Peter Mandl

TCP und UDP Internals

Protokolle und Programmierung

Springer Vieweg

TCP und UDP Internals

Peter Mandl

TCP und UDP Internals

Protokolle und Programmierung

Springer Vieweg

Peter Mandl
Hochschule München
München, Deutschland

ISBN 978-3-658-20148-7 ISBN 978-3-658-20149-4 (eBook)
https://doi.org/10.1007/978-3-658-20149-4

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

Springer Vieweg

Gedruckt auf säurefreiem und chlorfrei gebleichtem Papier

Springer Vieweg ist ein Imprint der eingetragenen Gesellschaft Springer Fachmedien Wiesbaden GmbH und ist Teil von Springer Nature
Die Anschrift der Gesellschaft ist: Abraham-Lincoln-Str. 46, 65189 Wiesbaden, Germany

Vorwort

Netzwerke sind die Basis für verteilte Systeme. Zur Kommunikation verteilter Anwendungskomponenten hat sich die TCP/IP-Protokollfamilie in der Praxis als De-facto-Standard durchgesetzt und ist heute State of the art. Das globale Internet mit seinen Anwendungen ist ohne TCP/IP heute nicht mehr vorstellbar. TCP/IP-Implementierungen sind auf allen wichtigen Betriebssystemen verfügbar. Insbesondere die Transportprotokolle TCP und UDP, deren Nutzung in gängigen Betriebssystemen über die Socket API ermöglicht wird, nehmen für die Anwendungsentwicklung eine Sonderstellung ein. Viele verteilte Anwendungssysteme wurden und werden auf Basis der Socket API implementiert.

In diesem Buch werden neben den Grundlagen, Konzepten und Standards auch vertiefende Aspekte der Transportschicht auf Basis der beiden Protokolle TCP und UDP vermittelt. Das Buch soll dazu beitragen, den komplexen Sachverhalt bis ins Detail verständlich zu machen. Dabei liegt der Schwerpunkt vor allem auf praktisch relevante Themen, aber auch die grundlegenden Aspekte sollen erläutert werden. Das Buch behandelt die folgenden Themenkomplexe:

1. Einführung in die Grundbegriffe der Datenkommunikation
2. Grundkonzepte der Transportschicht
3. Transportprotokoll TCP (Transport Control Protocol)
4. Transportprotokoll UDP (User Datagram Protocol)
5. Programmierung von TCP- und UDP-Anwendungen

Kap. 1 gibt in Kürze eine grundlegende Einführung in die wichtigsten Grundbegriffe der Datenkommunkation und in heute übliche Referenzmodelle wie beispielsweise das TCP/IP-Referenzmodell. Anhand einiger Beispiele wird aufgezeigt, wie moderne verteilte Anwendungen prinzipiell arbeiten. Kap. 2 fasst wichtige Grundkonzepte und Protokollmechanismen, die typisch für die Transportschicht sind, zusammen. Wenn beim Leser schon grundlegende Kenntnisse zu Netzwerken und Datenkommunikation vorhanden sind und vor allem Interesse an den TCP- und UDP-Funktionen bestehen, können die ersten beiden Kapitel übersprungen werden. In Kap. 3 werden die Basismechanismen und vertiefenden Protokollmechanismen von TCP vorgestellt. Kap. 4 geht entspechend auf

UDP ein. Schließlich wird in Kap. 5 die Programmierung vor allem mit der Socket API anhand ausgewählter Beispiele erläutert und vertieft, wobei der Schwerpunkt auf der Java-Socket-Implementierung liegt.

In diesem Buch wird ein praxisnaher Ansatz gewählt. Der Stoff wird mit vielen Beispielen und Skizzen veranschaulicht. Für das Verständnis einiger Programmbeispiele sind grundlegende Kenntnisse von Programmiersprachen (C, C++, Java) nützlich, jedoch können die wesentlichen Konzepte auch ohne tiefere Programmierkenntnisse verstanden werden. Vom Leser werden ansonsten keine weiteren Grundkenntnisse vorausgesetzt.

Der Inhalt des Buches entstand zum einen aus mehreren Vorlesungen über Datenkommunikation über mehr als 15 Jahre an der Hochschule für angewandte Wissenschaften München und zum anderen aus konkreten Praxisprojekten, in denen Netzwerke eingerichtet und erprobt sowie verteilte Anwendungen entwickelt und betrieben wurden. Insbesondere das Transportsystem und die dazugehörige Transportzugriffsschnittstelle spielen in der Entwicklung und für das Verständnis verteilter Anwendungen eine große Rolle. Das Buch ist daher als Spezialisierung zu diesem konkreten Themenkomplex gedacht und stellt damit eine Fortsetzung des Buches „Grundkurs Datenkommunikation – TCP/IP-basierte Kommunikation: Grundlagen, Konzepte und Standards“ dieses Verlags dar (Mandl et al. 2010).

Bedanken möchte ich mich sehr herzlich bei unseren Studentinnen und Studenten, die mir Feedback zum Vorlesungsstoff gaben. Ebenso gilt mein Dank meinen Projektpartnern aus Industrie und Verwaltung. Den Gutachtern danke ich für ihre guten Verbesserungsvorschläge. Dem Verlag, insbesondere Frau Sybille Thelen, möchte ich ganz herzlich für die großartige Unterstützung im Projekt und für die sehr konstruktive Zusammenarbeit danken.

Fragen und Korrekturvorschläge richten Sie bitte an peter.mandl@hm.edu.
Für begleitende Informationen zur Vorlesung siehe www.prof-mandl.de.

München, Dezember 2017 Peter Mandl

Noch ein Hinweis für die Leser

In diesem Buch werden häufig *Requests for Comments (RFC*) referenziert. Dies sind frei verfügbare Dokumente der Internet-Community, welche die wesentlichen Standards des Internets, wie etwa die Protokollspezifikation von TCP und UDP beschreiben. Die Dokumentation wird ständig weiterentwickelt. Jedes Dokument hat einen Status. Nicht alle Dokumente sind Standards. Bis ein Standard erreicht wird, müssen einige Qualitätskriterien (z. B. nachgewiesene lauffähige Implementierungen) erfüllt werden. Ein RFC durchläuft dann die Zustände „Proposed Standard", „Draft Standard" und schließlich „Internet Standard". Es gibt auch RFCs, die nur der Information dienen (Informational RFC) oder nur Experimente beschreiben (Experimental RFC). Zudem gibt es RFCs mit dem Status „Best Current Practice RFC", die nicht nur der Information, sondern vielmehr als praktisch anerkannte Vorschläge dienen. Schließlich gibt es historische RFCs, die nicht länger empfohlen werden.

Der RFC-Prozess an sich, auch IETF Standards Process (Internet Engineering Task Force) genannt, ist in einem eigenen RFC mit der Nummer 2026 (The Internet Standard Process – Revision 3) definiert.

Jeder RFC erhält eine feste und eindeutige Nummer. Wird er ergänzt, erweitert oder verändert, wird jeweils ein RFC mit einer neuen Nummer angelegt. Eine Referenz auf die Vorgängerversion wird mit verwaltet. Damit ist auch die Nachvollziehbarkeit gegeben. Im Internet können alle RFCs zum Beispiel über https://www.rfc-editor.org/ (zuletzt aufgerufen am 04.11.2017) eingesehen werden. Eine sehr gute Übersicht über die wichtigsten RFCs zu TCP ist im RFC 7414 (A Roadmap for Transmission Control Protocol (TCP) Specification Documents) zusammengefasst.

RFCs werden in diesem Buch direkt im Text referenziert und sind nicht im Literaturverzeichnis eingetragen.

Inhaltsverzeichnis

1 Grundbegriffe der Datenkommunikation 1
1.1 Überblick 1
1.2 ISO/OSI-Referenzmodell 2
1.3 TCP/IP-Referenzmodell 6
1.4 Beispiele bekannter verteilter Anwendungen 10
1.4.1 World Wide Web 10
1.4.2 Electronic Mail 13
1.4.3 WhatsApp 16
1.4.4 Skype 18
1.5 Nachrichtenaufbau und Steuerinformation 19
Literatur 22

2 Grundkonzepte der Transportschicht 23
2.1 Grundlegende Aspekte 23
2.1.1 Typische Protokollmechanismen der Transportschicht 23
2.1.2 Verbindungsorientierte und verbindungslose Transportdienste 25
2.2 Verbindungsmanagement und Adressierung 26
2.2.1 Verbindungsaufbau 27
2.2.2 Verbindungsabbau 29
2.3 Zuverlässiger Datentransfer 33
2.3.1 Quittierungsverfahren 33
2.3.2 Übertragungswiederholung 34
2.4 Flusskontrolle 37
2.5 Staukontrolle 40
2.6 Segmentierung 40
2.7 Multiplexierung und Demultiplexierung 41
Literatur 41

3 TCP-Konzepte und -Protokollmechanismen 43
3.1 Übersicht über grundlegende Konzepte und Funktionen 43
3.1.1 Grundlegende Aufgaben von TCP 43
3.1.2 Nachrichtenlänge 45
3.1.3 Adressierung 48
3.1.4 TCP-Steuerinformation 50
3.2 Ende-zu-Ende-Verbindungsmanagement 53
3.2.1 TCP-Verbindungsaufbau 53
3.2.2 TCP-Verbindungsabbau 55
3.2.3 Zustände beim Verbindungsauf- und -abbau 56
3.2.4 Zustandsautomat des aktiven TCP-Partners 58
3.2.5 Zustandsautomat des passiven TCP-Partners 59
3.2.6 Zusammenspiel der Automaten im Detail 60
3.3 Datenübertragungsphase 64
3.3.1 Normaler Ablauf 64
3.3.2 Timerüberwachung 66
3.3.3 Implizites Not-Acknowlegde (NAK) 67
3.3.4 Flusskontrolle 68
3.3.5 Nagle- und Clark-Algorithmus 70
3.4 TCP-Protokolloptionen 72
3.4.1 Maximum Segment Size Option 73
3.4.2 TCP Window Scale Option 74
3.4.3 SACK-Permitted-Option und SACK-Option 75
3.4.4 Timestamps Option 76
3.5 Protect Against Wrapped Sequences 77
3.5.1 Problemstellung 77
3.5.2 Maßnahmen zum Schutz vor Sequenznummern-Kollisionen 79
3.6 Staukontrolle 79
3.6.1 Slow-Start und Congestion Avoidance 80
3.6.2 Fast-Recovery-Algorithmus 83
3.6.3 Explicit Congestion Notification 84
3.6.4 Weitere Ansätze der TCP-Staukontrolle 86
3.7 Timer-Management 87
3.7.1 Grundlegende Überlegung 87
3.7.2 Retransmission Timer 88
3.7.3 Keepalive Timer 90
3.7.4 Time-Wait Timer 91
3.7.5 Close-Wait Timer 92
3.7.6 Persistence Timer 92
3.8 TCP-Sicherheit 92
Literatur 94

4 UDP-Konzepte und -Protokollmechanismen 95
4.1 Übersicht über grundlegende Konzepte und Funktionen 95
4.1.1 Grundlegende Aufgaben von UDP 95
4.1.2 Adressierung 96
4.1.3 UDP-Steuerinformation 97
4.1.4 Multiplexing und Demultiplexing 97
4.2 Datenübertragungsphase 98
4.2.1 Datagrammorientierte Kommunikation 98
4.2.2 Prüfsummenberechnung 99
4.3 UDP-Sicherheit 102

5 Programmierung von TCP/UDP-Anwendungen 105
5.1 Überblick 106
5.2 Grundkonzepte der Socket-Programmierung 106
5.2.1 Einführung und Programmiermodell 106
5.2.2 TCP-Sockets 107
5.2.3 Datagramm-Sockets 110
5.3 Socket-Programmierung in C 111
5.3.1 Die wichtigsten Socket-Funktionen im Detail 111
5.3.2 Nutzung von Socket-Optionen 116
5.3.3 Nutzung von TCP-Sockets in C 118
5.3.4 Nutzung von UDP-Sockets in C 121
5.4 Socket-Programmierung in Java 124
5.4.1 Überblick über Java-Klassen 124
5.4.2 Streams für TCP-Verbindungen 128
5.4.3 Verbindungsorientierte Kommunikation über TCP 129
5.4.4 Gruppenkommunikation über UDP 130
5.5 Einfache Java-Beispiele 131
5.5.1 Ein Java-Beispielprogramm für TCP-Sockets 131
5.5.2 Ein Java-Beispielprogramm für Datagramm-Sockets 133
5.5.3 Ein Java-Beispiel für Multicast Sockets 136
5.5.4 Zusammenspiel von Socket API und Protokollimplementierung . . . 137
5.6 Ein Mini-Framework für Java-Sockets 140
5.6.1 Überblick über vordefinierte Schnittstellen und Objektklassen 140
5.6.2 Basisklassen zur TCP-Kommunikation 142
5.6.3 Single-threaded Echo-Server als Beispiel 147
5.6.4 Multi-threaded Echo-Server als Beispiel 151
5.7 Weiterführende Programmierkonzepte und -mechanismen 154
Literatur 156

6 Schlussbemerkung 157
Literatur 158

7 Übungsaufgaben und Lösungen 159
7.1 Grundbegriffe der Datenkommunikation 159
7.2 Grundkonzepte der Transportschicht 160
7.3 TCP-Konzepte und -Protokollmechanismen 162
7.4 UDP-Konzepte und -Protokollmechanismen 166
7.5 Programmierung von TCP/UDP-Anwendungen 168

A Anhang: TCP/IP-Konfiguration in Betriebssystemen 173

Weiterführende Literatur 179

Stichwortverzeichnis 181

Grundbegriffe der Datenkommunikation

1

Zusammenfassung

Kommunikation ist der Austausch von Informationen nach bestimmten Regeln. Dies ist zwischen Menschen ähnlich wie zwischen Maschinen. Das Regelwerk fasst man in der Kommunikationstechnik unter dem Begriff Kommunikationsprotokoll (kurz Protokoll) zusammen. Die nachrichtenbasierte Kommunikation in verteilten Systemen ist aufgrund der vielen Protokolldetails sehr komplex. Aus diesem Grund entwickelte man Beschreibungsmodelle, sogenannte Referenzmodelle, in denen die Komplexität durch Schichtung und Kapselung der einzelnen Funktionen überschaubarer dargestellt wurde. Zwei Referenzmodelle, das ISO/OSI-Referenzmodell und das TCP/IP-Referenzmodell, haben sich heute durchgesetzt, wobei in der Praxis die konkreten Protokolle der TCP/IP-Welt deutlich mehr genutzt werden und das Internet dominieren. Die grundlegenden Begriffe der Datenkommunikation, die Referenzmodelle und einige Fallbeispiele sollen einen Überblick über die Datenkommunikation verschaffen und dienen damit als Basis für die weitere Betrachtung der Transportmechanismen.

1.1 Überblick

Auch beim Telefonieren sind von den Beteiligten Kommunikationsregeln einzuhalten, sonst funktioniert die Kommunikation nicht. Dieses Regelwerk wird auch als Kommunikationsprotokoll oder einfach kurz als Protokoll bezeichnet. Wenn z. B. beide Teilnehmer gleichzeitig reden, versteht keiner etwas. Ähnlich verhält es sich bei der Rechnerkommunikation. Konzepte und Techniken zur Übertragung von Daten über Übertragungskanäle fasst man unter dem Begriff der *Datenkommunikation* (siehe Definition) zusammen.

P. Mandl, *TCP und UDP Internals*, https://doi.org/10.1007/978-3-658-20149-4_1

▶ **Datenkommunikation** Datenkommunikation befasst sich mit dem Transport von Daten über beliebige Übertragungskanäle. Üblicherweise erfolgt der Transport der Daten in Nachrichten.

In mehreren Gremien und Organisationen wurde in den vergangenen Jahrzehnten versucht, die komplexe Materie der Datenkommunikation in Modellen zu formulieren und zu standardisieren, einheitliche Begriffe einzuführen und schichtenorientierte Referenzmodelle für die Kommunikation zu schaffen. Einen wesentlichen Beitrag leistete bis in die neunziger Jahre hinein die ISO (International Standardization Organization), aber vor allem in den vergangenen 30 Jahren setzte die TCP/IP-Gemeinde hier die wesentlichen Akzente. In diesem Kapitel werden die beiden Referenzmodelle erläutert. Die Problematik der Kommunikation wird anhand einiger Anwendungen wie das World Wide Web (WWW), Elektronische Mail (E-Mail), Instant Messaging und IP-Telefonie (Voice over IP) eingeführt, um sich in den weiteren Kapiteln vorwiegend auf die Funktionen spezieller Transportprotokolle und deren Nutzung konzentrieren zu können.

1.2 ISO/OSI-Referenzmodell

Die gesamte Funktionalität, die hinter der Datenkommunikation steckt, ist zu komplex, um ohne weitere Strukturierung verständlich zu sein. Im Rahmen der Standardisierungsbemühungen der ISO hat man sich daher gemäß dem Konzept der virtuellen Maschinen mehrere Schichten ausgedacht, um die Materie etwas übersichtlicher zu beschreiben.

Das ISO/OSI-Referenzmodell (kurz: OSI-Modell) teilt die gesamte Funktionalität in sieben Schichten (Abb. 1.1) ein. Jede Schicht stellt der darüberliegenden Schicht bzw.

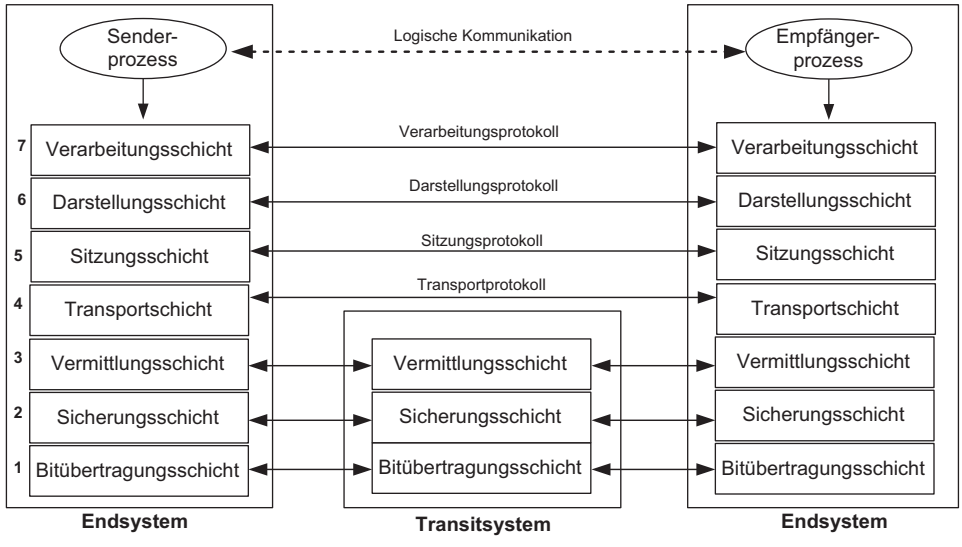


Abb. 1.1 ISO/OSI-Referenzmodell

bei Schicht 7 der Anwendung eine Schnittstelle, auch Dienst genannt, zur Nutzung ihrer Funktionen bereit. Die unterste Schicht beschreibt die physikalischen Eigenschaften der Kommunikation. Verschiedene Schichten sind je nach Netzwerk teilweise in Hardware und teilweise in Software implementiert.

In einem Rechnersystem, das nach OSI ein offenes System darstellt, werden die einzelnen Schichten gemäß den Standards des Modells implementiert. Es ist aber nicht vorgeschrieben, dass alle Schichten, und innerhalb der Schichten alle Protokolle, implementiert sein müssen. Das Referenzmodell mit seinen vielen Protokollen beschreibt keine Implementierung, sondern gibt nur eine Spezifikation vor. Die eigentliche Kommunikationsanwendung, die in einem oder mehreren Betriebssystemprozessen ausgeführt wird, gehört nicht in eine Schicht, sondern nutzt nur die Dienste des Kommunikationssystems.

▶ **Kommunikationsprotokoll** Ein Kommunikationsprotokoll oder kurz ein Protokoll ist ein Regelwerk zur Kommunikation zweier Rechnersysteme untereinander. Protokolle folgen in der Regel einer exakten Spezifikation. In der TCP/IP-Welt werden die Spezifikationen in Internet Standards (RFCs) festgehalten.

Gleiche Schichten kommunizieren horizontal über Rechnergrenzen hinweg über Kommunikationsprotokolle (siehe Definition). Da das OSI-Modell offen für alle ist, werden Rechnersysteme, die ISO/OSI-Protokolle bereitstellen, auch offene Systeme genannt. Jede Schicht innerhalb eines offenen Systems stellt der nächst höheren Schicht ihre Dienste zur Verfügung. Das Konzept der virtuellen Maschine findet hier insofern Anwendung, als keine Schicht die Implementierungsdetails der darunterliegenden Schicht kennt und eine Schicht n immer nur die Dienste der Schicht n−1 verwendet. Die Schicht, die einen Dienst bereitstellt, wird als *Diensterbringer* (siehe Definition) bezeichnet, diejenige, welche den Dienst nutzt, als *Dienstnehmer* (siehe Definition).

▶ **Diensterbringer, Dienstnehmer und Service Access Point (SAP)** Ein *Dienstnehmer* nutzt die Dienste einer darunterliegenden Schicht über einen *Service Access Point* (SAP). Ein SAP ist eine Schnittstelle zur Kommunikation einer Schicht mit einer darunterliegenden Schicht. Als *Diensterbringer* (Service Provider) bezeichnet man eine Schicht, die einen Dienst für eine darüberliegende Schicht bereitstellt.

Das ISO/OSI-Referenzmodell unterscheidet sogenannte Endsysteme und Transitsysteme (siehe hierzu Abb. 1.1). Endsysteme implementieren alle Schichten des Modells, während Transitsysteme nur die Schichten 1 bis 3 realisieren und als Verbindungssysteme dienen, die unterschiedliche Teilstrecken zwischen den Endsystemen abdecken. Die einzelnen Schichten haben im ISO/OSI-Referenzmodell folgende Aufgaben:

1. Die *Bitübertragungsschicht (Schicht 1)* ist die unterste Schicht und stellt eine physikalische Verbindung bereit. Hier werden die elektrischen und mechanischen Parameter festgelegt. Unter anderem wird in dieser Schicht spezifiziert, welcher elektrischen Größe ein Bit mit Wert 0 oder 1 entspricht. Hier werden nur Bits bzw. Bitgruppen ausgetauscht.

2. Die *Sicherungsschicht (Schicht 2)* sorgt dafür, dass ein Bitstrom einer logischen Nachrichteneinheit zugeordnet ist. Hier wird eine Fehlererkennung und -korrektur für eine Ende-zu-Ende-Beziehung zwischen zwei Endsystemen bzw. Transitsystemen unterstützt.
3. Die *Netzwerk- oder Vermittlungsschicht (Schicht 3)* hat die Aufgabe, Verbindungen zwischen zwei Knoten gegebenenfalls über mehrere Rechnerknoten hinweg zu ermöglichen. Auch die Suche nach einem günstigen Pfad zwischen zwei Endsystemen wird hier durchgeführt (Wegewahl, Routing).
4. Die *Transportschicht (Schicht 4)* sorgt für eine Ende-zu-Ende-Beziehung zwischen zwei Kommunikationsprozessen und stellt einen Transportdienst für die höheren Anwendungsschichten bereit.
5. Die *Sitzungsschicht (Schicht 5)* stellt eine Sitzung (Session) zwischen zwei Kommunikationsprozessen her und regelt den Dialogablauf der Kommunikation.
6. Die *Darstellungsschicht (Schicht 6)* ist im Wesentlichen für die Bereitstellung einer einheitlichen Transfersyntax zuständig. Dies ist deshalb wichtig, da nicht alle Rechnersysteme gleichartige Darstellungen für Daten verwenden. Manche nutzen z. B. den EBCDIC[1]- andere den ASCII[2]-Code und wieder andere den Unicode.[3] Auch die Byte-Anordnung bei der Integer-Darstellung (Little Endian und Big Endian Format) kann durchaus variieren. Unterschiedliche lokale Syntaxen werden in dieser Schicht in eine einheitliche, für alle Rechnersysteme verständliche Syntax, die auch als Transfersyntax bezeichnet wird, übertragen.
7. Die *Verarbeitungs-* oder *Anwendungsschicht (Schicht 7)* enthält schließlich Protokolle, die eine gewisse Anwendungsfunktionalität wie Filetransfer oder E-Mail bereitstellen. In dieser Schicht sind viele verschiedene Protokolle angesiedelt. Die eigentliche verteilte Anwendung zählt nicht zu dieser Schicht, sie nutzt aber das Anwendungsprotokoll.

Die Schichten 1 bis 4 werden auch gemeinsam als *Transportsystem* bezeichnet, die Schichten 5 bis 7 sind anwendungsorientierte Schichten. Die anwendungsorientierten Schichten nutzen die Transportdienste über die *Transportzugriffsschnittstelle*.

▶ **Tranportsystem und Transportzugriffsschnittstelle** Die Protokollschichten 1 bis 4 werden auch gemeinsam als Transportsystem bezeichnet. Das Transportsystem stellt den darüberliegenden Anwendungen einen Transportdienst zur Verfügung. Dieser ermöglicht es einer Anwendung bzw. dem darüberliegenden Anwendungsprotokoll, Nachrichten über ein Netzwerk zu senden. Die Schnittstelle zum Transportsystem wird als Transportzugriffsschnittstelle bezeichnet.

---

[1] EBCDIC (Extended Binary Coded Decimal Interchange Code) wird zur Kodierung von Zeichen verwendet und wird z. B. in Mainframes genutzt.

[2] ASCII (American Standard Code for Information Interchange) wird zur Kodierung von Zeichen verwendet und ist auch unter der Bezeichnung ANSI X3.4 bekannt.

[3] Unicode ist ein hardware- und plattformunabhängiger Code zur Zeichenkodierung und wird z. B. in der Sprache Java verwendet.

Für jede Schicht gibt es im OSI-Modell verschiedene Protokolle. In der Schicht 4 gibt es beispielsweise Transportprotokolle mit unterschiedlicher Zuverlässigkeit.

▶ **Ende-zu-Ende-Kommunikation** Die Schicht 2 dient dazu, eine Verbindung zwischen zwei Rechnern (Endsystemen oder Zwischenknoten) zu unterhalten, also eine Ende-zu-Ende-Verbindung zwischen zwei Rechnern herzustellen. Die Schicht 3 unterstützt Verbindungen im Netzwerk (mit Zwischenknoten), also Ende-zu-Ende-Verbindungen zwischen zwei Rechnern (Endsysteme) über ein Netz. Die Schicht 4 kümmert sich um eine Ende-zu-Ende-Kommunikation zwischen zwei Prozessen auf einem oder unterschiedlichen Rechnern.

Eine Anordnung von Protokollen verschiedener Schichten wird auch als *Protokollstack* (siehe Definition) bezeichnet. Auf verschiedenen Rechnersystemen muss die Anordnung der Protokolle gleich sein, sonst können die Systeme nicht miteinander kommunizieren.

▶ **Protokollstack** Eine konkrete Protokollkombination wird auch als Protokollstack (kurz Stack) bezeichnet. Der Begriff Stack (auch Kellerspeicher oder Stapelspeicher genannt) wird deshalb verwendet, weil Nachrichten innerhalb eines Endsystems von der höheren zur niedrigeren Schicht übergeben werden, wobei jedes Mal Steuerinformation ergänzt wird. Diese Steuerinformation wird im sendenden System von oben nach unten in jeder Protokollschicht angereichert und im empfangenden System beginnend bei Schicht 1 in umgekehrter Reihenfolge interpretiert und vor der Weiterreichung einer Nachricht an die nächst höhere Schicht entfernt.

Die Beziehungen zwischen zwei aufeinanderliegenden Protokollschichten sind in der Abb. 1.2 grafisch dargestellt. Dienstnehmer und Dienstgeber der Schicht i+1 kommunizieren

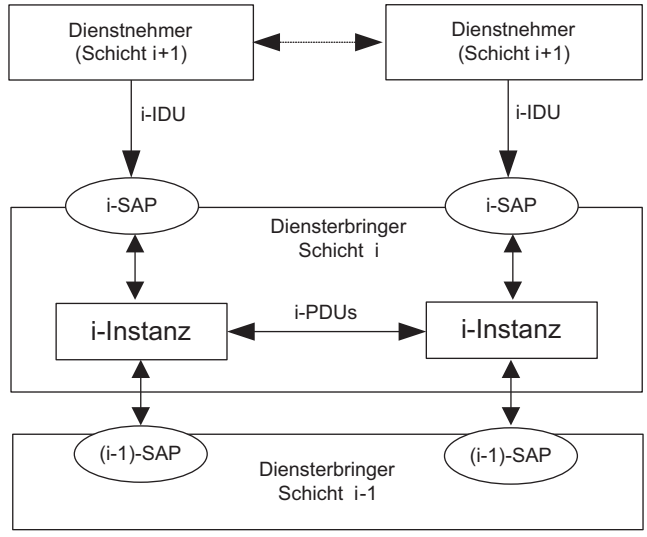


Abb. 1.2 Hierarchische Dienststruktur

miteinander, indem Sie die konkreten Implementierungen eines Dienstes der Schicht i nutzen. Die Dienste (siehe Definition) können über einen Dienstzuganspunkt bzw. *Service Access Point (SAP)* genutzt werden (siehe Definition). An einem SAP werden die Dienste einer Schicht bereitgestellt. Ein SAP ist dabei eine logische Schnittstelle, deren konkrete Realisierung etwa in einer Funktionsbibliothek oder in einem eigenen Prozess liegen könnte.

▶ **Dienst und Dienstelement** Ein Dienst ist eine Sammlung von Funktionen, die eine Schicht an einem SAP bereitstellt. Ein Dienst hat ggf. mehrere Dienstelemente (Primitiven). Beispielsweise verfügt der Dienst *connect* zum Aufbau einer Verbindung zwischen zwei Kommunikationspartnern über die Dienstelemente connect.request, connect.indication, connect.response und connect.confirmation.

Die Implementierung der Protokollfunktionen erfolgt in den entsprechenden Instanzen der jeweiligen Schicht. Die Schicht i nutzt ihrerseits wieder die Dienste der Schicht i−1 usw. Schicht i ist gegenüber der Schicht i+1 Diensterbringer.

Die Implementierungen bzw. *Protokollinstanzen* (siehe Definition) der gleichen Schicht tauschen Nachrichten, sogenannte Protocol Data Units (PDU) miteinander aus, die sowohl Steuerinformationen der jeweiligen Schicht als auch die Nutzdaten der nächsthöheren Schicht enthalten.

▶ **Protokollinstanz** Unter einer Protokollinstanz oder einer Instanz versteht man in der Datenkommunikation die Implementierung einer konkreten Schicht. Instanzen gleicher Schichten kommunizieren untereinander über ein gemeinsames Protokoll. Diese Kommunikation bezeichnet man im Gegensatz zur vertikalen Kommunikation aufeinander folgender Schichten auch als horizontale Kommunikation.

ISO/OSI-Protokolle
In den neunziger Jahren wurde eine Vielzahl von OSI-Protokollen in allen Schichten spezifiziert. In der Praxis haben sich nur wenige etabliert, wie z. B. die X.500-Protokolle für den Verzeichniszugriff in der Anwendungsschicht, das Routing-Protokoll IS-IS in der Vermittlungsschicht und in einigen Bereichen die Standards BER und ASN.1 für die Darstellungsschicht.[4]

1.3 TCP/IP-Referenzmodell

Das von der Internet-Gemeinde entwickelte TCP/IP-Referenzmodell (vgl. Abb. 1.3) ist heute der De-facto-Standard in der Rechnerkommunikation, natürlich vor allem im Internet. Es hat vier Schichten, wobei die Internet-Schicht (Netzwerkschicht) und die Transportschicht die tragenden Schichten sind. In der Netzwerkschicht wird neben einigen Steuerungsprotokollen im Wesentlichen das Protokoll IP (Internet Protocol) benutzt.

[4] Ein guter Überblick über OSI-Protokollstandards findet sich in Wikipedia: https://en.wikipedia.org/wiki/OSI_protocols. Zugegriffen am 07.07.2017.

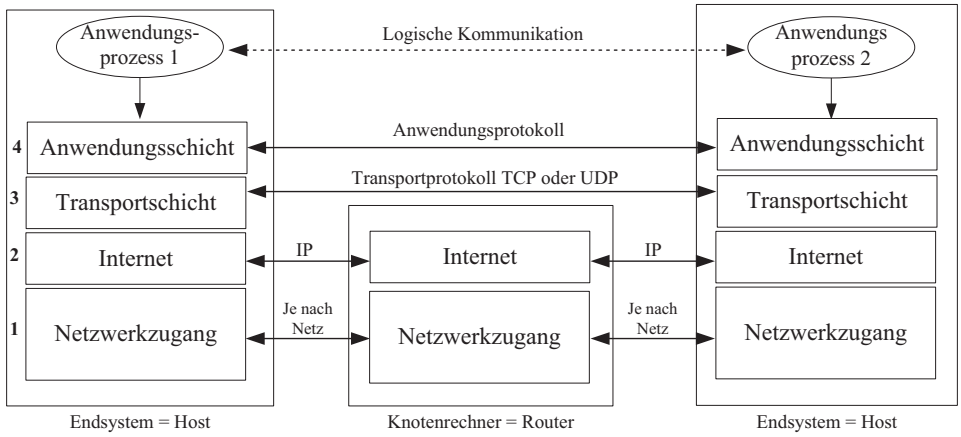


Abb. 1.3 TCP/IP-Referenzmodell

Steuerungsprotokolle der Internet-Protokollfamilie
Steuerungsprotokolle werden im Internet zur Unterstützung der Adressierung und für die Übertragung von Fehlermeldungen usw. benutzt. ARP (Address Resolution Protocol) ist beispielsweise ein Protokoll, das bei der Abbildung von IP-Adressen auf Netzwerkadressen unterstützt. ICMP (Internet Control Message Protocol) dient der Übertragung von Fehlermeldungen, beispielsweise wenn ein IP-Router einen Rechner nicht erreichen kann („Host unreachable“) oder auch zur Überprüfung, ob ein Rechner antwortet (ping).

Die ältere Version wird als IPv4 bezeichnet, die neue, bei weitem noch nicht durchgängig genutzte Version, hat die Bezeichnung IPv6. In der Transportschicht gibt es unter anderem zwei wichtige Standardprotokolle. Das mächtigere, verbindungsorientierte TCP (Transmission Control Protocol) und das leichtgewichtige, verbindungslose UDP (User Datagram Protocol). TCP gab dem Referenzmodell seinen Namen.

Im Gegensatz zum ISO/OSI-Referenzmodell wird im TCP/IP-Referenzmodell nicht so streng zwischen Protokollen und Diensten unterschieden. Die Schichten 5 und 6 sind im Gegensatz zum OSI-Modell leer. Diese Funktionalität ist der Anwendungsschicht vorbehalten, d. h. die Verwaltung eventuell erforderlicher Sessions und die Darstellung der Nachrichten in einem für alle Beteiligten verständlichen Format muss im Anwendungsprotokoll gelöst werden.

Kritik an den anwendungsnahen Schichten des ISO/OSI-Referenzmodells
Kritiker des ISO/OSI-Referenzmodells waren schon immer der Meinung, dass gerade die Funktionalität der Schichten 5 und 6 ohnehin sehr stark von der Anwendung abhängt und daher eine Aufteilung wenig sinnvoll ist. Diese Ansicht hat sich in der Internet-Praxis weitgehend durchgesetzt.

Alle Datenkommunikationsspezialisten sehen aber die Notwendigkeit für ein Transportsystem, wenn auch die Funktionalität je nach Protokolltyp hier sehr unterschiedlich sein kann. Beispielsweise gibt es verbindungsorientierte (wie TCP) und verbindungslose (wie UDP) Protokolle mit recht unterschiedlichen Anforderungen.

In der Netzwerkzugangsschicht (gemäß OSI-Modell sind das die Schichten 1 und 2) legt sich das TCP/IP-Referenzmodell nicht fest. Hier wird die Anbindung an das Netzwerk gelöst, und dies kann ein beliebiges Netzwerk sein. Ein Unterschied ergibt sich dabei bei der Adressierung von Partnern. Während im ISO/OSI-Referenzmodell auf statische Schicht-2-Adressierung gesetzt wird (Adressen müssen vorab a priori konfiguriert sein), nutzt man im TCP/IP-Referenzmodell eine Art dynamische Adressermittlung über das ARP-Protokoll. Hier wird zur Laufzeit eine Schicht-3-Adresse (IP-Adresse) auf eine Schicht-2-Adresse (auch MAC-Adresse, Medium Access Control) abgebildet. Dies trifft auf IPv4 zu. Bei IPv6 kann man auf ARP verzichten, da in diesem neuen Protokoll diese Funktionalität bereits enthalten ist. Die Grundidee der dynamischen Adressermittlung wird aber auch hier umgesetzt.

Wir konzentrieren uns im Weiteren auf die Transportschicht und gehen davon aus, dass die Schichten 1 bis 3 einen adäquaten Kommunikationsdienst zum Übertragen von Datenpaketen über beliebig große Netzwerke zur Verfügung stellen. Auch im TCP/IP-Referenzmodell werden der Netzwerkzugang, die Vermittlungsschicht und die Transportschicht gemeinsam als Transportsystem bezeichnet.

Das Anwendungsprotokoll bzw. der Entwickler des Anwendungsprotokolls sieht nur die Transportdienstschnittstelle und verwendet diese zur Implemenentierung der eigenen Kommunikationslogik. Die konkrete Implementierung des gesamten Transportsystems bleibt dem Entwickler eines Anwendungsprotokolls damit verborgen.

Wir werden uns in diesem Buch in erster Linie mit TCP und UDP beschäftigen. Anwendungsprotokolle für Instant Messaging (WhatsApp), IP-Telefonie (Skype), Filetransfer (FTP) und für die Webkommunikation (HTTP) nutzen die Dienste des TCP- und/oder des UDP-Transportsystems üblicherweise über eine API. In den meisten Betriebssystemen steht hierfür als Standard die Socket API zur Verfügung. Die Socket API wird klassisch in der Sprache C zur Verfügung gestellt, jedoch gibt es viele Sprachunterstützungen, unter anderem auch für die Programmiersprache Java.

In Abb. 1.4 wird die Nutzung des Transportdienstes aus Sicht der Anwendung gezeigt. Eine Anwendung nutzt in der Regel für die Kommunikation die Socket API. An der

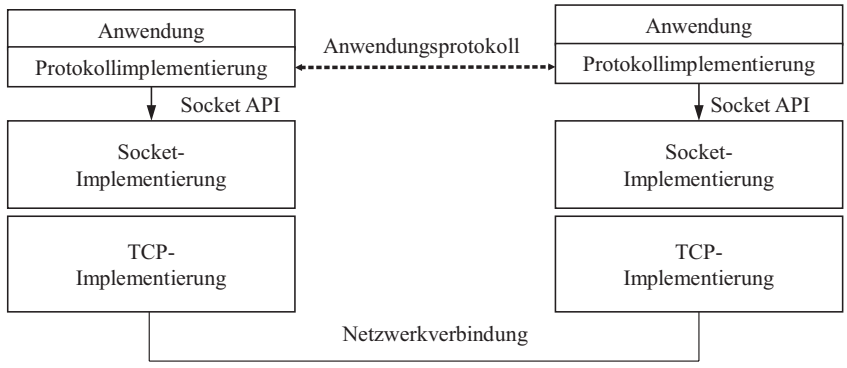


Abb. 1.4 Nutzung des Transportdienstes am Beispiel der Socket API

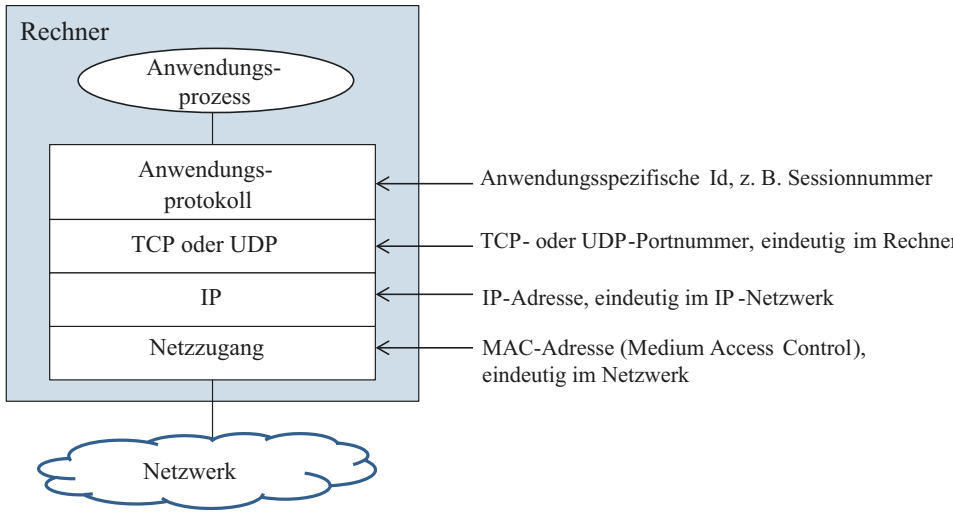


Abb. 1.5 Adressierungsinformationen der einzelnen Protokollschichten

Socket API werden sowohl verbindungsorientierte (TCP) als auch datagrammorientierte Dienste (UDP) bereitgestellt. Typische Dienstelemente sind send und receive. Verbindungsorientierte Dienste benötigen zusätzlich Dienstelemente für den Verbindungsauf- und -abbau (connect, disconnect).

Um Nachrichten zu senden muss eine Anwendung die Adresse der Partneranwendung kennen. In der TCP/UDP-Welt werden für jede Protokollschicht dedizierte Adressen vergeben (Abb. 1.5).

Schicht-2-Instanzen werden durch MAC-Adressen (Media Access Control), die eindeutig im Netzwerk sind, adressiert. Schicht-3-Instanzen erhalten eindeutige IP-Adressen (Internet Adresse), die für einen Rechner, genauer gesagt für seine Netzwerk-Schnittstelle, eindeutig sind. In der Schicht 4 werden TCP- oder UDP-Ports vergeben. Dies sind Nummern aus dem Wertebereich zwischen 0 und 65535, jeweils disjunkt für TCP und UDP. Die Anwendungsprotokolle nutzen individuelle Identifier wie z. B. eine Session-Identifikation.

In Abb. 1.6 wird ein Überblick über den gesamten TCP/IP-Protokollstack mit einigen wichtigen Protokollen gegeben. Das Bild soll zeigen, dass es viele verschiedene Protokolle gibt. FTP steht zum Beispiel für ein Filetransfer-Protokoll zum Übertragen von Dateien von einem Rechner zu einem anderen. SNMP steht für Simple Network Management Protokoll und dient dem Verwalten von Netzwerkkomponenten. Auf einzelne Anwendungsprotokolle soll hier nicht weiter eingegangen werden. Einige Protokolle werden aber in den Beispielanwendungen in diesem Kapitel noch erwähnt.

Ohne zu stark ins Detail zu gehen, sollen in diesem Absatz einige weit verbreitete Kommunikationsanwendungen gezeigt werden. Sie nutzen alle TCP und/oder UDP und verwenden für den Zugriff auf das Transportsystem auch die Socket API.

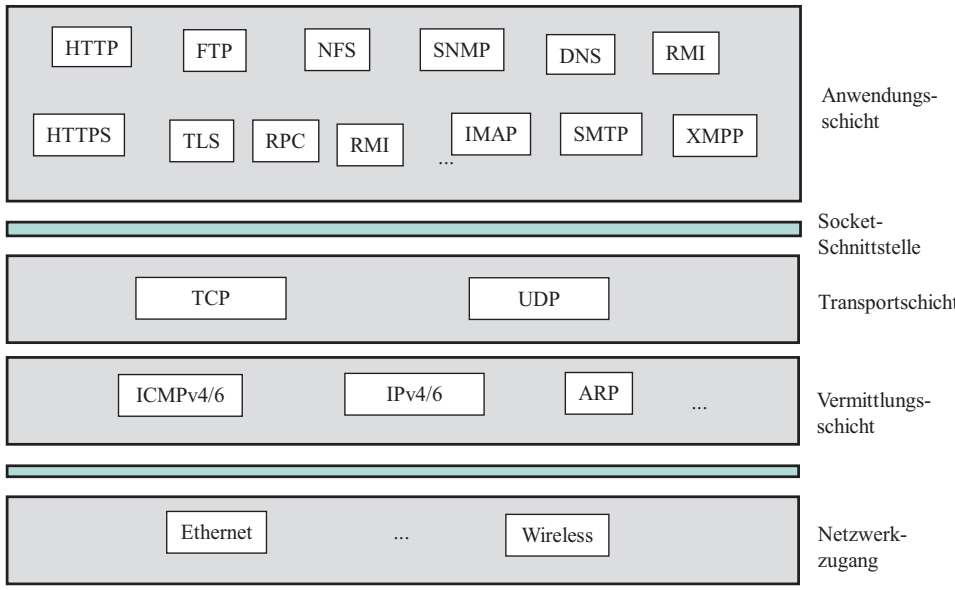


Abb. 1.6 Protokollbeispiele der TCP/IP-Welt

Vermittlungsschicht im Internet und Internet Protokoll (IP)
Im Internet bzw. in allen internetbasierten Netzwerken ist IP das Standardprotokoll für die Paketübermittlung. IP ist ein ungesichertes Vermittlungsprotokoll, das die Daten paketweise als Datagramme von der Quelle zum Ziel überträgt. Dabei wird ein Best-Effort-Ansatz gewählt. Datagramme können auch verloren gehen. Man spricht also von einem ungesicherten Schicht-3-Dienst, den IP bereitstellt. Die nächsthöhere Schicht, also die Transportschicht, muss sich, wenn nötig, um die Belange der Übertragungssicherheit kümmern.

Die derzeit noch am weitesten verbreitete IP-Version wird als IPv4 bezeichnet. Eine seit mittlerweile Jahrzehnten in Einführung befindliche neue Version hat die Bezeichnung IPv6. Beide Versionen unterscheiden sich erheblich in der Adressierung (4 Byte versus 16 Byte lange Adressen) und in vielen anderen Funktionen (Mandl et al. 2010).

1.4 Beispiele bekannter verteilter Anwendungen

1.4.1 World Wide Web

Webbasierte bzw. WWW-Anwendungen (World Wide Web) wie Onlineshop-Systeme und Suchmaschinen, aber auch viele Unternehmensanwendungen und vor allem Cloud-Anwendungen nutzen für die Kommunikation das Protokoll *HTTP* (Hypertext Transfer Protocol). Die Bestandteile der verteilten Anwendung sind ein Webbrowser auf der Nutzerseite und auf der Betreiberseite ein Webserver bzw. ein ganzer Cluster von Servern. Durch die Eingabe einer Adresse im Browser wird eine Betreiberseite adressiert. Dabei wird zunächst eine TCP-Verbindung aufgebaut, über die dann die Kommunikation mit Hilfe des HTTP-Protokolls abgewickelt wird.

HTTP/1.0, HTTP/1.1 und HTTP/2
HTTP 1.0 ist das Standard-Anwendungsprotokoll für die Kommunikation im Web. Es wurde 1991 im RFC 1945 standardisiert. In dieser Version musste für jede Anfrage zum Lesen eines Objektes bzw. einer Grafik innerhalb einer Webseite eine neue Verbindung aufgebaut werden. 1997 wurde das Protokoll in der Version HTTP/1.1 verbessert (RFC 2616). Unter anderem konnte nun auch eine TCP-Verbindung für mehrere Abfragen genutzt werden. Im Jahre 2015 (RFC 7540) wurde das Protokoll nochmals unter anderem um Push-Nachrichten, die vom Server initiiert werden können, erweitert.

HTTP ist also ein Anwendungsprotokoll, das auf Basis von TCP konzipiert wurde. Als Standardport wird der TCP-Port 80 verwendet. Die zu übertragenen Daten, in diesem Fall die Webseiten, werden in HTML (Hypertext Markup Language) beschrieben. HTML ist die Auszeichnungssprache, in der Dokumente (auch HTML-Seiten genannt) beschrieben werden.

Für die Kommunikation ist von Bedeutung, dass der ganze HTML-Code in den HTTP-PDUs in lesbarer Form vom Webserver zum Webclient übertragen wird. Weiterhin gibt es Möglichkeiten, in den PDUs auch Parameter aus den Eingabefeldern der HTML-Seiten an den Server zu senden.

Die Adressierung der HTML-Seiten auf den Webservern erfolgt über Uniform Resource Locators (URL) oder über sogenannte Uniform Resource Identifier (URI). URI ist ein allgemeinerer Begriff für alle Adressierungsmuster, die im WWW unterstützt werden.

Dieser Adressierungsmechanismus wurde für statische Webseiten erfunden, dient aber heute auch der Adressierung von Anwendungen, die dynamisch HTML-Content erzeugen und/oder komplexe Operationen im Server ausführen.

Die HTML-Dokumente liegen entweder statisch im Filesystem des Betreibers unter einem speziellen Verzeichnis oder werden – wie es heute üblicher ist – dynamisch z. B. über Datenbanken zusammengestellt. Man spricht hier auch von dynamisch erstellten Webseiten oder Content.

HTTP bildet als Kommunikationsprotokoll zwischen Webclient und Webserver das Rückgrat des WWW. Es ist ein *verbindungsorientiertes* und *zustandsloses* Request-/Response-Protokoll. Weder der Sender noch der Empfänger merken sich im Standardfall irgendwelche Status zur Kommunikation. Ein Request ist mit einem Response vollständig abgearbeitet. In den vergangenen Jahren wurden aber einige Erweiterungen entwickelt, die auch eine ereignisgesteuerte oder gleichberechtigte Kommunikation derart erlauben, dass der Server aktiv Nachrichten an Clients senden kann (Push). Dies wird über die sogenannten Websockets API (RFC 6455) realisiert. Auch muss man heute nicht mehr die ganze HTML-Seite anfordern, sondern kann auch kleine Aufrufe absetzen. Diese Erweiterung wird mit AJAX (Mandl et al. 2010) bezeichnet. AJAX steht für *Asynchronous JavaScript and XML* und bezeichnet eine nicht blockierende Zugriffsmethode auf Webserver.

In Abb. 1.7 ist der Protokollstack einer Webanwendung skizziert. Ein Web-Browser nutzt HTTP, das auf TCP aufsetzt. Dies bedeutet, dass für eine HTTP-Kommunikation ein TCP-Verbindungsaufbau durchgeführt werden muss.

HTTP nutzt sogenannte MIME-Bezeichner (Medientypen) für die Angabe des Inhalts der zu übertragenden Daten. Vom Web-Browser können also beliebige Dateitypen, die einen MIME-Typ besitzen, verarbeitet und über HTTP übertragen werden. Ein Web-Browser gibt im Request an, welche Formate er verarbeiten kann. Der Webserver

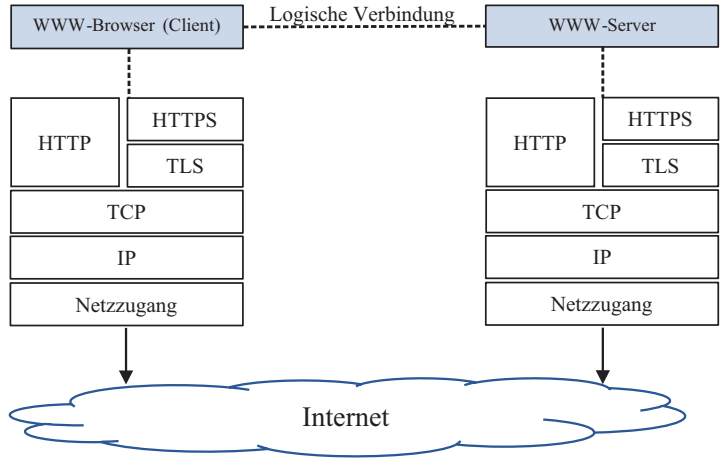


Abb. 1.7 Protokollstack für die webbasierte Kommunikation

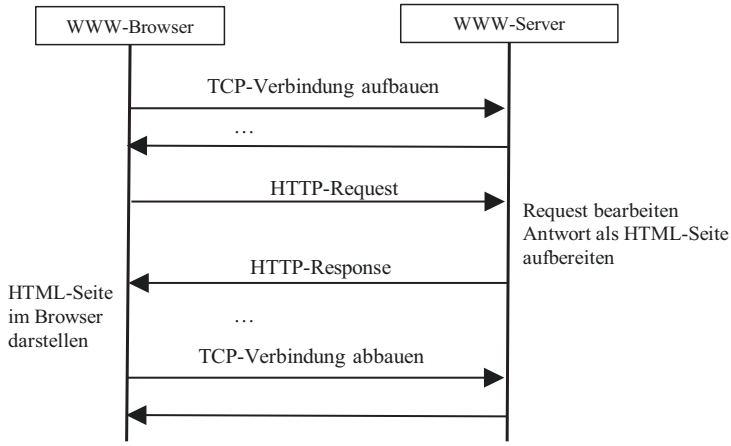


Abb. 1.8 Beispiel einer Kommunikation zwischen WWW-Browser und WWW-Server

teilt ihm in der Response mit, welches Format der gesendete Entity-Abschnitt nutzt (z. B. HTML, GIF, JPEG-Bilder, PDF).

Für alle Operationen werden im HTTP-Protokoll nur zwei PDU-Typen, die HTTP-Request- und die HTTP-Response-PDU, benötigt. Die PDUs sind sehr generisch aufgebaut und können daher mit vielen Spezialinformationen (hier Header genannt) versehen werden. Die wichtigsten *Protokolloperationen* sind GET und POST. Mit GET und POST können Dokumente vom Webserver angefragt werden. Eine entsprechende HTTP-Request-PDU mit einer inkludierten GET- oder POST-Operation wird vom Web-Browser zum Webserver gesendet. Das Ergebnis wird mit einer HTTP-Response-PDU vom Webserver zum Web-Browser kommuniziert.

Abb. 1.8 zeigt den grundsätzlichen Ablauf der Kommunikation. Man sieht, dass vor der HTTP-Kommunikation ein TCP-Verbindungsaufbau durchgeführt wird und nach dem

Empfang der Response-PDU die Verbindung (vom Client) wieder abgebaut wird. Wie der Verbindungsaufbau im Detail aussieht, werden wir noch ausführlich diskutieren.

Für sicherheitskritische Webzugriffe verwendet man heute eine Erweiterung von HTTP, das sogenannte HTTPS-Protokoll (HyperText Transfer Protocol Secure). Dabei erfolgt eine Verschlüsselung der Daten über TLS (Transport Layer Security). HTTPS-Verbindungen nutzen auch TCP als Transportprotokoll. Für HTTPS ist der TCP-Port 443 als Standardport reserviert. TLS ist ein Verschlüsselungsprotokoll, in dem auch ein Schlüsselaustausch erfolgt (Eckert 2014).

1.4.2 Electronic Mail

Ein weiterer, heute im Internet weit verbreiteter Dienst ist der E-Mail-Dienst zum Austausch von elektronischen Nachrichten (Electronic Mails). Auch hier handelt es sich um eine Client-/Serveranwendung. Ein Benutzer verwendet einen E-Mail-Client, *Mail User Agent* (MUA) genannt, um E-Mails zu senden und zu empfangen, und das Weiterreichen der E-Mails wird über E-Mail-Gateways abgewickelt. Jedem Benutzer, der E-Mails senden und empfangen möchte, wird eine elektronische Mailbox mit einer eindeutigen E-Mail-Adresse (wie beispielsweise mandl@cs.hm.edu) in einem E-Mail-Gateway zugeordnet. Die Mailboxen werden üblicherweise in den E-Mail-Gateways von Internet-Providern verwaltet. Die E-Mail-Gateways sind Serverrechner mit speziellen Softwarebausteinen für die Mail-Abwicklung. Sie haben in der Regel eine Doppelrolle. Zum einen nehmen sie Nachrichten von den MUAs entgegen. Diese Aufgabe wird als *Mail Submission Agent* (MSA) bezeichnet. Zum anderen leiten sie Nachrichten (E-Mails) für den Transport bis zum adressierten Empfänger in der Rolle von sogenannten *Mail Transfer Agents* (MTA) an andere E-Mail-Gateways weiter.

E-Mail-Gateways sind im Internet meist als SMTP-Server implementiert. Ein SMTP-Server übernimmt dabei die beiden Rollen von MSA und MTA. Die SMTP-Server kommunizieren untereinander über das Protokoll SMTP (Simple Mail Transfer Protocol,) über den Well-known TCP-Port =25.[5] Ein Benutzer sendet seine E-Mails von einem E-Mail-Client über SMTP an den zuständigen (konfigurierten) SMTP-Server. Der lesende Zugang eines Benutzers zu seiner Mailbox wird über sogenannte Mailzugangsprotokolle ermöglicht. Ein Internet-Benutzer kann sich z. B. bei einem Internet-Provider (z. B. T-Online) eine Mailbox einrichten und erhält zum Beispiel über das Protokoll POP3 (Post Office Protocol, Version 3) unter Nutzung des Well-known TCP-Ports 110 Zugang zu seiner Mailbox. POP3 ist im RFC 1939 definiert, SMTP im RFC 5321.

Technisch gesehen muss der Internet-Benutzer dann zunächst eine Verbindung zwischen seinem Mail-Client (das könnte *Microsoft Outlook* oder *Mozilla Thunderbird* sein) und dem zugeordneten SMTP-Server (bekannte Serverprogramme sind *sendmail* und *qmail*) bei seinem Internet-Provider herstellen. Dies erfolgt bei privaten Internet-Benutzern

[5] Es gibt noch einen zweiten Well-known TCP-Port mit der Nummer 587, der für dedizierte MSA-Implementierungen geschaffen wurde. Heute nutzt ein SMTP-Server meist beide Ports (25 und 587).

meist über eine Wählverbindung auf Basis von Telekom-Diensten wie DSL (Digital Subscriber Line).

Wenn die Verbindung über das POP3-Protokoll aufgebaut ist, kann die Mailbox ausgelesen werden, empfangene Mails können zum Clientrechner übertragen werden. Ein Mail-Client bietet heute meist eine komfortable Unterstützung zur Darstellung und zum Schreiben von Mails. Im POP3-Protokoll werden entsprechende Protokoll-Operationen unterstützt, um die Verbindung aufzubauen und die Nachrichten auszutauschen. Operationen sind zum Beispiel *LIST* zum Auflisten der vorhandenen Mails aus der Mailbox und *DELE* für das Löschen von Mails aus der Mailbox. Die Nachrichten werden in einem definierten Textformat übertragen.

Ein anderes, etwas komplexeres Zugangsprotokoll ist IMAP4 (RFC 3501, Internet Message Access Protocol) mit den Well-known TCP-Port 143. IMAP4 ist ebenfalls ein Internet-Standard. Im Gegensatz zum POP3-Protokoll verbleiben die E-Mails in der Regel auf dem Mailserver und werden nur bei Bedarf auf den Client-Rechner übertragen. IMAP4 unterstützt aber wesentlich mächtigere Verwaltungsfunktionen für die Mailboxen. IMAP4 wurde ursprünglich mit dem Ziel entwickelt, den Zugriff auf Mailboxen so bereitzustellen, als wenn diese sich auf dem lokalen Rechner befänden.

Der Nachrichtenaustausch von einem Senderrechner zu einem Empfängerrechner über zwei Mail-Gateways ist in Abb. 1.9 dargestellt. Die Mail-Clients des Senders und des Empfängers enthalten auch einen POP3- oder einen IMAP4-Client. Über diesen wird mit den entsprechenden IMAP-/POP-Servern auf den zugeordneten Mail-Gateways kommuniziert, um E-Mails abzuholen. Die beiden Mail-Clients unterhalten jeweils Mailboxen für die zugeordneten Mail-Benutzer.

POP3 wird heute meist von privaten Mailnutzern verwendet, während IMAP4 in Unternehmen eingesetzt wird. Meist wird dort dann auch ein eigenes Mail-Gateway unterhalten. POP3 und IMAP4 dienen nur zum Abholen der E-Mails von den zugeordneten Mailboxen in den SMTP-Servern. Wenn eine E-Mail vom Client abgesendet wird, erfolgt dies über das SMTP-Protokoll.

Prinzipiell funktioniert der Mail-Austausch nach dem sogenannten Store-and-Forward-Prinzip. Beim Mail-Gateway ankommende E-Mails werden zunächst in den Mailboxen

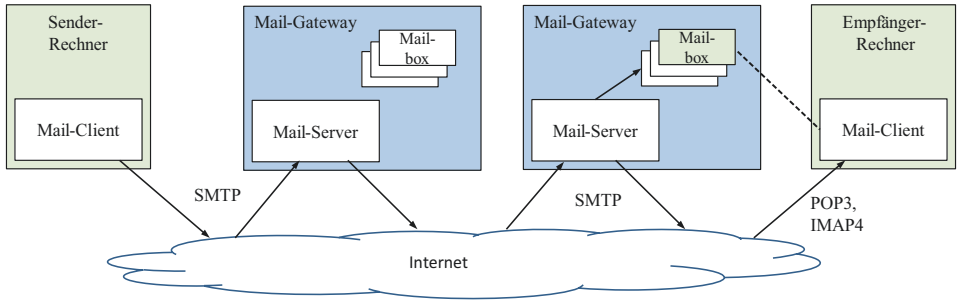


Abb. 1.9 Zusammenspiel der E-Mail-Komponenten

zwischengespeichert, um dann bei Bedarf, wenn die Verbindung zum adressierten Mail-Benutzer aufgebaut ist, an diesen weiterzuleiten.

Die elektronische Mailbox eines Benutzers wird mit einer eindeutigen E-Mail-Adresse (z. B. mandl@hm.edu) in einem E-Mail-Gateway adressiert. E-Mail-Clients können ihre Mails auch verschlüsseln (RFC 3207). Hierfür wird SMTPS eingesetzt, das auf TLS aufsetzt. Ebenso ist eine Verschlüsselung mit POP3S über den TCP-Port 995 und IMAP4S möglich. Allerdings müssen die SMTP-Server aktuell die Mail im Klartext bearbeiten, es wird also mit POP3S/IMAP4S und SMPTS keine Ende-zu-Ende-Sicherheit erreicht.

Ende-zu-Ende-Sicherheit
Unter Ende-zu-Ende-Sicherheit (end-to-end encryption) versteht man die sichere, verschlüsselte Übertragung von Daten über das gesamte Netzwerk von einem Kommunikationsendpunkt zum anderen. Auf der Senderseite erfolgt die Verschlüsselung und erst im Zielsystem, beim Empfänger, die Entschlüsselung. Zur Verschlüsselung benötigt man spezielle Protokolle wie beispielsweise TLS.

Abb. 1.10 zeigt den Protokollstack für die E-Mail-Kommunikation zwischen zwei E-Mail-Clients, die ihre Mailboxen über einen gemeinsamen E-Mail-Server verwalten. Das ist oft nicht so der Fall, aber für unsere Darstellung etwas einfacher. Wenn beide Partner ihre Mailboxen auf unterschiedlichen E-Mail-Servern verwalten, müssen die beiden E-Mail-Server zusätzlich über SMTP kommunizieren. In unserem Beispiel sendet der Client 1 eine E-Mail an Client 2, der sie dann aus seiner Mailbox ausliest. Die Kommandos DATA und FETCH sind nur angedeutet. Weitere Kommandos, die zum Beispiel für das Login notwendig sind, wurden nicht weiter dargestellt.

Das Mail-System im Internet wurde ursprünglich für Nachrichten im ASCII-Text konzipiert. Heute werden alle möglichen Objekte damit übertragen. Dies wird über einen Zusatz, den sogenannten *MIME-Standard* (Multipurpose Internet Mail Extensions), ermöglicht. Über MIME können prinzipiell Daten beliebigen Binärformats (wie Bilder, Sprache und Video) übertragen werden. Sender und Empfänger können sich über die

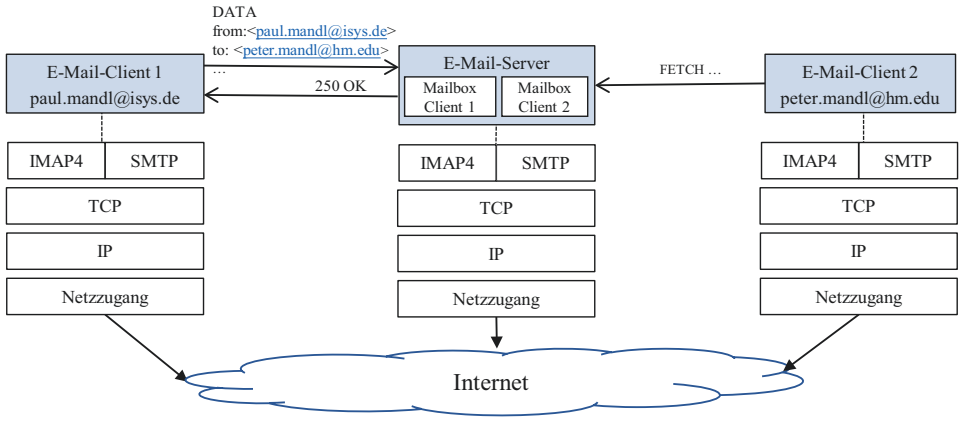


Abb. 1.10 E-Mail-Protokollstack

Kodierung der Daten verständigen, wobei ein Nachrichtenfeld namens *Content-Type* verwendet wird. Es sind mehrere Kodierungsmethoden spezifiziert, die die Übertragung im textbasierten E-Mail-System ermöglichen. Die binären Objekte werden beim Sender kodiert und beim Empfänger wieder dekodiert.

1.4.3 WhatsApp

Unter Instant Messaging (IM) versteht man den Austausch von textorientierten Nachrichten über einen gemeinsamen Kommunikationskanal. Bei Instant Multimedia Messaging (IMM) werden neben Textnachrichten auch multimediale Objekte (Bilder, Videos, Audio) kommuniziert. Heutige Instant-Messaging-Dienste übertragen zudem Dokumente und auch Standortinformationen. Ein aktuell sehr stark genutzter Instant-Messaging-Dienst ist *WhatsApp* vom gleichnamigen Unternehmen WhatsApp Inc.[6] Messenger oder Messenger-Anwendungen sind Programme, die den Zugang zum Instant-Messaging-Dienst ermöglichen. Der WhatsApp-Messenger ist auf gängigen Smartphones nutzbar. Es sind unterschiedliche Anwendungsszenarien vorgesehen:

- Zwei Personen kommunizieren direkt über ihre Messenger-Anwendungen miteinander;
- eine Gruppe kommuniziert zu einem bestimmten Topic miteinander (siehe z. B. WhatsApp-Gruppen).

Die Adressierung der Teilnehmer wird über Telefonnummern bzw. wie bei WhatsApp über eigene Identifikationsnummern ermöglicht.[7] Dies erfolgt zwar auf freiwilliger Basis, aber ohne Adressbuch funktioniert der Dienst nicht wirklich gut. Ankommende Nachrichten werden als Notifikation über den Push-Dienst des jeweiligen Smartphone-Betriebssystems an die Messenger-Anwendung übermittelt.

Der WhatsApp-Messenger kommuniziert mit verschiedenen WhatsApp-Serversystemen, die weltweit auf verschiedene Rechenzentren verteilt sind. Der konkrete Nachrichtenfluss ist nicht nicht bekannt, die Kommunikation erfolgt aber verbindungsorientiert über TCP. Die Verbindung erfolgt nicht direkt zwischen Teilnehmern, sondern zwischen Teilnehmer und Serversystemen. Ist einmal eine TCP-Verbindung zwischen dem WhatsApp-Messenger und den WhatsApp-Servern aufgebaut, bleibt diese bestehen bzw. wird bei einem Abbruch erneut aufgebaut.

Für den Nachrichtenaustausch wird auf der Anwendungsprotokollebene ein individuelles Messaging-Protokoll verwendet. Es soll dem Instant-Messaging-Standardprotokoll XMPP (Extensible Messaging and Presence Protocol) ähnlich sein.

---

[6] WhatsApp gehört heute zu Facebook. Im Jahre 2014 waren bei WhatsApp alleine in Deutschland etwas 30 Millionen Teilnehmer registriert.

[7] Leider wird bei WhatsApp auch das lokale Adressbuch des Smartphones an die WhatsApp-Systeme übertragen. Datenschutzrechtlich ist das sehr bedenklich.

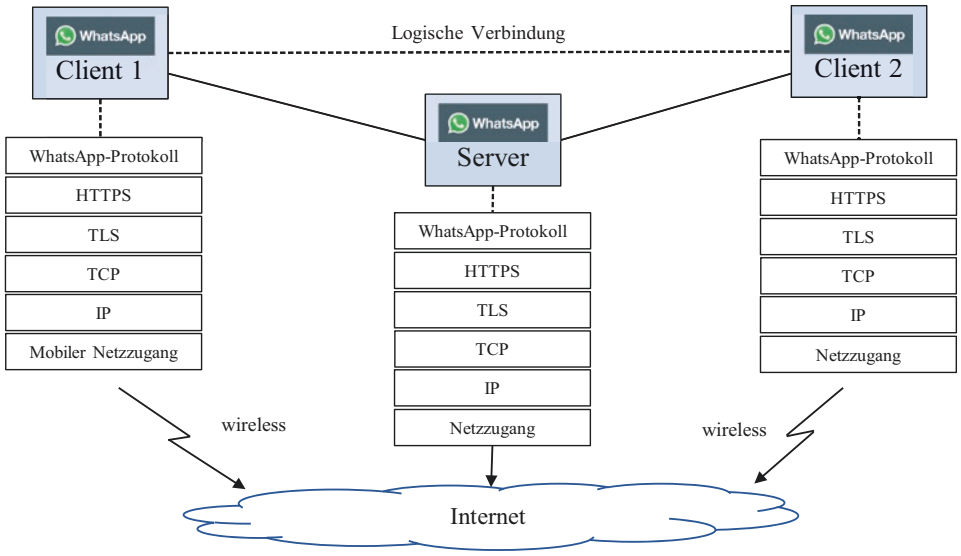


Abb. 1.11 WhatsApp-Protokollstack

Für Textnachrichten wird der TCP-Port 5222 verwendet. Multimediale Objekte werden aber über den aus der WWW-Kommunikation bekannten Port 80 übertragen. Der genaue Protokollstack ist nicht bekannt, aber er könnte so wie er in Abb. 1.11 dargestellt ist aussehen. Das proprietäre WhatsApp-Protokoll könnte über HTTPS übertragen werden. Zur Verschlüsselung der Daten kann zudem das Protokoll TLS (Transport Layer Security) eingesetzt werden. WhatsApp-Protokoll, HTTPS und TLS bilden gemeinsam aus Sicht des TCP/IP-Referenzmodells die Anwendungsebene, die auf der Transportschicht das verbindungsorientierte TCP nutzt.

Die Protokolle sollen im Einzelnen nicht weiter erläutert werden, aber der logische Nachrichtenfluss wird an einem Beispiel in Abb. 1.12 grob skizziert. Wie die Abbildung zeigt, erfolgt die logische Kommunikation zwischen zwei Endsystemen, die jeweils mit dem WhatsApp-Messenger-Client ausgestattet sind, nicht direkt, sondern über WhatsApp-Serversysteme. Die Übertragung und auch das Bereitstellen und Lesen der Nachrichten wird jeweils Ende-zu-Ende bestätigt. Wenn der Sender eine Nachricht im Messenger tippt, erfährt dies der adressierte Partner bereits während der Eingabe-Phase, indem implizit eine Nachricht gesendet wird. In der Messenger-Oberfläche erscheint dann der Hinweis „schreibt“. Ist die Nachricht erfolgreich versendet, wird in der Oberfläche ein Häkchen gesetzt. Ist die Nachricht auf dem Gerät des Empfängers angekommen, wird ein zweites Häkchen ausgegeben. Sobald der Empfänger die Nachricht erhalten, also gelesen hat, werden die beiden Häkchen anders eingefärbt.

Wie man sieht, ist das Protokoll aufgrund der Bestätigungen relativ aufwändig. Bei einer Gruppenkommunikation wird sogar versucht, die Häkchen erst dann zu setzen, wenn alle Gruppenmitglieder die Nachricht auf ihren Geräten empfangen bzw. ausgelesen haben.

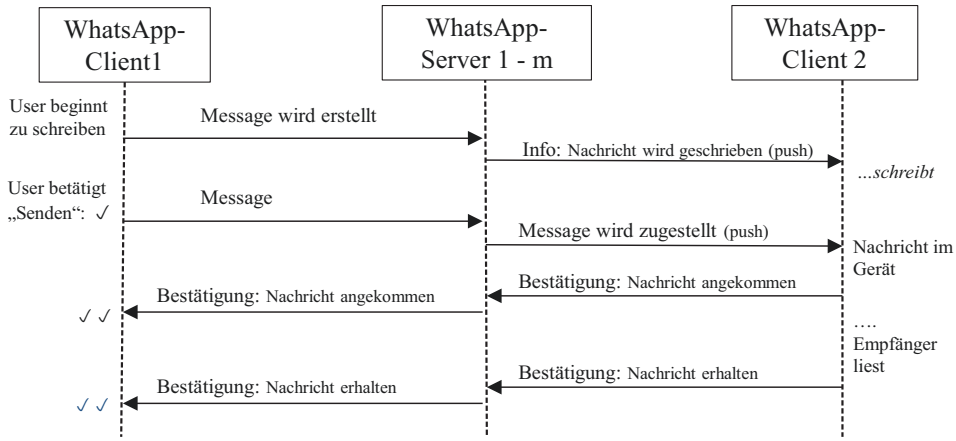


Abb. 1.12 WhatsApp-Nachrichtenfluss am Beispiel

► **Ende-zu-Ende-Bestätigung** Unter einer Ende-zu-Ende-Bestätigung (end-to-end acknowledgment) versteht man eine Quittierung der Ankunft einer Nachricht durch das Endsystem. Die kommunizierenden Endsysteme bestätigen sich also die Ankunft der Nachricht gegenseitig. Im Verständnis des Transportsystems bedeutet die Bestätigung, dass eine Nachricht im Empfangspuffer der Transportinstanz angekommen ist, aber nicht beim empfangenden Anwendungsprozess. Dieser muss die Nachricht erst von der Transportinstanz abholen.

1.4.4 Skype

Bei Skype handelt es sich um einen proprietären, internetbasierten Telefon-Dienst, also um eine verteilte Anwendung für die IP-Telefonie (Voice over IP) mit weiteren Funktionen für Konferenzschaltungen, Videokonferenzen, Instant-Messaging, Dateiübertragung und auch Screen-Sharing. Die Kommunikation der verteilten Komponenten erfolgt auf der Basis eines individuellen, nicht veröffentlichten Anwendungsprotokolls. Das Anwendungssystem wurde im Jahre 2003 von der Firma Skype Technologies entwickelt und gehört seit 2011 zu Microsoft.

Das ursprüngliche Skype-Anwendungsystem verfügte über folgende Komponenten:

- *Login-Server*: Dies ist die einzige zentrale Komponente und verwaltet Benutzernamen, Passwörter und Buddy-Lists. Für den Login-Server waren bis vor kurzem noch viele Replikate vorhanden, z. B. http1.sd.skype.net:80.
- *Skype Clients*: Dies sind Anwendungen, welche die Schnittstelle zum Benutzer darstellen. Sie stellen auch die *Peers* in dem ursprünglichen Peer-to-Peer-Netzwerk dar.

- *Super Nodes* oder *Super Peers*: Dies sind spezielle Skype-Clients mit Zusatzaufgaben für die Kommunikation. Jeder normale Skype Client musste eine Verbindung zu einem Super Node aufbauen. Jeder Skype Client konnte auch Super Node werden, das konnte man nicht verhindern.

Ursprünglich kontaktierte ein Skype Client beim Start ein Replikat des Login-Servers für den Anmeldevorgang. Jeder Skype Client verwaltete einen Host Cache mit den Adressen (IP-Adressen und Ports) von Super Nodes, damit ein Verbindungsaufbau zu einem Super Node in der Nähe möglich war. Super Nodes wurden für verschiedene Caching- und Kommunikationsaufgaben und auch als Vermittler eingesetzt. Falls keine Verbindung über einen Super Node möglich war, nutzte ein Skype Client fest eingebaute IP-Adressen von Bootstrap-Servern (hart kodiert in der ausführbaren Datei des Skype Clients). Wenn möglich, wurde die Kommunikation direkt zwischen den Peers ausgeführt, bei zu schlechter Verbindung wurden Super Nodes dazwischengeschaltet. Super Nodes waren stabilere Peers.

Von der Architektur her war Skype also eine hybride Peer-to-Peer-Anwendung. Bei einer reinen Peer-to-Peer-Anwendung gäbe es nur Peers, also Endsysteme, die in diesem Fall die Skype Clients darstellen. Nur diese würden miteinander kommunizieren. Bei hybriden Ansätzen sind auch ausgezeichnete Komponenten wie Super Nodes mit Spezialaufgaben etwa für das Registrieren erforderlich. In den vergangenen Jahren wurden die Super Nodes immer mehr auf dedizierte Server gelegt. Microsoft stellte das Skype-System mehr und mehr auf dedizierte Server um und platzierte diese in der hauseigenen Microsoft Azure Cloud. Damit wurde aus der Peer-to-Peer- eine Client-/Server-Lösung. Die Kommunikation läuft nun über Serversysteme und die Super-Node-Funktionalität liegt in den Servern der Cloud. Die Clients sind als Apps für Smartphones und Notepads oder als GUI-Anwendungen für Desktops oder als Webanwendung im Browser verfügbar.

Die Kommunikation zwischen Client und Server erfolgt über TCP und HTTPS/HTTP. Der Austausch von Audio- und Videodaten erfolgt über UDP.

1.5 Nachrichtenaufbau und Steuerinformation

Der Austausch von Nachrichten zwischen den Kommunikationspartnern wird durch die Anwendungen über das verwendete Anwendungsprotokoll initiiert. In der TCP/IP-Protokollfamilie unterscheiden wir vier Schichten. Demzufolge werden der Nutzdatennachricht einer Anwendung vier Header hinzugefügt. Header enthalten Kontroll- und Steuerinformationen. Beispielsweise enthält ein Header Adressinformationen, redundante Informationen zur Fehlererkennung, Zähler für die übertragenden Bytes und eine Bestätigungsinformation.

In jeder Schicht, welche die Nutzdatennachricht lokal auf den beteiligten Rechnern durchläuft, wird auf der Senderseite ein Header ergänzt und entsprechend auf der Empfängerseite wieder entfernt, bis schließlich bei der empfangenden Anwendung nur noch die

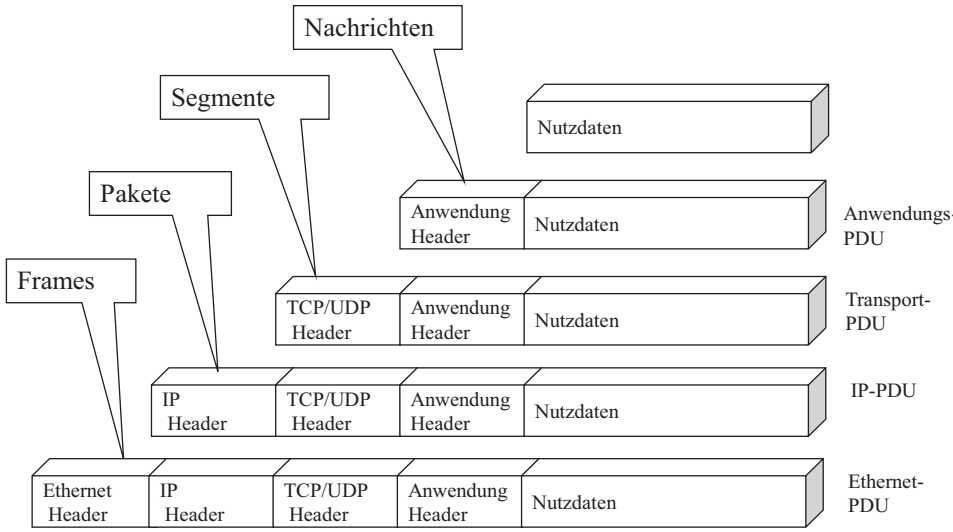


Abb. 1.13 Typischer Nachrichtenaufbau

Nutzdaten übrig bleiben, um die es ja eigentlich geht. In Abb. 1.13 sehen wir die Nutzdatennachricht, die zunächst durch das Anwendungsprotokoll um einen Anwendungs-Header ergänzt wird. Im Gesamten sprechen wir von der Anwendungs-PDU. In der TCP/IP-Welt wird hierfür auch der Begriff *Nachricht* verwendet. Ein typischer Header der Anwendungsschicht wäre zum Beispiel der Header eines Chat-Protokolls oder der Header für die Web-Kommunikation (meist HTTP). Entsprechend wird in der nächsten Schicht – je nach Transportprotokoll – ein TCP- oder UDP-Header ergänzt. Daraus ergibt sich die Transport-PDU, die in der TCP/IP-Welt auch als *Segment* bezeichnet wird.

Abb. 1.13 zeigt in der Schicht 3 (Vermittlungsschicht), dass ein Segment um einen IP-Header erweitert wird. Daraus entsteht die IP-PDU, die auch als *Paket* bezeichnet wird. Schließlich wird in der Netzzugangsschicht ein entsprechender Header ergänzt und damit ein *Frame* erzeugt. In unserem Beispiel wurde ein Ethernet-Header ergänzt. Die Ethernet-PDU wird schließlich über einen Netzwerkadapter physisch in das Netzwerk gesendet. Die gebräuchlichen Bezeichnungen für PDUs sind in der folgenden Definition zusammengefasst.

▶ **Bezeichnungen für PDUs** Die von den einzelnen Protokollschichten verwendeten Nachrichtentypen haben je nach Referenzmodell unterschiedliche Bezeichnungen. Während sich in der ISO/OSI-Welt der Begriff der PDU (Protocol Data Unit) etabliert hat, spricht man in der TCP/IP-Welt gerne von Nachrichten oder Paketen ohne Rücksicht auf die Schichtenzuordnung. Das ISO-/OSI-Modell ist hier exakter. Folgend sollen einige Synonyme erwähnt werden, die wir im Weiteren auch verwenden:

- Ein Frame ist in der ISO/OSI-Welt allgemein eine 2-PDU oder DL-PDU (DL = Data Link).
- Ein Paket ist in der ISO/OSI-Welt allgemein eine 3-PDU bzw. eine N-PDU (N = Network).

- Ein Segment wird in der ISO/OSI-Welt auch allgemein als 4-PDU oder T-PDU (T = Transport) bezeichnet.
- Nachrichten der Anwendungsschicht werden in der ISO/OSI-Welt allgemein als A-PDU (A = Application) bezeichnet.
- Setzt man konkrete Protokolle ein, so kann man beispielsweise auch Ethernet-PDU, TP4-PDU oder TCP-PDU usw. verwenden.
- Ganz spezielle PDUs eines Protokolls kann man auch exakter bezeichnen. Beispielsweise kann eine Bestätigungs-PDU als ACK-PDU, oder eine Verbindungsaufbau-PDU als Connect-PDU bezeichnet werden. Hier ergibt sich die Schichtenzugehörigkeit aus dem Kontext.

Die ISO/OSI-Welt kennt auch noch eine 1-PDU oder PH-PDU (PH = physical) für die PDUs, die in der Schicht an das Medium übertragen werden. In der TCP/IP-Welt gibt es hier keine Entsprechung.

Die Schichtenanordnung des Protokollstacks kann sich auf dem Weg von der Quelle zum Ziel erweitern, wenn die Nachricht zum Beispiel mehrere Rechnerknoten, in der TCP/IP-Welt als Router bezeichnet, überquert. Insbesondere die Netzzugangsschichten können hier variieren, jedoch ist es wichtig, das zwei direkt benachbarte Rechnerknoten jeweils den gleichen Protokollstack kennen. Dies trifft aber nur auf die Schichten zu, die auch für die Bearbeitung benötigt werden. Ein klassischer IP-Router in einem IP-Netzwerk betrachtet in der Regel die Protokoll-Header überhalb der Schicht 3 nicht.

Ethernet
Ethernet ist eine Netzwerktechnologie für lokale Netzwerke (LANs = Local Aread Networks), welche die Funktionalität der unteren beiden Schichten gemäß ISO/OSI-Referenzmodell abdeckt. Ethernet wurde Anfang der 70er-Jahre von R. Metcalfe[8] bei der Firma Xerox entwickelt und später von den Firmen Xerox, DEC und Intel zum IEEE 802.3-Standard ausgebaut. Ethernet war von der Topologie her ein Bussystem. Die angeschlossenen Rechner nutzten also ein gemeinsames Medium. In heutigen LANs wurde aber das Bussystem weitgehend durch sternförmige Vernetzung über Switches ersetzt.

Beispielsweise ergänzt ein DSL-Router in einem Netzwerk bei einem ausgehenden Verbindungsaufbauwunsch noch Protokollheader für die Protokolle PPP (Point-to-Point-Protocol) und PPPoE (PPP over Ethernet) im Ethernet-Frame. Rechnerknoten von Telekommunikationsanbietern nutzen zum Transport im Telekom-Netzwerk oft auch das Übertragungsverfahren ATM (Asynchronous Transfer Mode) mit seinen speziellen Protokollen. Beim adressierten Endsystem kommt ein Ethernet-Frame mit den eingebetteten Headern und den Nutzdaten an, wenn der Zielrechner auch in einem Ethernet-Netzwerk liegt. Eventuell während des Transports ergänzte „Interims-Header“ anderer Protokolle sind dann also schon wieder entfernt.

[8] R. Metcalfe ist der Gründer von 3COM, hat die Firma aber verkauft.

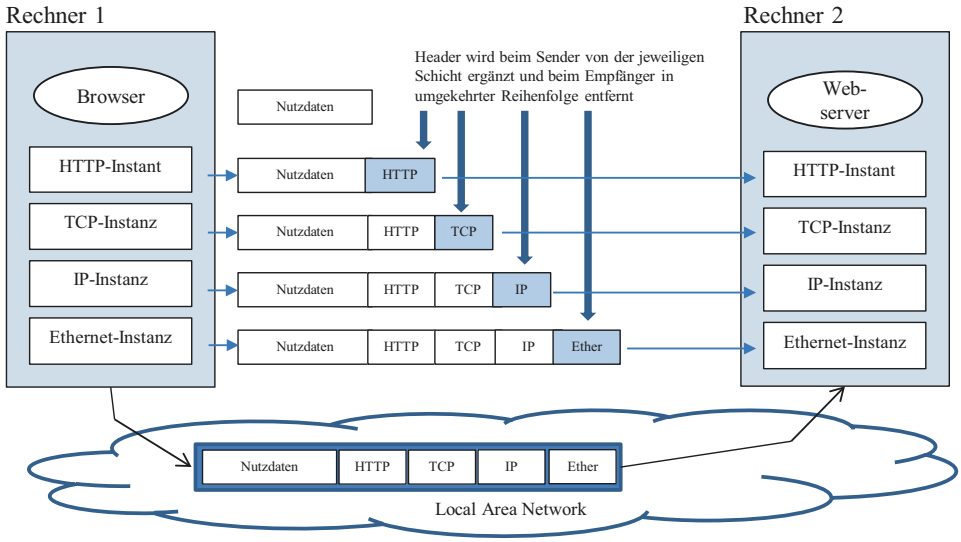


Abb. 1.14 Nachrichtenaufbau über die Schichten

Abb. 1.14 zeigt nochmals am Beispiel einer HTTP-Nachricht, wie beim Sender die Gesamtnachricht in den einzelnen Schichten aufgebaut und beim Empfänger wieder abgebaut wird. Der Browser sendet Nutzdaten an den Webserver. Im sendenden Rechner durchläuft die Nachricht die einzelnen Schichteninstanzen und wird jeweils um einen Header mit Steuerinformationen erweitert. Dann erst wird die gesamte Nachricht physisch an den empfangenden Rechner gesendet, der sie an seinem Netzwerkadapter entgegennimmt und zunächst – in unserem Fall – der Ethernet-Instanz zur Bearbeitung übergibt. Diese schickt die PDU an die IP-Instanz weiter und dies wird so lange fortgesetzt, bis sie beim Webserver angekommen ist.

Literatur

Eckert, C. (2014) IT-Sicherheit Konzepte – Verfahren – Protokolle, Oldenbourg Wissenschaftsverlag, 2014

Mandl, P.; Bakomenko A.; Weiß, J. (2010) Grundkurs Datenkommunikation: TCP/IP-basierte Kommunikation: Grundlagen, Konzepte und Standards, 2. Auflage, Vieweg-Teubner Verlag, 2010

Grundkonzepte der Transportschicht

2

Zusammenfassung

Transportprotokolle wie TCP und UDP sind die Basis für die höherwertigen Kommunikationsmechanismen der Anwendungsschicht. Das Transportsystem als Ganzes stellt Übertragungsfunktionen bereit und wickelt sehr viele Aufgaben im Hintergrund ab. Für den Nutzer eines Transportsystems, den Dienstnehmer, ist nicht ersichtlich, welche Nachrichten konkret gesendet werden und mit welchen Protokollmechanismen die Kommunikation durchgeführt wird. Wichtige Funktionen sind die Adressierung und die Datenübertragung. Je nach Funktionalität kann die Transportschicht verbindungsorientiert oder verbindungslos arbeiten. Bei verbindungsorientierten Protokollen werden zudem noch Mechanismen zum Verbindungsmanagement und während der Datenübertragung auch Bestätigungsverfahren, Übertragungswiederholungsverfahren, Flusskontroll- und der Staukontrollmechanismen sowie Mechanismen zu Stückelung (Segmentierung) von Nachrichten benötigt. Die grundlegenden Mechanismen der Transportschicht werden zunächst allgemein, ohne konkreten Bezug zu einem Transportprotokoll betrachtet.

2.1 Grundlegende Aspekte

2.1.1 Typische Protokollmechanismen der Transportschicht

In der Protokollentwicklung werden in verschiedenen Schichten vergleichbare Protokollfunktionen (Protokollmechanismen) eingesetzt. Viele dieser Mechanismen werden in der Transportschicht genutzt und daher sollen in diesem Absatz vorab einige wichtige Protokollfunktionen allgemein erläutert werden.

P. Mandl, *TCP und UDP Internals*, https://doi.org/10.1007/978-3-658-20149-4_2

Unter *Basisprotokollmechanismen* verstehen wir die klassischen Funktionen für die *Verbindungsverwaltung* und für den *Datentransfer*. Der klassische Datentransfer dient der Übertragung der Transport-PDUs. Manche Protokolle unterstützen auch einen *Vorrangtransfer*. Sogenannte *Vorrang-PDUs* können früher gesendete „normale" Transport-PDUs überholen.

Zu den *Protokollmechanismen für die Fehlerbehandlung* werden verschiedene Funktionen gezählt. Die Auslieferung von PDUs in der richtigen Reihenfolge und gegebenenfalls die lückenlose Zustellung sind hier anzusiedeln. Die *Quittierung* ist auch in diesen Funktionsbereich einzuordnen. Empfangene Nachrichten werden vom Empfänger quittiert, indem zum Beispiel eine eigene Bestätigungs-PDU (ACK-PDU) gesendet wird. Es gibt dafür verschiedene Verfahren. Beispielsweise kann man einzeln oder gruppenweise (kumulativ) quittieren, oder man kann auch nur eine negative Quittierung (NACK = Negative ACKnowledgement) senden, wenn ein Paket fehlt (negative acknowledgement). Auch Mischungen der einzelnen Verfahren sind möglich. Eine Bestätigung im sogenannten Huckepack-Verfahren (picky back acknowledgement) ist ebenso in manchen Protokollen vorzufinden. Dies bedeutet, dass eine Bestätigung für eine Nachricht mit der Antwort mitgesendet wird. Zur Fehlerbehandlung gehören weiterhin *Zeitüberwachungsmechanismen* über Timer, *Prüfsummen* und *Übertragungswiederholung* sowie die *Fehlererkennung* und *Fehlerkorrektur* (forward error correction). Oft wird eine Timerüberwachung für jede gesendete Nachricht verwendet. Ein Timer wird „aufgezogen", um das Warten auf eine ACK-PDU zu begrenzen. Hier ist die Zeitspanne von großer Bedeutung. Sie kann statisch festgelegt oder bei komplexeren Protokollen dynamisch anhand des aktuellen Laufzeitverhaltens eingestellt werden. Übertragungswiederholung wird durchgeführt, wenn eine Nachricht beim Empfänger nicht angekommen ist. Der Sender nimmt dies beispielsweise an, wenn eine ACK-PDU nicht vor dem Timeout ankommt. Durch Redundanz in den Nachrichten können Fehler festgestellt und bei ausreichender Redundanz auch gleich im Empfänger korrigiert werden. In der Regel verwenden heutige Protokolle nur Fehlererkennungsmechanismen und fordern eine PDU bei Erkennen eines Fehlers erneut an.

Protokollmechanismen zur Längenanpassung sind notwendig, wenn die unterliegende Protokollschicht die Transport-PDU nicht auf einmal transportieren kann und daher eine Zerlegung dieser erfordert. Diesen Mechanismus bezeichnet man als Assemblierung und Deassemblierung. Beim Sender wird die PDU in mehrere Segmente aufgeteilt. Beim Empfänger muss die PDU dann wieder zusammengebaut, also reassembliert werden. Eine ähnliche Funktion ist in der Vermittlungsschicht zu finden. Hier spricht man von Fragmentierung und Defragmentierung.

Protokollmechanismen zur Systemleistungsanpassung dienen der Flusssteuerung, und der *Überlast-* und der *Ratensteuerung*. Die Flusssteuerung schützt den Empfänger vor Überlastung durch den Sender. Hierfür werden z. B. Fenstermechanismen (sliding window) eingesetzt, die einem Sender einen gewissen dynamisch veränderbaren Sendekredit einräumen. Der Sender darf dann so viele Transport-PDUs oder Bytes ohne eine Bestätigung senden, wie sein Sendekredit vorgibt. Der Empfänger bremst den Sender aus, wenn sein

Empfangspuffer zu voll wird und er mit der Verarbeitung, also der Weiterleitung an die nächsthöhere Schicht, nicht nachkommt. Die Überlastsicherung (congestion control) schützt das Netzwerk vor einer Überlastung. Der Sender wird über die Überlast informiert und drosselt daraufhin das Sendeaufkommen, bis die Überlast beseitigt ist. Dies ist meist ein sehr dynamischer Vorgang.

Protokollmechanismen zur Übertragungsleistungsanpassung dienen schließlich zum *Multiplexieren* und *Demultiplexieren* (multiplexing, demultiplexing). Mehrere Verbindungen einer Schicht n können auf eine (n-1)-Verbindung abgebildet werden. Hier spricht man von Multiplexen. Auf der Empfängerseite wird es umgedreht. Der Vorgang wird als Demultiplexen bezeichnet. Im Gegensatz dazu ist auch der umgekehrte Weg möglich. Man spricht hier auch von Teilung und Vereinigung. Eine n-Verbindung wird auf mehrere (n-1)-Verbindungen verteilt und beim Empfänger wieder vereint. Dies kann vorkommen, wenn eine höhere Schicht über eine größere Übertragungsleistung verfügt als eine darunterliegende.

Unter *nutzerbezogenen Mechanismen* versteht man die *Dienstgütebehandlung* (quality of service) wie die Ratensteuerung (rate control). Diese Mechanismen werden für vor allem für Multimedia-Anwendungen benötigt. Dienstgüteparameter sind u. a. der Durchsatz, die Verzögerung und die Verzögerungssensibilität. Diese Parameter können je nach Protokoll beispielsweise beim Verbindungsaufbau zwischen Sender und Empfänger ausgehandelt werden. Unter Ratensteuerung versteht man z. B. das Aushandeln einer zulässigen Übertragungsrate zwischen Sender und Empfänger.

Welche Mechanismen in einer Transportinstanz für ein Transportprotokoll konkret implementiert sind, hängt sehr stark von der Funktionalität der darunterliegenden Vermittlungsschicht ab. Ist diese beispielsweise schon sehr zuverlässig, kann die Implementierung der Transportschicht einfacher sein.

2.1.2 Verbindungsorientierte und verbindungslose Transportdienste

Die Transportschicht stellt den anwendungsorientierten Schichten einen Transportdienst für eine Ende-zu-Ende-Verbindung zur Verfügung. Höhere Schichten nutzen die Dienste zum Austausch von Nachrichten über einen Dienstzugang, der allgemein als Transport Service Access Point (T-SAP) bezeichnet wird. Die Implementierung einer konkreten Transportschicht auf einem Rechnersystem wird als Transportinstanz (T-Instanz) bezeichnet. T-Instanzen tauschen zur Kommunikation Transport-PDUs (T-PDUs), die aus Steuerinformationen (Header) und Nutzdaten bestehen.

Je nach Ausprägung der Transportschicht ist der Dienst *zuverlässig* oder *unzuverlässig* im Sinne der Datenübertragung. Man unterscheidet weiterhin grundsätzlich zwischen *verbindungsorientierten* und *verbindungslosen* Transportdiensten. Bei Ersteren wird zwischen den Kommunikationspartnern vor dem eigentlichen Datenaustausch eine Transportverbindung etabliert, bei Letzteren ist dies nicht notwendig.

Verbindungsorientierte Transportdienste sind durch drei Phasen gekennzeichnet:

- Verbindungsaufbau
- Datenübertragung
- Verbindungsabbau

Der Sender adressiert den Empfänger beim Verbindungsaufbau über eine *Transportadresse*. Für eine Verbindung muss bei beiden Kommunikationspartnern in den Transportinstanzen ein gemeinsamer *Verbindungskontext* verwaltet werden. Der Verbindungskontext wird in den T-Instanzen verwaltet und enthält alle Informationen zum aktuellen Stand der Kommunikation, u. a. den Zustand der Verbindung oder einen Verweis auf die als nächstes zu erwartende Nachricht.

Verbindungsorientierte Transportprotokolle sichern zu, dass keine Daten während der Verbindung verloren gehen und auch die Reihenfolge der versendeten Segmente auch der Reihenfolge beim Empfang entspricht.

Verbindungslose Dienste sind dagegen meist durch folgende Eigenschaften charakterisiert:

- Der Verlust von Datenpaketen ist möglich.
- Die Daten können gegebenenfalls verfälscht werden.
- Die Reihenfolge ist nicht garantiert.

Verbindungsorientierte Protokolle sind aufgrund der erweiterten Funktionalität komplexer und daher aufwändiger zu implementieren, bieten aber in der Regel eine höhere Zuverlässigkeit und garantieren meist eine fehlerfreie und reihenfolgerichtige Auslieferung der Daten beim Empfänger.

Nicht jede Anwendung benötigt einen verbindungsorientierten Transportdienst und entsprechende Zuverlässigkeit. Dies ist z. B. für neuere Anwendungstypen wie etwa Videoübertragungen (Streaming) der Fall. Hier wird oft auf verbindungslose Dienste zurückgegriffen, da ein begrenzter Datenverlust vertretbarer ist als Verzögerungen in der Nachrichtenübertragung oder als Verzögerungsschwankungen, auch als Jitter bezeichnet.

2.2 Verbindungsmanagement und Adressierung

In Abb. 2.1 ist die Kommunikation zwischen zwei Anwendungsprozessen in zwei verschiedenen Rechnersystemen (Hosts) dargestellt. Die Anwendungsprozesse sind über die zugehörigen T-SAPs adressierbar, während die Transportschicht N-SAPs zur Adressierung der Rechnersysteme nutzt. Die Schicht-4-Adresse wird auch als Transportadresse bezeichnet. In der Abbildung ist zu erkennen, dass ein Anwendungsprozess neben dem N-SAP noch ein weiteres Identifikationsmerkmal aufweisen muss, da ja mehr als ein Anwendungsprozess eines Hosts über denselben N-SAP kommunizieren kann. Bei TCP und bei UDP sind dies beispielsweise die sogenannten Portnummern.

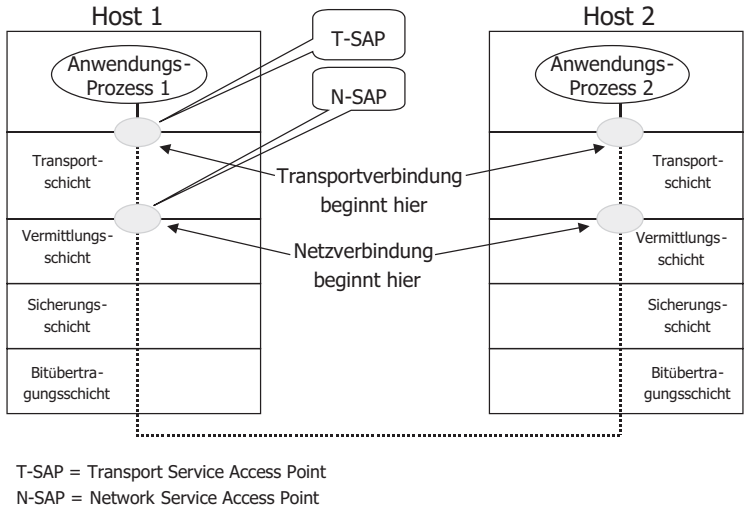


Abb. 2.1 Adressierung in der Transportschicht (nach Tanenbaum und Wetherall 2011)

Eine T-Instanz unterstützt in der Regel mehrere T-SAPs. Transportadressen sind meist sehr kryptisch, weshalb oft symbolische Adressen benutzt werden, die über sogenannte Naming-Services (Directory Services) auf Transportadressen abgebildet werden, ohne dass der Anwendungsprogrammierer diese kennen muss.

2.2.1 Verbindungsaufbau

Verbindungsorientierte Dienste erfordern einen Verbindungsaufbau, in dem zunächst auf beiden Seiten ein Verbindungskontext aufgebaut wird. Die Endpunkte der Verbindung werden im ISO/OSI-Jargon auch als *Connection End Points (CEP)* bezeichnet. Beim Verbindungsaufbau erfolgt eine Synchronisation zwischen den Partnern und die Einrichtung des Verbindungskontexts auf beiden Seiten. Hierfür werden Synchronisationsnachrichten ausgetauscht.

Ein Verbindungsaufbau erfolgt meist über einen bestätigten Dienst, in dem Folgenummern vereinbart werden. Ein klassisches Verbindungsaufbauprotokoll ist das Dreiwege-Handshake-Protokoll, das in Abb. 2.2 skizziert ist.

Der Ablauf sieht wie folgt aus:

- Host A (bzw. die T-Instanz in Host A) initiiert den Verbindungsaufbau mit einer Connect-Request-PDU und sendet dabei eine vorher ermittelte Folgenummer (seq=x) mit. Mit der Connect-Request-PDU wird auch die Transportadresse gesendet, um den Empfänger zu identifizieren.

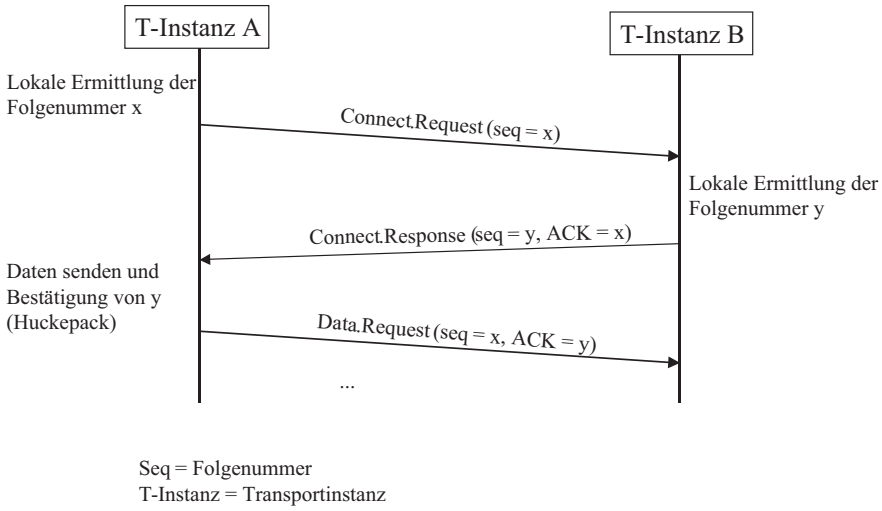


Abb. 2.2 Dreiwege-Handshake-Protokoll für den normalen Verbindungsaufbau

- Host B (bzw. die T-Instanz in Host B) bestätigt den Verbindungsaufbauwunsch mit einer ACK-PDU, bestätigt dabei die Folgenummer und sendet seine eigene Folgenummer (seq=y) mit der ACK-PDU an Host A.
- Host A bestätigt die Folgenummer von Host B mit der ersten Data-PDU im Huckepack-Verfahren, und damit ist die Verbindung aufgebaut.

Beim Verbindungsaufbau muss sichergestellt werden, dass keine Duplikat-PDUs aus alten Verbindungen erneut beim Empfänger ankommen. Hierzu sind entsprechende Protokollmechanismen, wie etwa die Verwaltung von Folgenummern (andere Bezeichnung: Sequenznummern), erforderlich. Folgenummern sind fortlaufende Zähler der abgesendeten Nachrichten oder Bytes. Kombiniert man diese mit einer maximalen Paketlebensdauer, so kann man einen gewissen Schutz vor einer falschen Zuordnung zu einer Verbindung gewährleisten.

Wie in Abb. 2.3 dargestellt, können beim Verbindungsaufbau gewisse Fehlersituationen auftreten. Beispielsweise kann eine alte Connect-Request-PDU bei Host B ankommen, der dann nicht feststellen kann, ob dies eine gültige PDU ist. Host A muss erkennen, dass es sich um einen Connect-Response für einen Connect-Request handelt, der nicht bereits in der Vergangenheit von ihm initiiert wurde. Er muss die ACK-PDU dann negativ beantworten und die Verbindung zurückweisen (Reject).

Eine noch kompliziertere Fehlersituation kann auftreten, wenn zusätzlich zum Connect-Request-Duplikat noch ein altes ACK-Duplikat auftritt. Auch hier darf die Verbindung nicht zustande kommen (vgl. Abb. 2.4).

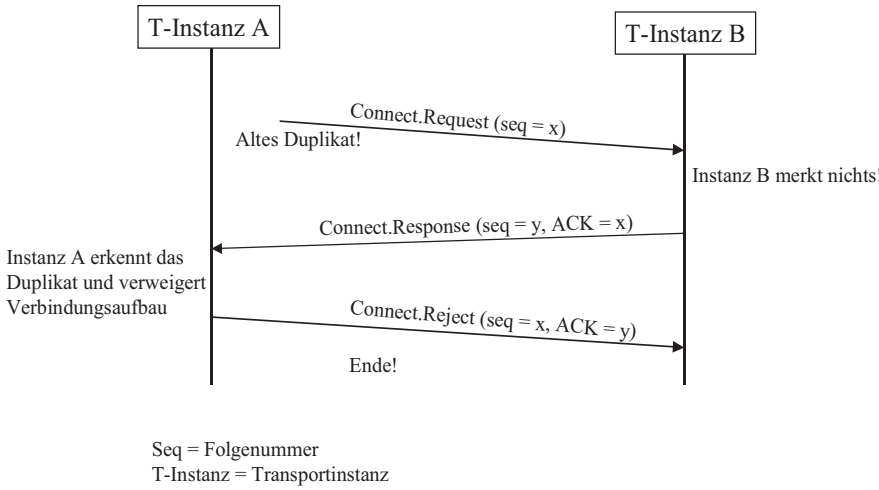


Abb. 2.3 Verbindungsaufbau, Fehlerfall 1: Connect-Request-Duplikat taucht auf

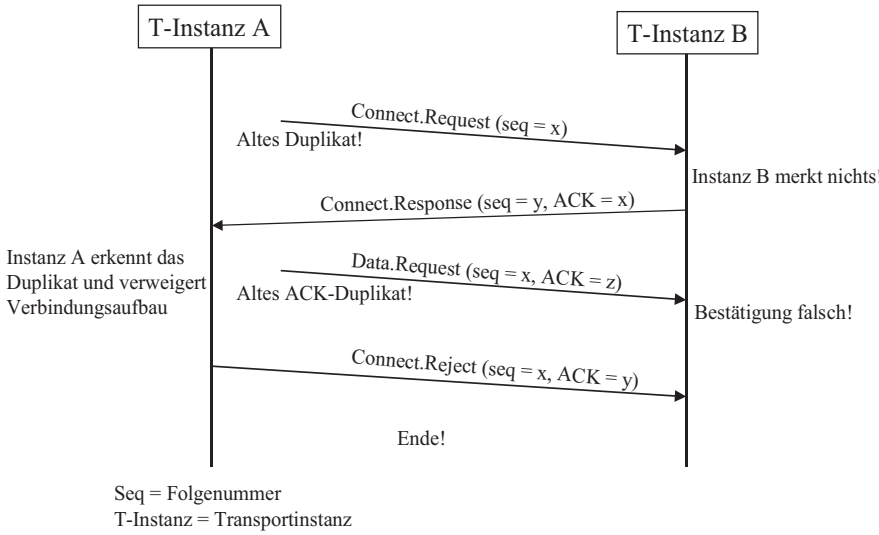


Abb. 2.4 Verbindungsaufbau, Fehlerfall 2: CR- und ACK-Duplikate tauchen auf

2.2.2 Verbindungsabbau

An einen ordnungsgemäßen Verbindungsabbau werden einige Anforderungen gestellt. Beim Verbindungsabbau dürfen keine Nachrichten verloren gehen. Ein Datenverlust kann nämlich vorkommen, wenn eine Seite einen Verbindungsabbau initiiert, die andere aber

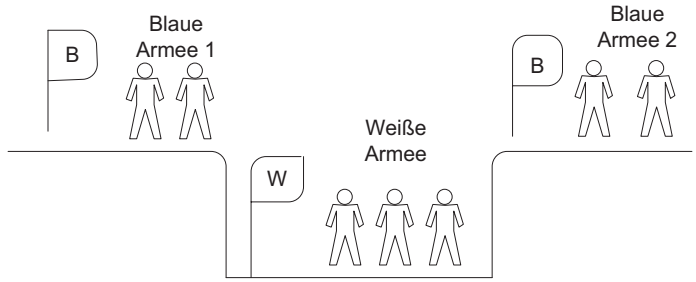


Abb. 2.5 Das Zwei-Armeen-Problem nach (Tanenbaum und Wetherall 2011)

vor Erhalt der Disconnect-Request-PDU noch eine Nachricht sendet. Diese Nachricht ist dann verloren (Datenverlust). Daher ist ein anspruchsvolles Verbindungsabbau-Protokoll notwendig. Auch hier verwendet man gerne einen Dreiwege-Handshake-Mechanismus, in dem beide Seiten[1] ihre „Senderichtung" abbauen.

Das Problem wird durch das Zwei-Armeen-Beispiel transparent. Die Armee der Weißröcke lagert in einem Tal. Auf zwei Anhöhen lagert jeweils ein Teil der Armee der Blauröcke. Die Blauröcke können nur gemeinsam gewinnen und müssen ihren Angriff synchronisieren. Zur Kommunikation der beiden Teilarmeen der Blauröcke existiert ein unzuverlässiger Kommunikationskanal, und zwar Boten, die zu Fuß durch das Tal rennen müssen (Abb. 2.5). Die eine Seite der Blauröcke sendet einen Boten mit einer Nachricht los. Damit sie aber sicher sein kann, dass er angekommen ist, muss die zweite Seite die Nachricht über einen weiteren Boten bestätigen. Was ist aber, wenn der nicht ankommt? Und wenn er ankommt, woher weiß es dann die zweite Seite? Kein Protokoll ist hier absolut zuverlässig, denn es wird immer eine Seite geben, die unsicher ist, ob die letzte Nachricht angekommen ist. Übertragen auf den Verbindungsabbau bedeutet dies, dass beim Dreiwege-Handshake jederzeit ein Disconnect-Request oder eine Bestätigung verloren gehen können. Man löst das Problem in der Praxis pragmatisch über eine *Timerüberwachung* mit begrenzter Anzahl an Nachrichtenwiederholungen. Dies liefert dann keine unfehlbaren, aber doch ganz zufriedenstellende Ergebnisse. Aber ein Problem bleibt bestehen: Falls der erste Disconnect-Request und n weitere verloren gehen, baut der Sender die Verbindung ab und der Partner weiß nichts davon. In diesem Fall liegt eine *halb offene* Verbindung vor. Aber auch dieses Problem kann gelöst werden, indem beim Senden einer Nachricht nach einer bestimmten Zeit die Verbindung abgebaut wird, wenn keine Antwort zurückkommt. Der Partner baut dann auch die Verbindung irgendwann wieder ab. Im Weiteren sind einige Szenarien für die Timerüberwachung beim Verbindungsabbau skizziert, die in einem verbindungsorientierten Transportprotokoll behandelt werden sollten.

[1] Wir gehen hier von einer Vollduplex-Verbindung aus.

Abb. 2.6 stellt den normalen Verbindungsabbau als Dreiwege-Handshake dar. In Abb. 2.7 ist ein Szenario skizziert, in dem ein Timer abläuft. In diesem Fall trennt die Instanz B die Verbindung beim Ablauf des Timers.

In Abb. 2.8 geht aus irgendeinem beliebigen Grund (z. B. Netzwerküberlastung) die Disconnect-Response-PDU der Instanz B verloren und kommt daher nicht bei Instanz A an. In diesem Fall läuft bei Instanz A der Timer ab, und die Disconnect-Request-PDU wird erneut gesendet.

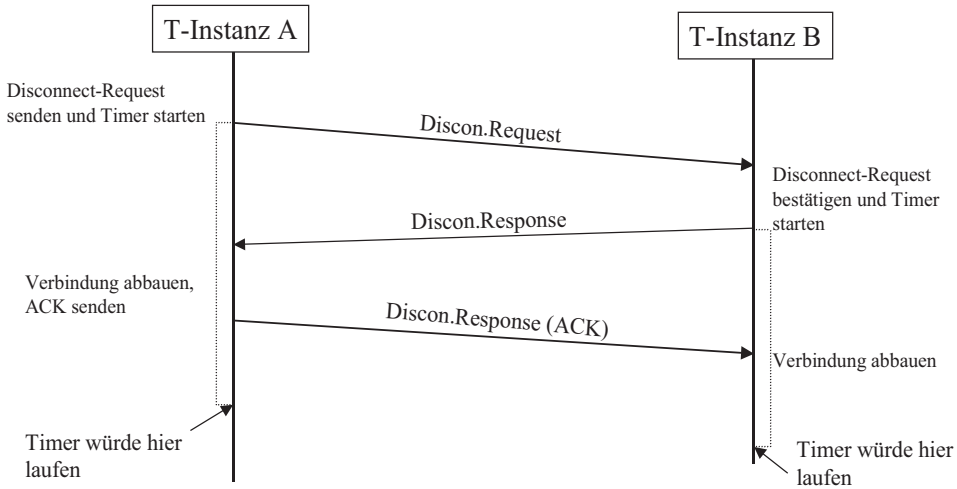


Abb. 2.6 Szenario beim Verbindungsabbau, normaler Ablauf

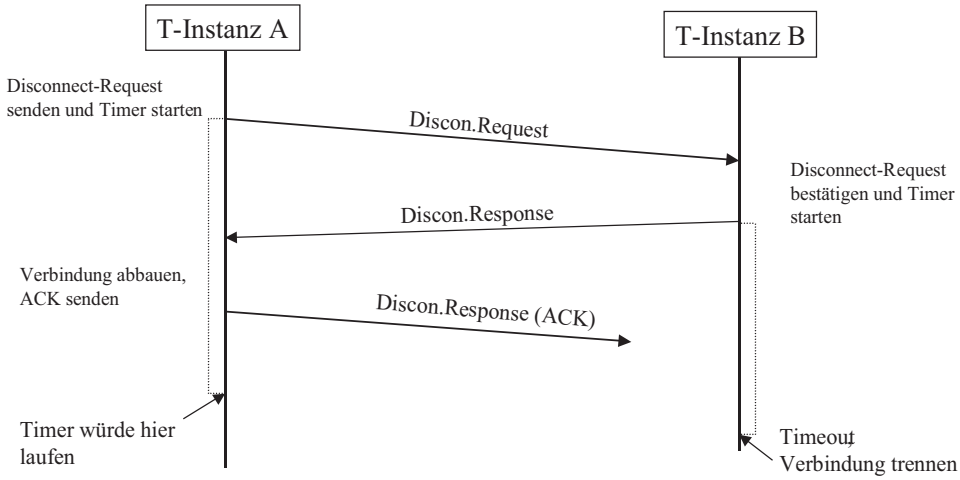


Abb. 2.7 Szenario beim Verbindungsabbau, Timerablauf

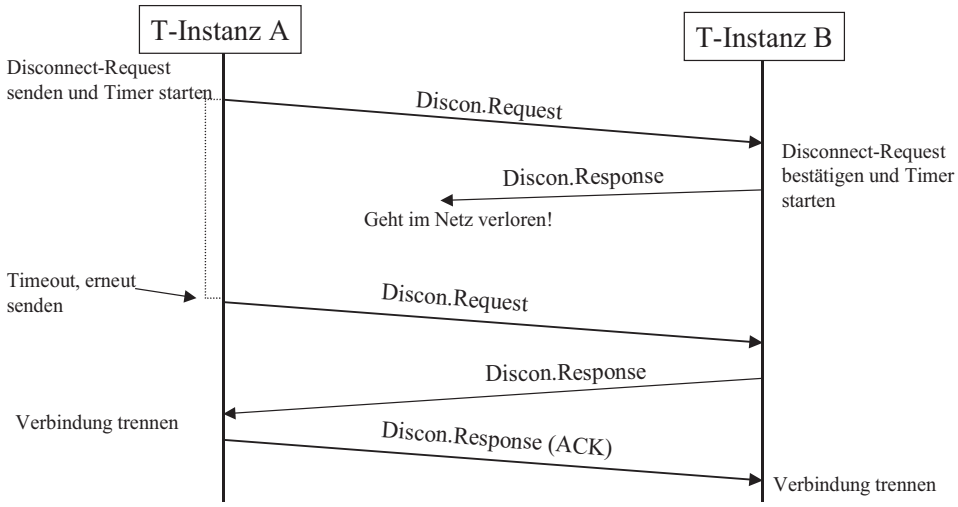


Abb. 2.8 Szenario beim Verbindungsabbau, Disconnect-Responses gehen verloren

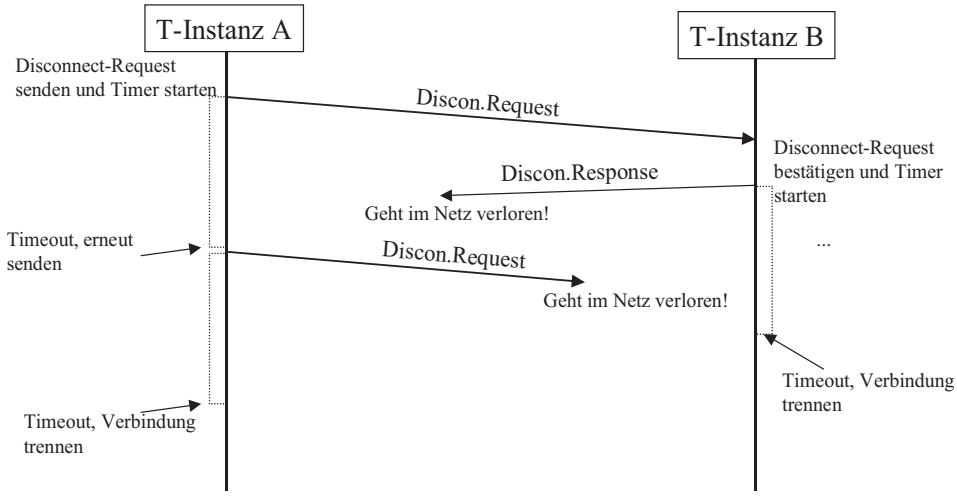


Abb. 2.9 Szenario beim Verbindungsabbau, zwei Disconnect-PDUs gehen verloren

Schließlich gehen in dem Szenario, das in Abb. 2.9 skizziert ist, die Disconnect-Response-PDU der Instanz B und eine erneut gesendete Disconnect-Request-PDU der Instanz A verloren. Hier greift auf beiden Seiten der Timeout-Mechanismus.

Die Verbindungen werden selbstständig von den beiden Instanzen getrennt, wenn die Wartezeit auf die Antwortnachricht des jeweiligen Partners abgelaufen ist. Wie oft eine Nachricht bei einem Timeout-Ereignis wiederholt wird, hängt vom jeweiligen Transportprotokoll ab. Es gibt sicherlich noch weitere Fehlervariationen, die aber alle über die Timerüberwachung lösbar sind.

2.3 Zuverlässiger Datentransfer

Bei einem zuverlässigen Datentransfer, wie ihn sowohl TCP als auch OSI TP 4,[2] ein Standardprotokoll der ISO/OSI-Welt, gewährleisten, werden die Daten in der richtigen Reihenfolge und vollständig übertragen. Fehler während der Übertragung werden erkannt und es erfolgt bei Bedarf eine Neuübertragung. Auch sonstige Probleme werden erkannt und vom Transport-Provider, sofern möglich, gelöst oder über die Diensteschnittstelle nach oben weiter gemeldet. Weiterhin werden bei einem zuverlässigen Datentransfer Duplikate vermieden. Ein zuverlässiger Datentransfer erfordert daher geeignete Protokollmechanismen zur Fehlererkennung und Fehlerbehebung, ein Empfänger-Feedback (Bestätigungen), eine Möglichkeit der Neuübertragung von Daten sowie eine geeignete Flusskontrolle.

2.3.1 Quittierungsverfahren

Welche Protokollmechanismen ein Transportprotokoll tatsächlich bereitstellen muss, um Zuverlässigkeit zu gewährleisten, hängt auch vom Dienst der Schicht 3 ab. Bei einem sehr zuverlässigen Kanal der Schicht 3 muss die Schicht 4 natürlich weniger tun als bei mehr verlustbehafteten Kanälen. Um sicher zu sein, dass Nachrichten in einem verlustbehafteten Kanal richtig ankommen, sind Quittierungsverfahren erforderlich. Hier gibt es gravierende Unterschiede hinsichtlich der Leistung. Einfache Verfahren quittieren jede Data-PDU mit einer ACK-PDU (Acknowledge), was natürlich nicht sehr leistungsfähig ist. Der Sender muss bei diesem Stop-and-Wait-Protokoll immer warten, bis eine Bestätigung eintrifft, bevor er weitere Data-PDUs senden kann. Aufwändigere Verfahren ermöglichen das Senden mehrerer Daten und sammeln die Bestätigungen gegebenenfalls kumuliert ein. Hier spricht man auch von *Pipelining*.

Man unterscheidet bei Quittungsverfahren:

- *Positiv-selektives* Quittierungsverfahren: Hier wird vom Empfänger eine Quittung (ACK) pro empfangener Nachricht gesendet. Dies hat einen hohen zusätzlichen Nachrichtenverkehr zur Folge. Abb. 2.10 zeigt dieses Verfahren.
- *Positiv-kumulatives* Quittierungsverfahren: Eine Quittung wird für mehrere Nachrichten verwendet. Diese Reduzierung der Netzlast hat aber den Nachteil, dass Information über einen Datenverlust verspätet an den Sender übertragen wird.
- *Negativ-selektives* Quittierungsverfahren: Es werden vom Empfänger nur selektiv nicht empfangene, also verloren gegangene Nachrichten erneut vom Sender angefordert. Alle Nachrichten, bei denen keine Nachfrage kommt, gelten beim Sender als angekommen. Dieses Verfahren reduziert die Netzlast weiter. Ein Verlust von negativen Quittungen (Nachfragen des Empfängers) ist jedoch möglich und muss behandelt werden.
- Eine *Kombination der Verfahren*: In Hochleistungsnetzen verwendet man z. B. das positiv-kumulative Verfahren kombiniert mit dem negativ-selektiven Verfahren.

[2] Genormt in TU-T Rec. X214 unter ISO/IEC 8072.

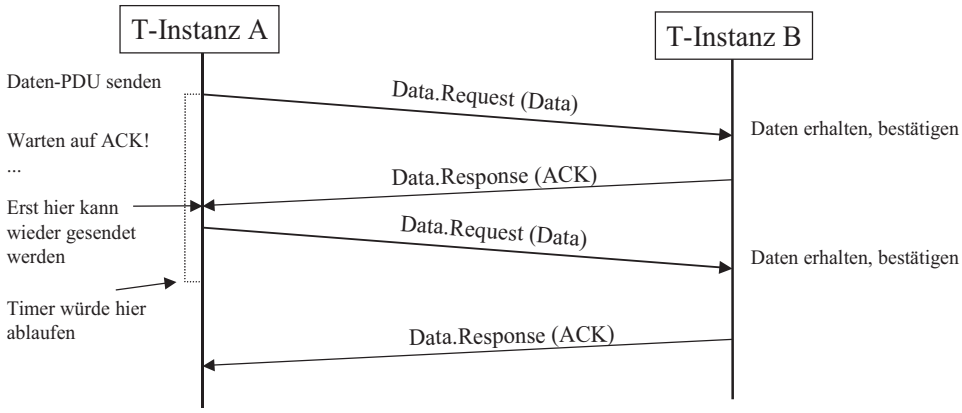


Abb. 2.10 Positiv-selektives Quittierungssverfahren

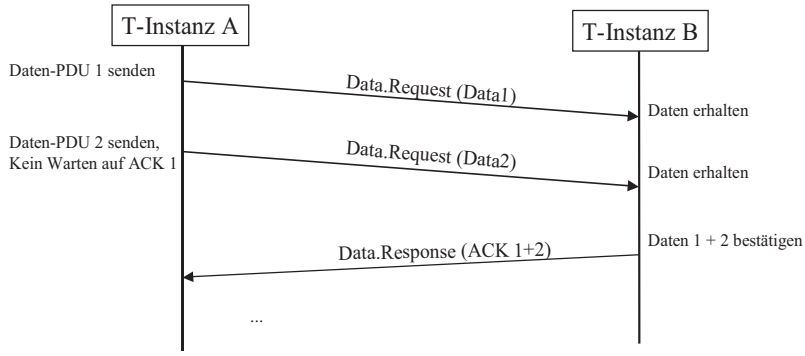


Abb. 2.11 Positiv-kumulatives Quittungsverfahren

Das positiv-kumulative Verfahren ist in Abb. 2.11 angedeutet. Im Szenario werden von Instanz A zwei Nachrichten gesendet und durch Instanz B gemeinsam bestätigt. Instanz A muss in diesem Fall nicht nach jeder Nachricht auf eine Bestätigung warten.

In Abb. 2.12 wird ein Beispiel für eine negativ-selektive Quittung dargestellt. Die zweite Nachricht kommt bei Instanz B nicht an und wird bei Eintreffen der Nachricht 3 nochmals angefordert. Falls die NAK-Nachricht (Not-Acknowledge) nicht beim Sender ankommt, weiß dieser vom Verschwinden der Nachricht 2 nichts. Daher muss die Instanz B durch Zeitüberwachung dafür sorgen, dass die NAK-Nachricht erneut gesendet wird.

2.3.2 Übertragungswiederholung

Verloren gegangene Nachrichten müssen erneut übertragen werden. Diesen Vorgang nennt man *Übertragungswiederholung*. Die positiven Quittierungsverfahren benötigen für jede gesendete PDU eine Timerüberwachung, damit nicht endlos auf eine ACK-PDU gewartet

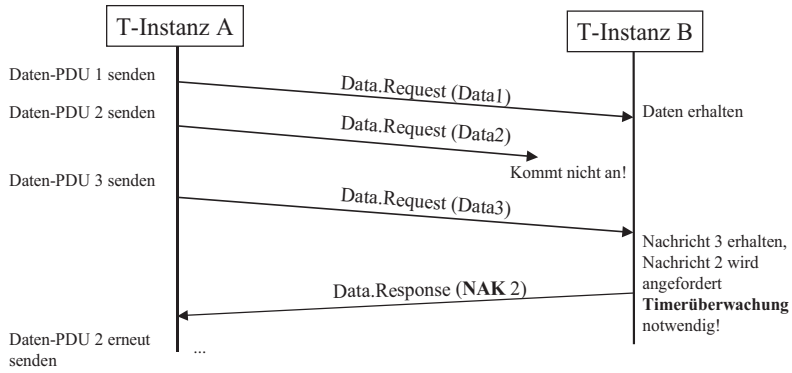


Abb. 2.12 Negativ-selektives Quittungsverfahren

wird. Die Übertragungswiederholung ist dabei auf zwei Arten üblich. Entweder man verwendet eine *selektive Wiederholung* oder ein *Go-Back-N-Verfahren*:

- Beim selektiven Verfahren werden nur die negativ quittierten Nachrichten wiederholt. Der Empfänger puffert die nachfolgenden Nachrichten, bis die fehlende da ist. Erst wenn dies der Fall ist, werden die Daten am Verbindungsendpunkt (T-SAP) nach oben zur nächsten Schicht weitergereicht. Nachteilig ist dabei, dass eine hohe Pufferkapazität beim Empfänger erforderlich ist. Von Vorteil ist, dass die reguläre Übertragung während der Wiederholung fortgesetzt werden kann.
- Beim Go-Back-N-Verfahren werden die fehlerhafte Nachricht sowie alle nachfolgenden Nachrichten erneut übertragen. Nachteilig ist hier, dass die reguläre Übertragung unterbrochen wird. Von Vorteil ist, dass beim Empfänger nur geringe Speicherkapazität erforderlich ist.
- Sender und Empfänger müssen sich über das verwendete Verfahren einig sein. Die genaue Vorgehensweise ist also in der Protokollspezifikation festzulegen.

Für beide Verfahren gilt generell, dass der Sender Nachrichten über einen gewissen Zeitraum zur Übertragungswiederholung bereithalten muss. Es kann ja jederzeit vorkommen, dass eine Nachricht erneut angefordert wird. Der Sender muss nur eine Nachricht speichern und kann sie bei Empfang der ACK-PDU verwerfen. Schwieriger ist es beim positiv-kumulativen und beim negativ-selektiven Verfahren. Beim negativ-selektiven Verfahren ist es für den Sender problematisch festzustellen, wann eine gesendete Nachricht aus dem Puffer eliminiert werden kann. Der Sender weiß ja nie genau, ob die Nachrichten beim Empfänger angekommen sind oder nicht. Deshalb wird dieses Verfahren in seiner reinen Form auch selten verwendet.

In Abb. 2.13 ist ein Go-Back-N-Verfahren skizziert, wobei eine negativ-selektive Quittierung (Empfänger meldet aktiv fehlende PDUs) angenommen wird. Die Data-PDU mit Nummer 2 wird von Instanz B angemahnt. Daraufhin werden von Instanz A die Data-PDUs mit der Nummer 2 und die ebenfalls bereits gesendete (aber angekommene) PDU

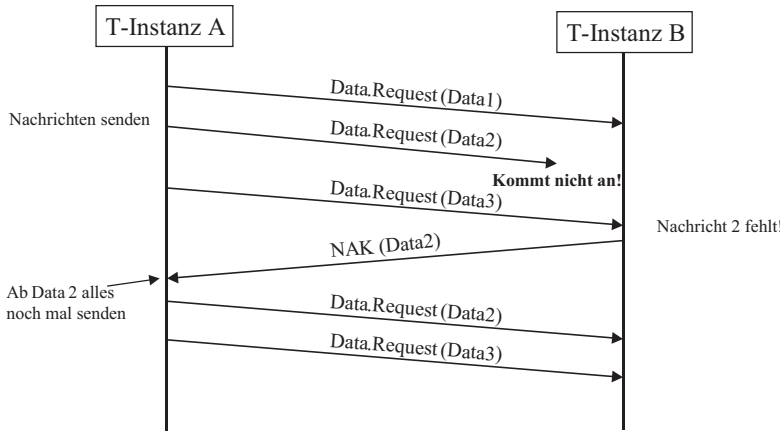


Abb. 2.13 Negativ-selektive Quittung und Go-Back-N-Verfahren

mit Folgenummer 3 erneut gesendet. Bei Netzen mit geringer Pfadkapazität (siehe Definition) ist mit Go-Back-N die Netzbelastung relativ gering. Anders dagegen ist dies in Netzen mit hoher Pfadkapazität, da bei entsprechender Fenstergröße (siehe Flusskontrolle) viele PDUs gesendet werden können, bevor die erste Negativ-Quittung beim Sender eintrifft. Dies führt dann gegebenenfalls zur unnötigen Übertragungswiederholung vieler PDUs.

Abschließend soll noch erwähnt werden, dass ein zuverlässiger Datentransfer noch keine Verarbeitungssicherheit und schon gar keine Transaktionssicherheit gewährleistet. Zuverlässiger Datentransfer bedeutet „lediglich“, dass die von einem Sendeprozess abgesendeten Nachrichten bei der T-Instanz des Empfängers ankommen, also dort in den Empfangspuffer eingetragen werden. Ob sie vom Empfängerprozess abgearbeitet werden oder gar zu den gewünschten Datenbankzugriffen führen, ist nicht gesichert. Hierfür werden höhere Protokolle, sogenannte Transaktionsprotokolle benötigt, die in der Anwendungsschicht platziert und noch komplexer als Transportprotokolle sind. Man findet Realisierungen von Transaktionsprotokollen in heutigen Datenbankmanagementsystemen.

► **[Pfadkapazität]** Unter der Pfadkapazität versteht man die Speicherkapazität eines Mediums entlang eines Kommunikationspfades, die sich aufgrund der begrenzten Ausbreitungsgeschwindigkeit der Signale ergibt.

Beispiel: Bei einer Übertragungsstrecke von 5000 km ergibt sich bei einer Signalausbreitungsgeschwindigkeit von $2 * 10^8$ m/s eine Signallaufzeit von ca. 25 ms. Bei einer Übertragungsrate von 64 Kbit/s ergibt das eine Speicherkapazität im Medium auf der Strecke von 1600 Bit. Bei einer Übertragungsrate von 2 Mbit/s ergibt sich schon eine Speicherkapazität von 50.000 Bit.

2.4 Flusskontrolle

Unter Flusskontrolle versteht man in der Schicht 4 in etwa das Gleiche wie in der Schicht 2, mit dem Unterschied, dass man in der Schicht 4 eine Ende-zu-Ende-Verbindung gegebenenfalls über ein komplexes Netzwerk verwaltet, während die Schicht 2 eine Ende-zu-Ende-Verbindung zwischen zwei Rechnersystemen unterstützt. Letzteres ist wesentlich einfacher in den Griff zu bekommen.

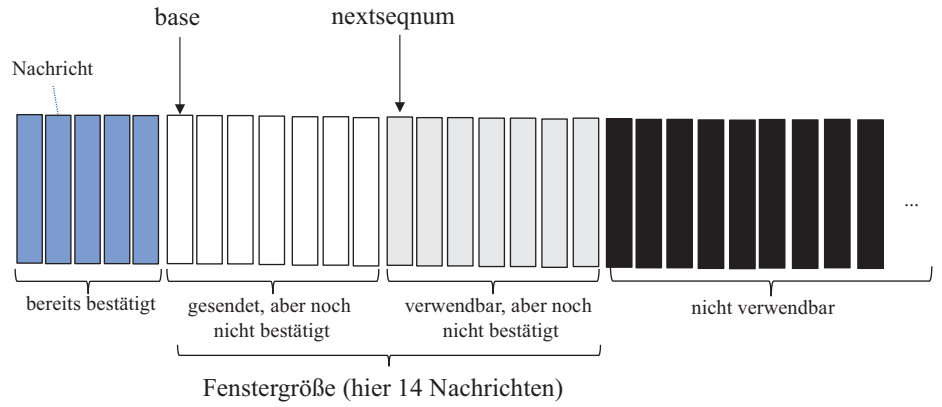
Durch die Steuerung des Datenflusses soll eine Überlastung des Empfängers vermieden werden. Traditionelle Flusskontroll-Verfahren sind:

- *Stop-and-Wait-Verfahren:* Dies ist das einfachste Verfahren, wobei eine Kopplung von Fluss- und Fehlerkontrolle durchgeführt wird. Die nächste Nachricht wird erst nach einer erfolgreichen Quittierung gesendet (Fenstergröße 1).
- *Fensterbasierte Flusskontrolle* (Sliding-Window-Verfahren): Der Empfänger vergibt einen sogenannten *Sendekredit*, also eine maximale Menge an Nachrichten oder Bytes, die unquittiert an ihn gesendet werden dürfen. Der Sendekredit reduziert sich bei jedem Senden. Der Empfänger kann den Sendekredit durch positive Quittungen erhöhen. Der Vorteil dieses Verfahrens ist, dass ein kontinuierlicher Datenfluss und höherer Durchsatz als bei Stop-and-Wait möglich ist.

Für die fensterbasierte Flusskontrolle werden in den Transportinstanzen für jeden Kommunikationspartner vier Folgennummern-Intervalle verwaltet, die grob in Abb. 2.14 dargestellt sind:

- *Das* linke Intervall sind Folgenummern, die gesendet und vom Empfänger bereits bestätigt wurden.
- Das zweite Intervall von links stellt alle Folgenummern dar, die gesendet, aber vom Empfänger noch nicht bestätigt wurden. Der Zeiger *base* verweist auf den Anfang dieses Intervalls.
- Das nächste Intervall gibt alle Folgenummern an, die noch verwendet werden dürfen, ohne dass eine weitere Bestätigung vom Empfänger eintrifft. Der Zeiger *nextseqnum* zeigt auf den Anfang des Intervalls. Das vierte Intervall zeigt die Folgenummern, die noch nicht verwendet werden dürfen.

Die beiden inneren Intervalle bilden zusammen das aktuelle Fenster, geben also den verfügbaren Sendekredit an. Trifft nun eine Bestätigung des Empfängers ein, wandert das Fenster um die Anzahl der bestätigten Folgenummern nach rechts, d. h. der Zeiger *base* wird nach rechts verschoben. Gleichzeitig wird eine entsprechende Anzahl an Folgenummern aus den bisher nicht verwendbaren Folgenummern dem dritten Intervall zugeordnet. Der Zeiger *nextseqnum* wird nicht verändert. Erst wenn wieder Nachrichten gesendet werden, wird dieser entsprechend der Anzahl an benutzten Folgenummern nach rechts verschoben. In Abb. 2.14 ist auch die Situation nach drei empfangenen Bestätigungen dargestellt.



Situation nach der Bestätigung von drei Nachrichten:

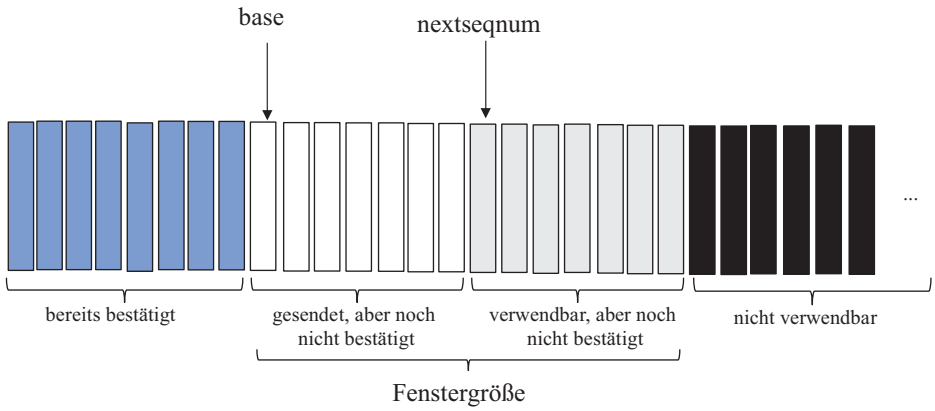


Abb. 2.14 Intervallverwaltung für die fensterbasierte Flusskontrolle

Die Größe des Sendekredits ist entscheidend für die Leistung des Protokolls. Zu große Kredite gewährleisten keinen richtigen Schutz der Ressourcen beim Empfänger. Eine zu starke Limitierung führt zu schlechterer Leistung. Die Größe des Sendekredits hängt auch von der Pfadkapazität ab, also davon, wie viele Nachrichten im Netzwerk zwischen Sender und Empfänger gespeichert werden können.

In Abb. 2.15 ist ein Beispiel für eine fensterbasierte Flusskontrolle in einem Netz mit niedriger Pfadkapazität gezeigt. Der anfänglich bereitgestellte Sendekredit wird immer wieder schnell genug durch den Empfänger bestätigt, so dass es zu keinen Wartezeiten im Sender kommt.

Bei Netzwerken mit hoher Pfadkapazität führen kleine Sendekredite häufig dazu, dass der Sender durch das Warten auf Kreditbestätigungen blockiert wird. Dies ist in Abb. 2.16 skizziert. Bei einem Sendekredit von drei Nachrichten entstehen beträchtliche Wartezeiten

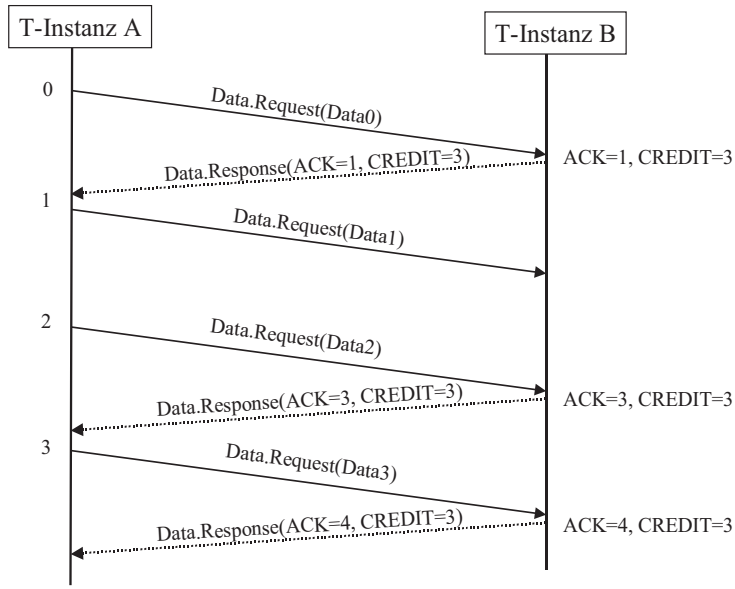


Abb. 2.15 Fensterbasierte Flusskontrolle bei geringer Pfadkapazität nach (Zitterbart und Braun 1996)

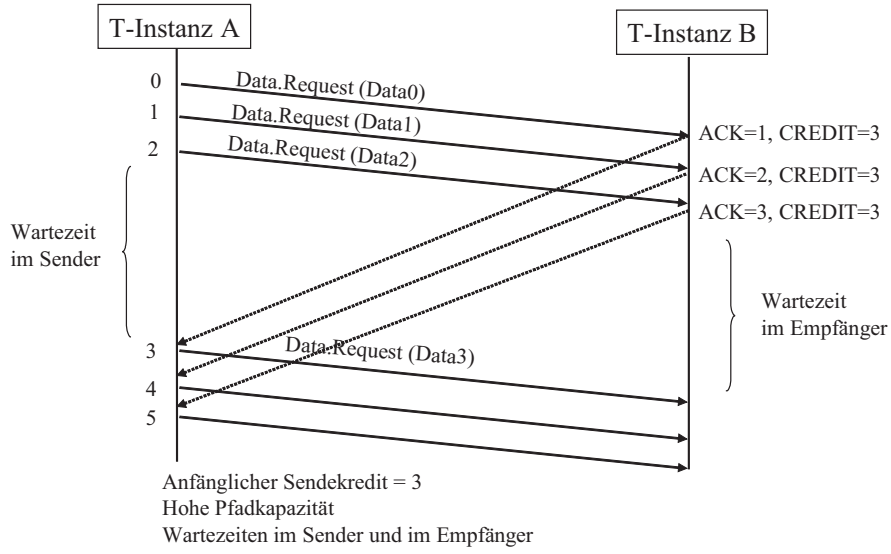


Abb. 2.16 Fensterbasierte Flusskontrolle bei hoher Pfadkapazität nach (Zitterbart und Braun 1996)

im Sender und auch im Empfänger. Die Instanz A hat häufig keinen Sendekredit. Wird schließlich wieder ein Sendekredit eingeräumt, entstehen Burst-Übertragungen (Übertragung von großen Mengen) in Fenstergröße.

2.5 Staukontrolle

Durch Staukontrolle (Congestion Control) sollen Verstopfungen bzw. Überlastungen im Netz vermieden werden. Maßnahmen zur Staukontrolle können in den Schichten 2 bis 4 durchgeführt werden und beziehen sich in der Schicht 4 überwiegend auf die Steuerung der Ende-zu-Ende-Beziehung.

Staukontrolle ist ein Mechanismus mit netzglobalen Auswirkungen. Einige Netzwerkprotokolle (wie ATM ABR)[3] liefern am N-SAP gewisse Informationen, andere (wie das Internet-Protokoll) nicht.

In der Transportschicht hatte man ursprünglich in der TCP/IP-Protkollfamilie nur die Möglichkeit, aus Sicht einer bestimmten Ende-zu-Ende-Verbindung Maßnahmen zu ergreifen, wenn ein Engpass erkannt wurde. Eine typische Maßnahme ist die Drosselung des Datenverkehrs durch Zurückhalten von Paketen. Heute versucht man sich mit weiteren Mechanismen, wie wir noch sehen werden (siehe *Explicit Congestion Notification*). Bei der Behandlung des Transportprotokolls TCP wird in Kap. 3 detailliert auf das Thema Staukontrolle am konkreten Protokollbeispiel eingegangen.

2.6 Segmentierung

Eine T-SDU, die an einem T-SAP übergeben wird, hat üblicherweise eine beliebige Länge. Eine T-PDU hat aber oft nur eine bestimmte Maximallänge, auch MSS (Maximum Segment Size) genannt. In diesem Fall muss die T-Instanz zu große T-SDUs in mehrere Segmente zerstückeln und einzeln übertragen. Diesen Vorgang nennt man Fragmentierung oder in der Schicht 4 auch Segmentierung.

Beim Empfänger ankommende Segmente, welche die N-Instanz einer T-Instanz liefert, müssen entsprechend wieder zusammengebaut werden, um eine vollständige T-SDU zu erhalten und nach oben weiterreichen zu können. Diesen Vorgang nennt man Defragmentierung oder Reassemblierung. Zuverlässige Transportprotokolle stellen sicher, dass auch keine Fragmente verloren gehen. Eine Schicht tiefer kann der gleiche Mechanismus übrigens nochmals angewendet werden.

Zur Optimierung versuchen leistungsfähige Protokolle jedoch, die Fragmentierung in Grenzen zu halten, da mit diesem Mechanismus viel Overhead verursacht wird.

[3] ATM ABR steht für Asynchronous Transfer Mode Available Bit Rate. ATM ist eine Protokollfamilie, die heute überwiegend in Zubringernetzen der Telekom und von Internet Providern im globalen Internet eingesetzt wird. ATM enthält neben dem ABR-Protokoll noch weitere Protokolle (Mandl et al. 2010).

2.7 Multiplexierung und Demultiplexierung

In vielen Fällen werden von einem Host zu einem anderen mehrere Transportverbindungen für eine oder sogar mehrere Anwendungen benötigt. Auf der Schicht 3 genügt aber eine Netzwerkverbindung, um alle Transportverbindungen zu bedienen.

Um N-Verbindungen besser zu nutzen, kann die Transportschicht eine N-Verbindung für mehrere Transportverbindungen verwenden. Diesen Mechanismus nennt man Multiplexieren oder Multiplexen bzw. den umgekehrten Vorgang Demultiplexieren bzw. Demultiplexen.

Die T-Instanz hat die Aufgabe, den Verkehr, der über verschiedene T-SAPs läuft, an einen N-SAP zu leiten. Umgekehrt werden ankommende Pakete, welche die Schicht 3 an die Schicht 4 weiterleitet, entsprechend den einzelnen T-SAPs zugeordnet. Dies geschieht mit Hilfe der Adress-Information.

Literatur

Mandl, P.; Bakomenko A.; Weiß, J. (2010) Grundkurs Datenkommunikation: TCP/IP-basierte Kommunikation: Grundlagen, Konzepte und Standards, 2. Auflage, Vieweg-Teubner Verlag, 2010

Tanenbaum, A. S.; Wetherall, D. J. (2011) Computer Networks, Fifth Edition, Pearson Education, 2011

Zitterbart, M.; Braun, T. (1996) Hochleistungskommunikation Band 2: Transportdienste und -protokolle, Oldenbourg Verlag, 1996

TCP-Konzepte und -Protokollmechanismen

3

Zusammenfassung

Das Transportprotokoll TCP ist in internetbasierten Netzwerken die Basis für die höherwertigen Kommunikationsmechanismen, wenn Verbindungsorientierung und zuverlässiger Transport erforderlich sind. TCP bzw. die implementierten TCP-Instanzen übernehmen die Aufgabe des Verbindungsaufbaus und der gesicherten Datenübertragung. Bestätigungs- und Wiederholungsverfahren sind ebenso in TCP vorhanden, wie eine Flusskontrolle. Die Segmentlängen werden ermittelt, so dass die TCP-Segmente auch durch die unterliegenden Schichten transportiert werden können. Optimierungen wie zum Beispiel eine kumulative Bestätigung sind in zugehörigen RFCs spezifiziert. Das Protokoll nutzt einen schlanken Header für die Steuerinformationen. In der TCP-Spezifikation sind der Verbindungsauf- und der Verbindungsabbau über Zustandsautomaten beschrieben. Weiterführende Konzepte und Mechanismen von TCP sind das Slow-Start-Verfahren zur Überlastkontrolle und auch der Einsatz spezieller Timer wie etwa der Retransmission Timer, der für die Arbeitsweise von TCP von großer Bedeutung ist. Auch die Sicherheit von TCP, die durch spezielle Angriffsvektoren gefährdet ist, darf nicht außer Acht gelassen werden. TCP wird ständig optimiert und weiterentwickelt und es gibt mittlerweile eine Fülle von RFCs, die sich mit TCP befassen. Diese Aspekte werden in diesem Kapitel im Detail erläutert.

3.1 Übersicht über grundlegende Konzepte und Funktionen

3.1.1 Grundlegende Aufgaben von TCP

TCP ist ein Transportprotokoll und gibt der TCP/IP-Protokollfamilie seinen Namen. Es ermöglicht eine Ende-zu-Ende-Beziehung zwischen kommunizierenden Anwendungsinstanzen. TCP ist sehr weit verbreitet und als Industrienorm akzeptiert. Es ist ein offenes,

P. Mandl, *TCP und UDP Internals*, https://doi.org/10.1007/978-3-658-20149-4_3

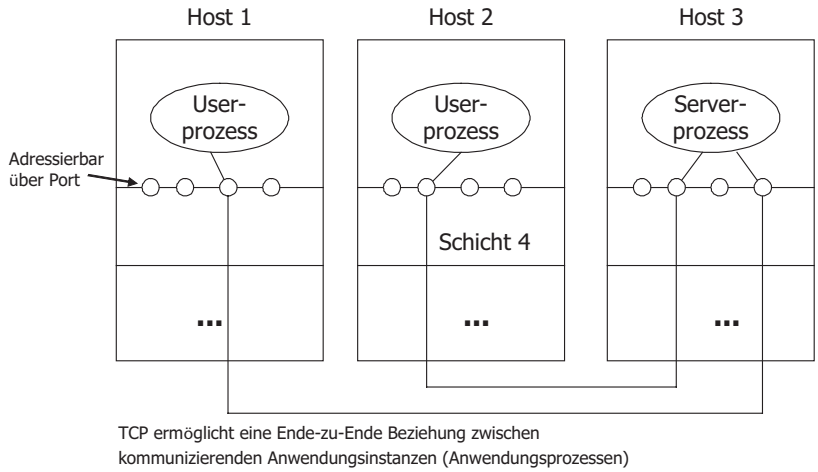


Abb. 3.1 Ende-zu-Ende-Kommunikation in TCP

frei verfügbares Protokoll und damit nicht an einen Hersteller gebunden. Neben UDP ist TCP das Transportprotokoll im Internet, auf das die meisten Anwendungen basieren.

Ursprünglich wurde TCP im RFC 793 spezifiziert. Er stammt aus dem Jahr 1981. Bis heute wurden viele Erweiterungen und Optimierungen umgesetzt. Ein Überblick über die wesentlichen RFCs, die TCP betreffen, ist im RFC 7414 zu finden.[1]

Als Transportzugriffsschnittstelle dient die Socket-Schnittstelle. Ein T-SAP wird in TCP durch einen Port mit einer Portnummer identifiziert. Ein Anwendungsprozess benötigt also einen lokalen TCP-Port und kommuniziert über diesen mit einem anderen Anwendungsprozess, der ebenfalls über einen TCP-Port adressierbar ist. Wie in Abb. 3.1 gezeigt, kann ein Anwendungsprozess durchaus mit mehreren anderen Anwendungsprozessen über den gleichen oder über verschiedene Ports kommunizieren.

TCP ist ein verbindungsorientiertes, zuverlässiges Protokoll. Grob gesagt kümmert sich TCP um die Erzeugung und Erhaltung einer gesicherten Ende-zu-Ende-Verbindung zwischen zwei Anwendungsprozessen auf Basis des Internet Protokolls IP. TCP garantiert weiterhin die Reihenfolge der Nachrichten und die vollständige Auslieferung, übernimmt die Fluss- und Staukontrolle und kümmert sich um die Multiplexierung sowie die Segmentierung.

TCP stellt also sicher, dass Daten

- nicht verändert werden,
- nicht verloren gehen,
- nicht dupliziert werden und
- in der richtigen Reihenfolge eintreffen.

[1] RFC 7414: A Roadmap for Transmission Control Protocol (TCP) Specification Documents. Februar 2015.

TCP ermöglicht die Kommunikation über *vollduplex-fähige*, *bidirektionale* virtuelle Verbindungen zwischen Anwendungsprozessen. Das bedeutet, beide Kommunikationspartner können zu beliebigen Zeiten Nachrichten senden, auch gleichzeitig.

Wichtig ist auch, dass TCP am T-SAP eine stromorientierte Kommunikation (Stream) im Unterschied zur blockorientierten Übertragung (wie bei OSI TP4) unterstützt. Daten werden also von einem Anwendungsprozess Byte für Byte in einen Bytestrom geschrieben. Die TCP-Instanz kümmert sich um den Aufbau von Segmenten, die übertragen werden. Dies bleibt für Anwendungsprozesse transparent.

Maßnahmen zur Sicherung der Übertragung sind die Nutzung von Prüfsummen, Bestätigungen, Zeitüberwachungsmechanismen mit verschiedenen Timern, Nachrichtenwiederholung, Sequenznummern für die Reihenfolgeüberwachung und das Sliding-Windows-Prinzip zur Flusskontrolle.

TCP nutzt folgende Protokollmechanismen:

- Mehr-Wege-Handshake-Verbindungsauf- und -abbau.
- Positiv-kumulatives Bestätigungsverfahren mit Timerüberwachung für jede Nachricht.
- Implizites, negatives Bestätigungsverfahren (NAK-Mechanismus): Bei drei ankommenden Duplikat-ACK-PDUs wird beim Sender das Fehlen des folgenden Segments angenommen. Ein sogenannter Fast-Retransmit-Mechanismus führt zur Neuübertragung des Segments, bevor der Timer abläuft.
- Go-Back-N zur Übertragungswiederholung.
- Flusskontrolle über Sliding-Window-Mechanismus und
- Staukontrolle über spezielle Verfahren.

Optional sind auch weitere Protokollmechanismen möglich. In diesem Kapitel werden zunächst für ein grundlegendes Verständnis die Basismechanismen erläutert, während weiterführende Mechanismen im folgenden Kapitel dargestellt werden.

3.1.2 Nachrichtenlänge

Für den TCP-Dienstnehmer wird die Übertragung der Nutzdaten über die Socket API als ankommender und abgehender Datenstrom abstrahiert. Die Anwendung übergibt nach dem Verbindungsaufbau eine Sequenz von Octets (Bytes) an die TCP-Instanz und empfängt eine Sequenz. Die TCP-Instanz unterteilt diese Octet-Sequenz zur Übertragung in TCP-Segmente oder T-PDUs, wobei eine Maximallänge wichtig ist. Die größtmögliche Segmentlänge ist durch einen Parameter namens *Maximum Segment Size* (MSS) begrenzt. In Abb. 3.2 ist zur Unterscheidung nochmals die Abgrenzung von TCP-Segmenten zu IP-Paketen bzw. IP-Fragmenten dargestellt. IP-Fragmente sind nochmals Stückelungen der IP-Pakete.

Wie lange darf ein TCP-Segment nun maximal sein, was gibt die MSS vor und wie wird sie ermittelt? Grundsätzlich wird die MSS durch eine Vereinbarung zwischen den

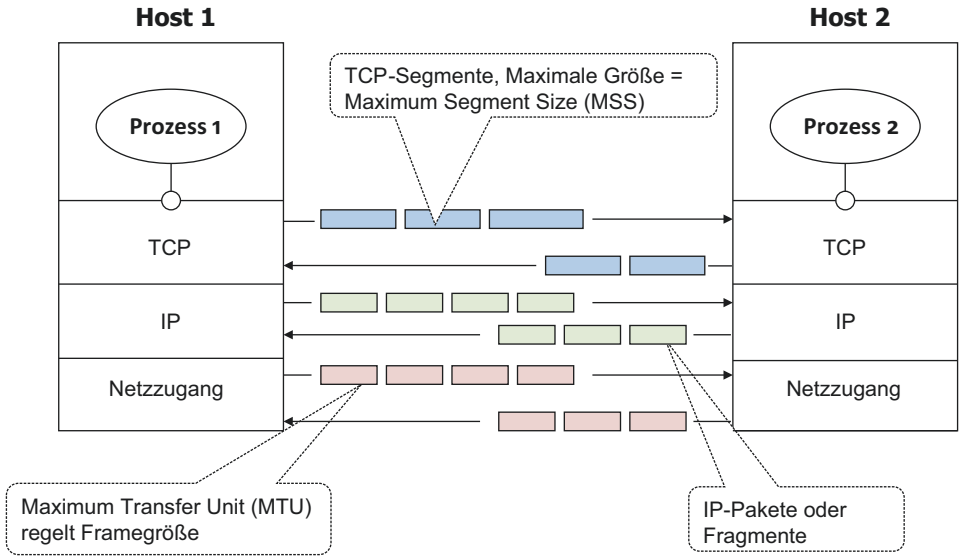


Abb. 3.2 TCP-Segmente in der Schichtenzuordnung

TCP-Partnern bzw. deren Instanzen begrenzt. Wie wir später noch beim Verbindungsaufbau sehen werden, kann die MSS optional beim Verbindungsaufbau für beide Richtungen vereinbart werden. Kann die IP-Instanz ein übergebenes TCP-Segment nicht in einem IP-Paket senden, dann wird eine IP-Fragmentierung nach festgelegten Regeln vorgenommen. Eine IP-Fragmentierung kann auch dadurch begründet werden, dass der Netzzugang nicht so große Frames verarbeiten kann.

Für die Bestimmung der Segmentgröße ist es also auch wichtig zu wissen, wie viele Nutzdatenbytes in den unteren Schichten in einem IP-Paket bzw. in einem Frame transportiert werden können. Die Größe der ausgehenden Nachrichten ist durch Einschränkungen in der benutzten Netzwerktechnologie limitiert. Diese Obergrenze wird als *Maximum Transmission Unit* oder *Maximum Transfer Unit* (MTU) bezeichnet. Endsysteme dürfen in IPv4-Netzwerken über ihren Netzwerkzugang nur Frames mit einer Länge senden, die die MTU nicht überschreiten. Ist nichts über die Kommunikationsverbindung bekannt, so gilt 576 Bytes als Standardeinstellung für die MTU. Bei IPv6 ist das zu unterstützende MTU-Minimum auf 1280 Bytes festgelegt. Das bedeutet auch, dass alle Router diese Größe ohne Fragmentierung verarbeiten können müssen.[2]

Der beim Sendevorgang an das Netzwerk übergebene Frame ist also durch die MTU begrenzt. Der Frame enthält jedoch die Nutzdaten und auch alle Header der beteiligten Protokolle TCP, IP und gegebenenfalls auch von weiteren Protokollen wie PPP und PPPoE

[2] Eine gute Diskussion über die Anzahl der Nutzdaten, die in einem Segment übertragen werden sollen, ist im RFC 879 zu finden.

(bei DSL-Netzzugängen). Daraus lässt sich für IPv4-Netze Folgendes ableiten, sofern nichts über die einzelnen Teilnetze zwischen Sender und Empfänger bekannt ist:

$$MTU \geq MSS + TCPHeaderLänge + IPv4HeaderLänge + Länge\,weiterer\,Header$$

Wir wissen, dass die Standard-TCP-Headerlänge 20 Octets und auch die Standard-IPv4-Headerlänge 20 Octets lang ist. Daraus ergibt sich die MSS gemessen in Octets wie folgt:

$$576 \geq MSS + 20 + 20$$

$$MSS \leq 576 - 40$$

$$MSS \leq 536$$

Berücksichtigt man beispielsweise in DSL-Zugängen noch die verwendeten Protokolle wie PPP und PPPoE (8 Octets), ist die MSS weiter zu reduzieren. Aus Sicherheitsgründen könnte man auch die maximale IPv4-Headerlänge von 60 Octets berücksichtigen und käme dann auf 496 Octets für die Nutzdaten, die in der Transportschicht in einem TCP-Segment maximal übertragen werden dürfen. Berücksichtigt man dies bereits in der TCP-Instanz, so kann Fragmentierung weitgehend vermieden werden. Das ist von Vorteil, da Fragmentierung in den IP-Routern und in den Endsystemen einen gewissen Overhead verursacht.

Die Begrenzung der MTU ist übrigens für jede Teilstrecke eines Netzes, durch das eine Nachricht transportiert werden muss, gegeben. Besser wäre es also, wenn man vor dem Senden einer Nachricht die MTU in Erfahrung bringen könnte, die durch alle Teilnetze bis zum Zielrechner problemlos ohne Fragmentierung gesendet werden kann. In diesem Fall könnte man Fragmentierung ganz vermeiden und man würde auch nicht zu kleine Nachrichten senden, da man ja die größtmögliche MTU kennt. Tatsächlich wird dies in TCP/IP-Netzen auch durchgeführt. Diesen Vorgang nennt man gemäß RFC 879 Path MTU Discovery.

Path MTU Discovery

Das Verfahren Path MTU Discovery oder kurz PMTUD dient einem Endsystem dazu, die für die Kommunikation mit einem anderen Endsystem (Host) beste MTU unter Berücksichtigung des gesamten Netzwerkpfads von der Quelle zum Ziel herauszufinden. Ziel ist es, die aufwändige IP-Fragmentierung, also die Zerlegung eines IP-Pakets in IP-Fragmente zu vermeiden. Für IPv4 ist dies im RFC 1191 geregelt.

Das Verfahren bedient sich eines kleines Tricks. Es werden nämlich spezielle IP-Pakete durch das Netzwerk gesendet, für die festgelegt wird, dass eine Fragmentierung nicht zulässig ist. Am Anfang sendet das Endsystem ein sehr großes Paket, so wie es etwa das lokale LAN erlaubt.

Die Router, welche das IP-Paket nicht ohne Fragmentierung weiterleiten können, senden eine Fehlermeldung in einer ICMP-Nachricht an das Endsystem zurück und geben dabei auch jeweils die mögliche MTU für das betroffene Teilnetz an. ICMP ist ein Steuerprotokoll, das für viele andere Aufgaben, wie etwa auch für das *ping*-Kommando, verwendet wird.

Das Endsystem wiederholt die Versuche mit reduzierten IP-Paketen so lange, bis das Paket ohne Fehler bis zum adressierten Endsystem übertragen wird. Im Weiteren kann dann bei jedem Senden diese MTU berücksichtigt werden. Ändert sich die Route zwischen den Endsystemen, muss der Vorgang gegebenenfalls wiederholt werden. PMTUD wird von TCP (und auch von UDP) unterstützt und auch kontinuierlich für jede Verbindung durchgeführt, da sich Routen auch ändern können. Die IP-Router müssen es ebenfalls unterstützen, wobei es in der Regel konfigurierbar ist. Firewalls dürfen bei diesem Verfahren die ICMP-Nachrichten nicht blockieren. Einige Probleme, die mit PMTUD auftreten können, werden im RFC 2923 diskutiert.

In IPv6-Netzwerken wird keine Fragmentierung mehr durchgeführt. Daher ist das Verfahren unbedingt notwendig, um Paketverluste zu vermeiden. Für IPv6 ist dies im RFC 1981 beschrieben.

MTU und MSS bei Ethernet-LANs

Die maximale Nutzdatenlänge beträgt bei einem Ethernet-Frame 1500 Bytes. Dies entspricht also der Ethernet-MTU. Der Ethernet-Header umfasst nochmals 18 Octets, so dass man insgesamt bei Ethernet-Frames auf 1518 Bytes kommt.

In reinen Ethernet-Umgebungen ist ein TCP-Segment, das Fragmentierung vermeiden soll, dann maximal 1460 Octets lang, sofern nur die Standard-Header von TCP und IP berücksichtigt werden. Dies wird durch folgende Berechnung bestätigt:

$$MTUEthernet \geq MSS + TCPHeaderLänge + IPvHeaderLänge$$
$$MSS \leq 1500 - 40 \rightarrow MSS \leq 1460$$

Eine TCP-Instanz sollte also maximal 1460 Octets in ein TCP-Segment packen. Der TCP-Header kommt noch hinzu.

Anmerkung: Man kann hier argumentieren, dass das Wissen über die maximale Frame-Länge bei genauer Betrachtung dem Kapselungsgedanken des Schichtenmodells widerspricht. Aus Optimierungsgründen ist es hier dennoch wichtig und wird in Kauf genommen.

3.1.3 Adressierung

Anwendungsprozesse sind über die Transportadressen ihrer T-SAP adressierbar. Bei TCP werden T-SAPs als *Sockets* bezeichnet. Eine Socketadresse setzt sich aus einem Tupel der Form (IP-Adresse, TCP-Portnummer) zusammen. Oft erfolgt die Kommunikation zwischen den Partnern auf der Basis einer Client-/Server-Rollenverteilung. Ein Serverprozess stellt einen Dienst bereit und bietet ihn über eine Portnummer an. Ein Port wird durch eine eindeutige Portnummer repräsentiert, die als 16-Bit-Integerzahl mit einem Wertebereich von 0 bis 65535 definiert ist.

Wenn ein Clientprozess die IP-Adresse und die Portnummer des Dienstes sowie das zugehörige Protokoll kennt, kann er mit dem Server kommunizieren und dessen Dienste nutzen, sofern dies nicht höhere Protokolle, wie etwa ein Sicherheitsprotokoll, einschränkt.

IANA[3] verwaltet die Ports und definiert

- sogenannte Well-known (wohlbekannte) Ports mit einem Nummernbereich von 0 bis 1023,
- registrierte Ports (Nummernbereich von 1024 bis 49151) und
- dynamische bzw. *private* Ports (Nummernbereich von 49152 bis 65535).

Es gibt eine Reihe von Well-known Ports für reservierte Services. Diese sind innerhalb der TCP/IP-Gemeinde bekannt und werden immer gleich benutzt. Wichtige TCP-Ports (Well-known Ports) und dazugehörige Dienste sind z. B.:[4]

[3] IANA = Internet Assigned Numbers Authority.

[4] In Unix- und Windows-Systemen sind Services mit reservierten Ports meist in einer Konfigurationsdatei wie z. B. /etc/services festgelegt.

- Port 23, Telnet (Remote Login),
- Ports 20 und 21, ftp (File Transfer Protocol),
- Port 25, SMTP (Simple Mail Transfer Protocol),
- Port 80, HTTP (Hyper Text Transfer Protocol) oder
- Port 443, HTTPS (HTTP auf der Basis von TLS).

Registrierte Ports werden durch IANA bestimmten Anwendungen fest zugeordnet. Die Hersteller der Anwendungen müssen die Zuordnung wie Domänennamen beantragen. Für folgende Anwendungen sind z. B. TCP-Ports registriert:

- Port 1109, Kerberos POP,
- Port 1433, Microsoft SQL Server,
- Port 1512, Microsoft Windows Internet Name Service (wins),
- Port 9000, Standardport des NameNode von Apache Hadoop,
- Port 23399, Standardport für Skype.

Dynamische bzw. private Ports können beliebig verwendet werden. Die Zuordnung wird von IANA nicht registriert. Mit diesen Portnummern ist es möglich, eigene Dienste zu definieren. Hier können in einem Netz auch Überlappungen auftreten. Beispielsweise kann ein Server 1 einen Dienst mit der Portnummer 52000 anbieten und ein Server 2 einen anderen Dienst mit derselben Portnummer 52000. Dies ist kein Problem, sofern die Anwendungsdomänen disjunkt sind. Auf einem Rechner kann ein Port aber nur einmal vergeben werden.

Eine Verbindung wird durch ein Paar von Endpunkten eindeutig identifiziert (Socket Pair). Dies entspricht einem Quadrupel, das die IP-Adresse von Host 1, die Portnummer, die in Host 1 genutzt wird, die IP-Adresse von Host 2 und die Portnummer auf der Seite von Host 2 enthält.

Dadurch ist es möglich, dass ein TCP-Port auf einem Host für viele Verbindungen verwendet werden kann, was in einigen Anwendungen auch intensiv genutzt wird.

Portnutzung durch HTTP-Server

Der HTTP-Port 80 wird für viele Verbindungen eines HTTP-Servers (hier im Beispiel mit der IP-Adresse 195.214.80.76) mit beliebig vielen Web-Clients (Browsern) verwendet. Mögliche TCP-Verbindungen aus Sicht des HTTP-Servers sind:

```
((195.214.80.76, 80) (196.210.80.10,6000))
((195.214.80.76, 80) (197.200.80.11,6001))
...
```

Man erkennt, dass serverseitig die Portnummer 80 für jede TCP-Verbindung mit Web-Clients genutzt wird und die Socket Pairs trotzdem alle eindeutig sind.

Wie schon angedeutet, nutzt man zur Ermittlung der Schicht-3-Adresse (IP-Adresse), die Bestandteil der Transportadresse ist, einen Naming-Service wie DNS (Domain Name System). In Programmen verwendet man für die Adressierung von Partneranwendungen sinnvollerweise symbolische Namen, die als Hostnamen bezeichnet werden.

Domain Name System (DNS)
Die Adressabbildung von symbolischen Hostadressen auf IP-Adressen erfolgt im Internet über ein weltweit verteiltes, massiv repliziertes Verzeichnissystem, das als DNS bezeichnet wird. Man nutzt also in der Regel symbolische Hostadressen wie zum Beispiel www.hm.edu. Über einen DNS-Dienst, Resolver genannt, kann man die dazugehörige IP-Adresse ermitteln und wendet sich dabei an den nächstgelegenen Name-Server. Das Herzstück von DNS bilden die sogenannten DNS-Root-Name-Server (kurz: Root-Name-Server). Diese stehen weltweit zur Verfügung, um eine DNS-Anfrage zu beantworten bzw. Informationen über die weitere Suche zu geben.

Es gibt derzeit weltweit 13 sogenannte Root-Name-Server (A bis M), von denen 10 in Nordamerika, einer in Stockholm, einer in London und einer in Tokio (M-Knoten) stehen. Da die Root-Name-Server vielfach das Ziel von Angriffen waren, wurde die Ausfallsicherheit durch die Nutzung von *Anycast* als Adressierungsart in den vergangenen Jahren nochmals wesentlich erhöht. Jeder Root-Name-Server ist mehrfach repliziert (Mandl et al. 2010). Anycast bedeutet, dass ein Request für eine Namensauflösung an mehrere Replikate gesendet wird, aber nur einer davon den Request bearbeitet.

3.1.4 TCP-Steuerinformation

Aus Sicht des TCP-Dienstnehmers ist die Nachrichtenübertragung jeweils ein Datenstrom (TCP-Stream von abgehenden und ankommenden Bytes. Die TCP-Instanzen nehmen den Datenstrom des lokalen Dienstnehmers, also des Sendeprozesses entgegen und erzeugen daraus TCP-Segmente (kurz: Segmente)). Ein Segment besteht aus einem mindestens 20 Byte langen TCP-Header, sofern das Nutzdatenteil leer ist.

Der in der Regel 20 Byte lange TCP-Header enthält die gesamte Steuerinformation, die TCP für den Verbindungsaufbau, den zuverlässigen Datentransfer und den Verbindungsabbau benötigt. Dazu können noch wahlweise bestimmte Optionen kommen, was dazu führt, dass der Header auch länger als 20 Byte sein kann. Die Headerlänge ist also variabel. Im Anschluss an den Header werden bei TCP-Data-PDUs die Daten übertragen. Abb. 3.3 zeigt den Aufbau des TCP-Headers.

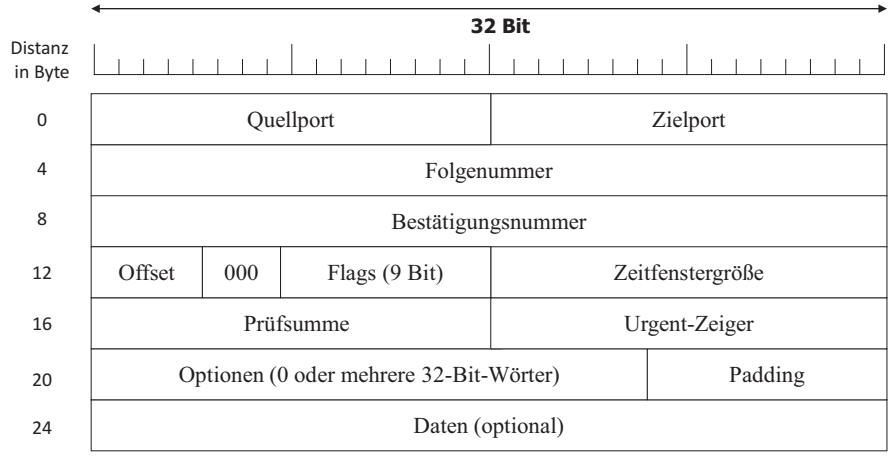


Abb. 3.3 TCP-Protokoll-Header

Im Einzelnen enthält der TCP-Header folgende Felder, die zum Teil auch in Beziehung miteinander stehen:

- *Quell- und Zielport*: Portnummer des Anwendungsprogramms des Senders (Quelle) und des Empfängers (Ziel).
- *Folgenummer*: Nächstes Byte innerhalb des TCP-Streams, das der Sender absendet.
- *Bestätigungsnummer*: Gibt das als nächstes erwartete Byte im TCP-Stream des Partners an und bestätigt damit den Empfang der vorhergehenden Bytes.
- *Offset*: Gibt die Länge des TCP-Headers in 32-Bit-Worten an. Dies ist nötig, da der Header aufgrund des variablen Optionenfeldes keine fixe Länge aufweist.
- *Reserviert (Res.):* Hat noch keine Verwendung.
- *Flags*: Steuerkennzeichen für diverse Aufgaben, die im Einzelnen noch diskutiert werden.
- *Zeitfenstergröße*: Erlaubt es einem Empfänger, in einer Bestätigungs-PDU seinem Partner den vorhandenen Pufferplatz in Bytes zum Empfang der Daten mitzuteilen (auch Window-Size genannt).
- *Prüfsumme*: Verifiziert das Gesamtpaket (TCP-Header + Nutzdaten) auf Basis eines einfachen für IP, UDP und TCP gleichermaßen beschriebenen Prüfsummenalgorithmus (RFCs 1071, 1141 und 1624). Er wird in Kap. 4 näher besprochen.
- *Urgent-Zeiger*: Beschreibt die Position (Byteversatz ab der aktuellen Folgenummer), an der dringliche Daten vorgefunden werden. Diese Daten werden vorrangig behandelt. Der Zeiger wird allerdings in der Praxis selten genutzt.
- *Optionen*: Optionale Angaben zum Aushandeln bestimmter Verbindungsparameter, die noch im Einzelnen diskutiert werden.
- *Padding*: Ergänzen des Headers mit dem Zeichen 0x00 bis auf die nächste Wortgrenze (ein Wort enthält vier Bytes), wenn das Optionsfeld kleiner als ein Vielfaches von vier Bytes ist.
- *Daten*: Nutzdatenlast, die bei reinen Steuersegmenten auch fehlen kann.

Die *Folgenummer* und die *Bestätigungsnummer* dienen der Flusskontrolle und der Synchronisation zwischen Sender und Empfänger bezüglich der übertragenen Daten. Die TCP-Flusskontrolle wird noch behandelt.

Die Flags im TCP-Header haben eine besondere Bedeutung. Diese sind in Tab. 3.1 beschrieben. Ursprünglich waren es sechs Flags, in neueren TCP-Implementierungen werden bereits acht Flags unterstützt. Hinzu kamen mit RFC 2481 die Flags CWR und ECN.[5]

Die Flags werden bei der Erläuterung der Protokollfunktionen noch intensiv betrachtet. SYN, ACK, FIN und RST sind für die Verbindungsverwaltung von Interesse. ACK wird für das Bestätigungsverfahren genutzt. Das PSH-Flag sollte eigentlich nur in Ausnahmefällen eingesetzt werden. Unsere Beobachtungen haben aber ergeben, dass es in heutigen Implementierungen sehr häufig gesetzt ist.

[5] Unterstützung für eine explizite Staukontrolle gibt es nun auch in der Vermittlungsschicht (in IP) über den Austausch von Stauinformationen zwischen Routern. Die Protokolle IP und TCP müssen für diese Aufgabe zusammenarbeiten.

Tab. 3.1 TCP-Flags

Flag (1 Bit)	Bedeutung
CWR	Dieses Flag wird für die explizite Staukontrolle (siehe Erläuterung zu ECN) genutzt.
ECE	Dieses Flag wird für die explizite Staukontrolle (siehe Erläuterung zu ECN) genutzt.
URG	Dieses Flag zeigt an, dass das Urgent-Feld im TCP-Header mit einem gültigen Wert belegt ist, also dringende Daten im TCP-Segment sind.
ACK	Dieses Flag gibt an, dass die Bestätigungsnummer im TCP-Header mit einem gültigen Wert belegt ist, d. h. es handelt sich um eine ACK-PDU (Bestätigungs-PDU).
PSH	Dieses Flag sagt aus, dass die Daten im TCP-Segment auf der Empfängerseite nicht zwischengepuffert werden dürfen, sondern sofort an den Empfängerprozess auszuliefern sind.
RST	Dieses Flag wird gesetzt, wenn ein ungültiges TCP-Segment empfangen wurde oder wenn ein Verbindungsaufbauwunsch nicht angenommen werden kann. Weiterhin wird es gesendet, wenn ein Anwendungsprozess abnormal beendet wurde, um dem Partner noch anzuzeigen, dass die Verbindung nicht aufrechterhalten werden kann.
SYN	Das Flag wird gesetzt, um den Verbindungsaufbau anzuzeigen (siehe TCP-Verbindungsaufbau).
FIN	Das Flag wird benutzt, um eine Verbindung abzubauen (siehe TCP-Verbindungsabbau).

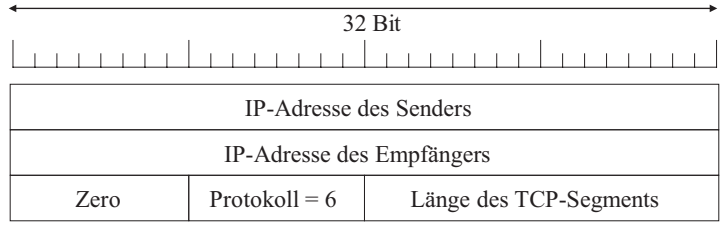


Abb. 3.4 TCP-Pseudoheader

Das Feld Prüfsumme dient der Überprüfung des TCP-Headers und der Nutzdaten. Alle 16-Bit-Wörter werden nach einem Verfahren, dass sowohl bei TCP als auch bei UDP gleichermaßen verwendet wird, addiert und dann in ein Einerkomplement transferiert. Wir verweisen zur Bildung der Prüfsumme daher auf das UDP-Kapitel. Es soll aber bereits jetzt erwähnt werden, das in die Prüfsummenberechnung neben dem TCP-Segment auch noch weitere Daten einbezogen werden, die als *Pseudoheader* bezeichnet werden. Der *Pseudoheader* enthält die IP-Adresse des Quell- und des Zielrechners, eine feste Protokollnummer (Nummer 6 = TCP) und die Länge der TCP-PDU in Bytes. Sinn und Zweck des Protokollheaders ist es, beim Empfänger fehlgeleitete PDUs zu erkennen. Dies kann die empfangende TCP-Instanz anhand der IP-Adresse prüfen. Informationen des Schicht-3-Protokolls werden in der Schicht 4 verwendet, was allerdings der reinen Kapselungslehre des Schichtenmodells widerspricht. Zudem kann die Angabe der Länge der TCP-PDU genutzt werden, um festzustellen, ob die ganze PDU beim Empfänger angekommen ist. Der Aufbau des Pseudoheaders ist in Abb. 3.4 skizziert.

3.2 Ende-zu-Ende-Verbindungsmanagement

3.2.1 TCP-Verbindungsaufbau

Beim Verbindungsaufbau erfolgt eine Synchronisation zwischen einem Partner, der aktiv den Verbindungsaufbauwunsch absetzt (aktiver Partner im Sinne des Verbindungsaufbaus), und einem Partner, der auf einen Verbindungsaufbauwunsch wartet (passiver Partner im Sinne des Verbindungsaufbaus). Während des Verbindungsaufbaus werden einige Parameter ausgetauscht bzw. auch ausgehandelt, die für die weitere Kommunikation als Basis dienen. Hierzu gehören die Größe der Flusskontrollfenster, die initialen Folgenummern für beide Partner und die Portnummern der Partner. TCP verwendet ein Dreiwege-Handshake-Protokoll für den Verbindungsaufbau, wie es vereinfacht in Abb. 3.5 dargestellt ist.

Der aktive Partner – als Instanz 1 bezeichnet – (im Sinne einer Client-/Server-Architektur ist dies typischerweise auch der Client) beginnt den Verbindungsaufbau mit einem Connect-Aufruf. Die zuständige TCP-Instanz richtet daraufhin lokal einen Verbindungskontext ein und ermittelt eine initiale Folgenummer, die sie im Feld *Folgenummer* mit einer Connect-Request-PDU an den Server sendet. Diese PDU ist durch ein gesetztes SYN-Flag markiert. Im TCP-Header werden weiterhin der adressierte Port und der eigene Port eingetragen.

Der passive Partner – als Instanz 2 bezeichnet – wartet auf einen Verbindungsaufbauwunsch, indem er an der Socket-Schnittstelle einen *Listen*-Aufruf tätigt. Die zuständige TCP-Instanz auf der passiven Seite baut ebenfalls einen Kontext auf und sendet eine Connect-Response-PDU, in der das SYN-Flag und das ACK-Flag auf 1 gesetzt sind. Weiterhin wird die Folgenummer des aktiven Partners dadurch bestätigt, dass sie um 1 erhöht und in das Feld „Bestätigungsnummer“ eingetragen wird. Dies bedeutet, dass das nächste gesendete Byte eine Folgenummer aufweisen muss, die der gesendeten Bestätigungsnummer des Partners entsprechen muss. Zudem berechnet die TCP-Instanz 2 vor dem Absenden der *Connect-Response-PDU* ebenfalls eine Folgenummer für die von ihr gesendeten Daten

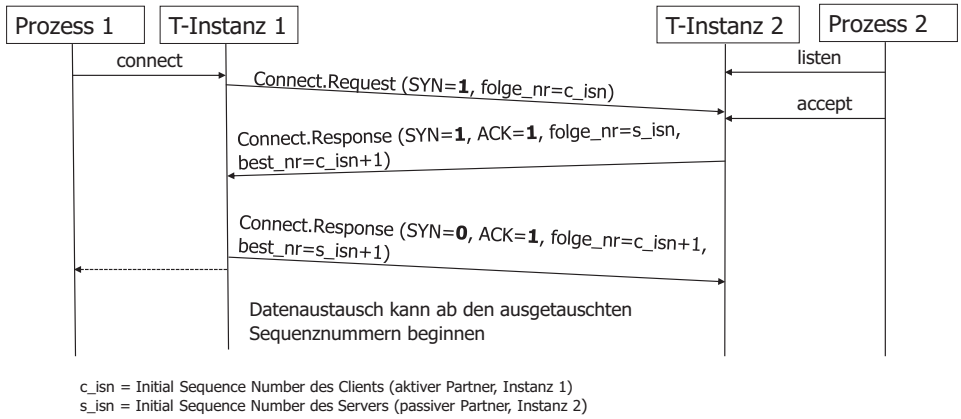


Abb. 3.5 Normaler Verbindungsaufbau in TCP

und nimmt sie in die PDU im Feld *Folgenummer* mit auf. Zu beachten ist auch, dass der passive Partner seine lokale genutzte Portnummer noch verändern kann.

Sobald der aktive Partner die ACK-PDU erhält, stellt er seinerseits eine Bestätigungs-PDU (ACK-PDU) zusammen, in der das SYN-Flag auf 0 und das ACK-Flag auf 1 gesetzt sind und im Feld *Bestätigungsnummer* die um 1 erhöhte Folgenummer des passiven Partners eingetragen wird. Die Bestätigung kann auch gleich im Piggypacking mit dem ersten anstehenden Datensegment gesendet werden.

Sollte der passive Partner auf den Verbindungswunsch nicht antworten, wird in der Regel ein erneuter Connection-Request abgesetzt. Dies wird je nach Konfigurierung der TCP-Instanz mehrmals versucht, wobei eine drei- bis fünfmalige Wiederholung häufig anzutreffen ist.

Bei jedem Datenaustausch werden im weiteren Verlauf der Kommunikation die Felder *Quell-* und *Zielport*, die *Folgenummer* und die *Bestätigungsnummer* sowie das Feld *Fenstergröße* gesendet. Der aktive Partner sendet in der *Connect-Request-PDU* seine Portnummer als Quellport und den adressierten T-SAP als Zielport im TCP-Header. Quell- und Zielport werden in jedem Segment aus Sicht des Senders belegt, dies gilt auch für die anschließende Datenübertragungsphase.

Beim Verbindungsaufbau sind Kollisionen möglich. Zwei Hosts können z. B. gleichzeitig versuchen, eine Verbindung zueinander aufzubauen. TCP muss dafür sorgen, dass in diesem Fall nur eine TCP-Verbindung mit gleichen Parametern (Zielport, Quellport) aufgebaut wird. Dies liegt daran, dass das Socket-Paar netzweit eindeutig sein muss.

Bei TCP kann nämlich das für Sequenznummern typische Problem auftreten, dass eine Verbindung abgebrochen und zufällig gleich wieder mit den gleichen Ports eine Verbindung aufgebaut wird. Gleichzeitig könnte noch eine Nachricht der alten Verbindung unterwegs sein. In diesem Fall könnte es in der neuen Verbindung zur Verwendung einer noch in der alten Verbindung genutzten Sequenznummer kommen. TCP hat zur Vermeidung dieses Problems einen Mechanismus eingebaut, der als *PAWS* bezeichnet und weiter unten noch diskutiert wird.

Abschließend soll der Vollständigkeit halber noch erwähnt werden, dass ein Verbindungsaufbauwunsch vom passiven Partner auch sofort abgewiesen werden kann, wenn z. B. kein Prozess im Listen-Zustand ist. In diesem Fall sendet die passive TCP-Instanz eine Reset-PDU (mit gesetztem RST-Flag), und man kann in der Regel davon ausgehen, dass die Kommunikationsanwendung nicht richtig initialisiert ist oder aber ein größeres Problem in der TCP-Instanz vorliegt.

Transaction TCP (T/TCP)

In den vergangenen Jahren wurden viele TCP-Erweiterungen vorgeschlagen. Eine davon war Transactional TCP (T/TCP). T/TCP war eine experimentelle Erweiterung von TCP und wurde in den RFCs 1379 und 1644 beschrieben. Ein wesentlicher Vorschlag von T/TCP war, dass man für aufeinanderfolgende Transaktionen von Request-/Ressponse-Folgen, wie dies etwa in HTTP der Fall ist, auf den Dreiwege-Handshake verzichtet. Diese Erweiterung wurde als *TCP Fast Open* bezeichnet. Zu diesem Zweck wurden sogenannte Connection Counts zum Zählen der Requests über TCP-Optionen (siehe TCP-Optionen) eingeführt. Passive Partner sollten beim Verbindungsaufbau anhand der Optionen im TCP-Header erkennen, ob es sich um eine Folge-Transaktion (Request) handelt und deswegen kein Dreiwege-Handshake erforderlich ist. Aus Sicherheitsgründen wurde T/TCP mittlerweile auf den Status „Historical“ gesetzt (siehe RFC 6247).

3.2.2 TCP-Verbindungsabbau

Sobald die Kommunikation, die theoretisch beliebig lange dauern kann, abgeschlossen ist, wird von einer Seite (egal von welcher) ein Verbindungsabbau initiiert. Dies geschieht an der Socket-Schnittstelle über einen *close*-Aufruf. Die initiierende Seite ist beim Verbindungsabbau der aktive Partner. Dieser startet eine etwas modifizierte Dreiwege-Handshake-, bei genauerer Betrachtung sogar eine Vierwege-Handshake-Synchronisation. Jede der beiden Verbindungsrichtungen der Vollduplex-Verbindung wird abgebaut, d. h. beide Seiten bauen ihre „Senderichtung“ ab. Der Ablauf ist in Abb. 3.6 skizziert und verläuft wie folgt:

- Der aktive Partner (T-Instanz 1) sendet zunächst ein TCP-Segment mit FIN-Flag = 1.
- Der passive Partner (T-Instanz 2) antwortet mit einem TCP-Segment, in dem das ACK-Flag auf 1 gesetzt ist.
- Wenn die Partner-Anwendung die *close*-Funktion an der Socketschnittstelle aufruft, sendet die TCP-Instanz auch ein TCP-Segment mit gesetztem FIN-und ACK-Flag.
- Der aktive Partner sendet abschließend ein TCP-Segment, in dem das ACK-Flag den Wert 1 hat.
- Die Folgenummern werden bei diesem Vorgang auf beiden Seiten um 1 erhöht.

Wir sehen also, dass für den normalen Verbindungsabbau vier Nachrichten gesendet werden, je zwei dienen dem Abbau der Senderichtung eines Partners. Wie wir bei der Betrachtung des TCP-Zustandsautomaten noch sehen werden, gibt es noch andere Möglichkeiten des Verbindungsabbaus. Auch eine abnormale Beendigung einer Verbindung ist möglich. In diesem Fall wird ein TCP-Segment mit RST-Bit = 1 gesendet und der Empfänger bricht die Verbindung sofort ab.

Wichtig ist, dass sich die Anwendungen richtig verhalten und auch auf beiden Seiten ordnungsgemäß die Verbindung über die Socket-Schnittstelle mit einem *close*-Aufruf beenden. Diesen Aspekt betrachten wir weiter unten noch genauer.

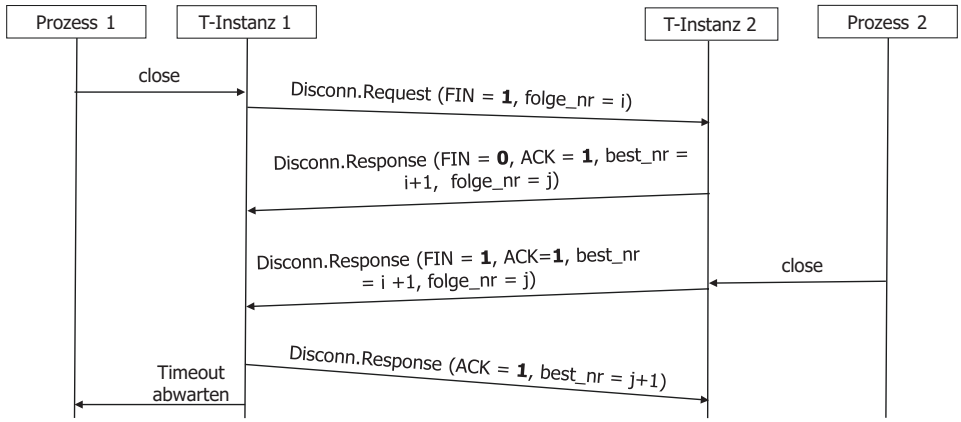


Abb. 3.6 TCP-Verbindungsabbau

Eine Besonderheit des Verbindungsabbaus beim aktiven Partner ist, dass er nach allen Bestätigungen des gegenüberliegenden Partners die Verbindung noch nicht abschließt, sondern erst noch in einen Wartezustand geht. Hier wartet er eine bestimmte Zeit auf Segmente, die eventuell noch im Netz unterwegs sind, damit nichts verloren geht. Nach Ablauf eines Timers wird die Verbindung beendet. Auch diesen Timer-Mechanismus werden wir weiter unten noch genauer diskutieren.

3.2.3 Zustände beim Verbindungsauf- und -abbau

Im RCF 793 und in vielen weiteren RFCs (siehe RFC 7414) ist TCP ausführlich spezifiziert. In RFC 793 wird auch ausführlich auf das Verbindungsmanagement eingegangen. Der Verbindungsaufbau und der Verbindungsabbau werden als Zustandsautomaten (Finite State Machine) modelliert. Eine vollständige Spezifikation der Datenübertragungsphase als endlicher Zustandsautomat wurde nicht vorgenommen.

Tab. 3.2 zeigt die möglichen Zustände, die beim Verbindungsaufbau und beim Verbindungsabbau sowohl auf der passiven als auf der aktiven Partnerseite eingenommen werden können. „Aktiv“ deutet hier jeweils auf den initiierenden Partner hin, also auf den Partner, der den Verbindungsauf- oder -abbau aktiv beginnt.

Tab. 3.2 TCP-Zustände beim Verbindungsauf- und -abbau

Flag (1 Bit)	Bedeutung
CLOSED	Keine Verbindung aktiv oder anstehend.
LISTEN	Der passive TCP-Partner wartet auch einen ankommenden Verbindungsaufbauwunsch.
SYN_RCVD	Ein Verbindungsaufbauwunsch ist beim passiven TCP-Partner angekommen und es wird noch auf eine Bestätigung gewartet.
SYN_SENT	Ein Verbindungsaufbau ist von Anwendungsprozess auf der aktiven Seite über einen Connect-Aufruf angekommen und wurde an den TCP-Partner kommuniziert.
ESTABLISHED	Eine Verbindung ist aufgebaut und befindet sich in der Datenübertragungsphase.
FIN_WAIT_1	Ein Anwendungsprozess möchte die Übertragung beenden, ein *close*-Aufruf wurde abgesetzt. Der Verbindungsabbauwunsch wurde an den TCP-Partner kommuniziert.
FIN_WAIT_2	Der passive TCP-Partner ist mit dem Verbindungsabbau einverstanden und hat dies bestätigt.
TIME_WAIT	Auf der passiven Seite wird gewartet, bis keine Segmente mehr über die Verbindung ankommen. Die Wartezeit ist durch den TIMED-WAIT-Timer begrenzt.
CLOSING	Beide TCP-Partner versuchen, die TCP-Verbindung gleichzeitig zu beenden.
CLOSE_WAIT	Auf der passiven TCP-Seite ist ein Verbindungsabbauwunsch angekommen. Es wird auf den *close*-Aufruf des Anwendungsprozesses gewartet.
LAST_ACK	Auf der passiven TCP-Seite wird gewartet, bis die letzte Bestätigung für den Verbindungsabbau ankommt.

Je nach aktueller Rolle eines TCP-Partners werden unterschiedliche Zustände durchlaufen. Einige der Zustände werden nur im aktiven Partner verwendet. Hierzu gehört der Zustand SYN_SENT beim aktiven Verbindungsaufbau sowie FIN_WAIT_1, FIN_WAIT_2 und TIME_WAIT beim passiven Verbindungsabbau. Andere Zustände werden nur im passiven TCP-Partner verwendet. Hierzu gehören z. B. die Zustände LISTEN und SYN_RCVD beim passiven Verbindungsaufbau sowie CLOSE_WAIT und LAST_ACK beim passiven Verbindungsabbau.

Zustandsautomaten in der Protokollspezifikation

Für die Spezifikation von Protokollen werden häufig deterministische endliche Automaten oder Finite State Machines (FSM) zur groben Beschreibung des Verhaltens von Protokollinstanzen verwendet. Deterministisch bedeutet, dass es bei einem Zustandsübergang immer einen definierten Folgezustand gibt. Automaten haben ihren Ursprung in der theoretischen Informatik (Herold et al. 2012).

Ein deterministischer endlicher Automat mit Ausgabe wird als Moore-Automat bezeichnet. Zustandsveränderungen hängen vom aktuellen Automatenzustand ab.

Beim Mealy-Automaten hängt die Ausgabe zusätzlich vom Input ab. Mealy-Automatien bilden auch die Grundlage von kommunizierenden endlichen Automaten, wie sie auch in der Protokollspezifikation bezeichnet werden. Ein derartiger Zustandsautomat lässt sich als 6-Tupel der Form

$$<S, I, O, T, s_o, F>$$

beschreiben, wobei gilt:

- S – endliche, nicht leere Menge von Zuständen
- I – endliche, nicht leere Menge von Eingaben
- O – endliche, nicht leere Menge von Ausgaben
- F – endliche, nicht leere Menge von Endzuständen, $F \subseteq S$
- $T \subseteq S \times (I \cup \{l\}) \times (O \cup \{l\}) \times S$ – eine Zustandsüberführungsfunktion, l bezeichnet eine leere Eingabe oder leere Ausgabe
- $s_o \in S$ – Initialzustand des Automaten

Eine Transition (Zustandsübergang) $t \in T$ ist definiert durch das Quadrupel

$$<s, i, o, s'>,$$

wobei

- $s \in S$ der aktuelle Zustand,
- $i \in (I \cup \{l\})$ eine Eingabe,
- $o \in (O \cup \{l\})$ eine zugehörige Ausgabe und
- $s' \in S$ der Folgezustand ist.

Abb. 3.7 stellt einen Zustandsübergang mit einer oder mehreren Ausgaben grafisch dar.

Häufig verwendet man zur Protokollspezifikation *erweiterte* endliche Automaten bzw. *Extended Finite State Machines* (*EFSM*) oder *Communicating Extended Finite State Machines* (*CEFSM*). Protokollinstanzen werden als endliche Automaten der Klasse CEFSM beschrieben. Als formale Beschreibungssprache, welche CEFSM nutzt, dient z. B. SDL (Specification and Description Language).

SDL nutzt für die Zerlegung eines Problems beim Entwurf sogenannte Blöcke. Eine grobe Darstellung der Blöcke und Schnittstellen des TCP-Protokollautomaten ist in Abb. 3.8 gegeben. Auf der Client-Seite kommuniziert eine Client-Anwendung lokal mit einer TCP-Instanz über ein Socket-Interface. Auf der Serverseite verhält es sich ähnlich. Die beiden TCP-Instanzen kommunizieren über TCP miteinander. Selbstverständlich sind in einer vollständigen Protokollspezifikation auch noch andere Ereignisquellen wie z. B. ein Zeitgeber für die Timerbearbeitung notwendig. Dies wird für unsere Betrachtung jedoch vernachlässigt.

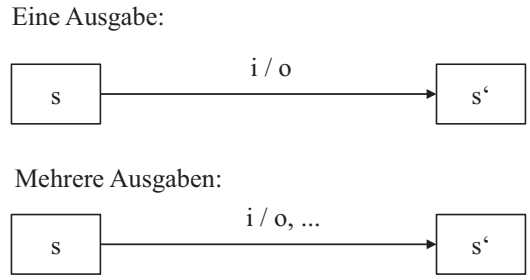


Abb. 3.7 Zustandsübergang von s nach s′ bei Eingabe i mit Ausgabe o

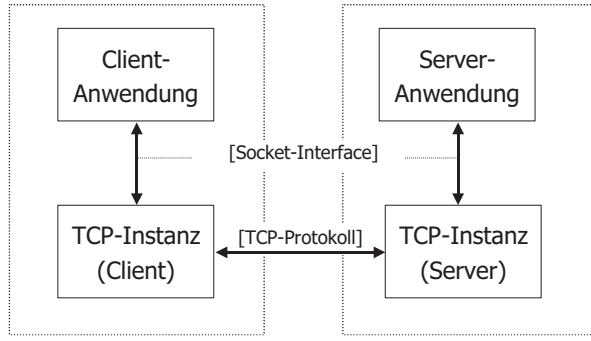


Abb. 3.8 Systemmodell aus Zustandsautomaten

3.2.4 Zustandsautomat des aktiven TCP-Partners

Betrachten wir zunächst den Zustandsautomaten des aktiven Partners, also der TCP-Instanz, die aktiv den Verbindungsaufbau und den Verbindungsabbau initiiert. Die aktive Rolle kann auch wechseln. Für jede einzelne TCP-Verbindung wird von der TCP-Instanz ein Zustandsautomat verwaltet. In Abb. 3.9 sind die Zustandsübergänge des aktiven Partners dargestellt. Zunächst befindet sich der aktive Partner im Zustand CLOSED. Um den Verbindungsaufbau zu aktivieren, ruft der Anwendungsprozess an der Socket API die Funktion *connect* auf, worauf die TCP-Instanz ein TCP-Segment sendet, in dem das SYN-Flag gesetzt ist. Als Folgezustand wird SYN_SENT eingenommen. Wenn ein Segment empfangen wird, in dem sowohl das SYN-Flag als auch das ACK-Flag gesetzt ist, wird als Antwort noch ein Segment mit gesetztem ACK-Flag gesendet und die Verbindung steht aus Sicht des aktiven Partners (ESTABLISHED).

Zum Einleiten des Verbindungsabbaus wird vom aktiven Anwendungsprozess ein *close*-Aufruf getätigt, der zum Senden eines Segments mit gesetztem FIN-Flag führt. Als Folgezustand wird FIN_WAIT_1 eingenommen. Bei normalem Verbindungsabbau wird anschließend vom Partner ein Segment mit gesetztem ACK-Flag empfangen und es wird der Zustand FIN_WAIT_2 eingenommen. Die Sendeseite ist damit abgebaut und es kann kein Segment mehr vom lokalen Anwendungsprozess gesendet werden. Nach erneutem Empfang eines Segments mit gesetztem FIN-Flag wird als Antwort ein Segment mit ACK-Flag gesendet und der Zustand TIME_WAIT eingenommen. Nach Ablauf eines

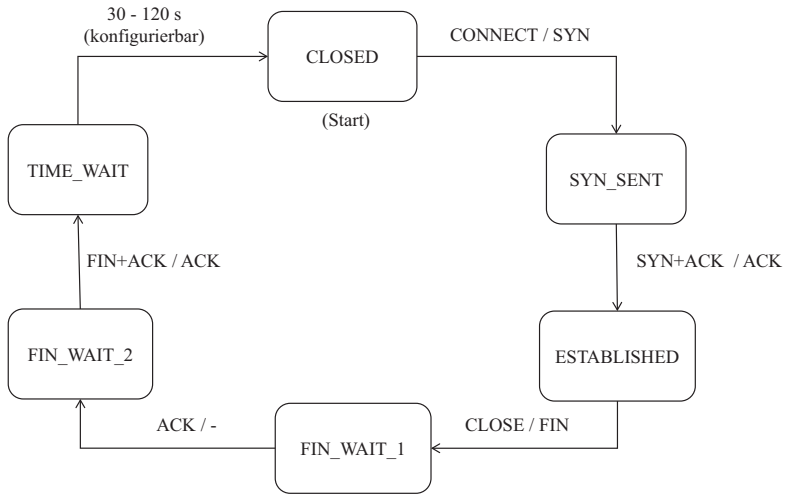


Abb. 3.9 TCP-Zustandsautomat des aktiven Partners

konfigurierten Timers wird die Verbindung dann endgültig abgebaut und der Zustand CLOSED eingenommen. CLOSED ist allerdings nur ein virtueller Zustand, weil in diesem Zustand der Verbindungskontext bereits gelöscht ist.

Neben dem eben beschriebenen Verbindungsauf- und -abbau sieht der TCP-Protokollautomat noch spezielle Varianten vor. So wird auch ein Verbindungsaufbau unterstützt, den beide Partner fast gleichzeitig vornehmen wollen, wobei dies bei Client/Server-Anwendungen, bei denen nur eine Seite auf Verbindungaufbauwünsche wartet, nicht vorkommt, wohl aber bei gleichberechtigten Kommunikationspartnern. Auch beim Verbindungsabbau kann es vorkommen, dass beide Partner gleichzeitig einen *close*-Aufruf absetzen. Auch diese Situation ist im TCP-Zustandsautomaten vorgesehen.

Der Zustand des Protokollautomaten wird im sogenannten TCB (Transmission Control Block) durch die TCP-Instanz verwaltet. Zustandsvariablen, die je TCP-Verbindung und Kommunikationsseite im TCB verwaltet werden, sind die Fenstergröße, Informationen zur Staukontrolle wie die aktuelle Größe des Überlastfensters, die verschiedenen Zeiger für die Fensterverwaltung, die verschiedenen Timer usw.

3.2.5 Zustandsautomat des passiven TCP-Partners

Auf der passiven Seite muss der Zustandsautomat synchron zum aktiven Partner gehalten werden. Dies geschieht über den spezifizierten Nachrichtenaustausch, der im Protokoll festgelegt ist. Der passive Verbindungsauf- und -abbau ist im Zustandsautomaten in Abb. 3.10 skizziert.

Die passive Seite startet eine TCP-Verbindung ebenfalls vom virtuellen Zustand CLOSED aus. Der passive Partner muss sich in die Lage versetzen, Verbindungsaufbauwünsche

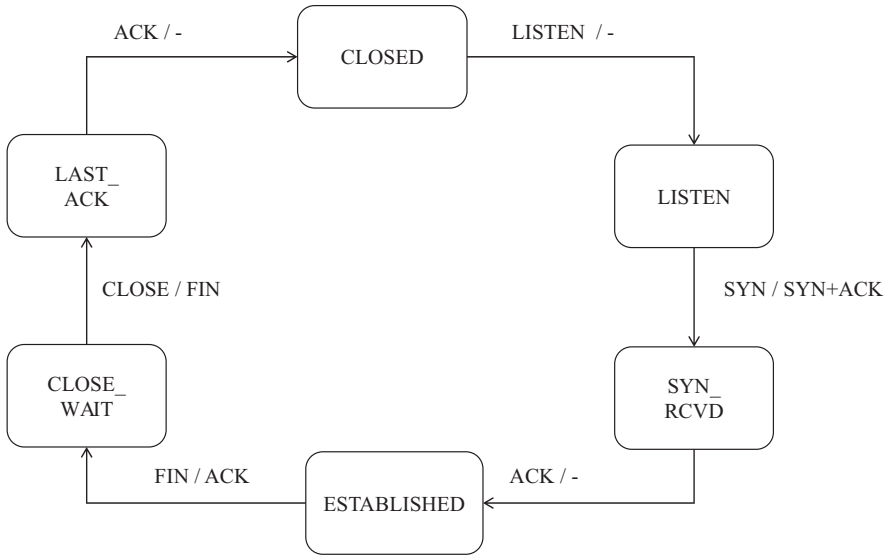


Abb. 3.10 TCP-Zustandsautomat des passiven Partners

entgegenzunehmen. Dies macht er, indem der Anwendungsprozess die Funktion *listen* an der SOCKET API aufruft, wodurch er in den Zustand LISTEN wechselt. In diesem Zustand kann er nun auf ankommende TCP-Segmente von potenziellen Partnern warten. Sobald von der TCP-Instanz ein Segment mit gesetztem SYN-Flag empfangen wird, wird es mit einem Segment beantwortet, das ebenfalls das SYN-Flag und auch das ACK.Flag gesetzt hat. Nach dem Absenden wird der Zustand SYN_RCVD eingenommen. Im dritten Schritt des Dreiwege-Handshakes empfängt die passive TCP-Instanz ein Segment mit gesetztem ACK-Flag, wonach die Verbindung steht (Zustand ESTABLISHED).

Zum Verbindungsabbau wartet der passive Partner auf ein Segment mit belegtem FIN-Flag. Hat er dieses empfangen, wechselt er den Zustand dieser Verbindung nach CLOSE_WAIT. Nun muss die Anwendung die *close*-Funtion aufrufen, sonst kann die Verbindung nicht ganz abgebaut werden. Dies ist Aufgabe der entsprechenden Anwendungsprotokoll-Instanz. Sobald der *close*-Aufruf abgesetzt wurde, wird noch ein weiteres Segment mit gesetztem FIN-Flag gesendet, womit der eigene Sendekanal geschlossen wird. Nun wird lokal kein *send*-Aufruf mehr akzeptiert. Es wird der Zustand LAST_ACK eingenommen, der erst verlassen wird, wenn nochmals ein letztes Segment mit gesetztem ACK-Flag empfangen wird. Danach ist die Verbindung abgebaut (Zustand CLOSED).

3.2.6 Zusammenspiel der Automaten im Detail

Das Zusammenspiel der aktiven und der passiven TCP-Instanz soll anhand des zusammengesetzten Zustandsautomaten weiter vertieft werden. Wichtig ist, dass für jede TCP-Verbindung zwei Zustandsautomaten zu verwalten sind, jede TCP-Instanz verwaltet einen für

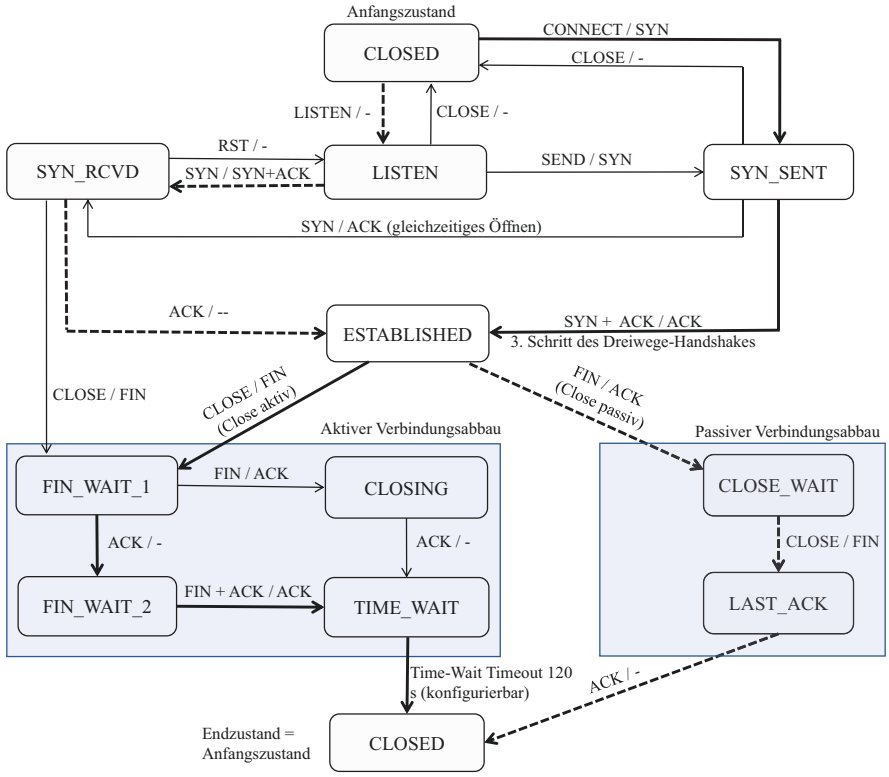


Abb. 3.11 Kompletter TCP-Protokollautomat nach RFC 793

ihre Kommunikationsseite. Der kombinierte Zustandsautomat ist in Abb. 3.11 gemäß RFC 793 skizziert. Die fetten, durchgezogenen Linien zeigen die normalen Zustandsübergänge im aktiven Partner, die fett gestrichelten Linien die normalen Zustandsübergänge im passiven Partner an. Die fein gezeichneten Linien deuten Spezialübergänge an. Die Beschriftung am Zustandsübergang gibt das auslösende Ereignis (z. B. eine ankommende PDU, der Aufruf einer Funktion des Anwendungsprozesses oder ein Timeout) an. In der Notation wird nach einem Schrägstrich die beim Zustandsübergang ausgeführte Aktion angegeben, z. B. ACK für das Senden eines Segments mit gesetztem ACK-Flag oder nur ein Minuszeichen, wenn nichts getan werden muss.

Es sind nur die Zustandsübergänge beim Verbindungsauf- und -abbau skizziert. Im aktiven Partner wird durch das Senden einer Connect-Request-PDU (SYN-Flag=1) in den Zustand SYN_SENT übergegangen. Dies ist der erste Schritt im Dreiwege-Handshake. Nach Empfang eines Segments mit gesetztem SYN- und ACK-Flag wird ebenfalls ein Segment mit gesetztem ACK-Flag gesendet und der Zustand auf ESTABLISHED gewechselt. Die Verbindung ist nun auf der aktiven Seite eingerichtet.

Der passive Partner (meist in der Serverrolle) befindet sich ursprünglich im Zustand LISTEN, d. h. er wartet auf ankommende Verbindungsaufbauwünsche. Trifft eine Connect-Request-PDU (SYN-Flag gesetzt) ein, antwortet der Server mit einem Segment, in dem SYN- und ACK-Flag gesetzt sind und geht in den Zustand SYN_RCVD. Nach dem erneuten Empfang eines Segmens mit ACK-Flag=1 wird der Zustand auf ESTABLISHED gestellt. Damit ist auch die Verbindung auf der passiven Seite eingerichtet, und die Datenübertragung kann in diesem Zustand beginnen. Der Zustand ESTABLISHED wird nur im Fehlerfall (im vereinfachten Zustandsautomaten nicht dargestellt) oder beim Beenden der Verbindung verlassen.

Der Verbindungsabbau ist noch etwas komplizierter. Ein aktiver Partner (aktiv im Sinne des Verbindungsabbaus) muss den Verbindungsabbau initiieren. Der Anwendungsprozess leitet den Verbindungsabbau durch Aufruf der *close*-Funktion ein. Im Normalfall geht der aktive Partner nach dem Senden einer Disconnect-Request-PDU (FIN-Flag=1) in den Zustand FIN_WAIT_1. Schon in diesem Zustand wird keine weitere Data-PDU mehr gesendet, es dürfen aber noch TCP-Segmente empfangen werden.

Der passive Partner empfängt eine Disconnet-Request-PDU, bestätigt diese und meldet der Anwendung den gewünschten Verbindungsabbau. Die Anwendung hat nun Zeit, auf den Verbindungsabbau zu reagieren und bestätigt den Verbindungsabbau mit der *close*-Funktion. TCP sendet daraufhin auch an den aktiven Partner eine Disconnect-Request-PDU und wechselt in den Zustand LAST_ACK, um auf die letzte Bestätigung zu warten. Der aktive Partner wechselt schon nach der erhaltenen Bestätigung in den Zustand FIN_WAIT_2 und wartet auf den Verbindungsabbauwunsch. Nach dem Erhalt der Disconnect-Request-PDU und dem Senden einer erneuten ACK-PDU wird im aktiven Partner in den Zustand TIME_WAIT übergegangen. Erst nach Ablauf des Timers wird der Zustand CLOSED eingenommen, und damit ist die Verbindung beendet. Der passive Partner wechselt sofort nach dem Empfang der letzten Bestätigung in den Zustand CLOSED.

In Abb. 3.12 sind die Zustandsübergänge beim Verbindungsabbau in Verbindung mit dem Nachrichtenaustausch skizziert. Der Grund für den Zustand *TIME_WAIT* ist – wie schon weiter oben angesprochen – darin zu sehen, dass noch zur Sicherheit so lange gewartet werden soll, bis tatsächlich kein Segment mehr unterwegs sein kann. In diesem Zustand wird der Time-Wait Timer aufgezogen, und zwar ursprünglich auf die doppelte Paketlebensdauer. Dieser Wert ist aber meist in Betriebssystemen konfigurierbar bzw. wie in Linux per Konstante fest eingestellt.

Beim Verbindungsabbau werden also bei normalem Ablauf insgesamt vier Nachrichten ausgetauscht. Laut Zustandsautomat gibt es weitere Varianten des Verbindungsabbaus. Eine Variante ist in Abb. 3.13 dargestellt. Beide Seiten beenden zeitgleich mit einem *close*-Aufruf ihre Verbindung. In diesem Fall wird im aktiven und im passiven Partner der Zustand CLOSING durchlaufen.

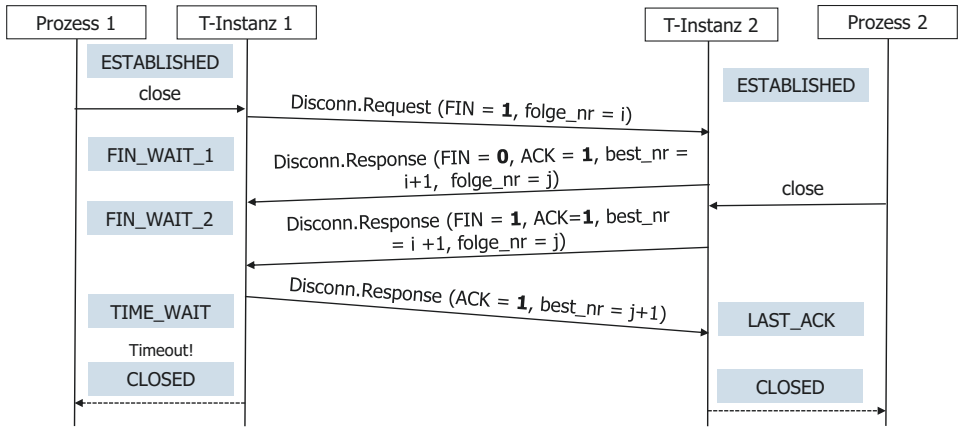


Abb. 3.12 TCP-Verbindungsabbau mit Zustandsübergängen

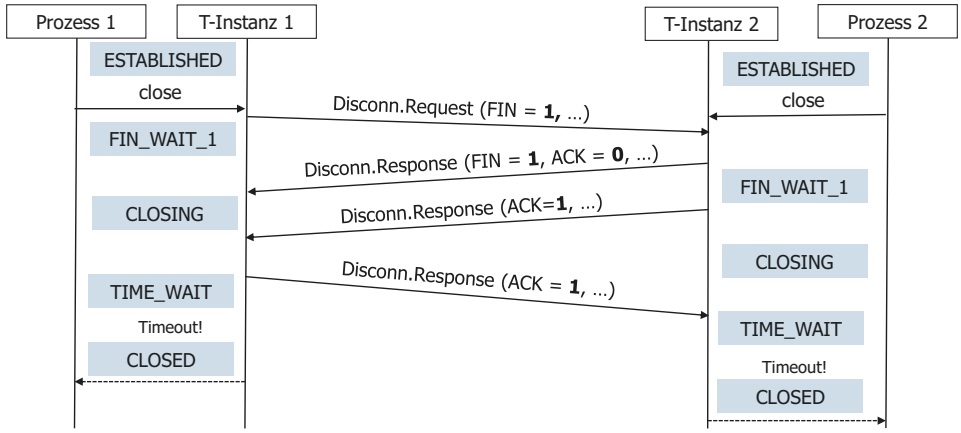


Abb. 3.13 Weitere Variante für TCP-Verbindungsabbau

Die eigentliche Datenübertragung mit den vielen zu beachtenden Protokollregeln für eine gesicherte Übertragung erfolgt im Zustand ESTABLISHED, worauf wir im Weiteren eingehen werden.

netstat, ss und TCP Viewer

Mit dem Kommando `netstat` kann man die aktuellen Kommunikationsbeziehungen und Portbelegungen für UDP und TCP zeilenorientiert anzeigen lassen. Auch die Ports eines Endsystems, die im Zustand LISTEN sind, kann man ermitteln. Ruft man das Kommando mit den Optionen `netstat -p tcp` auf, werden nur TCP-Verbindungen bzw. Portbelegungen angezeigt. Das Kommando ist auf gängigen Betriebssystemen wie Windows, Unix und Linux verfügbar. Unter Linux kann auch das Kommando *ss –tcp* verwendet werden, um aktuelle TCP-Verbindungen und deren Zustände eines Rechners in Erfahrung zu bringen.

Es gibt aber auch etwas komfortablere Tools zur Anzeige aktueller Kommunikationsbeziehungen. Beispielsweise kann man sich für Windows das frei verfügbare Tool *TcpView*[6] herunterladen. Das Tool verfügt über eine komfortable Oberfläche und zeigt sogar an, welche Prozesse an welchen Verbindungen beteiligt sind, was die Fehlersuche erleichtert.

3.3 Datenübertragungsphase

3.3.1 Normaler Ablauf

In diesem Abschnitt behandeln wir die fehlerfreie Datenübertragung in TCP. Der Ablauf einer Übertragung eines Datensegments sieht, wie in Abb. 3.14 dargestellt, bei TCP im Normalfall wie folgt aus:

- Der Sender stellt ein Segment zusammen, sendet es und zieht anschließend für jedes einzelne Segment einen Retransmission Timer zur Zeitüberwachung auf.
- Die Zeit für den Retransmission Timer wird dynamisch anhand der aktuellen RTT (Round Trip Time) ermittelt. Dies ist die Zeit, die eine Ende-zu-Ende-Übertragung eines Segments einschließlich der Bestätigung benötigt.
- Im Normalfall erhält der Empfänger das Segment und bestätigt es in einem kumulativen Verfahren. Man bezeichnet dies als kumulative Quittierung. Damit wird versucht, die Netzlast zu reduzieren. Immer dann, wenn es möglich ist, gleich mehrere Segmente zu bestätigen, wird dies gemacht. Es wird sogar eine Verzögerung der Bestätigung unterstützt, um noch auf weitere Segmente zu warten.
- Der Sender erhält schließlich ein Segment mit gesetztem ACK-Flag und Bestätigungsnummer und entfernt daraufhin den noch laufenden Timer vor seinem Timeout.

Wie bereits beim Verbindungsmanagement zeichnet sich auch hier eine ACK-PDU dadurch aus, dass das ACK-Flag gesetzt ist. In dem Segment ist auch die Bestätigungsnummer gültig und zeigt auf das als nächstes erwartete Octet im TCP-Stream. In Abb. 3.14 wird die Übertragung zweier Segmente dargestellt, die ordnungsgemäß (in diesem Falle jedes für sich) bestätigt werden. Bei Ankunft der Bestätigung wird der Timer in der Instanz 1 gelöscht, und die Übertragung ist damit abgeschlossen.

Für die Garantie der Reihenfolge und der Vollständigkeit wird bei TCP der Einsatz von Sequenznummern unterstützt, die im Sinne des Datenstrom-Konzepts auf einzelnen Bytes, nicht auf TCP-Segmenten basieren. Im TCP-Header wird hierfür das Feld *Folgenummer* verwendet. Die initiale Sequenznummer wird beim Verbindungsaufbau für beide Kommunikationsrichtungen festgelegt. Sie enthält jeweils die laufende Nummer des als nächstes erwarteten Bytes im Stream der Verbindung.

[6] Download aus www.sysinternals.com. Zugegriffen am 18.07.2017.

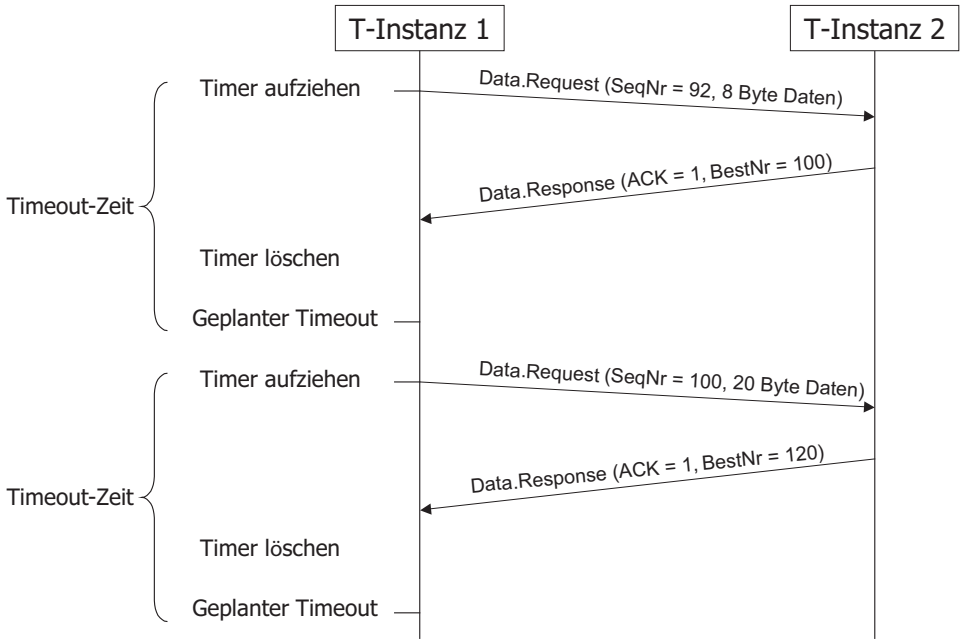


Abb. 3.14 Normale Übertragung eines TCP-Segments

Die Bestätigung der ordnungsgemäßen Ankunft eines Segments wird in der ACK-PDU über das Feld *Bestätigungsnummer* durchgeführt. In diesem Feld steht jeweils die als nächstes erwartete Sequenznummer des Partners, womit der Empfang aller vorhergehenden Byte im Stream bestätigt wird. Eine Bestätigung muss von der empfangenden TCP-Instanz nicht unbedingt sofort gesendet werden, sofern noch Platz im Empfangspuffer ist. Hier besteht eine gewisse Implementierungsfreiheit. Eine Ausnahme bilden sogenannte „Urgent-Daten“, die immer gesendet werden können, auch wenn der Empfangspuffer voll ist, wobei diese allerdings in der Regel nicht häufig verwendet werden.

Wie bereits angedeutet, werden TCP-Segmente wenn möglich kumulativ bestätigt. Dies hat den positiven Nebeneffekt, dass gegebenenfalls auch verloren gegangene ACK-PDUs durch folgende ACK-PDUs erledigt werden. Dies ist z. B. im Sequenzdiagramm aus Abb. 3.15 deutlich zu sehen. In diesem Szenario geht eine ACK-PDU verloren und wird durch die folgende kompensiert.

In den RFCs 1122 und 2581 werden einige Implementierungs-Empfehlungen für die Quittierungsmechanismen von TCP-Segmenten gegeben. Einige Beispiele sollen hierzu erwähnt werden:

- Der Empfänger eines TCP-Segments sollte maximal 500 ms auf weitere Segmente warten, um diese kumulativ zu bestätigen.
- Kommt ein TCP-Segment außerhalb der Reihe mit einer Sequenznummer bei einer TCP-Instanz an, die höher ist als die erwartete, so soll eine erneute ACK-PDU als

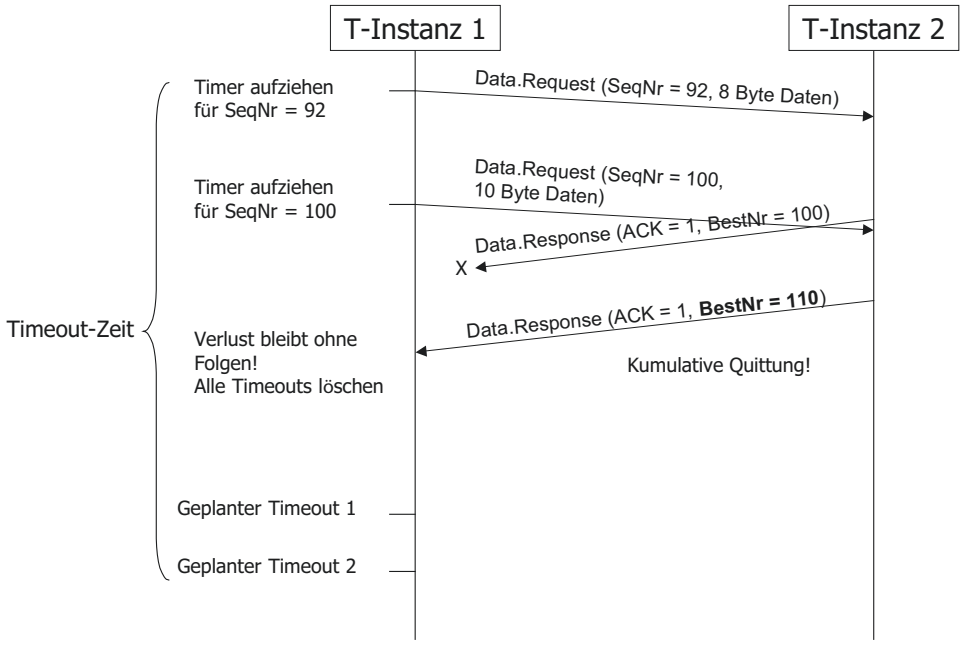


Abb. 3.15 Kumulative Quittierung in TCP

Duplikat der letzten ACK-PDU gesendet werden. Damit weiß der Sender sofort, dass er die fehlenden TCP-Segmente erneut senden muss.

- Kommt ein TCP-Segment bei einer TCP-Instanz an, das eine Lücke schließt, so wird dies sofort und ohne Verzögerung bestätigt, damit die sendende TCP-Instanz nicht behindert wird.

3.3.2 Timerüberwachung

Für jedes einzelne Segment wird gleich beim Absenden ein Retransmission Timer aufgezogen. Läuft im Sender z. B. der Retransmission Timer ab, bevor eine Bestätigung angekommen ist, wird das Segment noch einmal gesendet. Der genaue Grund, warum das Segment nicht bestätigt wurde, ist dem Sender nicht bekannt. Es kann sein, dass das Ursprungssegment verloren gegangen ist, es kann aber auch sein, dass die ACK-PDU nicht angekommen ist. Der Retransmission Timer wird im Verlustfall angepasst, aber das wird in diesem Kapitel weiter unten noch genauer diskutiert.

In Abb. 3.16 ist ein Szenario skizziert, in dem der Retransmission Timer vor Ankunft der ACK-PDU in Instanz 1 abläuft. Die ACK-PDU geht im Netz verloren. Das Segment wird erneut übertragen, und die zweite Übertragung wird von Instanz 2 bestätigt. Die zweite Bestätigung trifft rechtzeitig ein.

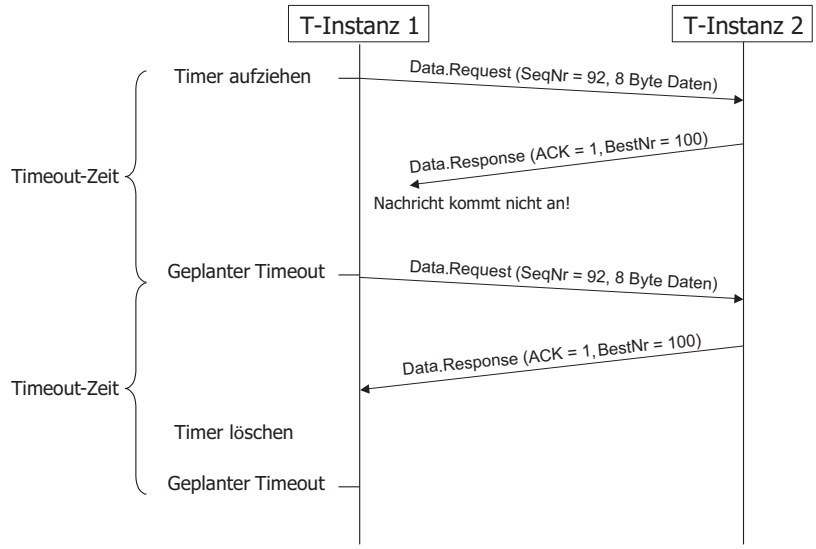


Abb. 3.16 Timerablauf bei der Übertragung eines TCP-Segments

Es sei hier nochmals darauf hingewiesen, dass ein Segment bei der Übertragung im Netz durchaus verloren gehen kann. Immerhin kann es sein, dass ein Router, durch den das Segment weitergeleitet werden soll, gerade überlastet ist. Außerdem kann eine Ende-zu-Ende-Kommunikation zwischen zwei Anwendungsprozessen über mehrere Router ablaufen. TCP stellt aber sicher, dass ein verloren gegangenes Segment noch einmal übertragen wird.

3.3.3 Implizites Not-Acknowlegde (NAK)

Um den Kommunikationsaufwand zu reduzieren, wird in TCP das sogenannte *implizite NAK* (Not-Acknowledge) unterstützt. Wenn der Empfänger ein Segment vermisst, so sendet er die vorhergehende Bestätigung noch einmal. Erhält der Sender viermal die gleiche Bestätigung (also drei ACK-Duplikat-PDUs),[7] so nimmt er an, dass die dem zuletzt bestätigten Segment folgenden Segmente nicht angekommen sind. Er sendet daraufhin diese vor Ablauf des Timers erneut. Dieser Ablauf ist in Abb. 3.17 dargestellt. Nach jedem Empfang eines weiteren TCP-Segments sendet die T-Instanz 2 erneut eine ACK-Duplikat-PDU.

Das dreimalige Empfangen der gleichen Bestätigung kann – je nach TCP-Implementierung – auch Auswirkungen auf die Staukontrolle haben. Dieser Vorgang wird als *Fast Retransmit* bezeichnet (siehe hierzu auch den mit dem *Fast Retransmit* gemeinsam behandelten Mechanismus, der als *Fast-Recovery* bezeichnet wird).

[7] Siehe hierzu RFC 2581, früher waren es laut RFC 2001 nur zwei ACK-Duplikate.

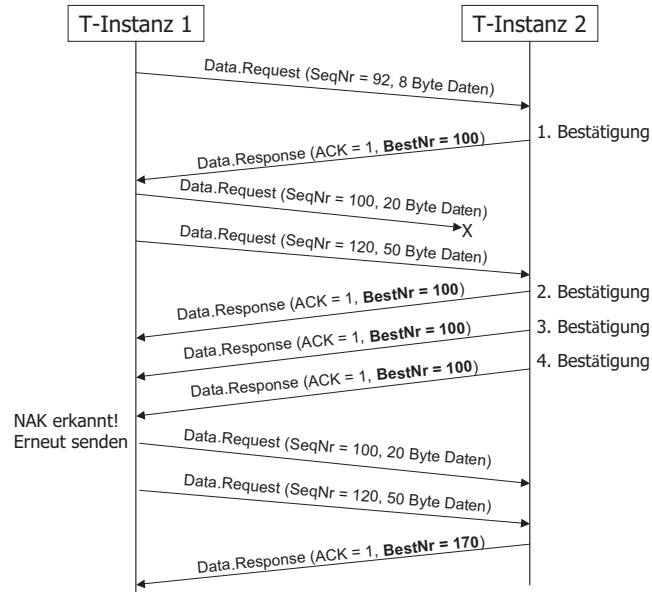


Abb. 3.17 Empfänger sendet implizites NAK

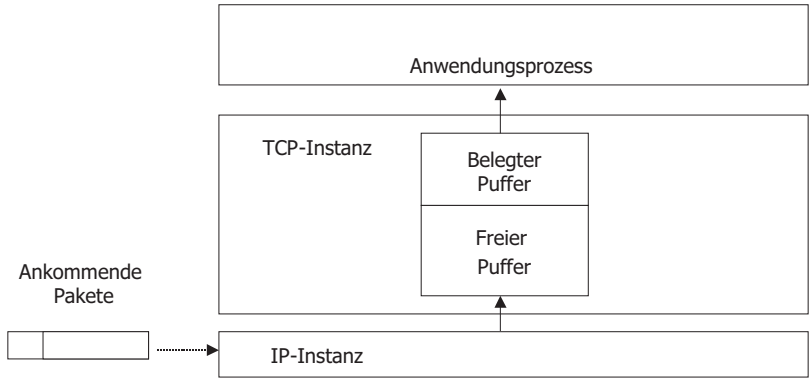


Abb. 3.18 Empfangspuffer für TCP-Verbindungen

3.3.4 Flusskontrolle

Bei der Einrichtung einer TCP-Verbindung sorgen die TCP-Instanzen beider Kommunikationspartner dafür, dass für die Verbindung jeweils Pufferbereiche für abzusendende und zu empfangende Daten eingerichtet werden. Aufgabe der Flusskontrolle ist es sicherzustellen, dass der Empfangspuffer immer ausreichend groß ist, um noch weitere TCP-Segmente aufnehmen zu können. In Abb. 3.18 ist ein Empfangspuffer innerhalb einer TCP-Instanz skizziert. Die TCP-Instanz muss über dessen Befüllungsgrad Buch führen.

Es ist eine der wichtigsten Aufgaben von TCP, eine optimale Fenstergröße für Transportverbindungen bereitzustellen.

TCP verwendet für die Flusskontrolle einen Sliding-Window-Mechanismus. Dieser erlaubt die Übertragung von mehreren TCP-Segmenten, bevor eine Bestätigung eintrifft, sofern die Übertragung die vereinbarte Fenstergröße nicht überschreitet.

Bei TCP funktioniert Sliding-Window auf der Basis von Octets (Bytes), worin sich auch der Streams-Gedanke widerspiegelt. Die Octets eines Streams sind sequenziell nummeriert. Die Flusskontrolle wird beiderseits durch den Empfänger gesteuert, der dem jeweiligen Partner mitteilt, wie viel freier Platz noch in seinem Empfangspuffer ist. Der Partner darf nicht mehr Daten senden und muss das Senden eines Segments bei Bedarf verzögern.

Zur Fensterkontrolle wird im TCP-Header das Feld *Zeitfenstergröße* verwendet. In jeder ACK-PDU sendet eine TCP-Instanz in diesem Feld die Anzahl an Bytes, die der Partner aktuell senden darf, ohne dass der Empfangspuffer überläuft. Wenn der Empfänger eine Nachricht mit einem Wert von 0 im Feld *Zeitfenstergröße* sendet, so bedeutet dies für den Sender, dass er sofort mit dem Senden aufhören muss. Erst wenn der Empfänger wieder ein Zeitfenster > 0 sendet, darf der Sender erneut Nachrichten senden.

In Abb. 3.19 ist dies beispielhaft skizziert. Anfänglich ist der Empfangspuffer auf der Empfängerseite leer. In diesem Beispiel ist der Puffer 4 KiB groß. In der TCP-Instanz auf der Empfängerseite läuft der Puffer voll, da der Anwendungsprozess die Daten vermutlich nicht oder zu langsam zur Verarbeitung abholt. Dies kann viele Gründe haben. Beispielsweise kann der Host zu stark ausgelastet sein oder ein Anwendungsprozess wartet auf ein Ereignis. Der Sender wird dadurch blockiert und wartet, bis der Empfänger wieder freien Pufferplatz meldet, indem er eine zusätzliche ACK-PDU absendet, in der im Feld „Fenstergröße“ (WIN) 2 KiB angegeben werden.

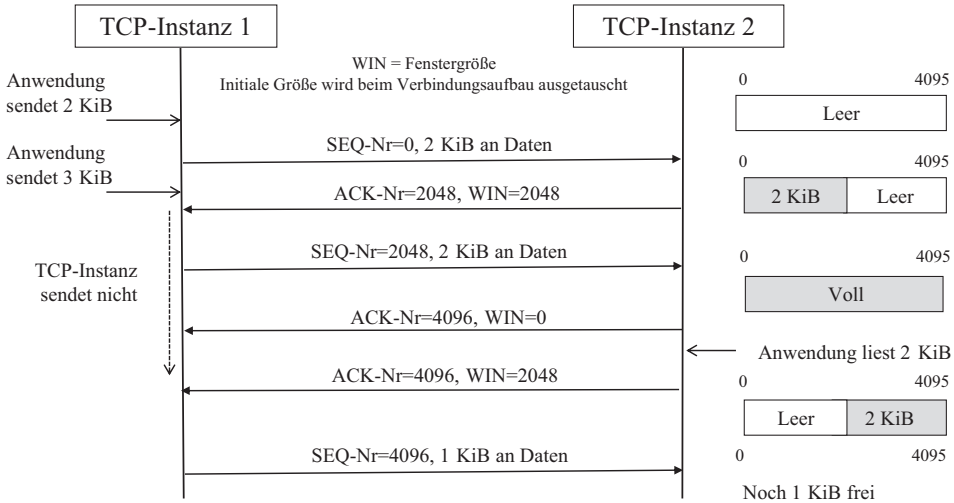


Abb. 3.19 Sliding-Window-Mechanismus bei TCP nach (Tanenbaum und Wetherall 2011)

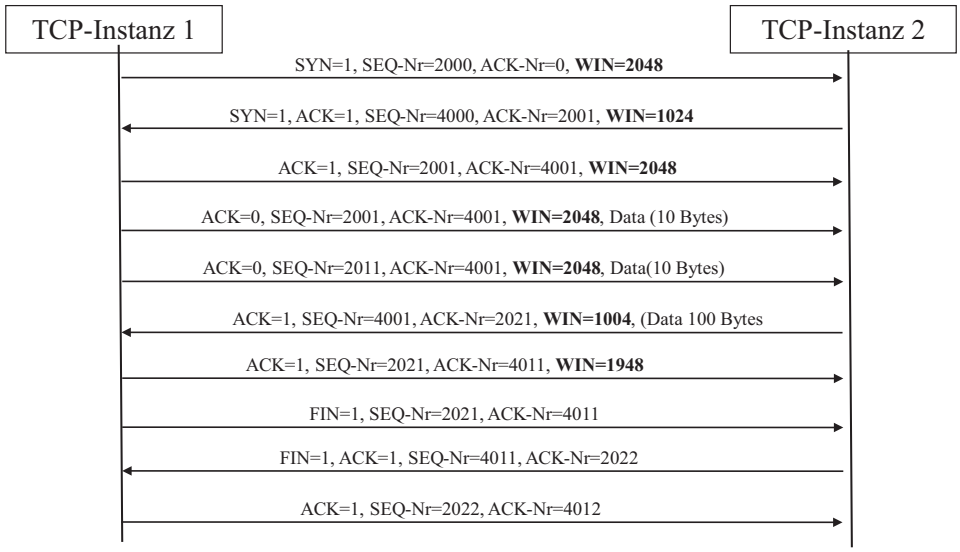


Abb. 3.20 Beispielablauf für Sliding-Window-Mechanismus bei TCP

Die Flusskontrollmechanismen wurden mehrmals optimiert, und es gibt in der TCP-Dokumentation eine Reihe von RFCs mit Vorschlägen für Optimierungs-Algorithmen. Einige Besonderheiten, die auch ausführlich in der angegebenen Literatur erläutert sind, werden im Weiteren noch diskutiert.

In Abb. 3.20 ist ein Szenario dargestellt, in dem zwei TCP-Instanzen eine Verbindung aufbauen, sich dann Daten-PDUs senden und danach die Verbindung wieder abbauen. Da die Anwendungsprozesse keine Daten auslesen (kein *receive*-Aufruf auf beiden Seiten), nimmt die initiale Fenstergröße (siehe WIN-Parameter) bei beiden ab.

Die initiale Fenstergröße für TCP wird im RFC 3390 (Increasing TCP's Initial Window) vorgegeben. Die Berechnung ergibt sich aus folgender Formel (gemessen in Bytes):

$$Initale\ Fenstergröße = min\left(4 * MSS, max\left(2 * MSS, 4380\right)\right)$$

3.3.5 Nagle- und Clark-Algorithmus

Es gibt Anwendungen, die oft nur sehr kleine Nachrichten versenden. Zur Leistungsoptimierung versucht TCP, möglichst wenige kleine Nachrichten zu senden, da diese schlecht für die Netzauslastung sind. Kleine Nachrichten werden also bei Bedarf zusammengefasst. Ein Algorithmus, der sich bei TCP durchgesetzt hat, ist der *Nagle-Algorithmus* (RFC 896 und RFC 1122), der in allen TCP-Implementierungen verwendet wird. Er funktioniert nach folgendem Prinzip:

- Wenn vom Anwendungsprozess an der Socket-Schnittstelle die Daten im Stream Byte für Byte ankommen, wird zunächst nur das erste Byte gesendet und die restlichen werden im Sendepuffer gesammelt.
- Danach werden Daten gesammelt, bis die Größe der MSS erreicht ist, und dann wird erst wieder ein Segment gesendet.
- Wenn allerdings ein Segment mit gesetztem ACK-Flag empfangen wird, wird der aktuelle Inhalt des Sendepuffers sofort gesendet.
- Anschließend wird erneut gesammelt und so geht es weiter.

Der Nagle-Algorithmus kann etwas verfeinert im Pseudocode wie folgt notiert werden:

```
Begin NagleAlgorithmus
       if (neue Daten zum Senden vorhanden){
            if (window size >= MSS) and
                    (verfügbare Datenlänge >= MSS){
                Sende ein komplettes Segment mit Länge = MSS;
            } else {
                if (noch Segmente nicht bestätigt) {
                            Lege Daten in den Sendepuffer so lange bis
                            ein Segment mit ACK-Flag empfangen wird;
                } else {
                             Sende Daten sofort;
                }
            }
       }
End
```

Die Vorgehensweise nach Nagle ist allerdings nicht immer optimal und insbesondere schlecht bei Anwendungen mit direktem Echo der Dialogeingaben. wie etwa bei interaktiven Sitzungsprotokollen wie Remote Login (rlogin) oder Secure Shell (ssh). Bei diesen Protokollen muss für jede Tastatureingabe eine Übertragung zum Partnerrechner, auf dem die eingegebenen Kommandos auszuführen sind, stattfinden. Auch für Echtzeit-Anwendungen, die schnelle Reaktionen erfordern, ist Nagle nur bedingt geeignet.

Es ist also manchmal in Abhängigkeit vom Anwendungstyp sinnvoll, den Nagle-Algorithmus auszuschalten. Anwendungsspezifisch lässt sich daher der Nagle-Algorithmus unter POSIX-kompatiblen Betriebssystemen mit der Socket-Option TCP_NODELAY (siehe TCP-Optionen weiter unten) abschalten (IEEE POSIX 2016). In der Praxis wird das zum Beispiel bei interaktiven Sitzungsprotokollen wie *ssh* getan, um die Reaktionszeit der Gegenseite auf Tastatureingaben oder bei Bildschirmausgaben zu verkürzen.

Ein anderes Problem ist als *Silly-Window-Syndrom* bekannt. Es kommt vor, wenn ein empfangender Anwendungsprozess die Daten byteweise ausliest, der Partner aber größere Segmente senden möchte. In diesem Fall wird der Empfangspuffer immer um ein Byte

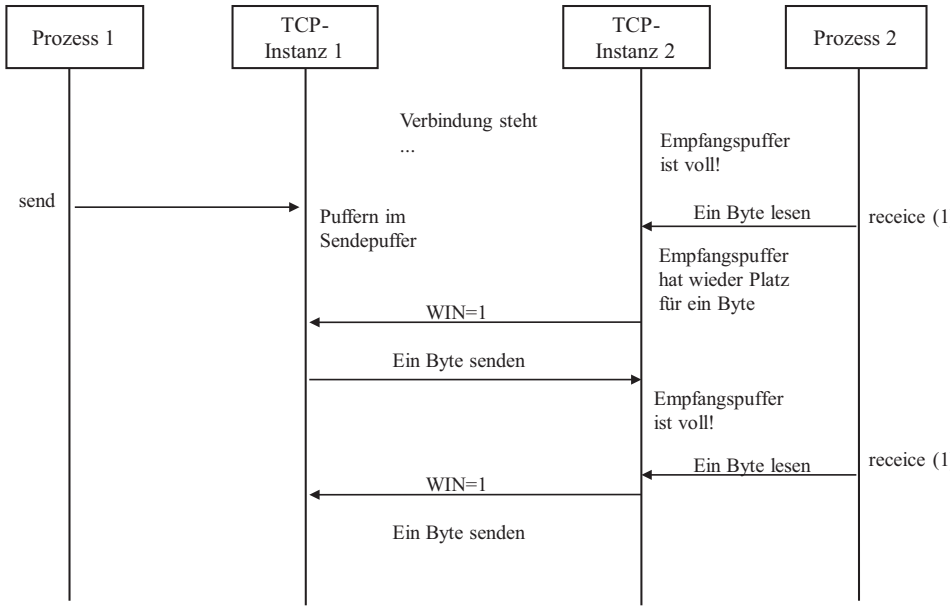


Abb. 3.21 Silly-Window-Syndrom

geleert, und die TCP-Instanz beim Empfänger sendet daraufhin eine ACK-PDU mit dem Hinweis, dass wieder ein Byte übertragen werden kann. In Abb. 3.21 wird das Silly-Window-Syndrom skizziert.

Dies verhindert *Clarks* Lösung, indem das Senden einer ACK-PDU bis zu einer vernünftigen Fenstergröße zurückgehalten wird. Clarks Lösung verhindert, dass die Sende-Instanz ständig kleine Segmente sendet, die der Empfängerprozess sehr langsam ausliest. Der Algorithmus funktioniert prinzipiell wie folgt:

- Die Empfangsinstanz sendet nach dem Empfang eines kleinen Segments eine ACK-PDU mit einer Fenstergröße von 0.
- Es wird so lange verzögert, bis der Empfänger eine Segmentlänge (MSS) gelesen hat oder der Empfangspuffer halb leer ist. Dann erst wird eine ACK-PDU mit normaler Fenstergrößen-Angabe gesendet.

Die Algorithmen von Nagle und Clark ergänzen sich in einer konkreten TCP-Implementierung.

3.4 TCP-Protokolloptionen

Die Optionen wurden anfangs relativ selten verwendet, aber mittlerweile werden diese zum Zeitpunkt des Verbindungsaufbaus üblicherweise verwendet. Sie werden an das Ende des TCP-Headers angefügt. Der Aufbau der Optionen ist variabel. Jede zulässige Option

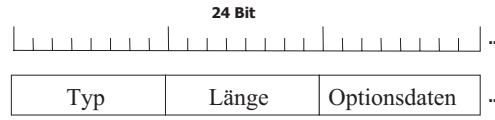


Abb. 3.22 Grundlegender Aufbau des TCP-Optionsfeldes

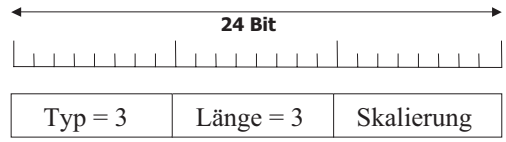


Abb. 3.23 Aufbau der WSOPT-Option

ist mit einer Typangabe, einem Längenfeld und den eigentlichen Optionsdaten gekennzeichnet (siehe Abb. 3.22).

Abb. 3.23 zeigt ein Beispiel für die Option *WSOpt* zur Skalierung der Fenstergröße. Als Typ wird die Zahl 3 verwendet und die Gesamtlänge der Option beträgt drei Byte.

Die TCP-Optionen werden in den entsprechenden RFCs mit Abkürzungen bezeichnet. Zu den Optionen gehören u. a. *MSS*, *WSopt*, *SACK*, *SACK Permitted* und *TSopt*. Die wichtigsten sollen im Anschluss diskutiert werden. Weitere Protokolloptionen wurden unter anderem für TCP-Erweiterungen wie *T/TCP* (*TCP Fast Open)* vorgeschlagen. Auf diese gehen wir allerdings nicht ein, da sie keine praktische Relevanz haben.

3.4.1 Maximum Segment Size Option

Mit der *MSS*-Option (*Maximum Segment Size Option*) nach RFC 879 können beim Verbindungsaufbau die maximal zulässigen Segmentlängen, also die *Maximum Segment Size (MSS*), ausgetauscht werden. Wenn diese Option in einem TCP-Header enthalten ist, muss auch gleichzeitig das SYN-Flag gesetzt sein. Sie wird also nur beim Verbindungsaufbau genutzt.

Für die MSS-Angabe stehen im TCP-Header genau 16 Bit zur Verfügung, d. h. ein Segment kann maximal 64 KiB groß sein. Beide Hosts übermitteln beim Verbindungsaufbau ihr Maximum und der kleinere der beiden Werte wird genommen. Als grundlegende Vereinbarung gilt, dass alle Hosts TCP-Segmente mit einer Länge von von 536+20 Byte akzeptieren müssen. Wird also nichts weiter vereinbart, gilt diese MSS für die Verbindung.

Für die Option wird der Typ auf „2“ gesetzt, die Optionsdaten sind insgesamt vier Byte lang.

MSS Clamping

IP-Router, die üblicherweise Netzwerke mit verschiedenen MTUs verbinden, greifen oft unter Verletzung der Schichtenaufgaben beim TCP-Verbindungsaufbau auf den TCP-Header zu, um die MSS auf eine geeignete Größe anzupassen. Ziel ist es, die aufwändige IP-Fragmentierung zu vermeiden. Dieses Verfahren wird *MSS Clamping* genannt. Üblich ist eine Anpassung der MSS an die ermittelte MTU (siehe Path MTU Discovery). Bei Routern kann man MSS Clamping in der Regel per Konfigurationskommando aktivieren bzw. deaktivieren.

Auch DSL-Router nutzen üblicherweise MSS Clamping, da der verwendete PPPoE-Protokoll-Header den Platz für die Nutzdaten in der MTU und damit auch die MSS nochmals verringert.

In Abb. 3.24 ist eine typische MSS-Vereinbarung beim Verbindungsaufbau skizziert.

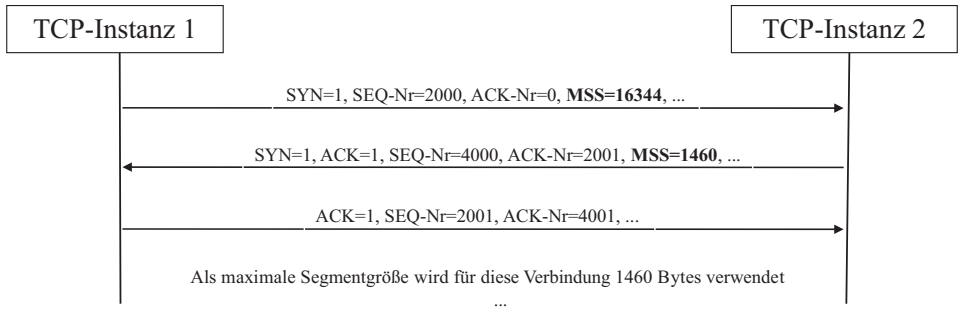


Abb. 3.24 MSS-Option beim Verbindungsaufbau vereinbaren

3.4.2 TCP Window Scale Option

Die *Window Scale* Option (*WSopt*) ist im RFC 1323 geregelt und dient zum Austausch der maximalen Fenstergröße für das Sliding-Window-Verfahren. Dieser Parameter wird auch als *TCP Receive Window Size* (kurz: *RWin*) bezeichnet.

Für Leitungen mit hoher Bandbreite bzw. hohem Bandbreiten-Verzögerungsprodukt (siehe Definition) sind nämlich die 64 KiB, die über das TCP-Headerfeld *Zeitfenstergröße* als maximale Fenstergröße einstellbar sind, zu klein. Ein Sender müsste dann eventuell zu lange warten, bis er erneut senden darf.

► **Bandbreiten-Verzögerungs-Produkt und long fat networks** Das Bandbreiten-Verzögerungs-Produkt (bandwidth-delay product) ist das Produkt aus der Nachrichtenverzögerung (delay) in einem Netzwerk (gemessen in s) und der Netzwerkbandbreite (gemessen in Bit/s). Ergebnis der Berechnung ist die Anzahl an Bytes, die sich im Netzwerk befinden können. Das Netzwerk ist damit gewissermaßen ein Puffer für in Übertragung befindliche Daten. Ist das Produkt groß, spricht man auch von sogenannten *long fat networks*. Im RFC 1072 (TCP Extensions for Long-Delay Path, vom Oktober 1988) werden Netze auch als long fat network bezeichnet, wenn sie ein Bandbreiten-Verzögerungs-Produkt von deutlich über 10^5 Bits und mehr aufweisen.

Das Bandbreiten-Verzögerungs-Produkt beeinflusst den Datendurchsatz durch einen Kommuikationskanal wesentlich. Bei TCP wird das Verzögerungs-Bandbreiten-Produkt auch über das Produkt aus RTT und Datenübertragungsrate berechnet. TCP hat mit long fat networks Probleme, da bei diesen Netzen der Durchsatz aufgrund des Bestätigungsverfahrens nicht optimal ist. Die Fenstergröße sollte mindestens so groß wie das Bandbreiten-Verzögerungs-Produkt sein, um die Netzwerkkapazität auszunutzen.

In einem Netz mit 1 GBit/s und einer RTT von 10 ms ist das Bandbreiten-Verzögerungs-Produkt = 40 ms * 1 Gbit/s = $4*10^{-4}$ s * 10^9 Bit/s = $4*10^5$ Bit (50 KByte).

Die Fenstergröße kann über die Option WSopt bis auf 2^{30} Bit (= 1 GiB) skaliert werden, indem der Inhalt des Feldes *Zeitfenstergröße* um max. 14 Bit nach links verschoben wird.

Dieser Verschiebewert wird als Skalierungsfaktor bezeichnet und wird in dieser Option übertragen.

Die Realisierung der Skalierung erfolgt durch Linksshift des Feldes *Zeitfenstergröße* um so viele Bit wie im Skalierungsfaktor angegeben werden (maximal 14). Der Wert für WSopt ist also eine Potenz von 2, mit der die aktuell im TCP-Segment angegebene Fenstergröße multipliziert wird, um die tatsächliche Fenstergröße für den Sliding-Window-Mechanismus zu erhalten (siehe WSopt-Beispiel).

WSopt-Beispiel

Wenn beim Verbindungsaufbau in der Option WSopt der Wert 8 im TCP-Header übertragen wird, so ergibt sich eine Fenstergröße-Skalierung mit 2^8; die ohne WSopt-Skalierung vorhandene Fenstergröße wird also mit 256 multipliziert.

Ist die Fenstergröße beispielsweise im TCP-Header mit 8192 belegt, so ergibt sich die skalierte maximale Fenstergröße mit 8192 * 256 Bytes = 2.097.152 Bytes

Das Aushandeln der Fenstergröße kann nur beim Verbindungsaufbau (SYN-Flag gesetzt) erfolgen und ist für beide Senderichtungen unabhängig möglich. Die Begrenzung des Skalierungsfaktors auf maximal 14 Bit hat mit dem Wertebereich der Sequenznummer zu tun. Die Fenstergröße muss kleiner als der Abstand zwischen linkem und rechtem Ende des Sequenznummernbereichs sein.

Die Window-Scale-Option ist mit Typ = 3 gekennzeichnet und die Optionsdaten sind 3 Byte lang.

WSopt-Einstellung in Betriebssystemen

Die Option WSopt ist in Betriebssystemen gelegentlich limitiert, unter Windows 2008 etwa auf eine maximale Fenstergröße von 16 MiB.

3.4.3 SACK-Permitted-Option und SACK-Option

TCP ermöglicht es, die Übertragungswiederholung per Option zu verändern. Das Standardverfahren ist Go-Back-N. Anstelle des Go-Back-N-Verfahrens kann selektive Wiederholung für eine TCP-Verbindung genutzt werden. Dies wird mit der SACK-Permitted-Option festgelegt (RFC 2018 und 2883)

Mit der Option *SACK* (RFC 2018 und 2883) kann man zudem erlauben, dass in einer ACK-PDU (ACK-Flag gesetzt) eine Liste von Sequenznummernbereichen übergeben wird, die bereits empfangen wurden. Die Liste ist variabel lang und enthält Nummernpaare (left edge und right edge) zur Angabe der bestätigten Datenblöcke. Die erste Nummer eines Paares (left edge) gibt dabei die erste Sequenznummer eines empfangenen Datenblocks an. Die zweite Nummer (right edge) gibt die erste nicht empfangene Sequenznummer an. Damit kann dedizierter angegeben werden, welche Daten bereits empfangen wurden. Bei Verbindungen mit umfangreicher Fenstergröße ist das von Vorteil.

Die SACK-Permitted-Option hat den Typ 4, die SACK-Option hat als Typ 5. Die Optionsdaten sind bei SACKOK zwei Byte und bei SACK variabel lang. Wenn in einer SACK-Option n Blocks angegeben sind, ist die Option insgesamt 8*n+2 Bytes lang. Da der TCP-Header maximal 40 Bytes für Optionen bereitstellt, kann eine SACK-Option maximal 4 Blöcke enthalten, sofern sonst keine weiteren Optionen benutzt werden.

Beide Optionen können beim Verbindungsaufbau vereinbart werden, also im TCP-Segment, in dem das SYN-Flag gesetzt ist. Die Option kann in Betriebssystemen (siehe TCP-Parameter im Anhang) auch meist für alle TCP-Verbindungen gesetzt werden.

3.4.4 Timestamps Option

Die *Timestamps Option* (TSopt) besteht aus zwei Zeitstempeln gemessen in ms zu je 4 Byte, einem sogenannten *Timestamp Value* (TSval) und einem *Timestamp-Echo-Reply* (TSecr). Das erste Feld wird von der sendenden TCP-Instanz belegt. Letzteres Feld ist nur bei ACK-Segmenten (ACK-Flag=1) erlaubt. Die *Timestamps Option* (TSopt) hat den Typ „8“, die Optionsdaten sind 10 Byte lang. Die TSopt-Option wird beim Verbindungsaufbau für die Dauer der Verbindung aktiviert. Manche Implementierungen wie etwa in Windows erlauben es aber auch, die TSopt-Option jederzeit während der Verbindung zu aktivieren.

Mit den beiden Werten TSval und TSecr können sich die TCP-Instanzen einer Verbindung über die sogenannte Round-Trip-Time (RTT) informieren. Man kann also damit die Zeit messen, die benötigt wird, um ein TCP-Segment zu senden und um die Bestätigung zu erhalten, dass sie beim Empfänger angekommen ist. Damit ist neben der klassischen Methode der RTT-Messung auch noch eine weitere Methode verfügbar.

Beim Sender wird das TSval-Feld im TCP-Header mit dem Zeitstempel aus der lokalen Uhr befüllt, bevor ein Segment gesendet wird. Das TSecr-Feld wird vom Empfänger bei der Bestätigung mit dem vom Sender übertragenen Wert befüllt. Der Sender erhält also mit dem Bestätigungssegment seinen eigenen Zeitstempel zur Absendezeit zurück und kann dann mit der aktuellen Zeit und dem TSecr-Wert aus seiner lokalen Uhr die RTT berechnen (Abb. 3.25).

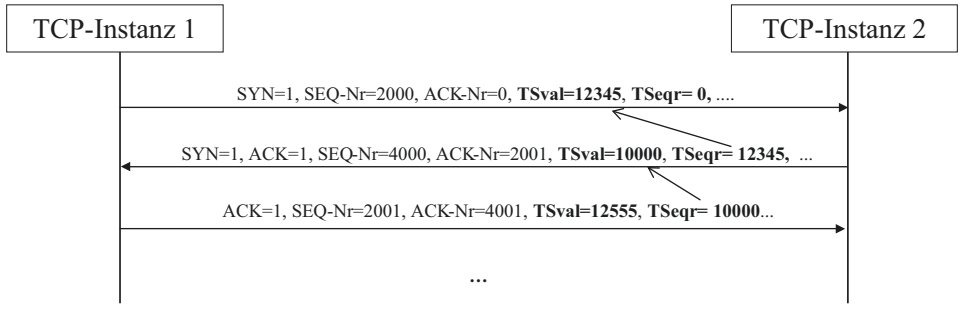


Abb. 3.25 Nutzung von TSval zur Unterstützung der RTT-Berechnung

Die TSopt-Option kann in mehreren Situationen genutzt werden. Zum einen kann sie, unter Berücksichtigung einiger in RFC 7323 erläuterten Regeln, für die RTO-Berechnung verwendet werden. Zum anderen ist eine Nutzung zur Lösung des PAWS-Problems möglich, um TCP-Segmente alter Verbindungs-Inkarnationen zu erkennen (siehe Diskussion zum Thema *Protect Against Wrapped Sequences*).

Die so ermittelte RTT kann in TCP-Implementierungen beim Verbindungsabbau auch zur Verkürzung der TIME_WAIT-Wartezeit genutzt werden, da ja recht exakte RTT-Werte vorliegen.

3.5 Protect Against Wrapped Sequences

3.5.1 Problemstellung

Beim TCP-Verbindungsaufbau tauschen die beteiligten TCP-Instanzen ihre initialen Folgenummern (Sequenznummern, 32 Bit-Feld im TCP-Header) aus und aktualisieren diese im Laufe der Datenübertragungsphase. Die übertragenen Bytes werden über die Sequenznummer gezählt. Die Sequenznummer gibt also für jede Senderichtigung an, wo sich die Übertragung gerade befindet. In der Bestätigungsnummer wird angezeigt, welche Bytes schon empfangen wurden bzw. welche Sequenznummer als nächstes erwartet wird. Damit wird eine lückenlose Übertragung unterstützt, fehlende Bytes können erkannt werden.

Da der Wertebereich der Folgenummer (Sequenznummer) einer TCP-Verbindung aber auf 2^{32} Bytes begrenzt ist, besteht die Gefahr, dass bei umfangreicher Kommunikation der Wertebereich der Sequenznummer erschöpft wird. Die Sequenznummernzählung wird nämlich nach einem „wrap around“ bei 0 fortgeführt. Abb. 3.26 skizziert, wie Sequenznummern innerhalb einer Verbindung doppelt vorkommen können.

Sequenznummern können sich während einer Verbindung wiederholen. Das ist prinzipiell kein Problem, sondern erst dann, wenn beim Empfänger TCP-Segmente mit Sequenz-

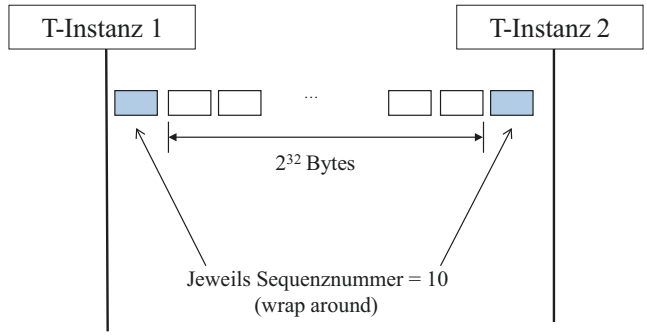


Abb. 3.26 Sequenznummern-Überlauf

nummern ankommen, die schon vergeben wurden und die bereits vorher vergebenen noch nicht beim Empfänger angekommen sind.

Dies könnte bei Netzen mit hoher Pfadkapazität und hoher Latenzzeit, also bei sogenannten *long fat networks* wie beispielsweise bei Hochleistungs-Satellitenübertragungsnetzen, vorkommen. Bei einer sehr aktiven TCP-Verbindung etwa für die Übertragung großer Dateien über ein Filetransfer-Protokoll könnte also ein Problem entstehen, das als Sequenznummern-Kollision bezeichnet wird. Bei einem Netzwerk mit einer Übertragungsrate von 1 Gbit/s und voller Auslastung könnte sich schon nach etwas mehr als einer halben Minute eine Sequenznummernwiederholung ergeben. Wenn das Netz dann noch eine hohe Kapazität aufweist, ist eine Sequenznummern-Kollision zumindest möglich (siehe Beispiel zu Sequenznummern in schnellen Netzen).

Sequenznummern in schnellen Netzen

Die Sequenznummer wird jeweils um die Anzahl an Bytes, die in einem TCP-Segment übertragen werden, erhöht, wobei die Berechnung wie folgt ausgeführt wird:

$$S_{neu} = (S_{alt} + 1) \text{ modulo } 2^{32},$$

S_{neu} entspricht der neu zu ermittelnden, S_{alt} der zuletzt benutzten Sequenznummer einer Senderichtigung.

Je nach Übertragungsgeschwindigkeit können unterschiedliche Situationen auftreten:

- Bei einer Übertragungsrate von 64 Kbit/s kommt es frühestens nach ca. 6,2 Tagen zu einer Wiederholung der Sequenznummer
- Bei 100 Mbit/s kommt es frühestens nach ca. 340 s zu einer Wiederholung
- Bei 1 Gbit/s kommt es frühestens nach ca. 34 s zu einer Wiederholung, wie die nachfolgende Brechnung zeigt:

$$2^{32*} \text{ 8 Bit} : 1.000.000.000 \text{ Bit/s} = 2^{35} : 10^{9}\text{s} \sim 34,35 \text{ s}$$

In lokalen Netzen ist Letzteres heute schon der Normalfall.

Eine Sequenznummern-Kollision könnte also vorkommen, wenn ein TCP-Segment mit einer alten Sequenznummer x noch im Netz unterwegs ist und gleichzeitig ein neues TCP-Segment mit der gleichen Sequenznummer x nach einem Wrapping gesendet wird. Bei einer Überholung der TCP-Segmente könnte es dann zu Inkonsistenzen kommen.

Die TCP-Spezifikation legt zwar fest, dass die Folgenummern zeitgebergesteuert mit einem bestimmten Zyklus (ursprünglich 4,6 Stunden) ermittelt werden sollen, um die Wahrscheinlichkeit von Kollisionen niedrig zu halten, jedoch reicht das in heutigen schnellen Netzen nicht aus.

Ein Wrapping der Sequenznummern mit möglicher Sequenznummern-Kollision könnte auch bei einem erneuten, schnellen Verbindungsaufbau nach einem Crash einer Verbindung oder eines Rechners entstehen. Ein noch altes TCP-Segment könnte nach neuer „Inkarnation“ der neuen TCP-Verbindung mit gleichen Adressparametern (gleiches Socket Pair) beim Zielrechner ankommen. Es würde dann möglicherweise versehentlich der neuen Verbindung zugeordnet.

3.5.2 Maßnahmen zum Schutz vor Sequenznummern-Kollisionen

Die Maßnahmen zur Vermeidung von Sequenznummern-Kollisionen wurden unter dem Begriff *Protect Against Wrapped Sequences* oder *PAWS* zusammengefasst.

Das Problem wird zum einen durch eine Vorgabe bei der Synchronisation der Sequenznummern im Dreiwege-Handshake-Verbindungsaufbau abgemildert, da eine initiale Sequenznummer anhand eines fiktiven Zeitgebers berechnet wird. Für den Verbindungsabbau wurde im RFC 792 die maximale Segmentlebensdauer (MSL) auf zwei Minuten festgelegt. Im TIME_WAIT-Zustand muss standardmäßig 2 * MSL abgewartet werden, bevor nach einem Crash einer Verbindung eine neue initiale Sequenznummer zugewiesen wird.

Weitere Lösungsansätze für das Wrapping-Problem bei Sequenznummern sind in RFC 7323 (TCP Extensions for High Performance) beschrieben. Eine vorgeschlagene Lösung ist die Nutzung der TCP-Option *TSopt*. Bei diesem Ansatz werden Timestamps verwendet, um ein Sequenznummern-Wrapping in schnellen Netzen zu erkennen. Wird die Option verwendet, kann sich eine TCP-Instanz nämlich den Zeitstempel des zuletzt empfangenen TCP-Segments für jede Verbindung merken. Kommen TCP-Segmente, die älter als das zuletzt empfangene TCP-Segment sind, können diese anhand des TSval-Wertes erkannt und verworfen werden. Diese Segmente müssen nämlich von einer vorherigen Verbindung sein.

3.6 Staukontrolle

Im Jahre 1986 wurde das Internet durch massive Stausituationen gestört. Aus diesem Grund suchte man für die Staukontrolle (auch Congestion Control oder Überlastkontrolle) eine Lösung, die man 1989 auch standardisierte und ständig weiterentwickelt. Staukontrolle darf nicht mit Flusskontrolle verwechselt werden. Während die Staukontrolle Netzprobleme zu vermeiden versucht, kümmert sich die Flusskontrolle darum, dass keine Empfangspuffer in den Endsystemen überlaufen. Beide Mechanismen müssen aber zusammenspielen.

Die Schicht 3 des Internets war ursprünglich nicht in der Lage, der TCP-Schicht irgendwelche Informationen zur Netzauslastung zu übergeben.[8] TCP musste also selbst dafür sorgen, dass es Stausituationen wenn möglich präventiv erkennt und auch darauf reagiert. Dies funktioniert ohne IP-Unterstützung nur auf Basis der einzelnen Verbindungen, also für jede Ende-zu-Ende-Beziehung separat. Die zur Verfügung stehende Information ergibt sich aus dem Bestätigungsverfahren von TCP. Ein Segmentverlust oder eine zu spät

[8] Wie wir in diesem Kapitel noch sehen werden, gibt es mit *Explicit Congestion Notifcation* (ECN) heute einen Ansatz, Informationen des Internet-Protokolls in die Staukontrolle mit einzubeziehen.

ankommende ACK-PDU (Retransmission Timer abgelaufen) wird von TCP als Auswirkung einer Stausituation im Netz interpretiert. TCP nimmt nämlich an, dass Netze prinzipiell stabil sind und daher eine fehlende Bestätigung nach dem Senden einer Nachricht auf ein Netzproblem zurückzuführen ist. Die grundlegende Idee der TCP-Staukontrolle ist also, beim Ausbleiben einer ACK-PDU die Datenübertragung für die Verbindung zu drosseln.

TCP kennt drei Mechanismen zur Staukontrolle, die als Slow-Start, Congestion Avoidance und Fast Recovery bezeichnet werden. Wir gehen im Weiteren auf das Zusammenspiel der Mechanismen ein. Ein neuer Mechanismus, der auch die Schicht 3 mit einbezieht, wird als Explicit Congestion Notification bezeichnet. Er wird ebenfalls in diesem Kapitel besprochen.

3.6.1 Slow-Start und Congestion Avoidance

Das erste Verfahren wird genauer als *reaktives Slow-Start-Verfahren* (Langsamstart-Verfahren) bezeichnet. Es ist in den RFCs 1122, 2581, und 5681 beschrieben. Eine andere Bezeichnung für das Verfahren ist *TCP Tahoe*. Alle TCP-Implementierungen müssen das Slow-Start-Verfahren unterstützen.

Neben dem Empfangsfenster des Sliding-Window-Verfahrens wird für das Slow-Start-Verfahren ein *Überlast-* bzw. *Staukontrollfenster* verwaltet. Es dient als zusätzlicher Mechanismus, um die Anzahl an unbestätigten Segmenten zu begrenzen, wenn zwar der Empfänger noch Sendekredit gewährt, aber das Netzwerk vermutlich überlastet ist.

Jede Verbindungsseite verwaltet die Größe des Überlastfensters aus seiner Sicht im Verbindungskontext und passt es dynamisch an. Die initiale Größe des Überlastfensters war ursprünglich die maximale Segmentgröße (MSS), wurde aber im Laufe der Jahre auf 10 und später sogar auf 15 MSS erhöht. Diese Größe ist aber ständig in Diskussion und auch abhängig von der Implementierung.

Wie bereits erwähnt, baut das Verfahren auf dem Erkennen von Datenverlusten auf. Bestätigungen dienen als Taktgeber für den Sender. Anfangs wird nur ein kleines Überlastfenster eingestellt, das aber innerhalb kurzer Zeit bei problemloser Übertragung exponentiell anwächst. Wenn ein Überlastfall auftritt, wird die Übertragungsrate vom Sender massiv gedrosselt, indem das Überlastfenster deutlich verringert wird. Die gesendeten Datenmengen werden damit also kontrolliert. Im Gegensatz zur Flusskontrolle drosselt bei der Staukontrolle der Sender die Datenübertragung.

Slow-Start funktioniert nun so, dass es sich für jede Verbindung beginnend mit einer kleinen Überlastfenstergröße an die optimale Datenübertragungsrate herantastet. Die Größe des Überlastfensters gibt also die maximale Menge an Daten an, die ohne Bestätigung gesendet werden darf. Gemeinsam mit dem Empfangsfenster der Flusskontrolle gilt dann für jeden Sendevorgang und für jede Seite einer TCP-Verbindung separat:

$$Sendekredit = \min\left(\ddot{U}berlastfenstergr\ddot{o}\beta e, Empfangsfenstergr\ddot{o}\beta e\right)$$

Dies bedeutet, dass der Sendekredit einer TCP-Verbindung nicht größer als das Minimum aus der Größe des Überlastfensters und des Empfangsfensters (Fenstergröße der Flusskontrolle) sein darf. Nach dem Aufbau einer TCP-Verbindung ist die MSS bekannt und die Verbindung befindet sich zunächst in der *Slow-Start-Phase.* Die Überlastfenstergröße wird auf eine initiale Größe (ein oder mehrere MSS) festgelegt. Jede Seite fängt also mit einer eigenen Einstellung an und verdoppelt die Überlastfenstergröße nach jedem erfolgreichen Senden, d. h. wenn eine ACK-PDU (TCP-Segment mit ACK-Flag und richtig belegter Bestätigungsnummer) ankommt. Verdoppeln bedeutet im Prinzip, dass für jedes bestätigte Byte im nächsten Übertragungszyklus ein weiteres Byte hinzukommt. Das geht bis zu einem eingestellten Schwellwert weiter, sofern keine ACK-PDUs ausbleiben. Das Wachstum des Überlastfensters verhält sich in dieser Phase daher exponentiell.

Nach dem Erreichen des voreingestellten Schwellwerts geht das Verfahren in die sogenannte *Probing-* oder *Überlastvermeidungsphase* (Congestion Avoidance) über. In dieser Phase vergrößert sich das aktuelle Überlastfenstergröße nach dem Empfang aller Quittungen für die als letztes gesendeten TCP-Segmente nur noch linear um ein Segment mit der maximal zulässigen Segmentgröße (MSS) der Verbindung.

Man sieht in Abb. 3.27, dass das Slow-Start-Verfahren gar nicht so langsam beginnt, wie sein Name andeutet. Es beginnt zwar mit einem kleinen Überlastfenster, das Überlastfenster wird aber schnell größer, solange alles gut geht. Als Obergrenze gilt natürlich die Empfangsfenstergröße, die auch dynamisch ermittelt wird. Die Empfangsfenstergröße kann allerdings durch Nutzung der Option WSopt (siehe TCP-Header) bis zu 1 GiB groß sein. Aufgrund eines Engpasses auf der Empfangsseite kann immer eine Drosselung des Datenverkehrs initiiert werden (siehe hierzu TCP-Flusskontrolle). Die Ermittlung der Größe des Sendekredits erfolgt für jede Kommunikationsrichtung separat.

Was passiert nun, wenn eine ACK-PDU nicht rechtzeitig eintrifft und ein Retransmission Timer abläuft? TCP geht in diesem Fall davon aus, dass z. B. ein weiterer Sender im Netz hinzugekommen ist und sich die Pfadkapazität mit diesem neuen Sender (und natürlich den schon vorhandenen Teilnehmern) teilen muss, oder aber, dass irgendwo ein Netzproblem vorliegt. Als Reaktion darauf wird der Schwellwert auf die Hälfte des aktuellen Staukontrollfensters reduziert, die Segmentgröße wieder auf das Minimum heruntergesetzt, und das Verfahren beginnt von vorne, also wieder mit einem „slow start“. Die Reaktion ist massiv, die Last wird sofort stark reduziert. Diese Reaktion gilt sowohl für die Slow-Start- als auch für die Probing-Phase.

Abb. 3.27 zeigt ein Beispiel für die Entwicklung des Überlastfensters. Die angenommene Ursprungsgröße des Überlastfensters ist hier ein TCP-Segment mit der zwischen den Partnern ausgehandelten MSS von 1 KiB. Der initial eingestellte Schwellwert liegt bei 16 KiB. Dies wird in der TCP-Implementierung festgelegt. Ab dieser Überlastfenstergröße steigt das Fenster linear um jeweils eine MSS. Bei einer Größe des Überlastfensters von 20 KiB tritt in diesem Szenario ein Timeout auf, die erwartete ACK-PDU bleibt

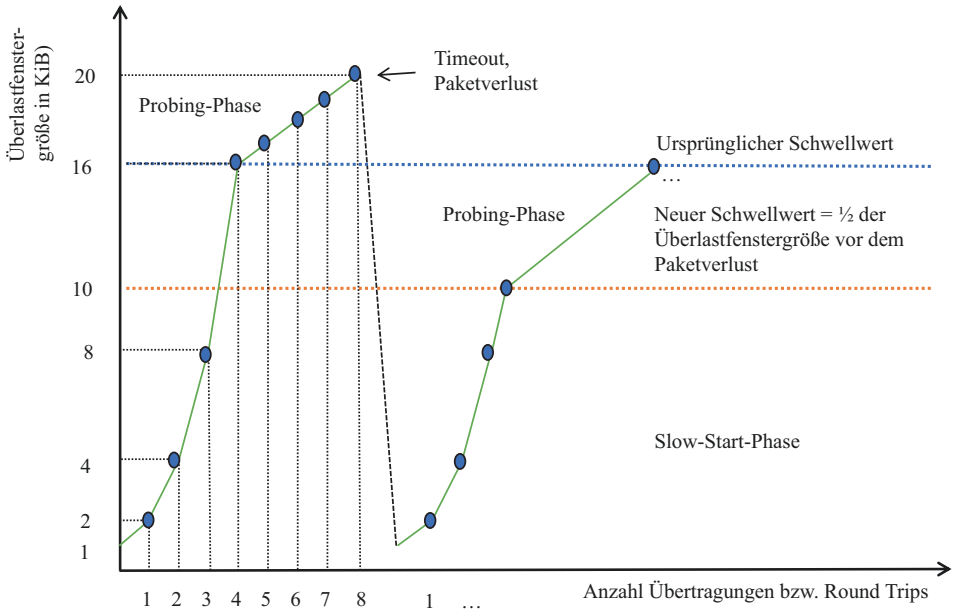


Abb. 3.27 Staukontrolle bei TCP nach (Tanenbaum und Wetherall 2011)

also aus. Der Schwellwert wird dann auf 10 KiB gesetzt, was der Hälfte der aktuellen Überlastfenstergröße entspricht, und das Verfahren beginnt von vorne. TCP geht also in unserem Beispiel bei einer Überlastfenstergröße von 20 KiB davon aus, dass die aktuelle Netzwerkkapazität erreicht ist.

Man sieht an diesem Verfahren, dass auch hier die Timerlänge ganz entscheidend ist. Ist ein Timer zu lang, kann es zu Leistungsverlusten kommen, ist er zu kurz, wird die Last durch erneutes Senden nochmals erhöht. TCP führt daher die Berechnung anhand der maximal erlaubten Umlaufzeit eines Segments dynamisch durch und versucht immer, einen der Situation angemessenen Retransmission Timer zu ermitteln.

Der Slow-Start-Algorithmus wird auch – zumindest in der Congestion-Avoidance-Phase – als AIMD-Algorithmus (Additive-Increase, Multiplicative-Decrease) bezeichnet, weil TCP die Übertragungsrate additiv erhöht, wenn der TCP-Pfad in Ordnung ist und multiplikativ senkt, wenn ein Datenverlust erkannt wird. In Abb. 3.28 wird nochmals das Zusammelspiel der Phasen aufgezeigt. Wenn das vermutete Netzwerklimit erreicht ist, egal ob in der Slow-Start- oder in der Congestion-Avoidance-Phase, wird sofort stark gedrosselt. Der neue Schwellwert wird bei einem Timeout (ACK-PDU kommt zu spät) immer auf die Hälfte der aktuellen Überlastfenstergröße reduziert. Danach tastet sich die TCP-Verbindung wieder an die aktuell verfügbare Bandbreite des Netzwerks heran.

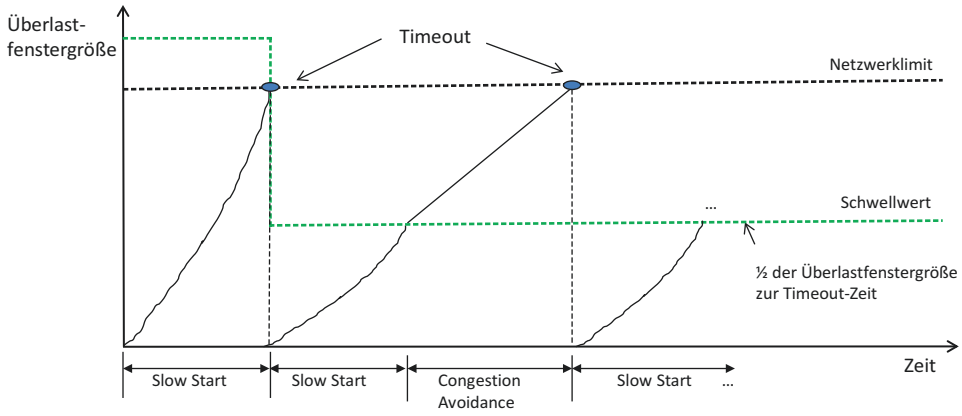


Abb. 3.28 TCP Tahoe bei fehlender Bestätigung

Wenn die Anwendung natürlich nur sehr wenig sendet, wird die Bandbreite nicht ausgereizt und es kommt kaum zu einer Überlastsituation.

3.6.2 Fast-Recovery-Algorithmus

Eine Ergänzung zu Slow-Start ist der *Fast-Recovery-Algorithmus,* auch als *TCP Renoe* bezeichnet. Der Algorithmus nutzt den mehrfachen Empfang von Bestätigungen für das gleiche TCP-Segment (RFC 2581, 5681) dazu, um einen Paketverlust zu diagnostizieren. Wenn der Sender vier Quittierungen (drei sogenannte ACK-Duplikate) für ein konkretes TCP-Segment empfängt, er aber schon weitere TCP-Segmente danach gesendet hatte, geht er von einem Paketverlust aus und veranlasst die sofortige Sendewiederholung des Segments. Hier spricht man auch von *Fast Retransmission*. Der Empfänger sendet immer dann ein ACK-Duplikat, wenn ein empfangenes TCP-Segment nicht in die Reihenfolge passt. (Abb. 3.17)

Die Sendeleistung wird nach dem dreimaligen Empfang der ACK-Duplikate auch entsprechend angepasst. Der Schwellwert aus dem Slow-Start-Verfahren wird auf die Hälfte des aktuellen Überlastfensters reduziert. Die dem verloren gegangenen Segment folgenden Segmente sind im Empfangspuffer angekommen und daher geht man nur von einem Paketverlust und nicht von einem Stau im Netzwerk aus. Aus diesem Grund wird das Überlastfenster auf einen Wert über dem Schwellwert gesetzt und es wird mit der Congestion-Avoidance-Phase fortgefahren. Damit wird ein zu starkes Reduzieren der Sendelast vermieden. Bei einem Retransmission Timeout wird aber wie bei TCP Tahoe verfahren (Abb. 3.29).

Das Fast-Recovery-Verfahren ist für eine TCP-Implementierung nicht zwingend vorgeschrieben, sondern nur eine Implementierungsempfehlung.

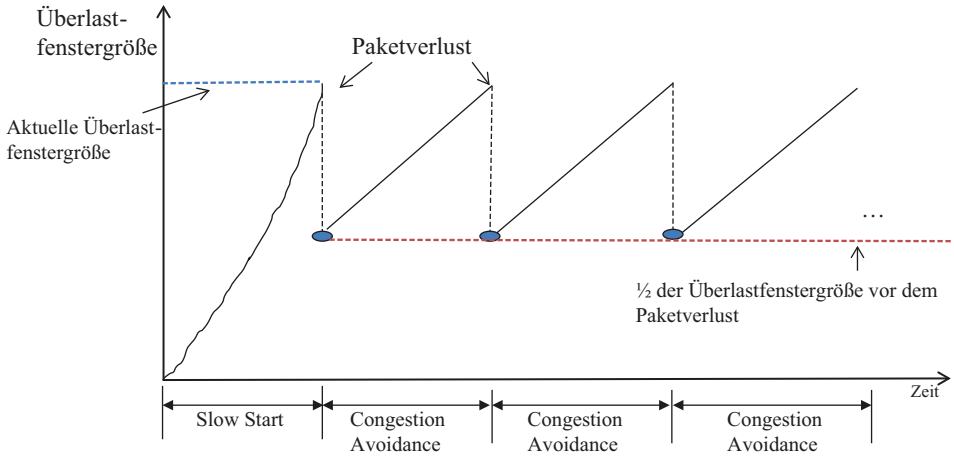


Abb. 3.29 TCP Renoe

3.6.3 Explicit Congestion Notification

Eine weitere Möglichkeit für die Staukontrolle ist durch die Einbeziehung der Vermittlungsschicht gegeben. In den RFCs 2481 und 3168 (The Addition of Explicit Congestion Notification (ECN) to IP) wurden zwei weitere Flags im TCP-Header vorgeschlagen, die für die Ende-zu-Ende-Signalisierung von Stauproblemen, also zur expliziten Stausignalisierung zwischen Endpunkten verwendet werden sollen. Das Verfahren wird als Explicit Congestion Notification (ECN) bezeichnet. Die zusätzlichen Flags im TCP-Header werden mit CWR (*Congestion Window Reduced*) und ECE (*Explicit Congestion Notification* Flag) bezeichnet.

Während des Verbindungsaufbaus handeln hierfür die beiden Partner über die Flags CWR und ECN aus, ob sie für die Transportverbindung eine Ende-zu-Ende-Stausignalisierung einrichten wollen. Der aktive TCP-Partner setzt in der Connect-Request-PDU das ECE-Flag auf 1 und das CWR-Flag auf 0, um dies anzuzeigen Der passive TCP-Partner antwortet in der Connect-Response-PDU mit einer Bestätigung, in der das SYN-Flag, das ACK-Flag und das ECE-Flag gesetzt sind, nicht aber das CWR-Flag. Ist kein ECN gewünscht, dann wird vom Partner in der Antwortnachricht weder das CWR- noch das ECE-Flag gesetzt. Der Verbindungsaufbau mit dem Signalisieren der ECN-Fähigkeit ist in Abb. 3.30 skizziert. Wie in der Abbildung zu sehen ist, muss auch im IP-Header entsprechende Information übertragen werden. In den Bits 6 und 7 des bisherigen ToS-Feldes des IP-Headers (Mandl et al. 2010) unterstützen nun die Bits ECN und ECE bei dem neuen Verfahren. Die Bitkombinationen 01 und 10 deuten an, das ECN auch im Router eingeschaltet ist.

Wie Abb. 3.31 zeigt, wird eine tatsächliche Stausituation dann über die IP-Router an die Endsysteme signalisiert, wobei ein Zusammenspiel der Schichten notwendig ist und alle IP-Router auch entsprechend mitarbeiten müssen. Wenn die Routerwarteschlange einen

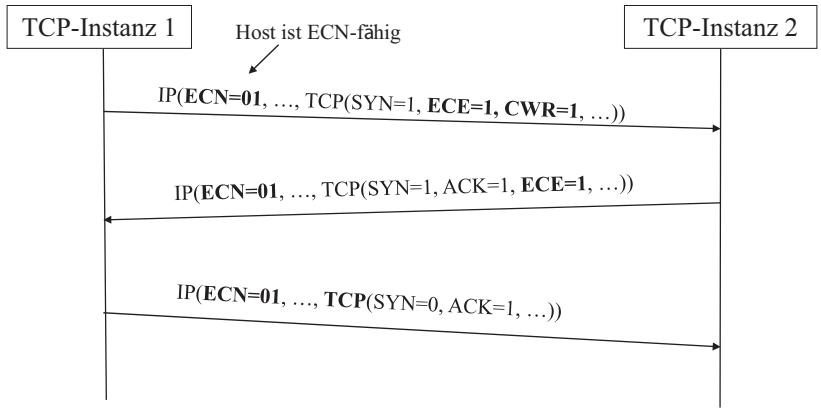


Abb. 3.30 ECN-Aktivierung im Zusammenspiel der Schichten

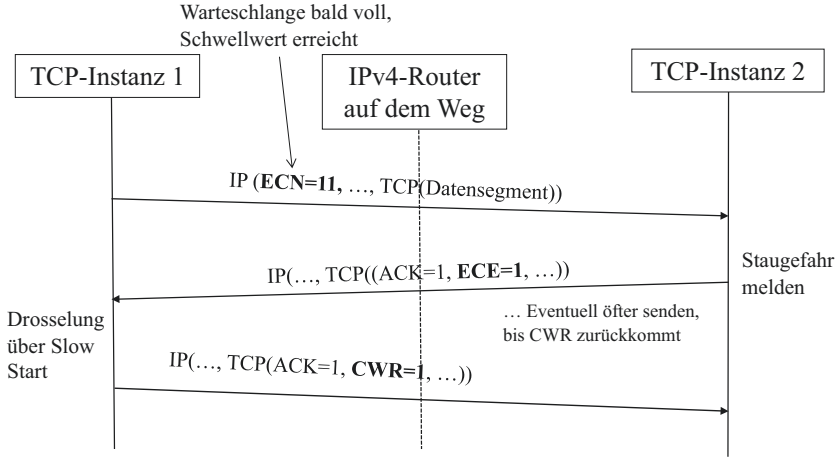


Abb. 3.31 Stausignalisierung mit ECN im Zusammenspiel der Schichten inklusive Router

bestimmten Schwellwert erreicht hat, kann das Ereignis durch einen IP-Router durch Setzen der ECN-Bits auf 11 im IP-Header zum Endsystem gemeldet werden. Die TCP-Instanzen haben dann die Möglichkeit, sich über den TCP-Header mit gesetztem ECE-Flag zu informieren und auch die Drosselung der Netzlast zu signalisieren (CWR-Flag=1).

Der Algorithmus wird auch als Weighted Random Early Detection (WRED) bezeichnet. Ein IP-Router kann damit also einen drohenden Stau frühzeitiger anzeigen als bei rein TCP-basierten Verfahren. TCP kann dann seine Sendeaktivitäten drosseln. Alle IP-Router auf dem Weg und die Endsysteme müssen aber zusammenarbeiten. Daher ist das ECN-Verfahren noch immer in der Erprobungsphase.

3.6.4 Weitere Ansätze der TCP-Staukontrolle

Mit zunehmender Netzwerkbandbreite wie etwa in long fat networks wurden die TCP-Staukontroll-Mechanismen immer wieder in Frage gestellt und es wurden Optimierungen vorgeschlagen. Insbesondere für Hochgeschwindigkeitsnetze eignen sich Slow-Start und Congestion-Control nur sehr bedingt. RFC 3649 (High Speed TCP for Large Congestion Windows) befasst sich mit der Staukontrolle in Hochgeschwindigkeitsnetzen. Insbesondere wird dort diskutiert, wie niedrig die Segmentverlustwahrscheinlichkeit bei den TCP-Staukontrollverfahren sein darf, wenn Übertragungsgeschwindigkeiten von 10 GBit/s erreicht werden sollen. Es wird in dem RFC aufgezeigt, dass in diesem Fall mit heutigen Verfahren nur noch alle fünf Milliarden Segmente ein Segmentverlust auftreten dürfte.

Eine Fülle von neuen Algorithmen bzw. Verbesserungen aktueller Algorithmen wurden untersucht und auch implementiert. Beispiele hierfür sind New Reno (RFC 3782, The NewReno Modification to TCP's Fast Recovery Algorithm) und HSTCP (RFC 3649, HighSpeed TCP for Large Congestion Windows). Die Forschung und Entwicklung geht hier weiter. Ein Überblick über einige Verfahren ist in (Sangtae et al. 2007, 2008) zu finden.

Moderne Betriebssysteme ermöglichen es auch, die Staukontrollmechanismen zu konfigurieren. Linux hat beispielsweise einen sehr modernen TCP/IP-Stack mit vielen Konfigurationsmöglichkeiten (siehe Beispiel). Die Implementierung verfügt auch über eine modulare Staukontroll-Architektur und unterstützt per Konfiguration auch eine Vielzahl von neueren Verfahren.

Auch bei mobilen Geräten ist die aktuelle TCP-Staukontrolle nicht optimal, da in drahtlosen Netzen TCP-Segmente häufiger durch Übertragungsfehler verlorengehen können, was meist nichts mit einem Netzwerkstau zu tun hat. Eine Lösung ist hier beispielweise, die Anbindung von drahtlosen Geräten in Wireless LANs so zu organisieren, dass für die Teilverbindung zwischen einem Wireless LAN Access Point (WLAN) und dem mobilen Gerät andere Mechanismen gelten als für die Teilverbindung zwischen dem WLAN Access Point und einem stationären Partner in einem Festnetz. Die Verbindung wird also in zwei Einzelverbindungen unter Verzicht auf die Ende-zu-Ende-Semantik aufgeteilt. Dieses Verfahren wird als indirektes TCP bezeichnet (indirektes TCP, I-TCP). Ein weiteres Verfahren ist Snooping TCP (Mandl et al. 2010).

Konfiguration der Staukontroll-Algorithmen unter Linux

Unter Linux kann man aus verschiedenen Staukontrollmechanismen eine Teilmenge per Konfiguration auswählen. Im Verzeichnis /proc/sys/net/ipv4/ sind auch Parameter für die Staukontrolle zu finden, die mit entsprechender Berechtigung auch verändert werden können:

- Der Parameter *tcp_available_congestion_control* enthält alle möglichen Staukontrollverfahren, die der aktuelle Linux-Kernel unterstützt. Möglich sind z. B. BIC, CUBIC, Westwood, Vegas, New Reno, HSTCP und weitere (Sangtae et al. 2008). Eine Übersicht über Linux-Implementierungen findet sich auch unter *http://sgros.blogspot.de/2012/12/controlling-which-congestion-control.html*.

- Der Parameter *tcp_allowed_congestion_control* enthält die aktuell eingestellten Staukontroll-Verfahren. Dies ist eine Teilmenge der Liste in *tcp_available_congestion_control.*
- Der Parameter *tcp_congestion_control* enthält die Standardeinstellung, die in aktuellen Linux-Versionen (4.x) mit „reno“ für TCP Reno belegt ist.

Unter Linux kann über die Socket-Schnittstelle auch für jede einzelne TCP-Verbindung festgelegt werden, welche Staukontrollmechanismen genutzt werden sollen. Hierzu wird der C-Funktionsaufruf *setsocketopt* verwendet.

3.7 Timer-Management

3.7.1 Grundlegende Überlegung

Die richtige Einstellung von Timer-Werten ist sehr wichtig für die Leistungsfähigkeit eines Protokolls. Dies gilt natürlich auch für TCP. Wie in Abb. 3.32 dargestellt, ist die optimale Länge eines Timers dann gegeben, wenn sich die Wahrscheinlichkeitsdichte der Ankunftszeiten von Bestätigungen wie in Bild (a) verhält. Dies ist aber in der Schicht 2 viel einfacher zu erreichen als in der Schicht 4. In Schicht 2 ist die erwartete Verzögerung gut vorhersehbar. In Schicht 4 ist dies weitaus komplizierter, da die Rundreisezeit (Round Trip Time, RTT) über mehrere Router gehen kann und sich dies auch noch dynamisch ändert. Eine Überlastung des Netzwerks verändert die Situation in wenigen Sekunden. Die Wahrscheinlichkeitsdichte der Ankunftszeiten von Bestätigungen ist bei TCP eher durch die Funktion in Bild (b) beschreibbar. Ein Timer-Intervall T1 führt hier zu unnötigen Neuübertragungen, und T2 führt zu Leistungseinbußen durch zu lange Verzögerungen.

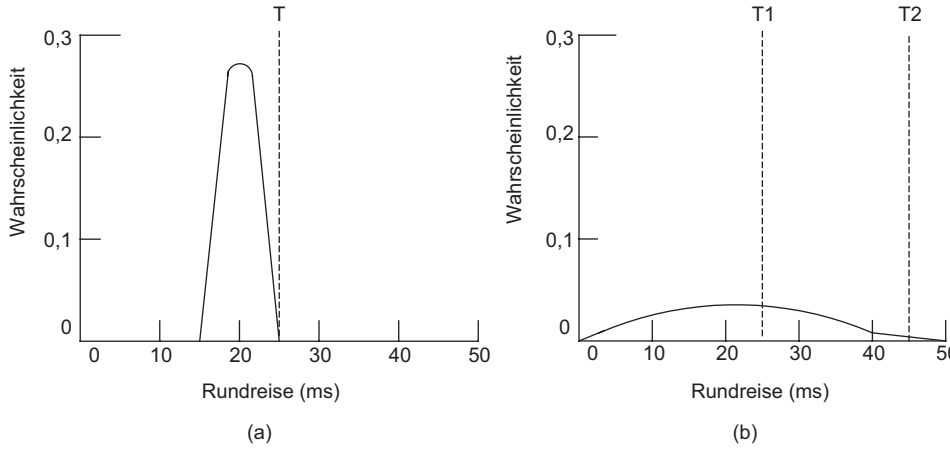


Abb. 3.32 Timerproblematik bei TCP nach (Tanenbaum und Wetherall 2011)

Gehen Datenpakete verloren, so müssen sie erneut gesendet werden. Die Round-Trip-Time wird von der letzten Wiederholungssendung bis zum Empfang einer Bestätigungs-PDU ermittelt, wodurch die RTT zu niedrig angesetzt wird. Neuere Implementierungen beziehen daher bei der Ermittlung der RTT die mehrfach gesendeten Datenpakete nicht mehr mit ein und haben die Bestimmung des Timeouts weiter optimiert.

3.7.2 Retransmission Timer

Der für die Leistungsfähigkeit des Protokolls wichtigste Timer ist ohne Zweifel der *Retransmission Timer*. Der Retransmission Timer dient zur Überwachung der Übertragung der TCP-Segmente nach dem Karn-Algorithmus. Beim Absenden eines TCP-Segments wird der Timer aktiviert (aufgezogen). Wenn der Timer abläuft, bevor eine Bestätigung empfangen wird, wird verdoppelt und das TCP-Segment wird erneut gesendet. Es kommt also zu einer Übertragungswiederholung. Wird die Bestätigung vor Ablauf des Timers empfangen, wird der Timer gelöscht und der Retransmission Timer für die folgenden Segmente neu berechnet.

In die Berechnung geht die Round Trip Time (RTT) eines TCP-Segments als wesentlicher Parameter mit ein. Die RTT ist im Wesentlichen die Verzögerungszeit bis zur Bestätigung, also die Zeit, die benötigt wird, bis die Bestätigung (ACK-Flag) für ein TCP-Segment empfangen wird. Die Zeitmessung beginnt zum Absendezeitpunkt und dauert bis zum Empfang der Bestätigung.

Die RTT verändert sich dynamisch je nach Lastsituation. Aus diesem Grund wird bei TCP ein sehr dynamischer Algorithmus verwendet, der das Timeout-Intervall auf der Grundlage von ständigen Messungen der Netzlast immer wieder anpasst. Bei jedem Roundtrip wird die neue RTT aus der RTT-Historie und der zuletzt gemessenen RTT ermittelt, wozu ein exponentiell gewichteter, gleitender Durchschnitt berechnet wird.

Der Berechnungsalgorithmus ist im RFC 6298 (Computing TCP's Retransmission Timer) erläutert und stammt ursprünglich von *Jacobson* und *Karn* (RFC 2988, 1122) Der Retransmission Timer wird auch kurz als *RTO* (Retransmission TimeOut) bezeichnet.

Im Verbindungskontext verwaltet eine TCP-Instanz für jede einzelne Verbindung Zustandsvariablen, die für die RTO-Berechnung verwendet werden:

- *SRTT* (Smoothed Round Trip Time) ermöglicht eine Glättung der historischen Werte, um Auswirkungen von Ausreißermessungen abzumildern.
- *RTTVAR* (Round Trip Time Variation) bringt die Streuung der RTT-Werte aus der Historie ein.

Mit der Variablen G wird im Berechnungsalgorithmus die Messgenauigkeit der lokalen Uhr bezeichnet. Der initiale Wert für RTO sollte gemäß RFC 6298 auf drei Sekunden, auf alle Fälle aber größer als eine Sekunde gesetzt werden. Bei der RTO-Berechnung wird jeweils ein Wert in Sekunden ermittelt.

Die erste Messung der RTT wird als R bezeichnet. Die erste RTO-Berechnung ergibt sich wie folgt:

$$SRTT = R$$

$$RTTVAR = R / 2$$

$$RTO = \mathrm{SRTT} + \max\left(\mathrm{G},\ \mathrm{K} * \mathrm{RTTVAR}\right), \mathrm{mit}\, \mathrm{K} = 4$$

Beim Empfang einer Bestätigung wird jeweils der neue RTO-Wert in folgender Reihenfolge berechnet, wobei R′ die jeweils zuletzt gemessene RTT, *RTTVARalt* ein Anhalt für die vorhergehende Streuung, *RTTVARneu* die neue Streuung, *SRTTalt* ein Maß für die historische Glättung und *SRTTneu* ein Maß für die neue Glättung ausdrückt :

$$RTTVARneu = \left(1-\beta\right) * RTTVARalt + \beta * \left|SRTTalt - R'\right|$$

$$SRTTneu = \left(1-\alpha\right) * SRTTalt + \alpha * R'$$

$$RTO = SRTTneu + max\left(G, K * RTTVARneu\right), mit\ K = 4$$

Im RFC 6298 wird für den Wert von α 1/8 und für β wird 1/4 empfohlen. Falls sich bei der Berechnung nach der Formel ein RTO-Wert < 1 ergibt, soll RTO auf 1 gesetzt werden. Als maximaler RTO-Wert ist 60 vorgesehen.

Zu beachten ist, dass RTT-Messungen nicht von erneut übertragenen Segmenten durchgeführt werden dürfen. Eine weitere Einschränkung wird zwingend empfohlen, wenn sich bei der RTO-Berechnung (K * RTTVARneu) ein Ergebnis von 0 ergibt. In diesem Fall muss auf G Sekunden gerundet werden:

$$RTO = SRTTneu + \max\left(G, K * RTTVARneu\right)$$

Wenn eine Bestätigung länger als die zuvor berechnete RTO ausbleibt, muss die TCP-Instanz wie folgt reagieren:

- Der Wert von RTO muss unter Beachtung der Obergrenze von 60 Sekunden verdoppelt werden.
- Zunächst ist das erste nicht bestätigte TCP- Segment erneut zu senden.
- Wenn es sich um eine verlorene Bestätigung für einen Verbindungsaufbauwunsch handelt (SYN-Flag gesetzt), so muss RTO für die Datenübertragungsphase auf 3 Sekunden gesetzt werden.
- Die Zustandsvariablen SRTT und RTTVAR sind zu löschen und der nächste RTO-Wert ergibt sich aus der nächsten Messung.

Durch diese Berechnung der RTO soll erreicht werden, dass sich die Timerlänge für den RTO der Netzwerksituation dynamisch anpasst. SRTT wird bei jeder Berechnung neu

ermittelt. Durch die Nutzung eines exponentiell gewichteten, gleitenden Durchschnitts verliert das Gewicht eines einzelnen RTT-Werts rasch an Bedeutung, womit Schwankungen geglättet werden. RTTVARneu ist auch ein exponentiell gewichteter, gleitender Durchschnitt aus der Differenz der historischen RTT und der zuletzt gemessenen RTT. Damit wird auch die Variabilität der RTT-Werte berücksichtigt. RTTVARneu wird dann groß, wenn auch die Schwankungen der gemessenen RTT-Werte groß sind. Dann wird auch die RTO für folgende Segmente höher eingestellt.

Insgesamt kann festgehalten werden, dass der Berechnungsalgorithmus keine Möglichkeit hat, auf Ursachen der RTO-Veränderungen einzugehen. Man geht allgemein davon aus, dass ein Ablaufen des Retransmission Timers durch eine hohe Last im Netzwerk verursacht wird. Während der Algorithmus in drahtgebundenen Netzwerken gut funktioniert, ist er für drahtlose Netzwerke nicht so gut geeignet, weshalb auch alternative Lösungen wie zum Beispiel I-TCP für drahtlose Verbindungen vorgeschlagen wurden (Mandl et al. 2010).

3.7.3 Keepalive Timer

Der *Keepalive Timer* wird verwendet, um zu überprüfen, ob ein Partner noch lebt, der schon längere Zeit nichts gesendet hat. Läuft der Timer ab, überprüft eine Seite durch das Senden einer speziellen Nachricht, ob der Partner noch lebt. Kommt trotz mehrfacher Versuche keine Antwort zurück, wird die Verbindung abgebaut.

Fällt nämlich während einer bestehenden Verbindung ein Partnerrechner, wie in Abb. 3.33 dargestellt, aus, kann er keine Nachricht mehr an den TCP-Partner (hier TCP-Instanz 2) senden, um ihn über den Ausfall zu informieren. Auf der Seite der TCP-Instanz 2 wird der Verbindungskontext aufrechterhalten, die Verbindung ist aber nur noch „halb offen“. Eine ähnliche Situation tritt auf, wenn der Netzwerkpfad zwischen den Partnern fehlerhaft ist. In diesem Fall halten beide Partner ihre Verbindung aufrecht, obwohl das Netzwerk dazwischen nicht mehr funktioniert.

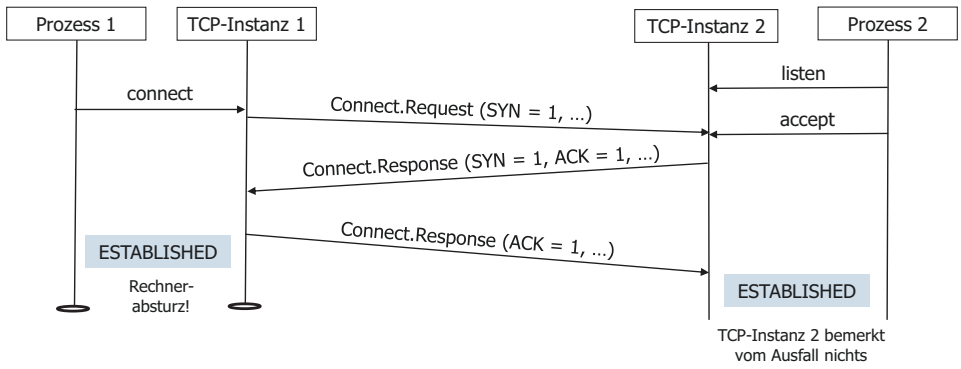


Abb. 3.33 Ausfallsituation während einer TCP-Verbindung

Durch Einsatz des Keepalive-Mechanismus kann bei TCP zyklisch überprüft werden, ob der jeweilige Partner noch erreichbar ist. Nach Ablauf des Keepalive Timers wird eine Keepalive-Nachricht an den Partner gesendet. Antwortet dieser nach mehrmaligem Wiederholen nicht, kann die Verbindung abgebaut werden.

Der Keepalive-Mechanismus ist bei TCP optional. Er ist standardmäßig ausgeschaltet. Wird der Mechanismus benutzt, wird der Keepalive Timer zunächst auf zwei Stunden eingestellt und kann in der Regel per Betriebssystemkonfiguration verändert werden. Zudem ist das *Keepalive-Intervall* zwischen zwei Keepalive-Nachrichten einzustellen. Diese Zeit wird abgewartet, bevor eine erneute Keepalive-Nachricht gesendet wird, sofern keine Antwort kommt. Über einen Implementierungsparameter kann in der Regel die Anzahl der maximalen Versuche angegeben werden, bevor eine Verbindung endgültig abgebaut wird.

TCP-Keepalive-Mechanismus unter Linux und Windows

Linux unterstützt den TCP-Keepalive-Mechanismus und erlaubt die Einstellung von drei Parametern über die Kernel-Konfiguration. Folgende Kernel-Parameter können konfiguriert werden: *tcp_keepalive_time*, *tcp_keepalive_intvl* und *tcp_keepalive_probes* (siehe auch Anhang).

Ähnliche Einstellungsmöglichkeiten gibt es unter Windows, beispielsweise über das Windows-Registry. Die Parameter können mit Hilfe eines Registry-Editors verändert werden. Die Parameter haben die Bezeichnungen *KeepAliveTime* und *KeepAliveInterval*. Ein Parameter für die Wiederholungen ist nicht definiert.

Die eigentliche Nutzung des Keepalive Timers wird schließlich je Verbindung über die Angabe der Socket-Option SO_KEEPALIVE vorgenommen (siehe Socket-Optionen).

3.7.4 Time-Wait Timer

Der *Time-Wait Timer* wird nach RFC 793 für den Verbindungsabbau auf der aktiven Partnerseite benötigt und läuft standardmäßig über die doppelte Paketlebensdauer, die ursprünglich auf 120 Sekunden eingestellt wurde. Der Timer soll sicherstellen, dass alle sich noch im Netz befindlichen TCP-Segmente einer Verbindung noch empfangen werden, bevor der endgültige Verbindungsabbau durchgeführt wird. Die aktiv verbindungsabbauende TCP-Instanz verbleibt bis zum Timerende im Zustand TIME_WAIT.

Im RFC 6191 werden Möglichkeiten erläutert, um über die Option *TCP Timestamps* die Wartezeit im TIME_WAIT-Zustand zu verringern, da mit exakteren Round-Trip-Zeiten eine Vorhersage besser machbar ist.

Time-Wait Timer unter Linux und Windows

Unter Linux gibt es keine Einstellungsmöglichkeit für den Time-Wait Timer. Er ist in einer Konstante im Sourcecode festgelegt.

Über den Parameter *tcp_tw_reuse* kann unter Linux ein schnelles Recycling von Sockets, die im TIME_WAIT-Zustand sind, eingestellt werden, was allerdings mit Vorsicht zu genießen ist.

Unter Windows kann der Time-Wait Timer über den Parameter *TcpTimedWaitDelay* verändert werden (siehe Anhang).

Lingering
Über eine Socket-Option (SO_LINGER) kann auch festgelegt werden, ob Daten, die bei einem *close*-Aufruf im Sendepuffer der TCP-Verbindung sind, noch vollständig gesendet werden, bevor der synchrone (blockierende) *close*-Aufruf zurückkehrt. Alternativ kann der *close*-Aufruf sofort zurückkehren und das Senden im Hintergrund weitergeführt werden. Der Sendepuffer sollte aber in jedem Fall noch vollständig geleert werden. In manchen TCP- bzw. Socket-Implementierungen (beispielsweise in der Java-Socket-Implementierung) kann man auch einstellen, dass der *close*-Aufruf sofort zurückkehrt und die Verbindung zurücksetzt (TCP-Segment mit RST-Flag). Es ist allerdings nicht gesichert, dass diese Implementierungen ordnungsgemäß funktionieren.

3.7.5 Close-Wait Timer

Der *Close-Wait Timer* gibt die Zeit an, die eine passive TCP-Instanz maximal wartet, bis ein Anwendungsprozess aktiv einen *close*-Aufruf absetzt, um seine Verbindungsseite abzubauen.

Wie bereits bei der Diskussion der TCP-Zustandsautomaten erörtert, ist es die Aufgabe der Anwendungsschicht, den Abbau der TCP-Verbindung ordentlich durchzuführen. Sollte die Anwendung dies aber nicht machen, ist eine Begrenzung der Wartezeit wichtig, um eine halb offene Verbindung schließlich endgültig abbauen zu können.

Leider ist in der TCP-Spezifikation nicht explizit definiert, dass es einen derartigen Timer gibt, weshalb es den TCP-Implementierungen in den Betriebssystemen überlassen bleibt, Timer zu nutzen und diese auch per Konfiguration zu verändern.

3.7.6 Persistence Timer

Der *Persistence Timer* verhindert ein ewiges Warten eines TCP-Senders, wenn der TCP-Empfänger die Fenstergröße auf 0 setzt. Wenn eine TCP-Instanz ein TCP-Segment empfängt und daraufhin eine Bestätigung mit einer Fenstergröße von 0 zurücksendet, muss die Partner-Instanz warten, bis sie wieder ein TCP-Segment mit einer Fenstergröße > 0 empfängt, um erneut senden zu können. Er hat nämlich gemäß Sliding-Window-Mechanismus keinen Sendekredit. Wenn die empfangende TCP-Instanz dieses Segment zwar sendet, dieses aber nicht ankommt, käme die sendende TCP-Instanz nicht mehr aus dem Wartezustand heraus.

Damit nicht ewig gewartet werden muss, wird beim Empfang einer Fenstergröße von 0 der Persistence Timer aktiviert. Bei Ablauf des Persistence-Timers wird über die Verbindung ein TCP-Segment mit einem Byte an Nutzdaten gesendet. Die empfangende TCP-Instanz kann als Antwort die aktuelle Fenstergröße übermitteln. Auf diese Weise wird der Wartezustand der sendenden TCP-Instanz wieder irgendwann verlassen.

3.8 TCP-Sicherheit

In der TCP-Spezifikation selbst sind keinerlei Sicherheitsmechanismen im Sinne der Informationssicherheit (Vertraulichkeit, Verfügbarkeit und Integrität) vorgesehen. Die Gewährleistung einer sicheren und vertrauenswürdigen Übertragung wird den Protokollen in der Anwendungsschicht oder in der Vermittlungsschicht überlassen. Es sind verschiedene

Angriffsmöglichkeiten bekannt, die direkt in der TCP-Ebene ansetzen. Mögliche Angriffstechniken sind das SYN-Flooding, der Sequenznummernangriff und das Session Hijacking.

- Beim *SYN-Flooding* sendet ein Angreifer in rascher Folge Connect-Requests mit SYN-Flag=1 und gefälschten Quelladressen an einen passiven TCP-Partner (meist einen Server), um viele Verbindungsaufbauwünsche vorzutäuschen und damit einen TCP-Server stark zu beeinträchtigen. Dessen Connect-Response-Segmente werden vom Angreifer nicht behandelt. Der angegriffene TCP-Server baut jeweils Verbindungskontexte mit halb offenen Verbindungen auf und muss diese auch eine gewisse Zeit aufbewahren, wodurch Ressourcen (Speicher, TCP-Verbindungen, Ports) belegt werden. Er bleibt also einige Zeit im Zustand SYN_RCVD. Reguläre TCP-Segmente kommen dann nicht mehr durch. Es handelt sich bei dieser Angriffstechnik um eine sogenannte Denial-Of-Service-Attacke (DDoS). Heutige TCP-Implementierungen nutzen Timer, um die belegten Ressourcen nach einer festgelegten Zeit wieder freizugeben. Eine andere Möglichkeit ist es, bei einer bestimmten Anzahl an halb offenen Verbindungen nur noch minimale Informationen für eine Verbindungsaufbauwunsch aufzubewahren, um Ressourcen einzusparen. Dadurch können die Auswirkungen eines SYN-Flooding-Angriffs zumindest abgemildert werden. Gegenmaßnahmen wie SYN-Cookies, RST-Cookies, SYN-Proxy und SYN-Cache, die nicht weiter ausgeführt werden sollen, sind in RFC 4987 (TCP SYN Flooding Attacks and Common Mitigations) dargestellt.
- Beim *Sequenznummernangriff* versucht ein Angreifer, fremde TCP-Segmente in eine bereits aufgebaute Verbindung einzuschleusen. Zwar wird einem Angreifer das Leben aufgrund des TCP-Bestätigungsverfahrens erschwert, weil die aktuellen Sequenz- und Betätigungsnummern bekannt sein müssen. Er muss die aktuellen Sequenznummern der Verbindung herausfinden und nutzt dabei eine Schwäche mancher TCP-Implementierungen. Die initialen Sequenznummern einer Verbindunung werden nach RFC 793 ermittelt. Manchmal wird aber die Implementierung auch noch vereinfacht und die Sequenznummern beginnen nicht zufällig. Dann kann ein Angreifer die initialien Sequenznummern auch noch leicht herausfinden. Als Gegenmaßnahmen gegen Sequenznummernagriffe dienen beispielsweise entsprechend konfiguriere Paketfilter, die die Angriffe erkennen, oder natürlich TCP-Implementierungen, die die initialen Sequenznummern über einen Zufallsgenerator ermitteln.
- Beim *Session Hijacking* versucht ein Angreifer, bestehende TCP-Verbindungen für kurze Zeit zu übernehmen. Er tut so, als ob er der echte Partner wäre. Dadurch kann er TCP-Segmente unbemerkt einschleusen und die Sequenznummernfolge zerstören. Nachdem sich der Angreifer wieder entfernt hat, ist die Kommunikation der echten Partner zerstört, weil eine Synchronisation der Sequenznummern nicht mehr möglich ist. Als Gegenmaßnahme dient zum Beispiel die Verschlüsselung der übertragenen Segmente, da in diesem Fall eingeschleuste Nachrichten erkannt werden.

Um verteilte TCP-Anwendungen sicherer zu gestalten, sind weitere Protokolle wie TLS (Transport Layer Security), IPsec und IPv6 sinnvoll. Auch eine entsprechende Firewall-/Paketfilter-Konfiguration kann derartige Angriffe abmildern (Eckert 2014).

Literatur

Eckert, C. (2014) IT-Sicherheit Konzepte – Verfahren – Protokolle, Oldenbourg Wissenschaftsverlag, 2014

Herold, H.; Lurz, B.; Wohlrab, J. (2012) Grundlagen der Informatik, 2 aktualisierte Auflage, 2012

IEEE POSIX (2016) The Open Group Base Specifications Issue 7, IEEE Std 1003.1™-2008, 2016 Edition, http://pubs.opengroup.org/onlinepubs/9699919799/, letzter Zugriff am 10.09.2017

Mandl, P.; Bakomenko A.; Weiß, J. (2010) Grundkurs Datenkommunikation: TCP/IP-basierte Kommunikation: Grundlagen, Konzepte und Standards, 2. Auflage, Vieweg-Teubner Verlag, 2010

Sangtae H.; Injong R.; Lisong X. (2008) CUBIC: A New TCP-Friendly High-Speed TCP Variant, ACM SIGOPS Operating System Review, Volume 42, Issue 5, July 2008, Page(s):64–74, 2008

Sangtae H.; Injong R.; Lisong X. (2007) Impact of Background Traffic on Performance of High-speed TCP Variant Protocols, Computer Networks: The International Journal of Computer and Telecommunications Networking, Volume 15, Issue 4, Aug. 2007, Page(s):852–865, 2007

Tanenbaum, A. S.; Wetherall, D. J. (2011) Computer Networks, Fifth Edition, Pearson Education, 2011

UDP-Konzepte und -Protokollmechanismen

4

Zusammenfassung

UDP ist im Gegensatz zu TCP ein Transportprotokoll für die verbindungslose und ungesicherte Kommunikation. Wenn es also für eine Anwendung akzeptabel ist, dass gelegentlich Nachrichten verloren gehen können, ohne dass dies schlimmere Konsistenzprobleme verursacht, dann ist UDP möglicherweise eine bessere Wahl als TCP. Beispielsweise ist es günstiger, einen Audiostrom über UDP zu übertragen als über TCP, weil es für das menschliche Ohr einfacher zu verarbeiten ist, wenn eine kleine Sequenz fehlt, als wenn ein ständiges Nachfordern fehlender Daten die Audioqualität stark durch „Ruckeln" beeinträchtigen würde. Die wichtigste UDP-Funktionalität ist, das Multiplexing mehrerer UDP-Kommunikationsendpunkte über der IP-Schicht ohne zusätzliche Sicherheitsmechanismen einzuführen. Außer einer Fehlererkennungsfunktion wird sonst zur Best-Effort-Übertragung des Internet-Protokolls nichts hinzugefügt. Ein weiterer Vorteil, der sich durch die UDP-Nutzung ergibt, ist die Möglichkeit, Broadcast- und Multicast-Nachrichten zu senden und zu empfangen. Die Besonderheiten des Transportprotokolls UDP werden im Weiteren diskutiert.

4.1 Übersicht über grundlegende Konzepte und Funktionen

4.1.1 Grundlegende Aufgaben von UDP

Im Gegensatz zu TCP ist UDP ein unzuverlässiges, verbindungsloses Transportprotokoll. Dieses Transportprotokoll hat sich seit seiner ersten Spezifikation kaum verändert und ist wie folgt charakterisiert:

- Die Nachrichtenübertragung erfolgt zwischen UDP-Kommunikationsendpunkten.
- Es werden keine Empfangsbestätigungen für Pakete gesendet.

P. Mandl, *TCP und UDP Internals*, https://doi.org/10.1007/978-3-658-20149-4_4

- UDP-Segmente, die bei UDP als Datagramme bezeichnet werden, können jederzeit verloren gehen.
- Eingehende Pakete werden nicht in der Reihenfolge sortiert, in der sie vom Sender gesendet wurden.
- Es gibt keine Fluss- und auch keine Staukontrolle.

Maßnahmen zur Erhöhung der Zuverlässigkeit, wie Bestätigungen, Timerüberwachung und Übertragungswiederholung, müssen bei Bedarf im darüberliegenden Anwendungsprotokoll ergriffen werden. Lediglich die Segmentierung und auch ein Multiplexieren mehrerer UDP-Kommunikationsbeziehungen über eine darunterliegende IP-Instanz wird von UDP übernommen. Trotzdem bietet UDP – richtig eingesetzt – einige Vorteile:

- Bei UDP ist keine explizite Verbindungsaufbau-Phase erforderlich und entsprechend auch kein Verbindungsabbau. UDP-basierte Anwendungsprotokolle sind recht einfach zu implementieren. UDP muss keine Kontextverwaltung durchführen, sondern arbeitet im Wesentlichen zustandslos. Da es keine Verbindungen gibt, braucht man für jeden Kommunikationsprozess genau einen Port (hier ein UDP-Port). Ein Anwendungsprozess erzeugt einen UDP-Socket und kann Nachrichten senden und empfangen.
- Man kann unter Umständen eine bessere Leistung erzielen, aber nur, wenn TCP im Anwendungsprotokoll nicht nachgebaut werden muss, sondern eine gewisse Unzuverlässigkeit in Kauf genommen werden kann.
- Mit UDP kann auch Multicasting und Broadcasting genutzt werden, d. h. eine Gruppe von Anwendungsprozessen kann mit wesentlich weniger Nachrichtenverkehr kommunizieren als bei Nutzung einzelner Punkt-zu-Punkt-Verbindungen. Beispielsweise kann man mit Multicast/Broadcast eine Nachricht, die einen Produzenten-Prozess erzeugt, mit einer einzigen UDP-PDU an mehrere Partner senden.
- Die Senderate wird bei UDP nicht wie bei TCP gedrosselt (siehe Staukontrolle), wenn eine Netzüberlast vorliegt. Es gehen dann zwar eventuell Nachrichten verloren, aber für Audio- und Videoströme oder sonstige Multimedia-Anwendungen kann dies günstiger sein.[1]

4.1.2 Adressierung

Ein UDP-SAP ist wie bei TCP durch ein Socket adressierbar, dessen Adresse aus dem Tupel von Internetadresse und einem UDP-Port gebildet wird. Der Wertebereich der UDP-Ports ist unabhängig vom TCP-Portbereich und geht von 0 bis 65535. Zu den wichtigen UDP-Portnummern, die auch als Well-known Ports definiert sind, gehören u. a.

- 53, DNS (Domain Name Service),
- 69, TFTP (Trivial File Transfer Protocol),

[1] Wie bereits erläutert, tolerieren diese Anwendungen einen gewissen Datenverlust und verkraften diesen eher als zu große und unkontrollierbare Verzögerungen und Verzögerungsschwankungen (Jitter).

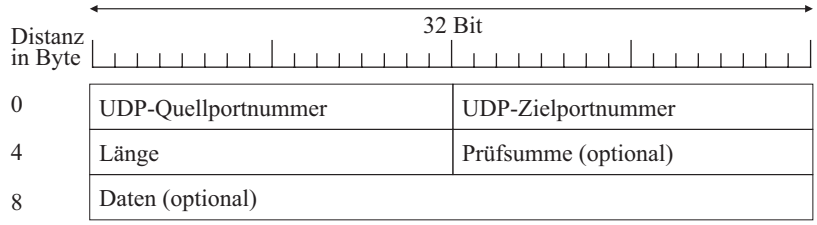


Abb. 4.1 UDP-Steuerinformation

- 161, SNMP (Simple Network Management Protocol),
- 520, RIP (Routing Information Protocol).

Im Gegensatz zu TCP bietet eine UDP-Instanz an der Schnittstelle zum Anwendungsprozess auch keinen Stream-Mechanismus. Datagramme werden in festen Blöcken gesendet und bei der empfangenden Anwendung als Blöcke entgegengenommen.

4.1.3 UDP-Steuerinformation

Der UDP-Header (Abb. 4.1) ist verglichen mit dem TCP-Header wesentlich einfacher und besteht nur aus acht Bytes. Neben der Adressierungsinformation enthält er noch die variable Länge des UDP-Datagramms sowie eine optionale Prüfsumme.

Die Felder des UDP-Headers enthalten folgende Informationen:

- *UDP-Quell-Portnummer*: Nummer des sendenden Ports.
- *UDP-Ziel-Portnummer*: Nummer des empfangenden Ports, also des adressierten Partners.
- *Länge*: Hier wird die Größe des UDP-Segments inklusive des Headers in Bytes angegeben. Dies ist notwendig, da UDP-Segmente keine fixe Länge aufweisen.
- *Prüfsumme* (optional): Die Prüfsumme ist optional und prüft das ganze UDP-Segment (UDP-Header + Nutzdaten) in Verbindung mit einem Pseudoheader. Die Berechnung der Prüfsumme wird weiter unten erläutert.
- *Daten*: Nutzdaten des Datagramms.

4.1.4 Multiplexing und Demultiplexing

UDP erweitert die Vermittlungsschicht durch eine Multiplexing/Demultiplexing-Funktionalität. Damit ist Folgendes gemeint:

- Auf einem Rechner können Anwendungsprozesse jeweils einen oder mehrere Kommunikationsendpunkte nutzen, für die jeweils ein eigener UDP-Ports belegt wird. Sendet ein Anwendungsprozess über einen UDP-Port eine Nachricht, wird es von der lokalen

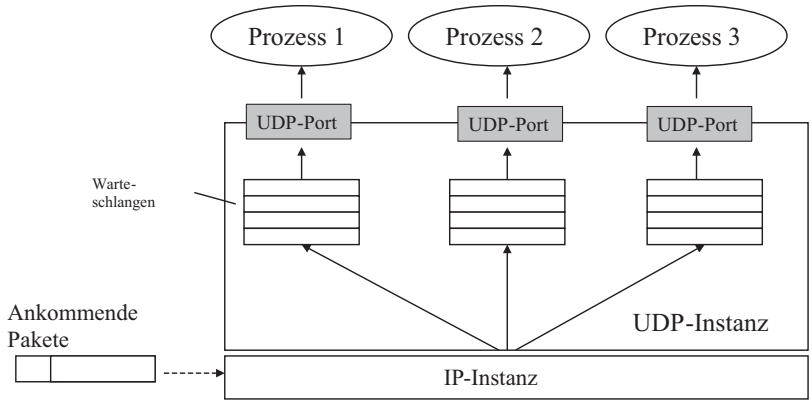


Abb. 4.2 UDP-Demultiplexing

UDP-Instanz übernommen, in ein UDP-Segment eingebettet und an die darunterliegende IP-Instanz weitergegeben. Auch Kommunikationsendpunkte anderer Anwendungen werden über dieselbe IP-Instanz weiterverarbeitet. Die UDP-Instanz führt also ein Multiplexing mehrerer UDP-Kommunikationsendpunkte über einen IP-Endpunkt durch.

- Bei der empfangenden UDP-Instanz ist es umgekehrt und es wird ein Demultiplexing durchgeführt. Ankommende UDP-Segmente werden dort geprüft und anhand der UDP-Portnummern den zuständigen Anwendungsprozessen übergeben. Dieser Vorgang ist in Abb. 4.2 illustriert. Ein ankommendes IP-Paket wird von der IP-Instanz analysiert. Das eingebettete UDP-Segment wird inklusive der UDP-Steuerinformation an die UDP-Instanz weitergegeben. Diese prüft, an welchen Endpunkt das UDP-Segment weitergeleitet werden soll und übergibt es schließlich in die Ankunftswarteschlange (Empfangspuffer) des entsprechenden UDP-Kommunikationsendpunkts, sofern die Prüfsumme fehlerfrei war.

4.2 Datenübertragungsphase

4.2.1 Datagrammorientierte Kommunikation

Die Datenübertragung ist in UDP relativ einfach. Eine UDP-basierte Kommunikationsanwendung muss nur dafür sorgen, dass die Kommunikationsprozesse ihre Ports untereinander kennen, über die Socket-Schnittstelle jeweils ein UDP-Socket erzeugen und dann können sie sich schon Datagramme zusenden. Das Senden der Daten erfolgt dabei völlig unabhängig voneinander. Das darüberliegende Anwendungsprotokoll muss auf der Senderseite die Nachrichten zusammenstellen bzw. auf der Empängerseite interpretieren.

Die UDP-Instanz wickelt bei Bedarf eine Segmentierung/Desegmentierung ab, aber übernimmt zusätzlich zur Übertragung der Datagramme keine weiteren Maßnahmen.

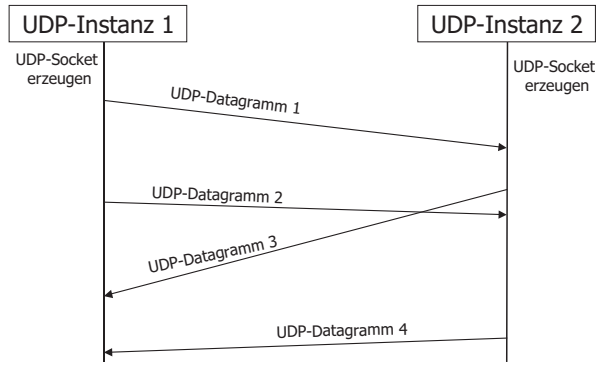


Abb. 4.3 Einfache Kommunikation über UDP

In Abb. 4.3 ist eine typische Kommunikation zweier über UDP kommunizierender Transportinstanzen dargestellt, die sich gegenseitig wahlfrei sich überlappende Datagramme (UDP-Segmente) zusenden. Die gesamte Protokolllogik liegt in den Anwendungs-PDUs des übergeordneten Protokolls. Die beiden in der Abbildung kommunizierenden Prozesse senden jeweils zwei Daten-PDUs zu ihrem jeweiligen Kommunikationspartner. Bestätigungen werden von der UDP-Instanz nicht versendet. Diese müssen bei Bedarf in den Anwendungs-PDUs versendet werden. Aufgrund des einfachen Protokolls wird an dieser Stelle auf die Beschreibung des Protokollautomaten verzichtet.

Eine UDP-Instanz kann unter Nutzung von IP-Broadcasting und IP-Multicasting auch UDP-Segmente an alle Rechner eines IP-Subnetzes (Broadcast) oder an eine bestimmte Gruppe von Rechnern (Multicast) senden. Hierfür müssen auf der IP-Ebene spezielle Adressen (IP-Broadcast- und IP-Multicastadressen) verwendet werden. Ob tatsächlich in der Netzwerkzugriffsschicht nur ein Frame gesendet wird, hängt von der vorhandenen Netzwerktechnologie ab. Bei ethernetbasierten Netzen ist dies möglich.

4.2.2 Prüfsummenberechnung

UDP nutzt für die Überprüfung empfangener UDP-Segmente auf Übertragungsfehler ein einfaches Prüfsummenverfahren, das in den RFCs 1071, 1141 und 1624 erläutert ist. Dasselbe Verfahren wird auch bei TCP und auch bei IP (nur für den IP-Header) verwendet, bei UDP ist es allerdings optional.

Das Verfahren sieht vor, dass die Prüfsumme über das ganze UDP-Segment und zusätzlich über einen sogenannten Pseudoheader ermittelt wird. Die entstehende Summe wird ins Prüfsummenfeld des UDP-Headers eingetragen. Dabei werden alle 16-Bit-Wörter beim Sender unter Berücksichtigung von Überträgen addiert und die Summe in das sogenannte „Einerkomplement“ umgewandelt.

Der Pseudoheader ist ein künstlich konstruierter Header, der zwar in die Berechnung der Prüfsumme mit einfließt, jedoch nicht zusammen mit dem UDP-Segment verschickt

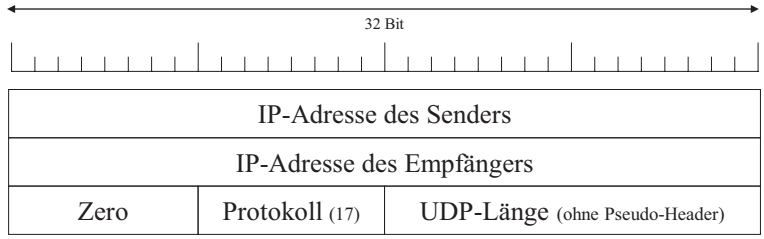


Abb. 4.4 UDP-Pseudoheader

wird. Er enthält die IP-Adressen des Senders und des Empfängers, ein Nullbyte, den in IP verwendeten Protokollcode für UDP (immer der Wert „17“) und die Gesamtlänge des UDP-Segments, also Länge des UDP-Headers einschließlich der Länge der UDP-Nutzdaten. Der Aufbau des Pseudoheaders ist in Abb. 4.4 dargestellt.

Das Verfahren funktioniert auf der Senderseite im Detail wie folgt (siehe hierzu das Beispiel zur Berechnung einer UDP-Prüfsumme):

- Der Sender erzeugt den Pseudoheader und setzt ihn vor das zu versendende UDP-Segment. Ein Auffüllen des UDP-Segments mit binären Nullen bis zu einer 16-Bit-Wortgrenze wird durchgeführt.
- Das Prüfsummenfeld des UDP-Headers wird mit binären Nullen belegt.
- Danach wird das ganze UDP-Segment inklusive des Pseudoheaders in 16-Bit-Wörter aufgeteilt.
- Alle 16-Bit-Wörter werden nacheinander unter Berücksichtigung möglicher Überlaufbits (Carry-out Bits) addiert. Diese Addition wird gelegentlich auch als Einerkomplement-Addition bezeichnet. Ganz korrekt ist die Bezeichnung bei diesem Verfahren nicht, da beim Einerkomplement der Wertebereich nicht überschritten werden darf, beim UDP-Prüfsummenverfahren aber schon.
- Die Summe wird dann in das Einerkomplement umgewandelt. Falls das Ergebnis nur aus binären Nullen besteht, wird es zu 0xFFFF konvertiert.
- Das Ergebnis der Umwandlung wird schließlich im UDP-Header in das Feld *Prüfsumme* eingetragen.
- Das UDP-Segment wird dann ohne den Pseudoheader übertragen.

Der Empfänger muss für jedes ankommende UDP-Segment, das eine Prüfsumme enthält, Folgendes unternehmen:

- Die IP-Adressen werden aus dem ankommenden IP-Paket gelesen, um ebenfalls einen Pseudoheader zu konstruieren.
- Anschließend wird das UDP-Segment mit dem ermittelten Pseudoheader konkateniert (Pseudoheader ist vorne). Die Prüfsumme wird nicht aus dem empfangenen UDP-Segment entfernt und es wird wie beim Sender die Prüfsumme erneut berechnet.

- Wenn nach der Einerkomplementbildung als Summe der 16-Bit-Wörter als Ergebnis 0x0000 herauskommt, ist die Übertragung gut gegangen und das UDP-Segment ist auch beim richtigen Zielrechner angekommen. Das Segment wird ausgeliefert. Im anderen Fall ist vermutlich ein Übertragungsfehler aufgetreten und das UDP-Segment wird verworfen.

Das Verfahren hilft also auch, fehlgeleitete UDP-Segmente zu erkennen. Denn wenn das Paket nicht für den empfangenden Rechner bestimmt war, stimmt der im Zielrechner ermittelte Pseudoheader nicht mit dem in die Berechnung beim sendenden Rechner eingegangenen Pseudoheader überein. Eine Fehlerbehebung findet aber nicht statt. Ein fehlerhaftes Paket wird nicht an den Anwendungsprozess weitergegeben, sondern in den meisten Implementierungen verworfen.[2]

Beispiel zur Berechnung einer UDP-Prüfsumme

Ein UDP-Segment enthält beispielsweise, stark vereinfacht und unter Vernachlässigung der Mindestlänge, nur vier 16-Bit-Wörter

1000 0110 0101 1110 1010 1100 0110 0000 0111 0001 0010 1010 1000 0001 1011 0101

Für dieses Segment soll eine UDP-Prüfsumme berechnet werden:

1000 0110 0101 1110 (1)
1010 1100 0110 0000 (2)
0111 0001 0010 1010 (3)
1000 0001 1011 0101 (4)

Berechnung:

1000 0110 0101 1110 (1)
1010 1100 0110 0000 (2) +
0011 0010 1011 1110 → 1 (Übertrag)
1
0011 0010 1011 1111
0111 0001 0010 1010 (3) +
1010 0011 1110 1001 (Kein Übertrag)
1000 0001 1011 0101 (4) +
0010 0101 1001 1110 → 1 (Übertrag)
1
0010 0101 1001 1111 = Summe (0x259F)
1101 1010 0110 0000 (Einerkomplement der Summe) = 0xCA60
0b1101100101100000 = 0xCA60 ist die Prüfsumme, die in des Prüfsummenfeld des UDP -Headers eingetragen wird.

[2] Manche UDP-Implementierungen scheinen ein fehlerhaftes UDP-Datagramm auch an den Anwendungsprozess weiterzugeben und eine Warnung zu ergänzen.

Es stellt sich die Frage, welche Fehler mit dem Verfahren gefunden werden können. Einbit-Fehler werden mit dem Verfahren erkannt, im Vergleich zu einfachen Paritätsprüfverfahren ist das UDP-Verfahren aber relativ schwach (siehe hierzu das Beispiel zur Fehlererkennung). Zweifache und dreifache Bitfehler werden nicht erkannt. Das Verfahren sollte eigentlich ursprünglich nach einer Prüfphase durch ein CRC-Verfahren (Cyclic-Redundancy-Check-Verfahren, wie bei Ethernet) ersetzt werden. Dazu kam es aber nie.

Fehlererkennung beim UDP-Prüfverfahren: Zweifache und dreifache Bitfehler

Im folgenden Beispiel werden zwei Bits bei der Übertragung vertauscht. Das Ergebnis der Addition zeigt keinen Unterschied zum fehlerfreien Fall. Zweibit-Fehler werden also nicht erkannt.

0010	00**0**0 (1 Bit vertauscht)
0001 +	00**1**1 (1 Bit vertauscht) +
0011	0011

Ebenso werden dreifache Bitfehler nicht erkannt, wie das folgende Beispiel zeigt:

0011	00**0**1 (1 Bit vertauscht)
0010	001**1** (1 Bit vertauscht)
0000 +	000**1** (1 Bit vertauscht) +
0101	0101

CRC-Verfahren

Beim Cyclic-Redundancy-Check (CRC) oder zyklischem Redundanzcode handelt es sich um ein Verfahren zur Fehlererkennung bei k redundanten Bits in einer n-Bit-Nachricht, das auch dann sehr nützlich ist, wenn $k \ll n$. Ethernet mit 1500-Byte-Frames = 12000 Bit nutzt beispielsweise einen 32-Bit-langen CRC-Code (n = 12000, k = 32).

CRC basiert auf sogenannten zyklischen Codes. Zyklisch bedeutet, dass gültige Codewörter solche sind, die sich in einem Schieberegister durch links hinausschieben und rechts hineinschieben ergeben. Die Nachrichten werden in einzelne Wörter (z. B. mit einer Länge von 32 Bit) zerlegt, die als Polynome betrachtet und gemäß Modulo-2-Arithmetik addiert werden.

Die Prüfsumme wird bei UDP vom Sender mit 0x0000 belegt, um anzuzeigen, dass keine gültige Prüfsumme berechnet wurde. In IPv4-Netzen kann die UDP-Prüfsummenberechnung nämlich über eine Socket-Optionen für ein UDP-Socket ausgeschaltet werden (siehe Option SO_NO_CHECK).

4.3 UDP-Sicherheit

Wie in der TCP-Spezifikation sind auch in der UDP-Spezifikation keinerlei Sicherheitsmechanismen im Sinne der Informationssicherheit (Vertraulichkeit, Verfügbarkeit und Integrität) vorgesehen. UDP-Kommunikation ist daher als nicht vertrauenswürdig einzustufen.

UDP ist sogar noch leichter anzugreifen, weil es zudem nicht über einen Bestätigungsmechanismus verfügt und daher ein Angreifer auch keine Sequenzunummern herausfinden muss. Ebenso fehlt ein Handshake für den Verbindungsaufbau, der das Einmischen eines Angreifer erschweren könnte. Es müssen lediglich Absenderadressen gefälscht werden, um einen Kommunikationspartner vorzutäuschen. Man spricht hier von *UDP-Spoofing*.

Gegenmaßnahmen sind die Nutzung von entsprechend konfigurierten Paketfiltern, die auch in Firewalls platziert sein können. Paketfilter können zum Beispiel alle UDP-Pakete mit bestimmten Zielports sperren. Zudem kommt man an einer Authentifikation und Autorisierung der Nutzer nicht vorbei. Hierzu sind aber weitere Protokolle oder Lösungen in der Anwendungsschicht notwendig, die eine sichere und vertrauenswürdige Übertragung gewährleisten. UDP ist ohne weitere Sicherheitsmaßnahmen also zu vermeiden und es sollten auf Rechnern generell nur sehr kontrollierte UDP-Ports geöffnet sein.

Programmierung von TCP/UDP-Anwendungen

5

Zusammenfassung

Viele Kommunikationsanwendungen werden heute auf Basis der Socket API entwickelt. Höhere Kommunikationsmechanismen wie Remote Procedure Call (RPC), Remote Method Invocation (RMI) oder Message Passing nutzen ebenfalls überwiegend die Socket API, die ursprünglich nur in der Programmiersprache C implementiert war, heute aber in allen gängigen Programmiersprachen zur Verfügung steht. In Java besteht eine komfortable Nutzungsmöglichkeit, die es gestattet, ganze Java-Objekte als Nachricht zu übertragen. Das Java-Laufzeitsystem übernimmt auch die Serialisierung und Deserialisierung der Objekte bei TCP-Sockets. Ebenso können Datagramm-Sockets und Multicast-Sockets auf Basis von UDP genutzt werden. Serveranwendungen, die sehr viele Anfragen von Clients pro Zeiteinheit bearbeiten müssen, können diese Anfragen nicht seriell in einem Thread bearbeiten, sondern implementieren in der Regel Mutithreading, wobei es mehrere Varianten gibt. Entweder wird jedem Client serverseitig ein eigener Workerthread zugewiesen oder es werden einzelne Client-Requests zur Bearbeitung an Workerthreads übergeben. Für Hochleistungsanwendungen ist die feingranularere Zuordnung eines Client-Requests an einen Workerthread effizienter. Die Threads werden in einem Threadpool verwaltet und ankommende Requests werden freien Threads zugeordnet. Damit ein Server hier effizient arbeiten kann, werden häufig über der Socket API liegende Frameworks bzw. APIs wie *nio* und *netty* verwendet, die auch ein effizientes, nicht blockierendes Warten auf ankommende Requests ermöglichen. Dies erfolgt unter Zuhilfenahme der speziellen Systemfunktion *select,* mit der die Programmierung eines zentralen Ereigniswartepunkts für viele Ereignisquellen wie TCP-Verbindungen realisiert werden kann. Im Folgenden wird die Programmierung von socketbasierten Anwendungen erläutert. Insbesondere wird die Socket API in Java besprochen. Programmierhilfen für einfachere Anwendungen auf Basis von TCP- und UDP-Sockets werden aufgezeigt, ein einfaches Framwork zur Erleichterung der Programmierung wird vorgestellt und die Programmierung von Servern für sehr leistungsstarke Anwendungen wird andiskutiert.

P. Mandl, *TCP und UDP Internals*, https://doi.org/10.1007/978-3-658-20149-4_5

5.1 Überblick

Die Entwicklung von Kommunikationsanwendungen ist in den vergangenen dreißig Jahren aufgrund neuer Technologien immer komfortabler geworden. Vor nicht allzu langer Zeit musste man sich bei der Programmierung noch mit allen Aspekten (auch der niedrigen Schichten) der Kommunikation befassen. Heute gibt es Programmiermodelle, die dies wesentlich erleichtern und man setzt mindestens auf eine vernünftige Transportzugriffsschicht auf. Für die Entwicklung von verteilten Anwendungen kann man also meistens einen funktionierenden Transportdienst nutzen.

Eine Methode der Programmierung basiert auf der Socket-Schnittstelle. Dies ist eine Transportzugriffsschnittstelle, die in mehreren Programmiersprachen implementiert ist und aus der Unix-Welt kommt. Moderne Programmiersprachen wie Java, Python und C# stellen eine komfortable Nutzungsmöglichkeit über vordefinierte Packages (Java) oder Namespaces (C#) bereit. Sockets sind die Basis für alle im Internet vorhandenen höheren Programmiermodelle. Weiter fortgeschritten sind Konzepte wie RPC (Remote Procedure Call) und objektorientierte Kommunikationsmechanismen wie Java RMI.

Die Art und Weise, wie man eine verteilte Anwendung programmiert, hängt von den Anforderungen ab. Die meisten verteilten Anwendungen stützen sich heute auf das Client-/Server-Modell, in dem ein Serverprozess oder gegebenenfalls mehrere Serverprozesse auf Requests von Clients warten und diese beantworten (z. B. Filetransfer und Webanwendungen). In diesem Fall geht die Kommunikation immer vom Client aus. Aber auch andere Kommunikationsparadigmen sind häufig anzutreffen. Ein anderes Modell ist die sogenannte Peer-to-Peer-Kommunikation (Beispiel: Bitcoin), in der Anwendungsprozesse gleichberechtigt miteinander kommunizieren. Beispielsweise benötigt man in Automatisierungsanwendungen meist eine gleichberechtigte Kommunikation, in der jeder Partner zu jeder Zeit eine Nachricht (oft Telegramm) senden kann, wenn ein bestimmtes Ereignis (wie z. B. „eine Palette ist an einem bestimmten Meldepunkt zum Einlagern in ein Hochregal bereit“) eintritt. Weiterführende Kommunikationsmechanismen befassen sich mit gesicherten Message-Queues, transaktionsgesicherter Kommunikation mehrerer Objekte usw. In diesem Kapitel sollen die Grundlagen der Programmierung von verteilten Anwendungen am Beispiel der Socket API aufgezeigt werden.

5.2 Grundkonzepte der Socket-Programmierung

5.2.1 Einführung und Programmiermodell

Die Socket-Schnittstelle ist eine klassische Transportzugriffsschnittstelle. Sie stellt ein API bereit, mit der man Kommunikationsanwendungen entwickeln kann. Die Socket-Schnittstelle ist heute der De-facto-Standard für die Programmierung von TCP- und UDP-Anwendungen und in allen gängigen Betriebssystemen und Programmiersprachen verfügbar.

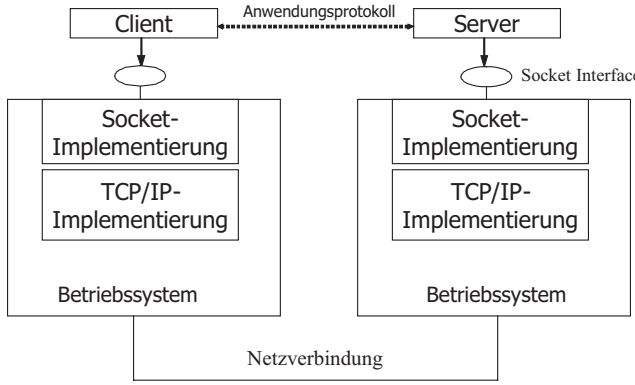


Abb. 5.1 Einbettung der Socket-Implementierung

Sockets wurden in der Universität von Berkeley entwickelt (Unix BSD), und zwar erstmals in der Version 4.1 des BSD-Systems für die VAX im Jahr 1982. Sie werden daher auch als Berkeley Sockets bezeichnet. Die Originalversion der Socket-Schnittstelle stammt von Mitarbeitern der Firma BBN und wurde während eines ARPA-Projekts (1981) entwickelt (Hafner und Lyon 2000). Sockets sind auch Bestandteil des POSIX-Standards IEEE Std 1003.1-2008 für unixähnliche Systeme (IEEE POSIX 2016). Man spricht daher auch von der *POSIX Socket API.*

In Abb. 5.1 ist die Einbettung der Socket-Implementierung in ein Betriebssystem skizziert. Meistens ist die Socket-Implementierung Bestandteil des Betriebssystems und der Zugang in den Laufzeitsystemen der Programmiersprachen bzw. in den virtuellen Maschinen (siehe Java Virtual Machine) implementiert.

Die Unterschiede verbindungsloser und verbindungsorientierter Kommunikationsabläufe spiegeln sich auch in den Aufrufen der Socket-Schnittstelle wider. Es werden daher grundsätzlich TCP- und Datagramm-Sockets unterschieden.

5.2.2 TCP-Sockets

Die Socket API für TCP-Sockets unterstützt vor allem Client-/Server-Anwendungen,[1] was aus dem Programmiermodell hervorgeht. Es gibt einen aktiven und einen passiven Partner. Als *Socket* bezeichnet man die Kommunikationsendpunkte innerhalb der Applikationen, die in der Initialisierungsphase miteinander verbunden werden. Dabei spielt es keine Rolle, auf welchen Rechnern die miteinander kommunizierenden Prozesse laufen. Auch zwei Prozesse auf demselben Rechner können miteinander kommunizieren.

Die TCP-Socket-Schnittstelle ist streamorientiert, d. h. beim Verbindungsaufbau wird ein Datenstrom zwischen den Kommunikationsendpunkten eingerichtet. Jeder

[1] Selbstverständlich kann man auch Peer-to-Peer-Anwendungen mit Sockets realisieren.

Anwendungsprozess, der eine Verbindung zu einem Partner aufgebaut hat, verfügt über einen Sendestrom und über einen Lesestrom. Aus Sicht des Anwendungsprozesses werden also nur Bytes in den Sendestrom übergeben und die TCP-Instanz übernimmt ganz unabhängig davon die Zusammenstellung der TCP-Segmente. Die UDP-Socket-Schnittstelle ist dagegen nachrichtenorientiert. Es ist demnach verbindungsorientierte (TCP-basierte) und verbindungslose (UDP-basierte) Kommunikation möglich. Die Adressierung der Kommunikationspartner erfolgt über die Kommunikationsendpunkte mit dem Tupel bestehend aus IP-Adresse und Portnummer als Adressierungsinformation. Das Tupel wird auch als Socket-Adresse bezeichnet.

In Abb. 5.2 sind die wesentlichen Funktionen der Socket API zur Entwicklung verbindungsorientierter Kommunikationsanwendungen skizziert, wobei eine Zuordnung der Funktionen auf die Client- und die Serverseite erfolgt.

Der Aufbau der Kommunikationsbeziehung bei verbindungsorientierter Kommunikation läuft wie folgt ab, wobei man im klassischen Sinne einer Client-/Server-Architektur gerne die Begriffe *Client* für den aktiven Partner und *Server* für den passiven Partner verwendet:

- Beide Seiten rufen zunächst die *socket*-Primitive auf, um Kommunikationsendpunkte einzurichten. Beide Partner binden einen Port an den Kommunikationsendpunkt, damit dieser eindeutig adressierbar ist. Hierzu wird im Server die Primitive *bind* verwendet, um z. B. einen Well-known Port an den Kommunikationsendpunkt zu binden. Im Client kann auch die Primitive *bind* verwendet werden, wenn ein fest definierter Port vergeben werden soll, dies ist aber nicht zwingend.
- Die Serveranwendung wartet auf ankommende Verbindungsaufbauwünsche (Aufruf einer *listen*-Primitive) an einem TCP-Port.

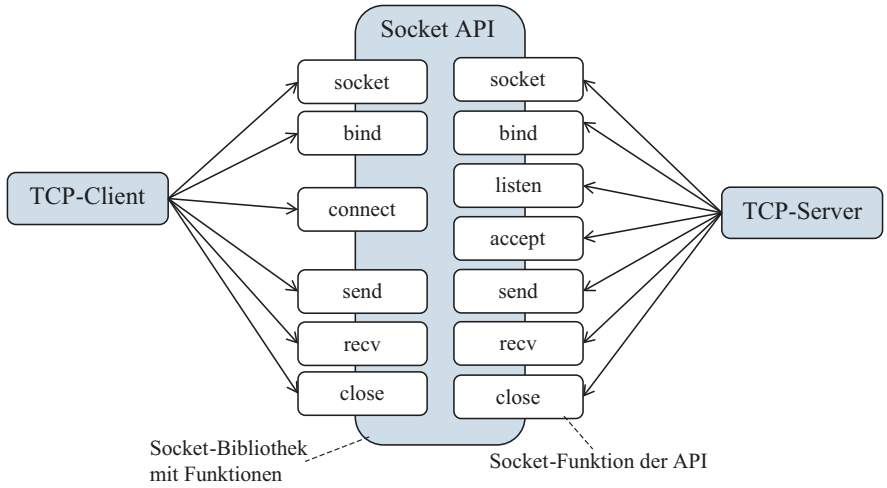


Abb. 5.2 TCP-Socket-Funktionen in der Client-Server-Beziehung

- Die Clientanwendung ruft eine *connect*-Primitive auf, die TCP-Instanz erzeugt daraufhin eine Connect-Request-PDU und sendet sie zum Server. Mit dem *connect*-Aufruf wird auch ein freier lokaler Port an die Clientanwendung vergeben, sofern noch keiner zugewiesen wurde.
- Die Serveranwendung nimmt den Verbindungswunsch über einen *accept*-Aufruf entgegen und beantwortet ihn mit einer Connect-Response-PDU. Implizit wird ein Dreiwege-Handshake ausgeführt.
- Anschließend kann mit *send*- und *receive*-Primitiven (*recv*) ein bidirektionaler und vollduplexfähiger Datenaustausch erfolgen.

Die aufeinanderfolgenden Aufrufe von Socket-Primitiven auf der Client- und auf der Serverseite sind in Abb. 5.3 skizziert.

Aus der Abbildung wird auch deutlich, dass der einfach strukturierte Server einen ankommenden Verbindungswunsch über die *accept*-Primitive bestätigt. Zur Darstellung des groben Ablaufs soll uns das ausreichen. Effizientere Serveranwendungen richten eine Verbindung ein und gehen dann sofort wieder an den Wartepunkt, der über die *accept*-Primitive realisiert ist.

Für den Verbindungsabbau wird von einer beliebigen Seite eine *close*-Primitive abgesetzt, die ein TCP-Dreiwege-Handshake einleitet. Der zweite Partner muss auch einen *close*-Aufruf absetzen.

Der Empfang von Nachrichten mit der Funktion *receive* kann sowohl synchron oder asynchron erfolgen. Mit synchron ist gemeint, dass die *receive*-Methode so lange blockiert, bis die zuständige TCP-Instanz eine Nachricht empfangen und vom Emfangspuffer an das Anwendungsprogramm übergeben hat. Im asynchronen Fall wird der Aufruf nicht blockiert, das rufende Programm erhält die Kontrolle sofort zurück und kann andere Dinge erledigen. Das

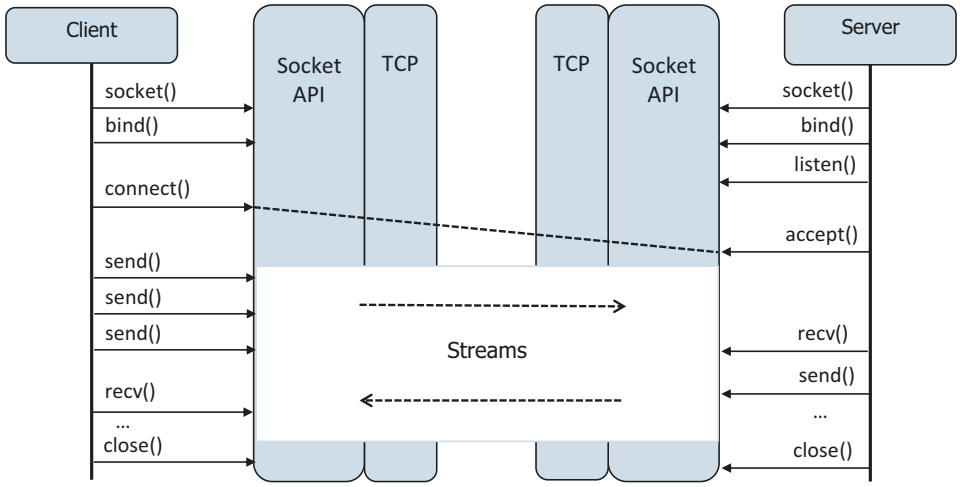


Abb. 5.3 Ablauf einer verbindungsorientierten Kommunikation

Programm muss also nicht warten, bis eine Nachricht zum Abholen bereit ist, sondern kann es später nochmals versuchen. Der *receive*-Aufruf kann auch mit einer maximalen Wartezeit aufgerufen werden, der Aufruf wird dann nur diese Zeit blockiert und danach wird die Kontrolle an den Aufrufer gegebenenfalls auch ohne empfangene Nachricht zurückgegeben.

Der *send*-Aufruf kehrt erst zurück, wenn die Nachricht im Sendepuffer der zuständigen TCP-Instanz, also in der Regel im Betriebssystemkernel, gelandet ist. Dies wird oft missverstanden. Gelegentlich wird angenommen, dass ein *send*-Aufruf erst zurückkommt, wenn die Nachricht beim Empfänger angekommen ist. Vielmehr ist die Nachricht zunächst erst einmal im lokalen Sendepuffer und TCP kann nach seinen Protokoll- und Implementierungsregeln entscheiden, wann ein Senden bzw. die Übergabe an die Netzwerkzugangsschicht tatsächlich erfolgen soll. In der Netzwerkkarte kann ebenfalls noch verzögert werden. Mit Absetzen eines *send*-Aufrufs wird also lediglich zugesichert, dass die Nachricht letztendlich übertragen wird, sofern keine schwerwiegenden Fehler oder ein Beenden der Partneranwendung zuvor kommen.

5.2.3 Datagramm-Sockets

Im Falle einer UDP-Sockets-Kommunikation fällt der Verbindungsaufbau im Vergleich zur TCP-Sockets-Kommunikation weg. In Abb. 5.4 sind die wesentlichen Funktionen der Socket API zur Entwicklung verbindungsloser Kommunikationsanwendungen skizziert. Client und Server nutzen jeweils die gleichen Funktionen.

Der Kommunikationsablauf ist in Abb. 5.5 dargestellt. Beide Partner erzeugen ein Socket (*socket*-Aufruf), binden jeweils eine Adresse (*bind*-Aufruf) und dann kann die Kommunikation beginnen. Hierfür stehen ähnliche Primitive wie bei TCP-Sockets bereit (*recvfrom*, *sendto*).

Im Unterschied zu TCP-Sockets gibt es keinen Verbindungsaufbau und dementsprechend keinen Verbindungsabbau. Wenn ein Kommunikationspartner die Kommunikation nicht weiterführt oder das Programm beendet wird, bekommt dies der andere nicht mit.

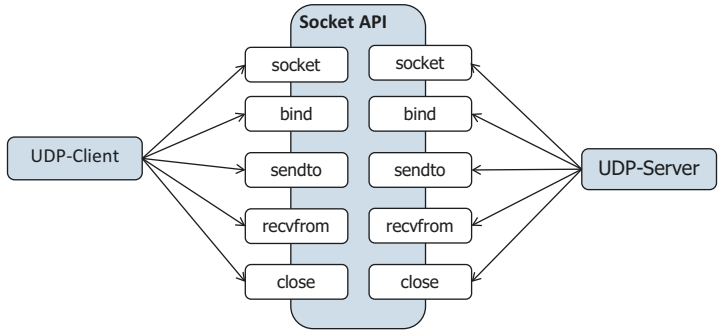


Abb. 5.4 Datagramm-Socket-Funktionen in der Client-Server-Beziehung

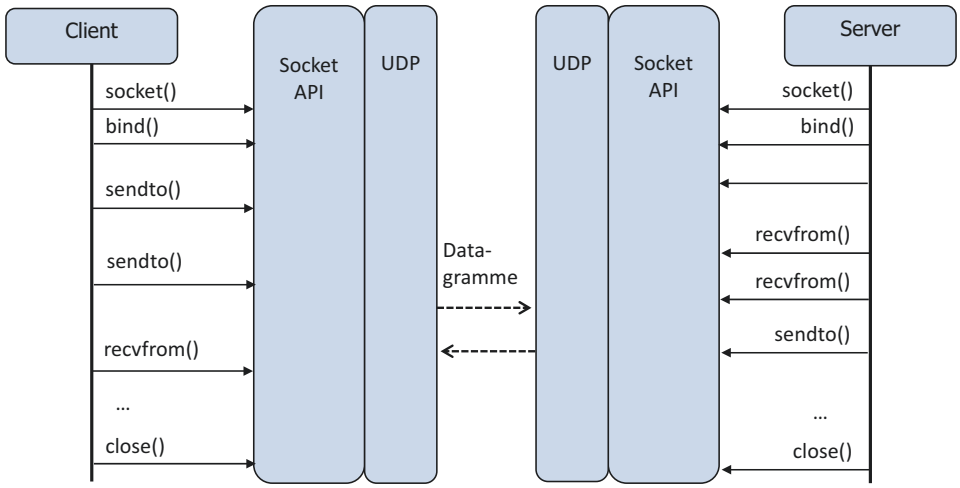


Abb. 5.5 Ablauf einer verbindungslosen Kommunikation

Für die Aufrufe *recfrom* und *sendto* gelten die gleichen Vorgaben zur Synchronisation wie bei TCP. Auch ein *revfrom*-Aufruf kann asynchron und mit einer maximalen Wartezeit genutzt werden. Im synchronen Fall wird so lange gewartet, bis eine Nachricht ankommt. Es könnte auch sein, dass nie eine Nachricht empfangen wird. In diesem Fall wartet der Anwendungsprozess, der den *recvfrom*-Aufruf tätigt, eventuell umsonst. Wie wir noch sehen werden, wird der nicht-blockierende Modus über eine Socket-Option aktiviert.

Datagramm-Sockets können im Gegensatz zu TCP-Sockets auch für Broad- und Multicasts verwendet werden. Mit einem *sendto*-Aufruf kann man also eine Nachricht an eine Gruppe von Empfängern oder sogar an alle Rechner im Subnetz senden. Man muss dazu nur die entsprechende Broadcast- oder Multicast-Adresse als Zieladresse angeben. Alle Empfänger müssen dann den gleichen UDP-Port benutzen.

5.3 Socket-Programmierung in C

5.3.1 Die wichtigsten Socket-Funktionen im Detail

Nach dieser kurzen Einführung in das Programmiermodell soll nun ein Überblick über die wichtigsten Socket-Funktionen gemäß POSIX-Spezifikation den Einstieg in die darauf folgenden Beispiele erleichtern (IEEE POSIX 2016).

Die C-Sockets-Schnittstelle wird in nahezu jedem Betriebssystem in einer Funktionsbibliothek bereitgestellt. Auf Basis der Funktionsbibliothek lässt es sich, verglichen mit anderen Sprachen wie Java, relativ aufwändig programmieren. Die meisten socketbasierten Kommunikationsanwendungen sind heute aber (noch) in C/C++ programmiert.

Der folgende Auszug aus der API der Berkeley Software Distribution (BSD) zeigt elementare Funktionen für die Entwicklung von Kommunikationsanwendungen auf

Basis von TCP und UDP (Stevens 2000; Stevens et al. 2005). Da für die Initialisierung von UDP und TCP zum Teil dieselben Socket-Funktionen mit verschiedenen Parametern verwendet werden, wird hier nach der Beschreibung zusätzlich aufgeführt, auf welcher Seite (Server, Client) und bei welchem Protokoll (TCP oder UDP) die Funktionen typischerweise verwendet werden.

Diese Schnittstellenspezifikation soll auch als Basis für die anderen Sockets-Implementierungen dienen. In Java, C# und Python ist die Nutzung der Socket API zwar komfortabler, aber im Prinzip läuft das Gleiche ab. Im Folgenden werden einige spezielle Datentypen genutzt, die in der Socket API erläutert sind. Hierzu gehört die Struktur *sockadr* zur Angabe von Adressen. Die Bedeutung der weiteren Datentypen ergibt sich aus der Bezeichnung.

socket()

```
#include <sys/socket.h>
int socket(int family, int type, int protocol);
```

initialisiert ein Socket zur Kommunikation und liefert bei Erfolg eine entsprechende Socket-Id (Socket-Deskriptor) zur Identifikation des erzeugten Sockets zurück. Ein Socket-Deskriptor wird im System wie ein File-Deskriptor für den Zugriff auf eine Datei behandelt.

Verwendet von: Client und Server, TCP und UDP

Parameter:

family: Die verwendete Adressfamilie (hier wird auch oft das Präfix PF_ vorgefunden).[2] Die zulässigen Adressfamilien sind in <*sys/socket.h*> definiert (Tab. 5.1).

type: Der Socket-Typ (bei TCP: SOCK_STREAM) (Tab. 5.2)

protocol: Das zu verwendende Protokoll der Adressfamilie. Oft 0, da die ersten beiden Angaben das Protokoll meist schon eindeutig spezifizieren.

bind()

```
#include <sys/socket.h>
int bind(int sockfd, const struct sockaddr *myaddr, socklen_t addrlen);
```

ordnet dem Socket eine lokale Adresse (myaddr) zu.

Verwendet von: Server und Client, TCP und UDP

Parameter:

sockfd: Der Socket-Deskriptor (siehe *socket()*)

myaddr: Zeiger auf eine Socket-Adressstruktur (bei TCP wird hier IP-Adresse und Port angegeben) zur Beschreibung der lokalen Adresse

addrlen: Länge der Server-Adressstruktur

[2] Mit der Trennung von Adressfamilie (AF_*) und Protokollfamilie (PF_*) wollte man sicherstellen, dass Protokollfamilien mit mehreren Adressstrukturen unterstützt werden können. Da dies bisher nicht vorkommt, werden die jeweiligen Konstanten gleichgesetzt. Hier wird nur ein Auszug der möglichen Werte gezeigt, vgl. auch (Stevens 2000).

Tab. 5.1 Belegung des Parameters family bei *socket*-Aufruf

Family	Beschreibung
AF_INET	IPv4-Protokolle
AF_INET6	IPv6-Protokolle
AF_LOCALE	Unix Domain-Protokolle
AF_ROUTE	Routing–Sockets
AF_KEY	Key Socket

Tab. 5.2 Belegung des Parameters *type* bei *socket*-Aufruf

type	Beschreibung
SOCK_STREAM	Stream-Socket: Bidirektionale, vollduplexfähige, verbindungsorientierte Kommunikation, typisch für TCP.
SOCK_DGRAM	Datagramm-Socket: Nachrichtenbasierte, unzuverlässige Kommunikation, typisch für UDP (Nachrichten mit fester Länge).
SOCK_RAW	Raw-Socket: Direkter Zugriff auf die Schicht 3.

connect()

```
#include <sys/socket.h>
int connect(int sockfd, const struct sockaddr *servaddr, socklen_t addrlen);
```

baut eine Verbindung vom Client zum Server mit der im Parameter *servaddr* angegebenen Adresse auf. Bei TCP wird hiermit der Dreiwege-Handshake initiiert. Findet zuvor kein *bind()* statt, wird hier ein freier lokaler Port zugeordnet.
Verwendet von: Client, TCP
Parameter:
sockfd: Der Socket-Deskriptor (siehe socket())
serveraddr: Zeiger auf eine Socket-Adressstruktur (bei TCP wird hier die IP-Adresse und der Port angegeben), in der die Partneradresse eingetragen ist
addrlen: Länge der Server-Adressstruktur

listen()

```
#include <sys/socket.h>
int listen(int sockfd, int backlog);
```

setzt das Socket in einen passiven, d. h. auf ankommende Verbindungswünsche wartenden Zustand.
Verwendet von: Server, TCP
Parameter:
sockfd: Der Socket-Deskriptor (siehe *socket()*)
backlog: Maximale Anzahl der ankommenden Verbindungen, die im Hintergrund (im Betriebssystemkernel) in einer Warteschlange eingereiht werden können

accept()

```
#include <sys/socket.h>
int accept(int sockfd, struct sockaddr *cliaddr, int *addrlen);
```

wird bei TCP-Verbindungen verwendet und gibt die nächste ankommende und aufgebaute Verbindung aus der Warteschlange zurück.
Verwendet von: Server, TCP
Parameter:
sockfd: Der Socket-Deskriptor (siehe *socket()*)
cliaddr: Zeiger auf die Datenstruktur, in welches die Client-Adresse der Verbindung gespeichert wird
addrlen: Größe der Socket-Adressstruktur des Clients

close()

```
#include <unistd.h>
int close(int sockfd);
```

schließt eine Verbindung. Bei TCP wird dabei versucht, alle noch nicht gesendeten Nachrichten zu senden und anschließend den Verbindungsabbau zu initialisieren. Nach Aufruf der Methode steht der Socket-Deskriptor nicht mehr zur Verfügung.
Verwendet von: Client und Server, TCP und UDP
Parameter:
sockfd: Der Socket-Deskriptor (siehe *socket()*)

recv()

```
#include <sys/types.h>
#include <sys/socket.h>
ssize_t recv(int sockfd, void *buf, size_t len, int flags);
```

liest Daten aus dem spezifizierten Socket und gibt die Anzahl der tatsächlich gelesenen Bytes zurück.
Verwendet von: Client und Server, TCP
Parameter:
sockfd: Der Socket-Deskriptor (siehe *socket()*)
buf: Zeiger auf den Puffer, in den die ankommenden Daten geschrieben werden sollen
len: Die Anzahl der zu lesenden Bytes
flags: Weitere Optionen zum Leseverhalten (vgl. Stevens 2000). Beispiel: Option MSG_WAITALL blockiert, bis alle erwarteten Daten (siehe *len*) im Stream zum Lesen bereit sind

send()

```
#include <sys/types.h>
#include <sys/socket.h>
ssize_t send(int sockfd, const void *msg, size_t len, int flags);
```

sendet Daten über das spezifizierte Socket und gibt die Anzahl der tatsächlich gesendeten Bytes zurück.

Verwendet von: Client und Server, TCP
Parameter:
sockfd: Der Socket-Deskriptor (siehe *socket()*)
msg: Zeiger auf den Puffer, in dem die zu sendenden Daten vorliegen
len: Die Anzahl der zu sendenden Bytes
flags: Weitere Optionen zum Sendeverhalten (vgl. Stevens 2000)

recvfrom()

```
#include <sys/socket.h>
ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags, struct
sockaddr *from, socklen_t *fromlen);
```

empfängt eine Nachricht an dem angegebenen Socket, speichert sie im Puffer *buf* und gibt die Anzahl der gelesenen Bytes zurück. Zusätzlich werden hier die Adressdaten des Senders in der übergebenen Adressstruktur *from* abgelegt.
Verwendet von: Client und Server, UDP
Parameter:
sockfd: Der Socket-Deskriptor (siehe *socket()*)
buf: Zeiger auf den Puffer, in dem die zu sendenden Daten vorliegen
len: Die Anzahl der zu sendenden Bytes
flags: Weitere Optionen zum Leseverhalten (vgl. Stevens 2000)
from: Zeiger auf die Adressstruktur, in der die Client-Adressdaten gespeichert werden sollen
fromlen: Zeiger auf einen Speicherbereich vom Typ *Integer*, in dem die Länge von *from* abgelegt wird

sendto()

```
#include <sys/socket.h>
ssize_t sendto(int sockfd, const void *msg, size_t len, int flags, struct
sockaddr *to, socklen_t tolen);
```

sendet eine Nachricht aus dem übergebenen Puffer *msg* an die Adresse *to* und liefert die Anzahl der gesendeten Bytes zurück.
Verwendet von: Client und Server, UDP
Parameter:
sockfd: Der Socket-Deskriptor (siehe *socket()*)
msg: Zeiger auf den Puffer, in dem die zu sendenden Daten vorliegen
len: Die Anzahl der zu sendenden Bytes
flags: Weitere Optionen zum Sendeverhalten (vgl. Stevens 2000)
to: Zeiger auf die Adresse des Empfängers
tolen: Länge der Adressstruktur
Die Funktion *fork* gehört zwar nicht zur Socket API, ist aber für die Verwaltung von mehreren TCP-Verbindungen wichtig und wird daher hier zusätzlich aufgeführt. Mit *fork* können unter Unix und unixähnlichen Betriebssystemen neue Prozesse erzeugt werden.

fork()

```
#include <unistd.h>
pid_t fork(void);
```

erstellt eine Kopie des aktuellen Prozesses. Der neue erzeugte Prozess läuft als Kindprozess parallel zum erzeugenden Prozess (Elternprozess). Der Rückgabewert der Funktion ist im Elternprozess die Prozess-Id (pid) des Kindprozesses. Im Kindprozess wird 0 zurückgeliefert.

Diese Funktion wird bei Socket-Anwendungen genutzt, um die parallele Bearbeitung mehrerer Verbindungen sowie das gleichzeitige Entgegennehmen neuer Verbindungswünsche durch nebenläufige Prozesse zu ermöglichen.

5.3.2 Nutzung von Socket-Optionen

Die in den vorhergehenden Kapitel diskutierten Socket-Optionen kann man im Socket API mit der Funktion *setsocketopt* für ein definiertes Socket verändern und mit *getsocketopt* auslesen (siehe Beispiel für das Setzen einer Socket-Option).

setsocketopt()

```
#include <sys/socket.h>
int setsockopt(int socket, int level, int option_name, const void
*option_value, socklen_t option_len);
```

Die Funktion dient zum Verändern von Socket-Optionen.
Verwendet von: Client und Server, TCP, UDP und weitere
Parameter:

sockfd:	Der Socket-Deskriptor (siehe *socket()*)
level:	Protokoll, für das die Option gesetzt werden soll
option_name:	Optionsbezeichnung
option:_value:	Optionswert
option_len:	Länge des Optionswerts

Die verschiedenen Protokoll-Level sind in Tab. 5.3 aufgeführt, mögliche Optionen in Tab. 5.4.

Beispiel für das Setzen einer Socket-Option

Für ein konkretes UDP-Socket kann die Prüfsummenberechnung mit folgender Codesequenz deaktiviert werden:

```
…
int disable = 1;
if (setsockopt(sock, SOL_SOCKET, SO_NO_CHECK, (void*)&disable,
sizeof(disable)) < 0) {
   perror("setsockop-Aufruf fehlerhaft");
}
…
```

Tab. 5.3 Protokoll-Level, für das die Option gültig ist

Protokoll-Level	Beschreibung
SOL_SOCKET	Socket-Level
IPPROTO_IP	Internet Protocol (IPv4)
IPPROTO_IPV6	Internet Protocol (IPv6)
IPPROTO_ICMP	ICMP, Kontroll- und Steuerprotokoll im Internet
IPPROTO_RAW	Direkter Zugriff auf das Internet Protocol IP
IPPROTO_TCP	TCP
IPPROTO_UDP	UDP

Tab. 5.4 Mögliche Optionen (Auszug)

Optionsname	Beschreibung
SO_BROADCAST	Broadcast-Nachrichten sind erlaubt (Integer-Wert als Boolean: 0 oder 1).
SO_REUSEADDR	Bei einem *bind*-Aufruf kann eine lokale Adresse sofort wiederverwendet werden (Integer-Wert als Boolean: 0, 1).
SO_KEEPALIVE	Periodisches Senden von Nachrichten zur Lebendüberwachung (Integer-Wert, semantisch als Boolean-Wert: 0, 1)
SO_LINGER	Regelt, wie nicht gesendete Daten bei einem *close*-Aufruf behandelt werden. Wenn SO_LINGER gesetzt ist, wird der *close*-Aufruf so lange blockiert, bis alle Daten übertragen wurden (linger-Struktur als Wert, siehe *socket.h)*. In der Struktur wird SO_LINGER ein- oder ausgeschaltet und eine maximale Blockierungszeit beim *close*-Aufruf angegeben.
SO_SNDBUF	Setzen der Sendepuffergröße des Sockets (Integer-Wert gibt Puffergröße in Bytes an).
SO_RCVBUF	Setzen der Empfangspuffergröße des Sockets (Integer-Wert gibt Puffergröße in Bytes an).
SO_NO_CHECK	Aktivieren oder Deaktivieren der UDP-Prüfsummenberechnung für ein UDP-Socket.

getsocketopt()

```
#include <sys/socket.h>
int getsockopt(int socket, int level, int option_name, const void
*option_value, socklen_t *option_len);
```

Die Funktion dient zum Auslesen von Socket-Optionen. Es werden die gleichen Wertebereiche für die Parameter verwendet wie bei *setsockopt*. Die Parameter *option_value* und *option_len* dienen hier als Ausgabeparameter. Die ausgelesenen Werte werden an die Adresse *option_value* übergeben und die Länge der Option an die Adresse *option_len*.

fcntl()

```
#include <unistd.h>
#include <fcntl.h>
int fcntl(int fd, int cmd, ... /* arg */ );
```

Über diese Funktion, die der Manipulation von Datei-Deskriptoren und damit auch Socket-Deskriptoren dient, kann zudem noch die Option *O_NONBLOCK* gesetzt werden, um ein Blockieren des *receive*-Aufrufs zu verhindern, wenn keine Nachrichten anstehen. Für eine weiteren Erläuterung wird auf die Linux-Literatur oder den POSIX-Standard verwiesen.[3]

5.3.3 Nutzung von TCP-Sockets in C

Im folgenden Beispielcode, nach (Stevens 2000) skizziert, wird der Verbindungsaufbau zwischen einem TCP-Client und einem TCP-Server skizziert. Die eigentliche Verarbeitungslogik und eine adäquate Fehlerbehandlung werden aus Vereinfachungsgründen weggelassen und die Adressen sind im Programm als Konstanten angegeben.[4]

Im Server (siehe Programmcode für TCP-Server) wird zunächst ein Socket erzeugt und an eine Adresse gebunden (*socket*- und *bind*-Primitive). Das Zusammenstellen der Adresse ist in C etwas umständlich, da man eine Struktur vom Typ *sockaddr_in* befüllen muss. Da es verschiedene Socket-Typen gibt, muss beim *socket*-Aufruf der passende Typ (hier SOCK_STREAM für TCP-Sockets) angegeben werden. Es wird ein Socket-Deskriptor zurückgegeben, der im Weiteren bei jedem Aufruf genutzt wird.

```
/* TCP-Server */
#include "inet.h"
#define SERV_TCP_PORT 5999
char *pname;...

main(int argc, char argv[]){
    int sockfd, newsockfd, clilen, childpid;
    struct sockaddr_in cli_addr, serv_addr;
    pname = argv[0];
    // Socket öffnen und binden
    if ((sockfd = socket(AF_INET, SOCK_STREAM, 0) ) < 0) {
        // Fehlermeldung ausgeben und Fehlerbehandlung durchführen ...
        exit(1);
    }

    // Lokale Adresse binden, vorher sockaddr mit IP-Adresse und Port
                                                              belegen
    bzero((char *) &serv_addr, sizeof(serv_addr));
```

[3] Siehe zum Beispiel (IEEE POSIX 2016), oder http://man7.org/linux/man-pages/man2/fcntl.2.html. Zugegriffen am 21.08.2017.

[4] In guten Socket-Programmen wird man keine verdrahteten IP-Adressen vorfinden, sondern Hostnamen, die über DNS aufgelöst werden.

```
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    serv_addr.sin_port = htons(SERV_TCP_PORT);

    if (bind(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr)
                                                              < 0) {
     // Fehlermeldung ausgeben und Fehlerbehandlung durchführen ...
        exit(1);
    } else {
        listen(sockfd, 5);
        for (;;) { // Auf Verbindungsaufbauwunsch warten
            clilen = sizeof(cli_addr);
            newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr,
                                                            &clilen);
            if ((childpid = fork()) < 0) {
            // Fehlermeldung ausgeben
            } else if (childpid == 0) { // Sohnprozess
                close(sockfd);
                // Client-Request bearbeiten...
                exit(0);
            } //else
            close (newsockfd); // Elternprozess
        } // for
    } // if
}
```

Anschließend geht der Server in den Zustand LISTEN, in dem er auf ankommende Verbindungswünsche wartet. Die Länge der Anfragewarteschlange besagt, dass gleichzeitig maximal fünf Clients auf eine Bearbeitung des Verbindungsaufbauwunsches warten können. Jeder Verbindungsaufbauwunsch wird zunächst im Server in die Anfragewarteschlange eingereiht. Mit jedem *accept*-Aufruf wird ein Verbindungsaufbauwunsch bearbeitet.

In der folgenden Endlosschleife wird für einen Verbindungsaufbauwunsch ein blockierender *accept* ausgeführt. Mit *accept* wird jeweils der erste in der Warteschlange stehende Verbindungsaufbauwunsch angenommen und für die neue Verbindung ein weiteres Socket mit den gleichen Eigenschaften wie das angegebene Socket generiert. Steht keine Anforderung an, blockiert der *accept*-Aufruf.[5] Nach Aufbau der Verbindung wird ein Kindprozess erzeugt und diesem die Verbindung zur weiteren Verarbeitung übergeben. Der Elternprozess geht sofort wieder zum *accept* und wartet auf den nächsten Verbindungsaufbauwunsch. Wenn der Kindprozess den Request abgearbeitet hat, terminiert er.

Diese Art der Programmierung ist typisch für viele heute existierende Anwendungen und ermöglicht durch die Einschaltung von Sohnprozessen die Parallelearbeitung

[5] Ein nicht blockierender Aufruf von *accept* ist ebenfalls möglich.

mehrerer Clients. Anstelle von mehreren Worker-Prozessen kann man hier auch Threads verwenden.

Der zugehörige Client (siehe Programmcode für TCP-Client) ist sehr einfach gestaltet. Er kennt die IP-Adresse des Servers sowie die Portnummer des gewünschten Serverdienstes (hier 5999) und initiiert nach dem Anlegen eines Sockets den Verbindungsaufbau über einen *connect*-Aufruf. Anschließend sendet er einen Request, verarbeitet das Ergebnis und schließt das Socket. Dies ist eine sehr vereinfachte Socket-Anwendung. In der Regel wird eine bestehende Verbindung für mehrere Requests benutzt, da der Verbindungsaufbau doch relativ aufwändig ist.

```
/* TCP-Client */
#include ...
#include "inet.h"
#define SERV_TCP_PORT 5999

// Serveradresse
#define SERV_HOST_ADDR "192.43.235.6"
char *pname;

main(int argc, char argv[]) {
    int sockfd;
    struct sockaddr_in serv_addr;
    pname = argv[0];

    // Serveradresse belegen...
    bzero((char *) &serv_addr, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = inet_addr(SERV_HOST_ADDR);
    serv_addr.sin_port = htons(SERV_TCP_PORT);
    // Socket öffnen
    if ((sockfd =socket(AF_INET, SOCK_STREAM, 0) < 0) {
        // Fehlermeldung ausgeben und Fehlerbehandlung durchführen ...
        exit(1);
    }
    // Verbindung zum Server aufbauen
    if (connect(sockfd,
        (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0) {
        // Fehlermeldung ausgeben...
    }
        // Verarbeitung: Request senden ...
        // Nicht weiter ausgeführt
    close(sockfd);
    exit(0);
}
```

5.3.4 Nutzung von UDP-Sockets in C

Die Nutzung von Datagramm-Sockets ist im folgenden Beispiel dargestellt. Es handelt sich hier um einen einfachen Echo-Service. Der Client sendet eine Anfrage mit einem Text an den UDP-Server, der diesen einfach zurücksendet.

Der Server (siehe Programmcode für UDP-Server) öffnet ein Datagramm-Socket (SOCK_DGRAM), bindet seine Adresse an das Socket und springt dann in eine Echo-Funktion. Diese Funktion macht nichts anderes als auf einen Request zu warten, ihn mit einem *recvfrom*-Aufruf zu lesen und die ganze Nachricht mit der *sendto*-Primitive an den Client zurückzusenden. Die Maximallänge der empfangenen Nachricht wird im *recvfrom*-Aufruf (MAXMESG) begrenzt.

```
/* UDP-Server */
#include „inet.h";
#include <sys/types.h>
#include <sys/socket.h>
#define MAXMESG 2048
#define SERV_UDP_PORT 5999
...
main(int argc, char argv[])
{
    int sockfd;
    struct sockaddr_in serv_addr, cli_addr;
    pname = argv[0];
    // UDP-Socket oeffnen
    if (sockfd = socket(AF_INET, SOCK_DGRAM,0)) < 0) {
                // Fehlerbehandlung durchführen
    }
    /* Adresse aufbauen */
    bzero((char *) &serv_addr, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = inet_addr(INADDR_ANY);
    serv_addr.sin_port = htons(SERV_UDP_PORT);
    // Binde lokale Adresse
    if (bind(socketfd, &serv_addr,...) {
           // Fehlerbehandlung durchführen
    }
    else {
        wait_and_send (sockfd, (struct sockaddr *) &cli_addr,
               sizeof(cli_addr));
    }
}
/* Jede Nachricht wird zum Client zurückgesendet, echo */
wait_and_send(int sockfd, sockaddr pcli_addr, int maxclilen)
```

```
{
    int n, clilen;
    char mesg[MAXMESG];
    for (;;) {
        clilen = maxclilen;
        n = recvfrom(sockfd, mesg, MAXMESG, 0, pcli_addr, &clilen);
        if (n < 0) {
            // Fehlerbehandlung durchführen
        }
        if (sendto(sockfd, mesg, n, 0, pcli_addr, clilen) != n) {
            // Fehlerbehandlung durchführen
        }
    }
}
```

Der Client (siehe Programmcode für UDP-Client) erzeugt zunächst ebenfalls ein Datagramm-Socket. Danach liest er einen Text aus der Standardeingabe ein, legt diesen in ein Datagramm und sendet dieses an den bekannten Server. Nach dem Empfang des Echos wird dieses auf die Standardausgabe ausgegeben, anschließend wird das Socket geschlossen und die Clientanwendung beendet.

```
/* UDP-Client */
#include „inet.h“
#include <stdio.h>
#include <sys/socket.h>
#define MAXLINE 512
#define SERV_UDP_PORT 5999
#define SERV_HOST_ADDR “192.43.235.6“

main(int argc, char argv[])
{
    int sockfd;
    struct sockaddr_in serv_addr, cli_addr;
    pname = argv[0];
    /* Adresse aufbauen */
    bzero((char *) &serv_addr, sizeof(serv_addr());
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = inet_addr(SERV_HOST_ADDR);
    serv_addr.sin_port = htons(SERV_UDP_PORT);
    // UDP-Socket öffnen und lokale Adresse binden
    if (sockfd = socket(AF_INET, SOCK_DGRAM,0)) < 0) {
            // Fehlerbehandlung durchführen
    }
```

```
    if (bind(socketfd, (struct sockaddr *) &cli_addr,...) {
            // Fehlerbehandlung durchführen
    }
    else {
        send_and_wait (stdin, sockfd,
               (struct sockaddr *) &serv_addr, sizeof(serv_addr));
    }
    close(sockfd);
    exit(0);
}

/* Nachricht über stdin einlesen, zum Server senden, Echo wieder */
/* empfangen und auf stdout ausgeben */
send_and_wait (FILE fd, int sockfd, struct sockaddr *pserv_addr,
               int servlen)
{
    int n;
    char sendline[MAXMESG]; recvline[MAXLINE+1];
    /* Nachricht von stdin einlesen … */
    if (sendto(sockfd, sendline, n, 0, pserv_addr, servlen) != n) {
            // Fehlerbehandlung durchführen …
    }
    n = recvfrom(sockfd, recvlen,
            MAXLINE, 0, (struct sockaddr *) 0, (int *) 0);
    if (n < 0)
    {
             // Fehlerbehandlung durchführen
    }
    /* Nachricht auf Standardausgabe (stdout) ausgeben (hier nicht */
    /* dargestellt)… */
}
```

Kommunikationsprotokolle sollten unabhängig von der lokalen Syntax sein, d. h. die Nachrichten sollten am besten in eine Transportsyntax transferiert werden, die alle beteiligten Partner verstehen. Generell muss man sich bei der Socket-Programmierung selbst um die Darstellung der Daten in den Nachrichten kümmern. Zwei kommunizierende Prozesse wissen ja nicht, ob ihre lokalen Syntaxen gleich sind oder nicht, da ihnen die Computerarchitekturen der Rechner, auf denen sie laufen, nicht bekannt sind. Beispielsweise kann es sein, dass die beiden Rechner unterschiedliche Integer-Formate nutzen. Aus diesem Grund stellt die Socket-Schnittstelle Konvertierungs-Routinen bereit. Beispiele für Konvertierungs-Routinen sind *htonl* und *htons* zur Umwandlung von lokalen Long- und Short-Integerwerten in eine unabhängige Transportsyntax sowie die Umkehrfunktionen *ntohl* und *ntohs*.

Darstellung von Nachrichten, Präsentationsschicht
In höheren RPC-basierten Protokollen werden für die Serialisierung von Nachrichten gewisse Automatismen bereitgestellt. Bei ONC-PRC, einer RPC-Implementierung von Sun Microsystems, wird z. B. XDR (eXternal Data Representation) zur Umwandlung beliebiger Datenstrukturen in eine Transportsyntax genutzt. Aus der ISO/OSI-Welt ist hier die *Abstract Syntax Notation* 1 (*ASN.1*) mit den zugehörigen *Basic Encoding Rules* (*BER*) bekannt, die über sogenannte ASN.1-Compiler unterstützt werden.

Heute verwendet man in Anwendungsprotokollen oft eine textorientierte Nachrichtenkodierung und nutzt dafür Auszeichnungssprachen wie die *Extensible Markup Language* (*XML*) oder man verwendet *JavaScript Object Notation (JSON)*[6] als kompaktes Datenformat vor allem in Web- und in mobilen Anwendungen unter Nutzung von Webservices.

5.4 Socket-Programmierung in Java

5.4.1 Überblick über Java-Klassen

Die Entwicklung von Socket-Programmen ist in Java etwas komfortabler als in C, da einfach zu nutzende Objektklassen bereitgestellt werden. Die Java-API stellt im Wesentlichen das Java-Package *java.net* für die Netzwerkprogrammierung zur Verfügung. In diesem Package sind auch die Objektklassen zur Socket-Programmierung enthalten. Das Package *java.net* stellt eine höherwertige Schnittstelle für Sockets zur Verfügung, die als objektorientiert bezeichnet werden kann und die Socket-Details gut kapselt. Die Programmierung wird durch die Abstraktion der Input- und Output-Streams, die den Sockets zugeordnet werden, vereinfacht.

In Abb. 5.6 ist ein Ausschnitt aus der Java-Dokumentation des Packages *java.net* in einem Klassenmodell skizziert. Wie in Java üblich, sind Klassen entweder direkt oder indirekt von der Basisklasse *java.lang.Object* abgeleitet. Wichtige Klassen für die Nutzung von TCP-Sockets sind

- *InetAddress* für die Nutzung von Internet-Adressen,
- *Socket* zur clientseitigen Socket-Nutzung,
- *ServerSocket* zur serverseitigen Socket-Nutzung.

Das Package *java.net* enthält auch Klassen zur Bearbeitung von UDP-Sockets. Wichtige Klassen sind hier

- *DatagramSocket* für die Nutzung von UDP-Sockets,
- *DatagramPaket* zur Aufbereitung und Bearbeitung von Datagrammen.

Die Klasse *MulticastSocket* ist eine Unterklasse von *DatagramSocket* und dient zur Multicast-Kommunikation über UDP.

Schicht-3-Adressen (IP-Adressen) müssen beim TCP-Verbindungsaufbau oder beim Senden von UDP-Datagrammen angegeben werden, um die Zielrechner zu identifizieren. Sie werden auch zur eindeutigen Zuordnung von IP-Adressen zu Sockets verwendet, da ein

[6] http://www.json.org. Zugegriffen am 10.09.2017.

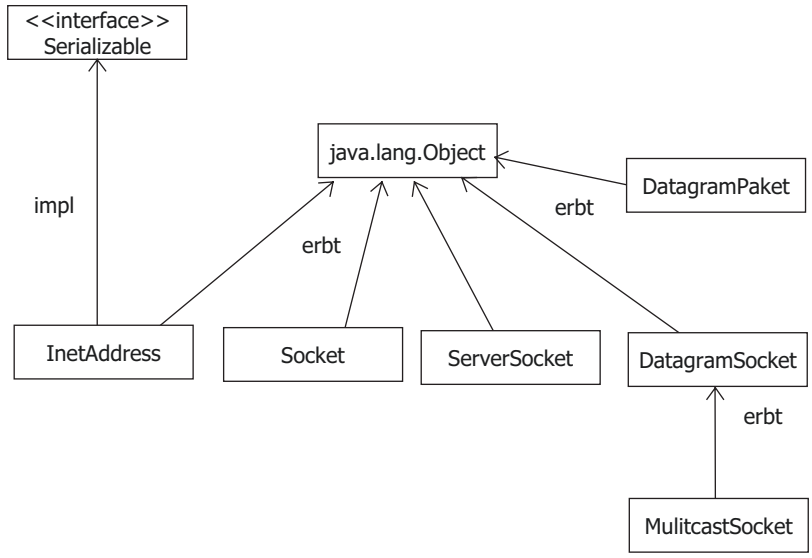


Abb. 5.6 Objektklassen für Sockets aus java.net

Rechner bzw. sogar einzelne Netzwerkschnittstellen in der Internet-Welt auch mehrere Schicht-3-Adressen haben können. An der Socket-Schnittstelle werden IP-Adressen unter Nutzung der Java-Klasse *InetAddress* angegeben. Als Adressangaben sind IPv4- und IPv6-Adressen sowie Hostnamen, die implizit über das Domain Name System (DNS) auf IP-Adressen abgebildet werden, zulässig. Für IPv4- und IPv6-Adressen existieren wiederum die *InetAddress*-Subklassen *Inet4Address* und *Inet6Address*. Auf die Unterschiede soll an dieser Stelle nicht eingegangen werden. Die wichtigsten Methoden sind in der Java-API zusammengefasst. Die Verwendung der Klassen wird in den weiter unten folgenden Beispielen deutlich.

In manchen Socket API-Klassen wird auch die abstrakte Klasse *SocketAddress* verwendet, um Adressangaben zu machen. Mit Objekten dieser Klasse können Socket-Adressen angegeben werden, die noch keinen Bezug zu einem konkreten Protokollstack aufweisen. Sie müssen bei konkreter Nutzung auch mit konkreten Adressangaben wie z. B. mit einem *InetAddress*-Objekt belegt werden.

Für die Entwicklung von TCP-Anwendungen werden die Klassen *Socket* und *ServerSocket* sowie *InetAdress* verwendet. Da aus Sicht von Java das Senden und Empfangen von Daten wie das Schreiben und Lesen von Dateien behandelt wird, werden bei TCP-Anwendungen auch die Stream-Klassen aus dem Java-Package *java.io* benutzt. Eine TCP-Verbindung wird also als Endpunkt für einen Ein- und Ausgabestrom abstrahiert. Man kann Daten in den Output-Stream schreiben oder aus dem Input-Stream herauslesen. Jede Kommunikationsseite nutzt zwei Kanäle, einen Input-Stream, der logisch mit dem Output-Stream des Partners verbunden ist und umgekehrt.

Eine weitere, wichtige Objektklasse ist *SocketChannel*. Im Gegensatz zu *ServerSocket* und *Socket* ist die Objektklasse *SocketChannel* threadsafe (siehe wichtige Anmerkung unten) und ermöglicht auch den Empfang im Nonblocking-Mode. Eine Abfrage, ob auf dem Socket

ein Empfangsereignis vorliegt, kann blockierungsfrei erfolgen. Dies ist bei Serverimplementierungen, die sehr viele Clients bedienen müssen, besonders wichtig. Die Objektklasse *SocketChannel* gehört zum Java-Package java.nio und wird weiter unten noch behandelt.

> *Threadsafe* oder auch *reentrant* sind Programmteile, die jederzeit ohne Nebenwirkungen gleichzeitig oder quasi-gleichzeitig von mehreren Prozessen oder Threads durchlaufen werden können. Datenstrukturen, die während des Durchlaufs verändert werden, werden dabei durch Synchronisationsmaßnahmen wie Sperren, Semaphore oder Monitore (in Java durch das Schlüsselwort *synchronized* definierbar) geschützt (Mandl 2014). Wenn eine Objektklasse als threadsafe bezeichnet wird, kann sie in beliebig nebenläufigen Programmen eingesetzt werden. Ist eine Klasse nicht threadsafe, muss die nutzende Klasse selbst für die Synchronisationsmaßnahmen sorgen. Bei den Klassen *Sockets* und *ServerSockets* ist der Anwender beispielsweise selbst verantwortlich. Es ist also Vorsicht geboten, wenn mehrere Threads die gleiche Verbindung nutzen.

In der Abb. 5.7 ist der Ablauf einer TCP-basierten Kommunikation mit Java-Mitteln schematisch und rudimentär skizziert. Der Server legt ein Server-Socket an, wobei ein Port angegeben wird. Implizit wird bei dieser Nutzungsart abhängig vom verwendeten Server-Socket-Konstruktor auch schon eine *bind*-Methode ausgeführt, um die lokale Adresse zuzuordnen und es wird ein *listen*-Aufruf ausgeführt, um auf ankommende Verbindungsaufbauwünsche zu horchen. Danach ruft der Server die Methode *accept* auf, die auf Verbindungaufbauwünsche wartet und diese gleich akzeptiert.

Der Client erzeugt eine Instanz der Klasse *Socket*, wobei ebenfalls implizit ein *bind*-Aufruf durchgeführt und ein Verbindungswunsch (*connect*-Aufruf) abgesetzt wird. Implizit wird ein TCP-Dreiwege-Handshake-Verbindungsaufbau ausgeführt.

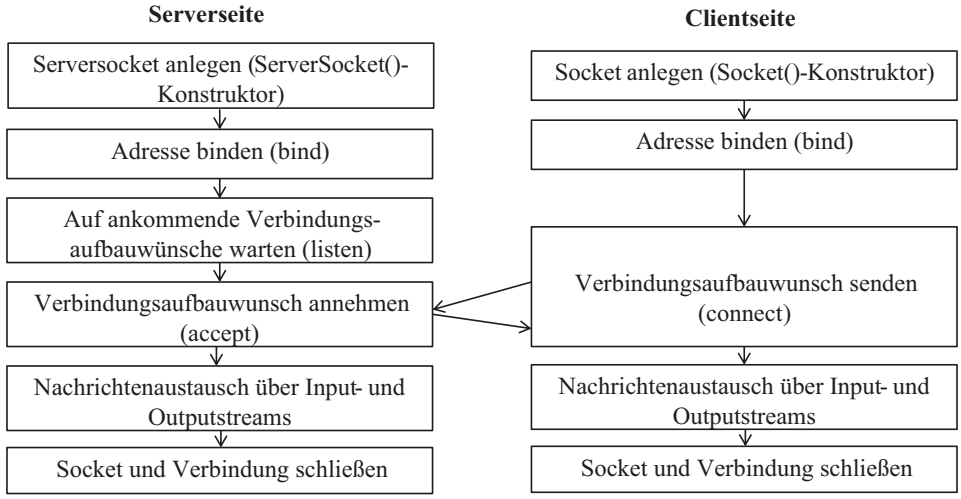


Abb. 5.7 Schematische Darstellung einer einfachen TCP-Socket-Kommunikation in Java

Anschließend stehen an den client- und serverseitigen Sockets jeweils Input- und Output-Streams zur Verfügung, über die beide Seiten unabhängig voneinander Nachrichten senden und empfangen können. Die Verbindung ist damit auch aufgebaut. Über die Streams können Daten byteweise gesendet und empfangen werden. Die Streams können, sofern gewünscht, mit ObjectStreams angereichert werden, so dass ein Senden und Empfangen von serialisierten Java-Objekten möglich wird.

Mit dieser einfachen Vorgehensweise wird also ein bidirektionaler und vollduplexfähiger Kommunikationskanal aufgebaut, wie man ihn von TCP her kennt, und den Anwendungen zur Nutzung bereitgestellt. Beide Seiten dürfen unabhängig voneinander Nachrichten senden. Zum Beenden der Kommunikationsbeziehung müssen beide Partner am Socket die Methode *close* aufrufen. Die Verbindung wird dann implizit abgebaut.

In der Abb. 5.8 ist der Ablauf einer UDP-Datagramm-Kommunikation mit Java-Mitteln skizziert. Beide Partner erzeugen eine Instanz der Klasse *DatagramSocket* und binden eine Adresse an die jeweilige Instanz. Danach kann sofort gesendet und empfangen werden, wobei Instanzen der Klasse *DatagramPacket* genutzt werden. Zum Abschluss der Kommunikation schließen beide Partner die Verbindung mit einem *close*-Aufruf.

Die wichtigsten Java-Klassen (Java Standard Edition Version 8) werden im Folgenden erläutert. Eine detaillierte Beschreibung aller Java-Methoden und -Konstruktoren in der Java-API-Beschreibung finden sich im WWW.[7]

Das Package *java.net* stellt auch vordefinierte Exceptions für die Ausnahmebehandlung bei Netzwerkproblemen bereit (siehe Programmcode für Java-Exceptions bei Socket-Aufrufen). Die Exception-Hierarchie für die Ausnahmebehandlung bei Java-Sockets sieht wie folgt aus, wobei die Bezeichnungen der Ausnahmen für sich sprechen:

```
/* Java-Exceptions bei Socket-Aufrufen */
java.lang.Object
  java.lang.Throwable
    java.lang.Exception
      java.io.IOException
        java.net.ProtocolException
        java.net.UnknownHostException
        java.net.UnknownServiceException
        java.net.SocketException
          java.net.ConnectException
          java.net.BindException
          java.net.NoRouteToHostException
          ...
```

Die Ausnahmen werden gegebenenfalls bei einem Methodenaufruf erzeugt (geworfen) und müssen entsprechend behandelt werden.

[7] https://docs.oracle.com/javase/8/docs/api/. Zugegriffen am 25.09.2017.

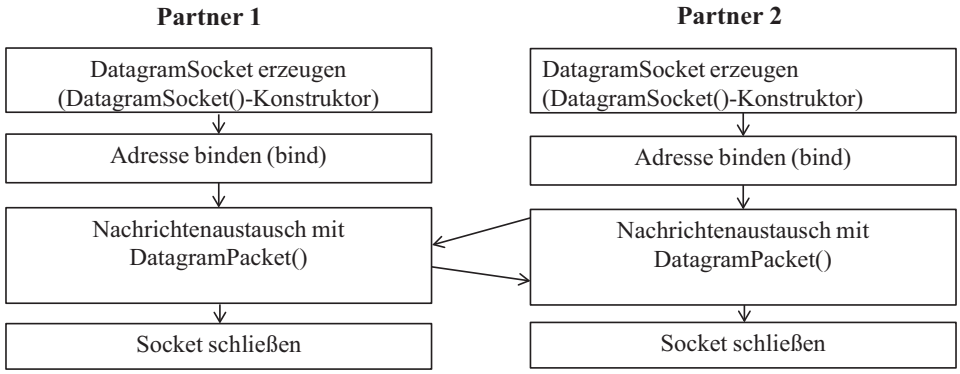


Abb. 5.8 Schematische Darstellung einer UDP-Datagram-Kommunikation in Java

5.4.2 Streams für TCP-Verbindungen

Jedes Socket wird mit zwei Streams versehen, einem Input-Stream zum Empfangen von Nachrichten über eine aufgebaute TCP-Verbindung und einem Output-Stream zum Senden von Nachrichten an den jeweiligen Verbindungspartner. Damit nutzt Java die Stream-Abstraktion, die für das Bearbeiten von Dateien verwendet wird, auch für die Kommunikation über Netzwerkverbindungen.

Konkret wird jedem Socket ein Stream-Paar bestehend aus Instanzen von *InputStream* und *OutputStream* (siehe Java-Package *java.io*) zugeordnet. Über diese Streams kann man Bytes oder Byte-Arrays senden und empfangen.

Befindet man sich in einer reinen Java-Umgebung, d. h. beide Partneranwendungen laufen in einer Java Virtual Machine, kann man auch Object-Streams verwenden, über die ganze Java-Objekte ausgetauscht werden können. Hierfür konstruiert man über die Input- und Output-Streams, die mit Getter-Methoden am Socket ermittelbar sind, entsprechende ObjectInput- und OutputStreams (siehe Programmcode zum Erzeugen von Object-Streams für ein Socket).

```
/* Erzeugen von Object-Streams für ein Socket */
…
try {
        Socket client = new Socket (Host, Portnummer);
        ObjectInputStream in =
          new ObjectInputStream(client.getInputStream();
        ObjectOutputStream out =
          new ObjectOutputStream(client.getOutputStream());
…
} …
```

In Abb. 5.9 ist die Einbettung von Java-Sockets in die virtuellen Maschinen (JVMs) von kommunizierenden Anwendungen skizziert. Jedem Socket ist also ein Stream-Paar

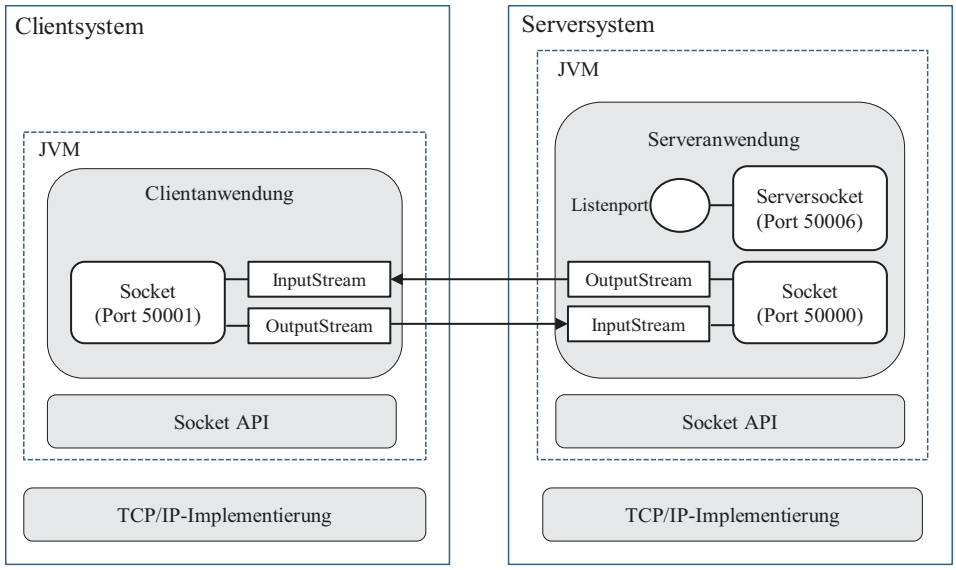


Abb. 5.9 Sockets und Streams in der JVM

zugeordnet, das logisch mit den korrespondierenden Streams der Partneranwendung kommuniziert. Das Senden und Empfangen von Nachrichten verhält sich also wie das Lesen und Schreiben von/in Dateien.

Die Objekte, die man über ObjectOutput-Stream sendet, müssen serialisierbar sein, d. h. die entsprechenden Objektklassen müssen das Interface *Serializable* implementieren. Beim Sender werden die Objekte dann automatisch über den Stream serialisiert und beim Empfänger entsprechend deserialisiert. Object-Streams sorgen dafür, das ganze Objekte gesendet und auch empfangen werden. Der Entwickler muss sich nicht um den mühseligen Aufbau einer Anwendungs-PDU kümmern, sondern definiert seine PDUs als Objektklassen. Das Senden eines Objekts erfolgt mit der Stream-Methode *writeObject*, das Empfangen mit der *readObject*-Methode.

Die Vorgehensweise mit Streams funktioniert standardmäßig nur bei TCP-Sockets, da Datagramm-Sockets nachrichtenorientiert und nicht streamorientiert konzipiert sind.

5.4.3 Verbindungsorientierte Kommunikation über TCP

Um ein Socket auf der Serverseite anzulegen, dass auf ankommende Verbindungen horcht, wird ein Objekt der Standardklasse Serversocket instanziiert. Zur Instanziierung stehen verschiedene Konstruktoren zur Verfügung, von denen einige in Tab. 5.5 kurz beschrieben sind. Hauptunterscheidungsmerkmal ist die Angabe von Adressbestandteilen, um ein Socket an einen TCP-Port zu binden.

Die Socket-Klasse bietet eine ganze Reihe von Methoden, von denen in Tab. 5.6 nur die wichtigsten erwähnt werden.

Tab. 5.5 Konstruktoren für serverseitige Sockets

Konstruktor	Beschreibung
ServerSocket()	Erzeugt serverseitig ein Socket, das noch keine Adresse zugeordnet bekommt.
ServerSocket (int port)	Erzeugt serverseitig ein Socket und bindet gleichzeitig eine Adresse (einen lokalen Port) an das Socket.
ServerSocket (int port, int backlog)	Wie oben. Zudem wird die maximale Länge der Warteschlange für Verbindungsaufbauwünsche (backlog) festgelegt.
ServerSocket (int port, int backlog, InetAddr bindAddr)	Wie oben, nur wird hier neben dem Port noch zusätzlich eine IP-Adresse festgelegt, wenn der Server mehr als eine unterhält und man diese konkretisieren möchte.

Tab. 5.6 Methoden für serverseitige Sockets

Methode	Beschreibung
bind(SocketAddress endpoint)	Binden einer Adresse an ein Socket.
accept()	Führt Listen-Operation durch und akzeptiert ankommende Verbindungsaufbauwünsche. Als Ergebnis wird ein neues Socket für den Zugang zur neuen Verbindung zurückgegeben.
close()	Schließen eines Sockets mit implizitem Verbindungsabbau.
setSoTimeout(int timeout)	Setzt die maximale Wartezeit in ms für den Empfang von Nachrichten über den Input-Stream, um die Blockierungszeit zu begrenzen.
setReuseAddress(boolean on)	Legt fest, ob die gebundene Adresse nach dem *close* sofort wieder benutzt werden kann, ohne den Time-Wait Timeout abzuwarten.
get../set..	Zum Lesen und Verändern von Socket-Attributen.

Um ein Socket auf der Clientseite oder serverseitig für eine konkrete Verbindung zu nutzen, die mit der Methode *accept* angenommen wird, kann ein Objekt der Standardklasse *Socket* instanziiert werden. Die Klasse Socket repräsentiert einen clientseitigen Kommunikationsendpunkt einer TCP-Verbindung. Zur Instanziierung stehen verschiedene Konstruktoren zur Verfügung, von denen einige in Tab. 5.7 kurz beschrieben sind.

Mit einer konkreten Socket-Instanz kann man verschiedene Methoden nutzen (siehe Tab. 5.8).

5.4.4 Gruppenkommunikation über UDP

Gruppenkommunikation ermöglicht die Kommunikation innerhalb einer Gruppe von Anwendungsprozessen. Wenn ein Prozess eine Nachricht sendet, erhalten alle anderen Prozesse derselben Gruppe diese Nachricht. Über die Objektklasse *MulticastSocket*, die wiederum von *DatagramSocket* erbt, wird in Java eine Gruppenkommunikation über UDP unterstützt. Eine Instanz der Klasse *MulticastSocket* ermöglicht also das Empfangen von

Tab. 5.7 Clientseitige Socket-Konstruktoren

Konstruktor	Beschreibung
Socket()	Erzeugt ein Socket, das noch keine Adresse zugeordnet bekommt. Es wird noch kein Verbindungsaufbau vorgenommen.
Socket(InetAddres address, int port) Alternativ: Socket(String host,…	Erzeugt ein Socket und baut eine Verbindung zu der angegebenen Adresse bzw. zum angegebenen Host auf. Es wird ein freier lokaler Port ausgewählt. Anstelle der IP-Adresse kann in einem weiteren Konstruktor auch ein Hostname angegeben werden.
Socket(InetAddress address, int port, InetAddress localAdress, int localPort) Socket(String host, …	Wie oben. Zusätzlich wird die lokale IP-Adresse und der lokale Port vorgegeben. Anstelle der IP-Adresse kann in einem weiteren Konstruktor auch ein Hostname angegeben werden.

Tab. 5.8 Wichtige Methoden für clientseitige Sockets

Methode	Beschreibung
bind(SocketAddress bindpoint)	Binden einer Adresse an ein Socket.

Multicast-Datagrammen. Ein Sender muss nicht unbedingt Gruppenmitglied sein, um eine Nachricht an eine Gruppe zu senden.

Ähnlich wie bei den Datagramm-Sockets gibt es Konstruktoren zum Anlegen von MulticastSockets mit und ohne Adressbindung.

Die Klasse *MulticastSocket* stellt neben den geerbten Methoden aus der Klasse *DatagramSocket* im Wesentlichen Methoden zum Beitritt bzw. Austritt aus einer Multicast-Gruppe bereit:

- *joinGroup*(InetAddress mcastaddr) zum Anbinden an eine Gruppe
- *leaveGroup*(InetAddress mcastaddr) zum Verlassen der Gruppe

Nach Aufruf der Methode *joinGroup* befindet sich ein Anwendungsprozess in der über die angegebene Multicast-Adresse festgelegten Multicast-Gruppe. Die Adresse *mcastaddr* muss eine gültige IP-Multicast-Adresse sein. Ein Datagramm, das mit Hilfe der oben beschriebenen DatagramSocket-Methode *send* gesendet wird, wird durch alle Gruppenmitglieder empfangen, die ihrerseits den Empfang über das lokale Datagram-Socket vornehmen.

5.5 Einfache Java-Beispiele

5.5.1 Ein Java-Beispielprogramm für TCP-Sockets

In Java ist das Streamkonzept für TCP-Klassen sehr komfortabel gelöst. Daten, die über einen TCP-Stream gesendet werden, können über einen sogenannten Objektstrom serialisiert, also in eine Java-Transfersyntax gebracht werden. Man muss nur einen in Java

vordefinierten ObjectInput- und Outputput-Stream über die eigentlichen Input- bzw. Output-Streams legen. Bei Datagramm-Sockets ist dies aber nicht ohne weiteres möglich.

Der folgende Programmrahmen zeigt einen (stark vereinfachten) Server, der Client-Requests nur sequenziell abarbeiten kann. Der Server übernimmt die Verbindung und arbeitet den Request ab. Erst nach der Bearbeitung kann er wieder neue Requests empfangen.

```
/* Java-Rumpf für einen TCP-Server */
ServerSocket server = new ServerSocket(Portnummer);
...
// Beispiel mit einem Thread, der auf einen Verbindungsaufbauwunsch
                                                              wartet,
// den Client bedient, die Verbindung anschließend wieder beendet und
// auf die nächste Anfrage wartet.
while (true) {
   Socket incoming = server.accept();
   ObjectInputStream in;
   ObjectOutputStream out;
   try {
     out = new ObjectOutputStream(incoming.getOutputStream());
     in = new ObjectInputStream(incoming.getInputStream());
     // Empfangen über Inputstream
     MyPDU pdu = (MyPDU) in.readObject();
     // Empfangene PDU verarbeiten
     // ...
     // Senden über OutputStream
     // MyPDU ist eine eigene Objektklasse
     out.writeObject(new MyPDU(...));
     // Stream und Verbindung schließen
     incoming.close();
     }
     catch (Exception e) { ... }
}
...
```

Die Klasse *MyPDU* repräsentiert hier eine beliebige Java-Klasse, in der die Nachricht beschrieben ist. Diese Klasse muss durch die Implementierung des Java-Interfaces *Serializable* als serialisierbar deklariert werden. Der stark vereinfachte Clientcode ist wie folgt aufgebaut:

```
/* Java-Rumpf für einen TCP-Client */
// Client sendet nur eine Anfrage
...
  try {
    Socket client = new Socket (Host, Portnummer);
    ObjectInputStream in =
      new ObjectInputStream(client.getInputStream();
```

```
        ObjectOutputStream out =
          new ObjectOutputStream(client.getOutputStream());
        // Senden über OutputStream
        // MyPDU ist eine eigene Objektklasse
        out.writeObject(new MyPDU(...));

        // Empfangen über Inputstream
        MyPDU pdu = (MyPDU) in.readObject();
        // Response verabeiten
        // Stream und Verbindung schließen
        incoming.close();
    }
...
```

Die Streams sind quasi Bestandteil eines Socket-Objekts und können dort direkt angesprochen werden. Um serialisierte Daten zu übertragen, muss man den Input- und Output-Stream jeweils um einen ObjectStream (Input bzw. Output) ergänzen. Alternativ zur hier im Beispielcode angedeuteten seriellen Abarbeitung von Requests könnte man für die Bearbeitung nebenläufiger Requests Java-Threads nutzen. Verbindungen würden dann über eigene Threads angenommen und bearbeitet werden.

5.5.2 Ein Java-Beispielprogramm für Datagramm-Sockets

Im Folgenden ist ein Beispielprogramm für eine einfache Echo-Anwendung auf Basis von UDP-Datagramm-Sockets dargestellt. Es sind die Klassen *UDPEchoServer* und *UDPEchoClient* definiert, die die eigentliche Arbeit ausführen. Beide Klassen sind im Package *UDPEchoExample* angelegt. In den Konstruktoren werden die Adressen (Portnummer und beim Client auch der Hostname) angegeben. Die Parameter werden in der *main*-Methode jeweils über die Startzeile an die Programme übergeben. Die weitere Interpretation der Programme ist dem Leser überlassen.

```
/* Java UDP-Server */
package UDPEchoExample;
import java.net.*;
import java.io.*;

public class UDPEchoServer {
protected DatagramSocket socket;
public UDPEchoServer (int port) throws IOException {
   socket = new DatagramSocket (port);
}

public void execute() throws IOException {
  while (true) {
    DatagramPacket packet = receive();
```

```
        sendEcho (packet.getAddress (), packet.getPort (),
           packet.getData(), packet.getLength());
    }
}

protected DatagramPacket receive () throws IOException {
  byte buffer[] = new byte[65535];
  DatagramPacket packet = new DatagramPacket (buffer,
    buffer.length);
  socket.receive(packet);
  System.out.println ("Received " + packet.getLength ()
+ " bytes.");
  return packet;
}

    protected void sendEcho (InetAddress address,
      int port, byte data[], int length) throws IOException
    {
        DatagramPacket packet =
          new DatagramPacket (data, length, address, port);
        socket.send (packet);
        System.out.println („Response sent");
    }

    public static void main (String args[]) throws IOException
    {
      if (args.length != 1) {
       throw new RuntimeException ("Syntax: UDPEchoServer <port>");

       UDPEchoServer echo = new UDPEchoServer (Integer.parseInt (args[0]));
       echo.execute();
        }
    }
}
```

Der zugehörige Client kann vereinfacht wie folgt aufgebaut sein:

```
/* Java UDP-Client */
package UDPEchoExample;
import java.net.*;
import java.io.*;
public class UDPEchoClient
{
  protected DatagramSocket socket;
```

```
protected DatagramPacket packet;
public UDPEchoClient (String message,
  String host, int port) throws IOException
{
  socket = new DatagramSocket ();
  // Datagramm aus message, host und port aufbauen
  // (hier nicht weiter ausgeführt) …
    try {
      sendPacket ();
      receivePacket ();
    }
    finally {
     socket.close ();
    }
}

 protected void sendPacket () throws IOException
{
  socket.send (packet); ...
}

 protected void receivePacket () throws IOException
{
   byte buffer[] = new byte[65535];
   DatagramPacket packet = new DatagramPacket (buffer, buffer.length);
   socket.receive (packet);
   ByteArrayInputStream byteI =
     new ByteArrayInputStream (packet.getData (), 0, packet.getLength
                                                                  ());
   DataInputStream dataI = new DataInputStream (byteI);
   String result = dataI.readUTF();
}

 public static void main (String args[]) throws IOException
 {
   if (args.length != 3) {
    throw new RuntimeException („EchoClient <host> <port> <message>");
   }
   while (true) {
    new UDPEchoClient (args[2], args[0], Integer.parseInt (args[1]);
        try {
          Thread.sleep (1000);
```

```
            }
            catch (InterruptedException ex)
            {
             // Exception Handling vernachlässigt ...
            }
        }
    }
}
```

5.5.3 Ein Java-Beispiel für Multicast Sockets

Das folgende stark vereinfachte Beispielprogrammstück soll nur prinzipiell aufzeigen, wie Multicast über Java funktioniert. Über die Methode *joinGroup* verbindet sich ein Anwendungsprozess mit einer vorgegebenen IPv4-Multicast-Adresse (hier 228.5.5.5) mit der Gruppe, sendet eine Nachricht in die Gruppe, empfängt eine Nachricht und verlässt anschließend die Gruppe wieder.

Alle angemeldeten Anwendungsprozesse erhalten die Nachricht, nach UDP-Konventionen allerdings ungesichert.

```
/* Java-Multicast-Programm */
...
public class UDPMulticastClient
  ...
  public static void main (String args[]) throws IOException
    {
      String message = "Testnachricht";
      InetAddress group = InetAddress.getByName("228.5.5.5");
      MulticastSocket s = new MulticastSocket(51000);
      s.joinGroup(group);
      DatagramPacket datagram = new DatagramPacket(message.getBytes(),
            msg.length(), group, 51000);
      s.send(datagram);
      byte[] inBuffer = new byte[1000];
      DatagramPacket recv = new DatagramPacket(inBuffer, inBuffer.length);
      s.receive(recv);
      …
      s.leaveGroup(group);
      …
    }
}
```

IP-Multicast
Das *Internet Group Management Protocol* (*IGMP*) basiert direkt auf IP und dient der Verwaltung von Gruppen, die gemeinsam über eine IPv4-Multicast-Adresse identifiziert werden und daher über eine IPv4-Multicast-Nachricht adressiert werden können (RFC 3376, IGMPv3).

IGMP bietet Funktionen, mit denen ein IPv4-fähiger Rechner einem Router mitteilt, dass er in der Lage ist, IP-Pakete aus einer Multicast-Gruppe zu empfangen. Für das IPv4-Multicasting werden im Internet die IPv4-Adressen 224.0.0.0 – 239.255.255.255, auch als Klasse-D-Adressen bezeichnet, verwendet. IP-Router benötigen für das Routing dieser Nachrichten an alle Beteiligten IPv4-Rechner spezielle Multicast-Routing-Protokolle wie *DVMRP* oder *MOSPF* (Tanenbaum und Wetherall 2011).

In IPv6 gibt es spezielle Multicast-Adressbereiche. Multicasting ist ähnlich wie bei IPv4 über das Steuerprotokoll ICMPv6 geregelt. Es beinhaltet auch eine sogenannte Multicast Listener Discovery-Funktion (MLD).

5.5.4 Zusammenspiel von Socket API und Protokollimplementierung

In den vorangegangenen Kapiteln wurde schon mehrfach auf das Zusammenspiel der Socket API mit der TCP-Instanz eingegangen. Am Beispiel von TCP sollen nochmals die wichtigsten Aspekte zusammengefasst werden.

Bei einem Methodenaufruf an der Socket API durch eine Anwendung wird immer ein Auftrag an die lokal zuständige TCP-Instanz abgesetzt. Wie die TCP-Instanz den Auftrag tatsächlich bearbeitet bzw. wann dies zum Senden eines TCP-Segments führt, hängt vom konkreten Zustand der TCP-Instanz und von deren Konfiguration ab.

Die Konstruktion einer *ServerSocket*-Instanz bewirkt kein Senden einer Nachricht, sondern hat vielmehr nur lokale Bedeutung. Durch Aufruf eines *ServerSocket*-Konstruktors wird ein Socket eingerichtet, dass an einem lokalen Port auf ankommende Verbindungsaufbauwünsche warten kann. Je nach Nutzung des Konstruktors kann schon ein Port gebunden werden oder er wird erst nachträglich über die *bind*-Methode bekanntgemacht. Ohne einen Port kann das Socket nicht aktiv arbeiten.

Die Konstruktion eines Sockets auf der Clientseite erfolgt über den Aufruf eines Konstruktors der Socket-Klasse. Je nachdem, welchen Konstruktor man verwendet, wird nach dem Konstruieren einer Socket-Instanz auch gleich ein Verbindungsaufbau initiiert oder nicht. Der Port kann gleich im Konstruktor oder später durch die *bind*-Methode festgelegt werden, es kann aber auch ein beliebiger freier Port vergeben werden, was der Normalfall ist. In letzterem Fall ist als lokaler Port 0 anzugeben oder der Socket-Kostruktor zu verwenden, bei dem man den lokalen Port nicht angeben muss.

Wenn gleich bei der Konstruktion des Sockets eine Verbindung aufgebaut wird, muss die *connect*-Methode nicht mehr explizit aufgerufen werden. Der Konstruktor gibt die Kontrolle erst wieder nach einem erfolgreichem oder nicht erfolgreichem Verbindungsaufbau an den Aufrufer zurück. Der Dreiwege-Handshake wird also vollständig ausgeführt. Mit dem *connect*-Aufruf auf der aktiven Seite (Clientseite) kann aktiv ein Dreiwege-Handshake eingeleitet werden.

Auf der Serverseite wird der von der klassischen POSIX-API bekannte *listen*-Aufruf implizit in der *accept*-Methode durchgeführt. Bei Aufruf von *accept* wird standardmäßig auf einen Verbindungsaufbauwunsch gewartet, das heißt ein Client muss ein TCP-Segment

mit gesetztem SYN-Flag senden. Ein ankommender Verbindungsaufbauwunsch führt dann zum Aufbau eines Verbindungskontexts. Die Anfrage wird über ein TCP-Segment mit gesetztem FIN- und ACK-Flag beantwortet, ohne dass der Anwendungsprozess dies zunächst mitbekommt. Erst wenn der Dreiwege-Handshake vollständig durchgeführt ist, wird dem Anwendungsprozess ein Verbindungssocket für die neu aufgebaute Verbindung übergeben. Die Verbindung steht dann auf der Transportebene. Die Authentifizierung muss im Anwendungsprotokoll erfolgen, hierfür ist TCP nicht zuständig. Der Vorgang bis zur etablierten Verbindung ist in Abb. 5.10 skizziert. Die Socket-Optionen können in der Java Socket API über konkrete *ServerSocket*- und *Socket*-Instanzen eingestellt und abgefragt werden (siehe set/get-Methoden, z. B. *setSoTimeout*). Sie gelten für das Socket und die über das Socket aufgebaute Verbindung (z. B. Puffereinstellungen für Sende- und Empfangspuffer). In der Skizze werden Optionen vernachlässigt.

Der Aufruf der Sendeprimitive *write*- oder *writeObject* (je nach Stream-Nutzung) bedeutet noch nicht, dass die Daten sofort gesendet werden. Die Anwendungsnachrichten könnten auch erst einmal gesammelt werden, bis der Sendepuffer gefüllt ist (siehe hierzu Nagle- und Karn-Algorithmus und TCP-Timer).

Bei Nutzung von Java-Objektströmen erfolgt die Serialisierung der Objekte im Java-Laufzeitsystem. Wann konkret gesendet wird, ist im Protokoll und in der konkreten Implementierung festgelegt. Der *read*- oder *readObject*-Aufruf (je nach Stream-Nutzung) liefert anstehende, bereits empfangene Daten aus dem Empfangspuffer. Ist keine Nachricht im Empfangspuffer, wird gewartet. Nach Empfang der Nachricht kann auch ein TCP-Segment mit gesetztem ACK-Flag und gesetzter Bestätigungsnummer abgesetzt werden, Wann dies genau passiert, entscheidet wiederum die TCP-Instanz. Das TCP-Segment dient dann auch zur Anpassung der Fenstergröße, sofern sich hier etwas verändert hat. Die dem Anwendungsprozess zugestellten Nachrichten werden in der Java-Umgebung wieder zu Objekten zusammengebaut (deserialisiert).

Es können auch TCP-Segmente mit Daten bei einer TCP-Instanz ankommen, die nicht gleich vom Anwendungsprozess über einen read-Aufruf gelesen werden. Diese verbleiben

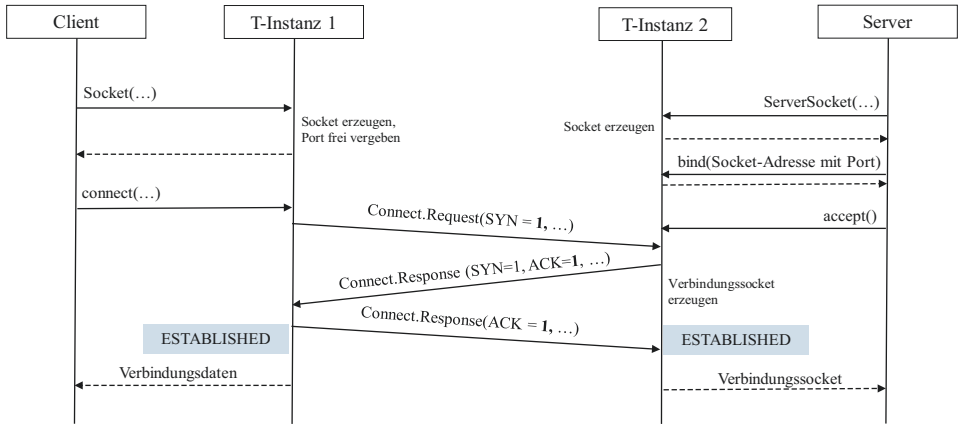


Abb. 5.10 Zusammenspiel Java Socket API und TCP-Instanz beim Verbindungsaufbau

im Empfangspuffer. Gegebenenfalls wird dann implizit von der TCP-Instanz ein TCP-Segment mit gesetztem ACK-Flag, der Bestätigungsnummer und einer Aktualisierung der Fenstergröße gesendet. Die Bestätigung kann also jederzeit erfolgen, der empfangende Anwendungsprozess hat die Daten dann noch nicht gesehen. Das Senden und Empfangen von Nachrichten ist vereinfacht in Abb. 5.11 dargestellt.

Der *close*-Aufruf führt zum Anstoß des Verbindungsabbaus, wobei zunächst ein TCP-Segment mit gesetztem FIN-Flag gesendet wird. Auf der passiven Seite muss dann ebenfalls zeitnah ein *close*-Aufruf erfolgen. Dafür muss das darüberliegende Anwendungsprotokoll für eine Synchronisation des Verbindungsabbaus auf logischer Ebene sorgen, z. B. indem es entsprechende Nachrichten vorsieht. Sonst könnte es passieren, dass die Verbindungskontexte des passiv agierenden Partners (hier des Servers) zu lange im Zustand CLOSE_WAIT verbleiben (Abb. 5.12).

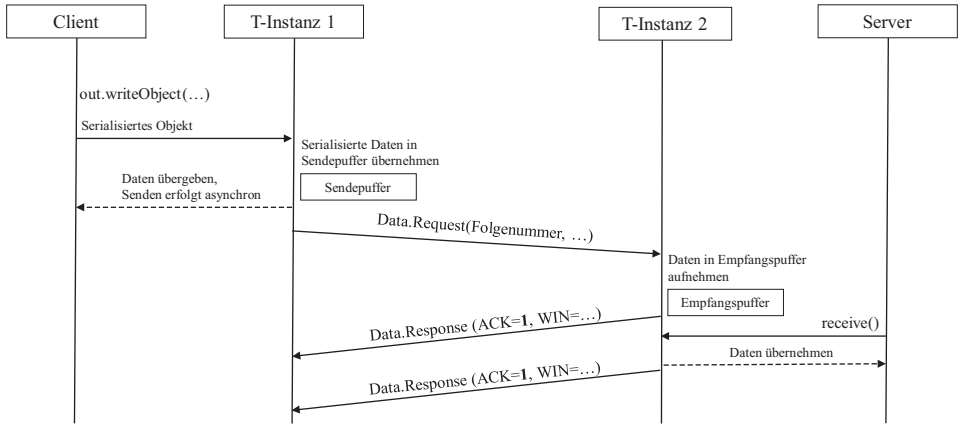


Abb. 5.11 Zusammenspiel Java Socket API und TCP-Instanz beim Datenaustausch

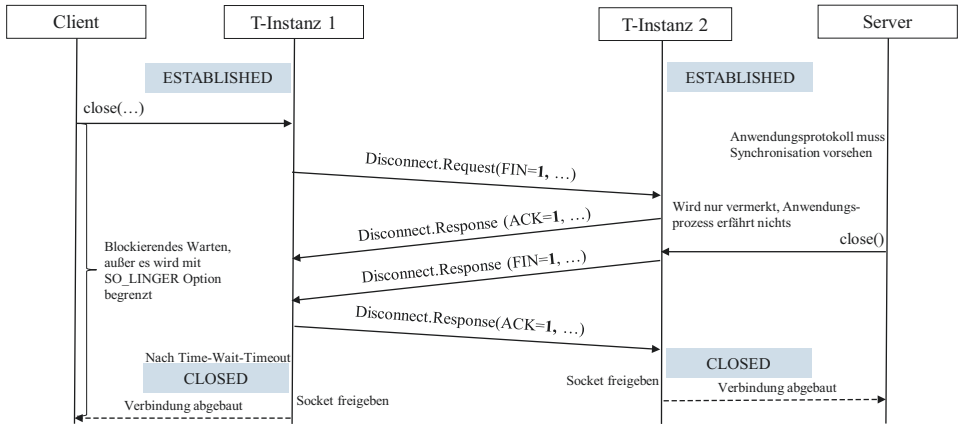


Abb. 5.12 Zusammenspiel Java Socket API und TCP-Instanz beim Verbindungsabbau

5.6 Ein Mini-Framework für Java-Sockets

Für die Programmierung von Kommunikationsanwendungen kann man die Nutzung der Socket API vereinfachen, indem man einige Aufgaben generisch in vordefinierte Objektklassen implementiert. In diesem Kapitel stellen wir ein Mini-Framework zur Nutzung von TCP-Sockets in der Programmiersprache Java vor, das einige Erleichterungen verschafft. Ausgehend von einer Abstraktion der konkreten Transportzugriffsschnittstelle über Java-Interfaces wird eine TCP-Implementierung einiger Basisklassen gezeigt.

Drei vordefinierte Java-Interfaces abstrahieren auf einfache Weise eine logische Verbindung zwischen zwei Partneranwendungen. Die Schnittstellen lassen sich auch auf Basis von Datagramm-Sockets implementieren, sofern bei einer UDP-Implementierung die Verwaltung eines Verbindungskontextes über ein eigenes Verbindungsauf- und Verbindungsabbauprotokoll ergänzt wird. Inwieweit TCP-Mechanismen wie Folgenummern, Duplikatserkennung usw. nachgebildet werden müssen, hängt von den Anforderungen der zu implementierenden Anwendung ab.

Das Mini-Framework soll eine Anregung für die eigene Entwicklung und für eigene Experimente sein und kein universelles Framework für die Entwicklung verteilter Anwendungen.

Zunächst wird ein Überblick über die vordefinierten Interfaces und Objektklassen des Mini-Frameworks gegeben. Anschließend wird anhand einfacher Nutzungsbeispiele aufgezeigt, wie man das Mini-Framework einsetzen kann. Der im Folgenden auszugsweise skizzierte Sourcecode wurde um Kommentare und Logging-Ausgaben bereinigt. Eigene Exception-Klassen werden auch nicht weiter erläutert und können im Sourcecode eingesehen werden.[8]

5.6.1 Überblick über vordefinierte Schnittstellen und Objektklassen

Das in Abb. 5.13 skizzierte Klassenmodell zeigt die wichtigsten Zusammenhänge des Mini-Frameworks für eine konkrete Implementierung auf Basis von TCP-Sockets.

Im Mini-Framework wird von einer abstrakten Verbindung ausgegangen, die in der TCP-Implementierung konkretisiert wird. Drei Java-Intefaces abstrahieren die Methoden für den Verbindungsaufbau und für das Senden und Empfangen von Nachrichten.

Das Interface *Connection* (siehe Programmcode zum generischen Connection-Interface) beschreibt allgemein die Schnittstelle einer Verbindung unabhängig vom verwendeten Transportprotokoll und bietet generische Methoden zum Verbindungsmanagement (*connect*, *close*) Senden (*send*) und Empfangen (*receive*) von serialisierten Objekten beliebigen Objekttyps an. Es ist zu empfehlen, die individuellen PDUs eines zu implementierenden Anwendungsprotokolls immer in einer eigenen Objektklasse zu definieren. Diese Objektklasse sollte dann auch die anwendungsspezifischen Steuerinformationen enthalten. Das Mini-Framework ist auf die Kommunikation zwischen Java-Partnern ausgelegt,

[8] Der komplette Sourcecode des Mini-Frameworks ist unter Github zum Download verfügbar.

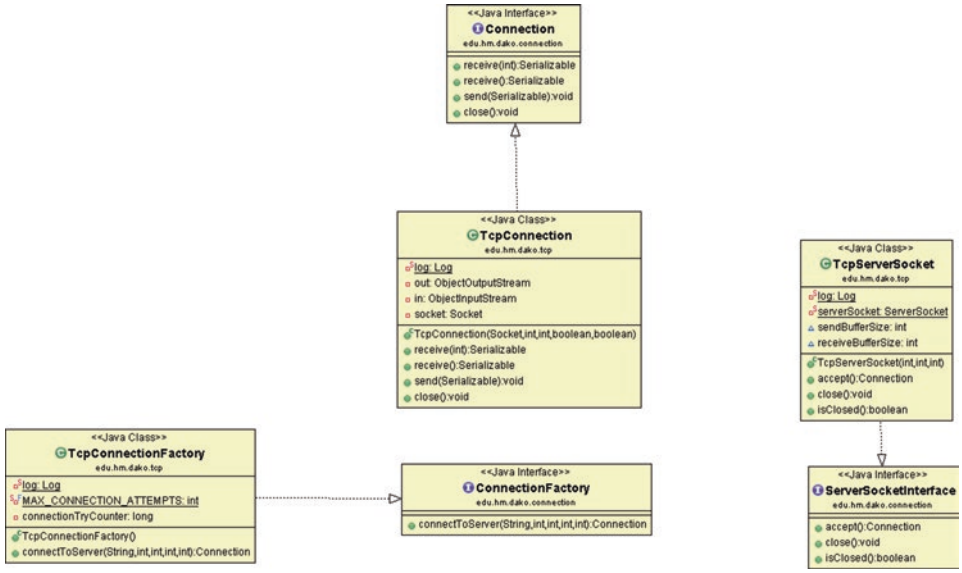


Abb. 5.13 Klassenmodell des Frameworks für die TCP-Kommunikation

weshalb eine Java-Objektserialisierung der Anwendungs-PDUs möglich ist. Man muss sich nicht um eigene Mechansimen zur Serialisierung und Deserialisierung kümmern.

```
/* Generisches Connection-Interfaces *(
package edu.hm.dako.connection;
import java.io.IOException;
import java.io.Serializable;

public interface Connection {
  public Serializable receive(int timeout) throws Exception;
  public Serializable receive() throws Exception;
  public void send(Serializable message) throws Exception;
  public void close() throws Exception;
}
```

Das Interface *ConnectionFactory* (siehe folgenden Programmcode) stellt eine Schnittstelle bereit, um clientseitig eine Verbindung zu einem Server aufzubauen. Hier wird das Factory-Pattern verwendet, um Verbindungen einheitlich zu erzeugen, wofür die Methode *connectToServer* bereitgestellt wird. In dieser Methode werden die lokalen und entfernten Adressen und auch die Puffergrößen für die Verbindung festgelegt.

```
/* Generisches ConnectionFactory-Interfaces */
package edu.hm.dako.connection;

public interface ConnectionFactory {
 public Connection connectToServer(String remoteServerAddress,
```

```
    int serverPort, int localPort, int sendBufferSize,
    int receiveBufferSize) throws Exception;
}
```

Das Interface *ServerSocket* (siehe folgenden Programmcode) stellt eine Schnittstelle für Serveranwendungen bereit, über deren Nutzung auf Verbindungsaufbauwünsche von Clients gewartet werden kann und mit der Verbindungen angenommen werden können (Methode *accept*).

```
/* Generisches ServerSocket-Interfaces */
package edu.hm.dako.connection;
public interface ServerSocketInterface {
  Connection accept() throws Exception;
  public void close() throws Exception;
  public boolean isClosed();
}
```

5.6.2 Basisklassen zur TCP-Kommunikation

Die vorhandenen Java-Interfaces wurden im Mini-Framework konkret für die TCP-Kommunikation auf Basis von TCP-Sockets implementiert. Das Interface *Connection* (siehe folgenden Programmcode) wird in der Klasse *TCPConnection* bereitgestellt. Die Klasse stellt Methoden für den Verbindungsauf- und –abbau sowie für das Senden und Empfangen von beliebigen Java-Objekten bereit. Die als Nachrichten vewendete Java-Objekte müssen lediglich serialisierbar sein, was durch die Implementung des Java-Standard-Interfaces *Serializable* gegeben ist. Die *receive*-Methode kann auch zeitlich über eine Zeitangabe begrenzt werden, so dass sie nicht ewig blockiert, wenn keine Nachrichten ankommen.

Beim Einrichten der Verbindung im Konstruktor können die lokalen Sende- und Empfangspuffergrößen und auch die TCP-Optionen *KeepAlive* und *NoDelay* angegeben werden. Weitere Optionen können im Konstruktor erprobt werden.

```
/* Connection-Implementierung auf Basis von TCP */
package edu.hm.dako.tcp;
import …
public class TcpConnection implements Connection {
  private ObjectOutputStream out;
  private ObjectInputStream in;
  private Socket socket;

    public TcpConnection(Socket socket, int sendBufferSize,
        int receiveBufferSize,boolean keepAlive, boolean TcpNoDelay) {
      this.socket = socket;
```

```
  try {
     out = new ObjectOutputStream(socket.getOutputStream());
     in = new ObjectInputStream(socket.getInputStream());
     socket.setReceiveBufferSize(receiveBufferSize);
     socket.setSendBufferSize(sendBufferSize);
     // TCP-Optionen einstellen
     socket.setTcpNoDelay(TcpNoDelay);
     socket.setKeepAlive(keepAlive);
     …

    } catch (SocketException e) {
      throw new RuntimeException(e);
    } catch (IOException e) {
      throw new RuntimeException(e);
    }
}

@Override
public Serializable receive(int timeout)
  throws IOException, ConnectionTimeoutException, EndOfFileExcep-
                                                              tion {
    if (!socket.isConnected()) {
     throw new EndOfFileException(new Exception());
    }
    socket.setSoTimeout(timeout);
    try {
     Object message = in.readObject();
     socket.setSoTimeout(0);
     return (Serializable) message;
    } catch (java.net.SocketTimeoutException e) {
      throw new ConnectionTimeoutException(e);
    } catch (java.io.EOFException e) {
      throw new EndOfFileException(e);
    } catch (Exception e) {
      throw new EndOfFileException(e);
    }
}

@Override
public Serializable receive() throws IOException, EndOfFileException,
        IOException {

    if (!socket.isConnected()) {
     throw new EndOfFileException(new Exception());
    }
```

```
        try {
         socket.setSoTimeout(0);
         Object message = in.readObject();
         return (Serializable) message;
        } catch (java.io.EOFException e) {
         throw new EndOfFileException(e);
        } catch (Exception e) {
         throw new IOException();
        }
    }

    @Override
    public void send(Serializable message) throws IOException {
     if (socket.isClosed()) {
      throw new IOException();
        }
        if (!socket.isConnected()) {
         throw new IOException();
        }
        try {
         out.writeObject(message);
        } catch (Exception e) {
          throw new IOException();
        }
    }

    @Override
    public synchronized void close() throws IOException {
      try {
        socket.getPort());
        socket.close();
        } catch (Exception e) {
          socket.getInetAddress());
          throw new IOException(new IOException());
        }
    }
}
```

Das Interface *ConnectionFactory* (siehe folgenden Programmcode) wird in der Klasse *TcpConnectionFactory* implementiert. Sollte ein Verbindungsaufbau etwa aufgrund eines nicht vorhandenen Servers nicht sofort möglich sein, wird er maximal 50 Mal wiederholt. Die in der Methode *connectToServer* übergebenen Parameter enthalten unter anderem die Adresse des entfernten Partners. Die Parameter mit lokaler Bedeutung (lokaler Port und Puffergrößen) werden an die lokale *Connection*-Instanz weitergegeben. Gibt man als lokalen Port den Wert 0 an, wird implizit ein freier TCP-Port vergeben.

```
/* Factory-Implementierung auf Basis von TCP */
package edu.hm.dako.tcp;
import …

public class TcpConnectionFactory implements ConnectionFactory {
    private static final int MAX_CONNECTION_ATTEMPTS = 50;
    private long connectionTryCounter = 0;

    public Connection connectToServer(String remoteServerAddress,
            int serverPort, int localPort, int sendBufferSize,
            int receiveBufferSize) throws IOException {
        TcpConnection connection = null;
        boolean connected = false;
        InetAddress localAddress = null;
        int attempts = 0;
        while ((!connected) && (attempts < MAX_CONNECTION_ATTEMPTS)) {
            try {
                connectionTryCounter++;
                connection = new TcpConnection(
                        new Socket(remoteServerAddress, serverPort,
                                                    localAddress,
                         localPort), sendBufferSize, receiveBufferSize,
                            false, true);
                connected = true;
            } catch (BindException e) {

                // Lokaler Port schon verwendet
            } catch (IOException e) {
                // Ein wenig warten und erneut versuchen
                attempts++;
                try {
                    Thread.sleep(100);
                } catch (Exception e2) {
                }

            } catch (Exception e) {
                throw new IOException();
            }
            if (attempts >= MAX_CONNECTION_ATTEMPTS) {
                throw new IOException();
            }
        }
        return connection;
    }
}
```

Die Objektklasse *TCPServerSocket* stellt eine konkrete Implementierung des Interfaces *ServerSocket* dar. Bei Ankunft eines Verbindungsaufbauwunsches wird in der *accept*-Methode ein lokales Socket für die Verbindung erzeugt.

```
/* ServerSocket-Implementierung auf Basis von TCP */
package edu.hm.dako.tcp;
import …

public class TcpServerSocket implements ServerSocketInterface {
    private static java.net.ServerSocket serverSocket;
    int sendBufferSize;
    int receiveBufferSize;
    public TcpServerSocket(int port, int sendBufferSize, int
            receiveBufferSize)throws BindException, IOException {
        this.sendBufferSize = sendBufferSize;
        this.receiveBufferSize = receiveBufferSize;
        try {
            serverSocket = new java.net.ServerSocket(port);
        } catch (BindException e) {
            throw e;
        } catch (IOException e) {
            throw e;
        }
    }

    @Override
    public Connection accept() throws IOException {
      return new TcpConnection(serverSocket.accept(), sendBufferSize,
            receiveBufferSize,false, true);
    }

    @Override
    public void close() throws IOException {
        serverSocket.close();
    }

    @Override
    public boolean isClosed() {
        return serverSocket.isClosed();
    }
}
```

5.6.3 Single-threaded Echo-Server als Beispiel

In diesem Beispiel werden die oben beschriebenen Framework-Klassen in einer einfachen Client-/Server-Anwendung, die einen Echo-Request über TCP ausführt, angewendet. Der Server wird zunächst *single-threaded* implementiert, d. h. er kann zu einer Zeit nur einen Client bedienen.

Der Client *EchoTcpClient* (siehe folgenden Programmcode) baut eine TCP-Verbindung zum Server *EchoTcpSingleThreaded* auf und sendet einen Echo-Request in einer einfachen PDU (*SimplePDU*) an den Server. Der Server wartet am TCP-Port 50000) und der Client nutzt eine frei vergebene lokale Portnummer. Der Client wartet auf eine Echo-Response-Nachricht als Antwort vom Server. Dieser Vorgang wird mehrfach wiederholt, wonach der Client die Verbindung schließt und sich beendet. Ebenso beendet sich in unserem Beispiel der Server, sobald er merkt, dass die Verbindung zum Client nicht mehr steht. Dies wird im blockierenden *receive*-Aufruf erkannt, da die Socket API in diesem Fall eine Exception wirft, wenn der Partner die Verbindung geschlossen hat. Das Klassendiagramm für die Anwendung ist in Abb. 5.14 skizziert.

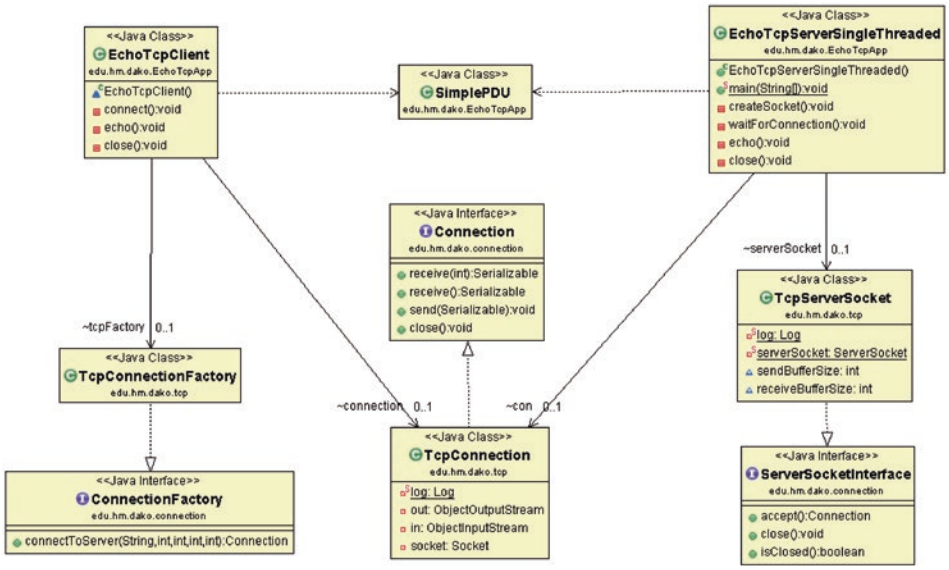


Abb. 5.14 Klassenmodell für die Echo-Beispielanwendung (Single-threaded Server)

```
/* Echo-Client */
package edu.hm.dako.EchoTcpApp;
import …

public class EchoTcpClient {
    static final int NR_OF_MSG = 10; // Anzahl zu sendender Nachrichten
    static final int MAX_LENGTH = 100; // Nachrichtenlaenge
    TcpConnectionFactory tcpFactory = null;
    TcpConnection connection = null;

    EchoTcpClient() {
        tcpFactory = new TcpConnectionFactory();
        System.out.println("Client gestartet");
    }

    public static void main(String[] args) {
        EchoTcpClient client = new EchoTcpClient();
        try {
            client.connect();
            for (int i = 0; i < NR_OF_MSG; i++) {
                client.echo();
            }
            client.close();
        } catch (Exception e) {
            System.exit(1);
        }
    }

    private void connect() throws Exception {
        try {
            connection = (TcpConnection) tcpFactory.connectToServer
    ("localhost", 50000, 0, 400000, 400000);
            System.out.println("Verbindung steht");
        } catch (Exception e) {
            System.out.println("Exception during connect");
            throw new Exception();
        }
    }

    private void echo() throws Exception {
        SimplePDU requestPDU = createMessage();
        try {
            connection.send(requestPDU);
            SimplePDU responsePDU = (SimplePDU) connection.receive();
        } catch (Exception e) {
            throw new Exception();
        }
    }
```

```
    private void close() throws Exception {
        try {
            connection.close();
            System.out.println("Verbindung abgebaut");
        } catch (Exception e) {
            throw new Exception();
        }
    }

    private static SimplePDU createMessage() {
        char[] charArray = new char[MAX_LENGTH];
        for (int j = 0; j < MAX_LENGTH; j++) {
            charArray[j] = 'A';
        }
        SimplePDU pdu = new SimplePDU(String.valueOf(charArray));
        return (pdu);
    }
}
```

Wie man sieht, nutzt der Client die Klasse *TCPConnection* und auch die Klasse *TCP-ConnectionFactory*, der Server ebenfalls die Objektklasse *TCPConnection* sowie die Objektklasse *TCPServerSocket*.

Der Echo-Server (siehe folgenden Programmcode) wartet auf einen Verbindungsaufbau und beantwortet dann alle Echo-Requests eines Clients, bevor er sich wieder beendet.

```
/* Single-Threaded Echo-Server */
package edu.hm.dako.EchoTcpApp;
import …

public class EchoTcpServerSingleThreaded {
    TcpServerSocket serverSocket = null;
    TcpConnection con = null;

    public static void main(String[] args) {
       EchoTcpServerSingleThreaded server = new EchoTcpServerSingleThreaded();
        try {
            server.createSocket();
            server.waitForConnection();
            while (true) {
                server.echo();
            }
        } catch (Exception e) {
```

```
            System.out.println("Exception beim Echo-Handling");
            server.close();
        }
    }

    private void createSocket() throws Exception {
        try {
            serverSocket = new TcpServerSocket(50000, 400000, 400000);
        } catch (Exception e) {
            System.out.println("Exception");
            throw new Exception();
        }
    }

    private void waitForConnection() throws Exception {
        try {
            con = (TcpConnection) serverSocket.accept();
            System.out.println("Verbindung akzeptiert");
        } catch (Exception e) {
            System.out.println("Exception");
            throw new Exception();
        }
    }

    private void echo() throws Exception {
        try {
            SimplePDU receivedPdu = (SimplePDU) con.receive();
            String message = receivedPdu.getMessage();
            con.send(receivedPdu);
        } catch (Exception e) {
            System.out.println("Exception beim Empfang");
            throw new Exception();
        }
    }

    private void close() {
        try {
            con.close();
            System.out.println("Verbindung geschlossen");
        } catch (Exception e) {
            System.out.println("Exception beim close");
        }
    }
```

5.6.4 Multi-threaded Echo-Server als Beispiel

Der Echo-Server wurde in diesem Beispiel so erweitert, dass er mehrere nebenläufige Verbindungen unterhalten kann. Jede Verbindung zwischen einem Client und einem Server wird serverseitig über einen eigenen Workerthread bearbeitet, der so lange lebt, bis die Verbindung wieder abgebaut wird.

Der Server beendet sich nach einem Verbindungsabbau eines Clients nicht und wartet auf den Verbindungsaufbauwunsch eines weiteren Clients. Lediglich bei schwerwiegenden Fehlern beendet sich auch der Server. Auf der Clientseite ändert sich im Vergleich zur single-threaded Lösung nichts. In Abb. 5.15 ist das Klassendiagramm für diese Anwendung skizziert.

Für die Bearbeitung aller Threads eines Clients wird eine eigene Thread-Klasse bereitgestellt. Für die Bearbeitung einer Verbindung wird ein neuer Thread erzeugt und gestartet (siehe vordefinierte *start*-Methode). Im Programmcode des Servers (siehe folgenden Programmcode) ist zu sehen, dass eine akzeptierte Verbindung mit einem Client sofort an eine neue Instanz von *EchoWorkerThread* übergeben wird.

```
/* Multithreaded Echo-Server */
package edu.hm.dako.EchoTcpApp;
import …

public class EchoTcpServerMultihreaded {
    TcpServerSocket serverSocket = null;
    TcpConnection con = null;
```

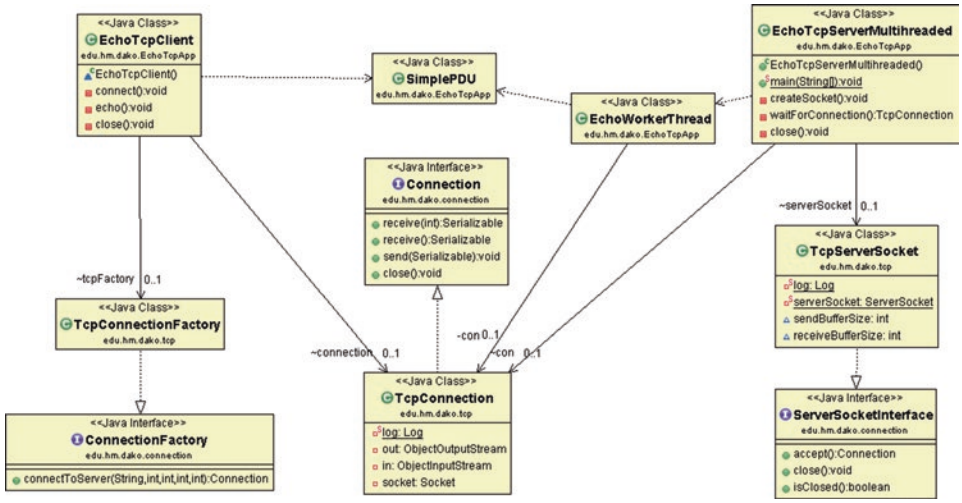


Abb. 5.15 Klassenmodell für die Echo-Beispielanwendung (Multi-threaded Server)

```
public static void main(String[] args) {
    System.out.println("Server gestartet");
   EchoTcpServerMultihreaded server = new EchoTcpServerMultihreaded();
    try {
        server.createSocket();
    } catch (Exception e) {
      System.out.println("Exception beim Erzeugen des Server-Sockets");
        System.exit(1);
    }

    boolean listening = true;
    while (listening) {
        try {
          System.out.println("Server wartet auf Verbindungsanfragen ...");
          TcpConnection con = server.waitForConnection();
          EchoWorkerThread w1 = new EchoWorkerThread(con);
          w1.start();
        } catch (Exception e3) {
          System.out.println("Exception in einem Workerthread");
          listening = false;
          server.close();
        }
    }
}

private void createSocket() throws Exception {
    try {
        serverSocket = new TcpServerSocket(50000, 400000, 400000);
    } catch (Exception e) {
        throw new Exception();
    }
}

private TcpConnection waitForConnection() throws Exception {
    try {
        TcpConnection con = (TcpConnection) serverSocket.accept();
        System.out.println("Verbindung akzeptiert");
        return (con);
    } catch (Exception e) {
        throw new Exception();
    }
}

private void close() {
    try {
```

```
            con.close();
            System.out.println("Verbindung geschlossen");
        } catch (Exception e) {
            System.out.println("Exception beim close");
        }
    }
}
```

Der Programmcode für den Workerthread (siehe unten) sieht vor, dass alle Echo-Requests eines Clients selbständig bearbeitet werden. Wenn der Client die Verbindung abbaut, wird auch der Thread beendet. Wie in Java üblich, kann ein Thread durch die Vererbung von der Standardklasse *Thread* programmiert werden (Mandl 2014). In der zu überschreibenden Methode *run* ist die eigentliche Logik des Threads zu programmieren. Bei Aufruf der Methode *start* im Server wird implizit die Methode *run* des entsprechenden Threads aufgerufen.

```
/* Workerthread für Echo-Bearbeitung */
package edu.hm.dako.EchoTcpApp;
import …

public class EchoWorkerThread extends Thread {

    private static int nrWorkerThread = 0;
    private TcpConnection con;
    private boolean connect;

    public EchoWorkerThread(TcpConnection con) {

        this.con = con;
        connect = true;
        nrWorkerThread++;
        this.setName("WorkerThread-" + nrWorkerThread);
    }

    public void run() {
        System.out.println(this.getName() + " gestartet");
        while (connect == true) {
            try {
                echo();
            } catch (Exception e1) {
                try {
                  System.out.println(this.getName() + ": Exception beim
                                                              Empfang");
```

```
                    con.close();
                    connect = false;
                  } catch (Exception e2) {
                    connect = false;
                }
            }
        }
    }

    private void echo() throws Exception {
        try {
            SimplePDU receivedPdu = (SimplePDU) con.receive();
            String message = receivedPdu.getMessage();
            con.send(receivedPdu);
          } catch (Exception e) {
            System.out.println("Exception beim Empfang");
            throw new Exception();
        }
    }
}
```

5.7 Weiterführende Programmierkonzepte und -mechanismen

Mit den bisher gezeigten Ansätzen lassen sich kleinere Anwendungen mit wenigen und vielleicht sogar – je nach Serversystem und Serverlast – bis zu mehreren Hundert nebenläufigen Clients akzeptabel bedienen. Serversysteme, die sehr viele Clients bedienen müssen, wie dies etwa bei Webservern oder komplexen Anwendungsservern der Fall ist, können nicht jedem Client einen Thread für die ganze Verbindung zur Verfügung stellen. Bei Hundertausenden von Clients würden die Ressourcen von Serverbetriebssystemen knapp und die Bearbeitungszeiten zu langsam. Optimierungen sind also dahingehend erforderlich, dass man mit weniger Ressourcen auskommen muss. In der Regel arbeiten ja nicht alle Clients gleichzeitig und daher reicht es oft, mit m Threads n Clients zu bedienen, auch wenn $m << n$ gilt.

Für die Entwicklung von Hochleistungsservern geht man nicht mehr den Weg, dass für jede TCP-Verbindung serverseitig auch ein Thread bereitgestellt wird, der blockierend auf ankommende Anfragen seines Clients wartet. Dies wird auch als Blocking-I/O bezeichnet, weil der Thread in der Regel in einem Receive-Aufruf blockiert, bis die nächste Nachricht ankommt (Abb. 5.16).

Für die Entwicklung von leistungsfähigen Serversystemen nutzt man Threadpools und die sogenannten Workerthreads aus dem Pool bedienen jeweils einzelne Anfragen. Wenn die Arbeit erledigt ist, steht der Workerthread für die nächste Anfrage zur Verfügung. Für diese Art der Requestbearbeitung benötigt man aber einen Mechanismus, der ein zentrales Warten auf Ereignisse (in diesem Fall meist Netzwerkereignisse wie die Ankunft einer

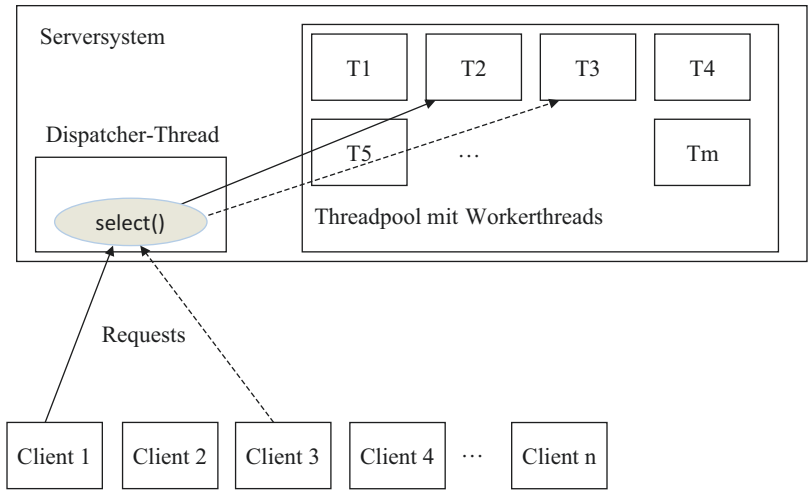


Abb. 5.16 Dispatching über zentralen Ereigniswartepunkt

Anfrage) ermöglichen. Hierfür bedient man sich der speziellen Systemfunktionen *select*. Diese ist auch im POSIX-Standard definiert (IEEE POSIX 2016), die in den heutigen Universalbetriebssystemen üblicherweise implementiert ist.

Über *select* ist es möglich, Ereignisquellen wie bestehende TCP-Verbindungen zentral zu überwachen. Wenn eine Nachricht ankommt, wird dies von einem dedizierten Dispatcher- oder Multiplexer-Thread an einem mit der *select*-Funktion implementierten zentralen Ereigniswartepunkt erkannt. Das Ereignis (ankommende Nachricht, Verbindungsabbau) kann analysiert und die Bearbeitung einem Workerthread übergeben werden. Anschließend kann ein Dispatcher-Thread sofort wieder auf weitere Ereignisse horchen.

Auch höherwertige Frameworks implementieren bereits standardmäßig derartige Mechanismen. Viele individuelle Anwendungen und Serverdienste wie Webserver oder Filetransfer-Server nutzen Eigenimplementierungen auf der Basis dieser systemnahen Mechanismen. Für die Java-Entwicklung stehen spezielle Java-Packages wie *java.nio* oder erweiterte Frameworks wie *netty* (Mauer und Wolfthal 2016) zur Verfügung.

Höherwertige Kommunikationsmechanismen speziell für die Client-/Serverkommunikation kapseln die gesamte Socket-Schnittstelle und bieten komfortable Mechanismen für die Requestbearbeitung. Das Verbindungsmanagement und die Fehlerbehandlung werden vom Framework übernommen. Der Anwendungsprogrammierer muss sich nicht mehr mit den Details der Kommunikation befassen. Aus seiner Sicht spielt es fast keine Rolle, ob der programmierte Request lokal oder entfernt auf einem anderen Server ausgeführt wird. Im Innern dieser Frameworks werden aber Sockets und Nonblocking I/O-Mechanismen sowie Threadpools genutzt. Beispiele für Frameworks im Java-Umfeld sind Java Remote Method Invocation (RMI) oder Java Enterprise Java Beans (EJB). Diese Frameworks bezeichnet man auch als Kommunikations-Middleware (Mandl 2009).

Literatur

Hafner, K; Lyon, M. (2000) ARPA KADABRA oder die Geschichte des Internet, dpunkt.verlag, 2000
IEEE POSIX (2016) The Open Group Base Specifications Issue 7, IEEE Std 1003.1™-2008, 2016 Edition, http://pubs.opengroup.org/onlinepubs/9699919799/, letzter Zugriff am 10.09.2017
Mandl, P. (2014) Grundkurs Betriebssysteme, 4. Auflage, Springer-Vieweg Verlag, 2014
Mandl, P. (2009) Masterkurs Verteilte betriebliche Informationssysteme, Prinzipien, Architekturen und Technologien, Springer-Vieweg Verlag, 2009
Mauer, N.; Wolfthal M. A. (2016) Netty in Action, Manning Publications, 2016
Stevens, R. W.; Fenner, B.; Rudoff A.M (2005) UNIX Network Programming. The Sockets Networking API. Volume 1. 3. Auflage. Addison Wesley, 2004
Stevens, R. W. (2000) Programmieren von UNIX-Netzen, Hanser Verlag, 2000
Tanenbaum, A. S.; Wetherall, D. J. (2011) Computer Networks, Fifth Edition, Pearson Education, 2011

Schlussbemerkung

6

Zusammenfassung

Dieses Buch sollte einen tieferen Einblick in die Funktionsweise und Nutzung die wichtigsten Transportprotokolle in heutigen Kommunikationssystemen und auch in die Programmierung von Kommunikationsanwendungen geben. Die Entwicklung von Transportprotokollen ist aufgrund der rasanten Entwicklung des Internets noch lange nicht abgeschlossen. Es werden immer wieder neue Protokolle bzw. Optimierungen bestehender Protokolle vorgeschlagen.

TCP und UDP sind zwei Protokolle, die in vielen verteilten Anwendungen für den Transport von Nachrichten verwendet werden und bilden die Grundlage vieler verteilter Anwendungen. Nach einer Einführung in die Grundbegriffe der Datenkommunikation und in heute anerkannte Referenzmodelle wurde der Fokus vor allem auf die Funktionalität der Transportschicht gelegt. Am Beispiel der Transportprotokolle TCP und UDP wurden Protokollmechanismen zur Datenübertragung detailliert diskutiert. Anschließend wurde die Nutzung der Transportzugriffsschnittstelle konkret mit Hilfe der Socket API eingeführt, um zu zeigen, wie man verteilte Anwendungen entwickeln kann, die den Nachrichtenaustausch über TCP und UDP organisieren.

Auf die anderen Kommunikationsschichten wie etwa auf die Vermittlungsschicht und die Netzzugangsschicht wurde nur insoweit eingegangen, wie es für das Verständnis der Funktionen des Transportsystems von Bedeutung ist. Vor allen die Vermittlungsschicht mit ihren vielfältigen Funktionen soll an anderer Stelle näher betrachtet werden (siehe z. B. Mandl et al. 2010).

P. Mandl, *TCP und UDP Internals*, https://doi.org/10.1007/978-3-658-20149-4_6

Transportprotokolle entwickeln sich ständig weiter. Ständig werden neue Implementierungsvorschläge, in letzter Zeit insbesondere für die Verbesserung der Staukontrolle, gemacht. Die Internet-Community diskutiert und forscht hier rege weiter, wie zahlreiche Ansätze zeigen (siehe z. B. Yanyan und Keyu 2012). Auch sind neue Protokollansätze wie etwa das Transportprotokoll *QUIC*[1] von Google weiter in Diskussion. Allerdings können sich Erweiterungen und Verbesserungen aufgrund der sehr starken Verbreitung von TCP nur langsam durchsetzen. Die Entwicklung bleibt weiterhin sehr interessant.

Literatur

Mandl, P.; Bakomenko A.; Weiß, J. (2010) Grundkurs Datenkommunikation: TCP/IP-basierte Kommunikation: Grundlagen, Konzepte und Standards, 2. Auflage, Vieweg-Teubner Verlag, 2010

Yanyan, L.; Keyu, J. (2012) Prospect for the Future Internet: A Study Based on TCP/IP Vulnerabilities, International Conference on Computing, Measurement, Control and Sensor Network, 2012

[1] Siehe z. B. in https://docs.google.com/document/d/1RNHkx_VvKWyWg6Lr8SZ-saqsQx7rFV-ev-2jRFUoVD34/edit#. Zugegriffen am 07.09.2017.

Übungsaufgaben und Lösungen

7

Zusammenfassung

Zur Vertiefung des Stoffes sollen die wichtigsten Fragestellungen aus den einzelnen Kapiteln nochmals in Form von Übungsaufgaben wiederholt werden. Mögliche Lösungen werden gleich mitgeliefert. Die Übungsfragen werden nach Kapiteln geordnet und sollen auch zum erneuten Lesen noch nicht ganz verstandener Aspekte aus den vorangegangenen Kapitel anregen.

7.1 Grundbegriffe der Datenkommunikation

1. *Was ist Datenkommunikation?*
   Datenkommunikation befasst sich mit dem Transport von Daten über beliebige Übertragungskanäle. Üblicherweise erfolgt der Transport der Daten in Nachrichten.
2. *Welchen Sinn hat die Schichteneinteilung in Referenzmodellen der Datenkommunikation?*
   Sinn der Schichtung ist die Kapselung von Funktionalität derart, dass eine Schicht nur die Funktionen der direkt darunterliegenden Schicht über eine dedizierte Schnittstelle kennen muss. Damit wird auch die komplexe Materie greifbarer.
3. *Was versteht man unter einem Dienst und was unter einem Protokoll?*
   Ein Dienst ist eine Sammlung von Funktionen, die eine Schicht an einem SAP bereitstellt. Ein Dienst hat gegebenenfalls mehrere Dienstelemente (Primitiven). Ein Kommunikationsprotokoll oder kurz ein Protokoll ist ein Regelwerk zur Kommunikation zweier Rechnersysteme untereinander. Protokolle folgen in der Regel einer exakten Spezifikation.

P. Mandl, *TCP und UDP Internals*, https://doi.org/10.1007/978-3-658-20149-4_7

4. *Erläutern Sie die Schichten des TCP-Referenzmodells!*
   Das TCP-Referenzmodell hat vier Schichten. Die oberste Schicht entspricht der Anwendungsschicht, darunter liegen die Transportschicht und die Netzwerkschicht. Die unterste Schicht ist die Netzwerkzugriffsschicht.
5. *Was versteht man unter einer Protokollinstanz?*
   Unter einer Protokollinstanz versteht man in der Datenkommunikation die Implementierung einer konkreten Schicht. Protokollinstanzen gleicher Schichten kommunizieren untereinander über ein gemeinsames Protokoll.
6. *Was ist ein Protokollstack?*
   Eine konkrete Protokollkombination wird auch als Protokollstack (kurz Stack) bezeichnet.
7. *Was ist ein Transportsystem?*
   Die Protokollschichten 1 bis 4 werden auch gemeinsam als Transportsystem bezeichnet. Das Transportsystem stellt den darüberliegenden Anwendungen einen Transportdienst zur Verfügung. Dieser ermöglicht es Anwendungsprozessen, Nachrichten untereinander über ein Netzwerk in einer Ende-zu-Ende-Kommunikation auszutauschen.
8. *Was ist eine Transportzugriffsschnittstelle?*
   Die Transportschicht (Schicht 4) sorgt für eine Ende-zu-Ende-Beziehung zwischen zwei Kommunikationsprozessen und stellt einen Transportdienst für die höheren Anwendungsschichten bereit.
9. *Wie setzt sich eine Transportadresse bei TCP zusammen?*
   Aus einer Portnummer und einer IP-Adresse. Die Portnummer ist rechnerweit eindeutig und die IP-Adresse ist einer Netzwerkschnittstelle des Rechners zugeordnet. Eine Portnummer darf zu einer Zeit nur einmal vergeben werden.
10. *Was ist eine Protocol Data Unit (PDU)?*
    Die Instanzen der gleichen Protokollschichten tauschen Protocol Data Units (PDU) miteinander aus, die sowohl Steuerinformationen der jeweiligen Schicht als auch die Nutzdaten der nächsthöheren Schicht enthalten.
11. *Was ist ein Service Access Point (SAP)?*
    Ein SAP ist eine Schnittstelle zur Kommunikation einer Schicht mit einer darunterliegenden Schicht.
12. *Erläutern Sie die Begriffe Nachricht, Segment, Paket und Frame?*
    Eine PDU der Anwendungsschicht wird als Nachricht bezeichnet, Segment ist eine Bezeichnung für eine Schicht-4-PDU, *Paket* bezeichnet eine PDU der Schicht 3 und Frame eine PDU der Schicht 2.

7.2 Grundkonzepte der Transportschicht

1. *Nennen Sie vier typische Protokollfunktionen, die in der Transportschicht implementiert sein sollten!*
   - Verbindungsmanagement und Adressierung
   - zuverlässiger Datentransfer

- Flusskontrolle
- Staukontrolle
- Multiplexierung und Demultiplexierung
- Fragmentierung (bzw. Segmentierung) und Defragmentierung

2. *Beschreiben Sie einen Dreiwege-Verbindungsaufbau und erläutern Sie kurz, wie man durch diese Art des Verbindungsaufbaus Duplikate erkennen kann!*
   Host A initiiert einen Verbindungsaufbau mit einer Connect-Request-PDU und sendet dabei eine vorher ermittelte Folgenummer mit. Mit der Connect-Request-PDU wird auch die Transportadresse gesendet, um den Empfänger zu identifizieren.

   Host B bestätigt den Verbindungsaufbauwunsch mit einer ACK-PDU, bestätigt dabei auch die Folgenummer und sendet seine eigene Folgenummer mit der ACK-PDU an Host A.

   Host A bestätigt die Folgenummer von Host B eventuell direkt oder mit der ersten Data-PDU im Pickypacking-Verfahren und damit ist die Verbindung aufgebaut.

   Duplikate können aufgrund der vereinbarten und jeweils bestätigten Folgenummern (fortlaufende Zähler der abgesendeten Nachrichten) in Kombination mit einer maximalen Paketlebensdauer erkannt werden.
3. *Was ist eine Ende-zu-Ende-Verbindung im Sinne der Transportschicht?*
   Hierunter versteht man eine Verbindung zwischen zwei kommunizierenden Prozessen auf unterschiedlichen Rechnersystemen oder auch auf demselben Rechnersystem.
4. *Nennen Sie je einen Vorteil für ein verbindungsloses und ein verbindungsorientiertes Protokoll der Transportschicht!*
   Vorteile eines verbindungslosen Protokolls:
   - schnellere Kommunikation durch geringen Overhead
   - einfachere Implementierung

   Vorteile eines verbindungsorientierten Protokolls:
   - sichere Zustellung der Daten (kein Verlust von einzelnen Paketen)
   - fehlerfreie Auslieferung der Daten
   - reihenfolgerichtige Auslieferung der Daten
5. *Was will man mit einer Timerüberwachung beim Verbindungsabbau einer Transportverbindung erreichen?*
   Beim Verbindungsabbau sollen keine Nachrichten verloren gehen. Durch Störungen im Netz kann jedoch jederzeit eine Disconnect-Request-PDU oder eine Bestätigung verloren gehen. Über eine Timerüberwachung mit begrenzter Anzahl von Nachrichtenwiederholungen wird versucht, dieses Problem zu lösen. Falls alle Disconnect-Requests verloren gehen, baut der Sender die Verbindung ab. Der Partner weiß davon nichts (halb offene Verbindung), baut jedoch auch nach Ablauf eines Timers die Verbindung automatisch ab.
6. *Bei der Fehlerbehandlung nutzt man in einer gesicherten Transportschicht u. a. das positiv selektive und das negativ selektive Quittierungsverfahren. Erläutern Sie die beiden Verfahren!*
   Positiv-selektives Verfahren: Es wird jede einzelne erfolgreich übermittelte Nachricht quittiert (hoher zusätzlicher Nachrichtenverkehr).

   Negativ-selektives Verfahren: Es werden nur selektiv nicht empfangene, also verloren gegangene Nachrichten erneut angefordert.

7. *Zur Fehlerbehebung nutzt man in der Transportschicht das Verfahren der Übertragungswiederholung. Nennen Sie hierzu zwei Verfahren und erläutern Sie diese kurz! Was muss der Sender tun, damit eine Übertragungswiederholung möglich ist?*
   Selektive Wiederholung: Nur negativ quittierte Nachrichten werden wiederholt. Der Empfänger puffert Nachrichten, bis die fehlende da ist.

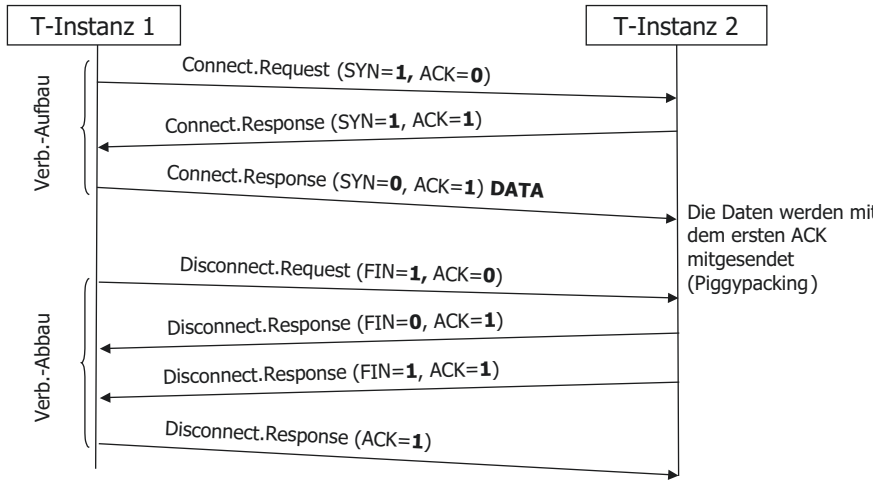
   Go-Back-N-Verfahren: Es werden die fehlerhafte, sowie alle darauf folgenden Nachrichten erneut übertragen. Der Empfänger benötigt eine geringere Speicherkapazität.
8. *Wie lange muss der Sender in der Transportinstanz die versendeten Nachrichten im Puffer vorhalten?*
   Bei allen Verfahren muss der Sender die Nachrichten über einen gewissen Zeitraum bereithalten. Er kann sie nur beim Empfang einer ACK-PDU verwerfen. Beim Stop-and-Wait-Verfahren ist dies einfach. Der Sender muss nur eine Nachricht speichern und kann sie bei Empfang der ACK-PDU verwerfen.

   Schwieriger ist es beim positiv-kumulativen und beim selektiven Verfahren. Beim positiv-kumulativen Verfahren muss er alle Nachrichten bis zum Empfang der kumulativen Bestätigung vorhalten. Das können bei long fat networks ziemlich viele Nachrichten sein. Beim selektiven Verfahren ist es für den Sender problematisch festzustellen, wann eine gesendete Nachricht aus dem Puffer eliminiert werden kann. Er muss Nachrichten puffern, bis er sicher ist, dass sie übertragen wurden.

7.3 TCP-Konzepte und -Protokollmechanismen

1. *Was ist bei TCP ein Port und was ist bei TCP ein Well-known Port?*
   Anwendungsprozesse sind über Sockets eindeutig adressierbar. Einem Socket werden eine IP-Adresse und eine Portnummer zugewiesen. Ein Serverprozess stellt einen Dienst bereit und bietet ihn über eine wohlbekannte Portnummer, einen sogenannten Well-known Port an. Wenn ein Clientprozess die IP-Adresse und die Portnummer des Dienstes sowie das zugehörige Protokoll kennt, kann er mit dem Server kommunizieren und dessen Dienste nutzen. Well-known Ports sind innerhalb der TCP/IP-Gemeinde bekannt und werden immer gleich benutzt. Wichtige Well-known Ports und dazugehörige Dienste sind z. B. die Ports 23 (Telnet), 20 und 21 (File Transfer Protocol) 25 (Simple Mail Transfer Protocol) und 80 (Hyper Text Transfer Protocol).
2. *TCP ist datenstromorientiert (streamortientiert). Was bedeutet das?*
   Ein Anwendungsprozess schreibt seine Daten Byte für Byte in einen Bytestrom. Im Gegensatz zur blockorientierten Übertragung kümmert sich hier die TCP-Instanz um die Segmentierung der Daten.
3. *Was sind TCP-Segmente, wie groß sind diese mindestens und warum?*
   TCP sieht den Datenstrom (Stream) als eine Sequenz von Octets (Byte) und unterteilt diese zur Übertragung in Segmente. Ein Segment besteht aus dem mindestens 20 Byte langen TCP-Header.

4. *Wie wird in TCP eine Verbindung eindeutig identifiziert?*
   Eine Verbindung wird durch ein Paar von Endpunkten eindeutig identifiziert (Socket Pair). Dies entspricht einem Quadrupel, das die IP-Adresse von Host 1, die Portnummer, die in Host 1 genutzt wird, die IP-Adresse von Host 2 und die Portnummer auf der Seite von Host 2 enthält.
   Beispiel: `((195.214.80.76, 80) (196.210.80.10,6000))`
5. *Wie viele TCP-Segmente werden für die Übertragung von 100 Byte Nutzdaten durchs Netz gesendet? Erläutern Sie dies anhand eines Time-Sequence-Diagramms!*



   T-Instanz 1 baut mittels Dreiwege-Handshake eine Verbindung zu T-Instanz 2 auf (3 TCP-Segmente) Mit dem letzten ACK des Dreiwege-Handshake wird hier das zu sendende Datenpaket übertragen (Piggy-Packing). Somit kann ein weiteres Daten-TCP-Segment eingespart werden. Anschließend wird die Verbindung gemäß dem Protokoll abgebaut (drei bis vier Segmente). Der Empfang der letzten Nutzdaten wird hier im Rahmen des Verbindungsabbaus bestätigt.

   Es werden also mindestens 7 Segmente benötigt, um 100 Byte Nutzdaten zu übertragen.
6. *Welche Quittierungsvariante wird bei TCP eingesetzt, damit der Empfänger den ordnungsgemäßen Empfang einer Nachricht bestätigen kann? Was macht der Sender eines Pakets, wenn er keine Quittung vom Empfänger erhält?*
   Es wird ein positiv-kumulatives Bestätigungsverfahren mit Timerüberwachung bei jeder Nachricht verwendet. Bei Ablauf des Timers ohne vorige Bestätigung wird die Nachricht erneut gesendet.
7. *Erläutern Sie das Silly-Window-Syndrom bei TCP!*
   Wenn der Anwendungsprozess im Empfänger die empfangenen Daten byteweise ausliest, wird nach jedem gelesenen Byte eine ACK-PDU mit dem Hinweis, dass wieder

ein Byte gesendet werden kann, übertragen. Das Netz wird also durch viele 1-Byte-Nachrichten (Nutzlast) belastet.

8. *Wozu dient das Slow-Start-Verfahren bei TCP? Beschreiben Sie das Verfahren kurz!*
   Das Slow-Start-Verfahren wird im Rahmen der Staukontrolle eingesetzt. Es wird versucht, sich an die maximale Menge an Daten, die über das Netz gesendet werden kann, heranzutasten. Zusätzlich zum Empfangsfenster der Flusskontrolle wird ein Überlastfenster eingeführt. Dabei gilt, dass der Sendekredit für eine TCP-Verbindung aus dem Minimum aus {Überlastfenstergröße, Empfangsfenstergröße} berechnet wird.
   Das Slow-Start-Verfahren läuft in zwei Phasen ab:
   – Slow-Start-Phase: Sender und Empfänger einigen sich beim Verbindungsaufbau auf eine erste zu sendende Anzahl an Segmenten mit der vereinbarten TCP-Segmentlänge. Der Sender sendet zunächst ein Segment dieser Länge. Kommt eine ACK-PDU hierfür rechtzeitig, wird die Anzahl der Segmente, die gesendet werden dürfen, also das Überlastfenster, verdoppelt. Das geht bis zu einem Schwellwert so weiter (exponentielle Steigerung).
   – Nach dem Erreichen des Schwellwerts geht das Verfahren in die sogenannte Probing- oder Überlastvermeidungsphase über. In dieser Phase erhöht sich bei jeder empfangenen Quittung die Anzahl der zulässigen Segmente nur noch linear um 1.
9. *Wozu dienen die beiden Zustände TIME_WAIT und CLOSE_WAIT im TCP-Zustandsautomaten? Gehen Sie dabei kurz auf den Verbindungsabbau ein!*
   Der Client initiiert den Verbindungsabbau mit einem *close*-Aufruf. Er sendet ein FIN-Segment und wechselt in den Zustand FIN_WAIT_1. Kommt eine ACK-PDU vom Server, wird die Verbindung in einer Richtung geschlossen und der Zustand wechselt zu FIN_WAIT_2. Schließt auch die andere Seite die Verbindung, wird eine FIN-PDU vom Client empfangen, das wiederum bestätigt wird.
   Der Client geht in den Zustand TIME_WAIT. Um sicherzustellen, dass beim Verbindungsabbau keine Nachrichten verloren gehen, wird ein Timer aufgezogen, der als Wartezeit die doppelte Paketlebensdauer verwendet. Erst nach Ablauf des Timers wird in den Zustand CLOSED gewechselt und somit die Verbindung beendet.
   Auf Serverseite wird nach Bestätigung des Abbauwunsches des Clients in den Zustand CLOSE_WAIT gegangen. Nachdem alle noch zu sendenden Pakete abgearbeitet wurden, wird auch vom Server der Verbindungsabbau mit dem Senden eines FIN-Segments signalisiert. Der Server befindet sich danach im Zustand LAST_ACK. In diesem Zustand wartet er noch auf die letzte Bestätigung und geht anschließend in den Zustand CLOSED.
10. *Nennen Sie die wichtigsten Timer, die eine TCP-Instanz verwendet und beschreiben Sie kurz deren Aufgabe!*
    – *Retransmission Timer*: Überwachung jeder einzelnen Übertragung der TCP-Segmente und Übertragungswiederholung nach Timerablauf.
    – *Keepalive* Timer: Überprüfung, ob der Partner noch lebt.
    – *Time-Wait Timer*: Sicherstellen, dass alle Pakete bei einem Verbindungsabbau noch übertragen werden.

- *Close-Wait Timer:* Gibt die Zeit an, die eine passive TCP-Instanz maximal wartet, bis ein Anwendungsprozess aktiv einen *close*-Aufruf absetzt, um seine Verbindungsseite abzubauen.
- *Persistence Timer:* Verhindert ein ewiges Warten eines TCP-Senders, wenn der TCP-Empfänger die Fenstergröße auf 0 setzt.

11. *Was möchte man bei der laufenden Ermittlung des RTO erreichen und wie wird das erreicht?*
    Bei jedem Roundtrip wird die neue RTT aus der RTT-Historie und der zuletzt gemessenen RTT ermittelt, wozu ein exponentiell gewichteter gleitender Durchschnitt berechnet wird. Durch diese Berechnung soll erreicht werden, dass sich die Timerlänge des Retransmission Timers (RTO) der Netzwerksituation dynamisch anpasst. Durch die Nutzung eines exponentiell gewichteten, gleitenden Durchschnitts der RTT-Werte der Vergangenheit verliert das Gewicht eines einzelnen RTT-Werts rasch an Bedeutung, womit Schwankungen geglättet werden. Auch die Variabilität der RTT-Werte der Vergangenheit wird bei der Berechnung eines neuen RTO berücksichtigt.
12. *Wie kann man bei TCP die maximal für eine Verbindung mögliche Segmentgröße einstellen?*
    Mit der MSS-Option (Maximum Segment Size Option) können beim Verbindungsaufbau die maximal zulässigen Segmentlängen, also die Maximum Segment Size (MSS), ausgetauscht werden. Wenn diese Option in einem TCP-Header enthalten ist, muss auch gleichzeitig das SYN-Flag gesetzt sein. Sie wird also nur beim Verbindungsaufbau genutzt.
13. *Kann man bei TCP die maximal mögliche Fenstergröße für eine Verbindung einstellen?*
    Die Windows-Scale-Option (WSopt) dient zum Austausch der maximalen Fenstergröße für das Sliding-Window-Verfahren. Dieser Parameter wird auch als TCP Receive Window Size (kurz: RWin) bezeichnet.
14. *Was kann man mit der TCP-Option SACK Permitted Option einstellen?*
    TCP ermöglicht damit, anstelle des Go-Back-N-Verfahrens für eine TCP-Verbindung eine selektive Übertragungswiederholung anzuwenden.
15. *Was kann man mit der TCP-Option SACK einstellen?*
    Mit der Option SACK kann man für eine Verbindung erlauben, dass in einer ACK-PDU (ACK-Flag gesetzt) eine Liste von Sequenznummernbereichen übergeben wird, die bereits empfangen wurden.
16. *Welches Problem kann bei TCP durch die begrenzten Wertebereich für Folgennummern auftreten und welche Maßnahmen gibt es, um die Schwierigkeiten einzudämmen?*
    Es können in einer Verbindung Sequenznummern-Kollisionen auftreten, wenn 2^{32} und mehr Bytes übertragen werden. Ebenso könnte es vorkommen, dass eine neu eingerichtete Verbindung die gleiche Adresse wie eine vorher benutzte ältere Verbindung zugewiesen bekommt. In diesem Fall könnten in der neuen Verbindung verzögert ausgelieferte Segmente der alten Verbindung empfangen werden. Maßnahmen zur Abmilderung sind unter dem Begriff *Protect Against Wrapped Sequences* oder *PAWS* zusammengefasst. Maßnahmen sind u. a. der TIME_WAIT-Timeout beim Verbindungsabbau, die zufällige Auswahl von initialen Folgenummern für eine Verbindung und die Nutzung der *TSOpt*-Option, um Sequenznummern mit Zeitstempeln zu versehen.

17. *Was bedeutet Fast Recovery bei TCP?*
    Der Algorithmus nutzt den mehrfachen Empfang von Bestätigungen für das gleiche TCP-Segment dazu, um einen Paketverlust zu diagnostizieren. Wenn der Sender vier Quittierungen (drei sogenannte ACK-Duplikate) für ein konkretes TCP-Segment empfängt, er aber schon weitere TCP-Segmente danach gesendet hatte, geht er von einem Paketverlust aus und veranlasst die sofortige Sendewiederholung des Segments. Hier spricht man auch von Fast Retransmission. Der Empfänger sendet immer dann ein ACK-Duplikat, wenn ein empfangenes TCP-Segment nicht in die Reihenfolge passt.
18. *Welchen Mechanismus gibt es in der Staukontrolle bei TCP/IP, der die Internetschicht einbezieht und wie funktioniert er grob?*
    Man hat in letzter Zeit auch noch über einen anderen Mechanismus konzipiert, der Informationen aus der Internetschicht nutzt. Dieser Mechanismus wird als Explicit Congestion Notification (ECN) bezeichnet. Der Mechanismus sieht vor, dass der IP-Router Informationen zu möglichen Engpässen an TCP-Verbindungen weitergibt.
19. *Wozu dienen die Status SYN_RECVD und SYN_SENT des TCP-Zustandsautomaten?*
    – SYN_RCVD: Ankunft einer Verbindungsanfrage und Warten auf Bestätigung.
    – SYN_SENT: Die Anwendung hat begonnen, eine Verbindung zu öffnen.
20. *Was passiert, wenn eine TCP-Verbindung von beiden Partnern nahezu gleichzeitig initiiert wird?*
    Wenn beide Seiten zeitgleich mit einem *close*-Aufruf ihre Verbindung beenden, erfolgt der Verbindungsabbau auf beiden Seiten über den Zustand CLOSING. Es werden aber wie beim normalen Verbindungsabbau vier TCP-Segmente ausgetauscht.
21. *Erläutern Sie kurz, was bei einer negativen Quittierung für eine Transport-PDU passiert, wenn in einem Netzwerk mit hoher Pfadkapazität eine fensterbasierte Flusskontrolle mit einem großen Fenster angewendet wird und die Übertragungswiederholung im Go-Back-N-Verfahren erfolgt.*
    Bei entsprechender Fenstergröße können viele PDUs gesendet werden, bevor die erste Negativ-Quittung beim Sender eintrifft. Dies führt dann gegebenenfalls zur unnötigen Übertragungswiederholung vieler PDUs.
22. *Ist TCP ein sicheres Protokoll im Sinne der Informationssicherheit?*
    In der TCP-Spezifikation selbst sind keinerlei Sicherheitsmechanismen im Sinne der Vertraulichkeit, Verfügbarkeit und Integrität vorgesehen. Die Gewährleistung einer sicheren und vertrauenswürdigen Übertragung wird den Protokollen in der Anwendungsschicht oder in der Vermittlungsschicht überlassen.

7.4 UDP-Konzepte und -Protokollmechanismen

1. *Was unternimmt die empfangende UDP-Instanz, wenn eine UDP-PDU ankommt? Gehen Sie hier auf den Sinn des UDP-Pseudo-Headers ein!*
    Nach Empfang einer UDP-PDU wird anhand einer Prüfsumme das Gesamtpaket (Daten+Header) überprüft. Dazu wird ein virtueller Header (Pseudo-Header) aufgebaut, der zusätzlich Informationen aus dem IP-Header enthält.

Der Empfänger muss bei Empfang einer UDP-Nachricht, die eine Prüfsumme enthält, Folgendes unternehmen:
- die IP-Adressen aus dem ankommenden IP-Paket lesen
- der Pseudo-Header muss zusammengebaut werden
- die Prüfsumme muss ebenfalls berechnet werden
- die im Header gesendete Prüfsumme muss mit der berechneten verglichen werden

2. *Nennen Sie zwei Vorteile von UDP im Vergleich zu TCP!*
   Folgende Vorteile hat eine UDP-Nutzung:
   - schneller, da (fast) keine Sicherungsmechanismen vorhanden sind
   - einfacher zu implementieren
3. *Welchen Sinn hat der UDP-Pseudoheader?*
   Der UDP-Pseudoheader ist ein virtueller Header, der u. a. die IP-Adresse des Senders und Empfängers beinhaltet. Über ihn wird eine Prüfsumme berechnet, die mit der übertragenen Prüfsumme im UDP-Header verglichen wird. Stimmen die Prüfsummen überein, ist sichergestellt, dass das Datagramm den richtigen Empfänger (IP und Port) erreicht hat.
4. *Welche Fehler können mit den UDP-Prüfsummenverfahren gefunden werden?*
   Einbit-Fehler werden mit dem Verfahren erkannt. Im Vergleich zu einfachen Paritätsprüfverfahren ist das UDP-Verfahren aber relativ schwach. Zweifache und dreifache Bitfehler werden nicht erkannt.
5. *Übernimmt eine UDP-Instanz die Aufgabe der Segmentierung eines langen Datagramms oder muss diese Aufgabe das Anwendungsprotokoll erledigen?*
   Eine UDP-Instanz segmentiert die Pakete. Dabei ist es sinnvoll, UDP-Segmente nicht länger als in der maximal möglichen IP-Paketlänge zu versenden, da sonst in der IP-Schicht fragmentiert wird.
6. *Was ist eine Einerkomplement-Addition im Sinne des UDP-Prüfverfahrens?*
   Das Prüfverfahrens sieht vor, dass die Prüfsumme über das ganze UDP-Segment und zusätzlich über einen vorangestellten Pseudoheader ermittelt wird. Dabei werden alle 16-Bit-Wörter des Segments beim Sender nacheinander unter Berücksichtigung möglicher Überlaufbits (Carry-out Bits) addiert. Diese Addition wird gelegentlich auch als Einerkomplement-Addition bezeichnet. Ganz korrekt ist die Bezeichnung bei diesem Verfahren nicht, da beim Einerkomplement der Wertebereich nicht überschritten werden darf, beim UDP-Prüfsummenverfahren aber schon. Die Summe wird dann in das Einerkomplement umgewandelt. Falls das Ergebnis nur aus binären Nullen besteht, wird es zu 0xFFFF konvertiert.
7. *Was bedeutet bei UDP Multiplexing?*
   Sendet ein Anwendungsprozess über einen UDP-Port eine Nachricht, wird es von der lokalen UDP-Instanz übernommen, in ein UDP-Segment eingebettet und an die darunterliegende IP-Instanz weitergegeben. Auch Kommunikationsendpunkte anderer Anwendungen werden über dieselbe IP-Instanz weiterverarbeitet. Die UDP-Instanz führt also ein Multiplexing mehrerer UDP-Kommunikationsendpunkte über einen IP-Endpunkt durch.

8. *Ist UDP ein sicheres Protokoll im Sinne der Informationssicherheit und ist es sicherer als TCP?*
   Wie in der TCP-Spezifikation sind auch in der UDP-Spezifikation keinerlei Sicherheitsmechanismen im Sinne der Informationssicherheit (Vertraulichkeit, Verfügbarkeit und Integrität) vorgesehen. UDP-Kommunikation ist daher als nicht vertrauenswürdig einzustufen.

   UDP ist sogar noch leichter anzugreifen als TCP, weil es zudem nicht über einen Bestätigungsmechanismus verfügt und daher ein Angreifer auch keine Sequenzunummern herausfinden muss. Ebenso fehlt ein Handshake für den Verbindungsaufbau, der das Einmischen eines Angreifer erschweren könnte.

7.5 Programmierung von TCP/UDP-Anwendungen

1. *Wie ist die Socket API in heutigen Rechnern eingebettet?*
   Die Socket API ist üblicherweise an der Schnittstelle zum Betriebssystem angesiedelt. Die Socket-Dienste werden also vom Betriebssystem bereitgestellt. Der Anwendungsprogrammierer nutzt eine Bibliothek oder wie in Java ein vorhandenes Java-Package, um Sockets zu nutzen.
2. *Was versteht man unter einem Objektstrom in Java und wozu wird der Mechanismus bei der TCP-Socket-Programmierung verwendet?*
   Die TCP-Socket-Schnittstelle ist streamorientiert. Es wird also auf dieser Ebene zwischen Sender und Empfänger über einen Datenstrom kommuniziert. Java bietet einen vordefinierten Objektstrom an, der die Serialisierung und Deserialisierung der Datenobjekte übernimmt und über den eigentlichen Input-/Output-Stream gelegt werden kann. Aus Sicht der Anwendung werden dann beliebig komplexe Objekte versendet bzw. empfangen und man braucht sich nicht mehr um die Details der Serialisierung kümmern.
3. *Kann man Java-Objektströme auch für Datagramm-Sockets verwenden? Erläutern Sie Ihre Entscheidung!*
   Java-Objektströme sind nicht geeignet für eine Kommunikation über UDP. Da UDP nachrichtenorientiert ist, müssen die zu übertragenden Daten als Byte-Array-Pakete einzeln übertragen werden. Allerdings kann man mit Java-Mitteln über die Byte-Arrays auch Java-Streams und Java-Objektströme legen und damit in die zu versendenden Nachrichten ganze Objekte einbetten. Derartige Mechanismen muss man selbst programmieren. Sie sind nicht Bestandteil von Datagramm-Sockets.
4. *Warum erzeugt ein TCP-basierter Server für einzelne Requests von Clients oder sogar für alle Requests eines Clients in der Regel einen neuen Thread für die neue Verbindung?*
   Die aufgebaute Verbindung wird in einen eigenen neuen Thread überführt, der die weitere Bearbeitung im Sinne der Datenübertragung übernimmt. Der Main-Thread der Anwendung ist dadurch entlastet und kann weiterhin auf neue Verbindungsanfragen warten. Diese Art von Server nennt man auch parallelen Server, weil er nebenläufig mehrere/viele Clients bedienen kann.

5. *Was muss in einem Client- und was in einem Serverprogramm programmiert werden, damit diese für die Kommunikation über UDP-Datagramm-Sockets vorbereitet sind?*
   Da UDP verbindungslos ist, findet auch kein Verbindungsaufbau bzw. -abbau statt. Daher muss sowohl auf Server- als auch auf Clientseite nur ein UDP-Socket erzeugt werden, bevor mit Sende- und Empfangsmethoden Daten ausgetauscht werden können. Zum Versand der Daten müssen die IP- und die Portadresse des Partners bekannt sein.
6. *Was macht die Socket-Programmierung mit TCP-Sockets in Java so einfach im Vergleich zur Nutzung der Standard-Socket-Bibliothek der Sprache C?*
   Java stellt dem Netzwerkentwickler das Package *java.net* zur Verfügung, welches Objektklassen beinhaltet, mit denen die Kommunikation einfach implementiert werden kann.
7. *Was hat ein Java-Objektstrom mit dem TCP-Stream zu tun? Erläutern Sie Ihre Antwort kurz!*
   Java-Objektströme serialisieren Java-Objekte und können über einen dem Socket zugeordneten Java-Input-/Output-Stream gelegt werden, sodass ganze Objekte über den Output-Stream versendet und über den Input-Stream empfangen werden können.
8. *Warum müssen Objekte, die in Java-Programmen mit TCP-Sockets gesendet werden, das Interface Serializable implementieren und wie macht man das in Java?*
   Die Implementierung des Interfaces wird in der Klassendeklaration wie folgt angegeben:

   ```
   public class DataObject implements Serializable
   ```

   Dadurch können Java-Objekte über einen Objektstrom serialisiert und versendet werden. Die JVM stellt dabei sicher, dass das Datenobjekte korrekt zerlegt (serialisiert) und wieder zusammengebaut (deserialisiert) wird.
9. *Warum bereitet der blockierende Empfang bei TCP-Servern mit sehr hoher Anzahl an Clients Probleme und wie können die Probleme gelöst werden?*
   Zu viele Ressourcen für Threads wären notwendig, um für jeden Client einen Serverthread zu reservieren. Um sehr viele Clients zu bedienen, nutzt man eine spezielle Betriebssystemfunktion (select), über die man einen gemeinsamen Ereigniswartepunkt für alle Verbindungen implementieren kann. An diesem Ereigniswartepunkt prüft ein Dispatcher, ob Nachrichten ankommen oder sonstige Ereignisse auftreten (Verbindungsauf- oder abbauwunsch, Verbindungsabbruch, ...) und leitet die Ereignisse zur Bearbeitung an einen Thread aus einem Threadpool weiter. Dadurch werden viel weniger Threads als Verbindungen benötigt. APIs wie *NIO* oder Frameworks wie *Netty* unterstützen diese Art der Programmierung.
10. *Wie bindet man eine Adresse an ein Java-Socket?*
    Dies kann man direkt bei der Konstruktion eines Socket-Objekts machen oder nachher über die *bind*-Methode.
11. *Was macht die accept-Methode der Klasse ServerSocket?*
    Die *accept*-Methode wartet auf der Serverseite (passive Seite) auf einen ankommenden Verbindungsaufbauwunsch (*connect*-Aufruf auf der aktiven Seite) und erzeugt

bei Ankunft ein neues Socket für die Verbindung. Implizit wird durch die beteiligten TCP-Instanzen ein Dreiwege-Handshake durchgeführt.

12. *Wie lassen sich in Java Socket-Optionen einstellen?*
    Die Einstellung einiger Socket-Optionen kann über set-Methoden der Socket- oder ServerSocket-Klasse erfolgen.
13. *Nennen Sie fünf Methoden der Java-Klasse Socket, mit denen man TCP-Optionen einstellen kann!*
    Die Methoden können in der Beschreibung der Klasse Socket (siehe Java-API) nachgelesen werden:
    - setKeepAlive()
    - setReuseAddress()
    - setSoLinger()
    - setSoTimeout()
    - setTcpNoDelay()
14. *Was passiert in der TCP-Instanz beim Verbindungsaufbau im Zusammenspiel Socket API mit TCP-Instanz?*
    Der Client setzt durch Nutzung eines entsprechenden Socket-Konstruktors oder durch Aufruf der *connect*-Methode einen Verbindungsaufbauwunsch zunächst an die lokale TCP-Instanz ab. Diese leitet einen Dreiwege-Handshake ein. Auf der Serverseite muss an einem ServerSocket-Objekt ein *accept*-Aufruf abgesetzt worden sein, so dass die entfernte TCP-Instanz bereits auf einen Verbindungsaufbauwunsch wartet und der Dreiwege-Handshake durchgeführt werden kann. Nach dem erfolgreichen Dreiwege-Handshake hat jede TCP-Instanz einen Verbindungskontext aufgebaut. Die Partner sind synchronisiert. Die *connect*- und auch die *accept*-Methode liefern den aufrufenden Anwendungen ein positives Ergebnis (keine Exception) und die Anwendungsprozesse können ihre Arbeit fortsetzen; *accept* und *connect* blockieren also so lange bis die Verbindung steht oder ein Fehler aufgetreten ist, der keinen Verbindungsaufbau zulässt.
15. *Was passiert in der TCP-Instanz, wenn über den Output-Stream eines Sockets die out- oder write-Methode aufgerufen wird?*
    Die in der *out*- oder *write*-Methode angegebenen Daten werden zunächst in der JVM serialisiert. Dann werden die serialisierten Daten von der TCP-Instanz entgegengenommen und in den Sendepuffer der Verbindung gelegt. Aus den daten im Sendepuffer werden TCP-Segmente aufgebaut. Wann die TCP-Segmente genau gesendet wird, entscheidet die TCP-Instanz entsprechend den Protokollregeln. Dies kann auch durch die Betriebssystemkonfiguration oder durch Optionen beeinflusst werden (z. B. Nagle-Algorithmus).
16. *Wann übergibt die TCP-Instanz die Daten eines ankommenden TCP-Segments an den Empfängerprozess?*
    Die Daten ankommender TCP-Segmente werden zunächst durch die TCP-Inszanz im Empfangspuffer der TCP-Verbindung gespeichert. Möglicherweise werden sie gleich mit

einer ACK-PDU bestätigt, um auch die Fenstergröße anzupassen. Die Bestätigung sagt also nichts darüber aus, ob der adressiere Anwendungsprozess bereits die Daten hat. Sie liegen zum Abholen bereit, können aber im Fehlerfall nie zum Anwendungsprozess kommen, z. B. bei Rechnerabsturz. Erst wenn der Anwendungsprozess die *receive*-Methode oder in Java entsprechend die *read*- oder *readObject*-Methode aufruft, werden die Daten aus dem Empfangspuffer ganz oder teilweise an den Anwendungsprozess übergeben.

A Anhang: TCP/IP-Konfiguration in Betriebssystemen

TCP/IP-Stacks sind fester Bestandteil im Betriebssystemkernel heutiger Universalbetriebssysteme. Die Implementierungen sind zum Teil sehr unterschiedlich und verfügen oft über Konfigurierungsmöglichkeiten. Die Parameter sind aber alle mit Sorgfalt zu nutzen. Exemplarisch betrachten wir als Fallstudien die Betriebssysteme Linux und Microsoft Windows.

A.1 Linux

Unter Linux gibt es eine Fülle von Kernelparametern, die sowohl dynamisch für die Laufzeit bis zum nächsten Neustart des Systems und dauerhaft eingestellt werden können.

Alle Kernelparameter sind unter Linux im Verzeichnis */proc/sys/net* zu finden. Kernelparameter für die TCP/IP-Implementierung auf der Basis der IPv4-Spezifikation liegen im Unterverzeichnis */proc/sys/net/ipv4*.

Die aktuelle Einstellung der Parameter kann mit entsprechenden Zugriffsrechten über das Verzeichnis */proc* mit dem Linux-Kommando *cat* angesehen werden.

Beispiel zur Einstellung des Keepalive Timers:

$$cat\ /\ proc\ /\ sys\ /\ net\ /\ ipv4\ /\ tcp\_keepalive\_time$$
$$> 7200\,(Angabe\,in\,Sekunden)$$

Eine temporäre Veränderung für die Systemlaufzeit ist zum Beispiel über Kommando *echo* möglich:

$$echo\ 600 > /\ proc\ /\ sys\ /\ net\ /\ ipv4\ /\ tcp\_keepalive\_time$$

Alternativ kann das Kommando *sysctrl* verwendet werden, das ebenfalls eine Veränderung bis zum nächsten Neustart ermöglicht:

$$sysctl\ -w\ net.ipv4.tcp\_keepalive\_time = 600$$

P. Mandl, *TCP und UDP Internals*, https://doi.org/10.1007/978-3-658-20149-4

Eine permanente Einstellung kann über Einträge in der Datei */etc/sysctl.conf* vorgenommen werden. Der Eintrag für den Keepalive Timer sieht dann wie folgt aus:

$$net.ipv4.tcp\_keepalive\_time = 600$$

TCP-Parameter in Linux anzeigen lassen

Die aktuell eingestellten TCP-Parameter kann man sich beispielsweise mit folgendem Kommando anzeigen lassen:

$$sysctl - a / grep\, ipv4.tcp$$

Terminalausgabe für eine Standardeinstellung der Linux-Distribution CentOS 7:

```
net.ipv4.tcp_abort_on_overflow = 0
net.ipv4.tcp_adv_win_scale = 1
net.ipv4.tcp_allowed_congestion_control = cubic reno
net.ipv4.tcp_app_win = 31
net.ipv4.tcp_autocorking = 1
net.ipv4.tcp_available_congestion_control = cubic reno lp
net.ipv4.tcp_base_mss = 512
net.ipv4.tcp_challenge_ack_limit = 100
net.ipv4.tcp_congestion_control = cubic
net.ipv4.tcp_dsack = 1
net.ipv4.tcp_early_retrans = 3
net.ipv4.tcp_ecn = 2
net.ipv4.tcp_fack = 1
net.ipv4.tcp_fastopen = 0
net.ipv4.tcp_fin_timeout = 60
net.ipv4.tcp_frto = 2
net.ipv4.tcp_invalid_ratelimit = 500
net.ipv4.tcp_keepalive_intvl = 75
net.ipv4.tcp_keepalive_probes = 9
net.ipv4.tcp_keepalive_time = 7200
net.ipv4.tcp_limit_output_bytes = 262144
net.ipv4.tcp_low_latency = 0
net.ipv4.tcp_max_orphans = 4096
net.ipv4.tcp_max_ssthresh = 0
net.ipv4.tcp_max_syn_backlog = 128
net.ipv4.tcp_max_tw_buckets = 4096
net.ipv4.tcp_mem = 21846 29129 43692
net.ipv4.tcp_min_tso_segs = 2
net.ipv4.tcp_moderate_rcvbuf = 1
net.ipv4.tcp_mtu_probing = 0
net.ipv4.tcp_no_metrics_save = 0
net.ipv4.tcp_notsent_lowat = -1
net.ipv4.tcp_orphan_retries = 0
```

```
net.ipv4.tcp_reordering = 3
net.ipv4.tcp_retrans_collapse = 1
net.ipv4.tcp_retries1 = 3
net.ipv4.tcp_retries2 = 15
net.ipv4.tcp_rfc1337 = 0
net.ipv4.tcp_rmem = 4096 87380 6291456
net.ipv4.tcp_sack = 1
net.ipv4.tcp_slow_start_after_idle = 1
net.ipv4.tcp_stdurg = 0
net.ipv4.tcp_syn_retries = 6
net.ipv4.tcp_synack_retries = 5
net.ipv4.tcp_syncookies = 1
net.ipv4.tcp_thin_dupack = 0
net.ipv4.tcp_thin_linear_timeouts = 0
net.ipv4.tcp_timestamps = 1
net.ipv4.tcp_tso_win_divisor = 3
net.ipv4.tcp_tw_recycle = 0
net.ipv4.tcp_tw_reuse = 0
net.ipv4.tcp_window_scaling = 1
net.ipv4.tcp_wmem = 4096 16384 4194304
net.ipv4.tcp_workaround_signed_windows = 0
```

In der Tab. A.1 sind einige TCP- und UDP-Kernelparameter kurz erläutert. Die Aufzählung ist nicht vollständig und soll nur einen Eindruck über die Möglichkeiten der Konfigurierung geben. Alle Parameter sind mit Vorsicht zu genießen, da die Auswirkungen zum Teil nur sehr schwer nachzuvollziehen sind. Wir beschränken uns auch auf die Einstellungen für die IPv4-Implementierung, vor allem natürlich auf die TCP- und UDP-relevanten Parameter, und beschäftigen uns nicht weiter mit Parametern speziell für IPv6. Alle Parameter können

Tab. A.1 Wichtige Linux-Kernelparameter unter /proc/sys/net/ipv4

Parameter	Bedeutung
tcp_mem	Drei Werte: Minimaler, maximaler und standardmäßig vergebener Speicherbereich, der im System für alle TCP-Sockets zusammen verwendet werden darf (in Bytes).
tcp_rmem	Drei Werte: Minimaler, maximaler und standardmäßig vergebener Speicherbereich, der für den Empfangspuffer eines einzelnen TCP-Sockets (in Bytes) verwendet werden darf.
tcp_wmem	Drei Werte: Minimaler, maximaler und standardmäßig vergebener Speicherbereich, der für den Sendepuffer eines einzelnen TCP-Sockets (in Bytes) verwendet werden darf.
tcp_keepalive_time	Der Parameter gibt die Zeit zwischen dem zuletzt gesendeten TCP-Segment und der ersten Keepalive-Nachricht an.
tcp_keepalive_intvl	Mit diesem Parameter wird die Zeit zwischen zwei aufeinanderfolgenden Keepalive-Nachrichten eingestellt, wenn der Partner nicht antwortet.

(Fortsetzung)

Tab. A.1 (Fortsetzung)

Parameter	Bedeutung
tcp_keepalive_probes	Dieser Parameter gibt die Anzahl der Keepalive-Versuche an, die unternommen werden soll, bevor die Verbindung eingestellt wird.
tcp_sack	Kennzeichen, ob Selective Acknowledgement anstelle von Go-back-N (SACK) aktiviert ist.
tcp_fack	Kennzeichen, ob Congestion Avoidance und Fast Retransmit aktiv ist. Nur möglich, wenn tcp_sack nicht aktiviert ist.
tcp_fin_timeout	Längste Wartezeit, nach der eine Verbindung, die sich im FIN_WAIT_2-Zustand befindet, eliminiert wird.
tcp_window_scaling	Kennzeichen, ob eine Unterstützung für große TCP-Fenster (siehe Sliding-Window-Mechanismus) nach RFC 1323 zulässig ist. Ein Wert 1 bedeutet, dass das TCP-Window größer als 65535 Byte sein kann.
tcp_max_ssthresh	Maximal möglicher Schwellwert beim Slow-Start-Verfahren gemessen in Bytes.
tcp_max_syn_backlog	Maximale Anzahl an Verbindungsaufbauversuchen, die sich ein passiver TCP-Partner in seinem Backlog merkt. Dient zum Ressourcenschutz bei SYN-Flooding.
tcp_synack_retries	Maximale Anzahl an Versuchen des passiven Partners, um die Verbindung vollständig aufzubauen, wenn im Dreiwege-Handshake das dritte Segment (ACK-Flag=1) ausbleibt.
tcp_retries1	Anzahl an Übertragungsversuchen ohne Bestätigung, nach denen eine Verbindung als gestört angesehen wird.
tcp_retries2	Anzahl an Übertragungsversuchen für Keepalive-Probes ohne Bestätigung, nach denen eine Verbindung abgebaut wird.
tcp_syncookies	Kennzeichen, ob SYN-Cookies als Maßnahme gegen SYN-Flooding eingeschaltet sind. SYN-Cookies werden gesendet, wenn der TCP-Backlog überlastet ist.
tcp_allowed_congestion_control	Zulässige Staukontrollmechanismen (reno usw.).
tcp_available_congestion_control	Insgesamt im Kernel verfügbare Staukontrollmechanismen.
tcp_congestion_control	Aktuell eingestellte Staukontrollmechanismen.
tcp_mtu_probing	MTU Path Discovery aktivieren.
tcp_tw_reuse	Kennzeichen, ob eine vorzeitige Wiederverwendung von Verbindungen im Zustand TIME_WAIT zulässig ist, ohne dass der Time-Wait Timer abgewartet werden muss. Die Wiederverwendung wird nur durchgeführt, wenn sie aus Protokollsicht sicher ist.
tcp_ecn	Kennzeichen, ob Explicit Congestion Notification aktiv ist oder nicht.
udp_mem	Drei Werte: Minimaler, maximaler und standardmäßig vergebener Speicherbereich, der im System für alle UDP-Sockets zusammen verwendet werden darf (in Speicherseiten).
udp_wmem_min	Minimalgröße eines Sendepuffers für ein UDP-Socket.
udp_rmem_max	Maximalgröße eines Empfangspuffers für ein UDP-Socket.

in der entsprechenden Linux-Kerneldokumentation nachgelesen werden (Linux-TCP-1 2017 und Linux-TCP-2 2017).

Der Sourcecode der TCP-Implementierung für Linux kann in (Linux-TCP-1 2017) und (Linux-TCP-2 2017) studiert werden.

A.2 Windows

Die TCP/IP-Parameter sind im Windows-Betriebssystem im Windows Registry gespeichert. Sie können mit dem Programm *RegEdit* (Registrierungs-Editior) in der Rolle des Administrators verändert werden. Die Parameter befinden sich in verschiedenen Registry-Pfaden und sind zum Teil auch noch verschiedenen Netzwerk-Interface zugeordnet. In jedem Windows-Derivat ist dies etwas anders. Ein wichtiger Pfad für TCP/IP-Parameter im Windows Registry hat die Bezeichnung *HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services/Tcpip\Parameters*, jeweils im entsprechenden Netzwerkzugang (Interface). Ein anderer wichtiger Pfad ist *HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services/AFD\Parameters*. Wichtige TCP/IP-Parameter sind in der Tab. A.2 zusammengefasst.

Eine Veränderung im Registry ist mit Sorgfalt auszuführen. Man muss die Auswirkungen einschätzen können. Änderungen sind erst nach einem Neustart des Systems wirksam.[1]

Neben der Möglichkeit der Editierung im Windows Registry ist auch eine Veränderung von Parametern über das Kommando *netsh* (ältere Windows-Versionen) oder über die *Microsoft Powershell*[2] (aktuell empfohlene Variante) als Administrator möglich. Das Kommando netsh wird in Zukunft wohl nicht mehr gepflegt. Ein Anzeigen der aktuellen TCP-Parametereinstellungen kann in der Kommandoeingabe zum Beispiel mit dem Kommando

$$netsh \text{ int } tcp\ show\ global$$

erfolgen. Über die Powershell sind vielfältige Einstellmöglichkeiten vorhanden. Unter anderem können verschiedene Staukontrollmechanismen für Hochleistungsnetze, Unterstützung für ECN, die initiale und die minimale RTO eingestellt werden. Die eingestellten Parameterwerte lassen sich mit dem Kommando (cmdlet) *Get-NetTCPSetting* anzeigen. Ein Verändern von Parametern kann über entsprechende Kommandos erfolgen. Beispielsweise kann man mit dem Powershell-Kommando

$$Set - NetTCPSetting - SettingName\ InternetCustom - EcnCapability\ Enabled$$

ECN zugelassen werden. Über netsh lautet der entsprechende Befehl

$$netsh \text{ int } tcp\ set\ globalecncapability = enabled.$$

[1] Informationen hierzu findet man unter https://technet.microsoft.com/en-us/library/cc957548.aspx. Zugegriffen am 20.07.2017.

[2] https://technet.microsoft.com/en-us/library/dn264983.aspx. Zugegriffen am 17.08.2017.

Tab A.2 Wichtige Windows-TCP/IP-Parameter

Parameter	Bedeutung
TcpMaxDataRetransmissions	Maximale Anzahl an Übertragungswiederholungen für das Senden eines TCP-Segments.
TcpWindowSize	Fenstergröße für den Sliding-Window-Mechanismus in Bytes (8192 bis 65535).
TcpMaxConnectTransmissions oder TcpMaxSYNRetransmissions	Anzahl an Versuchen, die unternommen werden, um eine Verbindung aufzubauen (Senden von Connect-Requests mit SYN-Flag=1).
TcpNumConnections	Anzahl an maximal gleichzeitig geöffneten TCP-Verbindungen.
TcpTimedWaitDelay	Wartezeit im Zustand TIME_WAIT beim Verbindungsabbau auf der passiven Seite.
TcpNoDelay	Kennzeichen, ob der Nagle-Algorithmus angewendet werden soll oder nicht.
TcpAckFrequency	Der Parameter legt fest, dass jedes ankommende TCP-Segment sofort bestätigt wird, also Clark`s Algorithmus nicht angewendet wird.
SackOpts	Kennzeichen zum Aktivieren von Selective Acknowledgement (SACK) gemäß RFC 2018.
Tcp1323Timestamps	Der Parameter legt fest, ob die Timestamps-Option verwendet wird (RFC 1323).
MaxUserPort	Größte Portnummer, die vergeben werden darf.
EnableDynamicBacklog MinimumDynamicBacklog MaximumDynamicBacklog	Kennzeichen, ob eine dynamische Festlegung der Obergrenze anstehender Verbindungsaufbauversuche zulässig ist. Die weiteren Parameter wie MinimumDynamicBacklog und MaximumDynamicBacklog legen dann die Grenzen fest. Dies dient der Abmilderung der Auswirkungen von SYN-Flooding-Angriffen.
KeepAliveInterval	Anzahl an maximalen Versuchen von Keepalive-Nachrichten ohne eine Antwort des Partners zu erhalten.

Weiterführende Literatur

Abts, D. (2007) Masterkurs Client/Server-Programmierung mit Java, 2. Auflage, Vieweg Verlag, 2007

Badach, A.; Hoffmann, E. (2001) Technik der IP-Netze, Hanser-Verlag, 2001

Bengel, G. (2001) Verteilte Systeme, 2. Auflage, Vieweg-Verlag, 2001

Bhuiyan, H.; McGinley, M.; Malathi, T. L.; Veeraraghavan, M. (2017) TCP Implementation in Linux: A Brief Tutorial, http://www.ece.virginia.edu/cheetah/documents/papers/TCPlinux.pdf, letzer Zugriff am 25.08.2017

Comer, Douglas, E. (2002) Computernetzwerke und Internets, 3. überarbeitete Auflage, Pearson Studium, 2002

Coulouris, G.; Dollimore, J.; Kindberg, T. (2012) Verteilte Systeme Konzepte und Design, Pearson Studium, 2012

Kurose, J. F.; Ross, K. W. (2014) Computernetzwerke, 6. aktualisierte Auflage, Pearson Studium, 2014

Linux-TCP-1 (2017) https://github.com/ahmadrezamontazerolghaem/Project/wiki/Tcp-implementation-in-linux-kernel., letzter Zugriff am 25.08.2017

Linux-TCP-2 (2017) https://github.com/torvalds/linux/blob/master/net/ipv4/tcp.c, letzter Zugriff am 25.08.2017

Salman, A.; Schulzrinne, H. (2004) An Analysis of the Skype Peer-to-Peer Internet Telephony Protocol, 2004, http://arxiv.org/abs/cs/0412017v1. Zugegriffen: 05.07.2017

Schill, A.; Springer, T. (2007) Verteilte Systeme, Springer Verlag, 2007

Tanenbaum, A. S.; van Steen, M. (2008) Verteilte Systeme Grundlagen und Paradigmen, 2. Aktualisierte Auflage, Pearson Studium, 2008

Turner, K. J. (1993) Using Formal Description Techniques An Introduction to ESTELLE, LOTUS und SDL, John Wiley & Sons, 1993

Weber, M. (1998) Verteilte Systeme, Spektum Akademischer Verlag, 1998

Zitterbart, M. (1995) Hochleistungskommunikation Band 1: Technologie und Netze, Oldenbourg Verlag, 1995

P. Mandl, *TCP und UDP Internals*, https://doi.org/10.1007/978-3-658-20149-4

Stichwortverzeichnis

A
ACK-Duplikat 65, 83
ACK-Flag 55
AIMD-Algorithmus 83
AJAX (Asynchronous JavaScript and XML) 11
Anwendungsschicht 4
Anycast 50
Asynchronous JavaScript and XML (AJAX) 11

B
Bandbreiten-Verzögerungs-Produkt 74
Berkeley Sockets 107
Bestätigungsnummer 51
Bitübertragungsschicht 3
Blocking I/O 155

C
CEFSM 57
CEP (Connection End Points) 27
Clarks Algorithmus 72
Congestion Control 40
Connection End Point (CEP) 27

D
Darstellungsschicht 4
Datagramm 96
DatagramPaket 124
DatagramSocket 124
Defragmentierung 24, 40
Demultiplexieren 41
Deserialisierung 138
Dienst 6
Diensterbringer 3
Dienstnehmer 3
DNS 50
  -Root-Name-Server 50
Dreiwege-Handshake 53

E
EFSM 57
Einerkomplement 99
EJB 155
Electronic Mail 13
Empfangsfenster 81
Ende-zu-Ende-Kommunikation 5, 44
Ereigniswartepunkt 155
Ethernet 21
Explicit Congestion Notification 84

F
Fast
  -Recovery 67, 83
  Retransmission 83
  -Retransmit 67
Fcntl() 117
Fenstermechanismus 25
FIN-Flag 55
Finit State Machine 57
Flusskontrolle
  fensterbasierte 37
  Stop-and-Wait-Protokoll 37

P. Mandl, *TCP und UDP Internals*, https://doi.org/10.1007/978-3-658-20149-4

Flusssteuerung 24
Folgenummer 28, 51
Fragmentierung 24, 40
Frame 20

H
HTTP 11
  1.0 10
  2.0 10
HTTPS 13
Huckepack-Verfahren 24

I
IGMP 137
IMAP4 14
InetAddress 124
Instant
  Messaging 16
  -Messaging-Dienst 16
  Multimedia Messaging (IMM) 16
Instanz 6
Internet Protokoll 10
IPv4 10
IPv6 10
ISO/OSI-Referenzmodell 2

J
Java-Sockets 124

K
Karn-Algorithmus 88
Kommunikationsprotokoll 3

L
Längenanpassung 24
Linux TCP-Kernelparameter 173
Login Server 18
long fat networks 74, 78

M
Mail User Agent (MUA) 13
Maximum Segment Size Option (MSS) 73
Mealy-Automat 57
Messenger 16
Middleware 155
MIME (Multipurpose Internet Mail Extensions) 15
Mini-Framework 140
Moore-Automat 57
MSA (Mail Submission Agent) 13
MSS 45
  Clamping 73
MSS (Maximum Segment Size Option) 73
MTA (Mail Transfer Agents) 13
MTU (Maximum Transfer Unit) 46
Multicast Socket 136
Multicast Socket 130
Multiplexieren 25, 41

N
Nachricht 20
Nagle-Algorithmus 70
NAK, implizites 67, 68
netsh 177
Netty 155
Netzwerkschicht 4
NIO 155
Nonblocking I/O 155
N-SAP 26

O
OSI-Modell 2

P
Paket 20
Path MTU Discovery 47
PAWS 54
PAWS (Protect Against Wrapped Sequences) 79
Pfadkapazität 36, 38
Pipelining 33
POP3 (Post Office Protocol, Version 3) 13
POSIX Socket API 107
Powershell 177
Probing-Phase 81
Protect Against Wrapped Sequences (PAWS) 79
Protokollfunktion 23
Protokollinstanz 6
Protokollstack 5

Q
Quality of Service 25
Quittierung 24
Quittierung, kumulative 64
Quittierungsverfahren
  negativ-selektives 33
  positiv-kumulatives 33
  positiv-selektives 33

R
RMI 155
Root-Name-Server 50
Round-Trip-Time (RTT) 87
RST-Flag 55
RTT (Round Trip Time) 88
RTT (Round-Trip-Time) 87

S
SACK
  Option 75
  Permitted Option 75
SAP (Service Access Point) 3, 6
Segment 20
select 155
Sequenznummer 28
Sequenznummernangriff 93
Sequenznummernkollision 78
Serialisierung 138
ServerSocket 124
„ServerSocket“ 129
Service
  Access Point (SAP) 3, 6
  Provider 3
Session Hijacking 93
Sicherungsschicht 4
Silly-Window-Syndrom 71
Sitzungsschicht 4
Skype 18
  Client 18
Sliding-Windows-Verfahren 37
Slow-Start 80
  -Algorithmus 80
  -Phase 81
Slow-Start-Algorithmus 83
SMTP-Server 13
Socket 129
Socket-API 107
  accept() 113
  bind() 112
  close() 114
  connect() 113
  fork() 116
  getsocketopt() 117
  listen() 113
  recv() 114
  recvfrom () 115
  send() 114
  sendto() 115
  setsocketopt() 116
  socket() 112
SocketChannel 125
Socket-Schnittstelle 106
Staukontrolle 40, 79
Stop-and-Wait-Protokoll 33
Super Node 19
Synchronisation von close 139
SYN-Flooding 93

T
TCP 44
  -Flags 51
  -Flusskontrolle 68
  -Header 50
  /IP-Referenzmodell 6
  -Port 51
  -Portnummer 48
  Reno 83
  -Sockets 107
  -Stream 50
  Tahoe 80
  -Verbindungsabbau 55
  -Verbindungsaufbau 53
  -Zustand 56, 58, 60
  -Zustandsautomat 57
TCP-Timer-Management 87
  Close-Wait Timer 92
  Keepalive Timer 90
  Persistence-Timer 92
  Retransmission-Timer 88
  RTO 88
  Time-Wait Timer 91
Timestamps Option (TSopt) 76
T-PDU 25

Transfersyntax 4
Transportadresse 26
Transportschicht 4
Transportsystem 4
Transportzugriffsschicht 4
T-SAP 25
TSopt (Timestamps Option) 76

U
Überlastfenster 80
Überlaststeuerung 25
Überlastvermeidungsphase 81
Übertragungswiederholung
  Go-Back-N 35
  selektiv 35
UDP 95
  -Header 97
  -Port 96
  -Pseudoheader 99
  -Sockets 110
  -Spoofing 103
URI (Uniform Resource Identifier) 11
URL (Uniform Resource Locators) 11

V
Verarbeitungsschicht 4
Vermittlungsschicht 4
Vorrangtransfer 24

W
WhatsApp 16
Windows
  Registry 177
  TCP-Kernelparameter 177
Window Scale Option (WSopt) 74
WSopt (Window Scale Option) 74

Z
Zwei-Armeen-Problem 30