 3-89319-386-3, 1997. 717 p.

[Tann 00] TANNENBAUM, ANDREW S.: *Computernetzwerke*. Prentice Hall Verlag GmbH, ISBN 3-8272-9536-X, 2000. 847 p.

[TTK 98] TEO, Y., S. TAY und K. KONG: *Structured Parallel Simulation Modeling and Programming*, 1998, citeseer.nj.nec.com/teo98structured.html .

Index