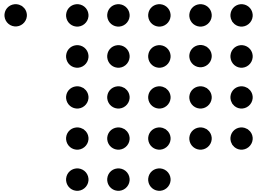
SIEMENS

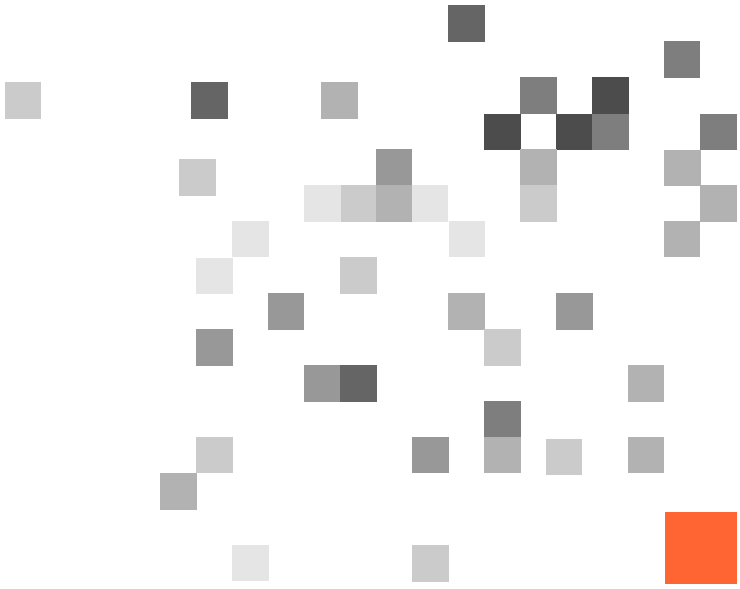


Fachhochschule Köln
University of Applied Sciences Cologne

07 Fakultät für Informations-, Medien- und Elektrotechnik, Studiengang Master of Science in Information Engineering

MASTERARBEIT

Entwicklung eines ISO/OSI 8073 / TP4-Netzwerk-Stacks für Linux



Dipl.-Ing. Ralf Weiden
mail*ralf-weiden.de (mit @ statt *)
Matr.-Nr. 11006584

Abgabe : 05.12.2003
Referent : Prof. Dr. rer. nat. Carsten Vogt
Korreferent : Prof. Dr.-Ing. Uwe Dettmar

Vorwort

Wie groß kann ein Hut sein?

Am Ende meiner Masterarbeit angelangt, weiß ich zumindest, dass es Hüte gibt, groß genug, um meine Vollzeit-Berufstätigkeit bei der Siemens AG Köln und einen im Institut für Nachrichtentechnik der Fachhochschule Köln parallel durchgeführten Master-Studiengang unter sich zu vereinen.

Zu guter Letzt bot dieser Hut noch Platz für diese Masterarbeit, welcher ein im Rahmen meiner Berufstätigkeit durchgeführtes kundenspezifisches IT-Projekt zugrunde liegt.

Damit nicht genug, steckt hinter der vorliegenden Masterarbeit das Bestreben, gleich mehrere Ziele unter einen Hut zu bringen, geht es doch darum,

- dem Endkunden einen Überblick über die interne Arbeitsweise eines kleinen Teils des gelieferten Systems zu bieten,
- den Projektkollegen oder bislang unbeteiligten Technikern einen hilfreichen und schnellen Einstieg für Erweiterungen oder eventuelle Störungsbeseitigung zu ermöglichen,
- und nicht zuletzt eine wissenschaftlich fundierte Masterarbeit als krönenden Abschluss meines Studiums zu verfassen, ohne dabei allzu sehr auf altbekannte Hüte zurückzugreifen.

Meinen Hut ziehe ich nun vor all jenen, die mich im Verlauf dieses Vorhabens tatkräftig unterstützten. Dazu zählen ganz sicher meine Siemens-Projektkollegen, die ich auch in der heißen Projekt-Endphase noch mit „semi-projektbezogenen“ Fragen löchern durfte, meine Betreuer auf akademischer Seite, Hr. Prof. Vogt und Hr. Prof. Dettmar, die mich wohlbehütet bis zur Ziellinie brachten, und meine Korrekturleser Axel Seinsche, Kai Werner und Dr. Axel Buch mit ihren behutsamen Verbesserungen.

Köln, November 2003
Ralf Weiden

Ich versichere, dass ich die Masterarbeit selbständig angefertigt und keine anderen als die angegebenen und bei Zitaten kenntlich gemachten Quellen und Hilfsmittel benutzt habe. Sämtlicher Programmcode unterliegt dem Copyright und ist Eigentum der Siemens AG, Köln.

_____

Ralf Weiden Köln, 16.01.04

Inhalt

Abbildungen

Tabellen

Listings

Kapitel-Übersicht

Thema dieser Masterarbeit ist die Ablösung eines bestehenden kundenspezifischen IT-Systems durch moderne PC-Technologie mit dem Betriebssystem Linux. Die Software des neuen Systems sollte sich nach außen identisch mit der des abgelösten Systems verhalten. Dies erforderte unter anderem die Implementierung eines ISO/OSI 8073/TP4-Netzwerkprotokoll-Stacks.

- **Kapitel 1** gibt einen Überblick über das abgelöste und neue System und definiert die von mir übernommenen Aufgaben. Eine Teilmenge daraus ist Grundlage für diese Masterarbeit.
- In **Kapitel 2** wird das geschaffene Applikations-Framework beschrieben, welches die Basis für die Software-Prozesse des neuen Systems bildet.
- Thema von **Kapitel 3** ist das ISO/OSI 8073/TP4-Protokoll, kombiniert mit allgemeinen Grundlagen für die Realisierung von gesicherten Transportprotokollen der OSI-Ebene 4.
- **Kapitel 4** behandelt die OSI-Ebene 2 mit den Themen *Data Link Control* und *Raw Sockets*. Diese bilden die technische Grundlage zur Implementierung des TP4-Protokollstacks.
- Der für den Empfang und die Vorverarbeitung von Datentelegrammen zuständige Prozess *RECCP* wird in **Kapitel 5** beschrieben.
- Den Kern dieser Arbeit bildet **Kapitel 6** mit der Beschreibung des Prozesses *SENDCP*. Dieser übernimmt sämtliche für die Realisierung eines Transportprotokolls mit gesicherter Datenübertragung erforderlichen Aufgaben.
- **Kapitel 7** fasst die zur Analyse des SENDCP-Laufzeitverhaltens durchgeführten Performance-Messungen zusammen.
- Gleichzeitig zurück und nach vorn schließlich blickt **Kapitel 8** mit dem Schlusswort.
- Eine Auflistung der verwendeten Dokumente und Informationsquellen enthält das Literaturverzeichnis in **Kapitel 9.**

Allgemeine Vereinbarungen

In dieser Masterarbeit werden folgende Schriftstile und Kennzeichnungen verwendet:

- In *Kursivschrift* erscheinen alle Klassen-, Funktions-, Variablen- und Prozessnamen, die ihren Ursprung im erstellten Programmcode haben.
- Namen von Funktionen bzw. Methoden erscheinen angelehnt an die C++ -Notation in der Form *Funktionsname()*.
- Namen von C++ - Klassen beginnen mit dem Buchstaben *„C“*, etwa *CTP4Telegram*, sofern es sich um eine von mir erstellte Klasse handelt.
- Wenn nicht besonders gekennzeichnet, sind alle Zahlen Dezimalzahlen. Hexadezimalzahlen ist das in C++ übliche ‚0x' vorangestellt.
- Verweise auf das Literaturverzeichnis stehen in eckigen Klammern [ ].
- Die Begriffswelt der Netzwerkprotokolle ist durchsetzt mit *Rahmen, Paketen* und *Segmenten* beziehungsweise deren englischen Pendants. Diese Namen wurden zur Unterscheidung der in den einzelnen OSI-Schichten ausgetauschten Protokolldateneinheiten eingeführt. Bei allgemeinen Beschreibungen bevorzugt diese Masterarbeit den übergeordneten Begriff *Telegramm*. Handelt es sich konkret um ein Telegramm der OSI-Schicht 4, kommt zur Kennzeichnung die Abkürzung *TPDU* für *Transport Protocol Data Unit* zur Verwendung.

1 Einleitung

1.1 Allgemeines

Kern dieser Masterarbeit ist ein im Rahmen meiner Berufstätigkeit bei der Siemens AG Köln, Bereich *Information Technology Plant Solutions* (I&S IT PS), durchgeführtes kundenspezifisches IT-Projekt. Im Verlauf dieses Projekts wurde unter anderem ein ablösebedürftiger AEG-Rechner durch aktuelle Rechnertechnologie ersetzt sowie die kundenspezifische Anwender-Software auf die neue Plattform portiert. Ferner musste ein vormals durch eine Hardware-Baugruppe realisierter ISO/OSI-TP4-Netzwerk-Protokollstack durch eine neu zu schaffende Software-Lösung emuliert werden.

Dieses Kapitel beschreibt die Struktur des abzulösenden Systems, gibt einen Überblick über die neue Lösung und beschreibt abschließend die von mir innerhalb des Projekts übernommenen Aufgaben.

1.2 Die Struktur des abzulösenden Systems

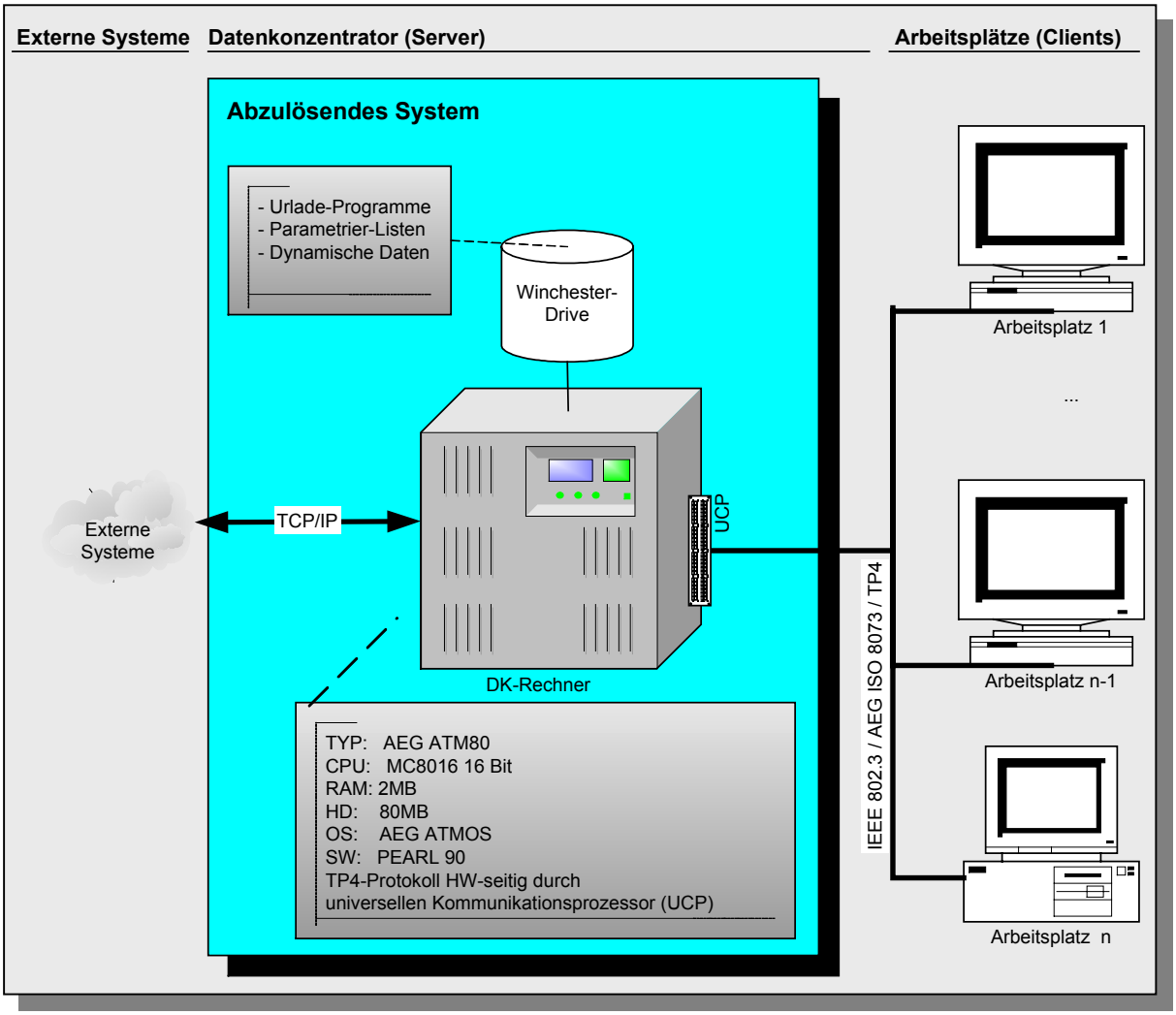


Abbildung 1.1: Übersicht über das abzulösende System

Die in Abbildung 1.1 skizzierte Kundenanlage wurde in den 1980er Jahren installiert und ist seitdem in Benutzung. Nach heutigen Begriffen handelt es sich um eine Client-Server-Architektur, bei der ein zentraler Server (*„Datenkonzentrator“*, kurz *„DK-Rechner“*) und mehrere angeschlossene Clients (*„Arbeitsplätze“*) miteinander kommunizieren.

Der Datenaustausch zwischen DK-Rechner und den angeschlossenen Arbeitsplätzen erfolgt auf den OSI-Schichten 1 und 2 über ein LAN nach IEEE 802.3 („Ethernet“). Auf der OSI-Ebene 4 wird eine AEG-Implementation des TP4-Transportprotokolls gemäß ISO/OSI 8073 eingesetzt.

Als einheitliches LAN ohne Subnetze und Vermittlungseinrichtungen wie Router etc. ist beim vorliegenden Netz ein Vermittlungsprotokoll der OSI-Schicht 3 verzichtbar und nicht realisiert.

1.2.1 Der Datenkonzentrator

Das Herzstück der Anlage ist der *Datenkonzentrator*. Er übernimmt in der Hauptsache folgende Aufgaben:

- Urladen der Arbeitsplätze mit einem Betriebs- und Anwendungssystem
- „Fileserver-Funktionalität“: Zentrale Speicherung von statischen Konfigurationsdateien, Parametrierlisten etc., sowie Versorgung der Arbeitsplätze mit diesen Daten. Ferner werden im laufenden Betrieb entstehende Daten abgelegt bzw. versendet
- Kommunikation mit und Datenaufbereitung für über TCP/IP angekoppelte externe Systeme
- Systemüberwachung

Alle angeschlossenen Arbeitsplätze können eine TP4-Verbindung mit dem DK-Rechner aufbauen und Daten von dort anfordern beziehungsweise auf diesem ablegen.

Zusätzlich übernimmt der DK-Rechner Systemüberwachungs-Funktionen. Dazu wertet er die periodisch von den Arbeitsplätzen als Broadcast versendeten Lebenszeichen-Telegramme aus. Bleibt ein solches aus, wird vom DK-Rechner eine entsprechende Störmeldung generiert und der als ausgefallen erkannte Arbeitsplatz in einer internen Liste als „nicht im Netz“ markiert. Der DK-Rechner selbst ist aus Verfügbarkeitsgründen doppelt im System vorhanden. Aktiv ist jeweils nur ein DK-Rechner.

Der Datenkonzentrator ist ein AEG-Prozessrechner aus der Reihe AEG ATM 80 mit folgenden Kenndaten:

CPU	Motorola MC8016 16 Bit
Betriebssystem	AEG ATMOS
Arbeitsspeicher	2 MB RAM
Externer Speicher	80MB Winchester-Platte
Anwendungs-Software geschrieben in	PEARL90

Tabelle 1.1: Daten des DK-Rechners

Die Kommunikations-Abwicklung über das ISO/OSI-TP4-Protokoll wird mittels einer sogenannten *UCP-Baugruppe* (*„Universal Communications Processor“*) als Hardware realisiert. Diese Baugruppe übernimmt alle kommunikationsrelevanten Aufgaben der OSI-Schichten 1..4. Dem Software-Entwickler stehen die Dienste des UCP über Funktionsaufrufe zur Verfügung.

Als permanenter Datenspeicher kommt ein 80MB Winchester-Drive zum Einsatz. Insbesondere für dieses Laufwerk ist die Ersatzteil-Versorgung nicht mehr gewährleistet. Daher resultiert die Entscheidung, den gesamten DK-Rechner durch eine aktuelle PC-basierte Rechnerplattform zu ersetzen.

1.2.2 Die Arbeitsplätze

Auf den Arbeitsplätzen laufen verschiedene, kundenspezifische Anwenderprogramme.

Die Hardware der Arbeitsplätze stammt aus der gleichen Modellreihe wie die des DK-Rechners. Allerdings sind die Arbeitsplätze als Discless-Stationen ohne eigene Laufwerke ausgeführt.

Nach dem Einschalten eines Arbeitsplatzes fordert dieser zunächst über das Netz das Urladen eines Betriebssystems zusammen mit einem arbeitsplatz-spezifischen Anwendungs-Programm an. Diese befinden sich auf der Festplatte des DK-Rechners.

Im Anschluss an den TP4-Verbindungsaufbau zum DK-Rechner werden das Betriebssystem, das dem Arbeitsplatz zugeordnete Anwendungsprogramm sowie eventuell zugehörige Parametrierdaten zum Arbeitsplatz übertragen. Nach fehlerfreier Übertragung wird die Netzverbindung zum DK-Rechner abgebaut. Der Arbeitsplatz startet daraufhin sein Betriebssystem und das geladene Anwendungsprogramm.

Im laufenden Betrieb kann der Arbeitsplatz eine neue Verbindung zum DK-Rechner aufbauen und dynamische (Betriebs-) Daten mit diesem austauschen. Jeder Arbeitsplatz kann jeweils genau eine Verbindung zur Zeit etablieren.

1.3 Die Struktur des neuen Systems

Im neu geschaffenen System ist der DK-Rechner durch heutige marktübliche Rechner-Technologie auf PC-Basis mit Linux als Betriebssystem ersetzt. Die Kenndaten dieses neuen Systems:

Hardware	Industrie-PC mit Standard-Komponenten
CPU	Intel Pentium III 1400MHz Dual-CPU
Arbeitsspeicher	1 Gigabyte RAM
Externer Speicher	RAID-System mit gespiegelten Platten
Betriebssystem	SuSE Linux 8.2
Anwendungs-Software geschrieben in	C / C++

Tabelle 1.2: Daten des neuen DK-Rechners

Im Verlauf des Projekts wurde die Anwendungssoftware des DK-Rechners von PEARL nach C / C++ portiert und an die neue Betriebssystem- und Hardware-Plattform angepasst. Die logischen Abläufe der Software blieben unverändert. Abbildung 1.2 gibt einen Überblick über die neue Systemstruktur.

Sämtliche Urladeprogramme, Parametrierlisten und weitere für die Arbeitsplätze erforderliche Dateien liegen nun unverändert auf der Systemfestplatte des neuen DK-Rechners.

Eine Modernisierung der Arbeitsplätze und deren Anwendungssoftware war nicht Teil des Projekts. Die bestehenden Arbeitsplätze sind unverändert über TP4 mit dem neuen DK-Rechner gekoppelt.

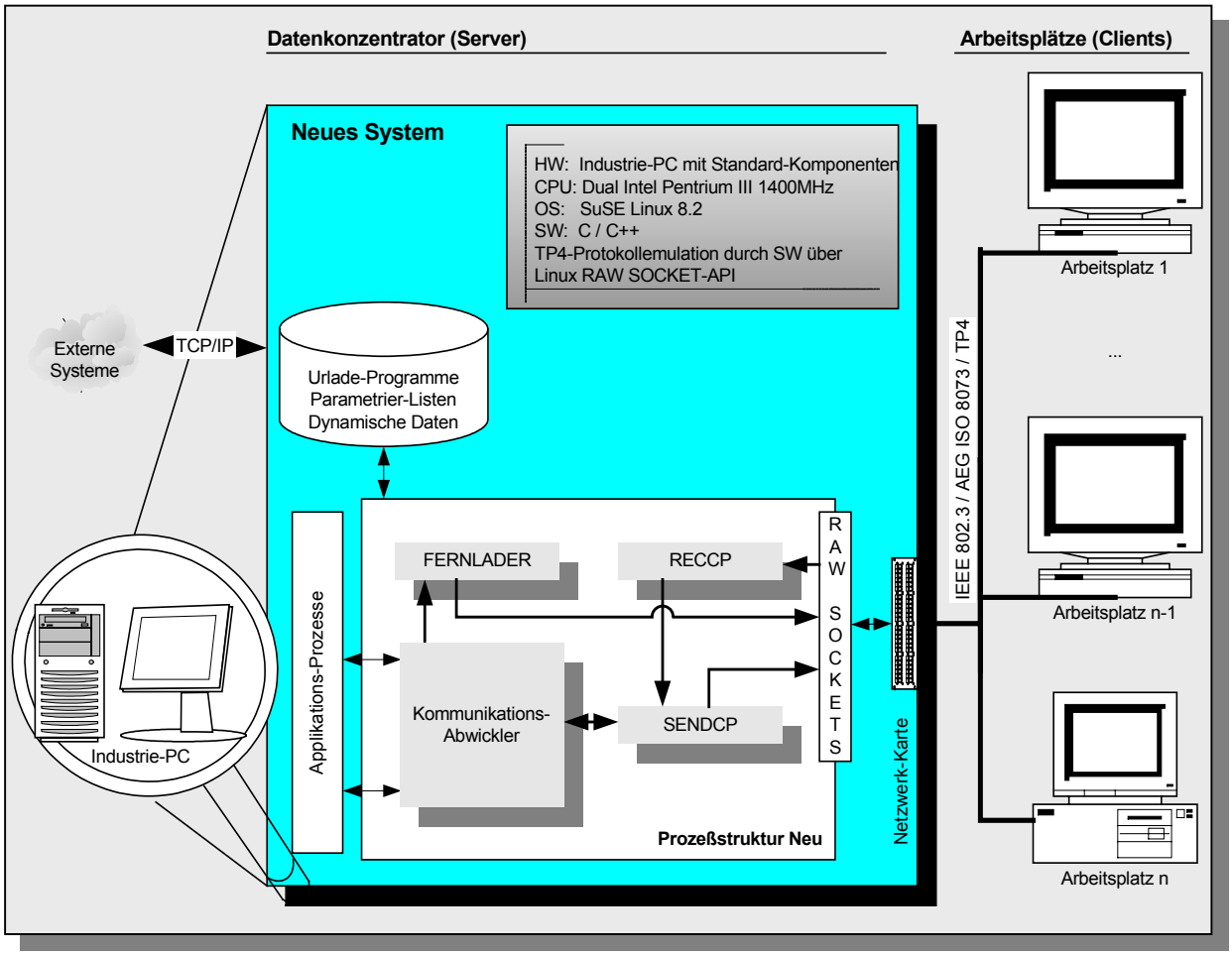


Abbildung 1.2: Struktur des neuen Systems

Auf der DK-Rechner-Seite wird das vormals durch die UCP-Baugruppe realisierte TP4-Protokoll durch die beiden neu geschaffenen Software-Prozesse *RECCP* und *SENDCP* emuliert. Die beiden Prozesse stellen die typischen Dienste eines Transportprotokolls, wie Verbindungs-Auf/Abbau und zuverlässige Datenübertragung, bereit. Der aus dem Altsystem portierte und angepasste *Kommunikationsabwickler* nutzt diese Dienste.

Abbildung 1.3 zeigt die für diese Masterarbeit relevanten Prozesse mit deren Verschaltung innerhalb des neuen Systems und ordnet sie ungefähr den einzelnen OSI-Schichten zu.

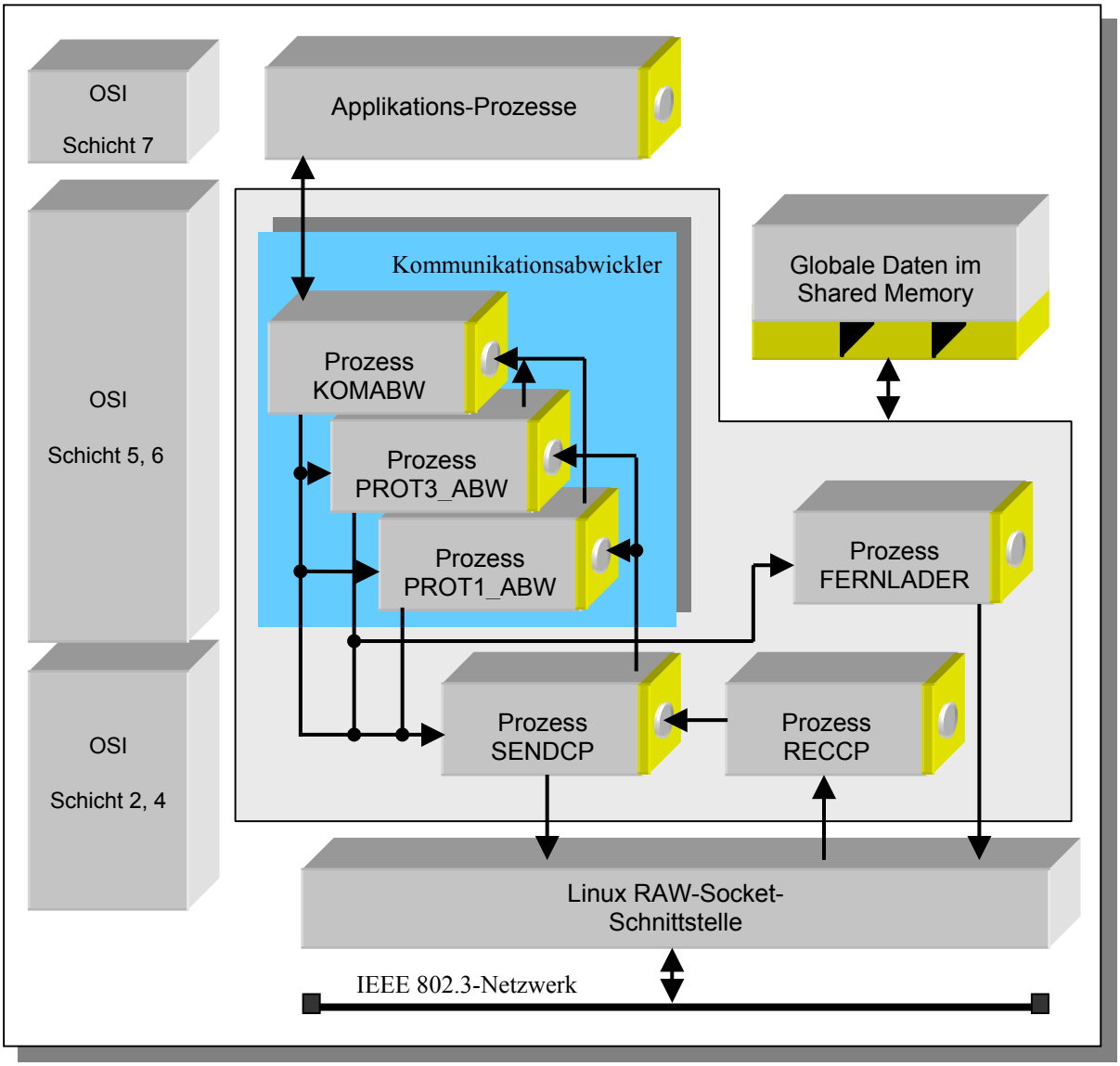


Abbildung 1.3: Prozesse des neuen Systems

Die Aufgaben der OSI-Schichten 2 und 4 werden durch die beiden Prozesse ***SENDCP*** und ***RECCP*** übernommen. Sie realisieren mit dem ISO/OSI 8073/TP4-Protokoll ein zuverlässiges verbindungsorientiertes Transportprotokoll der OSI-Ebene 4.

Netzwerkseitig senden bzw. empfangen diese Prozesse vollständige IEEE 802.3-Rahmen (OSI Schicht 2) über die RAW-Socket-Schnittstelle des Linux-Bestriebssystems. Durch Nutzung dieser standardisierten Programmierschnittstelle ist für die Übertragung von IEEE 802.3-Rahmen kein direkter Zugriff auf die Netzwerkkarten-Hardware erforderlich, was die Erstellung eines hardware-spezifischen Treibers überflüssig macht. Somit ist der Einsatz einer beliebigen linux-kompatiblen Netzwerkkarte möglich.

Die Aufgaben der Prozesse im Überblick:

- Der Prozess **RECCP** *(„Receive Communication Process")* übernimmt die Netzwerk-Empfangsrichtung. Über einen *Raw Socket* erhält er fortlaufend vollständige IEEE 802.3-Rahmen, die er filtert, in eine interne Struktur konvertiert und an den Prozess SENDCP weiterleitet.
- Die gesamte für ein gesichertes Transportprotokoll erforderliche Verarbeitungslogik wurde im Prozess **SENDCP** *(„Send Communication Process")* zusammengefasst. SENDCP übernimmt den Aufbau und die Verwaltung von Verbindungen, überwacht Timeout-Zeiten und das Versenden bzw. Empfangen von Acknowledgements. Vom Prozess RECCP erhaltene Datentelegramme werden bei korrekter Sequenznummer an den Kommunikationsabwickler weitergeleitet und mit einem Acknowledge-Telegramm der Gegenseite bestätigt. Umgekehrt bietet SENDCP dem Kommunikationsabwickler alle benötigten OSI-Schicht 4-Dienstelemente zum Aufbau einer Verbindung und zum Übertragen von Telegrammen. Zu sendende Transportdateneinheiten (TPDUs) werden in IEEE 802.3-Rahmen gekapselt und über einen RAW-Socket an das Linux-Betriebssystem übergeben. Linux übernimmt die anschließende unveränderte Weiterleitung über die Netzwerkkarte auf das Netzwerk.

Der ***Kommunikationsabwickler*** organisiert die Zugriffe der Applikations-Prozesse auf das Transportprotokoll und übernimmt Aufgaben der OSI-Layer 5 und 6. Er besteht im wesentlichen aus 3 Teilprozessen:

- Dem Protokollabwickler **PROT1_ABW** zur Bearbeitung von verbindungslosen Broadcast-Telegrammen und zum Verbindungsauf- und -abbau,
- dem Protokollabwickler **PROT3_ABW** für den Nutzdatenaustausch sowie
- dem eigentlichen Kommunikationsabwickler-Prozess **KOMABW**. KOMABW koordiniert die beiden Protokollabwickler-Prozesse und übernimmt übergeordnete Aufgaben zur Kommunikationssteuerung. Dazu zählt die Lebenszeichen-Überwachung angeschlossener Arbeitsplätze und das Auflösen „hängender Verbindungen" bei Ausfall eines solchen. Außerdem regelt er die Zugriffssteuerung auf die TP4-Ebene und multiplext Anfragen höherer Schichten auf genau eine TP4-Verbindung je angeschlossenem Arbeitsplatz.

Die Prozesse des Kommunikationsabwicklers wurden aus dem abgelösten System portiert und in ihrem Verhalten nicht verändert. Daher sind sie in dieser Masterarbeit nicht weiter behandelt.

Nicht direkt in das OSI-Modell einordnen lässt sich der Prozess FERNLADER. Er übernimmt das Urladen der Arbeitsplätze mit dem zugeordneten Betriebssystem. Dazu benutzt er eine Minimal-Implementation des TP4-Protokolls und sendet TP4-Telegramme ohne Umweg über SENDCP unmittelbar über das RAW-Socket-Interface. Dies ergab sich zum einen aus der Projektstruktur, da der Fortschritt des Projekts frühzeitig einen betriebsbereiten Arbeitsplatz bedingte. Zum anderen existierte auch im abgelösten System ein Fernlade-Prozess in der geschilderten Form, zu dem allerdings keine PEARL-Sourcen vorlagen. Daher wurde dieser Prozesses neu entwickelt.

Die Interprozesskommunikation erfolgt bei den Prozessen des neuen Systems über den Message-Queue-Mechanismus des Linux-Betriebssystems. Jeder Prozess richtet bei seinem Start eine Message Queue ein, über die er Nachrichten von anderen Prozessen empfangen und ereignisgesteuert verarbeiten kann.

Zur Ablage globaler Systemdaten wird darüber hinaus beim Hochfahren des Gesamtsystems ein Shared-Memory-Speicherbereich eingerichtet, auf den alle Prozesse Zugriff haben. Außerdem kann ein Prozess innerhalb des Shared Memories dynamisch Speicherblöcke anfordern, um Nutzdaten für andere Prozesse zu hinterlegen, deren Länge die Maximallänge einer Message-Queue-Nachricht überschreiten.

1.4 Meine Aufgabenstellung

Die Projektphasen Analyse, Spezifikation und Realisierung wurden durchgeführt im Zeitraum Sommer 2002 bis November 2003. Das Projektteam bestand mit Unterbrechungen aus fünf technischen Mitarbeitern:

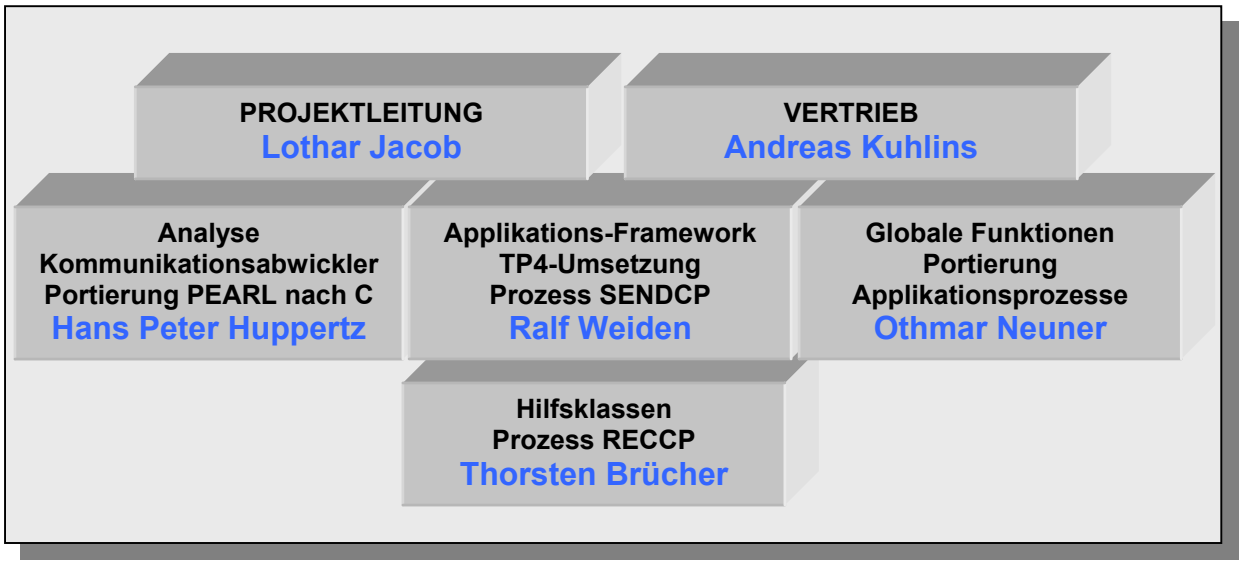


Abbildung 1.4: Das Projektteam

Einige der von mir eigenverantwortlich übernommenen Aufgaben in diesem Projekt sind Grundlage der nachfolgenden Kapitel dieser Masterarbeit. Insbesondere sind dies:

- Entwurf und Implementierung eines Software-Applikationsgerüsts (das „Applikations-Framework“) zur Bereitstellung und Kapselung einheitlicher Basisfunktionalitäten für alle zu portierenden und neu zu erstellenden Prozesse. Das Framework vereinheitlicht Initialisierung, Laufzeit-Funktionalitäten wie Logging und Interprozesskommunikation über Message Queues. Dies ist Thema von Kapitel 2.
- Die Implementierung des TP4-Protokolls. Zu dieser Aufgabe gehört die Analyse des bestehenden Protokolls sowie dessen Umsetzung im neu geschaffenen Prozess SENDCP. Die Beschreibung des Werdegangs von der Vorstellung des TP4-Protokolls über Design und Realisierung des Prozesses SENDCP bis zum Laufzeit-Verhalten der fertigen Implementierung bildet den Hauptteil dieser Masterarbeit ab Kapitel 3.

Diese Masterarbeit befasst sich demnach vorwiegend mit den OSI-Schichten 2 bis 4. Das OSI-Modell selbst findet sich beschrieben in zahlreichen Publikationen und wird daher hier als bekannt vorausgesetzt. Einführungen dazu finden sich unter anderem in [Tanenbaum00], [Detttmar02] und [Weiden99].

Die Prozesse für das neue System wurden in der Programmiersprache C++ für das Betriebssystem Linux entwickelt. Als integrierte Entwicklungsumgebung kam das Linux-Paket KDevelop 2.1 zum Einsatz. Ein Eingehen auf die Besonderheiten der Software-Entwicklung für die Unix/Linux-Plattform würde den Blick auf das Wesentliche unnötig erschweren und ist ebenfalls nicht Thema dieser Masterarbeit.

Insbesondere zur Beschreibung der objektorientierten Software-Struktur mit ihren Klassen und deren Zusammenhänge wird auf Elemente der *Unified Modeling Language* (UML) zurückgegriffen. Informationen dazu enthält [Oesterreich01].

2 Das Applikations-Framework

2.1 Allgemeines

Sowohl die aus dem Altsystem nach C++ portierten als auch die neu geschaffenen Prozesse besitzen gemeinsame Eigenschaften und Abläufe, wie:

- Initialisierungs-, Ausführungs- und Beendigungsphase
- Shellprozesse ohne grafische Oberfläche
- Ausgabe von Betriebs- und Störmeldungen mit wählbarer Ausführlichkeit in Logdateien und wahlweise zusätzlich auf die Terminal-Shell
- Einheitliche Interprozess-Kommunikation über Linux Message Queues
- Zugriff auf Konfigurationsdateien
- Abfangen und sauberes Verarbeiten von Linux-System-Signalen wie SIGTERM oder SIGINT nach Programmabbruch via *kill*-Befehl oder STRG-C

Diese elementaren Funktionen werden in einem nachfolgend beschriebenen Applikations-Framework zusammengefasst, welches die einheitliche wiederverwendbare Software-Basis aller Prozesse des neuen Systems bildet.

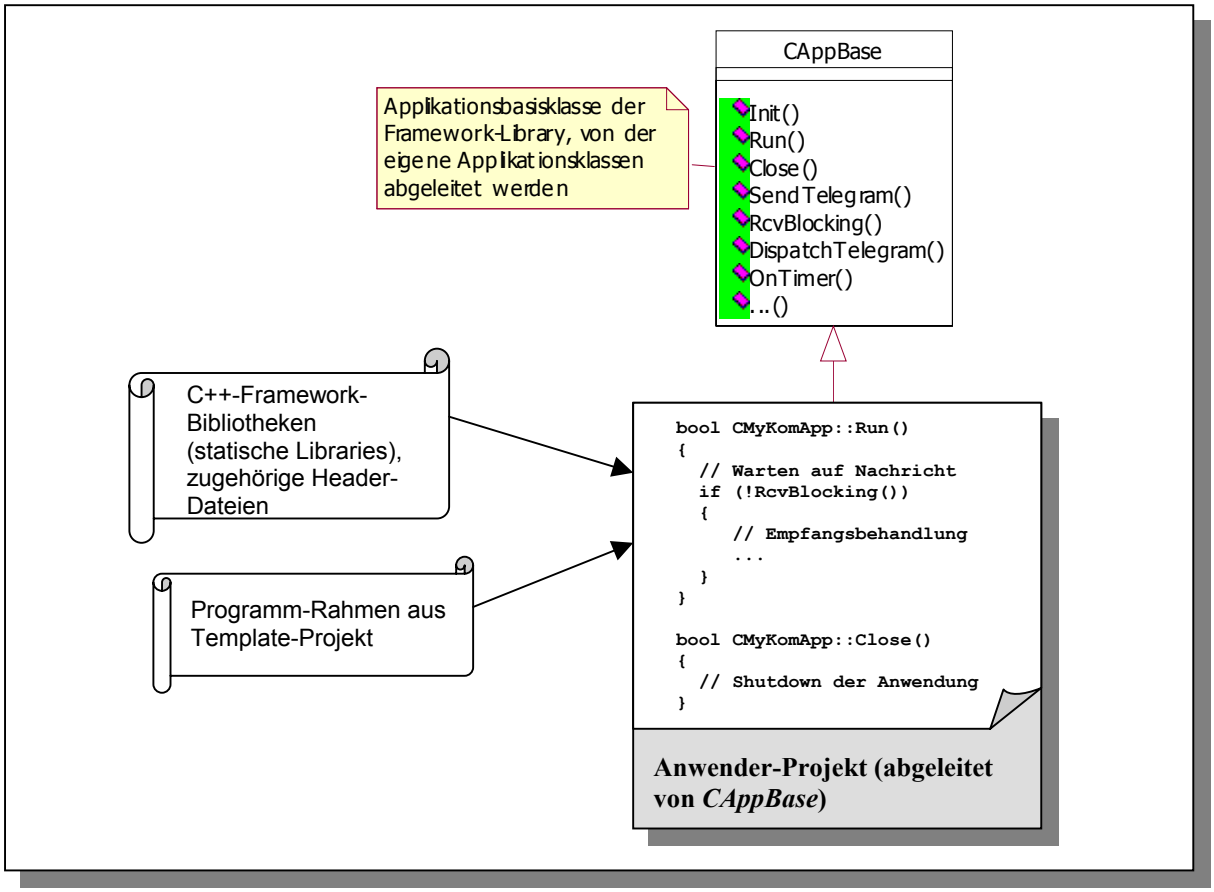


Abbildung 2.1: Anwendungs-Erstellung auf Framework-Basis

Das Framework ist realisiert als C++ - Klassenbibliothek, deren Klassen die eingangs genannten Funktionalitäten kapseln. Dem Applikations-Entwickler (dem „Benutzer“ des Frameworks) steht diese Framework-Bibliothek in Form dreier vorkompilierter statischer C++ - Programm-Libraries zur Verfügung. Zusammen mit den zugehörigen Header-Dateien werden diese zu eigenen Anwendungs-Projekten hinzu gebunden. Der Framework-Benutzer kommt auf diese Art nicht mit dem Framework-Sourcecode in Berührung.

Zusätzlich existiert ein Muster-Entwicklungsprojekt (Template), welches bereits einen ablauffähigen Beispiel-Programmrahmen enthält. In dessen Funktionsrümpfe kann mittels bereitgestellter Steuer- und Schnittstellenfunktionen des Frameworks der anwendungsspezifische Programmcode eingebracht werden.

Damit ist sichergestellt, dass alle auf Framework-Basis realisierten Prozesse den gleichen Programmrahmen benutzen und auf einheitliche Weise miteinander kommunizieren. Neben der durch diese Standardisierung erreichten Qualitätssicherung sind so neue Prozesse bei zukünftigen Erweiterungen einfach in das System einzugliedern.

Änderungen an den Framework-Libraries gehen durch einfaches Neukompilieren der Anwendungsprozesse in diese ein.

2.2 Übersicht über die Framework-Komponenten

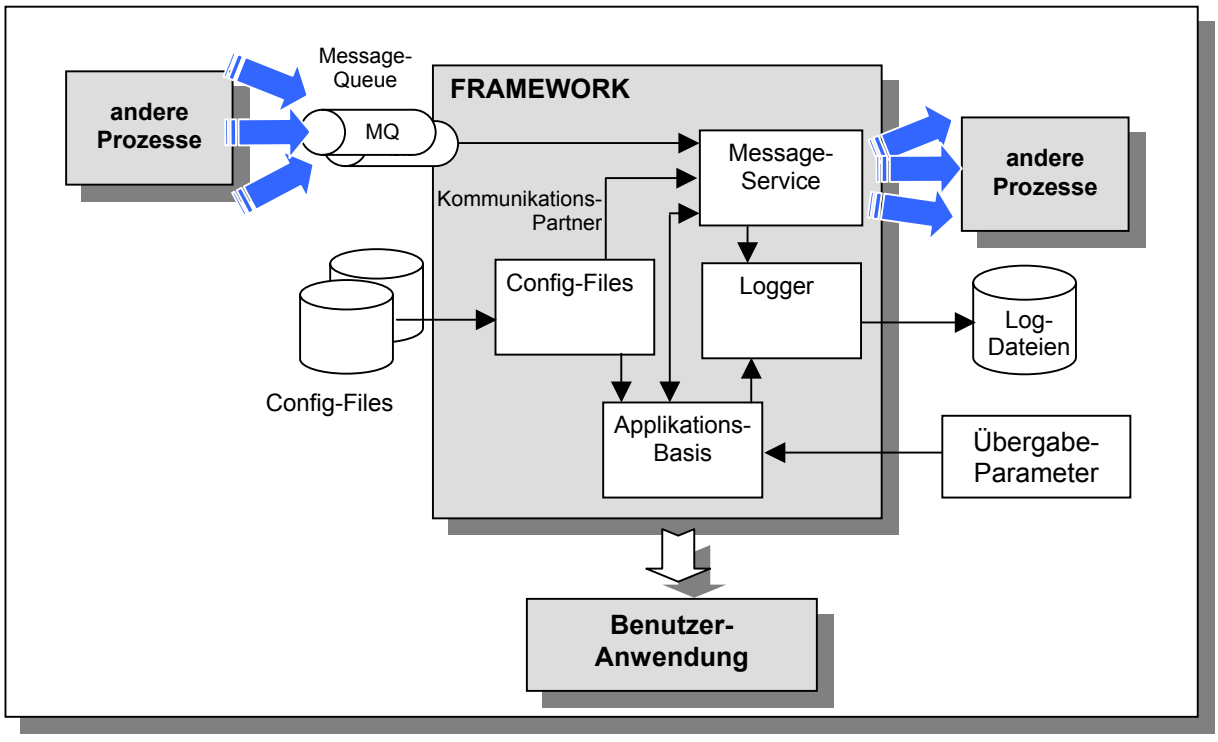


Abbildung 2.2: Die Framework-Komponenten

Die zu erfüllenden Aufgaben des Frameworks werden durch die in Abbildung 2.2 gezeigten Komponenten übernommen. Die Komponenten sind jeweils durch eine oder mehrere C++ - Klassen realisiert. Ein Überblick:

- **Message Service** (Klassen *CMsgService, CKomPartners, CTelegram,* in Library *libkomframe.a):* Bereitstellung eines Nachrichtendienstes (*Message Service)* zur Interprozesskommunikation über Linux Message Queues. Dazu Verwaltung einer Liste aller bekannten Kommunikationspartner.
- **Logger** (Klasse *CLogger,* in Library *liblogger.a*): Bereitstellung von Funktionalitäten für Stör- und Betriebsmeldungsausgabe in Dateien und auf die Konsole.
- **Config-Files** (Klasse *CConfigFile,* in Library *libconfigfile.a*): Einlesen, Auswerten und Bearbeiten von Konfigurationsdateien in einem XML-Format.
- **Applikations-Basis** (Klasse *CAppBase,* in Library *libkomframe.a*): Bereitstellen von Basis-Funktionalität wie Anlauf mit Einlesen der Übergabe-Parameter, Koordination der obigen Komponenten, sowie Bereitstellung von Laufzeit-Funktionen für den Framework-Benutzer.

Diese Komponenten und die zugehörigen Klassen werden nun einzeln beschrieben.

2.3 Der Message Service des Frameworks

2.3.1 Interprozess-Kommunikation mit Linux Message Queues

Message Queues sind ein Unix/Linux-Standardmechanismus zum prozess-übergreifenden Nachrichtenaustausch und als solcher z. B. in [VOGT01] beschrieben. Die nachfolgenden Ausführungen beschränken sich auf die zur Realisierung des Frameworks benutzten Möglichkeiten der Linux Message Queues. Ausführliche weiterführende Informationen enthalten insbesondere die zu den vorgestellten C-Befehlen gehörenden man-Pages des Linux-Systems.

Für die Arbeit mit Message Queues unter Linux kommen vier verschiedene Aufrufe des Linux-C-APIs zum Einsatz:

- ***int msgget(key_t key, int msgflg)***

  Message Queues werden über einen systemweit eindeutigen *Queue Key* identifiziert. Dieser Key ist eine nicht-negative Integerzahl und kann vom Benutzer vergeben werden. Kennt eine Applikation den Key der Queue einer anderen Applikation, kann sie an deren Queue eine Nachricht senden. `msgget()` liefert zu einem bekannten `key` den aktuellen *Identifier* der Queue zurück. Der *Identifier* ist gewissermassen das Betriebssystem-Handle der Queue und wird für die folgenden Befehle benötigt. Wird als Parameter für `msgflg` die Konstante `IPC_CREAT` übergeben, erzeugt Linux eine neue Queue zum angegeben Key. Alle auf dem Framework basierenden Anwendungen erzeugen auf diese Art beim Start eine eigene Message Queue, über die Nachrichten von anderen Prozessen empfangen werden können.

- ***int msgsnd(int msqid, struct msgbuf *msgp, size_t msgsz, int msgflg)***

  Sendet die Message an Speicheradresse *msgp* an die Queue zum Identifier *msqid*, welcher vorab mit *msgget()* ermittelt werden kann. Messages haben die allgemeine Struktur:

```
struct msgbuf {
      long mtype;     // message type, > 0
      char mtext[1];  // message data
};
```

  Die eigentlichen Nachrichten-Daten stehen im Feld *mtext*, welches nicht auf ein char-Feld der Länge eins beschränkt ist, sondern ein Bytefeld oder eine Struktur der tatsächlichen Größe *msgsz* darstellen kann. Der Nachrichtentyp *mtype* dient zur individuellen nachrichtentyp-spezifischen Behandlung beim Empfänger. Spezielle Flags im Parameter ***msgflg*** kommen im Framework nicht zum Einsatz. Die maximale Länge einer Message Queue-Nachricht ist bei SuSE Linux 8 standardmäßig auf 4096 Bytes beschränkt.

- ***ssize_t msgrcv(int msqid, struct msgbuf *msgp, size_t msgsz, long msgtyp, int msgflg)***

  Empfängt eine Message aus der durch den Identifier *msqid* identifizierten Queue und gibt die Länge der empfangenen Nachricht zurück. Die Nachricht wird an der durch *msgp* angegebenen Stelle im Speicher abgelegt, *msgsz* ist die Größe dieses Speicherbereichs und damit die maximal empfangbare Nachrichtenlänge. Im Normalfall blockiert der Aufruf, wenn keine Nachricht in der Queue ansteht. Wird für den Parameter *msgflg* die Konstante *IPC_NOWAIT* übergeben, kehrt der Aufruf unmittelbar zurück, wenn keine Nachricht zur Abholung ansteht. In einem solchen Fall enthält die System-Fehlervariable *errno* den Wert *ENOMSG*. Das Framework erlaubt sowohl blockierendes als auch nicht blockierendes Empfangen von Nachrichten. Nach dem Empfang einer Nachricht wird innerhalb des Frameworks anhand des Nachrichtentyps (aus der empfangenen *struct msgbuf)* in eine nachrichten-spezifische Empfangsroutine verzweigt („*Dispatching“).*

- ***int msgctl(int msqid, int cmd, struct msqid_ds *buf)***

  Der Aufruf ermöglicht verschiedene über den Parameter *cmd* auszuwählende Steuerungsfunktionen für die Queue mit dem Identifier *msqid.* Das Framework löscht bei Applikationsende die eigene Queue mit *cmd=IPC_RMID*. Außerdem kann mit diesem Aufruf z. B. die Anzahl der Nachrichten in einer Queue abgefragt werden.

Die Begriffe *Identifier* und *Key* können zu Verwechslungen führen. Um zu verdeutlichen, dass der *Identifier* ein temporär vom Betriebssystem bei der Queue-Erzeugung vergebener Integerwert ist, benutzt das Framework den Begriff *Queue-Handle.* Dieser wird auch in den folgenden Ausführungen vorgezogen.

2.3.1.1 Die Klasse *CMsgService*

Alle mit der Kommunikation über Message Queues in Verbindung stehende Funktionalitäten sind durch das Framework in der Klasse *CMsgService* gekapselt.

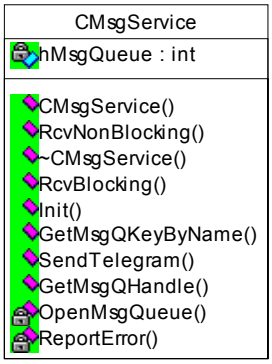


Abbildung 2.3: Die Klasse CMsgService

Ein Objekt dieser Klasse wird während des Framework-Starts angelegt und initialisiert. Dabei wird mit *OpenMsgQueue()* die eigene Queue erzeugt und das erhaltene Queue-Handle in der Membervariablen *hMsgQueue* abgelegt. Danach ist die Queue empfangsbereit für Nachrichten von anderen Prozessen. Gelöscht wird die Queue erst wieder im Destruktor *~CMsgService()*.

Nachrichten können mit den Methoden *ReceiveBlocking()* und *ReceiveNonBlocking()* sowohl blockierend als auch nicht blockierend empfangen werden.

Zum Versenden von Nachrichten steht die Methode *SendTelegram()* zur Verfügung.

Eventuell auftretende Fehler werden mit Hilfe der Methode *ReportError()* als Klartext in Logdateien ausgegeben.

Die weiteren Funktionen werden im Verlauf dieses Kapitels erläutert.

2.3.2 Adressierung von Kommunikationspartnern

Früh in der Designphase des Frameworks fiel die Entscheidung, Message Queues von Kommunikationspartnern nicht über ihre kryptische Queue-Key-Nummer, sondern über sprechende Klartext-Prozessnamen wie „SENDCP“ zu adressieren. Dies bietet einige Vorteile:

- Gesteigerte Lesbarkeit des Programmcodes
- Vermeidung von Uneindeutigkeiten durch doppelt vergebene Keys
- Verbesserte Nachvollziehbarkeit der Kommunikation insbesondere in der Test / Debug-Phase
- Schnelles Erkennen von fehlgeleiteten Nachrichten durch Mitversand von Sende- und Empfangsprozessnamen als Nachrichtenbestandteil

Diese Vorgehensweise erfordert eine Umsetzung der Klartext-Prozessnamen in die vom Linux-System intern benutzten Queue-Keys. Für eine solche Namensauflösung kamen zwei Vorgehensweisen zur Betrachtung:

Lösung 1: Zuordnung zur Compilierzeit über enum-Konstanten: Diese Art der Zuordnung sieht die Vergabe von symbolischen Konstanten-Namen über einen enum-Datentypen nach folgendem Muster vor:

```
enum QUEUE_KEYS
{
       SENDCP = 4000,      // Key zu Prozess SENDCP
       RECCP,              // Key zu Prozess RECCP
       ...
};
```

Der enum-Typ kann etwa in einer von allen Prozessen eingebundenen globalen Include-Datei definiert werden.

Diese Vorgehensweise ist einfach und schnell in der Programmausführung, da der Compiler bereits zur Übersetzungszeit die Prozessnamen in die zugeordneten Queue-Keys umsetzt und eine automatische Durchnummerierung ab dem angegeben Startwert (hier 4000) vornimmt.

Dem stehen allerdings zwei Nachteile gegenüber:

1. Eine Änderung des Enums, wie bei Änderung oder Hinzufügung eines neuen Eintrags, erfordert eine unbedingte Neuübersetzung *aller* einbindenden Prozesse. Wird dies übersehen, verursachen die entstehenden Inkonsistenzen fehlgeleitete Telegramme. Die Methode ist damit fehlerträchtig.

2. Durch die Umsetzung zur Compilierzeit steht der Klartext-Name zur Laufzeit nicht mehr zur Verfügung und kann entsprechend nicht zur Log-Ausgabe herangezogen werden. Damit sind obige Punkte der einfachen Nachvollziehbarkeit von Kommunikationsabläufen nicht erfüllt.

Im Framework ist daher eine andere Vorgehensweise realisiert:

Lösung 2: Die Zuordnung von Partnernamen zu Queue-Keys erfolgt über Einträge in einer globalen Konfigurationsdatei *globalconfig.ini*. Diese Datei enthält von allen Framework-Applikationen gemeinsam benutzte Konfigurationsdaten im XML-Format. In der Hauptsache ist das die Applikationsliste mit allen Kommunikationspartnern und den jeweils zugeordneten Queue-Keys:

```
// Message Queue Kommunikationspartner (Queue-ID>0!)
<MSGQUEUES>
  <APPLIST>
    <APPLICATION>
      <NAME>RECCP</NAME>          // Prozess RECCP
      <KEY>1</KEY>                // zug. Queue-Key
    </APPLICATION>
    <APPLICATION>
      <NAME>SENDCP</NAME>         // Prozess SENDCP
      <KEY>2</KEY>                // zug. Queue-Key
    </APPLICATION>
      ...                         // weitere Einträge
  </APPLIST>
  <KEY_OFFSET>4000</KEY_OFFSET>
</MSGQUEUES>
```

Abbildung 2.4: Die Applikationsliste in der Datei globalconfig.ini

Zusätzlich zu den vergebenen Queue-Keys enthält die Liste einen Key-Offset. Dieser wird framework-intern auf jeden vergebenen Key addiert. Dadurch ist ein gemeinsames Verschieben aller Keys innerhalb des erlaubten Wertebereichs möglich, was z. B. bei Kollisionen mit schon anderweitig im System benutzten Keys hilfreich ist.

Alle Prozesse lesen diese Liste bei ihrem Start ein und legen die Paare *(Prozessname, Queue-Key)* in einer internen Tabelle ab. Über diese Tabelle erfolgt nun die Namensauflösung zur Laufzeit des Programms.

2.3.2.1 Die Klasse *CKomPartners*

Zur Verwaltung der Kommunikationspartner-Tabelle wird der Klasse *CMsgService* die Klasse *CKomPartners* zur Seite gestellt:

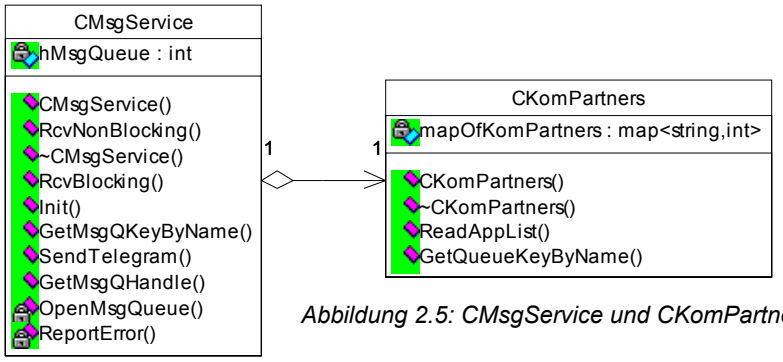


Abbildung 2.5: CMsgService und CKomPartners

Ein Objekt dieser Klasse liest in *CKomPartners::ReadAppList()* die Liste aller bekannten Kommunikationspartner aus der globalen Konfigurationsdatei und speichert die Paare *(Prozessname, Queue-Key)* zum schnellen Abruf in der privaten Tabelle *mapOfKomPartners.*

Im laufenden Betrieb benutzt *CMsgService* die Methode *CKomPartners::GetQueueKeyByName(),* um zu einem als String-Parameter übergebenen Empfangsprozess-Namen den Key der zugehörigen Message Queue zu ermitteln.

Die interne Tabelle *mapOfKomPartners* ist eine Instanz der Template-Klasse *map* aus der Standard Template Library (STL).

Die Klasse *map* realisiert einen Container für zweiwertige Elemente der Form *(Schlüssel, Wert).* Im vorliegenden Fall entspricht das *(Schlüssel, Wert)*-Paar dem Tupel *(Prozessname, Queue-Key)* mit den Datentypen *(string, int)*.

Das Auffinden von Elementen in einer Instanz von *map* über ihren Schlüssel erfordert den Zeitaufwand $O(\log n)$. Damit ist die Auflösung von Prozessnamen zu zugehörigen Queue-Keys während der Laufzeit effizient möglich. Weiteres zu *map* und der STL enthält [Stroustrup00].

2.3.3 Die Telegramm-Klasse *CTelegram*

Linux-intern werden Message-Queue-Nachrichten stets über den unter 2.3.1, *„Interprozess-Kommunikation mit Linux Message Queues“* vorgestellten Datentyp `struct msgbuf` übertragen. Das Handling dieser Struktur ist unkomfortabel und wenig flexibel. Eine dritte Klasse, *CTelegram*, vervollständigt daher die Message-Service-Komponente des Frameworks.

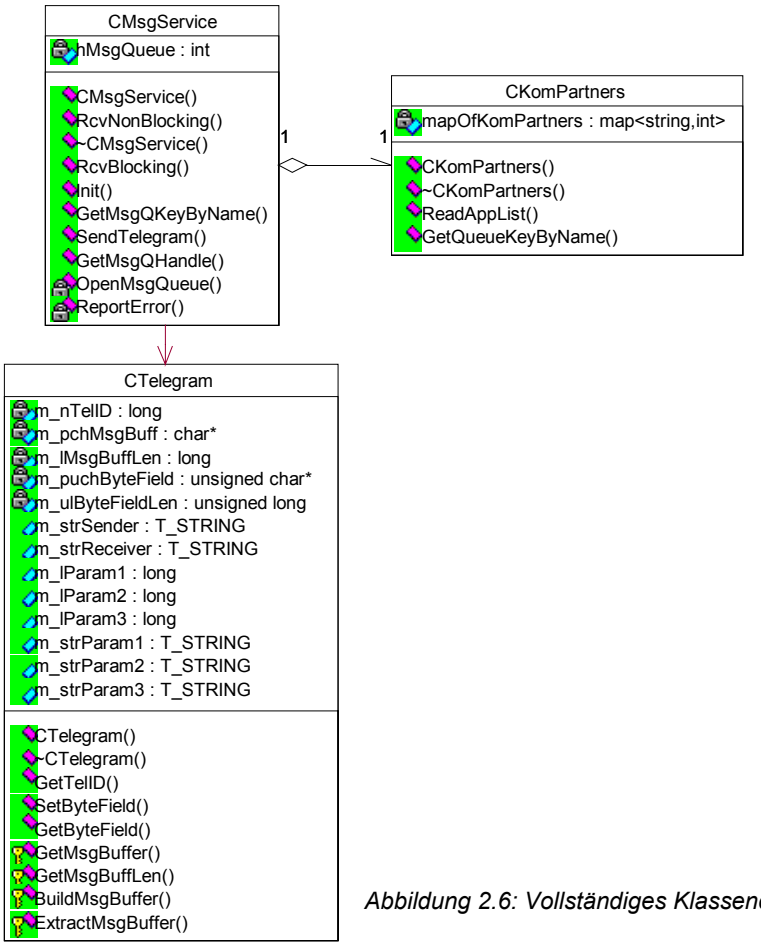


Abbildung 2.6: Vollständiges Klassendiagramm des Message Service

Objekte der Klasse *CTelegram* dienen als Container für beliebige Benutzerdaten, die von *CMsgService* über den Message-Queue-Mechanismus übertragen werden sollen.

Vor der Datenübertragung erzeugt der Benutzer ein eigenes Objekt der Klasse *CTelegram* und weist diesem die zu übertragenden Daten zu. Zu diesem Zweck stellt *CTelegram* mehrere Member-Variablen bereit:

- 3 *long*-Variablen: *m_lParam1, m_lParam2, m_lParam3*
- 3 *String*-Variablen: *m_strParam1, m_strParam2, m_strParam3*

Zusätzlich kann innerhalb eines *CTelegram*-Objekts ein beliebiges Byte-Datenfeld transportiert werden. Dazu dient die Methode *SetByteField()*, die als Parameter einen Pointer auf das Datenfeld sowie dessen Länge erwartet. Umgekehrt liefert *GetByteField()* Pointer und Länge zurück.

Bei der Objekterzeugung wird dem Konstruktor *CTelegram()* eine long-Konstante als Telegramm-ID zur eindeutigen Kennzeichnung des Nachrichten-Typs (siehe `struct msgbuf`) übergeben. Der Empfänger der Nachricht kann die ID mit *GetTelID()* abfragen und anhand dieser in eine nachrichtenspezifische Verarbeitungsroutine verzweigen.

Das so vorbereitete Telegramm wird durch Aufruf von *CMsgService::SendTelegram()* versendet. Ergänzend erwartet die Methode die Übergabe von Sende- und Empfangsprozessnamen. Beide werden zur Gewährleistung der Nachvollziehbarkeit des Telegrammverkehrs mit jedem *CTelegram*-Objekt versendet. Überschreitet die Gesamtlänge der zu transportierenden Daten die 4KB-Grenze von SuSE Linux 8.2, kehrt *SendTelegram()* mit Fehler zurück.

Vor der Übertragung des *CTelegram*-Objekts ist eine Konvertierung der enthaltenen Daten in ein linux-konformes Format erforderlich. Dies wird framework-intern mittels *CTelegram::BuildMsgBuffer()* erledigt. Ergebnis dieses Aufrufs ist ein interner *„Message Buffer“* des *CTelegram*-Objekts, in welchem die zu übertragenden Daten aufeinanderfolgend in der in Abbildung 2.7 dargestellten Form angeordnet werden:

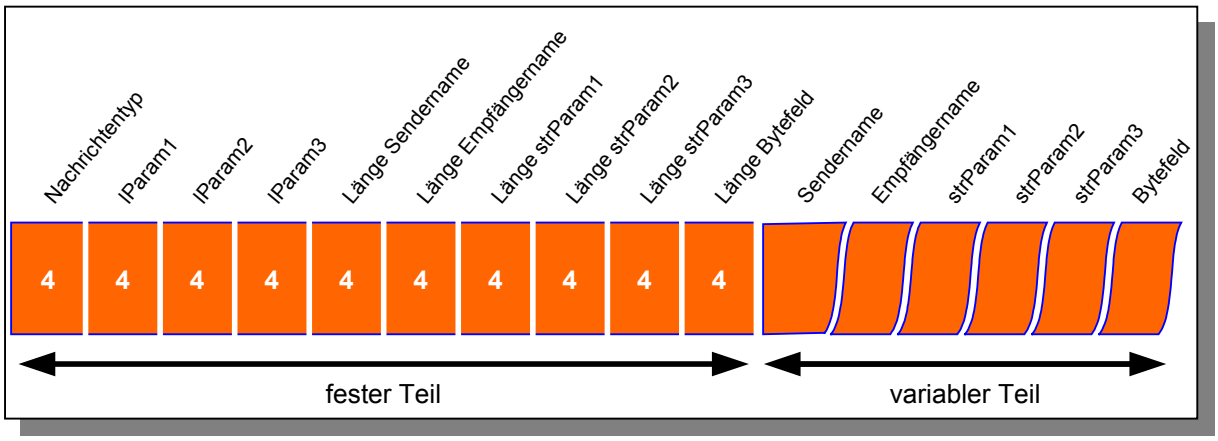


Abbildung 2.7: Messagebuffer-Format mit Feldlängen in Byte nach Erstellung mit BuildMsgBuffer()

Der erstellte Message Buffer wird in *SendTelegram()* per Typecast auf `(struct msgbuf*)` der Linux-*msgsnd()*-Funktion zum Versand übergeben.

Beim Nachrichtenempfang extrahiert analog die Methode *CTelegram::ExtractMsgBuffer()* die Daten aus dem Message Buffer in die Member-Variablen des *CTelegram*-Objekts. Der Aufrufer von *CMsgService::RcvBlocking()* bzw. *RcvNonBlocking()* erhält ein solcherart für die Weiterverarbeitung vorbereitetes *CTelegram*-Objekt zurück.

2.4 Weitere Framework-Klassen

Nachdem in den vorangegangenen Abschnitten die für die Kommunikation über Message Queues relevanten Klassen beschrieben wurden, werden diese nun in den Framework-Gesamtkontext eingeordnet und die verbleibenden Klassen vorgestellt. Alle Framework-Klassen und deren statische Zusammenhänge zeigt Abbildung 2.8.

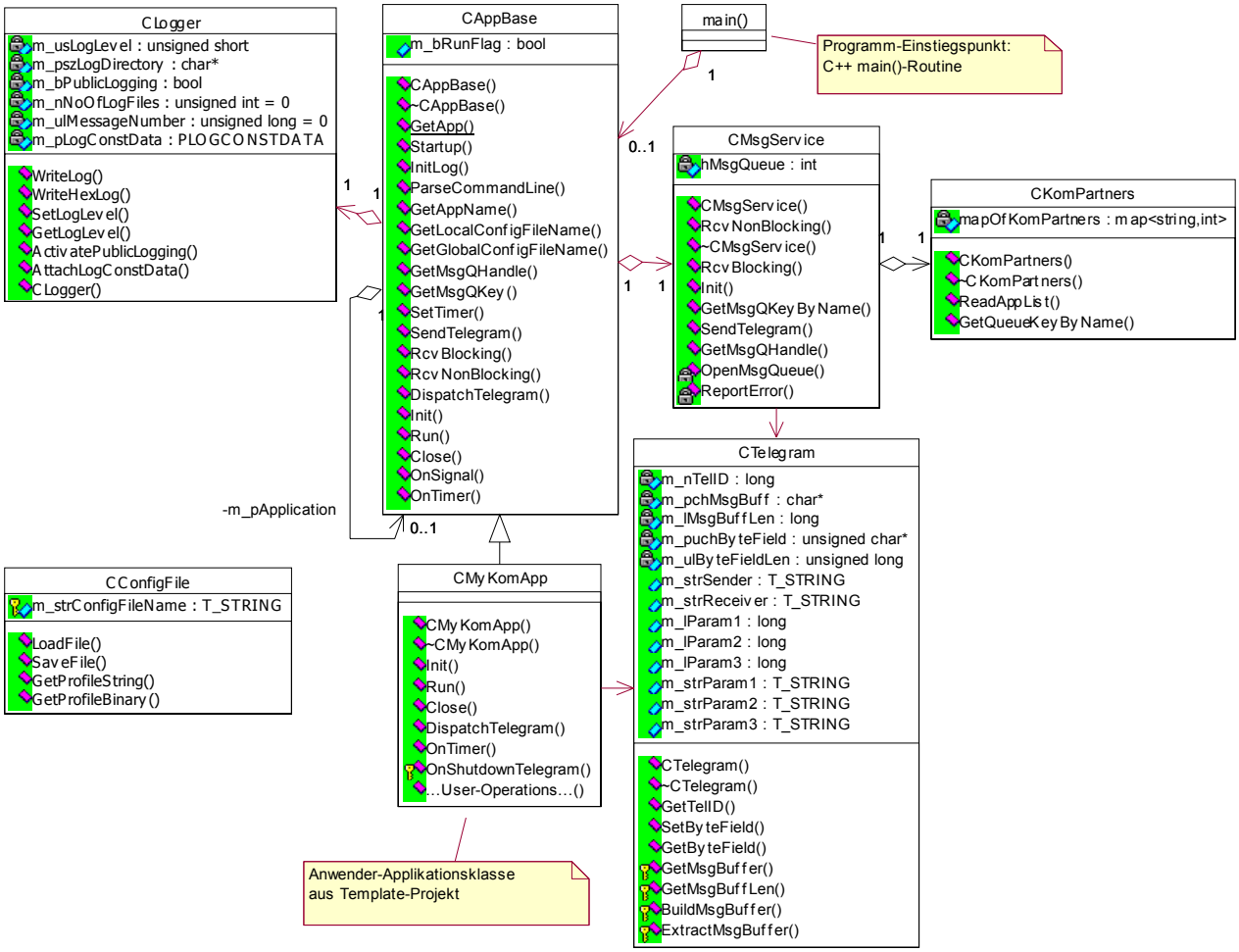


Abbildung 2.8: Vollständiges Klassendiagramm der Framework-Architektur

2.4.1 Die Klasse *CLogger*

Die Klasse *CLogger* kapselt die Funktionalitäten für Stör- und Betriebsmeldungs-Logging des Frameworks.

Die Meldungs-Ausgaben erfolgen mit per *SetLogLevel()* einstellbarer Ausführlichkeit in Dateien und optional auf eine Linux-Terminal-Shell. Letzteres kann mit *ActivatePublicLogging()* aktiviert werden.

Das Datei-Logging ist in Form von Tagesdateien (pro Tag eine eigene Log-Datei) oder in Form von Umlauf-Dateien mit Größenbegrenzung auswählbar.

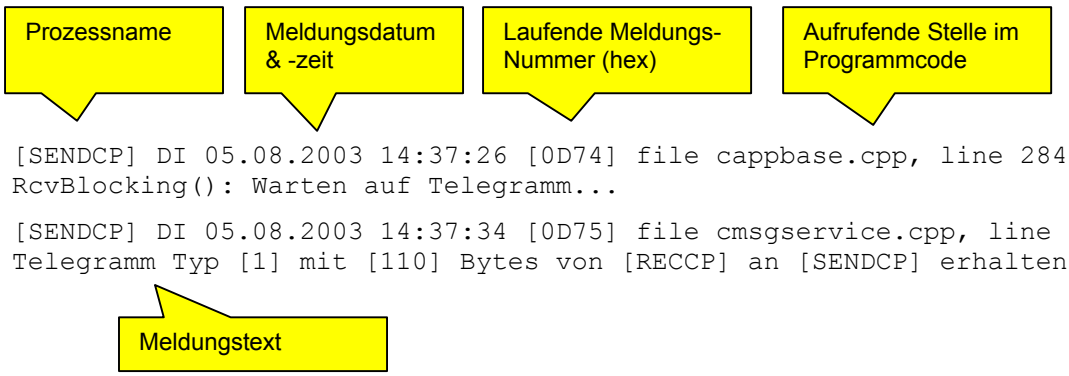
Standardmäßig werden jeweils eine Logdatei für Betriebsmeldungen (<dateiname.*log*>) und eine Datei für separate Fehlermeldungen (<dateiname.*err*>) angelegt. Dies kann bei Bedarf erweitert werden.

Bei der Option *Tages-Datei* wird an den eigentlichen Logdatei-Namen das Tages-Datum in der Form <*dateiname*>_*JJJJMMTT* angehangen.

Bei der Option *Umlauf-Datei* ist eine maximale Logdatei-Größe (in Byte) einstellbar. Erreicht eine Datei diese Größe, wird diese Datei nach <*dateiname*>_*old* umbenannt und das Logging in einer neuen Datei fortgesetzt. Durch diesen Ringpuffer-Mechanismus ist die maximale Log-Menge auf 2*Logdatei-Größe begrenzt.

Dateinamen und maximale Dateigröße werden *CLogger* mit der Methode *AttachLogConstData()* bekannt gegeben.

Mit der Methode *WriteLog()* gibt *CLogger* eine als Parameter zu übergebende Klartext-Meldung wie folgt formatiert in eine auswählbare Logdatei aus:



Ähnlich erzeugt *WriteHexLog()* einen Dump eines Byte-Feldes in Form von Hexadezimal- und ASCII-Darstellung. Dies ist beispielsweise nützlich zum Dump von per TP4 zu übertragenden Datenbuffern.

2.4.2 Die Klasse *CConfigFile*

CConfigFile stellt Methoden zum Einlesen, Auswerten, Bearbeiten und Abspeichern von Konfigurations-Dateien im XML-Format bereit.

Die Klasse wird als Hilfs-Klasse an verschiedenen Stellen innerhalb des Frameworks benutzt und kann auch vom Benutzer für eigene Zwecke instanziiert werden.

Als zusätzliches Feature bringt die Klasse *CConfigFile* die Klasse *T_STRING* mit sich, die Zeichenketten als Objekte behandelt und damit flexibles String-Handling bereitstellt, wie es z. B. aus der Windows-MFC-Klasse *CString* bekannt ist.

CConfigFile sowie die zugrunde liegenden Basisklassen sind Bestandteil des allgemein benutzten Software-Pools der Siemens IT PS Köln und werden daher hier nicht weiter dokumentiert. Eine kurze Vorstellung enthält [Weiden03].

2.4.3 Die Basisklasse *CAppBase*

2.4.3.1 Allgemeines

Die Klasse *CAppBase* ist die zentrale Klasse des Applikations-Frameworks. Neben der Initialisierung des Frameworks inklusive Auswertung der Kommandozeilen-Parameter instanziiert und verwaltet *CAppBase* Objekte der Klassen *CLogger* und *CMsgService.* Deren Funktionalität wird über Schnittstellenmethoden vor dem Framework-Benutzer verborgen, das heißt der Applikationsentwickler kommt zum Nachrichtenaustausch nur mit Methoden der Klasse *CAppBase* in Berührung.

Die Benutzer-Applikation wird in einer von *CAppBase* abgeleiteten Klasse eingebracht, welche in diesem Dokument mit *CMyKomApp* bezeichnet wird.

Durch die Ableitung von *CAppBase* stehen dem Applikations-Entwickler eine Anzahl von Methoden zur Verfügung, die teilweise durch den Applikations-Entwickler aufgerufen werden können und teilweise durch diesen zu implementieren sind. Diese Methoden werden nun vorgestellt.

2.4.3.2 Zu überschreibende Methoden von *CAppBase*

Jede Framework-Applikation durchläuft die Phasen Initialisierung, Ausführung und Beendigung. Zu diesen Phasen gehören die drei virtuellen *CAppBase*-Methoden *Init()*, *Run()* und *Close()*. Der Benutzer hat diese Methoden in seiner Anwendungsklasse *CMyKomApp* zu überschreiben und mit dem jeweils zugehörigen Anwendungscode zu füllen.

- **Init()**
  Init() wird in der Startphase des Frameworks aufgerufen, bevor der Message Service initialisiert wird. An dieser Stelle können demnach noch keine Message-Queue-Nachrichten übermittelt werden. Eigene Initialisierungen wie das Einlesen von Konfigurationsdateien etc. finden hier ihren Platz. Schlägt diese Initialisierung fehl, sollte *Init()* FALSE zurückliefern. Das Framework wird daraufhin mit einer Fehlermeldung beendet.
- **Run()**
  Run() wird aufgerufen, wenn das gesamte Framework erfolgreich initialisiert und die Benutzer-*Init()*-Methode mit TRUE beendet wurde. *Run()* ist die Hauptroutine für den Ablauf einer Framework-Anwendung und enthält die Hauptprogrammschleife. In dieser steht z. B. der Aufruf zum Empfang von Nachrichten über die Message Queue sowie weiterer, applikations-spezifischer Code. Die *Run()*-Routine wird während der gesamten Applikations-Ausführungsphase nicht mehr verlassen. Erst am Ende der Ausführungsphase wird mit der Rückkehr aus *Run()* die Beendigung des Frameworks eingeleitet.
- **Close()**
  Close() wird aufgerufen, nachdem *Run()* verlassen wurde. In *Close()* kann Code eingebracht werden, der zum Aufräumen und sauberen Beenden der Applikation benötigt wird.

Drei weitere Methoden vervollständigen die durch den Benutzer zu überschreibenden Methoden:

- **DispatchTelegram()**
  Sobald ein Telegramm über die Message Queue empfangen wurde, wird framework-intern *DispatchTelegram()* aufgerufen und ein *CTelegram*-Objekt mit den empfangenen Daten übergeben. Innerhalb von *DispatchTelegram()* sollte eine Auswertung des empfangenen Nachrichtentyps erfolgen und anhand dessen in eine telegramm-spezifische, benutzerdefinierte Telegramm-Verarbeitungsmethode verzweigt werden.

Die folgenden beiden Methoden sind optional und müssen nicht zwangsläufig durch den Benutzer überschrieben werden:

- **OnTimer()**
  Für periodisch auszuführende Programmteile bietet das Framework dem Benutzer einen Timer an. *OnTimer()* ist die Callback-Funktion, die bei jedem Ablauf dieses Timers aufgerufen wird. Das Zeitintervall der Aufrufe kann (z. B. in *Init()*) mittels *SetTimer()* festgelegt werden.
- **OnSignal()**
  OnSignal() ist der Signal-Handler für die Linux-Systemsignale *SIGINT*, *SIGTERM*, *SIGABRT* und *SIGALARM*, um auf diese Signale, z. B. nach einem kill-Befehl oder Druck auf STRG-C, definiert zu reagieren. Dazu implementiert das Framework in *CAppBase::OnSignal()* ein Default-Verhalten, welches im Normalfall zum Verlassen der *Run()*-Methode und der definierten Beendigung des Frameworks führt. Bei Bedarf kann der Benutzer diese Methode in *CMyKomApp* überschreiben und so ein eigenes Verhalten implementieren.

2.4.3.3 Aufrufbare Methoden

In diese Kategorie fallen Funktionen, die das Framework bereitstellt und die vom Applikations-Entwickler zu Initialisierungs- und Steuerzwecken aufgerufen werden können. Die wichtigsten drei sind die Schnittstelle zum Message Service des Frameworks:

- **SendTelegram()**
  Dient zum Versenden von Telegrammen an bekannte Kommunikationspartner. Der Methode wird ein Objekt der Klasse *CTelegram* sowie der Name des Empfängers übergeben.
- **RcvNonBlocking()**
  Alle in der Message Queue zum Empfang anstehenden Telegramme werden nacheinander ausgelesen und implizit an *DispatchTelegram()* zur Weiterverarbeitung übergeben. Steht kein Telegramm (mehr) an, kehrt *RcvNonBlocking()* ohne zu Blockieren zum Aufrufer zurück.
- **RcvBlocking()**
  Wie *RcvNonBlocking()*, allerdings blockiert der Prozess, solange kein Telegramm ansteht.

- **SetTimer()**
  Setzt das Timer-Intervall für periodische Aufrufe von *OnTimer()*. Das Zeitintervall kann durch zwei Übergabeparameter in Sekunden und Mikrosekunden eingestellt werden.
- **GetMsgQKey(), GetMsgQHandle()**
  Die beiden Methoden liefern Queue Key bzw. Betriebssystem-Handle (Identifier) der eingerichteten Message Queue zurück. Dies kann z. B. zu Debugzwecken nützlich sein.

Bevor diese Benutzer-Routinen aber zum Zuge kommen, muss eine Framework-Anwendung erst gestartet und mit Übergabeparametern versorgt werden.

2.4.3.4 Start des Frameworks, Übergabeparameter

Ergebnis der Anwendungsentwicklung auf Framework-Basis ist ein ausführbarer, eigenständiger Linux-Prozess. Alle diese Prozesse werden nach nachstehendem Muster gestartet:

```
<prozessname> -N APPLIKATIONSNAME -G globalconfigfile -C configfile -L logdirectory
```

Dabei sind:

- `-N APPLIKATIONSNAME`
  Name, über den der Framework-Message-Service den Prozess im System identifiziert. `APPLIKATIONSNAME` muss mit einem Eintrag in der Applikationsliste der globalen Konfigurationsdatei korrespondieren, wie in Abschnitt 2.3.2 *„Adressierung von Kommunikationspartnern“* beschrieben. Über diesen Namen können andere Framework-Anwendungen Messages an diesen Prozess senden.
- `-G globalconfigfile`
  Pfad und Name der globalen Konfigurationsdatei.
- `-C configfile`
  Pfad und Name der applikations-spezifischen Konfigurationsdatei.
  Manche Anwendungen benötigen neben den Daten aus der globalen Konfigurationsdatei weitere Daten, die nur für diese Anwendung relevant sind. Solche Daten können in einer anwendungs-eigenen („lokalen“) Konfigurationsdatei gespeichert werden. Der Parameter ist optional. Als Default wird `<APPLIKATIONSNAME>.ini` angenommen.
- `-L logdirectory`
  Verzeichnis, in welchem die Logdateien abgelegt werden.

Die mit den erstgenannten drei Parametern übergebenen Werte können durch den Framework-Benutzer über die *CAppBase*-Methoden *GetAppName()*, *GetGlobalConfigFileName()* und *GetLocalConfigFileName()* abgefragt werden.

Ein Zugriff auf die Inhalte der Konfigurationsdateien ist unter Verwendung der Klasse *CConfigFile* möglich.

2.4.3.5 Ablauf der Startphase

Einsprungpunkt für den Start jeder C / C++ - Anwendung ist die *main()*-Routine. Diese Routine ist Bestandteil der vorkompilierten Framework-Programmbibliothek und so durch den Framework-Benutzer nicht veränderbar.

Innerhalb der *main()*-Routine wird zunächst ein Signal-Handler zum Abfangen der Systemsignale *SIGINT*, *SIGTERM*, *SIGABRT* und *SIGALARM* installiert und auf *CAppBase::OnSignal()* gerichtet. Daraufhin wird bereits in der Anlaufphase definiert auf diese Signale reagiert.

Nun beginnt *main()* mit der Initialisierung des Frameworks. Dies erfolgt in mehreren Schritten:

1. Übergabeparameter auswerten durch Aufruf von *CAppBase::ParseCommandLine()*. Dieser Methode werden die bekannten *argc*- und *argv[]*-Argumente der *main()*-Routine als Parameter übergeben.
2. Logging initialisieren über *CAppBase::InitLog()*. Erfolg oder Misserfolg aller nachfolgenden Aktionen kann danach in Logdateien protokolliert werden.
3. Initialisierung des Benutzer-Teils der Applikation durch Aufruf von *Init()*. Kehrt dieser Aufruf mit TRUE zurück, wird mit Schritt 4 fortgesetzt, ansonsten wird die Anwendung hier beendet.
4. Übrige Framework-Initialisierung, insbesondere des Message Service, über *CAppBase::Startup()*.

Mit erfolgreicher Beendigung des Schrittes 4 ist das Framework betriebsbereit und für andere Applikationen über die Message Queue erreichbar. Durch den Aufruf von *Run()* beginnt nun die Ausführungsphase der Applikation. Vorab wurde die *CAppBase*-Variable *m_bRunFlag* auf TRUE gesetzt. Über diese Variable kann ein bevorstehendes Beenden des Frameworks signalisiert werden, woraufhin der Framework-Benutzer seine *Run()*-Routine zu beenden hat.

Erst wenn der Benutzer die *Run()*-Routine mit TRUE verlässt und die Kontrolle damit an *main()* zurückgibt, wird durch den Aufruf von *Close()* die Beendigung des Frameworks eingeleitet. Ein Verlassen von *Run()* mit FALSE führt zur sofortigen Beendigung ohne Aufruf von *Close()*. Dies kann erforderlich sein, wenn nach einem Fehler ein sauberes Ausführen der *Close()*-Routine nicht mehr gewährleistet ist.

Alle vom Framework selbst belegten Ressourcen werden bei Programmende durch die Destruktoren der jeweiligen Klassen wieder freigegeben.

2.4.4 Die Benutzer-Klasse *CMyKomApp*

Der Anwender des Frameworks erhält die oben beschriebenen Klassen vorkompiliert in den Libraries *libkomframe.a, liblogger.a* und *libconfigfile.a*. Diese bindet er nebst den zugehörigen Header-Dateien in eigene Software-Projekte ein. Für eine eigene Applikation leitet er eine Klasse von *CAppBase* ab und bringt den Applikations-Code darin ein.

Diese abgeleitete Applikationsklasse wird in diesem Dokument mit *CMyKomApp* bezeichnet. Innerhalb von *CMyKomApp* sind die vier von *CAppBase* geerbten Methoden *Init()*, *Run()*, *Close()* und *DispatchTelegram()* wie beschrieben zu überschreiben. Als Muster kann die Datei *cmykomapp.cpp* bzw. *cmykomapp.h* aus dem bereitgestellten Template-Projekt (ab nächster Seite) benutzt werden. Darin enthalten ist ein ablauffähiger Programm-Rahmen mit Beispielen für das Versenden und Empfangen von Nachrichten.

Durch die Ableitung erbt *CMyKomApp* alle in *CAppBase* definierten Funktionalitäten und Abläufe. Zusammen mit der für den Benutzer verborgenen *main()*-Routine wird so für alle Framework-Anwendungen ein einheitlicher grundsätzlicher Aufbau und Ablauf erreicht.

2.4.4.1 Anlegen des Applikations-Objektes

Zu Recht sind globale Variablen in einem objektorientierten Programm verpönt. Da aber die Benutzer-Klasse *CMyKomApp* zur Erstellungszeit der Framework-Bibliotheken noch nicht bekannt ist, kann das Framework selbst auch kein entsprechendes Objekt anlegen.

Dies bleibt also dem Benutzer überlassen, der dazu in der Implementationsdatei zu seiner Klasse *CMyKomApp* eine globale Variable nach folgendem Muster anzulegen hat:

```
CMyKomApp theApp;
```

Auf diese Art wird bereits beim Laden des Prozesses ein Objekt der Klasse *CMyKomApp* angelegt. Intern wird dabei automatisch auch der Konstruktor der Vaterklasse, *CAppBase::CAppBase()*, ausgeführt. Innerhalb dieses Konstruktors wird durch die Zeile

```
CAppBase::m_pApplication=this;
```

der statischen *CAppBase*-Membervariablen `m_pApplication` ein Pointer auf das gerade erzeugte Applikationsobjekt zugewiesen. Die Klasse *CAppBase* verwaltet also einen Verweis auf ihr einziges zugelassenes Anwendungsobjekt.

Die statische Klassenmethode *CAppBase::GetApp()* liefert diesen Pointer als Ergebnis zurück. Über diesen Weg kann jederzeit auf das Applikationsobjekt zugegriffen werden, was z. B. in der *main()*-Routine zum Aufruf von *Init(), Run()* und *Close()* ausgenutzt wird.

2.4.4.2 Eine Beispiel-Anwendung

Benutzung und Ablauf der Klasse *CMyKomApp* demonstriert das folgende Beispiel-Listing, welches im Kern der Datei *cmykomapp.cpp* aus dem Template-Projekt entspricht. Am Beispiel der Telegrammtypen BOTSCHAFTSDIENSTTEL und SHUTDOWNTEL wird die Reaktion auf erhaltene Telegramme sowie das Versenden eines Telegramms an den Prozess *KOMABW* gezeigt.

```
/**  cmykomapp.cpp
*
*    begin                : Mit Sep 18 2003
*    copyright            : (C) 2003 by Ralf Weiden
*    email                : ralf.weiden@siemens.com
*    @author ralf Weiden
*
* Demo-Applikation auf Basis von CAppBase. Demonstriert Benutzung der Framework-
* Funktionalität.
*****************************************************************************/

/* ÜBERGABEPARAMETER:
-G: Pfad+Name des globalen Config-Files
-N: Name der Applikation, wie in globalem Config angegeben
-C: Name des lokalen Configfiles, falls vorhanden
-L: Pfad für Log-Dateien
*/

// FRAMEWORK-INCLUDES
#include "../../KLASSEN/global.h"   // Globale Includes für Telegrammtypen etc.
#include "local.h"                  // Eigene Includes für Loglevel etc.

// Include der eigenen Header-Datei
#include "cmykomapp.h"

// Anlegen des Applikations-Objektes
CMyKomApp theApp;

// Konstruktor / Destruktor
CMyKomApp::CMyKomApp()
{
  // Hier eigenen Konstruktor-Code einfügen...
}

CMyKomApp::~CMyKomApp()
{
  // Hier eigenen Destruktor-Code einfügen...
}

/** OnTimer(): Timer-Callback
*
* Wird aufgerufen, wenn Framework-interner Timer abgelaufen ist.
* Setzen des Timers mit SetTimer (seconds, microseconds)
*****************************************************************************/
bool CMyKomApp::OnTimer()
{
  // Als Beispiel für eine Timer-Routine erfolgt nur eine Log-Ausgabe
  this->Log->WriteLog (LOG,   FILE  ,   LINE  , LOG ALWAYS, "OnTimer()");

  return true;
}

/** Init(): Aufruf beim (Re-)Start der Applikation. Initialisiert Logging und Weiteres...
*
* @return true on success, else false
*****************************************************************************/
bool CMyKomApp::Init()
{
  // Init des Loggings mit Filenamen und max. Größe. Eigene Einträge nach Bedarf ergänzen
  static LOGCONSTDATA LogConstData[]=
  {
    {"MyKomApp.log", 10000},        // Standard-Logfile, korrespondiert mit Konstante LOG
    {"MyKomApp.err", 10000},        // Standard-Errfile, korrespondiert mit Konstante ERR
  };
  this->Log->AttachLogConstData (LogConstData, sizeof (LogConstData)/sizeof(LOGCONSTDATA),
                                 INIT LOGLEVEL);
  Log->ActivatePublicLogging(true);         // Log-Ausgaben auch auf Konsole

  //  <ToDo> Eigene Initialisierung hier einfügen...
  SetTimer (1, 0);    // Timer starten (Sekunden, Mikrosekunden)

  Log->WriteLog (LOG,   FILE  ,   LINE  , LOG ALWAYS,
                 "Init von [%s] beendet", this->GetAppName());
  return true;
}
```

```
/** Run(): Applikations-Hauptschleife für den Hauptprogrammcode, ersetzt main().
* Der eigentliche User-Programmcode wird hier eingebracht.
*
* Wird nach erfolgreichem Abarbeiten von Init() aufgerufen.
* Beim Verlassen mit true wird Close() aufgerufen und die Applikation beendet.
* Beim Verlassen mit false wird die Applikation sofort beendet
*
* @return true on success, else false
*****************************************************************************************/
bool CMyKomApp::Run()
{
  Log->WriteLog (LOG,   FILE  ,   LINE  , LOG ALWAYS, "Run(): Applikation läuft");

  // Haupt-Applikationsschleife; ausführen solange m_bRunFlag==TRUE
  while (this->m_bRunFlag)
  {
    // Beispiel: Blockierendes Empfangen eines Telegramms
    // Telegramm-Auswertung erfolgt durch impliziten Aufruf von DispatchTelegram(), s.u.
    if (!this->RcvBlocking())
    {
      Log->WriteLog (ERR,   FILE  ,   LINE  , LOG ALWAYS, "RcvBlocking() schiefgegangen!");
      this->m bRunFlag=false; // FEHLER: Framework beenden
    }

    // Beispiel: Versenden eines Botschaftsdienst-Telegramms
    CTelegram Telegram (BOTSCHAFTSDIENSTTEL);     // Telegrammobjekt anlegen
    Telegram.m lParam1   = 4711;                  // Einfach mal den Wert '4711' zuweisen
    Telegram.m strParam1 = "Hallo Welt!";         // Und einen netten Text dazu...
    this->SendTelegram ("KOMABW", Telegram);      // Und ab damit an Prozess KOMABW
  } //while (this->m bRunFlag);                   // Wiederholen bis !m bRunFlag

  Log->WriteLog (LOG, __FILE__, __LINE__, LOG_ALWAYS,
                 "Verlassen von Run(): Applikation beenden...");
  return true;
}

/** Close(): Zum Aufräumen der Applikation. Wird vom Framework nach Verlassen von Run()
aufgerufen.
*
* @return true on success, else false
*****************************************************************************************/
bool CMyKomApp::Close()
{
  // <ToDO> Hier eigenen Aufräum-Code einbauen...
  Log->WriteLog (LOG,   FILE  ,   LINE  , LOG ALWAYS, "Close(): Applikation beendet.");
  return true;
}

/** DispatchTelegram(): Routine für Telegram-Dispatching.
* Wird bei Eingang jedes MsgQueue-Telegramms implizit durch RcvBlocking() oder
* RcvNonBlocking()aufgerufen.
* Hier sollte ein Dispatching und Verzweigung in telegramm-spezifische Routinen erfolgen
*
* @param Telegram CTelegram-Objekt mit den empfangenen Daten
* @return true wenn OK, sonst false
*****************************************************************************************/
bool CMyKomApp::DispatchTelegram(CTelegram& Telegram)
{
  bool bResult;

  Log->WriteLog (LOG, __FILE__, __LINE__, LOG_TELHANDLING,
                 "Telegramm Typ [%d] von [%s] erhalten",
                 Telegram.GetTelID(), Telegram.m strSender.GetText());

  // Dispatching des Telegramms
  switch (Telegram.GetTelID())
  {
    case BOTSCHAFTSDIENSTTEL:
      bResult=this->OnBotschaftsdienstTelegram(Telegram);
      break;

    case SHUTDOWNTEL:
      bResult=this->OnShutdownTelegram(Telegram);
      break;
```

```
    /* <ToDo>: Eigene Telegramme hier einfügen
    // Telegramm-ID-Konstanten sind in Datei global.h definiert
    case MEINERSTESEIGENESTELEGRAMM:
      this->...
      break;
    </ToDo> */
    default:
    {
      this->Log->WriteLog (ERR, __FILE__, __LINE__, LOG_TELHANDLING,
                           "Telegramm Typ [%d] unbekannt!",
                           Telegram.GetTelID(), Telegram.m strSender.GetText());
      bResult=false;      // Dies führt zum Beenden der Applikation
    }
  } // switch...
  return bResult;
}

/** Wird aufgerufen von DispatchTelegram(), wenn ein Botschaftsdiensttelegramm empfangen
* wurde. Ziel: Simulation des Botschaftsdienstes aus dem bestehenden Altsystem
*
* @param Telegram CTelegram-Objekt mit den empfangenen Daten
* @return true on success, else false
*********************************************************************************/
bool CMyKomApp::OnBotschaftsdienstTelegram (CTelegram& Telegram)
{
   this->Log->WriteLog (LOG, __FILE__, __LINE__, LOG_USERTELHANDLING,
                        "Botschaftsdienst-Telegramm von [%s] wird ausgewertet...",
                        Telegram.m strSender.GetText());

  // <ToDo> Hier erfolgt die Auswertung des empfangenen Telegramms...
  return true;
}

/** Wird aufgerufen, wenn ein Shutdown-Telegramm empfangen wurde.
* Setzt m bRunFlag auf false; damit wird Run()-Routine beendet
*
* @param Telegram CTelegram-Objekt mit den empfangenen Daten
* @return always true
*********************************************************************************/
bool CMyKomApp::OnShutdownTelegram (CTelegram& Telegram)
{
  this->Log->WriteLog (LOG,   FILE  ,   LINE  , LOG ALWAYS,
                       "SHUTDOWN-Telegramm von [%s] erhalten!",
                       Telegram.m_strSender.GetText());

  this->m bRunFlag=false;    // Damit wird die Hauptapplikationsschleife in Run() verlassen
  return true;
}
```

Listing 2.1: Beispiel für eine Framework-Anwendung mit der Klasse CMyKomApp

3 Das ISO/OSI-TP4-Protokoll

3.1 Einführung

Unter Zoologen galt der Quastenflosser lange Zeit als ausgestorben. In Fachbüchern fanden sich über diese Spezies nur vage Zeichnungen und Vermutungen. Dies änderte sich in der ersten Hälfte des 20. Jahrhunderts, nachdem Fischern in den Tiefen des indischen Ozeans ein lebendes Exemplar ins Netz ging.

Ähnlich verhält es sich mit dem vorgefundenen ISO/OSI-TP4-Transportprotokoll. In den 1980er Jahren als konkrete Implementierung eines Protokolls zu OSI-Schicht 4 als Teilmenge von ISO 8073 spezifiziert, kann es heute als der Dinosaurier der Transportprotokolle gelten, der den durch den Konkurrenten TCP ausgeübten Evolutionsdruck nicht überstand. Nach dem Siegeszug des weltweiten Internets mit seiner TCP/IP-Protokollfamilie sind die OSI-Protokolle inzwischen nahezu bedeutungslos.

Entsprechend schwer gestaltet sich das Auffinden von hilfreicher und lesbarer Dokumentation zu diesem Protokoll. Am aussichtsreichsten sind alte Auflagen bewährter Bücher zu Netzwerktechnologien, wie [Tanenbaum92]. Einen Überblick enthält außerdem [Proebster02].

Varianten des OSI-Protokolls fanden seinerzeit neben dem Einsatz in öffentlichen Netzen auch einige Verbreitung in der Automatisierungstechnik. So basiert die Siemens-Spezifikation SINEC H1 auf diesem Protokoll. Mit [Siemens88] hatten alteingesessene Kollegen glücklicherweise noch Referenzmaterial in Form alter Schulungsunterlagen verfügbar. Anhand dieser Unterlagen wurden die Protokollanalysen der vorgefundenen AEG-Variante des TP4-Protokolls durchgeführt.

Wenig sinnreich erscheint heutzutage ein tiefer gehendes Beschreiben der Hintergründe des TP4-Protokolls. Die nachfolgenden Kapitel dieser Masterarbeit beschränken sich daher auf das für das Verständnis des zugrunde liegenden Projektes erforderliche Maß. Hauptaugenmerk liegt auf der Implementation eines Netzwerk-Protokolls mit objektorientierten Techniken. Die praktischen Herausforderungen bei einer solchen Aufgabe sind weitgehend unabhängig vom konkret implementierten Protokoll und damit auch heute noch uneingeschränkt aktuell.

3.2 Aufgaben von Transportprotokollen

Allgemeine Aufgabe eines Transportprotokolls gemäß OSI-Schicht 4 ist die zuverlässige, bidirektionale End-to-End-Datenübertragung in einem Netzwerk. Dazu unterteilen die Transportprotokoll-Instanzen zweier entfernter Rechner die zu übertragenden Daten in sogenannte Transportdateneinheiten *(Transport Protocol Data Unit*, TPDU).

Gesendete TPDUs sollen fehlerfrei und in der richtigen Reihenfolge beim Empfänger eintreffen. Außerdem sollen Topologie und Technologie der unterlagerten Netzwerk-Schichten vor dem Transportschicht-Dienstbenutzer verborgen werden. Die Transportschicht benötigt also robuste Mechanismen zur Fehlerbeseitigung und zur flexiblen Anpassung an die Gegebenheiten der vorgefundenen Infrastruktur.

Transportprotokolle, darunter TP4, arbeiten oft verbindungsorientiert: Vor und nach der Übertragung der Nutzdaten werden zusätzliche TPDUs zum Auf- bzw. Abbau einer Verbindung ausgetauscht und die nötigen Ressourcen für den zügigen Datentransport reserviert. Außerdem können dabei z.B. Dienstgüte-Parameter ausgehandelt werden.

Auftretende Probleme in einem typischen Netz nach Abbildung 3.1 sind vor allem durch Störungen verloren gegangene TPDUs sowie der Empfang von TPDUs in verdrehter Reihenfolge, wenn die Netztopologie ein „Überholen“ über Netzwerkpfade mit unterschiedlichen Laufzeiten erlaubt.

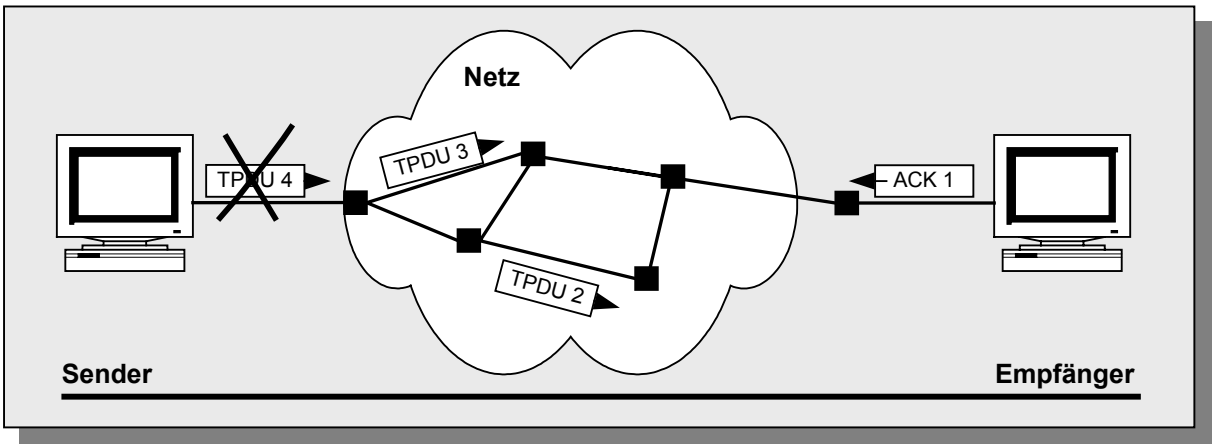


Abbildung 3.1: Ende-zu-Ende-Kommunikation über ein typisches Netzwerk

Als Mittel gegen diese Widrigkeiten kommen drei Basismechanismen zum Einsatz:

1. **Empfangsbestätigungen (Data Acknowledges, ACK)**
   Der Empfänger bestätigt dem Sender jede korrekt empfangene TPDU über ein Acknowledge. Dies ist bei TP4 eine gesonderte TPDU. Implizites Versenden von ACKs zusammen mit Nutzdaten *(piggibacking)*, wie es TCP kennt, ist nicht vorgesehen.

2. **Timer**
   Sowohl Nutzdaten-TPDUs als auch ACKs können auf der Übertragungsstrecke verloren gehen. In beiden Fällen erhält der Sender keine Rückbestätigung vom Empfänger. Damit der Sender nicht unbegrenzt auf das Eintreffen eines ACKs wartet, wird pro gesendeter TPDU ein gesonderter *Retransmission-Timer* gestartet. Läuft ein solcher vor dem Eintreffen des Acknowledge ab, wird (mindestens) die zugehörige Daten-TPDU erneut gesendet.

3. **Sequenznummern (TPDU-Nummern)**
   Für die Zuordnung eines ACKs zur damit quittierten Daten-TPDU erhält jede TPDU eine eindeutige Sequenznummer. Über diese Sequenznummer erkennt der Sender beim Eintreffen eines ACKs die verursachende TPDU als bestätigt und löscht zugehörige Ressourcen, darunter den Timer. Der Empfänger nutzt die Sequenznummern zur Feststellung der korrekten Reihenfolge eintreffender TPDUs. Außerdem erkennt er daran doppelt empfangene TPDUs. Dieser Fall tritt auf, wenn eine TPDU korrekt empfangen wurde, das ACK dazu aber verloren ging. Der Sender wiederholt die TPDU in einem solchen Fall nach Ablauf des Retransmission Timers unnötigerweise. Das Duplikat muss durch den Empfänger verworfen werden.

3.3 Flusskontrolle, Sendefenster und go-back-n

Hätte der Sender nach jeder gesendeten TPDU zunächst auf das zugehörige Acknowledge zu warten, wäre ein solches Protokoll recht ineffizient, denn die Wartezeit beträgt, bedingt durch Hin- und Rückweg, mindestens das Doppelte der TPDU-Laufzeit des Netzes. Ferner erzeugt jede ACK-TPDU zusätzlichen Overhead, der Netzwerk-Kapazitäten und Rechenzeit beansprucht, ohne Nutzdaten zu übertragen. Unabhängig davon muss durch eine Flusskontrolle vermieden werden, dass der Empfänger mit Daten „überflutet“ wird, wenn zwischenzeitlich die Ressourcen knapp werden.

TP4 verwendet dazu das in Abbildung 3.2 demonstrierte Kreditschema. Beide Stationen gewähren sich beim Verbindungsaufbau gegenseitig und unabhängig einen Sendekredit. Eine Station darf gemäß des zugeteilten Kredits mehrere TPDUs in Folge senden. Erst wenn der Kredit aufgebraucht ist, wartet sie auf ein Acknowledge.

Da im Netz verloren gegangene Telegramme wiederholt werden müssen, puffert ein Sender alle gesendeten, aber noch nicht bestätigten TPDUs mit aufsteigenden Sequenznummern in einer Liste. Mit ihrer dem Sendekredit entsprechenden Maximallänge bildet diese Liste das *Sendefenster.*

Der Empfänger zählt die erhaltenen TPDUs und schickt nach Verbrauch des zugeteilten Sendekredits ein Acknowledge. Mit diesem ACK teilt er dem Sender die nächste erwartete TPDU-Nummer mit und bestätigt alle TPDUs mit kleinerer Nummer. Gleichzeitig wird ein neuer Kredit zugeteilt, der sich vom vorherigen unterscheiden und auch Null sein kann, falls der Empfänger vorübergehend nicht empfangsbereit ist.

Der Sender entfernt nach Erhalt des ACKs alle bestätigten Telegramme aus seiner Sendefenster-Liste und schiebt eventuell anstehende weitere TPDUs entsprechend des neu zugeteilten Kredits nach.

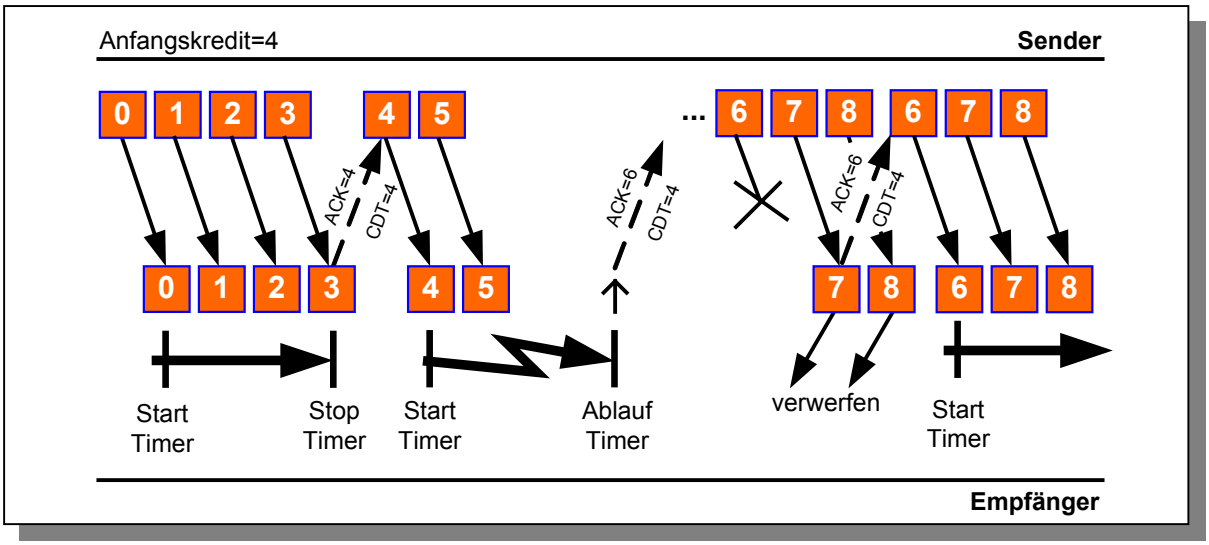


Abbildung 3.2: Flusskontrolle mit Kreditvergabe und go-back-n

Möglicherweise brauchen diese weiteren TPDUs (Nr. 4 und 5 in Abbildung 3.2) den Sendekredit nicht vollständig auf. Trotzdem muss der Empfänger sie rechtzeitig mit einem ACK bestätigen. Da er sich in diesem Fall nicht auf das Kriterium „Sender hat Kredit aufgebraucht“ stützen kann, zieht er einen zweiten Mechanismus heran: Wird nach dem Verbindungsaufbau oder nach dem Versand eines ACK die erste gültige TPDU empfangen, startet der Empfänger mit diesem Ereignis einen *Acknowledge-Timer*.

Brauchen weitere TPDUs den Sendekredit innerhalb der Timeout-Zeit auf, schickt der Empfänger wie oben beschrieben ohne Verzögerung ein ACK und löscht den Timer vor dessen Ablauf. Ein ACK wird nun auch bei Ablauf des Acknowledge Timers gesendet, wenn weniger TPDUs eintrafen.

Erhält der Sender innerhalb des Retransmission-Timer-Intervalls der ersten in seiner Sendefensterliste wartenden TPDU kein ACK, sendet er die gesamte Sendeliste erneut und dekrementiert einen Wiederholungszähler. Erreicht dieser Zähler Null, werden die Sendeversuche aufgegeben und die Verbindung gelöscht. Der Acknowledge-Timer des Empfängers muss also so kurz eingestellt sein, dass ACKs rechtzeitig vor Ablauf des Retransmission-Timers beim Sender eintreffen. Dazu ist die Laufzeit auf dem Netz zu berücksichtigen.

Der Empfänger erwartet die TPDUs mit Sequenznummern in aufsteigender Reihenfolge. Werden TPDUs auf dem Weg durch das Netz in ihrer Reihenfolge vertauscht oder geht eine TPDU durch eine Störung verloren (Nr. 6 in Abbildung 3.2), arbeitet TP4 mit der *go-back-n*-Strategie: Der Empfänger verwirft alle TPDUs mit unerwarteter Sequenznummer und fordert mittels ACK die erwartete TPDU erneut an. Der Sender muss daraufhin ab dieser angeforderten TPDU seine Sendeliste wiederholen.

Das geschilderte Kreditschema nimmt die TPDU-Laufzeit auf dem Netz als vernachlässigbar an. Die Laufzeit des Netzes kann sich jedoch insbesondere bei großen Entfernungen zwischen Sender und Empfänger, wie bei einer Sattelitenstrecke, negativ auswirken, da zwischen dem Versenden der letzten TPDU und dem Eintreffen des zugehörigen ACKs mindestens die doppelte Laufzeit vergeht. Während dieser Zeit ist der Sender blockiert.

Natürlich bereitet dies ein weites Feld für Optimierungsalgorithmen. So könnte ein ACK bereits vor vollständigem Aufbrauch des Kredits gesendet werden, bei einem Kredit von 8 also z. B. bereits nach jeder vierten TPDU. Dem Sender würde so bereits neuer Kredit eingeräumt, bevor er blockieren muss. Damit nähert sich das Verfahren den sogenannten *Schiebefenster-Protokollen mit Pipelining*. Das TCP-Protokoll geht noch weiter und versucht mit Hilfe statistischer Methoden eine dynamische Anpassung der Timer an die Gegebenheiten des Netzes.

Für Netzwerke wie das vorgefundene „Standalone-LAN“ ist allerdings die einfache Lösung mit vernachlässigter Netzlaufzeit eine durchaus akzeptable Annahme.

3.4 Protokoll-Klassen

Verschiedene Netze machen verschiedene Fehler. Transportprotokolle müssen mit diesen Fehlern umgehen können und sind demnach mehr oder weniger komplex. Die Komplexität eines Transportprotokolls hängt daneben stark von der Zuverlässigkeit des unterlagerten Vermittlungsdienstes und der vom Dienstbenutzer geforderten *Quality of Service* ab. OSI definiert fünf Transportprotokoll-Klassen, die nach absteigender Netzwerk-Zuverlässigkeit geordnet sind:

- **Klasse 0 (einfache Klasse)** entspricht der CCITT-Empfehlung T.70 für Telex und basiert auf einem zuverlässigen Vermittlungsdienst, der Flusskontrolle, Fehlerüberwachung etc. übernimmt. Das Klasse-0-Transportprotokoll veranlasst lediglich Verbindungsauf- / -Abbau und Datentransfer.

- **Klasse 1 (mit einfacher Fehlerüberwachung)** entspricht Klasse 0, ist zusätzlich aber in der Lage, nach einem Abbruch der Vermittlungsverbindung eine neue Verbindung aufzusetzen und die Datenübertragung an der Unterbrechungsstelle fortzusetzen.

- **Klasse 2 (Multiplexing-Klasse)** entspricht Klasse 0, kann allerdings mehrere Transportverbindungen auf eine Vermittlungsverbindung multiplexen.

- **Klasse 3 (Multiplexing und Fehlerüberwachung)** entspricht Klasse 1 mit Multiplexing von Transportverbindungen auf eine Vermittlungsverbindung.

- **Klasse 4 (mit Fehlererkennung und –behebung)** bietet alle Funktionen der Klasse 3, geht aber von unzuverlässigen unterlagerten Protokollschichten aus, die TPDUs verlieren und vertauschen können. Ein Transportprotokoll der Klasse 4 übernimmt alle oben geschilderten Aufgaben der Flusskontrolle, Fehlererkennung, Zeitüberwachung, Wiederholung, etc. Hier ist die Transportschicht selbst zuständig für die Bereitstellung eines zuverlässigen Transportdienstes.

Das OSI-Transportprotokoll der Klasse 4 wird kurz **TP4** genannt und ist eine Untermenge der ISO/OSI-8073-Spezifikation. Alle Ausführungen in diesem Dokument beziehen sich auf die für diese Arbeit relevante TP4-Variante der OSI-Protokolle, sofern nicht anders gekennzeichnet.

3.5 Verbindungsauf- und -abbau

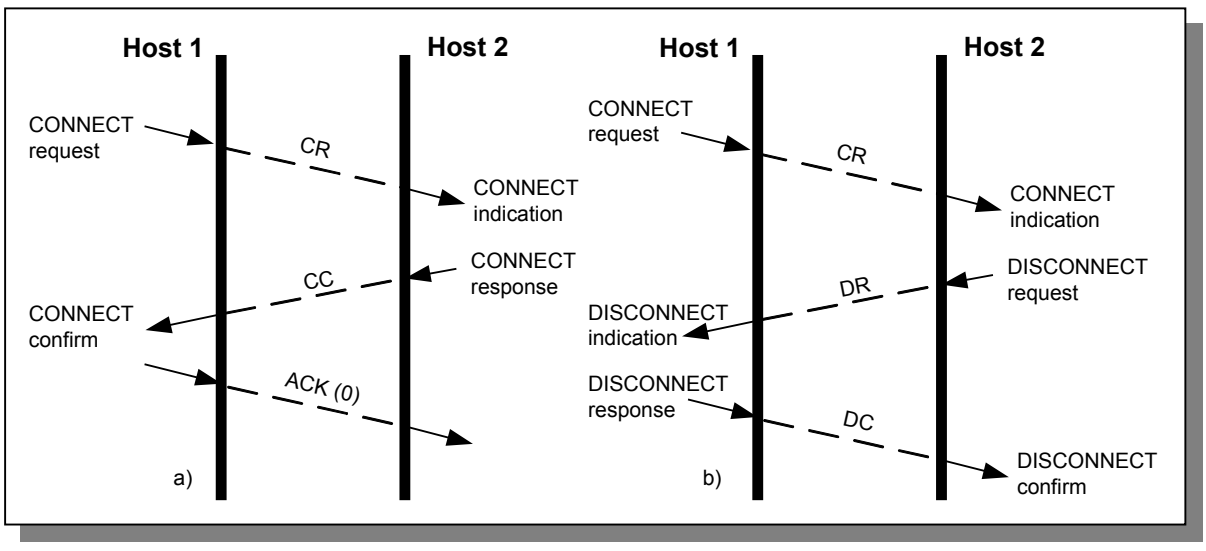


Abbildung 3.3: Verbindungsaufbau. a) 3-Wege-Handshake bei erfolgreichem Aufbau. b) Host 2 lehnt Verbindungswunsch ab

Der TP4-Verbindungsaufbau arbeitet, ähnlich TCP, mit einem Drei-Wege-Handshake gemäß Abbildung 3.3a). Möchte Host 1 eine Verbindung zum entfernten Host 2 aufbauen, sendet er eine *Connection Request (CR)*-TPDU. Trifft diese bei Host 2 ein, löst sie dort nach der OSI-Nomenklatur ein CONNECT.indication-Ereignis aus. Host 2 reagiert darauf im Normalfall durch das Versenden einer *Connection Confirm (CC)*-TPDU. Deren Eingang bei Host 1 verursacht dort wiederum ein CONNECT.confirm-Ereignis. Host 1 weiß nun, dass Host 2 den Verbindungswunsch angenommen hat. Abschließend bestätigt Host 1 den Eingang des *CC* mit einer *Acknowledge (ACK)*-TPDU und teilt Host 2 gleichzeitig die Bereitschaft zum Empfang der Daten-TPDU mit Sequenznummer 0 mit. Danach können bidirektional Daten-TPDUs ausgetauscht werden.

Bei TP4 werden bestehende Verbindungen über eindeutige Verbindungsnummern identifiziert, die beide Partner individuell vergeben. Die Hosts tauschen ihre jeweilige Verbindungsnummer als Bestandteil der *CR*- bzw. *CC*-TPDU beim Verbindungsaufbau aus. Alle weiteren über diese Verbindung gesendeten TPDUs enthalten mindestens die von der Zielstation vergebene Verbindungsnummer. Anhand dieser kann die Transportinstanz des Empfängers eingehende TPDUs zu einer bestimmten aus mehreren gleichzeitig bestehenden Verbindungen zuordnen.

Das Szenario in Abbildung 3.3b) zeigt den Ablauf, wenn Host 2 den Verbindungswunsch ablehnt. In diesem Fall wird er den Eingang des *CR* mit einer *Disconnect Request (DR)*-TPDU beantworten. Innerhalb der DISCONNECT.indication-Behandlungsroutine von Host 1 werden höhere Protokollebenen über dieses Ereignis informiert und eine *Disconnect Confirm (DC)*-TPDU an Host 2 gesendet. Darauf erfolgt die Freigabe aller für diese Verbindung bereits reservierten Ressourcen.

Ähnlich verläuft der reguläre Verbindungsabbau: In einem Zwei-Wege-Handshake werden *DR*- und *DC*-TPDUs gemäß nebenstehender Abbildung ausgetauscht. ISO/OSI 8073 sieht eine abrupte Trennung der beiden Partner vor. Stehen beim Empfang eines *DR* noch Telegramme aus, wird die Verbindung ohne Rücksicht auf Datenverluste getrennt. Für die Vermeidung dieses Falls ist nach OSI die Sitzungsschicht zuständig, die ein *DR* erst nach korrekter Übertragung aller Daten veranlassen soll.

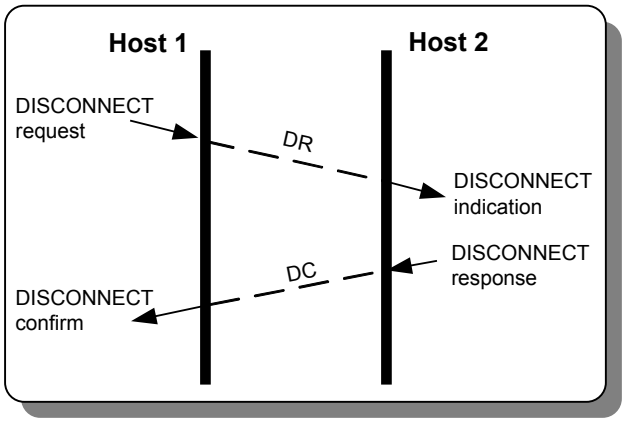


Abbildung 3.4: Verbindungsabbau

Die Transportschicht bietet ihre Dienste den übergeordneten Schichten über *Transport Service Access Points (TSAPs)* an. Natürlich können sich auf einem Rechner mehrere Prozesse zugleich an einen TSAP ankoppeln und auf Verbindungsanfragen warten. Da diese Prozesse gezielt adressierbar sein sollen, wird jedem Prozess eine eindeutige *TSAP-ID* zugeordnet. Innerhalb eines *Connection Request* kann mit Hilfe der *TSAP-ID* nun gezielt ein Prozess auf der Gegenseite kontaktiert werden. Beim TCP-Protokoll wird der TSAP *TCP-Port* genannt, das Pendant zur TSAP-ID ist dort die *Port-Nummer.*

3.6 Die TPDU-Typen des ISO/OSI-8073-Protokolls

3.6.1 Allgemeine Struktur

Das ISO/OSI-8073-Protokoll kennt zehn verschiedene TPDU-Typen, die jeweils aus bis zu vier Teilen bestehen, wie in Abbildung 3.5 dargestellt.

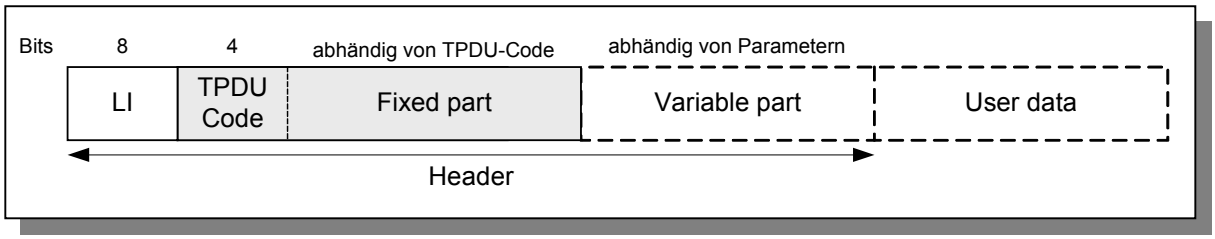


Abbildung 3.5: Allgemeiner Aufbau der ISO/OSI-8073-Dateneinheiten

TPDUs enthalten in folgender Reihenfolge:

1. Den ***Header*** mit den Feldern:

   - **Length Indicator (LI)** im ersten Header-Byte. *LI* gibt die Länge des Headers bis zu einer Gesamtlänge von 254 Bytes an. Das *LI*-Byte selbst wird bei der Längenberechnung nicht berücksichtigt.
   - **Fester Teil (Fixed part)**: Der feste Teil des Headers enthält wichtige, in jeder TPDU eines Typs vorhandene Parameter. TPDU-Typen werden durch den *TPDU-Code* unterschieden, der stets in den ersten vier Bits des festen Teils steht. Die übrige Struktur des festen Teils sowie dessen Gesamtlänge sind durch den TPDU-Code festgelegt. Der feste Teil ist in Abbildung 3.5 und den folgenden Abbildungen zu TPDU-Formaten jeweils grau hinterlegt.
   - **Variabler Teil (Variable part)** : Im variablen Teil des Headers stehen optionale Parameter, die nicht immer benötigt werden. Sind keine optionalen Parameter erforderlich, entfällt der variable Teil. Ein Parameter im variablen Teil beginnt mit dessen Parameter-Code, gefolgt von der Länge des Parameter-Daten-Feldes. Letzteres enthält die Daten des im Code-Feld bezeichneten Parameters, falls zu diesem Code weitere Daten existieren. Verschiedene Parameter im variablen Teil können in beliebiger Reihenfolge aufeinander folgen.

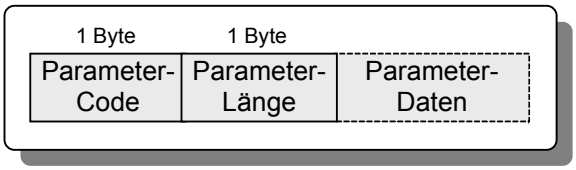


Abbildung 3.6: Form eines variablen Parameters

2. Die ***Benutzerdaten (User data)***, falls vorhanden. Zum Benutzerdaten-Transport dient vorwiegend die dazu definierte Daten-TPDU. Manche anderen TPDUs können jedoch ebenfalls Benutzerdaten transportieren. Ausnutzen lässt sich dies beispielsweise beim Verbindungsaufbau, indem mit der *Connection Request*-TPDU gleich ein Passwort übertragen wird.

3.6.2 Die zehn ISO/OSI-8073-TPDUs

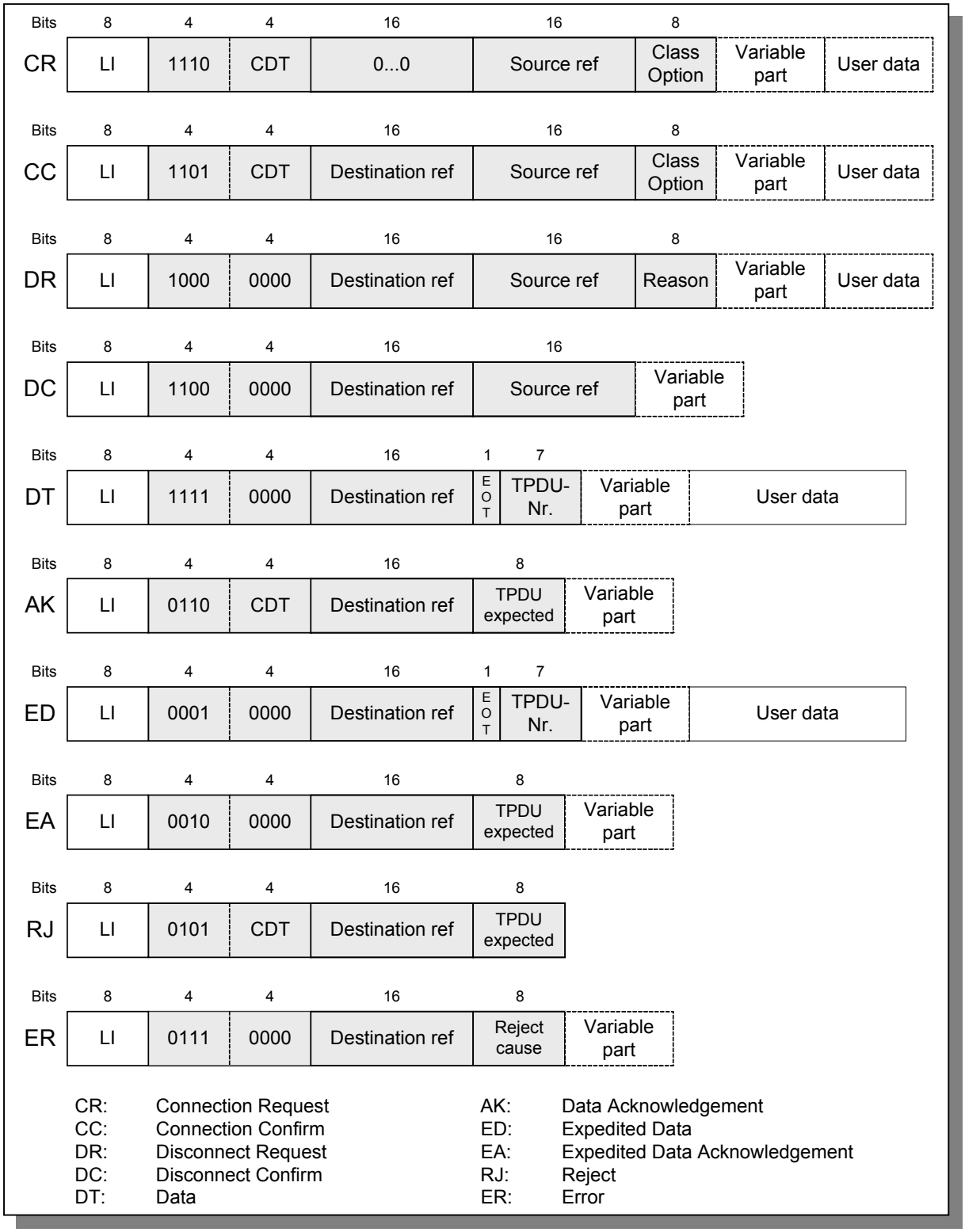


Abbildung 3.7: Die ISO/OSI-8073-TPDUs

Alle zehn TPDUs des ISO/OSI-8073-Standards sind in Abbildung 3.7 zusammengefasst. Davon werden im zugrunde liegenden System die ersten sechs verwendet. Zu einigen dieser TPDUs existieren spezielle Varianten, die hier nicht von Bedeutung sind.

Da viele Parameter des festen Teils (in Abbildung 3.7 jeweils grau unterlegt) mehreren TPDUs gemeinsam sind, folgt nun zunächst eine Aufstellung dieser Parameter. Die Besonderheiten der einzelnen TPDUs werden im Anschluss daran in Verbindung mit den jeweiligen variablen Teilen beschrieben.

3.6.2.1 Parameter im festen Teil

- **Credit (CDT)** 4 Bit

  Anfangs-Sendekreditzuweisung für Flusskontrolle. Der Credit belegt die unteren vier Bits des ersten Bytes im festen Teil der TPDU. Die oberen vier Bits sind jeweils durch den TPDU-Code belegt.

- **Source Reference** 16 Bit

  Quellverweis. Von der Absender-Seite für diese Verbindung vergebene („lokale“) Verbindungsnummer.

- **Destination Reference** 16 Bit

  Zielverweis. Von der Empfänger-Seite für diese Verbindung vergebene („remote“) Verbindungsnummer.

- **Class Option** 8 Bit, Format: xxxx 0000; xxxx: Protokollklasse (0..4)

  Dient zur Aushandlung der Protokollklasse gemäß Abschnitt 3.4. Der Verbindungs-Initiator liefert einen Vorschlag, den der Angerufene annehmen oder durch Senden eines *DR* ablehnen kann. Der Angerufene kann in seinem *CC* auch eine niedrigere Protokoll-Kategorie als Gegenvorschlag festlegen. Die Protokollklasse steht in den oberen vier Bits. Die unteren vier Bits ermöglichen zusätzliche, hier irrelevante Optionen.

- **Reason** 8 Bit

  Grund für Verbindungsabbau, insbesondere:
  0: Normaler oder nicht weiter spezifizierter Verbindungsabbau
  1: Überbelastung
  2: TSAP existiert nicht

- **EOT (End of Transport)** 1 Bit

  Kennzeichnet die letzte TPDU einer größeren Datenmenge, die zum Transport auf mehrere TPDUs segmentiert wurde. Die empfangene Transport-Instanz kann daraufhin die reassemblierten Daten an die übergeordnete Instanz leiten. Im vorliegenden System immer gesetzt (Segmentierung erfolgt durch Sitzungsschicht).

- **TPDU-Nr.** 7 Bit

  Sequenzfolgenummer der TPDU.

- **TPDU expected** 8 Bit (höchstwertiges Bit stets 0)

  Als nächstes von der Gegenseite erwartete TPDU.

- **Reject cause** 8 Bit

  Ablehnungsgrund bei Protokollfehlern. Nur in Klasse 1 und 3 relevant.

3.6.2.2 Connection Request (CR) und Connection Confirm (CC)

Diese beiden TPDUs dienen zum Verbindungsaufbau in der auf Seite 43 unter 3.5, *„Verbindungsauf- und -abbau“* beschriebenen Abfolge.

Parameter im variablen Teil:

- **TSAP-ID**

Parameter-Code:	0xC1	für lokalen TSAP der aufrufenden Transportinstanz
	0xC2	für entfernten TSAP, an den angeschlossen werden soll
Parameter-Länge:	2 Byte	
Parameter-Wert:	16-Bit-Kennung des aufgerufenen bzw. aufrufenden TSAPs	

  Die beiden TSAP-IDs für aufrufenden und aufgerufenen TSAP sind für TP4 in der *CR*-TPDU Pflicht. Beim *CC* können sie als Bestätigung optional angegeben werden.

- **TPDU-Größe**

Parameter-Code:	0xC0	
Parameter-Länge:	1 Byte	
Parameter-Wert:	0000 1101	max. TPDU-Größe 8192 Bytes
	0000 1100	max. TPDU-Größe 4096 Bytes
	0000 1011	max. TPDU-Größe 2048 Bytes
	0000 1010	max. TPDU-Größe 1024 Bytes
	0000 1001	max. TPDU-Größe 512 Bytes
	0000 1000	max. TPDU-Größe 256 Bytes
	0000 0111	max. TPDU-Größe 128 Bytes

  Mit diesem Parameter handeln die beiden Partner eine maximal erlaubte TPDU-Größe für die Verbindung aus. Der Verbindungs-Initiator schlägt in seinem *CR* einen Wert vor, den der Partner in seinem *CC* übernehmen oder durch einen kleineren Wert ersetzen kann. Im vorgefundenen System ist die TPDU-Größe auf 1024 Bytes festgesetzt.

- **Checksum**

  Parameter-Code: 0xC3
  Parameter-Länge: 2 Byte
  Parameter-Wert: 16-Bit-Checksumme

  Pflichtparameter bei *CC* und *CR*. Enthält die Prüfsumme über alle Elemente der TPDU. Der Wert wird bei den vorgefundenen Telegrammen nicht ausgewertet, da dies mit der folgenden Option deaktiviert ist.

- **Additional Option Selection**

  Parameter-Code: 0xC6
  Parameter-Länge: 1 Byte
  Parameter-Wert: 0000 wxyz

  w=1: Eildatenverkehr in Klasse 1 aktiviert
  x=1: Empfangsbestätigung in Klasse 1 aktiviert
  y=1: Auswertung der 16-Bit-Checksumme deaktiviert
  z=1: Benutzung Eildaten-Transportdienst aktiviert

  Dieser Parameter ist 0x02 in allen TPDUs des vorgefundenen Systems. Damit ist die Auswertung der Telegramm-Checksumme (s. o.) für alle Telegramme über diese Verbindung deaktiviert.

- **Acknowledge Time**

  Parameter-Code: 0x85
  Parameter-Länge: 2 Byte
  Parameter-Wert: 16-Bit-Zahl als Zeit in Millisekunden

  Die *Acknowledge Time* ist die Zeit, die maximal verstreichen darf zwischen dem Aussenden einer Daten-TPDU und dem Empfang des korrespondierenden Acknowledge. Im vorgefundenen System beträgt diese Zeit 500ms.

- **Weitere mögliche Parameter**, im Projekt nicht benutzt:

  - **Versionsnummer** der Transportprotokoll-Software
  - **Sicherheitsparameter** (vorgesehen, aber nicht näher spezifiziert)
  - **Alternative Protokollklassen**, die der Verbindungsinitiator akzeptiert
  - **Throughput**: Erwarteter Datendurchsatz und akzeptables Minimum
  - **Restfehlerrate**: Gewünschter Durchschnitt und akzeptables Minimum
  - **Priorität**: Verbindungen mit niedrigerer Priorität werden bevorzugt behandelt
  - **Transit Delay**: Tolerierte TPDU-Durchlaufverzögerung zwischen den Transportinstanzen des Senders und Empfängers
  - **Dauer der Verbindungs-Wiederherstellungsversuche** nach einem Absturz

3.6.2.3 Disconnect Request (CR) und Disconnect Confirm (CC)

Diese beiden TPDUs dienen zum Verbindungsabbau in der auf Seite 43 unter 3.5, *„Verbindungsauf- und -abbau“* beschriebenen Abfolge. Für den variablen Teil sind zwei Parameter definiert, die beide im vorliegenden System nicht auftreten:

Parameter im variablen Teil:

- **Checksum,**
  siehe Abschnitt 3.6.2.2, Parameter ***Checksum***.
  Der Parameter kann entfallen, wenn die Auswertung der Checksumme beim Verbindungsaufbau über ***Additional Option Selection*** deaktiviert wurde.

- **Additional Information**
  ermöglicht dem Dienstbenutzer die Übertragung zusätzlicher, frei definierbarer Informationen zum Verbindungsabbau.

3.6.2.4 Data (DT) und Data Acknowledge (AK)

Die TPDUs für Datenübertragung und Bestätigung erlauben in ihrem variablen Teil die Angabe der **Checksum**, wie oben. Für die Acknowledge-TPDU existiert ein weiterer, ebenfalls hier nicht verwendeter Parameter ***Flow Control Confirmation***. Bei dessen Einsatz wird an den Host, der eine Acknowledge-TPDU gesendet hat, zur Bestätigung eine Kopie dieser TPDU zurückgesendet. Auf diese Art kann der Absender des ACK sich über den Zustand des Ziel-Hosts vergewissern.

3.6.2.5 Noch mehr TPDUs

Neben den oben vorgestellten TPDUs definiert ISO/OSI 8073 vier weitere, im Projekt nicht auftretende Transportdateneinheiten. Der Vollständigkeit halber zusammengefasst sind dies:

- **Expedited Data (ED), Expedited Data Acknowledge (EA)**

  Diese beiden TPDUs ermöglichen das Versenden von bevorzugt behandelten *Eildaten* sowie deren Bestätigung. Die Struktur entspricht den TPDUs *DT* und *AK* für normale Daten, jedoch ist die maximale Nutzdatenlänge bei Eildaten auf 16 Byte begrenzt.

- **Reject (RJ)**

  Bei einem Fehler kann der Partner mit dieser TPDU aufgefordert werden, alle TPDUs ab derjenigen mit Sequenznummer *TPDU expected* zu wiederholen. Die *Reject*-TPDU wird nur in den Protokollklassen 1 und 3 benutzt.

- **Error (ER)**

  Mit dieser TPDU können Protokollfehler berichtet werden. Das Feld *Reject Cause* informiert über den aufgetretenen Fehler, wie *falscher Parameter im variablen Teil* oder *ungültiger TPDU-Typ.* Die *Error*-TPDU wird ebenfalls nur in Klasse 1 und 3 verwendet.

3.7 Verbindungslose Datenübertragung

Neben dem in den vorigen Abschnitten dargestellten verbindungsorientierten Transportdienst sieht die OSI-Transportschicht auch eine Möglichkeit zur verbindungslosen Datenübertragung vor.

Verbindungslos zu übertragende Daten werden mit der TPDU aus Abbildung 3.8 transportiert, deren Aufbau den TPDUs für verbindungsorientierte Übertragung ähnelt. Im variablen Teil stehen Parameter für Source- und Destination-TSAP sowie, falls benötigt, die Prüfsumme. Das Format dieser Parameter stimmt mit der Beschreibung aus Abschnitt 3.6.2.2, *„Connection Request (CR) und Connection Confirm (CC)"*, überein.

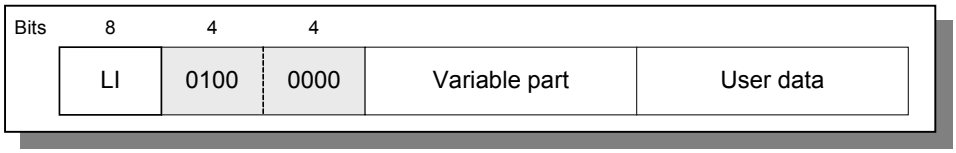


Abbildung 3.8: Struktur der TPDU für verbindungslose Übertragung

Definitionsgemäß werden verbindungslos gesendete Daten nicht bestätigt und es gibt weder eine Gewähr für die zuverlässige Übermittlung noch für die Wahrung der Reihenfolge. Sinn macht eine solche verbindungslose Übertragung z. B. für allgemeine Broadcast-Nachrichten. Im vorliegenden Fall senden alle Rechner im Netz zyklisch Lebenszeichen-Broadcasts an alle anderen Rechner. Durch deren Auswertung ist prinzipiell jeder Rechner über den Zustand jedes anderen Rechners informiert.

3.8 Vergleich von TP4 mit TCP

Nachdem nun die wesentlichen Merkmale des TP4-Protokolls ans Tageslicht gerückt wurden, bietet sich eine Gegenüberstellung der Gemeinsamkeiten und Unterschiede zum heute übermächtigen „Sieger-Transportprotokoll“ TCP an.

Gemeinsamkeiten:

- Zuverlässige Ende-zu-Ende-Transportdienste auf unzuverlässigem Vermittlungsdienst
- Verbindungsorientiert mit Drei-Wege-Handshake beim Verbindungsaufbau
- Zusätzlich verbindungslose Übertragung definiert (UDP-Pendant bei TCP)

Unterschiede:

- **TPDU-Typen:**
  - TP4 kennt zehn verschiedene TPDU-Typen.
  - Nur eine TPDU bei TCP. Als Konsequenz ist TCP einfacher, erzeugt aber durch den größeren Nachrichtenkopf deutlich mehr Overhead (mindestens 20 Header-Bytes gegenüber 5 Bytes der TP4-Daten-TPDU).

- **Quality of Service (QoS):**
  - TP4 kann umfangreiche Dienstgüte-Parameter beim Verbindungsaufbau aushandeln.
  - TCP kennt keinerlei QoS-Parameter. Stattdessen sieht das unterlagerte IP ein 8-Bit-Feld für QoS-Parameter vor, die aber häufig nicht ausgewertet werden.

- **Nachrichtenstrom:**
  - TP4 überträgt eine Folge geordneter Nachrichten, d. h. die Nachrichtengrenzen bleiben bei der Übertragung gewahrt.
  - TCP sendet einen kontinuierlichen Bytestream ohne ausdrückliche Nachrichtengrenzen. Die Ausfilterung einzelner, vollständiger Nachrichten obliegt dem Benutzer.

- **Vorrangdaten:**
  - TP4 multiplext normale und Eildaten über separate TPDUs.
  - TCP kennzeichnet über den *Urgent*-Pointer des Headers wichtige, gesondert zu handhabende Daten im Datenstrom.

- **Huckepack-Bestätigung (*piggibacking):***
  - Bei TP4 nicht vorgesehen.
  - Bei TCP über das ACK-Feld im Header; somit Bestandteil jeder Übertragung.

- **Flusskontrolle*:***
  - Bei TP4 über Kreditschema.
  - Bei TCP über Sliding Window-Verfahren mit Übermittlung der Fenstergröße in jeder TPDU.

- **Verbindungsabbau:**
  - In TP4 erfolgt eine abrupte Trennung. Dabei kann es zu Datenverlusten kommen, wenn das *Disconnect Request* vorab gesendete Daten-TPDUs überholt oder solche verloren gehen. Nach OSI sind zur Vermeidung von derartigen Problemen höhere Protokollschichten zuständig.
  - TCP vermeidet diese Schwierigkeiten durch einen Drei-Wege-Verbindungsabbau.

3.9 Adress-Auflösung

Zum Abschluss dieses Kapitels ist die Frage zu klären, wie sich zwei Rechner finden, die miteinander kommunizieren sollen.

Eine besonders charmante Eigenschaft des OSI-Modells ist die durch die Kapselung der einzelnen Schichten erreichte Modularisierung. Bei konsequenter Umsetzung dieses Konzepts ist es möglich, einzelne Schichten bzw. das darin implementierte Protokoll gegen ein anderes auszutauschen, ohne angrenzende Schichten zu beeinflussen.

Dies setzt zum einen natürlich voraus, dass die Schnittstelle, also der *Service Access Point*, zwischen angrenzenden Schichten unverändert bleibt. Zum anderen ist die Art und Weise, wie Schicht n ihre Partnerinstanz auf der Gegenseite adressiert, internes Charakteristikum von Schicht n: Ob Schicht n dazu MAC-Adressen oder Funkkanäle benutzt, sollte auf Schicht n+1 keinen Einfluss haben.

Allerdings bedeutet die Einhaltung dieser Vorgaben zusätzlichen Aufwand zur Adressauflösung. TCP/IP verwendet beispielsweise das *address resolution protocol (arp)*, über das per *Broadcast*-Anfrage ins Netz zu einer gegebenen IP-Adresse (OSI-Schicht 3) die zugehörige MAC-Adresse (OSI-Schicht 2) ermittelt werden kann. Das vorliegende System vermeidet diesen Aufwand und bricht zur Adressierung mit der Vorgabe des OSI-Modells: Die Transportschicht kennt die MAC-Adressen der anzusprechenden Partner.

Das System identifiziert alle Arbeitsplätze über eine Arbeitsplatz-Kennung, auch mit *AP-Typ* bezeichnet. Die Ermittlung der MAC-Adresse zu einer Arbeitsplatz-Kennung erfolgt anhand einer Liste, die im Altsystem Bestandteil der Systemsoftware war. Unser neues System lagert diese *Arbeitsplatz-Liste* in die globale Konfigurationsdatei *globalconfig.ini* aus. Beim Systemstart wird die Liste vom Anlaufprozess in einen reservierten Shared Memory-Bereich eingelesen. Neben des AP-Typs und der zugehörigen MAC-Adresse enthält die Liste weitere arbeitsplatz-spezifische Informationen, darunter die TSAP-ID und den Dateinamen des beim Urladen zu versendenden Systemprogramms:

```
<AP_LISTE>
  <AP>
    <NAME>Arbeitsplatz A20</NAME>       // symbolischer Name
    <AP_NR>097</AP_NR>                  // Arbeitsplatz-Nr.
    <MAC>00:0A:12:19:89:61</MAC>        // MAC-Adresse
    <TSAP_ID>4261</TSAP_ID>             // TSAP-ID
    <AP_TYP>A20</AP_TYP>                // Eindeutige AP-Kennung
    <STATUS>7</STATUS>                  // Arbeitsplatz-Status
    <PROGRAMM>apsys</PROGRAMM>          // Urlade-Programm
  </AP>
  <AP>
    <NAME>Datenkonzentrator DK1</NAME>  // Auch die DK-Rechner beziehen
    <AP_NR>253</AP_NR>                  // ihre eigene Konfiguration aus
    <MAC>00:0A:12:19:89:FD</MAC>        // dieser Liste
    <TSAP_ID>42FD</TSAP_ID>
    <AP_TYP>DK1</AP_TYP>
    <STATUS>7</STATUS>
    <PROGRAMM></PROGRAMM>
  </AP>
</AP_LISTE>
```

Abbildung 3.9: Die Arbeitsplatz-Liste aus der Datei globalconfig.ini

Die Funktionen zum Einlesen und Auflösen von Adressen wurden aus dem Altsystem portiert und sind daher hier nicht von weiterem Interesse.

4 Raw Sockets und Data Link Control

Die MAC-Adresse einer Netzwerkkarte ist in der Regel durch den Hersteller fest mit der Netzwerkkarten-Hardware verbunden. Weil aber alle Arbeitsplätze, wie im letzten Abschnitt beschrieben, die DK-Rechner über eine in der Betriebssoftware verankerte MAC-Adressenliste adressieren, bringt dies eine Schwierigkeit mit sich: Beim Austausch der DK-Rechner mitsamt Netzwerkkarte durch neue Hardware wird sich unweigerlich deren MAC-Adresse ändern. An die alte MAC-Adresse gerichtete Pakete gehen ohne weiteren Aufwand verloren. Da ein Eingriff in die Arbeitsplatz-Betriebssoftware zwecks Änderung der internen Liste nicht in Frage kam, wurden zwei Lösungsmöglichkeiten auf der DK-Seite untersucht.

Erste Überlegung war die Verwendung einer speziellen Netzwerkkarte mit per Software einstellbarer Parameter, darunter die MAC-Adresse. Allerdings bedeutete dies den Einsatz einer proprietären Komponente mit ebensolchen Treibern, deren Linux-Unterstützung nicht alle gestellten Anforderungen an Flexibilität, Einfachheit und Robustheit erfüllen konnte. Zudem wäre bei einer Hardware-Lösung die neue Software nur auf entsprechend ausgerüsteten Rechnern lauffähig.

Ein zweiter Aspekt kommt hinzu: Zur Implementation eines eigenen Netzwerk-Protokolls, wie TP4, bedarf es einer Möglichkeit, eigene Protokoll-Dateneinheiten über das Netz zu senden und zu empfangen.

Gesucht und gefunden wurde also eine reine Software-Lösung, die unabhängig von der eingesetzten Netzwerkkarte mit jedem Linux-System funktioniert. Diese Lösung nutzt den *promiscuous mode* der Netzwerkkarte sowie die bereits angesprochenen *Raw Sockets*.

4.1 Raw Sockets

Ein *Socket* ist ein vom Betriebssystem bereitgestellter Endpunkt zur Kommunikation. Erzeugen lässt sich ein Socket allgemein durch den Systemaufruf

```
int socket(int domain, int type, int protocol),
```

zu dem sich auf den einschlägigen Manpages des Linux-Betriebssystems ausführliche Informationen finden. Für die Erstellung eines Sockets im RAW-Mode sind die folgenden Parameter vorgesehen:

- ***domain*=PF_PACKET**
  Mit der Angabe von *PF_PACKET* für den *domain*-Parameter wird die Protokoll-Familie *packet socket* ausgewählt. Ein *packet socket* erlaubt das Senden und Empfangen von OSI-Schicht-2-Rahmen, also von Telegrammen unmittelbar oberhalb der physikalischen Schicht. Dazu setzt das *packet socket interface* auf dem Netzwerkkartentreiber auf. Dies ermöglicht die Erstellung vollständiger eigener Netzwerk-Protokollmodule, wie hier die TP4-Implementation.

- ***type*=SOCK_RAW**
  Mittels der Angabe von `SOCK_RAW` für den Socket-Typ wird ein packet socket im *RAW-Mode* erstellt. Mit dieser Option enthalten die über den Socket übergebenen Telegramme vollständige, unveränderte *raw packets* mitsamt zugehörigen *Data Link*-Headern (OSI-Schicht 2). Die Alternative, `SOCK_DGRAM`, würde diesen Header vor der Übergabe an den Benutzer entfernen.

- ***protocol*=htons(ETH_P_ALL)**
  Es existieren verschiedene mögliche OSI-Schicht-2-Protokolle, die über diesen Parameter gezielt ausgefiltert werden könnten. Mit der gesetzten Option `ETH_P_ALL` empfängt unser *packet socket* kurzerhand alle diese Protokolle. Dies ist unspektakulär, da auf dem vorliegenden Netzwerk ohnehin nur IEEE802.3-Telegramme übertragen werden.

Rückgabewert des `socket()`-Aufrufs ist ein als Socket-Descriptor bezeichnetes Betriebssystem-Handle für den erstellten Socket. Für das Erstellen eines *Raw Sockets* benötigt der aufrufende Prozess root-Rechte.

Zum Senden und Empfangen von Daten über den Socket dienen die Systemaufrufe ***sendto()*** und ***recvfrom()***. Beide erwarten als Parameter unter anderem den Socket-Descriptor, einen Pointer auf den Buffer für zu sendende bzw. zu empfangende Daten sowie die Bufferlänge. Im *Raw Socket*-Mode ist es Aufgabe des Benutzers, die Daten im Buffer vor dem Versenden mit den erforderlichen Protokollheadern, in diesem Fall IEEE 802.3 mit der TP4-Payload, zu versehen. Umgekehrt enthält der Buffer nach einem erfolgreichen `recvfrom()`-Aufruf einen vollständigen IEEE 802.3-Rahmen mit komplettem Header, der durch den Benutzer weiter ausgewertet werden kann.

Die beiden Funktionen kommen auch zum verbindungslosen Datentransfer über UDP/IP-Sockets zum Einsatz und wurden entsprechend oft anderweitig beschrieben. Daher sei ein weiteres Mal auf die zugehörigen Manpages als erste Anlaufstelle für weitere Informationen verwiesen.

4.2 Der *Promiscuous Mode*

Nach wie vor empfängt der oben erstellte *Raw Socket* nur Telegramme, deren Ziel-MAC-Adresse mit derjenigen der eigenen Netzwerkkarte übereinstimmt. Alle übrigen werden durch die Netzwerkkarte ausgefiltert.

An dieser Stelle hilft eine Technik weiter, die sich auch Programme zur Netzwerk-Protokollanalyse (*„Network-Sniffer“)* zunutze machen: Das Betreiben der Netzwerkkarte im *promiscuous mode*. Die treffende Bezeichnung *promiscuous* („*bunt gewürfelt*“) besagt nichts anderes, als dass die Netzwerkkarte in diesem Modus alle Datenrahmen mit beliebiger Zieladresse ungefiltert empfängt und an den Devicetreiber weiterreicht. Auf diese Art wird das Mitlesen des gesamten Netzwerk-Datenverkehrs möglich.

Das Umschalten in den *promiscuous mode* geschieht über den Systemaufruf ***ioctl()*** und wird mit einem Codebeispiel demonstriert:

```
int nSocket;                    // Variable für Socket-Descriptor
char achRcvBuffer[4096];        // Buffer für Datenempfang
int nBytesInRcvBuffer;          // Anzahl Bytes im Buffer
unsigned char* pbyFrameHead;    // Pointer auf den Rahmenanfang


// RAW-Socket öffnen
if ((nSocket = socket(PF_PACKET, SOCK_RAW, htons(ETH_P_ALL)))<0)
{
  printf ("\nFehler: Socket konnte nicht geöffnet werden!");
  exit(-1);
}

// Netzwerkkarte in promiscuous mode schalten
ifreq ethreq;       // Struktur für ioctl-Parameter

// Aktuelle Konfiguration von 'eth0' (Erste Netzwerkkarte im System) auslesen
strncpy (ethreq.ifr_name, "eth0", IFNAMSIZ);
ioctl (nSocket, SIOCGIFFLAGS, &ethreq);       // Konfiguration lesen

// Die ausgelesene Konfiguration zusätzlich mit dem Promiscuous-Flag verODERn
// und Konfiguration zurückschreiben
ethreq.ifr_flags |= IFF_PROMISC;
ioctl(this->m_nSocket, SIOCSIFFLAGS, &ethreq); // Konfiguration zurückschreiben

// Bereit zum Datenempfang über den Socket...
nBytesInRcvBuffer = recvfrom (nSocket, achRcvBuffer, sizeof (achRcvBuffer),
                              0, NULL, 0);

printf ("\n%d Bytes empfangen", nBytesInRcvBuffer);

// Pointer auf Beginn des IEEE802.3-Rahmens setzen
pbyFrameHead = (unsigned char*) achRcvBuffer;

// Hier empfangenes Telegramm auswerten...
...
```

Listing 4.1: Code-Beispiel zum Datenempfang im Promicuous Mode

Selbstverständlich birgt das Mitlesen des gesamten Netzwerkverkehrs enorme Sicherheitsrisiken. Das Umschalten der Netzwerkkarte in den promiscuous mode erfordert daher root-Rechte. Eine solcherart betriebene Netzwerkkarte verrät sich durch die Kennzeichnung *PROMISC* in der Ausgabe des Shellkommandos *ifconfig:*

```
ralf@grisu:~> ifconfig eth0
eth0      Protokoll:Ethernet  Hardware Adresse 00:E0:7D:B6:0B:CF
          inet Adresse:192.168.100.50  Bcast:192.168.100.255
Maske:255.255.255.0
          inet6 Adresse: fe80::2e0:7dff:feb6:bcf/64
Gültigkeitsbereich:Verbindung
          UP BROADCAST RUNNING PROMISC MULTICAST  MTU:1500  Metric:1
          RX packets:3264394 errors:0 dropped:0 overruns:0 frame:0
          TX packets:5232147 errors:0 dropped:0 overruns:0 carrier:0
          Kollisionen:0 Sendewarteschlangenlänge:100
          RX bytes:261476960 (249.3 Mb)  TX bytes:2299281609 (2192.7 Mb)
          Interrupt:11 Basisadresse:0x8000
```

Abbildung 4.1: Ausgabe des ifconfig*-Kommandos mit Netzwerkkarte im Promiscuous Mode*

4.3 Data Link Control

Beschäftigt man sich mit der Thematik „LAN mit CSMA/CD-Zugriffsverfahren nach IEEE 802.3“, so stößt man unweigerlich auf den heutzutage landläufig synonym benutzten Begriff „Ethernet“.

In der Tat bildete die von den Firmen DEC, Intel und Xerox *(„DIX“)* 1980 unter dem Namen *Ethernet* auf den Markt gebrachte LAN-Technologie in weiten Bereichen die Grundlage für die spätere Norm IEEE 802.3. Ein Unterschied besteht jedoch in der Struktur ihrer OSI-Schicht-2-(MAC)-Rahmen.

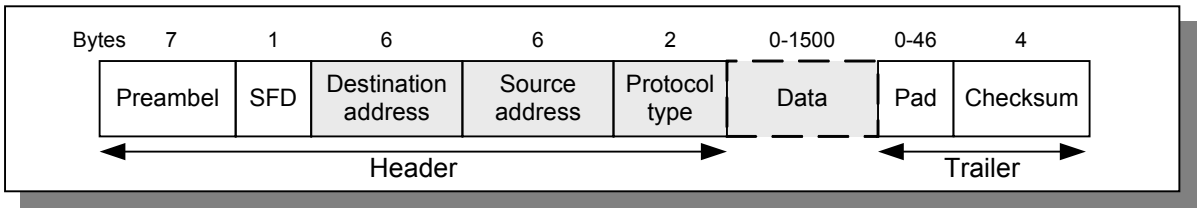


Abbildung 4.2: Ethernet-Rahmenformat

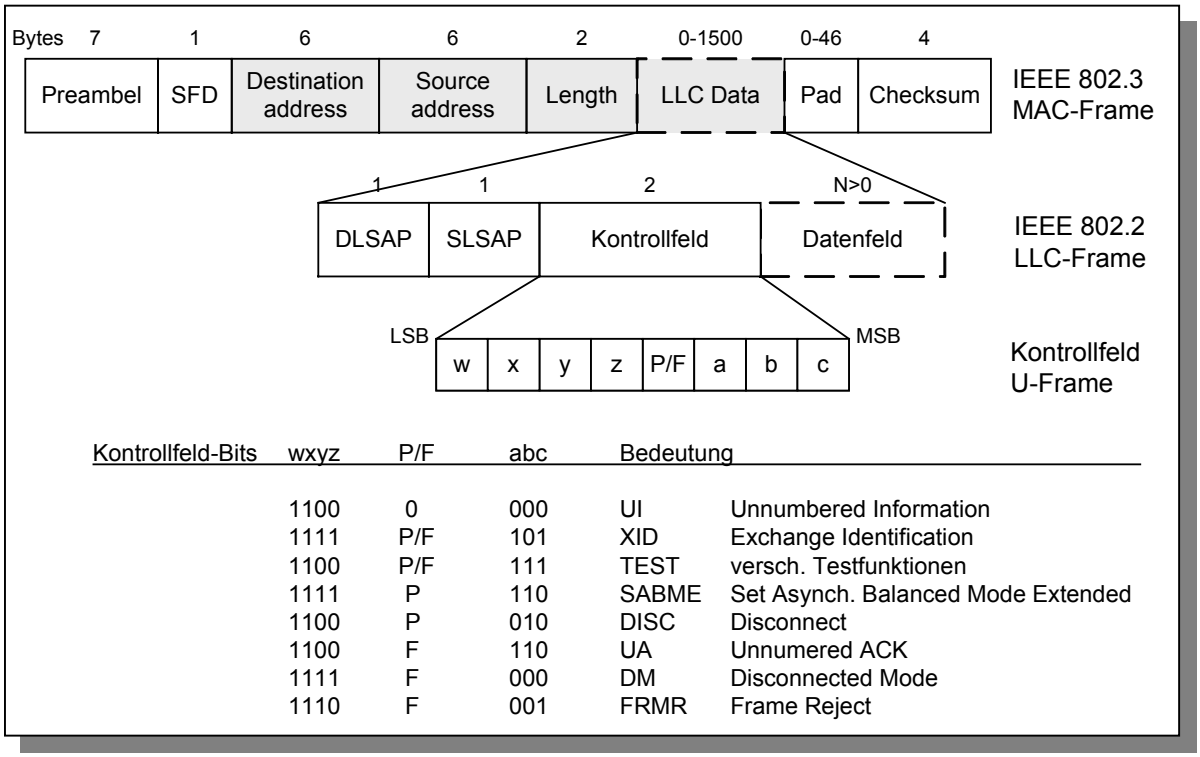
Das Ethernet-Rahmenformat ist in Abbildung 4.2 gezeigt. Es besteht aus den Feldern:

- **Preambel**
  Die Preambel dient der Sendetakt-Synchronisation des Empfängers. Sie enthält in sieben aufeinanderfolgenden Bytes die Bitfolge 10101010.
- **Start Frame Delimiter (SFD)**
  Kennzeichnet den Beginn der MAC-Daten durch die Bitfolge 10101011.
- **Destination Address, Source Address**
  Jeweils 6 Byte für die MAC-Adresse des Empfängers und Senders.
- **Protocol type**
  Kennzeichnet den Protokolltyp der Daten im folgenden Datenfeld, etwa 0x0800 für IP.
- **Data**
  Enthält die Nutzdaten der höheren Protokollschichten. Das Feld muss eine Mindestlänge von 46 Bytes haben.
- **Pad**
  Enthält das Datenfeld weniger als 46 Bytes, wird das Pad-Feld aufgefüllt, um die geforderte Mindest-Rahmenlänge von 64 Bytes zwischen Destination Address und Checksum zu erreichen.
- **Checksum**
  CRC-Prüfsumme über alle Daten zwischen Destination Address und Pad-Feld.

Im eigentlichen Sinne sind **Preambel** und **SFD**-Feld nicht Bestandteil des MAC-Rahmens, sondern werden auf OSI-Ebene 1 ergänzt. Selbiges gilt für die Felder **Pad** und **Checksum**. Bei der Datenübertragung über einen RAW-Socket werden daher nur die inneren, in Abbildung 4.2 grau gekennzeichneten Felder übergeben. Dies erspart dem RAW-Socket-Nutzer die Berechnung der Prüfsumme.

Das Ethernet-Format ist heute etablierter Standard für LAN-Systeme. Allerdings hat dessen Rahmenstruktur einen Schönheitsfehler: Die Angabe des Protokolls der übergeordneten Schicht im Feld *protocol type* widerspricht der durch das OSI-Modell angestrebten Kapselung der Protokollschichten. Ethernet ist streng genommen nicht OSI-konform, denn dazu sollte Schicht n einfach die Payload von Schicht n+1 transportieren, ohne weitere Informationen über den Typ dieser Daten zu benötigen.

Die Norm IEEE 802.3 umgeht dieses Manko, indem sie an Stelle des Header-Feldes **Protocol type** das **Length**-Feld vorsieht, welches die Gesamtlänge des folgenden Nutzdatenfeldes beinhaltet. So entsteht der in Abbildung 4.3 oben gezeigte IEEE 802.3-MAC-Rahmen.



Kontrollfeld-Bits	wxyz	P/F	abc	Bedeutung	
	1100	0	000	UI	Unnumbered Information
	1111	P/F	101	XID	Exchange Identification
	1100	P/F	111	TEST	versch. Testfunktionen
	1111	P	110	SABME	Set Asynch. Balanced Mode Extended
	1100	P	010	DISC	Disconnect
	1100	F	110	UA	Unnumered ACK
	1111	F	000	DM	Disconnected Mode
	1110	F	001	FRMR	Frame Reject

Abbildung 4.3: IEEE 802.3- und IEEE 802.2-Rahmen

Selbstverständlich muss ein solcher Rahmen auch ohne das *Protocol type*-Feld an die zuständige Protokollinstanz der übergeordneten Schicht weitergeleitet werden können. Dazu definiert die Norm IEEE 802.2 das *Logical Link Control*-Protokoll.

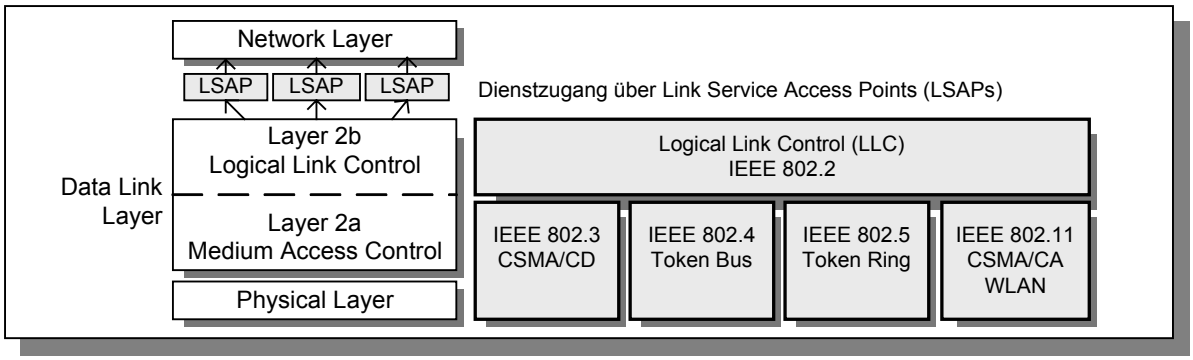


Abbildung 4.4: Normen für LLC- und MAC-Sublayer und Dienstzugang über LSAP

Die OSI-Schicht 2 ist für lokale Netzwerke in die in Abbildung 4.4 dargestellten Sublayer Logical Link Control (LLC) und Medium Access Control (MAC) unterteilt. Das LLC-Protokoll nach IEEE 802.2 fungiert als Schnittstelle zur übergeordneten Schicht und bietet dieser ihre Dienste über die Link Service Access Points (LSAPs) an. Da selbstredend mehrere verschiedene übergeordnete Protokolle auf die Dienste des LLC-Sublayers zugreifen können, ist jede dieser Protokoll-Instanzen über einen individuellen LSAP angekoppelt. Für bestimmte Protokolle sind feststehende LSAP-IDs definiert, beispielsweise hängt an LSAP 0xFE immer der hier relevante ISO/OSI-Network-Layer.

Während die LLC-Teilschicht für alle LAN-Technologien einheitlich ist, variiert die MAC-Teilschicht je nach eingesetzter Zugriffstechnologie des Netzwerks und dem davon abhängigen MAC-Rahmenformat. In der Riege der IEEE 802-Normen definiert 802.3 die Netzwerk-Physik nebst Rahmenformat für das CSMA/CD-Verfahren. Die feststehende Schnittstelle des LLC-Sublayers ermöglicht den Austausch dieser Technologie ohne Änderungen an den übergeordneten Schichten, etwa gegen WLAN nach IEEE 802.11

Zurück zum IEEE 802.3-MAC-Rahmen in Abbildung 4.3. Dieser technologie-abhängige Rahmen transportiert in seinem Nutzdatenfeld also immer einen technologie-unabhängigen LLC-Rahmen nach IEEE 802.2, in dessen Datenfeld wiederum die Nutzdaten der höheren Schichten stehen.

Das LLC-Protokoll nach IEEE 802.2 ist strukturell eng mit dem HDLC-Protokoll verwandt. Durch diese Verwandtschaft erbt es prinzipiell die Fähigkeiten von HDLC zu zuverlässiger, verbindungsorientierter Datenübertragung auf OSI-Ebene 2, genauso wie dessen Möglichkeiten für ungesicherten, paketorientierten Datenaustausch. Wie HDLC kennt IEEE 802.2 drei Rahmenformate:

- **Information (I)-Rahmen**
  I-Rahmen dienen zum gesicherten Nutzdatentransport mit Empfangs-Bestätigung. Entsprechend trägt jeder I-Rahmen eine eindeutige Sequenznummer.
- **Supervisory (S)-Rahmen**
  Mit S-Rahmen werden Steuerfunktionen und Flusskontrolle durchgeführt.
- **Unnumbered (U)-Rahmen**
  Die U-Rahmen übertragen verschiedene, über Kontrollfeld-Bits (siehe Abbildung 4.3) codierte Befehle. Außerdem werden sie für die verbindungslose Datenübertragung herangezogen. U-Rahmen tragen grundsätzlich keine Sequenznummer („*unnumbered*“).

Alle drei Rahmen haben den gleichen, in Abbildung 4.3 gezeigten Aufbau, unterscheiden sich aber in ihrem 2 Byte langen **Kontrollfeld**. Von diesem benötigt der U-Rahmen nur das erste Byte, das zweite Byte enthält beim U-Rahmen stets 0.

Üblicherweise übernimmt in einem LAN das Transportprotokoll auf OSI-Ebene 4 alle Aufgaben für eine zuverlässige, verbindungsorientierte Datenübertragung, wohingegen die Ebene 2 verbindungslos und ungesichert arbeitet. In diesem, auch hier gegebenen Fall kommt nur der U-Rahmen zum Einsatz. Dessen Kontrollfeld enthält dann den festen Wert 3, was ihn als Träger von *„Unnumbered Information“*, also den Nutzdaten, in seinem **Datenfeld** kennzeichnet. Die Felder **DLSAP** und **SLSAP** bestimmen den Destination- und Source Link Service Access Point des übergeordneten Protokolls, hier je 0xFE für TP4.

Abschließend lässt sich dieser Abschnitt in einem Satz zusammenfassen: Der LLC-Rahmen enthält im vorgefundenen Netz die Bytefolge `0xFE 0xFE 0x03 0x00`, gefolgt von der TP4-Payload.

5 Der Empfangsprozess RECCP

Die eingangsseitige Schnittstelle zwischen dem Netzwerk und den internen Prozessen bildet der Prozess *RECCP („Receive Communication Process"),* der von meinem Kollegen Thorsten Brücher auf Framework-Basis erstellt wurde. Thorstens Lösung für einen schlanken Prozess zum Filtern, Extrahieren und Weiterleiten empfangener TPDUs verhält sich gemäß des Ablaufplans in Abbildung 5.1. RECCP hat eine bewusst einfache Struktur, um einen zügigen Empfang über die RAW-Socket-Schnittstelle zu gewährleisten.

Über einen beim Start im promiscuous mode geöffneten Raw Socket erhält RECCP fortlaufend vollständige IEEE 802.3-Rahmen, auch solche, die nicht für den eigenen Rechner bestimmt sind und / oder keine TP4-Payload enthalten. Letztere können einfach durch Vergleich des LLC-Headers mit der Bytefolge `0xFE 0xFE 0x03 0x00` ausgefiltert werden. Bei Nichtübereinstimmung wird die Bearbeitung abgebrochen.

Für die Ausfilterung von „Fremdrahmen" anhand der Ziel-MAC-Adresse muss RECCP mindestens die MAC-Adresse des simulierten DK-Rechners und die Broadcast-Adresse `FF:FF:FF:FF:FF:FF` akzeptieren. In bestimmten Situationen, etwa bei der parallelen Entwicklung auf mehreren Rechnern, kann es hilfreich sein, weitere Ziel-MAC-Adressen zuzulassen. Daher liest RECCP in der Initialisierungsphase alle erlaubten Ziel-MAC-Adressen aus einer Konfigurationsdatei in eine interne Liste ein. Im laufenden Betrieb werden empfangene Rahmen nur dann weiterverarbeitet, wenn deren Ziel-MAC-Adresse Element dieser Liste ist.

Die weitere Verarbeitung erfolgt in TPDU-spezifischen Behandlungsroutinen, in die anhand des empfangenen TPDU-Codes verzweigt wird. Hier werden die Header-Informationen extrahiert und in eine interne Struktur *`BOT_EMPF_SEND`* umgewandelt. Eventuelle Nutzdaten kopiert RECCP in einen Shared-Memory-Block und weist den Verweis darauf ebenfalls der Struktur `BOT_EMPF_SEND` zu. Abschließend wird diese Struktur via Message Queue an den Prozess *SENDCP* weitergeleitet, der sämtliche weitere TP4-spezifische Verarbeitung, wie Verbindungsüberwachung und –verwaltung, übernimmt.

Die Struktur `BOT_EMPF_SEND` stammt mit leichten Erweiterungen aus dem abgelösten System, wo sie zur internen Übermittlung aller netzwerk-spezifischen Daten zum Einsatz kam. Die im neuen System benutzte Struktur hat den folgenden Aufbau, wobei einige nicht mehr benötigte Felder aus Kompatibilitätsgründen beibehalten wurden:

```
typedef struct _BOT_EMPF_SEND
{
  BOTEMPF_ALLG_TYP  ALLG;     // Struktur mit allg. Informationen (obsolet)
  BYTE         FKBYT;         // Funktionsbyte für übergeordnete Prozesse
  BYTE         LGNR;          // logische Gerätenummer (obsolet)
  WORD         VBNR;          // Hilfs-Verbindungsnummer, meist Kopie von DEST_VBNR
  BYTE         STAT2, STAT1;  // Fehlerstatus
  INT          SYNRM;         // Empfänger für Rückmeldung (obsolet)
  BYTE         TYP,ANZTEI;    // Ergänzungen zum Funktionsbyte
  BYTE         TPDU_CODE;     // Der TPDU-Code
  BYTE         CREDIT;        // Zugeteilter Kredit bei ACK oder Verbindungsaufbau
  WORD         ACK_TIME;      // Acknowledge Time
  WORD         DEST_TSAP;     // Destination TSAP-ID
  WORD         SRC_TSAP;      // Source TSAP-ID     (des Absenders)
  WORD         DEST_VBNR;     // Verbindungsnummer des Zielsystems
  WORD         SRC_VBNR;      // Verbindungsnummer Absender
  INT          SHM_DAT_INDEX; // Datenblock-Index im Shared Memory
  WORD         LEN_DAT;       // Länge der Daten im Shared Memory
  BYTE         TPDU_NR;       // Sequenznummer (TPDU-Nr.)
  BYTE         NEXT_TPDU_NR;  // Nächste erwartete TPDU (bei ACK)
  BYTE         DR_REASON;     // Disconnect Reason
  char         ENDPROZESS[12];// Empfangsprozess-Name, an den SENDCP die TPDU
                              abschließend weiterleiten soll.
  BYTE       SENDERMAC[10];   // MAC-Adresse des Senders (SOURCE Address)
} BOT_EMPF_SEND;
```

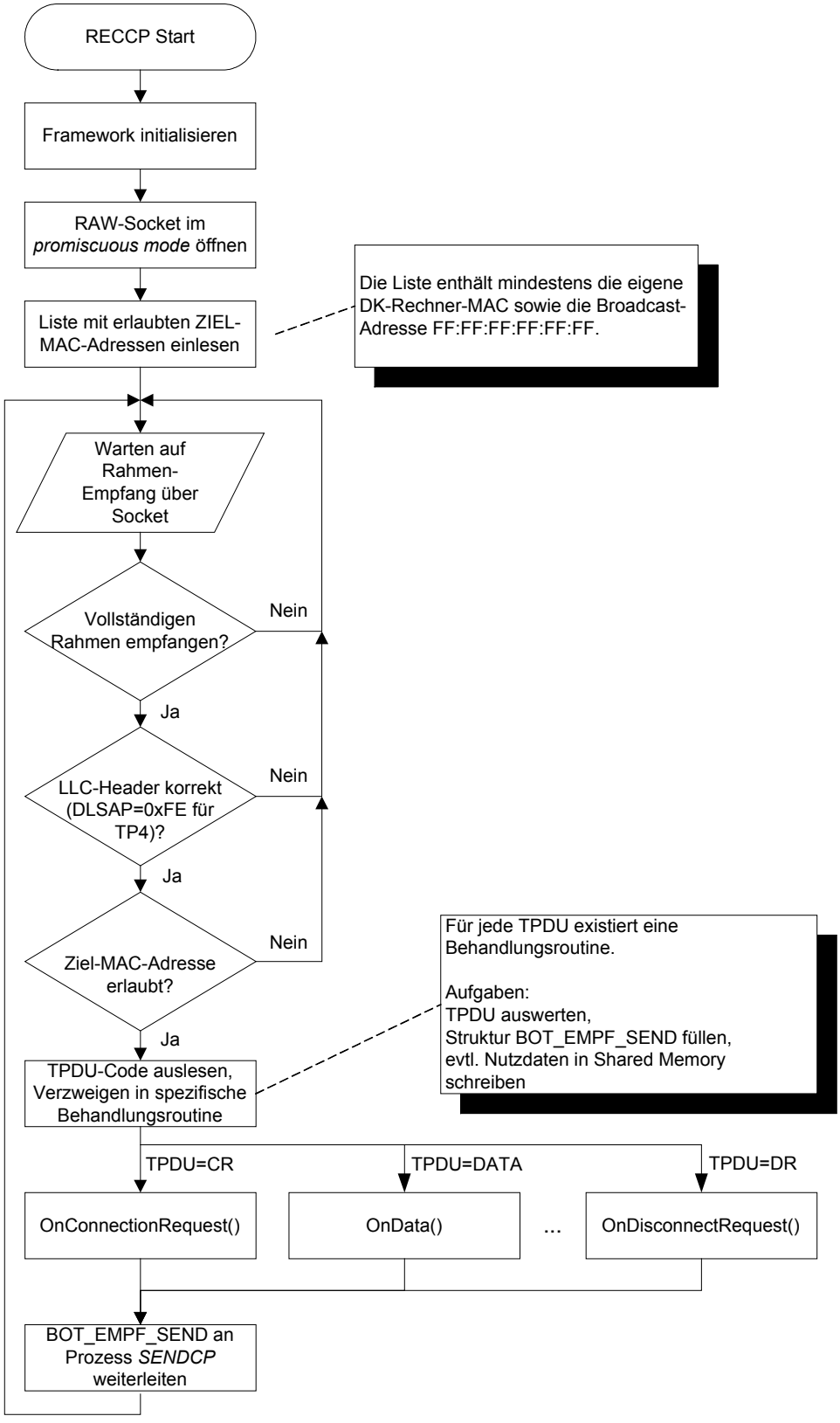


Abbildung 5.1: Ablaufplan des Prozesses RECCP

6 Der Prozess *SENDCP*

6.1 Allgemeines

Zentraler Zweck des Prozesses *SENDCP* ist die Realisierung des gesicherten Transportprotokolls nach ISO/OSI 8073/TP4 gemäß den in den vorangegangenen Kapiteln dargestellten Methoden und Spezifikationen.

Dieses Kapitel beschreibt nach einer Zusammenfassung der zu erfüllenden Aufgaben und Anforderungen zunächst die Aufteilung der Aufgaben auf mehrere Komponenten und Klassen sowie deren statische Abhängigkeiten. Im Anschluss folgt die Schilderung des dynamischen Zusammenspiels der Objekte dieser Klassen im laufenden Betrieb, wobei detailliert auf deren Innenleben zur Umsetzung spezieller Abläufe, wie Timer-Steuerung, Verbindungsüberwachung etc. eingegangen wird.

6.2 Aufgaben und Anforderungen

Aufgaben des Prozesses, kurz zusammengefasst:

- Implementierung der OSI-Schichten 2 und 4, konkret der Protokolle nach IEEE 802 und ISO/OSI 8073/TP4.
- Zentralisierung der Verarbeitungslogik für ein gesichertes Transportprotokoll, wie Verbindungsauf-/-abbau, Datenübertragung und Verbindungsüberwachung mit Timern, Sequenznummern und Acknowledges. Zusätzlich soll ein verbindungsloser Transportdienst angeboten werden.
- Bereitstellung der verfügbaren Dienste an übergeordnete Prozesse, insbesondere dem Kommunikationsabwickler, über Dienstzugangspunkte (Service Access Points).
- Weiterleitung der über das Netz via *RECCP* empfangenen TPDUs an den endgültigen Verarbeitungsprozess.

Dies sind die Aufgaben, die *SENDCP* innerhalb des Gesamtsystems zu übernehmen hat. Zu ihrer Erfüllung wurden an die Software des Prozesses einige Anforderungen gestellt, die beim Entwurf zu berücksichtigen waren. Die folgende Auflistung fasst die Vorgaben zusammen:

Design-Anforderungen an die Software des Prozesses:

- **Robust und fehlertolerant**
  Etwaige Probleme auf einer Verbindung dürfen sich nicht auf andere Verbindungen auswirken. Insbesondere bei einer blockierten Kommunikation darf die Behandlung anderer Verbindungen nicht verzögert werden. Auf in einem bestimmten Verbindungszustand unerwartet empfangene Telegramme (etwa ein ACK, obwohl keine Daten gesendet wurden) wird tolerant reagiert. Verbindungsabbruch erfolgt erst bei nicht behebbaren Fehlern.

- **Software-Erstellung auf Basis des Applikations-Frameworks**
- **Weitestgehend portier- und wiederverwendbar**
  Der Netzwerk-Protokollteil sollte als weitgehend eigenständige Komponente realisiert werden, so dass dessen Nutzung ohne großen Aufwand auch ohne Framework und unter anderen Betriebssystemen, wie Microsoft Windows, möglich ist. Dies kann Grundlage für mögliche zukünftige Anwendungen in Nachfolgeprojekten sein.
- **Trennung der OSI-Schichten**
  Die durch SENDCP realisierten OSI-Schichten 2 und 4 sind in der Implementierung sauber voneinander zu trennen. Ferner soll der Prozess unabhängig von übergeordneten Prozessen arbeiten.
- **Dienste von allen übergeordneten Prozessen nutzbar**
  Die durch SENDCP angebotenen Dienste sollen für übergeordnete Prozesse auf eine einheitliche Art über definierte Dienstzugänge nutzbar sein.
- **So flexibel wie nötig, so einfach wie möglich**
  Das vorgefundene IT-System benutzt die in Kapitel 3.6.2 beschriebene Untermenge der durch ISO/OSI 8073 vorgesehenen Möglichkeiten. SENDCP kann und soll sich auf diese Untermenge beschränken. Daher sind einige Vereinfachungen möglich:
  - Beschränkung auf sechs von zehn durch OSI definierte TPDU-Typen
  - Prüfsummenberechnung bzw. Auswertung deaktiviert
  - Keine Unterstützung von QoS-Parametern sowie weiterer, im abgelösten System nicht benutzte Parameter im variablen Teil der TP4-TPDUs
  - Festlegung der Verbindungsparameter auf die im abgelösten System vorgefundenen Werte:

Eingeräumter Sendekredit	8
Acknowledge Time	500ms
TPDU-Größe	1024 Bytes

Tabelle 6.1: Festgesetzte Verbindungsparameter

Bei den genannten festgelegten Parametern handelt es sich um beim Verbindungsaufbau von den beiden Kommunikationspartnern ausgetauschte Werte. Im neuen System sind diese per Konstanten-Definition im Code auf die im abgelösten System vorgefundenen Werte festgesetzt. Nicht vorgefundene variable Parameter werden nicht unterstützt. Diese Maßnahmen vereinfachen insbesondere die Dienstzugangs-Schnittstellenfunktionen für die Anwender-Prozesse, da sich die zu berücksichtigenden Parameter auf ein Minimum reduzieren.

Nicht Bestandteil des Aufgabenkatalogs ist die Segmentierung von großen Datenmengen in einzelne Segmente mit TPDU-gerechten Größen sowie umgekehrt deren Reassemblierung. Dies wird von den übergeordneten Schichten übernommen. SENDCP tauscht mit diesen jeweils maximal 1024 Bytes große Nutzdaten-Segmente aus.

6.3 Die Komponenten des Prozesses SENDCP

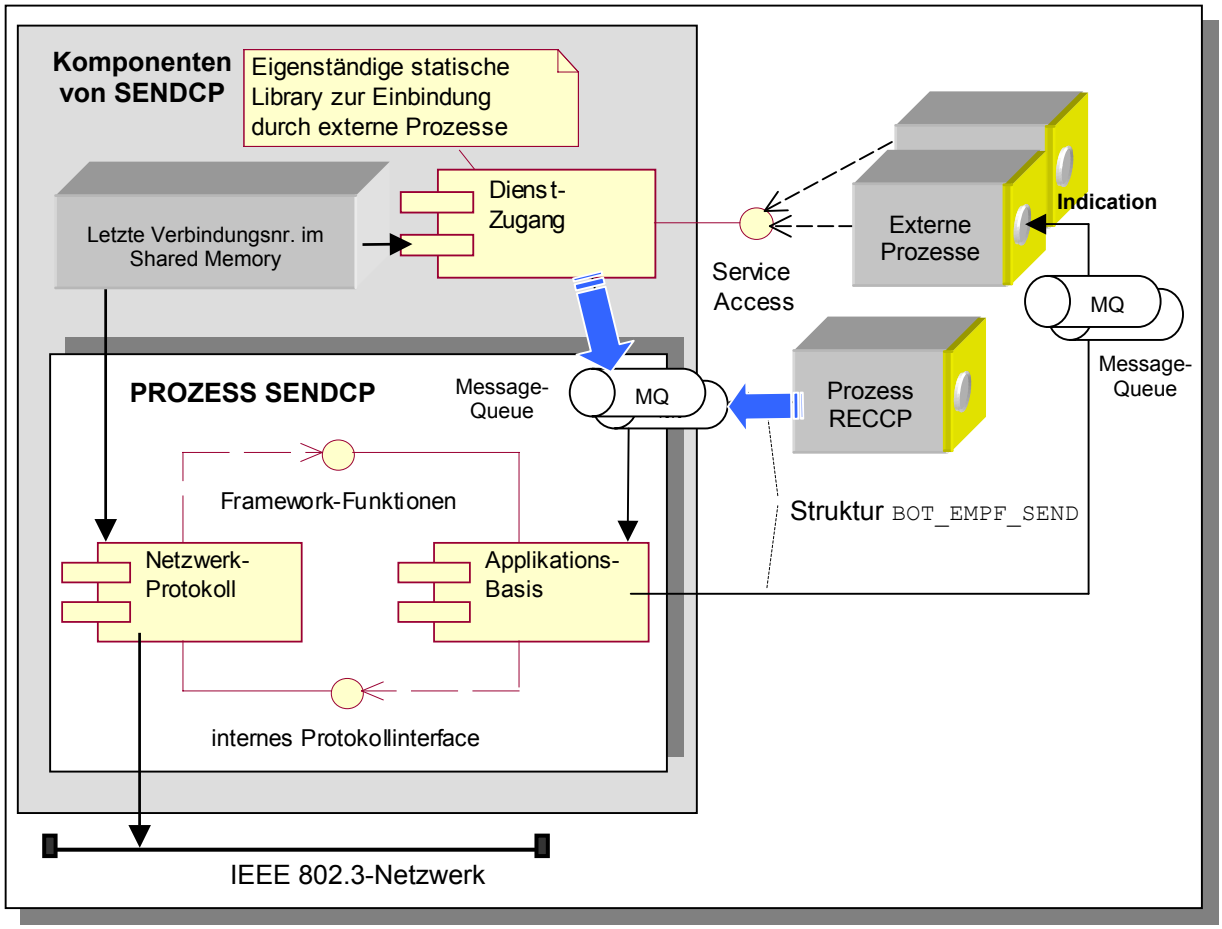


Abbildung 6.1: Die Komponenten des Prozesses SENDCP

Zur Bewältigung der komplexen Aufgaben und Anforderungen wurde SENDCP in die drei in Abbildung 6.1 gezeigten Software-Komponenten aufgeteilt:

- Die Komponente **Dienstzugang** ermöglicht externen Prozessen (hier jenen des Kommunikationsabwicklers) den Zugang zu den von SENDCP angebotenen Diensten. Dazu wird die Komponente zu einer eigenständigen C++-Library compiliert, die externe Prozesse hinzu binden können. Die Komponente gehört somit zwar zum Software-Projekt *SENDCP*, geht aber nicht in den ausführbaren SENDCP-Prozess ein, sondern wird Bestandteil der einbindenden, externen Prozesse.
- Die gesamte IEEE802- und TP4-Protokoll-Logik ist in der Komponente **Netzwerk-Protokoll** zusammengefasst. Der Zugriff erfolgt über das interne Protokoll-Interface. Dieses wird von der Komponente *Applikationsbasis* zum Aufruf der angebotenen öffentlichen Kommunikations- und Steuermethoden angesprochen.
- Alle mit dem von SENDCP benutzten Applikationsframework in Zusammenhang stehenden Software-Elemente enthält die Komponente **Applikationsbasis**. Insbesondere auf die dadurch bereitgestellten Log-Funktionalitäten sowie den Message Queue-Mechanismus greift die Netzwerkprotokoll-Komponente zu.

6.4 Die Komponente *Dienstzugang*

Die Interprozesskommunikation verläuft zwischen allen Prozessen des erstellten Systems über Message Queues. Auch der Prozess SENDCP bildet da keine Ausnahme. Insofern wäre es möglich, für jedes von SENDCP bereitgestellte Dienstelement, etwa Verbindungsaufbau und Datenübertragung, ein spezielles Message Queue-Telegramm zu spezifizieren. Ein Prozess, der diese Dienstelemente nutzen wollte, müsste dann das zugehörige Telegramm erstellen, über die Queue an SENDCP senden und danach auf die Antwort in der eigenen Queue warten, die SENDCP nach getaner Arbeit zurücksenden würde.

Dieses „Low Level"-Vorgehen ist allerdings in mehrerlei Hinsicht unelegant:

1. Jedem Prozess selbst das Zusammenbauen bzw. Auswerten der Message Queue-Telegramme zu überlassen ist aufwändig und fehlerträchtig
2. Der Zugriffsmechanismus auf den Prozess SENDCP ist nicht gekapselt: Die Art und Weise der Ankopplung sind Implementierungsdetails von SENDCP und sollten vor anderen Prozessen verborgen werden.
3. Als Folge von Punkt 2 kann der Zugriffsmechanismus nicht einfach ausgetauscht werden. Ändert sich der Mechanismus, erfordert dies Eingriffe in alle beteiligten Prozesse. Dies trifft auch schon zu, wenn sich, beispielsweise bei Erweiterungen, nur das intern benutzte Queue-Telegrammformat ändert.
4. Anwendungsentwickler sind API-Aufrufe für Kommunikationsaufgaben gewohnt, wie sie etwa die bekannten Funktionsaufrufe für die Benutzung von Sockets darstellen. Auch das abgelöste Altsystem benutzt eine solche Programmierschnittstelle zum Zugriff auf die Transportprotokoll-Dienste. Zwecks einfacher Portierbarkeit der übernommenen Software-Komponenten sollte die Schnittstelle zum TP4-Protokoll weitgehend mit derjenigen des abgelösten Systems identisch sein.

Ziel der Komponente **Dienstzugang** ist es demnach, externen Prozessen den Zugang zu den Transportprotokolldiensten von SENDCP zu ermöglichen, ohne mit den obigen Schwierigkeiten konfrontiert zu werden. Der genannte Punkt 4 deutet an, wie eine Lösung aussehen kann: Die Komponente *Dienstzugang* bietet eine Programmierschnittstelle für den Dienstzugang *(Service Access*, siehe Abbildung 6.1*)*, die externen Prozessen Methoden zum Verbindungsauf- und –abbau und zum Datenversand bereitstellt.

6.4.1 Die Klasse *CTP4SAP*

Softwaretechnisch ist diese Programmierschnittstelle durch die Methoden der Klasse *CTP4SAP* (für *„TP4 Service Access Point"*) realisiert. *CTP4SAP* ist alleiniger Bestandteil der Komponente *Dienstzugang* und kann von externen Prozessen instanziiert werden. Möglich wird dies durch die oben erwähnte Auslegung der Komponente *Dienstzugang* als statische C++-Library. Der zugehörige Programmcode ist Bestandteil des Software-Projektes SENDCP und wird bei jeder SENDCP-Compilierung mit übersetzt. Die daraus resultierende Objektdatei *ctp4sap.o* lässt sich durch das Linux-Kommando `ar -rs libTP4sap.a ctp4sap.o` umwandeln in die Lib-Datei *libTP4sap.a*, welche nebst der zugehörigen Headerdatei *ctp4sap.h* in externe Prozesse übernommen und dort benutzt werden kann. Angelegte Objekte von *CTP4SAP* befinden sich also in den externen Prozessen.

In jeder der nachfolgend vorgestellten *CTP4SAP*-Dienstzugangs-Methoden wird ein spezifisches Objekt von *CTelegram* (siehe Kapitel 2.3.3, *„Die Telegramm-Klasse CTelegram“,* Seite 27) angelegt, mit den zugehörigen Steuer- und ggf. Nutzdaten versorgt und via Message Queue an den Prozess SENDCP gesendet. Dort erfolgt die weitere Verarbeitung.

Der gesamte Kopplungsmechanismus ist auf diese Art innerhalb der *CTP4SAP*-Methoden gekapselt und für den Anwender der Dienstzugangs-Komponente transparent. Der zugehörige Programmcode wird an zentraler Stelle innerhalb des SENDCP-Projektes gepflegt. Änderungen gehen nach Erstellen der Dienstzugangs-Library durch einfaches Neukompilieren der einbindenden Prozesse in diese ein.

So weit, so gut, allerdings ist der geschilderte Weg eine Einbahnstrasse: Externe Prozesse können zwar unter Benutzung der Dienstzugangs-Komponente den Prozess SENDCP zu Aktivitäten veranlassen und Daten an entfernte Rechner senden, Antworten kommen über diesen Weg jedoch nicht zurück. Wollte man zum Empfang eine Methode in *CTP4SAP* bereitstellen, ließen sich anstehende Daten nur durch Polling, also zyklisches Aufrufen dieser Methode, erkennen und entgegennehmen.

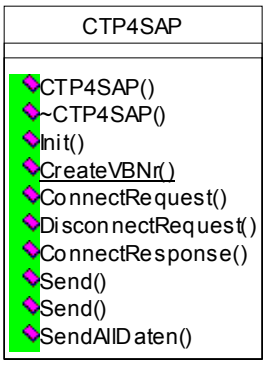


Abb. 6.2: Die Klasse CTP4SAP

Weil alle relevanten Prozesse im System aber auf dem Applikations-Framework basieren, bringt jeder externe Prozess seine eigene Message Queue mit. Der Prozess SENDCP wird empfangene Telegramme also in die Queue des anzusprechenden Prozesses schreiben, was ereignisgesteuertes Reagieren auf diese Telegramme ermöglicht. Die OSI-Nomenklatur bezeichnet ein solches Ereignis auch als *Indication.*

In Empfangsrichtung werden alle Daten in Form der bereits durch den Prozess *RECCP* angelegten Struktur `BOT_EMPF_SEND` (siehe Kapitel 5) bis zum Endprozess durchgereicht. Zwar kommen durch diese Vorgehensweise insbesondere die oben unter den Punkten 1 und 3 aufgeführten Nachteile zum Tragen, allerdings ist dies in Empfangsrichtung kein großes Manko, da bereits das Altsystem die Struktur `BOT_EMPF_SEND` verwendete und die Auswerteroutinen dazu übernommen wurden.

Die OSI-Nomenklatur sieht für die Dienstelemente eines Netzwerk-Protokolls die Bezeichnungen *Request, Indication, Response* und *Confirm* vor, wie es in den Abbildungen zu Kapitel 3.5 angedeutet wurde. Die Software des abgelösten Systems hat sich nicht konsequent an diese OSI-Vorgaben gehalten, sondern benutzt die Funktionsbezeichnungen der nun folgenden Übersicht. Zwecks Erreichens von Aufrufkompatibilität der portierten Software wurden die alten Funktionsnamen für die Methoden der Klasse *CTP4SAP* übernommen. Noch einmal zur Erinnerung: Die folgenden Methoden machen sich externe Prozesse zunutze und werden in diesen aufgerufen.

- **ConnectRequest()** veranlasst den Aufbau einer TP4-Verbindung. Die Methode erwartet die Kennung des zu verbindenden Arbeitsplatzes. Aus dieser ermittelt *ConnectRequest()* intern die zugehörige MAC-Adresse und TSAP-ID, wie in Kapitel 3.9, *„Adress-Auflösung“*, Seite 53, beschrieben. Außerdem wird intern eine neue Verbindungsnummer für die aufzubauende Verbindung vergeben. Zusammen mit der eigenen TSAP-ID gelangen diese Daten via Message Queue zu *SENDCP*. Optional kann ein Nutzdatenstring als Parameter übergeben werden, der im Datenteil der *Connection Request*-TPDU an die Gegenseite übertragen wird. Rückgabewert ist die vergebene lokale Verbindungsnummer, die die folgenden Methoden benötigen.
- **Send()** dient erwartungsgemäß zum Versenden von Nutzdaten über eine bestehende Verbindung, deren lokale Verbindungsnummer als Parameter zu übergeben ist. Weitere Parameter sind der Pointer auf den zu sendenden Datenblock sowie dessen Länge. *Send()* existiert zusätzlich in einer zweiten Version mit einer internen Datenstruktur als Parameter, wodurch Schnittstellen-Kompatibilität zur portierten Altsoftware gewährleistet ist. Intern ruft diese zweite Version die erstgenannte Methode auf.
- **DisconnectRequest()** trennt die Verbindung mit der zu übergebenden Verbindungsnummer. Zusätzlich ist der *Disconnect Reason* (siehe Abschnitt 3.6.2.1) zu übergeben.
- **ConnectResponse()** kann aufgerufen werden, wenn von einem entfernten Arbeitsplatz ein Verbindungswunsch via *Connection Request* einging. *ConnectResponse()* löst das Senden eines *Connection Confirm* an die Gegenseite aus, wonach die Verbindung angenommen ist. Die Methode benötigt lediglich die zugehörige lokale Verbindungsnummer. Diese wurde bereits bei der Vorauswertung des *Connection Request* durch den Prozess SENDCP vergeben und als Element der Struktur `BOT_EMPF_SEND` innerhalb des *Connect Indication*-Telegramms mitgeteilt. Gemäß Kapitel 3.5 kann ein Prozess einen Verbindungswunsch auch ablehnen. Anstelle von *ConnectResponse()* wäre dann *DisconnectRequest()* aufzurufen.
- **SendAllDaten()** ist die Schnittstelle für den verbindungslosen Transportdienst. Die Methode versendet einen Datenblock gegebener Länge ohne vorherigen Verbindungsaufbau an einen Arbeitsplatz, dessen Kennung als Parameter anzugeben ist. Die Adressauflösung erfolgt wie in Kapitel 3.9 beschrieben.
- **Init()** ist keine Dienstzugangsmethode, sondern muss einmalig vor der Benutzung der obigen Methoden aufgerufen werden. Der Funktion ist der Pointer auf das Framework-Applikationsobjekt (in diesem Dokument *CMyKomApp* genannt) zu übergeben. *CTP4SAP* benötigt diesen Pointer zum Zugriff auf die Framework-Funktionen für Logging und den Telegrammversand über die Message Queue.

Die jede TP4-Verbindung identifizierende Verbindungsnummer wird implizit in *ConnectRequest()* mit Hilfe der Methode *CreateVBNr()* generiert. Sie berechnet die neue Verbindungsnummer gemäß

$$NeueVBNr = (letzteVBNr + 1) \bmod 0x5000 + 0x2000 .$$

Verbindungsnummern liegen somit im Bereich 0x2000..0x6999. Die Wahl des Wertebereichs erfolgte willkürlich, allerdings musste gewährleistet sein, dass zu keiner Zeit identische Verbindungsnummern für verschiedene Verbindungen existieren. Mit maximal 50 existierenden Arbeitsplätzen im Gesamtsystem kann dieser Fall beim gewählten Wertebereich ausgeschlossen werden.

Wie eingangs erwähnt, ist die Komponente *Dienstzugang* eine statische Library, die mehrere Prozesse gleichzeitig einbinden. Folglich legt jeder dieser Prozesse im laufenden Betrieb ein eigenes, unabhängiges Objekt der Klasse *CTP4SAP* an. Dies erfordert eine weitere Vorkehrung für die Sicherstellung der systemweiten Eindeutigkeit aller vergebenen Verbindungsnummern: Die zuletzt vergebene Verbindungsnummer wird zentral in einem Shared-Memory-Bereich abgelegt und bei jedem Aufruf von *CreateVBNr()* inkrementiert. Den konkurrierenden Zugriff aus verschiedenen Prozessen koordiniert eine Semaphore.

CreateVBNr() ist als statische Klassenmethode angelegt, um Verbindungsnummern auch ohne das Anlegen eines Objektes von *CTP4SAP* erzeugen zu können. Erforderlich ist dies innerhalb des Prozesses SENDCP, der natürlich keinen Dienstzugang auf sich selbst benötigt und infolgedessen kein Objekt von *CTP4SAP* erzeugt. Trotzdem muss SENDCP beim Empfang einer *Connection Request*-TPDU von einem entfernten Arbeitsplatz in der Lage sein, unmittelbar eine eindeutige lokale Verbindungsnummer für diese neu aufzubauende Verbindung zu vergeben. Mit *CreateVBNr()* als statischer Klassenmethode ist genau dies möglich. Danach kann SENDCP das empfangene *Connection Request* in Form der Struktur `BOT_EMPF_SEND` mitsamt der neuen Verbindungsnummer als *Connect Indication*-Telegramm in die Message Queue des endgültigen Verarbeitungsprozesses einstellen.

6.5 Die Komponente Netzwerk-Protokoll

6.5.1 Allgemeines

Die Forderung nach weitgehender Portier- und Wiederverwendbarkeit der Netzwerkprotokoll-Implementierung setzt voraus, dass dieser Software-Teil möglichst keine betriebssystem- oder framework-spezifischen Elemente enthält. Erreicht wird dies durch Kapselung der gesamten IEEE 802- und TP4-Implementierung in der Komponente *Netzwerk-Protokoll*. Innerhalb dieser Komponente laufen nahezu alle dynamischen Abläufe des Prozesses SENDCP ab. Somit bildet sie mit ihren Klassen dessen eigentliches Innenleben. Schritt für Schritt beschreiben die folgenden Abschnitte die Evolution und Funktion der Komponente, zunächst mit einem Überblick über die beteiligten Klassen. Das darauf folgende Kapitel beschreibt dann die Klasseninteraktionen zur Realisation des dynamischen Verhaltens. Verschiedene Aspekte der Klassen werden jeweils im entsprechenden Zusammenhang vertieft. Zur besseren Übersichtlichkeit sind die zunächst vorgestellten Einzelklassen-Diagramme in Auszügen dargestellt. Das vollständige Klassendiagramm zeigt Abbildung 6.10.

6.5.2 Software-Design: Die Evolution des Klassenmodells

6.5.2.1 Klassen für OSI-Schicht 4

Oftmals basiert Software-Design auf den drei Zutaten Erfahrung, Intuition und Beobachtung. Eine Mischung aus diesen dreien brachte auch die Entwicklung des Klassenmodells für den Prozess SENDCP zum Ziel.

Am Anfang stand die Beobachtung, in diesem Fall die Untersuchung des zu modellierenden TP4-Protokolls. Dieses arbeitet *verbindungsorientiert*, und jede Verbindung hat charakteristische Parameter. Ohne viel Intuition fand sich so die erste Klasse, ***CConnection***, deren Objekte in ihren Membervariablen je eine Verbindung mit den zugehörigen Parametern abbilden. Dazu gehört die Zustandssteuerung nach Abschnitt 6.6.2, auf dessen Zustandsdiagramme bereits hier hingewiesen sein soll.

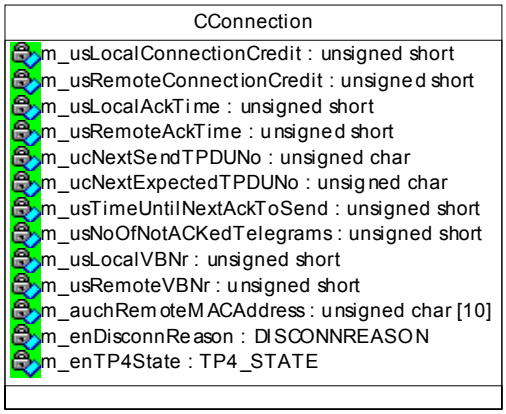


Abbildung 6.3: Die Klasse CConnection

Über eine etablierte TP4-Verbindung werden TP4-Telegramme gesendet. Konsequenz ist deren Verwaltung in Objekten der Klasse ***CTP4Telegram***. Ein solches Objekt speichert zum einen Header und Payload genau einer TPDU, zum anderen zugehörige Informationen, wie deren Sequenz- (TPDU-) Nummer, die verbleibende Anzahl Wiederholversuche und die aktuelle Wartezeit bis zum Sende-Timeout.

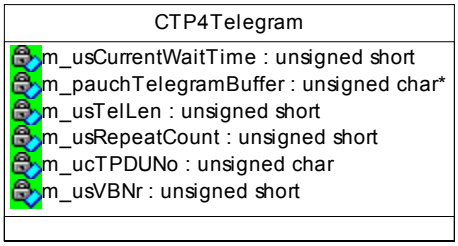


Abbildung 6.4: Die Klasse CTP4Telegram

Entsprechend den Ausführungen in Kapitel 3.3 bilden alle gesendeten, aber noch nicht bestätigten TP4-Telegramme die *Sendefensterliste.* Es bedarf also einer Container-Klasse, die in der Lage ist, Objekte von *CTP4Telegram* in Form einer linearen Liste zu verwalten. Neu gesendete Telegramme werden an diese Liste hinten angehängt, wohingegen per *Acknowledge*-TPDU bestätigte Telegramme vorne aus der Liste entfernt werden. Dazu dient die Klasse ***CTelegramList***.

Die Sendefensterliste ist allerdings nicht die einzige benötigte TP4-Telegrammliste: Ist das Sendefenster voll (der Sendekredit ist also aufgebraucht), soll SENDCP darüber hinaus zum Versand anstehende TP4-Telegramme in einer zweiten Liste, der *Warteliste (Pending List)* zwischenspeichern. Bei Freiwerden des Sendefensters müssen entsprechend viele TP4-Telegramme aus der Pending List in die Sendefensterliste verschoben und gleichzeitig versendet werden. Auch für die Pending List kommt *CTelegramList* zum Einsatz.

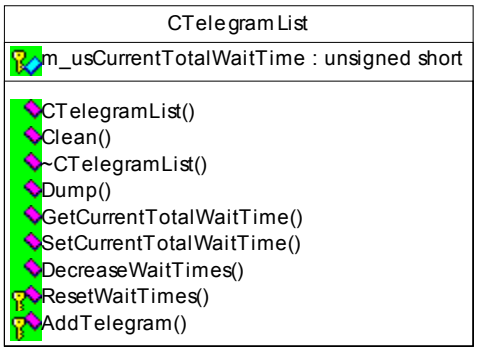


Abbildung 6.5: Die Klasse CTelegramList

Die typischen Listenoperationen, wie das Ein- und Ausketten von Elementen sowie das Iterieren über die Liste, stehen innerhalb der Standard Template Library (STL) in Form der Template-Klasse *list* zur Verfügung. In [Stroustrup00] sind sie dokumentiert. Die Klasse *CTelegramList* macht sich die von der *STL::list*-Klasse gebotenen Funktionalitäten zunutze und ergänzt sie um einige spezifische Eigenschaften, wie die zu Debug-Zwecken hilfreiche *Dump()*-Funktion zur Log-Ausgabe des gesamten Listeninhalts sowie Methoden für die in Abschnitt 6.6.1 behandelte Timeout-Verwaltung.

Mit den bis hierhin vorgestellten Klassen lässt sich nun ein Ausschnitt des SENDCP-Gesamt-Klassendiagramms entwerfen.

6.5.2.2 Erster Teilentwurf: Listen in einer *hat eine*-Beziehung

Der erste Teilentwurf orientierte sich an der in Kapitel 2.3.2.1, *„Die Klasse CKomPartners“*, Seite 26, vorgestellten Struktur der Klasse *CKomPartners.*

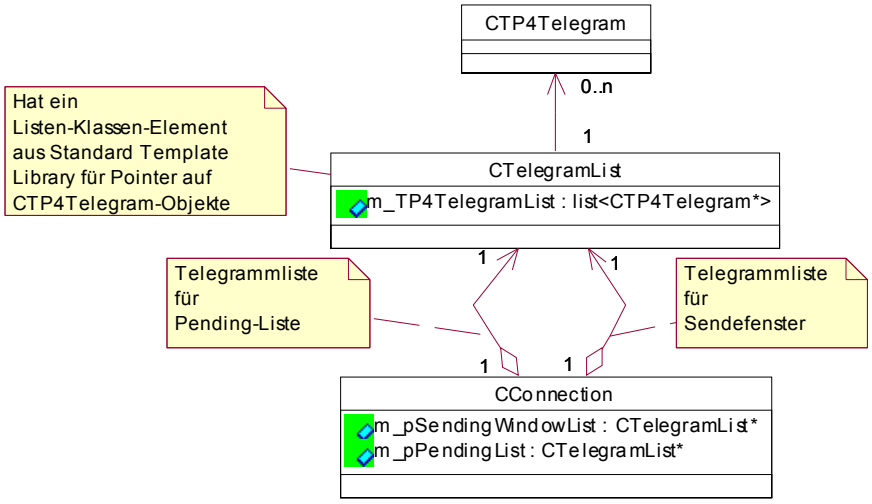


Abbildung 6.6: Erster Teilentwurf des SENDCP-Klassendiagramms

Hat besagte Klasse ein Element der STL-Klasse *map*, so sollte der erste Entwurf der Klasse *CTelegramList* ein Element der STL-Klasse *list* haben, in welchem die *CTP4Telegram*-Objekte eingekettet würden. Analog hätte die Klasse *CConnection* je einen Pointer auf ein *CTelegramList*-Objekt für die Sendefensterliste und die Pending-Liste gehabt. Abbildung 6.6 zeigt diesen ersten Entwurf.

Spätestens bei der Implementierung offenbart die gewählte *hat eine*-Beziehung zwischen den Klassen jedoch einen Nachteil, wenn aus einem externen Codeabschnitt der Zugriff auf Methoden des STL::list-Elements erforderlich ist. Demonstriert sei dies anhand des Anhängens eines neuen *CTP4Telegram*-Objekts an die Sendefensterliste mit Hilfe der STL::list-Methode *push_back()*. Dies führt zu folgendem Code, wobei *pTP4Telegram* ein Pointer auf ein *CTP4Telegram-Objekt* und *pConnection* der Pointer auf das *CConnection*-Objekt sei, an dessen Sendefensterliste das *CTP4Telegram*-Objekt anzuhängen ist:

```
// Der Code befinde sich außerhalb der CConnection-Klasse
pConnection->m_pSendingWindowList->m_TP4TelegramList.push_back(pTP4Telegram);
```

Die erforderliche Pointer-Verkettung macht den Code nicht nur unleserlich, sondern auch fehlerträchtig und aufwändig. Dies insbesondere, da neben den STL::list-Methoden zum Ein- und Ausketten auf eine Vielzahl weiterer Methoden, etwa zum Iterieren über die gesamte Liste oder zur Längenabfrage, zugegriffen wird. Dies führte zum zweiten Teilentwurf:

6.5.2.3 Zweiter Teilentwurf: Eine Verbindung *ist eine* Liste

Teilentwurf zwei führt zu einem dramatisch einfacheren Zugriff auf die Listenfunktionen, indem Vererbung eingesetzt und die *hat eine*-Beziehung zwischen den Klassen durch eine *ist eine*-Beziehung ausgetauscht wird. *CConnection* leitet sich in diesem Entwurf von *CTelegramList* ab, welche wiederum alle Listen-Operationen durch Ableitung von STL::list erbt.

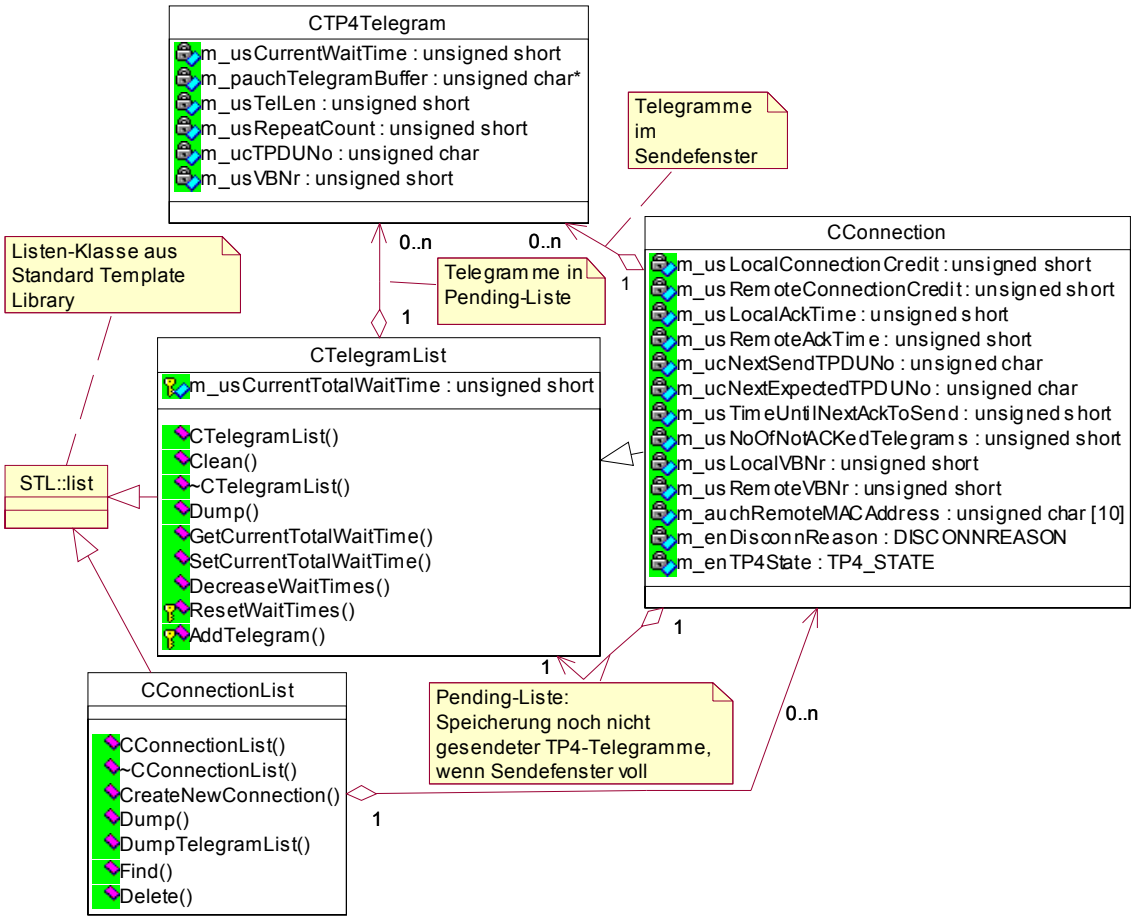


Abbildung 6.7: Realisierter, zweiter Teilentwurf des SENDCP-Klassendiagramms

Durch diesen Entwurf vereinfacht sich das obige Codebeispiel zu der Zeile:

```
pConnection->push_back(pTP4Telegram);
```

Zusammengefasst *ist* jedes *CConnection*-Objekt gleichzeitig eine Sendefensterliste. Nach wie vor existiert daneben eine *hat eine*-Beziehung zwischen *CConnection* und *CTelegramList* für die Pending-Liste. Die genannten Gegenargumente fallen hier nicht ins Gewicht, da im Falle der Pending-Liste tatsächlich nur *CTP4Telegram*-Objekte hinten angefügt und vorne entfernt werden. Dies lässt sich ohne großen Aufwand in zwei *CConnection*-Memberfunktionen *PopFirstPendingTel()* und *AddTelToPendingList()* kapseln, die im Gesamt-Klassendiagramm in Abbildung 6.10 eingeblendet sind.

Selbstverständlich kann SENDCP nicht nur eine, sondern prinzipiell beliebig viele parallel bestehende Verbindungen verwalten. Zu diesem Zweck wurde im Entwurf nach Abbildung 6.7 die Klasse ***CConnectionList*** neu hinzugefügt. Nach dem oben beschriebenen Muster ist auch *CConnectionList* von der STL::list-Klasse abgeleitet. *CConnectionList* verwaltet Pointer auf alle Objekte der Klasse *CConnection* und bildet so eine *Verbindungsliste.*

Alle *CConnection*-Objekte speichern in ihrer Membervariablen *m_usLocalVBNr* die lokal vergebene, eindeutige Verbindungsnummer der repräsentierten Verbindung. Die Methode *CConnectionList::Find()* liefert anhand dieser Verbindungsnummer den Pointer auf das zugehörige *CConnection*-Objekt zurück. Darüber ist danach der Direktzugriff auf das *CConnection*-Objekt und dessen Verbindungsverwaltungs-Methoden möglich.

CConnection-Objekte werden beim Verbindungsaufbau mit der *CConnectionList*-Fabrikmethode *CreateNewConnection()* erzeugt und implizit an die Verbindungsliste angehangen. Dabei werden die Variablen für lokalen Verbindungskredit und Acknowledge Time gemäß Abschnitt 6.2 vorbelegt. Umgekehrt wird nach vollständigem Verbindungsabbau über *CConnectionList::Delete()* sowohl das eigentliche *CConnection*-Objekt als auch der Verweis in der Verbindungsliste gelöscht. Beide Funktionen erwarten jeweils die lokale Verbindungsnummer als Parameter, *CreateNewConnection()* zusätzlich die MAC-Adresse des entfernten Verbindungspartners.

6.5.2.4 OSI-Schicht 2 und die Klasse *CEthernetLayer*

Die bis hierhin eingeführten Klassen dienen der Realisierung des TP4-Transportprotokolls, also der OSI-Schicht 4. Alle die OSI-Schicht 2 betreffenden Funktionalitäten sind in der Klasse *CEthernetLayer* gekapselt. Ihre Methoden *Startup()* und *Shutdown()* öffnen bzw. schließen einen Raw Socket, über den mit *SendEthernetTel()* ein vollständiges Schicht-2-Telegramm versendet werden kann.

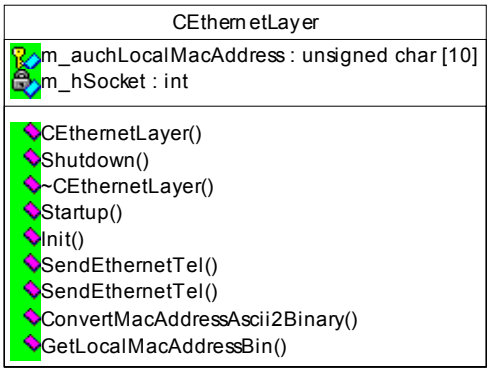


Abbildung 6.8: Die Klasse CEthernetLayer

SendEthernetTel() erwartet als Parameter die MAC-Adresse des Empfängers sowie die Speicheradresse und Länge der zu übertragenden Nutzlast, hier eine TP4-TPDU. Vor der Übergabe an den Raw Socket wird der Nutzlast ein IEEE 802.3-Header vorangestellt. Die beiden Versionen der Methode akzeptieren die Empfänger-MAC einmal als ASCII-String (AA:BB:CC:DD:EE:FF), zum anderen in einer 6-Bytes-Binärform.

Zur Komplettierung des IEEE 802.3-Headers benötigt *SendEthernetTel()* ferner die Absender-MAC-Adresse, also jene des eigenen Rechners. Diese ist bei der Initialisierung des *CEthernetLayer*-Objekts über *Init()* zu übergeben. Für den Zugriff auf die Logger-Funktionen des Frameworks erwartet *Init()* außerdem den Pointer auf das Framework-Applikationsobjekt. Nach der Initialisierung kann die eigene MAC-Adresse über *GetLocalMacAddressBin()* im Binärformat abgefragt werden.

Da die Klasse *CEthernetLayer* keine Ethernet- sondern IEEE 802.3-Rahmen zusammenstellt und versendet, ist der Klassenname gemäß den Ausführungen in Kapitel 4.3, *„Data Link Control“,* Seite 57, unpräzise. Aufgrund der Gängigkeit des Begriffs „Ethernet“ wurde dennoch dieser Name gewählt, um Sinn und Zweck der Klasse auch Außenstehenden schnell erkennbar zu machen.

CEthernetLayer ist mit seinem Raw Socket-Zugriff an deren Betriebssystem-Implementierung gebunden. Dadurch werden bei einer Portierung der Komponente *Netzwerk-Protokoll* gegebenenfalls Anpassungen erforderlich, die allerdings aufgrund der Kapselung in *CEthernetLayer* lokal begrenzt und einfach durchzuführen sind.

6.5.2.5 Die Klasse *CTP4Layer*

Zu guter Letzt bedarf es einer Klasse, welche die kleine SENDCP-Klassenwelt im Innersten zusammenhält, alle Abläufe koordiniert und als Schnittstelle der Komponente *Netzwerk-Protokoll* zur Außenwelt fungiert. Diese Klasse ist *CTP4Layer.*

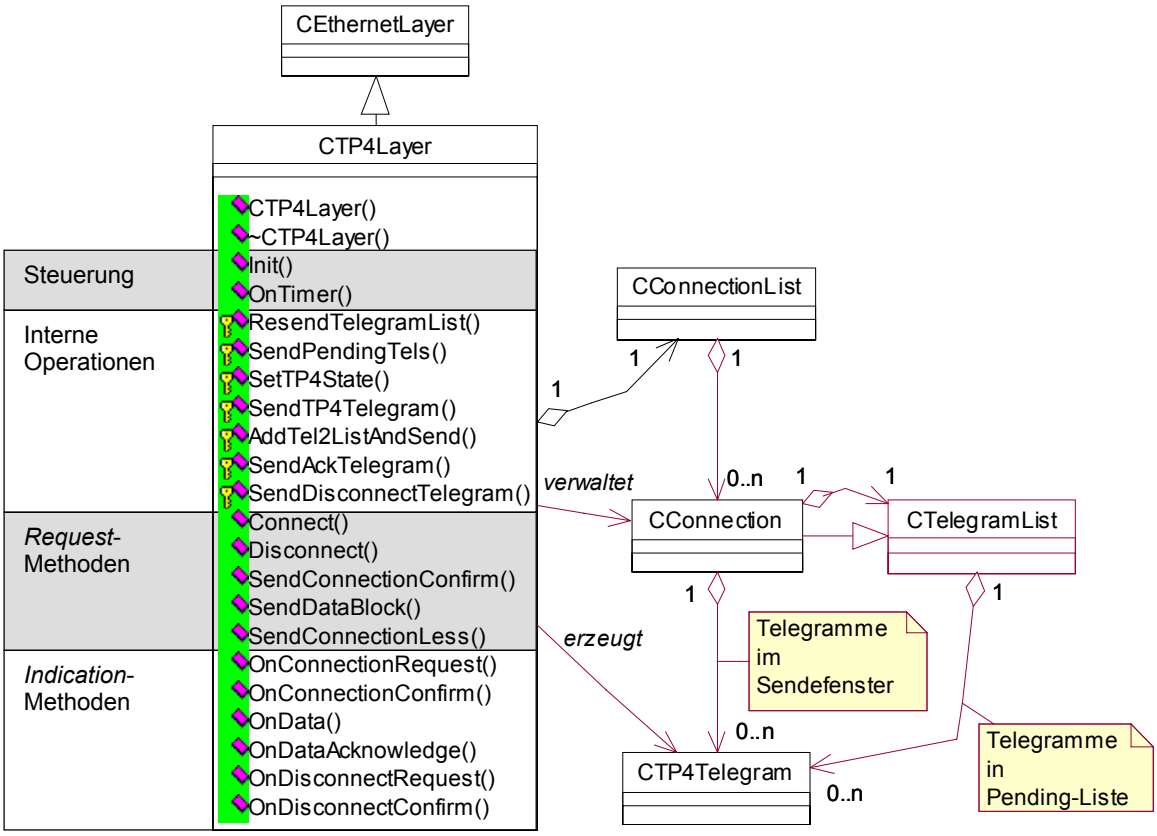


Abbildung 6.9: Die Klasse CTP4Layer

CTP4Layer ist abgeleitet von *CEthernetLayer*. Auf diese Art ist der Zugriff auf die durch letztere Klasse angebotenen Schicht-2-Dienste zum Telegrammversand möglich. Gleichzeitig wird durch die Kapselung in getrennte Klassen die durch das OSI-Konzept angestrebte Trennung der Netzwerk-Schichten erreicht.

Über die von *CTP4Layer* bereitgestellten Methoden werden alle zum Verbindungsauf- und –abbau, zum Datentransport und zur Verbindungsverwaltung mit Timern und Acknowledges benötigten Abläufe ausgelöst und gesteuert. Bei Bedarf werden dabei neue Objekte von *CConnection* und *CTP4Telegram* angelegt.

Die zuständigen Methoden lassen sich in die vier in Abbildung 6.9 gekennzeichneten Kategorien einteilen. Bei der Kategorie *„Interne Operationen“* handelt es sich um private Methoden, die *CTP4Layer* als interne Unterroutinen für diverse Aufgaben heranzieht. Die drei anderen Kategorien *„Steuerung“*, *„Request-Methoden“* und *„Indication-Methoden“* bilden das Protokollinterface der Komponente *Netzwerk-Protokoll*. Über sie erfolgt die Ankopplung der weiter unten beschriebenen Komponente *Applikationsbasis*. Der Aufruf dieser Methoden erfolgt aus eben diese Komponente. Aber nun zu den Methoden der genannten Kategorien:

- **Steuerung**
  - *Init()* ist analog der oben beschriebenen *CEthernetLayer::Init()*. Letztere wird implizit mit aufgerufen. Der Aufruf von *Init()* erfolgt einmalig beim Start der Gesamtanwendung.
  - *OnTimer()* wird in einem 100ms-Intervall aus der Komponente *Applikationsbasis* getriggert. Von diesem Master-Timerintervall leiten sich alle anderen Zeitabläufe ab, wie sie z.B. für das rechtzeitige Versenden von Acknowledges und das Wiederholen von nicht bestätigten Telegrammen nötig sind. Eben diese zeitabhängigen Aktionen steuert *OnTimer()*.
- **Interne Operationen**
  Die in dieser Kategorie zusammengefassten privaten Methoden erfüllen allgemeine Aufgaben, die sich die Methoden der übrigen Kategorien zunutze machen.
  - *SendTP4Telegram()* erhält als Übergabeparameter ein *CTP4Telegram*-Objekt und versendet die darin gespeicherte TP4-TPDU durch Aufruf von *CEthernetLayer::SendEthernetTel()*. *SendTP4Telegram()* ist somit das Bindeglied zur Vaterklasse *CEthernetLayer* und zum Netzwerk.
  - *AddTel2ListAndSend()* ist die Vorstufe zu *SendTP4Telegram()*. Besteht noch Sendekredit, wird das übergebene *CTP4Telegram*-Objekt an die Sendefensterliste des die Verbindung repräsentierenden *CConnection*-Objekts gehangen und per *SendTP4Telegram()* versandt. Ist das Sendefenster voll, erfolgt stattdessen ein Anhängen an die Pending-Liste des *CConnection*-Objekts. Sofern nicht explizit beschrieben, durchlaufen alle zu versendenden TPDUs diesen Standard-Weg.
  - *ResendTelegramList()* wiederholt nach einem ausgebliebenen *Acknowledge* alle Telegramme in der Sendefensterliste einer Verbindung. Der Aufruf erfolgt aus *OnTimer()* nach einem Timeout des ersten Telegramms in der Liste. Erreicht der zugehörige Wiederholungszähler Null, wird der Verbindungsabbruch ausgelöst.
  - *SendPendingTels()* verschiebt nach dem Empfang eines *Acknowledge* eventuell auf Versand wartende *CTP4Telegram*-Objekte solange aus der Pending-Liste in die Sendefensterliste, bis letztere voll oder erstere leer ist. Gleichzeitig werden die verschobenen Telegramme versendet.

- *SetTP4State()* setzt den Verbindungszustand einer TP4-Verbindung auf den als Parameter zu übergebenden neuen Zustand und löst die beim Zustandsübergang auszuführenden Funktionen aus. Jedes *CConnection*-Objekt speichert den gegenwärtigen Verbindungszustand in der Membervariablen *m_enTP4State.* Erlaubte Zustände sind durch einen enum-Typen wie folgt definiert:

```
enum TP4_STATE
{
  // Initial-Zustand, gesetzt im CConnection-Konstruktor
  TP4_STATE_CLOSED=0x00,
  // Verbindungsaufbau läuft, warten auf Connection Confirm
  TP4_STATE_CONNECTING,
  // Arbeitszustand bei bestehender Verbindung
  TP4_STATE_CONNECTED,
  // Verbindungsabbau vorgemerkt, aber noch TPDUs unbestätigt
  TP4_STATE_DISCONNECTING,
  // Disconnect Request versendet, warten auf Confirm
  TP4_STATE_DISCONNECTED,
  // 3-Wege-Verbindungsaufbau: Warten auf ACK nach Versand von CC
  TP4_STATE_WAITFORCONNACK
};
```

  Mehr dazu in Abschnitt 6.6.2, *„Zustandssteuerung“,* Seite 88.

- *SendDisconnectTelegram()* erstellt eine *Disconnect Request*-TPDU und puffert sie in einem *CTP4Telegram*-Objekt. Über *AddTel2ListAndSend()* wird die TPDU versendet und an die Sendefensterliste angereiht, bis nach Eintreffen des *Disconnect Confirm* die Verbindung endgültig gelöscht werden kann. Die nötigen Verbindungsdaten enthält das als Parameter zu übergebende *CConnection*-Objekt.
- *SendAckTelegram()* erstellt und versendet eine *Acknowledge*-TPDU für die durch das als Parameter übergebene *CConnection*-Objekt repräsentierte Verbindung. Weil Acknowledges nie selbsttätig wiederholt werden, ist ein Einreihen in die Sendefensterliste und damit der Umweg über ein *CTP4Telegram*-Objekt unnötig. *SendAckTelegram()* erstellt die TPDU als Bytefeld im lokalen Speicher und verschickt dieses mit *SendEthernetTel()*.

❍ **Request-Methoden**

Die Methoden in dieser Kategorie erledigen die Dienste, die externe Prozesse über die Komponente *Dienstzugang* anfordern können. Sie kommen demnach nach Eingang einer Message Queue-Nachricht aus dieser Komponente mit den zugehörigen Daten zur Ausführung. Die Methoden *ConnectRequest(), DisconnectRequest(), ConnectResponse(), Send()* und *SendAllDaten()* aus der Klasse *CTP4SAP* korrespondieren mit *Connect(), Disconnect(), SendConnectionConfirm(), SendDataBlock()* und *SendConnectionLess()* aus *CTP4Layer*. Zugegeben, die Namensangleichung ist ein Kosmetik-Fall für das nächste Software-Update.

- *Connect()* erzeugt über *CConnectionList::CreateNewConnection()* ein *CConnection*-Objekt als Repräsentant für die neue Verbindung. Außerdem wird eine *Connection Request*-TPDU zusammengestellt, in einem neuen *CTP4Telegram*-Objekt abgelegt und via *AddTel2ListAndSend*() an die Gegenseite versendet. Die junge Verbindung wartet nun im Zustand `TP4_STATE_CONNECTING` auf das *Connection Confirm* der Gegenseite.

- *Disconnect()* veranlasst durch Aufruf von *SetTP4State()* lediglich den Zustandsübergang nach `TP4_STATE_DISCONNECTING`, wodurch der bevorstehende Verbindungsabbau vermerkt ist. Sobald das Sendefenster keine auf Bestätigung wartenden Telegramme mehr enthält, beginnt der Verbindungsabbau. Näheres folgt in Abschnitt 6.6.2.
- *SendConnectionConfirm()* erzeugt und versendet ein *CTP4Telegram*-Objekt mit einer *Connection Confirm*-TPDU. Dies erfolgt in Reaktion auf die mittels *CTP4SAP::ConnectResponse()* signalisierte gewünschte Verbindungsannahme. Die Verbindung verharrt im Zustand TP4_STATE_WAITFORCONNACK bis zum Eintreffen der zugehörigen *Acknowledge*-TPDU (Drei-Wege-Handshake beim Verbindungsaufbau, siehe Abschnitt 3.5). Erst danach ist die Verbindung vollständig aufgebaut.
- *SendDataBlock()* vollzieht das über *CTP4SAP::Send()* angeforderte Versenden einer Daten-TPDU über eine bestehende Verbindung.
- *SendConnectionLess()* überträgt die per *CTP4SAP::SendAllDaten()* eingereichten Nutzdaten verbindungslos. Da dazu naturgemäß kein Sendefenster existiert, versendet *SendConnectionLess()* die TPDU unmittelbar als lokales Bytefeld über *CEthernetLayer::SendEthernetTel()*.

❍ **Indication-Methoden**

Diese Methoden sind zuständig für die Reaktion auf durch den TPDU-Empfang von der Gegenseite ausgelöste Ereignisse *(„Indications")*. Hier erfolgt die Weiterverarbeitung der vom Prozess *RECCP* in Form der Struktur `BOT_EMPF_SEND` aufbereiteten TPDUs. Viele davon (bzw. die resultierende Struktur `BOT_EMPF_SEND`) haben mit dem Prozess SENDCP noch nicht ihr Endziel erreicht, sondern müssen durchgereicht werden an den endgültigen Verarbeitungsprozess („externe Prozesse" in Abbildung 6.1). Die nachfolgend beschriebenen Funktionen signalisieren daher über einen booleschen Rückgabewert, ob eine Weiterleitung erfolgen soll. Die Weiterleitung übernimmt die aufrufende Komponente *Applikationsbasis*. Gründe gegen das Weiterleiten sind zum Beispiel eine im aktuellen Verbindungszustand ungültige TPDU oder eine unerwartete Sequenznummer. TPDUs für eine bestehende Fernlade-Verbindung werden von SENDCP immer unverändert durchgeschleust. Der endgültige Verarbeitungsprozess ist festgelegt durch das Feld `BOT_EMPF_SEND.ENDPROZESS`, das RECCP während der Vorverarbeitung mit dessen Namen gefüllt hat. Im Normalfall ist dies PROT1_ABW oder PROT3_ABW, siehe Kapitel 1.3, *„Die Struktur des neuen Systems"*, Seite 14.

- *OnConnectionRequest()* ist gleich die Ausnahme zur genannten Weiterleitungsregel. Hier wird aufgrund der erhaltenen Verbindungsanfrage mit *CConnectionList::CreateNewConnection()* ein neues *CConnection*-Objekt erzeugt und danach mit Verbindungsnummer, Kredit und Acknowledge Time der Remote-Seite versorgt. Vor alledem wird über *CTP4SAP::CreateVBNr()* eine neue lokale Verbindungsnummer vergeben, die als Rückgabewert innerhalb von `BOT_EMPF_SEND` immer zum Zielprozess gelangt. Dieser muss darauf über die *CTP4SAP*-Methoden *ConnectResponse()* oder *DisconnectRequest()* reagieren.

- *OnDisconnectRequest()* wird, sofern kein Fehler in der Verarbeitung auftrat, immer die Weiterleitung an den Endprozess auslösen, um den von der Gegenseite geforderten Verbindungsabbau zu signalisieren. Wurden von der Gegenseite empfangene *Data*-TPDUs noch nicht bestätigt, geschieht dies nun mit *SendAckTelegram().* Da ISO/OSI 8073 einen abrupten Verbindungsabbau vorsieht, erstellt und versendet die Methode daraufhin unmittelbar eine *Disconnect Confirm*-TPDU und löscht über *ConnectionList::Delete()* alle zur Verbindung gehörenden Ressourcen.
- *OnConnectionConfirm()* veranlasst im fehlerfreien Normalfall ebenfalls die Weiterleitung an den Endprozess. Mit Eingang einer *Connection Confirm*-TPDU wechselt die Verbindung in den Zustand `TP4_STATE_CONNECTED` und ist ab sofort für den Austausch von Nutzdaten bereit. Der Eingang des *Connection Confirm* wird der Gegenseite als Abschluss des Drei-Wege-Verbindungsaufbaus mit *SendAckTelegram()* bestätigt.
- *OnData()* zeigt den Empfang einer Nutzdaten-TPDU an. Entspricht deren Sequenznummer der Erwarteten, erfolgt die Weiterleitung an den Endprozess. Andernfalls wird die TPDU verworfen und mittels einer *Acknowledge*-TPDU an die Gegenseite die erwartete TPDU angefordert. Gemäß des in 3.3, *„Flusskontrolle, Sendefenster und go-back-n“*, Seite 41 beschriebenen Mechanismus wird gegebenenfalls der Start des *Acknowledge Timers* oder das Bestätigen der bisher empfangenen TPDUs per *Acknowledge* erforderlich. Diese Abläufe beschreibt Abschnitt 6.6.1.2, *„Datenempfang und Acknowledge Timer“,* Seite 85.
- In *OnDataAcknowledge()* werden alle mit der soeben empfangenen *Acknowledge*-TPDU bestätigten Telegramme aus der Sendefensterliste gelöscht und mit *SendPendingTels()* eventuell wartende Telegramme nachgeschoben. Befand sich die Verbindung vorher im Zustand `TP4_STATE_WAITFORCONNACK`, so war das *Acknowledge* die Quittung auf ein vorab versendetes *Connection Confirm*, worauf die Verbindung in den Arbeitszustand `TP4_STATE_CONNECTED` wechselt. *Acknowledges* werden nicht an den übergeordneten Prozess weitergeleitet, da sie keine Nutzinformation böten. Ausnahme ist die genannte Fernlade-Verbindung. Siehe dazu den Ablaufplan in Abbildung 6.16, Seite 84.
- In *OnDisconnectConfirm()* ausgewertete *Disconnect Confirm*-TPDUs führen zum endgültigen Löschen aller zur Verbindung gehörenden Ressourcen via *ConnectionList::Delete().* Der Zielprozess selbst hat vorab den Verbindungsabbau über ein *Disconnect Request* veranlasst und erwartet durch den abrupten TP4-Verbindungsabbau kein *Disconnect Confirm*. Weitergeleitet wird daher nur bei Fernlader-Verbindungen.

6.5.2.6 Die TPDU-Erstellung

Bislang wurde beschrieben, welche *CTP4Layer*-Methoden TP4-TPDUs in Form eines *CTP4Telegram*-Objekts erstellen und über die Standardoperation *AddTel2ListAndSend()* versenden. Wie entsteht aber nun ein *CTP4Telegram*-Objekt?

Hier hilft die statische Fabrikmethode *CTP4Telegram::CreateTelegram(),* die einen Pointer auf ein neues *CTP4Telegram*-Objekt zurückgibt. Sie benötigt vier Parameter: Erstens die zugehörige TPDU-Nummer, welche *GetNextTPDUNumber()* des zuständigen *CConnection*-Objekts inkrementiert und als 7-Bit-Zahl rückliefert. Zweitens die zugehörige lokale Verbindungsnummer. Ferner einen Pointer auf die komplette TPDU mit Header und Nutzdaten, sowie deren Länge. Der Speicherbereich mit der TPDU wird in einen objektinternen Puffer kopiert. Zur Erstellung eines *CTP4Telegram*-Objekts verfahren alle Methoden auf eine typische, anhand *CTP4Layer*::*SendDataBlock()* für eine *Data*-TPDU gezeigten Weise:

```
bool CTP4Layer::SendDataBlock(unsigned short usVBNr, unsigned char* pauchDataBuffer,
                              unsigned short usBufferLen)
{
  CConnection*   pConnection;        // Pointer auf CConnection-Objekt
  CTP4Telegram*  pTelegram;          // Pointer auf zu erzeugendes CTP4Telegram-Objekt
  unsigned char* pTP4TelegramBuffer; // Temporärer Buffer für Header+Payload
  unsigned short usTP4TelSize;       // Die Länge des temp. Buffers

  static unsigned char achDataTelHeader[]= {        // Header für DATA-TPDU
    /* LLC-Header */ 0xFE, 0xFE, 0x03, 0x00,
    /* DATA-TPDU */  0x04, 0xF0, 0x05, 0x01, 0x00 };

  // Eintrag zur Verbindungsnummer in Connection-Liste finden
  if ((pConnection=this->m_ConnectionList.Find(usVBNr))==NULL) return false;
  // aktuellen Verbindungszustand überprüfen
  if (pConnection->GetTP4State()!=TP4 STATE CONNECTED) return false;

  // Verbindungsnummer der Gegenseite eintragen
  *((unsigned short*) &achDataTelHeader[6])= htons(pConnection->GetRemoteVBNr());
  // TPDU-Nummer eintragen
  unsigned char ucTPDUNo=pConnection->GetNextTPDUNumber()|TP4_TPDUMASK; // EOT-Bit einodern
  *((unsigned char*) &achDataTelHeader[8])= (unsigned char) ucTPDUNo;

  // Gesamt-Telegrammlänge kalkulieren, Payload und Header zusammenfügen
  usTP4TelSize=sizeof(achDataTelHeader)+usBufferLen;
  pTP4TelegramBuffer=new unsigned char[usTP4TelSize];
  memcpy (pTP4TelegramBuffer, achDataTelHeader, sizeof(achDataTelHeader));
  memcpy (&pTP4TelegramBuffer[sizeof(achDataTelHeader)], pauchDataBuffer, usBufferLen);

  // Telegrammobjekt erzeugen und schicken
  pTelegram=CTP4Telegram::CreateTelegram(usVBNr, ucTPDUNo, pTP4TelegramBuffer, usTP4TelSize);
  delete[] pTP4TelegramBuffer; // Der Bufferinhalt wurde in CTP4Telegram kopiert
  return this->AddTel2ListAndSend (pTelegram); // Telegramm senden bzw. parken
}
```

Listing 6.1: Typischer Ablauf der Telegramm-Erstellung am Beispiel der Methode SendDataBlock()

Alle TPDU-Header sind demnach über statische Bytefelder definiert. Nur Bestandteile, bei denen das sinnvoll und nötig ist, werden in der jeweiligen Methode dynamisch eingetragen. Damit ist der unter 6.2, *„Aufgaben und Anforderungen“* genannte Punkt *„So flexibel wie nötig, so einfach wie möglich“,* erfüllt. Der LLC-Header `0xFE 0xFE 0x03 0x00` ist, getreu dieses Mottos, ebenfalls Bestandteil des Bytefeldes. Zwar gehörte er als OSI-Schicht-2-Element exakter in die Klasse *CEthernetLayer*, allerdings brächte der dann nötige Mehraufwand zur Vervollständigung der TPDUs keinen funktionalen Mehrwert.

6.5.2.7 Gesamt-Klassendiagramm

Als Abschluss des Themas *Software-Design* zeigt Abbildung 6.10 alle SENDCP-Klassen mit ihren Membervariablen und –methoden. Die Klasse *CSendCP* bildet die Komponente *Applikationsbasis,* deren Beschreibung in Kapitel 6.5.3 folgt.

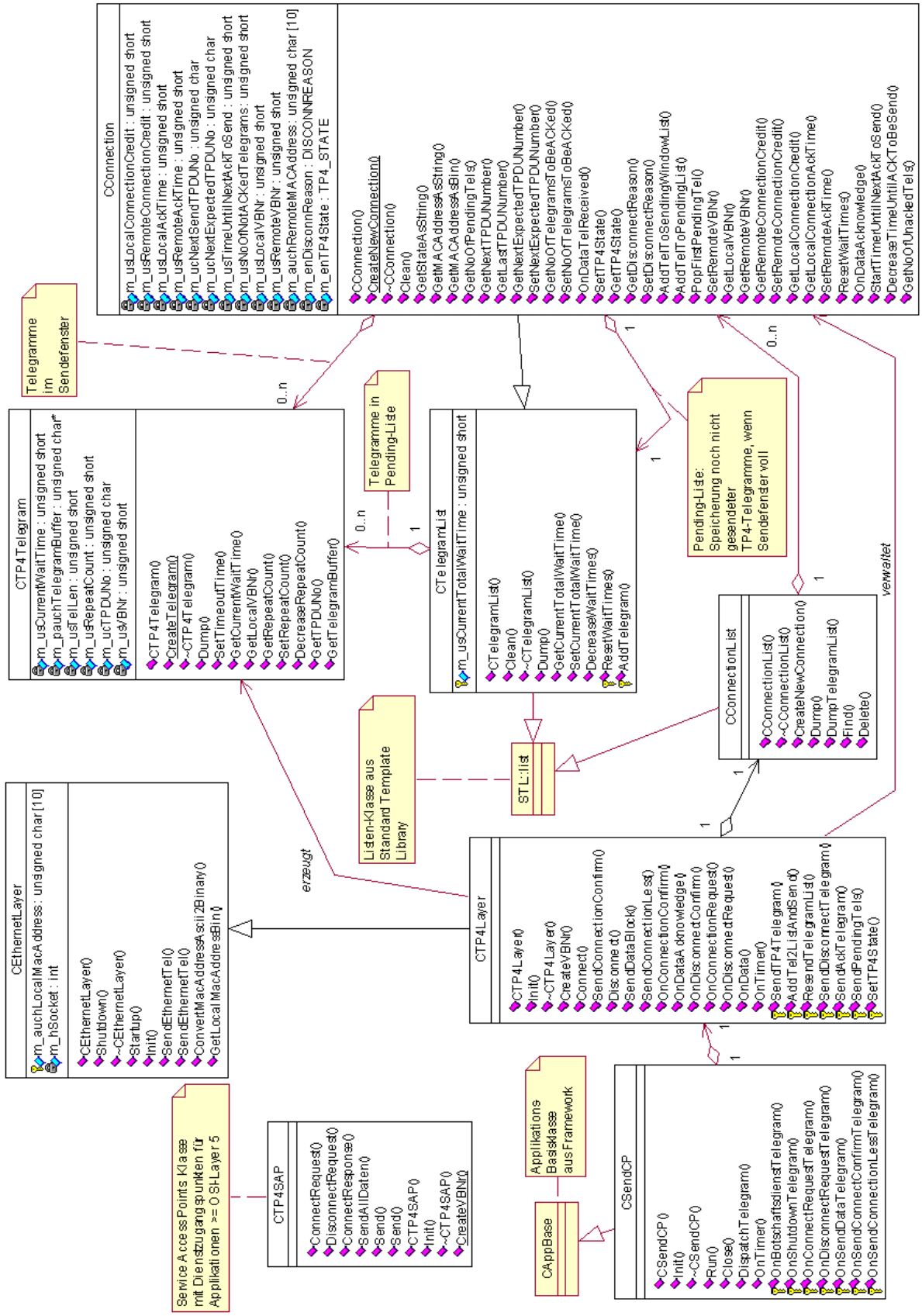


Abbildung 6.10: Gesamt-Klassendiagramm des Prozesses SENDCP

6.5.3 Die Applikationsbasis mit der Klasse *CSendCP*

Alle auf dem Applikationsframework basierenden Anwendungen sind abgeleitet von der Framework-Applikationsbasisklasse *CAppBase*. Trug die davon abgeleitete Anwendungsklasse in der allgemeinen Beschreibung in Kapitel 2, *„Das Applikations-Framework“*, den Namen *CMyKomApp*, heißt sie im konkreten Fall des Prozesses SENDCP *CSendCP*. Genau diese Klasse *CSendCP* bildet die Komponente *Applikationsbasis.* Über sie stehen dem Prozess SENDCP alle Funktionen aus der Framework-Library, wie Logging und Interprozesskommunikation via Message Queue, zur Verfügung.

Als Basisklasse des Prozesses SENDCP steuert *CSendCP* das Initialisierungs- und Laufzeitverhalten in den framework-typischen Methoden *Init(), Run(), Close()* sowie *OnTimer()*.

Die *Init()*-Routine initialisiert vorwiegend das Logging und den Zugriff auf den Shared Memory-Bereich und ruft *Init()* des von *CSendCP* angelegten *CTP4Layer*-Objekts auf.

Im laufenden Betrieb wartet der Prozess in *Run()* auf Message Queue-Nachrichten aus der Komponente *Dienstzugang* sowie vom Prozess RECCP und delegiert sie zur Verarbeitung an die Komponente *Netzwerk-Protokoll* weiter. Umgekehrt kann letztere über *CSendCP* auf die Framework-Funktionen zugreifen.

Nachrichten von *RECCP* enthalten jeweils extrahierte Daten empfangener TPDUs in Form der Struktur `BOT_EMPF_SEND`. In der *CSendCP*-Methode mit dem historisch bedingten Namen *OnBotschaftsdienstTelegram()* wird anhand des TPDU-Codes in die zugehörige *Indication*-Methode der Komponente *Netzwerkprotokoll* verzweigt. Bei Rückkehr daraus mit *true* erfolgt abschließend die Weiterleitung der Nachricht in die Message Queue des externen Zielprozesses, der durch das Element `BOT_EMPF_SEND.ENDPROZESS` festgelegt ist.

Nachrichten aus der Komponente *Dienstzugang* gelangen zunächst in die ihrem Nachrichtentyp zugeordneten *CSendCP*-Methoden *OnConnectRequestTelegram(), OnDisconnectRequestTelegram(), OnSendConnectConfirmTelegram(), OnSendDataTelegram()* oder *OnSendConnectionLessTelegram()*. Dort werden sie mit den Parametern aus dem Message-Queue-Telegramm an die zuständigen *Request*-Methoden der Komponente *Netzwerk-Protokoll* weiterdelegiert.

Zu guter Letzt sorgt die Framework-Methode *OnTimer()* für die Generierung des Master-Zeitintervalls. Dieses ist im Prozess SENDCP auf 100ms gesetzt. Auch die *OnTimer()*-Methode delegiert alle anfallenden Arbeiten an die Komponente *Netzwerk-Protokoll* und triggert dort periodisch *CTP4Layer::OnTimer()*. Letztere leitet von diesem Master-Zeitintervall alle anderen Zeitabläufe ab, wie sie etwa für das rechtzeitige Versenden von Acknowledges benötigt werden.

CSendCP fungiert somit hauptsächlich als einfache Interface-Klasse zwischen dem Applikations-Framework mit seiner Message Queue und der eigentlichen Protokoll-Verarbeitung in der Komponente *Netzwerkprotokoll*. Durch die so erreichte Trennung kann die Protokollkomponente mit leichten Anpassungen auch in zukünftigen Anwendungen eingesetzt werden, die nicht auf dem Applikations-Framework basieren, wie es z.B. bei windows-basierten Anwendungen der Fall wäre.

Damit sind alle Komponenten des Prozesses SENDCP mit ihren statischen Zusammenhängen beschrieben. Was jetzt noch fehlt, ist Dynamik. Die belebt das nächste Kapitel.

6.6 Dynamisches Verhalten

6.6.1 Zeitabhängige Funktionen

Jede Transportverbindung benötigt zur Verbindungsüberwachung die unter Kapitel 3.2 und 3.3 beschriebenen *Acknowledge-* und *Retransmission*-Timer. Wie diese Timer im Prozess SENDCP mit Hilfe der Klassenmethoden und -variablen aus der Gesamt-Klassenübersicht in Abbildung 6.10 realisiert sind, ist Thema der folgenden Abschnitte.

Alle zeitabhängigen Funktionen steuert die Funktion *CTP4Layer::OnTimer()*. Diese wiederum wird in einem 100ms-Zeitintervall aus der framework-eigenen *OnTimer()*-Methode der Klasse *CSendCP* aufgerufen. Die Abläufe sind im Ablaufplan in Abbildung 6.19 zusammengefasst. Vorher aber einige Worte über den realisierten Algorithmus.

6.6.1.1 Retransmission-Timer und Sendefenster-Verwaltung

Beim Verbindungsaufbau teilen sich die beiden Partner innerhalb der *Connection Request-* und *Connection Confirm*-TPDUs unter anderem die *Acknowledge Time* mit. Dies ist die Wartezeit, nach der jede gesendete Daten-TPDU spätestens durch den Eingang eines *Acknowledge* bestätigt werden muss. Bleibt dieses aus, werden aufgrund des *go-back-n*-Prinzips alle TPDUs im Sendefenster wiederholt oder die Verbindung wird nach Überschreiten der maximalen Wiederholversuche abgebrochen.

Die der Gegenseite gewährte *Acknowledge Time* ist auf den im abgelösten System vorgefundenen konstanten Wert 500ms gesetzt. In der Testphase zeigte sich allerdings, dass die verbundenen Arbeitsplätze diese Zeitspanne restlos ausnutzen, ehe sie ein *Acknowledge* versenden. Dadurch kam es aufgrund von Timingschwankungen und Verarbeitungsverzögerungen gelegentlich zu unnötigem Neuversenden der Sendefensterliste. Seither wartet SENDCP vor Wiederholungen intern die doppelte Zeit, also 1000ms. Dies ist der konstante Wert, mit dem *CConnection::m_usLocalAckTime* im Konstruktor initialisiert wird. Der Gegenseite werden beim Verbindungsaufbau nach wie vor 500ms mitgeteilt.

Theoretisch bedeutet ein derart verzögertes Wiederholen eine Verringerung des Datendurchsatzes, da im Falle eines tatsächlichen Datenverlusts nun jedes Mal eine Sekunde bis zur Wiederholung und Fortsetzung verstreicht. Allerdings sind im vorliegenden einfachen LAN auf dem Übertragungsweg verlorengehende Datenpakete unwahrscheinlich. Sollte es tatsächlich zu Störungen kommen, die zu einer Sendewiederholung führen, so ist die wahrscheinliche Ursache an anderer Stelle zu suchen, etwa ein Ausfall des Arbeitsplatzes oder eine Leitungsunterbrechung. Solchen Ursachen ist nur durch manuelles Eingreifen beizukommen, nicht durch Wiederholen.

Gesendete TPDUs puffert, älteste voran, die durch *CConnection* bzw. deren Vaterklasse *CTelegramList* gebildete Sendefensterliste. Jedes *CTP4Telegram*-Objekt darin speichert seine aktuelle Wartezeit bis zum Timeout in *m_usCurrentWaitTime*, die Anzahl verbleibender Wiederholversuche in *m_usRepeatCount.* Letztere wird im Konstruktor auf den konstanten Wert 3 initialisiert.

Wie lässt sich nun mit der 100ms-Zeitscheibe des Framework-Timers ein Retransmission Timer realisieren, nach dessen Ablauf bei ausgebliebenem *Acknowledge* das Sendefenster wiederholt wird?

Die naheliegendste Lösung basiert auf absoluten Zeitangaben: Jedes einzelne *CTP4Telegram*-Objekt bekommt beim Anfügen an die Sendefensterliste die maximale interne *Acknowledge Time*, also 1000ms, per *SetTimeoutTime()* zugewiesen. Nach jedem Ablauf der Zeitscheibe iteriert *CTP4Layer::OnTimer()* über die Sendefensterliste jeder bestehenden Verbindung und verringert die Wartezeit aller Telegramme um 100ms. Erreicht ein Timeout-Zähler Null, wird die Sendefensterliste wiederholt.

Dieser Ansatz durchliefe also innerhalb jedes Zeitintervalls sowohl die komplette, durch *CConnectionList* gebildete Verbindungsliste als auch die Sendefensterliste jedes *CConnection*-Objekts in der Verbindungsliste. Dabei wird unterschiedslos auf jedes existierende *CTP4Telegram*-Objekt zugegriffen, um dessen Wartezeit zu verringern. Mag dieses Vorgehen bei den maximal 50 zu erwartenden gleichzeitigen Verbindungen und einer durch den Kredit von acht begrenzten Sendefenstergröße undramatisch sein, bindet es doch Rechenzeit, die anderen Prozessen nicht zur Verfügung steht.

Eine etwas trickreichere Vorgehensweise vermeidet das komplette Durchlaufen der Sendefensterliste. Stattdessen verändert sie nur die Wartezeit des ersten, ältesten Telegramms in der Liste, welches durch *STL::list.front()* leicht erreichbar ist. Alle weiteren Telegramme erhalten beim Anhängen jeweils die relative Wartezeit, die sich aus der *Acknowledge Time* von 1000ms minus der Summe der Wartezeiten der Listenvorgänger ergibt, zugewiesen. Danach bleiben sie unverändert. Allgemein gilt für die Wartezeit *m_usCurrentWaitTime* eines an Stelle *n* eingefügten *CTP4Telegram*-Objekts:

$$m\_usCurrentWaitTime_n = m\_usLocalAckTime - \sum_{i=0}^{n-1} m\_usCurrentWaitTime_i$$

Hier ist *m_usLocalAckTime* die konstante, im *CConnection*-Konstruktor festgesetzte interne *Acknowledge Time* der Verbindung von 1000ms. Die Summe der Wartezeiten in der Liste wird zur schnellen Verfügbarkeit in *CTelegramList::m_usCurrentTotalWaitTime* abgelegt. Wie das Verfahren abläuft, erläutern die folgenden Beispielfälle.

1. Anfügen von Telegrammen

Sei die Sendefensterliste zunächst leer. Zwei *CTP4Telegram*-Objekte mit den TPDU-Nummern 1 und 2 werden nun innerhalb der gleichen Zeitscheibe versendet und angehängt. Nummer 1 erhält die Gesamt-Wartezeit von 1000ms. Ist diese abgelaufen, gilt dies auch für die Wartezeit von Nummer 2. Dessen Wartezeit relativ zum Vorgänger beträgt daher 0ms.

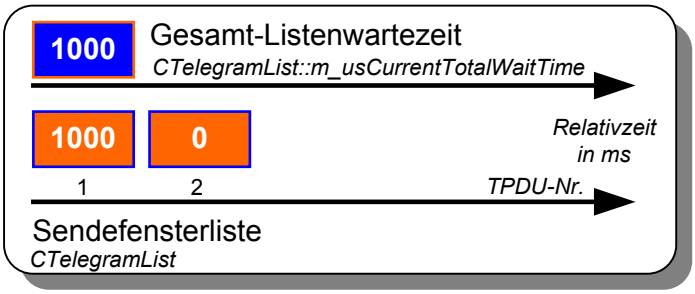


Abbildung 6.11: Die beiden ersten TPDUs im Sendefenster

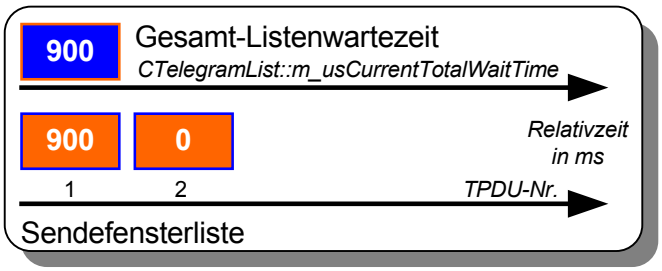


Abbildung 6.12: Wartezeiten nach der ersten Zeitscheibe

CTP4Layer::OnTimer() nutzt nach Ablauf der ersten Zeitscheibe *CTelegramList::DecreaseWaitTimes()* zum Verringern der Wartezeit von Nummer 1 und der Gesamtwartezeit in *m_usCurrentTotalWaitTime* um 100ms. TPDU 2 und eventuelle weitere TPDUs bleiben unverändert.

Wird nun TPDU Nummer 3 angehangen, errechnet sich für ihre Wartezeit mit der aktuellen Gesamt-Listenwartezeit von 900ms gemäß obiger Formel ein Wert von 100ms. Anschließend beträgt die Gesamt-Wartezeit, wie stets nach dem Einfügen einer neuen TPDU, wieder 1000ms. Jede weitere in der gleichen Zeitscheibe angehangene TPDU läuft gleichzeitig mit ihrem Vorgänger, also wiederum 0 ms später, ab.

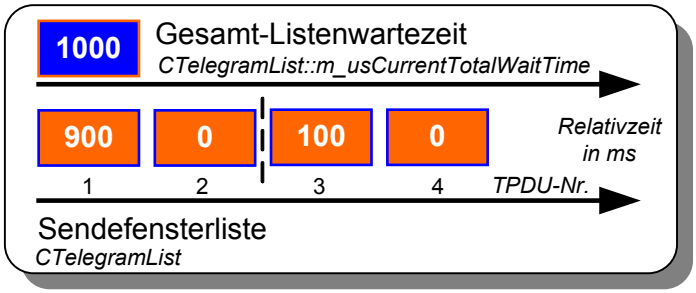


Abbildung 6.13: Neue TPDUs in Zeitscheibe 2

2. Der Empfang eines *Acknowledge*

Zwei Zeitscheiben später stellt sich die Listen-Gesamtsituation wie in Abbildung 6.14 dar. Angenommen, ein eingehendes *Acknowledge* bestätigt nun die TPDUs Nummer 1 und 2, wodurch diese aus der Liste entfernt werden könnten. Allerdings verliert die Wartezeit von Nummer 3 dann ihren relativen Bezug, verbleiben ihr doch, neben den eigenen 100ms, die Summe aller Zeiten der Vorgänger-TPDUs, in diesem Fall 700ms. Diese Summe ist TPDU 3 vor dem Löschen der bestätigten TPDUs gutzuschreiben. Allgemein gilt bei einem eingehenden *Acknowledge* für TPDU n:

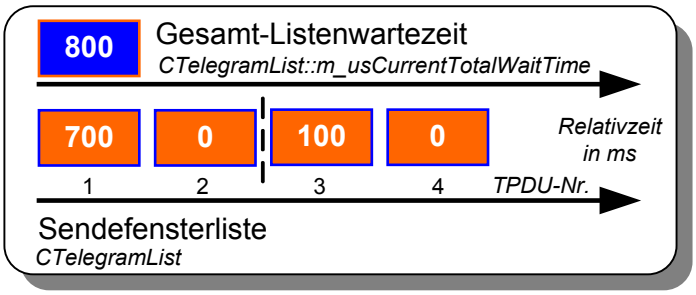


Abbildung 6.14: Sendefenster zwei Zeitscheiben später

$$m\_usCurrentWaitTime_{n+1} = m\_usCurrentWaitTime_{n+1} + \sum_{i=0}^{n} m\_usCurrentWaitTime_i$$

Die Rechnung und anschließendes Löschen übernimmt die *OnDataAcknowledge*()-Methode des zuständigen *CConnection*-Objekts, die nach jedem *Acknowledge* aus *CTP4Layer::OnDataAcknowledge()* aufgerufen wird. Danach ergibt sich die Situation gemäß nebenstehender Abbildung 6.15.

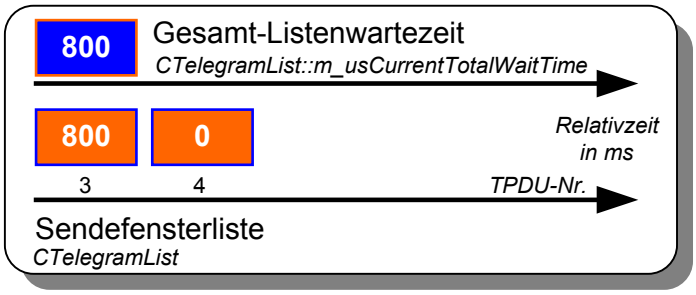


Abbildung 6.15: Situation nach ACK für TPDU 1 und 2

Die Gesamt-Listenwartezeit bleibt bei einem *Acknowledge* unverändert, solange noch TPDUs in der Liste warten. Erst wenn alle TPDUs bestätigt sind, ist die Gesamt-Wartezeit naturgemäß null.

Nach dem Empfang eines *Acknowledge* ist zunächst *CTP4Layer::OnDataAcknowledge()* aktiv. Deren Abläufe und Interaktionen mit den Methoden des beteiligten *CConnection*-Objektes zeigt der Ablaufplan in Abbildung 6.16. Weitere Beschreibungen enthält Abschnitt 6.5.2.5.

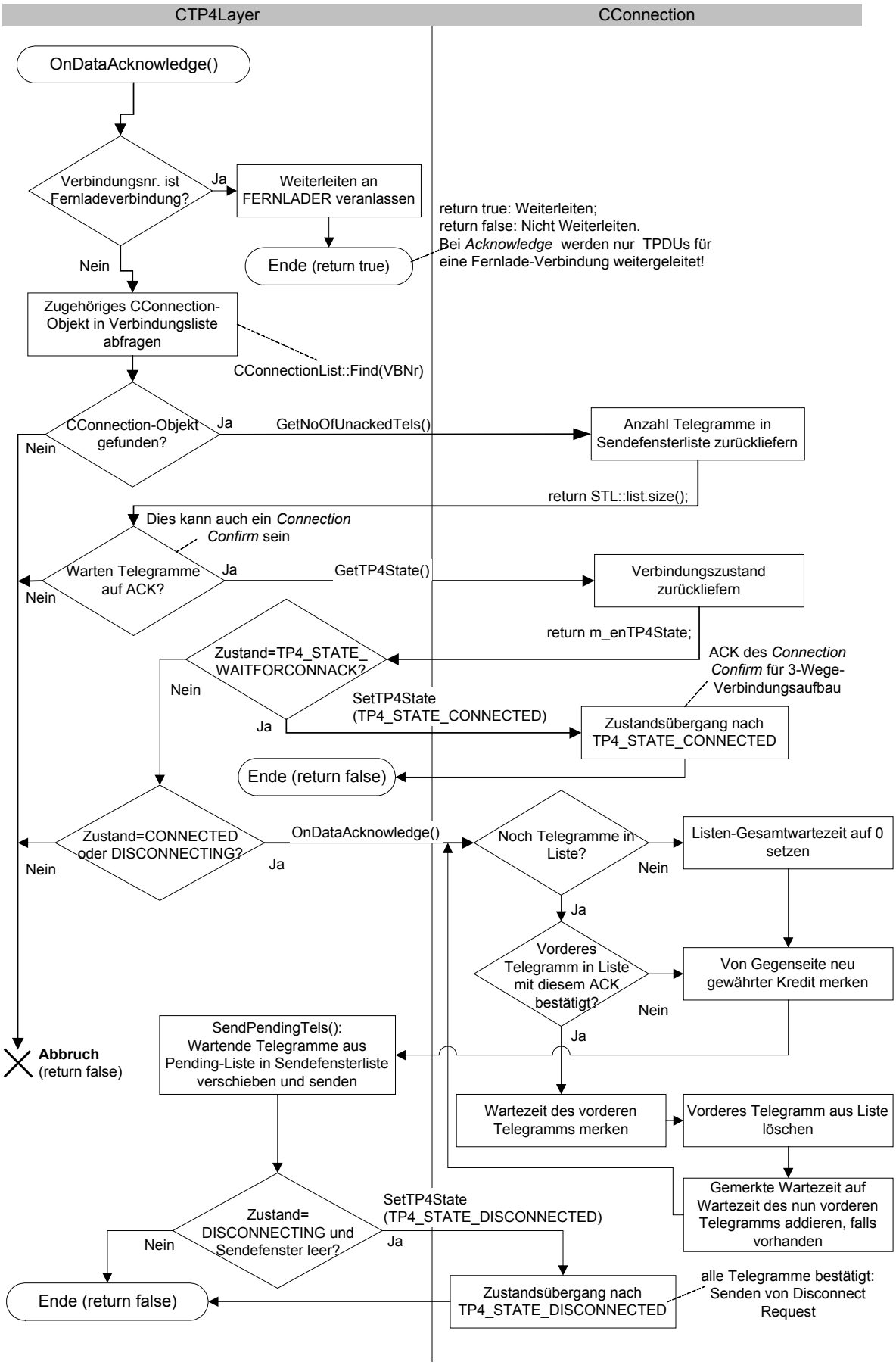


Abbildung 6.16: Ablaufplan CTP4Layer::OnDataAcknowledge()

3. Timeout

Zurück zur mit vier TPDUs gefüllten Sendefensterliste nach Abbildung 6.14. Dieses Mal warten die TPDUs allerdings vergebens auf ein *Acknowledge*. Nach 700 weiteren Millisekunden erreicht *m_usCurrentWaitTime* der ersten wartenden TPDU dann den Wert Null. *CTP4Layer::OnTimer()* reagiert darauf durch Aufruf von *ResendTelegramList()* der gleichen Klasse. Zunächst wird hier der Wiederholungszähler des ersten *CTP4Telegram*-Objekts in der Liste über dessen *DecreaseRepeatCount()*-Methode dekrementiert. Ist dieser nun Null, kehrt *ResendTelegramList()* mit *false* zurück. *OnTimer()* veranlasst daraufhin das Versenden eines *Disconnect Request* an die Gegenseite und löscht über *CConnectionList::Delete()* alle zur Verbindung gehörenden Ressourcen aus dem Speicher. Der angeschlossene übergeordnete Prozess erhält in Form der Struktur `BOT_EMPF_SEND` eine *Disconnect*-Nachricht über die Message Queue.

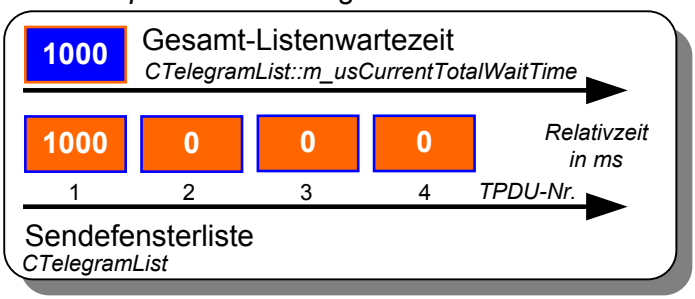


Abbildung 6.17: Sendefensterliste nach Sendewiederholung

Bevor es zum Verbindungsabbruch kommt, bestehen allerdings drei Sendeversuche. Alle TPDUs in der Sendefensterliste wiederholt *ResendTelegramList()*. Die nötige Neuinitialisierung der Wartezeiten erledigt *CTelegramList::ResetWaitTimes()*, wonach sich das Resultat nach Abbildung 6.17 einstellt. Siehe dazu den *OnTimer()*-Ablaufplan in Abbildung 6.19.

Fazit: Die vorgestellte Lösung durchläuft die Sendefensterliste nur beim Empfang eines *Acknowledge* und unvermeidlicherweise im Fehlerfall bei der Sendewiederholung. Im *Acknowledge*-Fall ist die Löschung der bestätigten Telegramme aus der Liste kombinierbar mit dem Aufaddieren ihrer Wartezeiten.

6.6.1.2 Datenempfang und *Acknowledge Timer*

Von der Gegenseite empfangene Nutzdaten-TPDUs müssen innerhalb der von der Remote-Seite gewährten *Acknowledge Time* bestätigt werden. Diese wird beim Verbindungsaufbau in *CConnection::m_usRemoteAckTime* abgelegt und kann sich von der lokalen Acknowledge Time (in *m_usLocalAckTime)* unterscheiden. SENDCP halbiert die zugeteilte Zeit und rundet sie auf das nächstkleinere geradzahlige Vielfache der *OnTimer()*-Zeitscheibe von 100ms ab. Eine *Acknowledge Time* von 500ms führt demnach spätestens 200ms nach Eingang einer *Data*-TPDU zum Senden eines *Acknowledge*.

CTP4Layer::OnData() übernimmt die beim Empfang einer *Data*-TPDU erforderlichen Aufgaben. Ist die Verbindung im Zustand `TP4_STATE_CONNECTED` und entspricht die empfangene TPDU-Nummer der Erwarteten, wird geprüft, ob die TPDU die erste war, für die noch kein ACK gesendet wurde. Trifft dies zu, folgt der Start des *Acknowledge Timers* durch Setzen von *CConnection::m_usTimeUntilNextAckToSend* auf 200ms. Die danach aufgerufene Methode *CConnection::OnDataTelReceived()* fasst das Inkrementieren der nächsten erwarteten TPDU-Nummer und des Zählers für empfangene, aber unbestätigte TPDUs *(m_usNoOfNotACKedTelegrams)* zusammen. Wird anhand letzterem festgestellt, dass die Gegenseite ihren Sendekredit verbraucht hat, erfolgt unmittelbar das Senden eines *Acknowledge*. Ansonsten übernimmt dies *CTP4Layer::OnTimer()*, falls die darin aufgerufene Methode *CConnection::DecreaseTimeUntilACKToBeSend()* den *Acknowledge Timer* auf Null gezählt hat. Dies zeigen die folgenden Ablaufdiagramme.

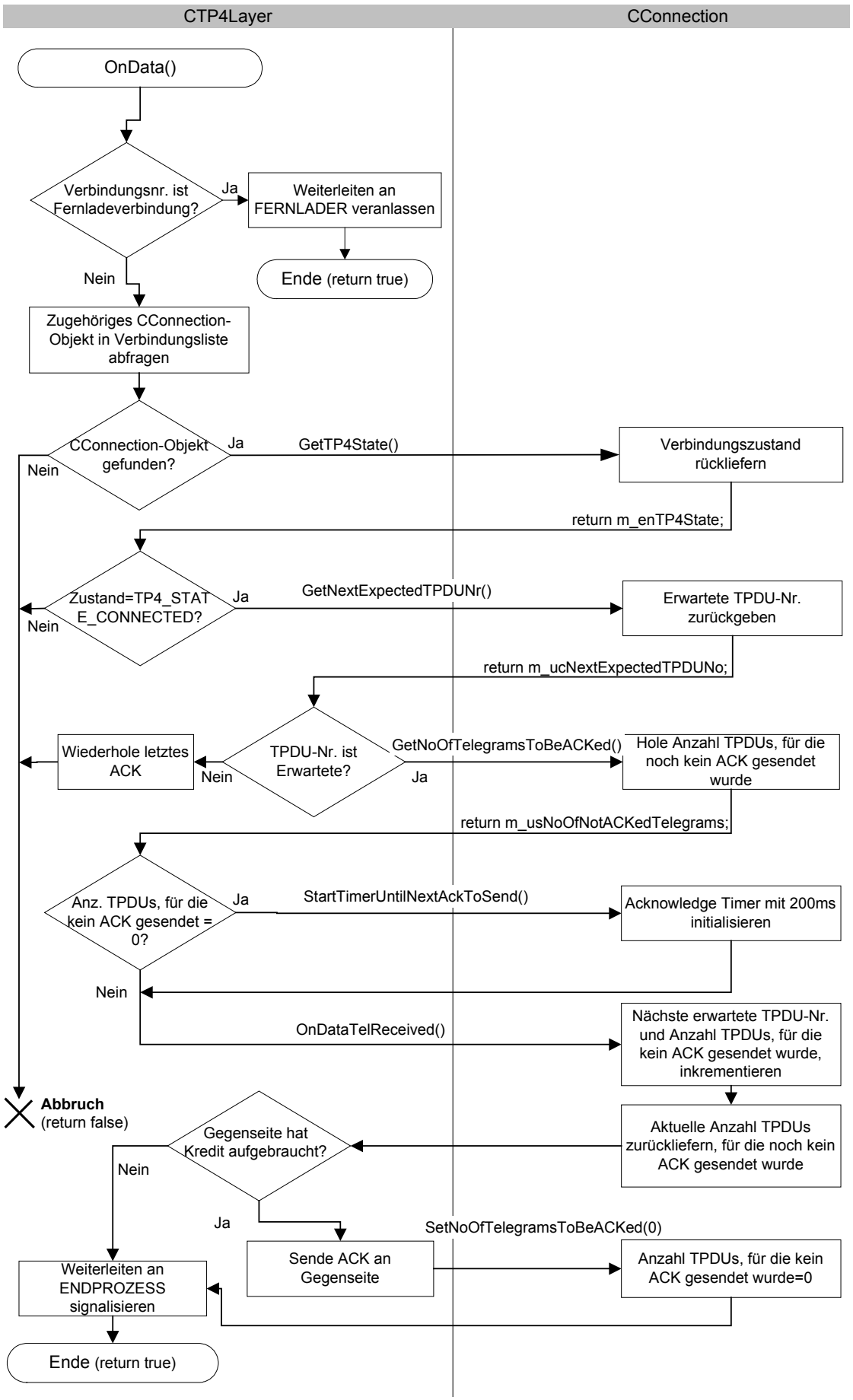


Abbildung 6.18: Ablaufplan CTP4Layer::OnData()

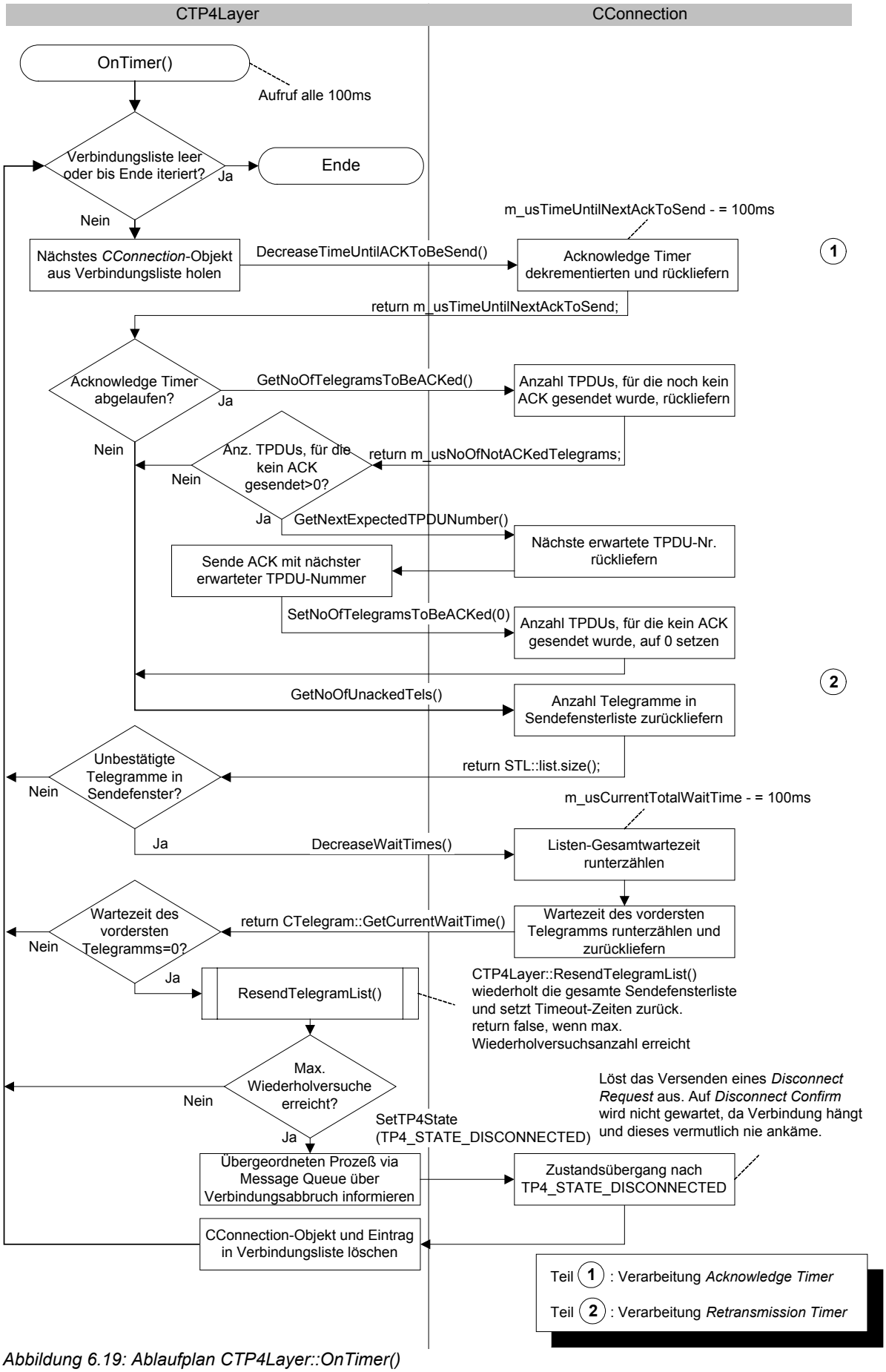


Abbildung 6.19: Ablaufplan CTP4Layer::OnTimer()

6.6.2 Zustandssteuerung

Die Steuerung des Verhaltens von SENDCP erfolgt in Abhängigkeit des aktuellen Verbindungszustandes. Je nach Zustand werden Ereignisse bearbeitet oder unterdrückt. Ferner kann ein eintretendes Ereignis einen Zustandsübergang und die Ausführung einer daran gekoppelten Funktion bewirken.

Die beiden folgenden Zustandsdiagramme zeigen die Abläufe zusammengefasst. Verbindungen sind bidirektional, d. h. beide Partner können Verbindungen aufbauen und Telegramme senden. Für größere Übersichtlichkeit erfolgt die Darstellung jedoch in je einem Diagramm für den selbst initiierten Aufbau einer Verbindung mit dem Senden von Telegrammen (Abbildung 6.20), und umgekehrt dem Handling einer von der Gegenseite initiierten Verbindung mit Telegramm-Empfang (Abbildung 6.21).

Die meisten in den beiden Zustandsdiagrammen gezeigten Aktionen haben ihre unmittelbare Entsprechung in einer *CTP4Layer*-Methode und tragen demnach deren Namen inklusive der abschließenden Klammern „*()*“. Einige Aktionen lassen sich nicht eindeutig zuordnen, sei es, weil sie mehrere Funktionalitäten zusammenfassen oder an verschiedenen Stellen im Code auftreten. Als Kennzeichnung fehlen ihren Namen die abschließenden Klammern.

Jede ausgeführte Aktion ist an das Eintreten eines Ereignisses gekoppelt. Ist dessen Ursache der Empfang einer TPDU oder ein SENDCP-internes Ereignis, wie Timer-Ablauf, ist der Ereignisname GROSSGESCHRIEBEN. Ist die Ursache eine Message Queue-Nachricht aus der Komponente *Dienstzugang*, ausgelöst durch einen übergeordneten Prozess, so trägt dieses Ereignis den zugehörigen Methodennamen der Klasse *CTP4SAP,* zusätzlich durch Unterstreichen gekennzeichnet. Beispiel: *ConnectRequest()*. Bei nicht gegebener direkter Zuordnung zu Klassenmethoden hilft folgende Aufstellung:

Aktionen:	
To_Layer_5	TPDU_EXPECTED inkrementieren; empfangene TPDU an übergeordneten Prozess weiterleiten. Letzteres übernimmt die Klasse *CSendCP*.
SendDisconnectConfirm	Sendet eine *DC*-TPDU als Reaktion auf *Disconnect Request.* Bestandteil von *CTP4Layer::OnDisconnectRequest()*
SendCR	Sendet eine *Connection Request*-TPDU. Erstmalig mit Erstellung des *CTP4Telegram*-Objekts in *CTP4Layer::Connect()*, danach bei Wiederholung desselben in *ResendTelegramList()*.
Ereignisse, Signale:	
CREDIT	Diagramm 1: Im Sendefenster ist noch Platz für weitere Telegramme Diagramm 2: Gegenseite hat noch Sendekredit, kein ACK wird gesendet
CC, DC, DATA, DATA_ACKNOWLEDGE	Entsprechende TPDU wurde empfangen. Auswertung in *OnConnectionConfirm()*, *OnDisconnectConfirm(), OnData(), OnDataAcknowledge()*.
CONNECTION_REQUEST	Eine Verbindungsanfrage von der Gegenseite wurde empfangen. Der übergeordnete Prozess muss darauf mit *ConnectResponse()* oder *DisconnectRequest()* reagieren. Auswertung in *OnConnectionRequest()*.
DISCONNECT_REQUEST	Eine Verbindungsabbau-Anforderung von der Gegenseite wurde empfangen. Führt zu Verbindungsabbau in *OnDisconnectRequest()*.
TIMEOUT	Für auf Bestätigung wartende TPDUs ist der Retransmission Timer (in *OnTimer()*) abgelaufen.
ACKTIMER	Ablauf des Acknowledge Timers. Empfangene TPDUs werden nun quittiert.
REPEATCOUNT	TPDU-Wiederholungszähler. Initialisierung auf 3 bei TPDU-Erstellung.
SENDINGWINDOWEMPTY	Sendefensterliste ist leer. Entspricht *CConnection::GetNoOfUnackedTels()==0*
TPDU_NR; TPDU_EXPECTED	TPDU- (Sequenz-) Nr. der empfangenen TPDU (aus `BOT_EMPF_SEND`); Erwartete TPDU-Nr. (*CConnection::GetNextExpectedTPDUNumber()*)
GetNoOfTelegramsToBeACKed()	*CConnection-Methode*. Liefert die Anzahl zu quittierender Telegramme.

Tabelle 6.2: Symbolische Namen der Aktionen und Ereignisse in den Zustandsdiagrammen

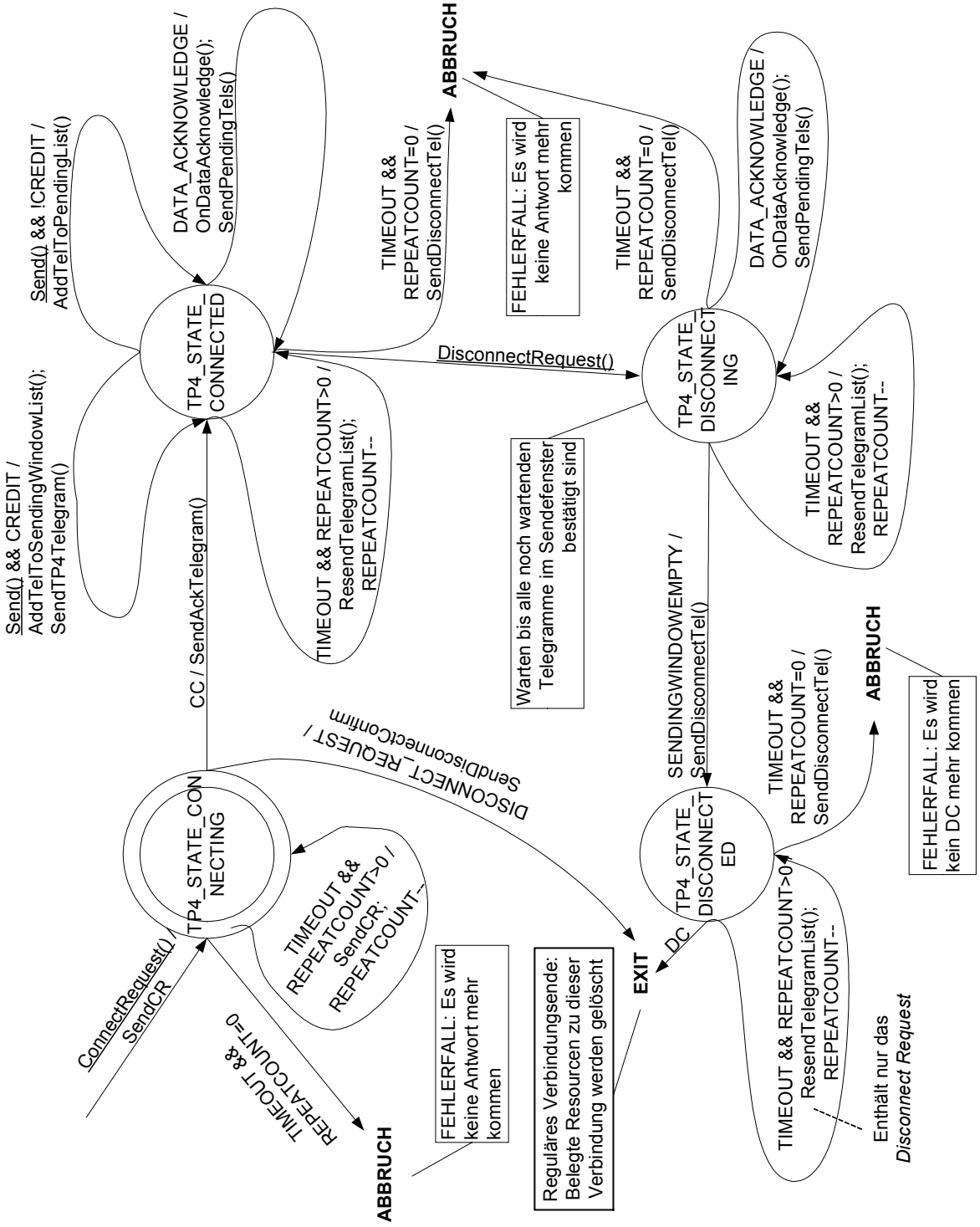


Abbildung 6.20: Zustandsdiagramm 1: Aktiver Verbindungsaufbau und Senderichtung

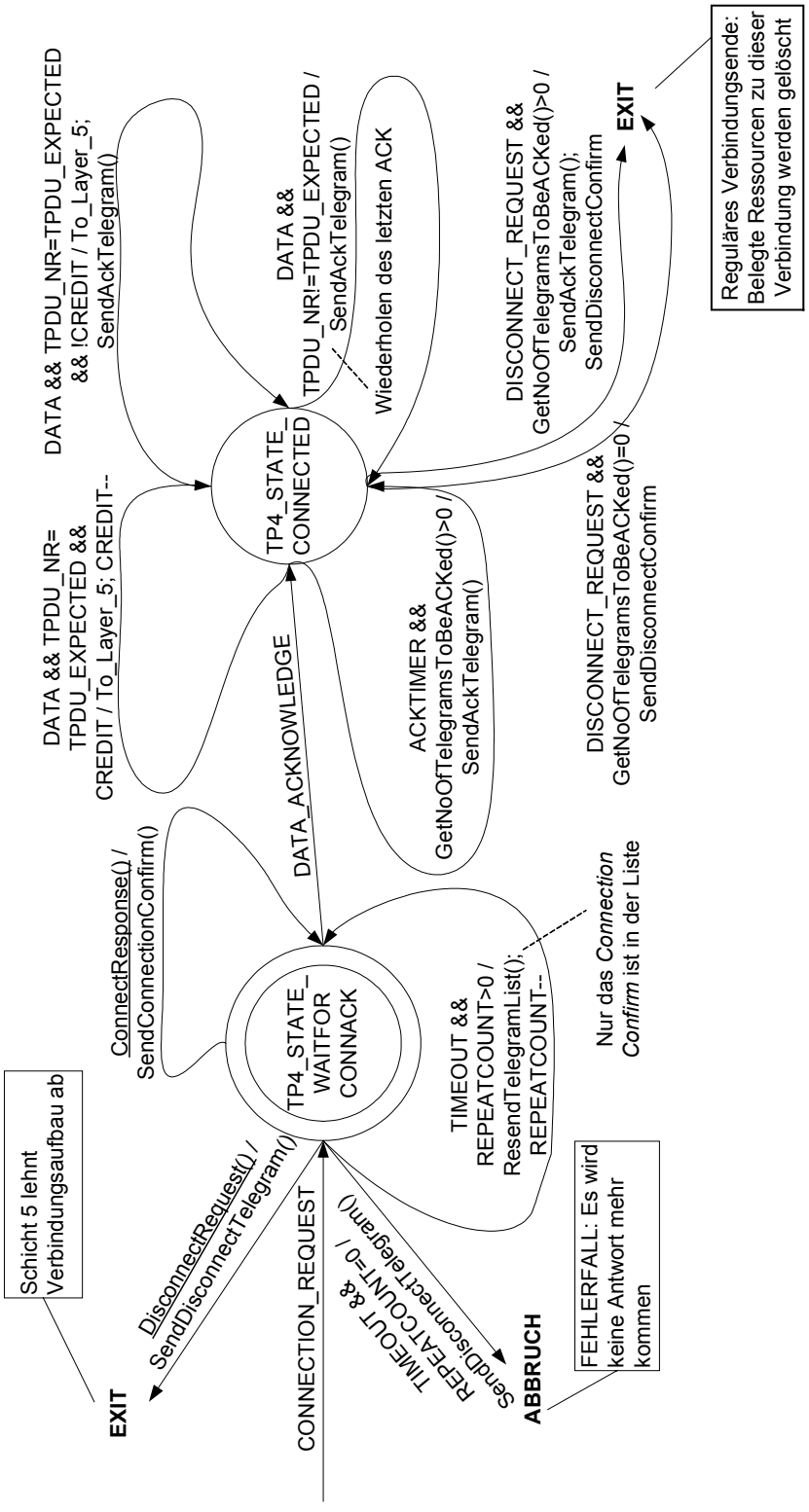


Abbildung 6.21: Zustandsdiagramm 2: Verbindungsaufbau durch Gegenseite und Telegrammempfang

Softwareseitig sind die möglichen Zustände abgebildet durch den in Abschnitt 6.5.2.5, *„Die Klasse* CTP4Layer*“*, Seite 73, eingeführten Enumerations-Typen *TP4_STATE*. Die fünf verschiedenen Zustände haben die folgenden Bedeutungen:

1. Zustände während des 3-Wege-Verbindungsaufbaus

- **TP4_STATE_CONNECTING** ist ein Startzustand, der dem zur Verbindung gehörenden *CConnection*-Objekt bei dessen Neuerzeugung in *CTP4Layer::Connect()* zugewiesen wird. In diesem Zustand wartet eine im Aufbau befindliche, durch einen lokalen, übergeordneten Prozess initiierte Verbindung auf die Antwort auf die versendete *Connection Request*-TPDU. Die Antwort ist entweder ein *Connection Confirm* oder ein *Disconnect Request*. Entsprechend wechselt die Verbindung bei Eintreffen des Ersteren nach TP4_STATE_CONNECTED oder reagiert bei Letzterem mit dem Versenden eines *Disconnect Confirm* und anschließendem Löschen aller zu dieser Verbindung gehörenden Ressourcen.
- Analog ist **TP4_STATE_WAITFORCONNACK** der Startzustand, wenn die entfernte Seite über eine *Connection Request*-TPDU einen Verbindungsaufbauwunsch signalisiert. In diesem Fall entsteht das neue *CConnection*-Objekt in *CTP4Layer::OnConnectionRequest()* und wird dort in den Zustand **TP4_STATE_WAITFORCONNACK** geschaltet. Der weitere Ablauf richtet sich nach dem lokalen, übergeordneten Prozess, der auf das erhaltene *Connection Request* entweder mit *CTP4SAP::ConnectResponse()* oder *CTP4SAP::DisconnectRequest()* zu reagieren hat. Akzeptiert er mit der erstgenannten Methode die Verbindungsanfrage, sendet SENDCP eine *Connection Confirm*-TPDU an die Gegenseite und verharrt weiterhin in TP4_STATE_WAITFORCONNACK, bis der 3-Wege-Verbindungsaufbau durch den Empfang eines *Acknowledge* abgeschlossen ist. Danach wechselt der Verbindungszustand nach TP4_STATE_CONNECTED. Andernfalls wird der Gegenseite mittels einer *Disconnect Request*-TPDU die Ablehnung der Verbindung signalisiert.

2. Der Zustand während einer bestehenden Verbindung

- Der Zustand **TP4_STATE_CONNECTED** ist der Arbeitszustand während einer bestehenden Verbindung. Hier können beide Partner beliebig Nutzdaten-TPDUs austauschen. Auch die beschriebenen Mechanismen für das Senden und Empfangen von *Acknowledges* mitsamt der Timer-Verwaltung laufen in diesem Zustand ab. Regulär verlassen wird der Zustand, wenn einer der beiden Partner einen Verbindungsabbau auslöst. Ist dessen Initiator die entfernte Seite, wird die empfangene *Disconnect Request*-TPDU mit einem *Disconnect Confirm* quittiert. Für gegebenenfalls von der Gegenseite empfangene, aber noch nicht bestätigte TPDUs wird vorab ein *Acknowledge* gesendet. Abschließend erfolgt das Löschen der Verbindungsressourcen. Veranlasst die lokale Seite den Verbindungsabbau, erfolgt die weitere Verarbeitung nach Zustandswechsel zu TP4_STATE_DISCONNECTING. Bei Fehlern nach Ablauf eines Wiederholungszählers bricht SENDCP selbsttätig die Verbindung ab und sendet ein *Disconnect Request*. Da ein *Disconnect Confirm* darauf vermutlich nicht mehr eingeht, löscht SENDCP unmittelbar alle zugehörigen Ressourcen.

3. Zustände während des Verbindungsabbaus

- Der Zustand **TP4_STATE_DISCONNECTING** wird aktiv, nachdem der lokale übergeordnete Prozess über *CTP4SAP::DisconnectRequest()* den Verbindungsabbau auslöste. Der Methode kann ein *Disconnect Reason* übergeben werden, der zunächst in *CConnection::m_enDisconnectReason* geparkt wird. Enthält das Sendefenster der Verbindung nach Wechsel in diesen Zustand keine auf *Acknowledge* wartenden Telegramme, erfolgt unmittelbar der Übergang nach TP4_STATE_DISCONNECTED. Ansonsten verharrt die Verbindung in **TP4_STATE_DISCONNECTING**, bis alle wartenden Telegramme bestätigt sind. Der Zustandswechsel nach TP4_STATE_DISCONNECTED erfolgt dann nach Eingang des letzten erforderlichen *Acknowledge.* Siehe dazu den Ablaufplan in Abbildung 6.16.
- Wechselt der Verbindungszustand nach **TP4_STATE_DISCONNECTED**, bewirkt dies das Versenden einer *Disconnect Request*-TPDU. Die Verbindung wartet in diesem Zustand nun noch auf den Empfang des zugehörigen *Disconnect Confirm*, worauf die Verbindung regulär mit dem Löschen der belegten Ressourcen beendet ist.

Durch die Aussende-Verzögerung der *Disconnect Request*-TPDU bis zur Bestätigung aller wartenden Telegramme im Sendefenster weicht diese TP4-Implementierung von der ISO/OSI 8073-Spezifikation ab. Sieht letztere einen abrupten Verbindungsabbruch vor, ist hier zur Vermeidung von Datenverlusten ein aufwändigerer Mechanismus implementiert. Fehler in höheren Protokollschichten, aufgrund derer ein verfrühter Verbindungsabbau ausgelöst würde, sind so unterdrückbar.

Spätestens beim Betrachten der obigen Zustandsdiagramme mag auffallen, dass auch die TPDUs zur Verbindungssteuerung, wie *Connection Request* und *Disconnect Request*, eine Timeout-Zeit und einen Wiederholungszähler haben. Auch sie werden intern als *CTP4Telegram*-Objekt repräsentiert und in der Sendefensterliste gespeichert, bis die zugehörige *Confirm*-TPDU eintrifft. Die beschriebenen Abläufe für Timeout- und Sendewiederholungsabwicklung sind identisch.

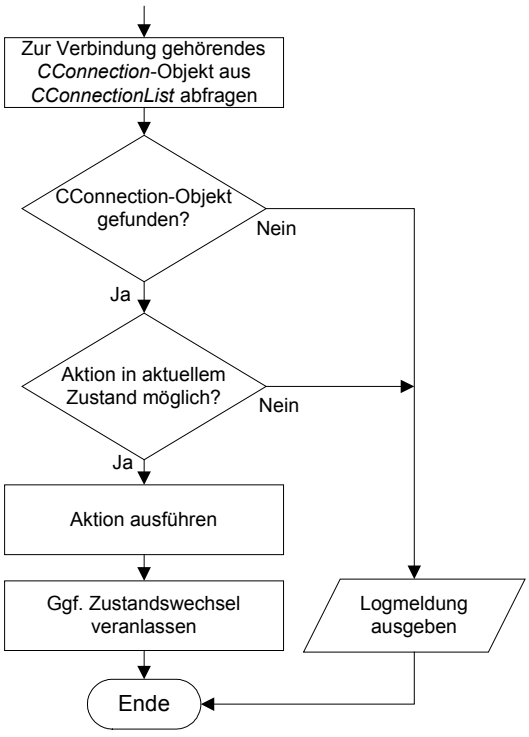


Abbildung 6.22: Prinzipieller Ablauf der CTP4Layer-Request- und Indication-Methoden

Zustandsübergänge können ausgelöst werden in den *Request*- und *Indication*-Methoden der Klasse *CTP4Layer,* wie Abschnitt 6.5.2.5, *„Die Klasse CTP4Layer“* erläutert. Diese Methoden folgen im Prinzip dem nebenstehenden Ablauf. Ausnahmen sind *OnConnectionRequest()* und *Connect()*, da in diesen eine neue Verbindung nebst *CConnection*-Objekt erst entsteht und dies von keinem Zustand abhängt.

Jeder Zustandswechsel wird ausgeführt über die Methode *CTP4Layer::SetTP4State().* Sie speichert den neuen Zustand im zugehörigen *CConnection*-Objekt ab und führt beim Zustandsübergang auszuführende Funktionen gemäß des Ablaufplans in Abbildung 6.23 aus.

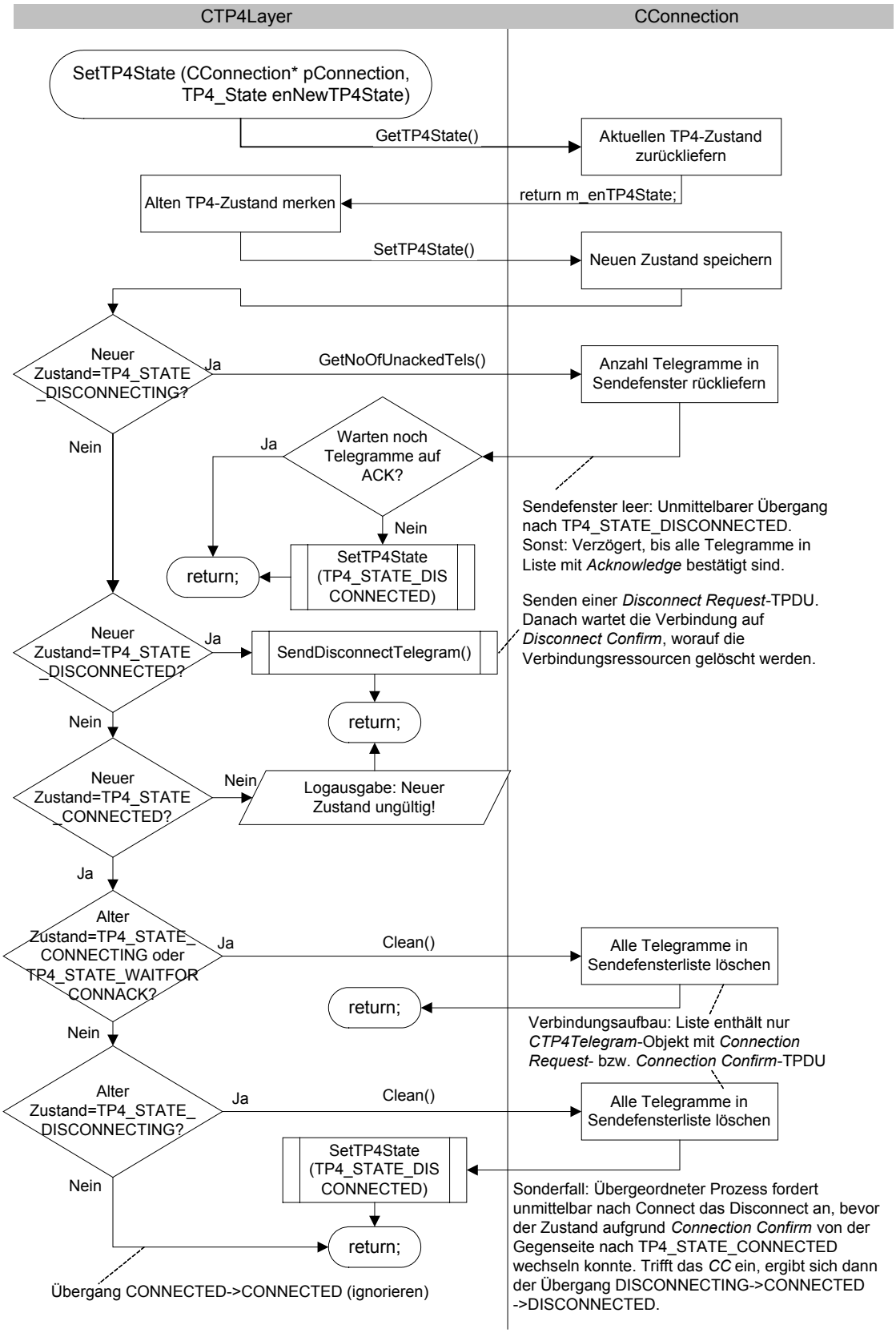


Abbildung 6.23: Zustandssteuerung in CTP4Layer::SetTP4State()

6.7 Diskussion

An dieser Stelle ist die Beschreibung des Prozesses SENDCP abgeschlossen. Nun erfolgt noch einmal ein Blick zurück auf einige Aspekte der gefundenen Lösungen, um sie einer Analyse zu unterziehen und Ideen für zukünftige Weiterentwicklungen zu bieten.

6.7.1 Analyse des erwarteten Laufzeitverhaltens

Netzwerkprotokolle erfordern das Einhalten verschiedener Zeitvorgaben, wie die Bestätigung von Datentelegrammen innerhalb der gewährten *Acknowledge Time.* Selbstverständlich sollten außerdem alle Verarbeitungszeiten zur Erreichung eines hohen Datendurchsatzes möglichst kurz sein. Dies bedingt den Verzicht auf zeitintensive Algorithmen oder Programmschleifen.

Die vorgestellten SENDCP-Funktionen bestehen zumeist aus einfachen Operationen und werden geradlinig abgearbeitet. Programmschleifen, aufgrund derer Programmteile in Abhängigkeit einer Eingangsgröße wiederholt würden, kommen nicht vor. Somit ist die Laufzeit der SENDCP-Operationen zur TPDU-Erstellung oder -Auswertung konstant.

Ausnahme ist das notwendige Auffinden des zu einer Verbindung gehörenden *CConnection*-Objekts in der durch *CConnectionList* gebildeten Verbindungsliste. Wie Abbildung 6.22 verdeutlicht, wird diese Operation im laufenden Betrieb innerhalb jeder *CTP4Layer-Indication-* oder *Request*-Methode ausgeführt, demnach zwingend auch bei jedem Versenden oder Empfangen einer TPDU. Die *CConnectionList*-Operation *Find()* durchläuft dazu die Verbindungsliste beginnend vom ersten *CConnection*-Objekt solange, bis die Verbindungsnummer des Objekts an der aktuellen Position mit der gesuchten Verbindungsnummer übereinstimmt. Da die benötige Zeit im Schnitt linear mit der Anzahl bestehender Verbindungen steigt, ist die lineare Laufzeit-Komplexität $O(n)$ zu erwarten.

Verbessern ließe sich dies, wenn die Klasse *CConnectionList* anstelle von der STL-Klasse *list* von *map* abgeleitet wäre. Wie in Kapitel 2.3.2.1, *„Die Klasse CKomPartners“,* auf Seite 26 beschrieben, ließe sich die Laufzeitkomplexität damit auf logarithmisches Maß $O(\log n)$ reduzieren. Bei den zu erwartenden maximal fünfzig Verbindungen ist dieser theoretische Vorteil allerdings für die Praxis irrelevant, wie das nächste Kapitel 7, *„Performance-Messungen“,* zeigen wird. Auf den nötigen Aufwand zur Umstellung wurde daher verzichtet.

6.7.2 Analyse des Software-Designs

TPDUs werden im gewählten Entwurf in verschiedenen *CTP4Layer*-Methoden als Bytefeld erstellt und diese dann in einem *CTP4Telegram*-Objekt abgelegt. Dieser Ansatz vermischt das Wissen um die Struktur der TPDUs mit den zur Kommunikation durchzuführenden Abläufen. Im Hinblick auf einen konsequent objektorientierten Software-Entwurf wäre es eleganter, für jeden möglichen TPDU-Typen eine eigene Klasse vorzusehen und dieser die TPDU-Erstellung zu überlassen. Objekte von diesen Klassen ließen sich dann an jeder beliebigen Codestelle instanziieren und benutzen. Obsolet wären fortan spezielle *CTP4Layer*-Methoden wie *SendDisconnectTelegram()* oder *SendAckTelegram()*. Diese Idee verdeutlicht Abbildung 6.24.

Abbildung 6.24: TPDUs objektorientiert

Nach wie vor kennt dieser Entwurf die Klasse *CTP4Telegram*. Allerdings dient diese nun als abstrakte Basisklasse, von der niemals direkt ein Objekt angelegt wird. Sie fasst die Methoden und Variablen zusammen, die allen TPDUs unabhängig von ihrem Typen gemeinsam sind. Dazu gehört zum einen natürlich die Verwaltung eines Buffers mit der eigentlichen TPDU-Struktur (Header und Nutzdaten), zum anderen die für Timeout-Überwachung und Sendewiederholung erforderlichen Elemente.

Von dieser Basisklasse abgeleitet werden dann die auf einen bestimmten TPDU-Typen spezialisierten Klassen, wie *CConnectionConfirmTPDU*, *CDataTPDU,* etc. Jede dieser abgeleiteten Klassen ist ergänzt um spezifische Membervariablen, die jeweils den individuellen TPDU-Parametern im festen oder variablen Teil entsprechen. Der Konstruktor belegt diese, wo sinnvoll, mit Standardwerten vor, etwa 500ms für die *Acknowledge Time* in *CConnectionRequestTPDU*.

Ein Codeabschnitt, der die Erstellung einer TPDU erfordert, könnte nun beispielsweise ein neues *CConnectionConfirmTPDU*-Objekt erzeugen und dessen Membervariablen mit den gewünschten Verbindungsparametern versorgen. Ein Aufruf der spezifischen Methode *Build()* erstellt dann mit den Werten aus den Membervariablen ein Bytefeld mit der Header- und Datenstruktur der jeweiligen TPDU. Versendet würde solch ein TPDU-Objekt wie gehabt über *CTP4Layer::AddTel2ListAndSend().* Analog diente die Methode *Extract()* zum Füllen der Membervariablen mit den Daten aus dem Bytefeld einer empfangenen TPDU. Auf diese Art ließen sich auch die bislang nicht unterstützten Parameter aus den variablen Teilen der TPDUs bei zukünftigem Bedarf leicht berücksichtigen. Die Membervariablen der TPDU-Klassen kommen so dem *Interface Control Information (ICI)*-Konzept aus dem OSI-Modell recht nahe.

7 Performance-Messungen

7.1 Ziel der Messungen

Vor der Auslieferung und Inbetriebsetzung des neuen Systems beim Endkunden wurde das Laufzeitverhalten des Prozesses SENDCP eingehenden Untersuchungen unterzogen. Ziel dieser Untersuchungen war es insbesondere, eventuelle Schwierigkeiten im Zeitverhalten zu erkennen und zu beheben. Besonderes Augenmerk lag auf der unbedingten Einhaltung der *Acknowledge Time*, also der rechtzeitigen Quittierung eingegangener Daten-TPDUs. Ein weiterer Aspekt war die Analyse der praktischen Auswirkungen der linearen Suche durch die Verbindungsliste, wie in Abschnitt 6.7.1 angedeutet.

7.2 Versuchsaufbau

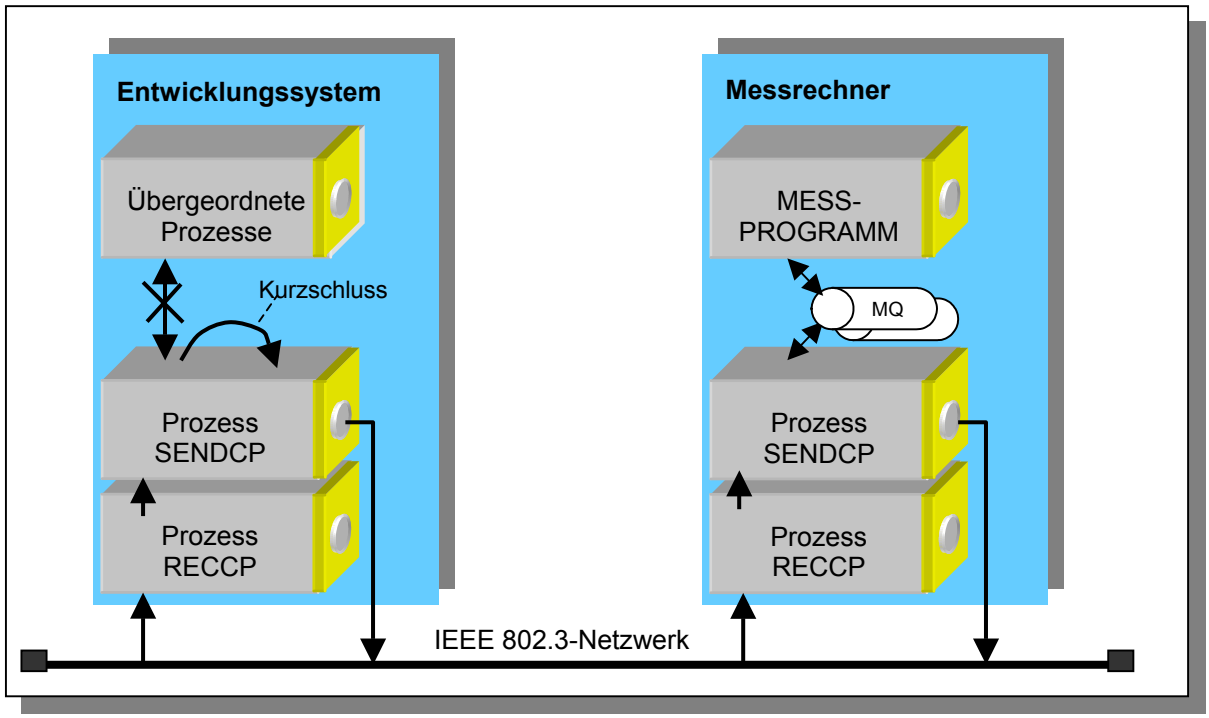


Abbildung 7.1: Versuchsaufbau für Performance-Messungen

Für sämtliche Messungen lief der zu analysierende Prozess SENDCP auf dem auch für die gesamte Systementwicklung benutzten Rechner. Alle am Projekt beteiligten Entwickler arbeiteten parallel per Remote-Zugang auf dieser Maschine. Der Entwicklungsbetrieb lief auch während der Messreihen weiter, so dass die gewonnenen Ergebnisse das Verhalten bei vorhandener Systemlast widerspiegeln.

SENDCP akzeptiert eingehende *Connection Requests* normalerweise erst dann, wenn ein übergeordneter Prozess dies per *CTP4SAP::ConnectResponse()* anfordert. Allerdings erlauben die vorhandenen, zuständigen Kommunikationsabwickler-Prozesse mit je einem entfernten System jeweils nur genau eine Verbindung, was den angestrebten vielfachen Verbindungsaufbau unmöglich macht.

Um außerdem Verfälschungen der Messergebnisse durch Laufzeiten der übergeordneten Prozesse zu vermeiden, wurde SENDCP derart modifiziert, dass übergeordnete Prozesse nicht mehr angesprochen werden. Stattdessen sind die Methoden für eingehendes *Connection Request* und ausgehendes *Connection Confirm* „kurzgeschlossen“, wodurch SENDCP nun unmittelbar mit dem Aussenden eines *Connection Confirm* den Verbindungsaufbau akzeptiert. Der für die Vorverarbeitung eingehender TPDUs erforderliche Prozess RECCP blieb unverändert.

Die Durchführung der Zeitmessungen erfolgte mit Hilfe eines separaten Messrechners im Netz. Auf diesem befanden sich die im wesentlichen unmodifizierten Versionen von SENDCP und RECCP. Ein zusätzliches Messprogramm auf Framework-Basis löste per Message Queue den Verbindungsaufbau und das Versenden von TPDUs aus. Die benötigte Zeit bis zum Eintreffen der Antwort wurde gemessen und in einer Logdatei festgehalten. Abbildung 7.1 zeigt den gesamten Versuchsaufbau.

Zwangsweise gehen durch diese Versuchsanordnung sowohl die benötigten Verarbeitungszeiten innerhalb des Messrechners als auch die Laufzeiten auf dem Netzwerk in die Messergebnisse ein. Ziel der Messreihen war allerdings weniger, absolute Verarbeitungszeiten für den Prozess SENDCP zu ermitteln. Vielmehr sollte, wie erwähnt, die Einhaltung der geforderten Zeitlimits, also insbesondere der *Acknowledge Time*, verifiziert werden. Gemäß Abschnitt 6.6.1.2, *„Datenempfang und Acknowledge Timer“*, Seite 85, soll SENDCP empfangene Daten-TPDUs spätestens nach 200ms mit einem *Acknowledge* bestätigen, um das von den angeschlossenen Arbeitsplätzen gewährte Zeitlimit von 500ms sicher einzuhalten. Gegenüber diesen Zeiten sind Netzlaufzeiten und die rechnerinternen Verarbeitungszeiten vernachlässigbar. Ähnliches gilt für die Größenordnung möglicher Messfehler: Die Zeitmessungen wurden mit Hilfe des Linux-Systemaufrufs *gettimeofday()* realisiert. Dieser arbeitet mit Mikrosekunden als kleinster Zeiteinheit. Deren Genauigkeit sollte durch einen Hardware-Taktgeber bestimmt sein und daher mindestens im Mikrosekunden-Bereich liegen.

7.3 Acknowledge-Timing

Zunächst die essentiellste aller Messungen: Bestätigt SENDCP innerhalb der gewährten *Acknowledge Time* zuverlässig alle empfangenen Telegramme? Dies auch dann, wenn es sich um die letzte Verbindung in der Verbindungsliste handelt, also zunächst jedes Mal über die gesamte *CConnectionList* iteriert werden muss, um das zugehörige *CConnection*-Objekt zu finden?

Zur Beantwortung dieser Frage wurde das Messprogramm so angepasst, dass es 500 parallele Verbindungen zum SENDCP-Prozess auf dem Entwicklungsrechner aufbaut. Diese Verbindungsanzahl liegt um den Faktor zehn höher als im realen System maximal zu erwarten.

Über die letzte aufgebaute Verbindung erfolgte anschließend das Versenden von 200 Daten-TPDUs. Nach jeder TPDU wartete das Messprogramm zunächst auf das zugehörige *Acknowledge*, ermittelte die benötigte Zeit und schickte erst dann die nächste TPDU. Das *Acknowledge* sollte daher nach Ablauf des *Acknowledge Timers*, also nach 200ms, von SENDCP versendet werden.

Messergebnis für 200 TPDUs über die 500. Verbindung:

Durchschnittliche Acknowledge-Zeit für eine einzeln gesendete TPDU: 202.35ms

Die ermittelte durchschnittliche Zeit von 202,35ms liegt um 1,17% über den erwarteten 200ms und damit deutlich innerhalb der gewährten Obergrenze von 500ms.

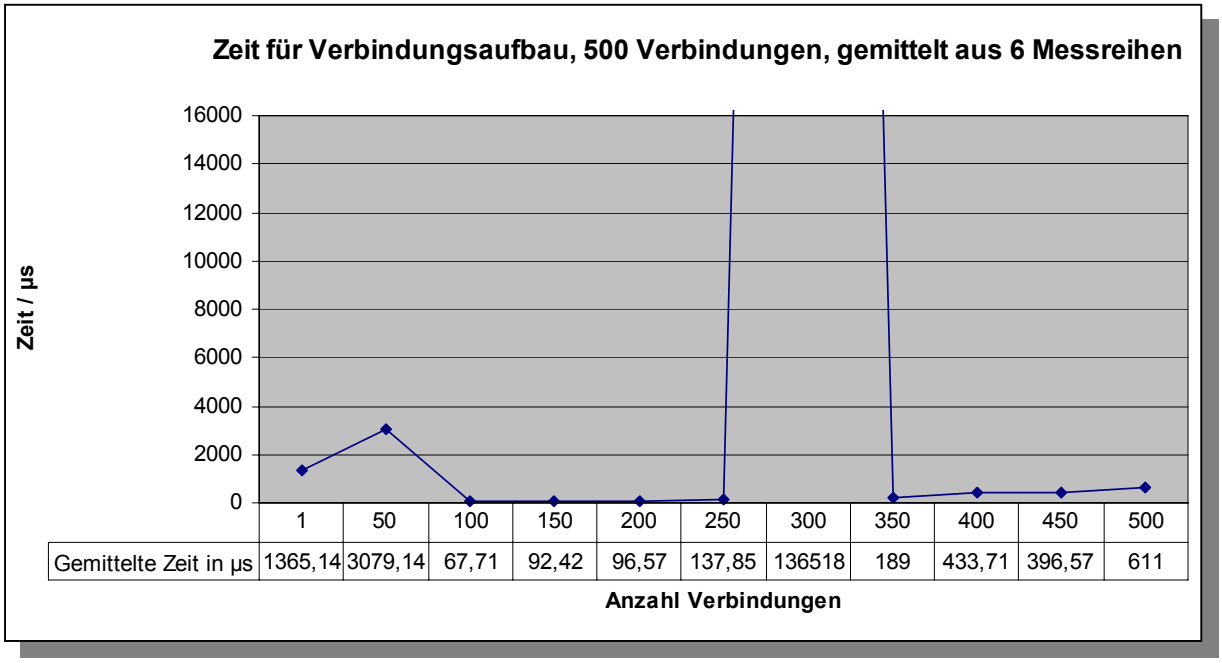
Werden statt einer TPDU acht Stück unmittelbar nacheinander gesendet, soll SENDCP nicht bis zum Ablauf des *Acknowledge Timers* warten, sondern sofort ein *Acknowledge* senden, da die Gegenseite den Sendekredit aufgebraucht hat. Eine entsprechende Messung verifiziert dies mit folgendem Ergebnis:

Messergebnis für 200x8 TPDUs über die 500. Verbindung:

Durchschnittliche Acknowledge-Zeit bei acht gesendeten TPDUs: 31.69ms

Dieser Wert ist nun nicht durch Timer, sondern durch Lauf- und Verarbeitungszeiten bedingt.

7.4 Einfluss der linearen Verbindungsliste



	1	50	100	150	200	250	300	350	400	450	500
Gemittelte Zeit in µs	1365,14	3079,14	67,71	92,42	96,57	137,85	136518	189	433,71	396,57	611

Abbildung 7.2: Mittlere Zeit für Verbindungsaufbau in Abhängigkeit von der Verbindungsanzahl, nicht optimiert

Die Untersuchung des Einflusses der linearen Suche über die Verbindungsliste fand mit einigen weiteren Messungen statt.

Für die erste Messreihe wurden mit Hilfe des Messprogramms erneut 500 Verbindungen aufgebaut. Nach jeder fünfzigsten Verbindung erfolgte eine Messung der Zeit zwischen dem Aussenden des *Connection Request* und dem Eintreffen des *Connection Confirm*.

Der auszumessende SENDCP-Prozess erzeugt bei jedem Eingang eines *Connection Request* ein neues *CConnection*-Objekt und hängt es an die Verbindungsliste. Das einfache Anfügen an das Listenende erfordert noch kein Durchlaufen der Liste.

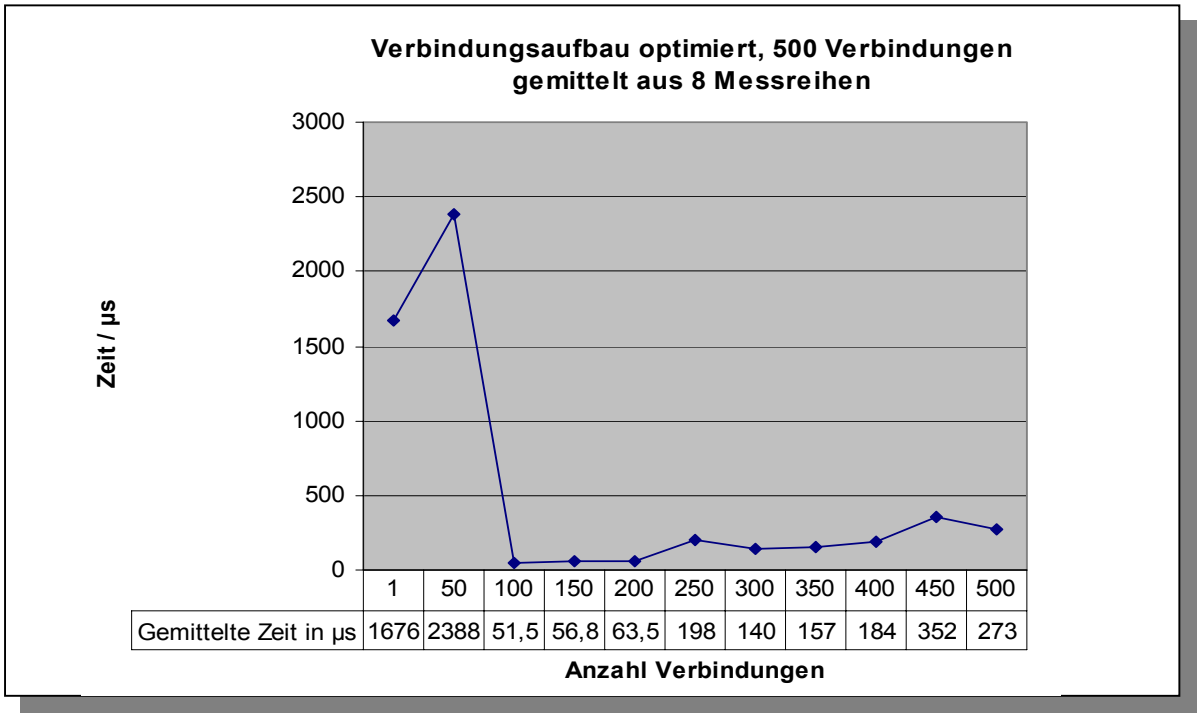
Durch den eingefügten „Kurzschluss“ folgt jedoch unmittelbar nach Abarbeitung des *Connection Request* der Aufruf von *CTP4Layer::SendConnectionConfirm()*. Innerhalb dieser Methode wird nun die Verbindungsliste vorne beginnend durchsucht, bis sich schließlich, an deren Ende, das gerade zuvor angehangene *CConnection*-Objekt findet. Erst dann kann über die zugehörige Verbindung das *Connection Confirm* gesendet werden.

Abbildung 7.2 zeigt die benötigte Zeit bis zum Eintreffen des *Connection Confirm* in Abhängigkeit von der Anzahl bestehender Verbindungen. Die dargestellten Zeiten sind die Mittelwerte aus sechs durchgeführten Messreihen. Auffällig ist der Ausbruch bei der 300. Verbindung, ausgelöst durch einen Spitzenwert von 954ms bei einer der sechs Messreihen. Wahrscheinliche Ursache ist ein gleichzeitig laufender Kompiliervorgang, gestartet durch einen zweiten Benutzer auf der gleichen Maschine.

Hier zeigt sich, welche Einflüsse um die verfügbare Rechenzeit konkurrierende Prozesse haben können. Linux als normalerweise nicht echtzeit-fähiges System weist ohne weitere Maßnahmen allen Benutzerprozessen die gleiche Priorität zu. Welcher von diesen Prozessen wann ausgeführt wird, ist nicht vorhersagbar.

Natürlich sind solche Timing-Verzögerungen bei zeitkritischen Anwendungen wie Netzwerkprotokollen nicht hinnehmbar. Linux ermöglicht daher darauf angewiesenen Prozessen mit dem Systemaufruf `sched_setscheduler()` die Aktivierung eines Realtime-Scheduling-Algorithmus. Dabei kann eine Prozesspriorität von 0 (normaler User-Prozess) bis 99 (extrem hochprior) angegeben werden. Die beiden Prozesse SENDCP und RECCP laufen seither mit der Priorität 60.

Noch eine weitere Maßnahme zur Vermeidung von Timing-Schwierigkeiten wurde getroffen: Mit `mlockall()` lässt sich verhindern, dass einmal durch einen Prozess allokierte Speicherseiten wieder freigegeben oder ausgelagert werden. Auf diese Art ist zeitaufwändiges Speichermanagement nur noch beim erstmaligen Neubelegen von Speicherressourcen erforderlich. Weitere Informationen zu den genannten Systemaufrufen finden sich auf den zugehörigen Linux-Manpages.



Anzahl Verbindungen	1	50	100	150	200	250	300	350	400	450	500
Gemittelte Zeit in µs	1676	2388	51,5	56,8	63,5	198	140	157	184	352	273

Abbildung 7.3: Mittlere Zeit für Verbindungsaufbau in Abhängigkeit von der Verbindungsanzahl, optimiert

Wie sehr die ergriffenen Maßnahmen das Zeitverhalten von SENDCP optimieren, zeigt Abbildung 7.3, gebildet aus den gemittelten Messwerten von acht Messreihen. Geblieben ist die Spitze bei Verbindung 50, mit einer gemittelten Zeit von 2,4ms. Diese Spitze ist reproduzierbar bei jeder Messreihe. Ursache ist vermutlich die nötige Erstallokation von Speicher, da Linux diesen möglicherweise vorausschauend „auf Vorrat“ reserviert, um den Aufwand bei weiterem Bedarf zu minimieren.

Erkennbar ist in Abbildung 7.3 auch, wie erwartet, die mit der Verbindungsanzahl linear ansteigende benötigte Zeit für den Verbindungsaufbau. Allerdings ist selbst der Spitzenwert von 2,4ms bei Verbindung 50 weit innerhalb des tolerierbaren Bereichs. Dies gilt selbstverständlich auch für die bei der 500. Verbindung erforderliche Zeit von rund 300µs.

7.5 Stresstest mit 1000 TPDUs

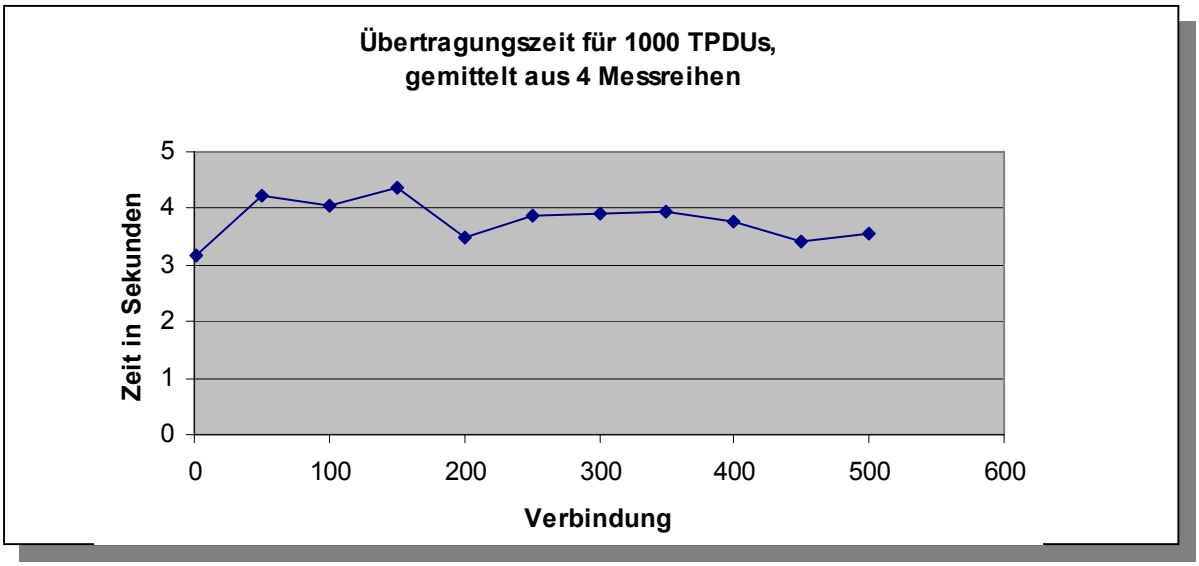


Abbildung 7.4: Mittlere Übertragungszeit von 1000 TPDUs in Abhängigkeit von der Verbindung

Zweck einer abschließenden Messreihe war vorwiegend die Prüfung von RECCP und SENDCP auf Stabilität. Eventuell noch vorhandene sporadische Softwarefehler sollte ein „Stresstest“ zur rechtzeitigen Beseitigung offen legen.

Wiederum wurden 500 Verbindungen aufgebaut. Gemessen wurde die benötigte Übertragungszeit für 1000 TPDUs als Funktion der benutzten Verbindung. Abbildung 7.4 zeigt die gemittelten Ergebnisse aus vier durchgeführten Messreihen.

In der Ergebniskurve ist keinerlei Auswirkung der linearen Suche mehr erkennbar. Vielmehr dominieren durch Verarbeitungs-, Scheduling- und Übertragungszeiten bedingte Schwankungen, die bei 1000 übertragenen TPDUs durchaus eine Schwankungsbreite von gut einer Sekunde verursachen können.

Fazit: Die Auswirkungen der linearen Suche sind zwar messtechnisch mit den Ergebnissen aus Abbildung 7.3 nachvollziehbar, haben in der Praxis aber keine Bedeutung. Bedeutungsvoll ist dagegen, dass RECCP und SENDCP den Beschuss mit rund 45000 TPDUs nach der Behebung kleinerer Schönheitsfehler reibungslos verkrafteten.

8 Resümee

Mit dem Erreichen dieser Zeilen sind die in dieser Masterarbeit besprochenen Komponenten unbeanstandet seit gut vier Wochen im ununterbrochenen Kundeneinsatz. Dies ist zweifellos das Erfreulichste und Wichtigste, was in diesem Schlusswort nach eineinhalb Jahren Projektarbeit und hundert vormals leeren Seiten festgehalten werden kann.

Dass diese Masterarbeit von meiner Berufstätigkeit und natürlich dem zugrunde liegenden IT-Projekt profitierte, liegt auf der Hand. Umgekehrt hat aber auch die Masterarbeit ihren unbestreitbaren Anteil am Erfolg des Projekts. Neben der durch sie erreichten permanenten Hinterfragung und Bestätigung des Lösungskonzepts führte sie zu einer besonders intensiven Beschäftigung mit der Netzwerk-Materie und den theoretischen Hintergründen. Zweifellos trugen auch die durchgeführten Messreihen und die in deren Verlauf ausgemerzten Schönheitsfehler zur Qualitätssicherung des ausgelieferten Systems bei. Statt der oftmals zitierten Diskrepanz zwischen akademischer Arbeit und Berufspraxis gingen beide auf diese Art eine gewinnbringende Symbiose ein.

Natürlich ist aber ein solches IT-Projekt vor allem durch eines bestimmt: Geld – und dazu proportional verhält sich die verfügbare Zeit. So lag das Hauptaugenmerk während der Realisierungsphase definitiv auf dem schnellen Erreichen eines lauffähigen Software-Stands. Schließlich war es für meine Projektkollegen und deren Arbeitsfortschritt unabdingbar, möglichst frühzeitig Verbindung mit dem uns zur Verfügung gestellten Arbeitsplatz aufnehmen zu können. Aus dieser Notwendigkeit resultiert das eher prozedural denn objektorientierte Konzept der Klasse *CTP4Layer* mit ihren Kommunikationsmethoden und den darin als statische Bytefelder angelegten TPDUs.

Ein reines „Masterprojekt“ ohne Business-Hintergrund hätte sicherlich an einigen Stellen eine eingängigere, leichter in Worten fassbare Struktur gehabt. Dazu gehört das in Kapitel 6.7.2, *„Analyse des Software-Designs“,* auf Seite 94 als Idee vorgestellte objektorientierte TPDU-Konzept. Aber auch kosmetische Dinge wie eine konsistentere Methoden-Benamung zählen dazu.

Ganz bewusst wurde allerdings auf eine nachträgliche Änderung und damit der Schaffung einer „Masterthesis-Version“ der Software verzichtet: Bleibt mir doch am Ende das gute Gefühl, mit meiner Arbeit nicht nur den „akademischen Bücherschrank“, sondern auch jenen von Kollegen und Kunden zu füllen, in der Hoffnung, ein auch in Zukunft hilfreiches Nachschlagewerk geschaffen zu haben. Schließlich haben mögliche Folgeprojekte mit dem Framework und den Kommunikationsprozessen nun eine universelle, funktionstüchtige Basis, auf der mit den beschriebenen Ideen und Konzepten weiter aufgebaut werden kann.

Für die Zukunft jedenfalls bin ich nach dieser Masterarbeit optimistisch.

9 Literaturverzeichnis

9.1 Weblinks & Online-Dokumente

[Moning] Moning, Adrian; Lanz, Rolf: „Datenkommunikation und Rechnernetze", Mai 2003, http://www.hta-be.bfh.ch/~lanz/SKRIPTE/DComm.pdf

9.2 Gedrucktes

[Proebster02] Proebster, Walter: „Rechnernetze", 2. Auflage, Oldenbourg 2002

[Tanenbaum92] Tanenbaum, Andrew S.: „Computernetzwerke", 2. Auflage, Wolfram's Fachverlag 1992

[Tanenbaum00] Tanenbaum, Andrew S.: „Computernetzwerke", 3. Auflage, Prentice Hall 2000

[Detttmar02] Dettmar, Uwe: „Telekommunikationssysteme", Hilfsblätter zur Vorlesung, Teil A, Version 1.3, September 2002, FH Köln

[Weiden99] Weiden, Ralf: „Erweiterung einer PID-Digitalreglersoftware einschließlich TCP/IP-Kopplung an einen PC", Diplomarbeit 1999, FH Köln, Institut für Nachrichtenverarbeitung und Prozessautomatisierung

[Weiden03] Weiden, Ralf: „Fahrgastinformationssystem SiMAP®-FIS", Praxissemester-Bericht Februar 2003, FH Köln

[Vogt01] Vogt, Carsten: „Betriebssysteme", Spektrum Verlag 2001

[Siemens88] „International Standard ISO 8073: Verbindungsorientiertes Transportprotokoll", Schulungsunterlagen, Siemens Trainingscenter für Automatisierung, August 1988

[Stroustrup00] Stroustrup, Bjarne: „Die C++ Programmiersprache", 4. erw. Auflage, Addison-Wesley 2000

[Oesterreich01] Oesterreich, Bernd: „Objektorientierte Software-Entwicklung. Analyse und Design mit der Unified Modeling Language", Oldenbourg 2001

[Gamma96] Gamma, Helm, Johnson, Vlissides: „Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software", Addison-Wesley 1996