LEHRBUCH

Peter Mandl

TCP, UDP und QUIC Internals

Protokolle und Programmierung

2. Auflage

Springer Vieweg

TCP, UDP und QUIC Internals

Peter Mandl

TCP, UDP und QUIC Internals

Protokolle und Programmierung

2. Auflage

Springer Vieweg

Peter Mandl
Fakultät für Informatik und Mathematik
Hochschule München
München, Deutschland

ISBN 978-3-658-43987-3 ISBN 978-3-658-43988-0 (eBook)
https://doi.org/10.1007/978-3-658-43988-0

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

Planung/Lektorat: Leonardo Milla
Springer Vieweg ist ein Imprint der eingetragenen Gesellschaft Springer Fachmedien Wiesbaden GmbH und ist ein 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 Transportprotokolle TCP, UDP und QUIC vermittelt. Das Buch soll dazu beitragen, den komplexen Sachverhalt bis ins Detail verständlich zu machen. Dabei liegt der Schwerpunkt vor allem auf praktisch relevanten 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. Transport Layer Security Protocol (TLS)
6. Transportprotokoll QUIC (UDP-Based Multiplexed and Secure Transport)
7. Programmierung von TCP- und UDP-Anwendungen

Kap. 1 gibt in Kürze eine grundlegende Einführung in die wichtigsten Grundbegriffe der Datenkommunikation 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 ver-

tiefenden Protokollmechanismen von TCP vorgestellt. Kap. 4 geht entsprechend auf UDP ein. Um die Authentifizierungs- und Verschlüsselungsmechanismen von QUIC zu verstehen, wird in Kap. 5 ein knapper Überblick über TLS gegeben. TLS ist bereits in QUIC, das in Kap. 6 eingeführt wird, integriert. QUIC ist ein neueres, ursprünglich von Google entwickeltes und mittlerweile standardisiertes Transportprotokoll. Schließlich wird in Kap. 7 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.

Zu jedem Kapitel stehen Aufgaben und deren Lösung online auf der Webseite link.springer.com zur Verfügung. Sie sind über den auf der jeweils ersten Kapitelseite aufgeführten Link erreichbar.

Der Inhalt des Buches entstand zum einen aus mehreren Vorlesungen über Datenkommunikation und verteilte Systeme über mehr als 20 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), aus dem einige Inhalte in das vorliegende Werk übernommen wurden.

Aus Gründen der besseren Lesbarkeit verwenden wir in diesem Buch überwiegend das generische Maskulinum. Dies impliziert immer beide Formen, schließt also die weibliche Form ein.

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.

München, Deutschland Peter Mandl
September 2023

Literatur

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

Zusammenfassung der Erweiterungen in Auflage 2

Dieses Buch befasst sich mit den Transportprotokollen UDP und TCP und deren Nutzung in der Entwicklung von Kommunikationsanwendungen. In der 2. Auflage wurde Kap. 3 zu TCP um die Betrachtung aktueller Staukontrollmechanismen wie TCP NewReno, TCP BIC und das heute in vielen Betriebssystemen standardmäßig verwendete TCP CUBIC ergänzt. Auch das mittlerweile standardisierte Protokoll QUIC wurde im neuen Kap. 6 ergänzt. QUIC ist zwar gemäß TCP/IP-Referenzmodell in der Anwendungsschicht angesiedelt, stellt aber auf der Basis von UDP einen gesicherten Transportdienst vor allem für die WWW-Kommunikation bereit. Die wichtigsten QUIC-Protokollmechanismen werden erläutert, wobei auch auf die Schwächen von TCP und deren Behebung in QUIC eingegangen wird. QUIC nutzt das Transport Layer Security Protocol in der Version 1.3 (TLS 1.3) für die Authentifizierung der Kommunikationspartner und für die Verschlüsselung von Nachrichten bereits implizit. Für Interessierte wird daher in Kap. 5 auch eine kurze Einführung in die Arbeitsweise von TLS gegeben. In allen weiteren Kapiteln wurden kleinere Verbesserungen und Fehlerkorrekturen vorgenommen.

Noch ein Hinweis für die Leser

In diesem Buch werden häufig Requests for Comments (RFCs) 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

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* zusammen.

Ergänzende Information Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann [https://doi.org/10.1007/978-3-658-43988-0_1].

P. Mandl, *TCP, UDP und QUIC Internals*, https://doi.org/10.1007/978-3-658-43988-0_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 1990er-Jahre hinein die ISO (International Organization for Standardization), 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 World Wide Web (WWW), Electronic 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, wobei OSI als Abkürzung für *Open Systems Interconnection* steht, (kurz: OSI-Modell) teilt die gesamte Funktionalität in sieben Schichten (Abb. 1.1) ein. Jede Schicht stellt der darüberliegenden Schicht bzw. 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 gemäß dem OSI-Modell 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 Internetstandards (RFCs) festgehalten.

Gleiche Schichten kommunizieren horizontal über Rechnergrenzen hinweg über *Kommunikationsprotokolle*. 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 nächsthöheren 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* bezeichnet, diejenige, welche den Dienst nutzt, als *Dienstnehmer*.

▶ **[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 (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 Schichten 1 bis 7 haben im ISO/OSI-Referenzmodell folgende Aufgaben:

1. Die *Bitübertragungsschicht* ist die unterste Schicht und stellt eine physikalische Verbindung bereit. Hier werden die elektrischen und mechanischen Parameter festgelegt.

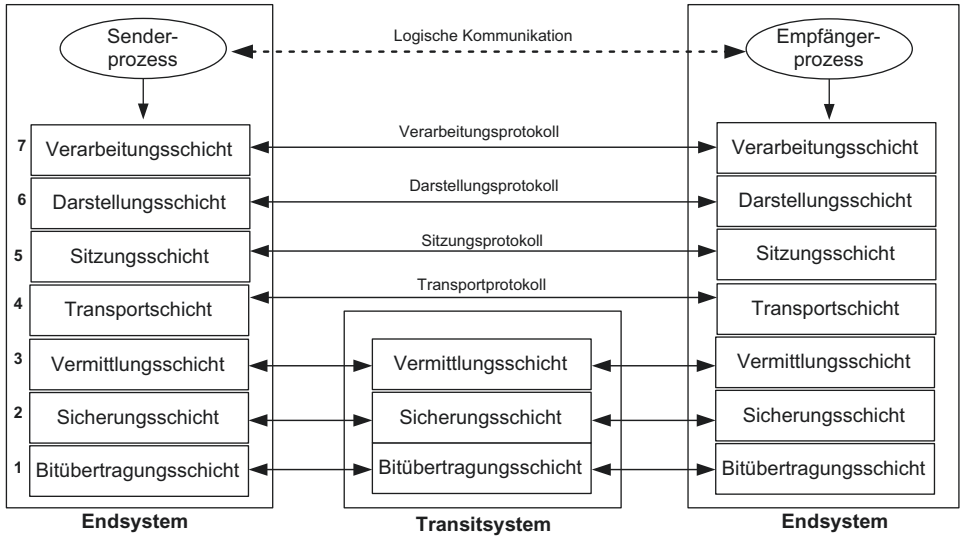


Abb. 1.1 ISO/OSI-Referenzmodell

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* 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* hat die Aufgabe, Verbindungen zwischen zwei Knoten ggf. über mehrere Rechnerknoten hinweg zu ermöglichen. Auch die Suche nach einem günstigen Pfad zwischen zwei Endsystemen wird hier durchgeführt (Wegewahl, Routing).
4. Die *Transportschicht* sorgt für eine Ende-zu-Ende-Beziehung zwischen zwei Kommunikationsprozessen und stellt einen Transportdienst für die höheren Anwendungsschichten bereit.
5. Die *Sitzungsschicht* stellt eine Sitzung (Session) zwischen zwei Kommunikationsprozessen her und regelt den Dialogablauf der Kommunikation.
6. Die *Darstellungsschicht* ist im Wesentlichen für die Bereitstellung einer einheitlichen Transfersyntax zuständig. Dies ist 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* 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*.

► **[Transportsystem und Transportzugriffsschnittstelle]** 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 Codierung von Zeichen verwendet und wird z. B. in Mainframes genutzt.

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

[3] Unicode ist ein hardware- und plattformunabhängiger Code zur ZeichenCodierung 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* 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 nächsthöhere Schicht entfernt.

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

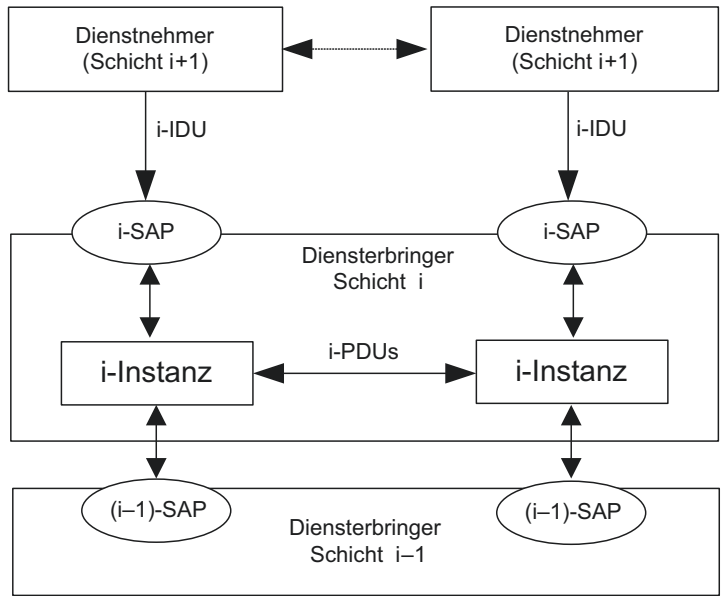


Abb. 1.2 Hierarchische Dienststruktur

ren miteinander, indem sie die konkreten Implementierungen eines Dienstes der Schicht i nutzen. Die Dienste können über einen Dienstzugangspunkt bzw. *Service Access Point* (SAP) genutzt werden. An einem SAPwerden 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* derselben Schicht tauschen Nachrichten, sogenannte Protocol Data Units (PDUs), 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 aufeinanderfolgender Schichten auch als horizontale Kommunikation.

[ISO/OSI-Protokolle]
In den 1990er-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 Internetgemeinde entwickelte TCP/IP-Referenzmodell (Abb. 1.3) ist heute der De-facto-Standard in der Rechnerkommunikation, natürlich vor allem im Internet. Es hat vier Schichten, wobei die Internetschicht (Netzwerkschicht) und die Transportschicht die tragenden Schichten sind. In der Netzwerkschicht wird neben einigen Steuerungsprotokollen im Wesentlichen das Internetprotokoll (Internet Protocol, IP) benutzt.

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

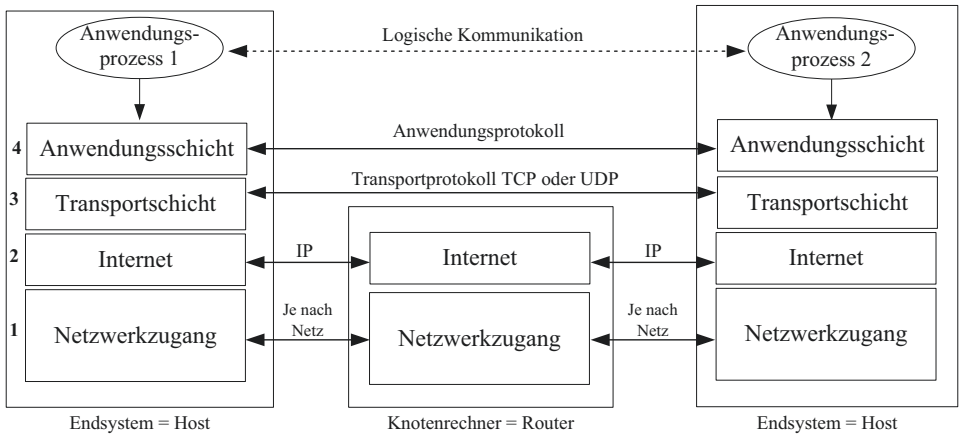


Abb. 1.3 TCP/IP-Referenzmodell

[Steuerungsprotokolle der Internetprotokollfamilie]
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 u. a. 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 müssen 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 Internetpraxis 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 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; MAC = 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 Implementierung 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, u. a. 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 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, die eindeutig im Netzwerk sind, adressiert. Schicht-3-Instanzen erhalten eindeutige IP-Adressen (Internetadresse), die für einen Rechner, genauer gesagt für seine Netzwerkschnittstelle, 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.

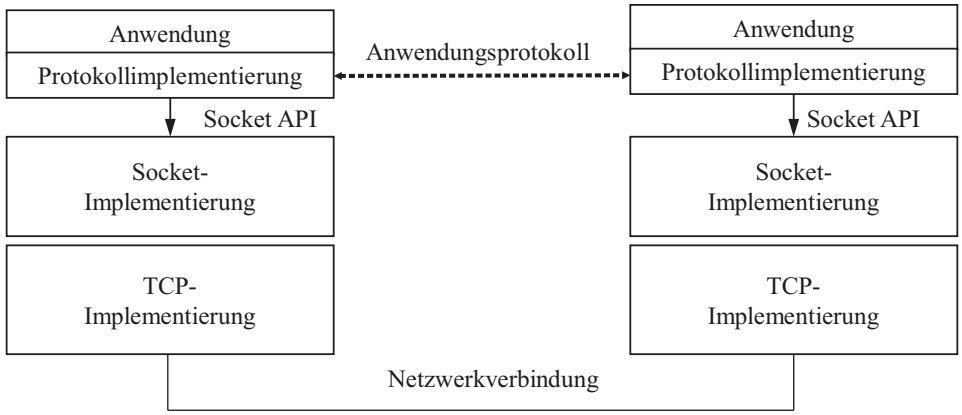


Abb. 1.4 Nutzung des Transportdienstes am Beispiel der Socket API

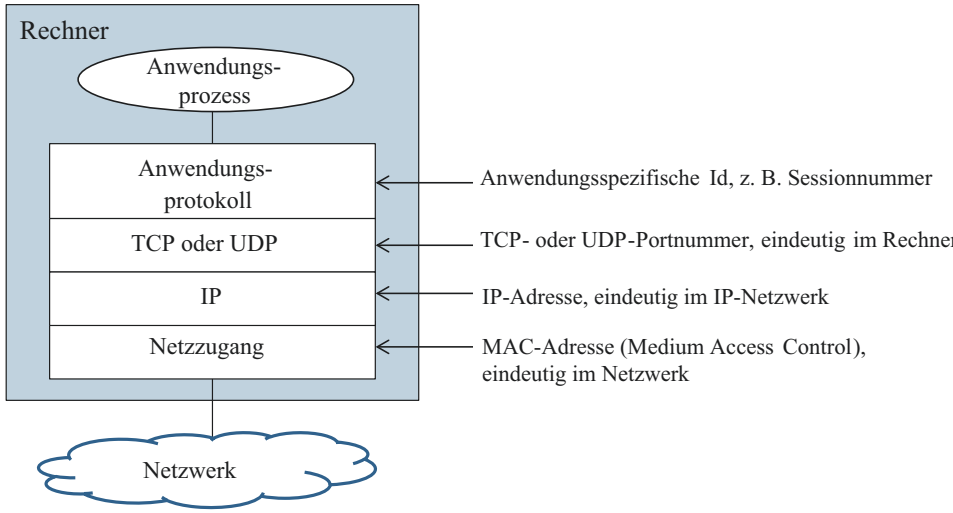


Abb. 1.5 Adressierungsinformationen der einzelnen Protokollschichten

In Abb. 1.6 wird ein Überblick über den gesamten TCP/IP-Protokollstack mit einigen wichtigen Protokollen gegeben. Die Abbildung 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 im folgenden Abschnitt 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.

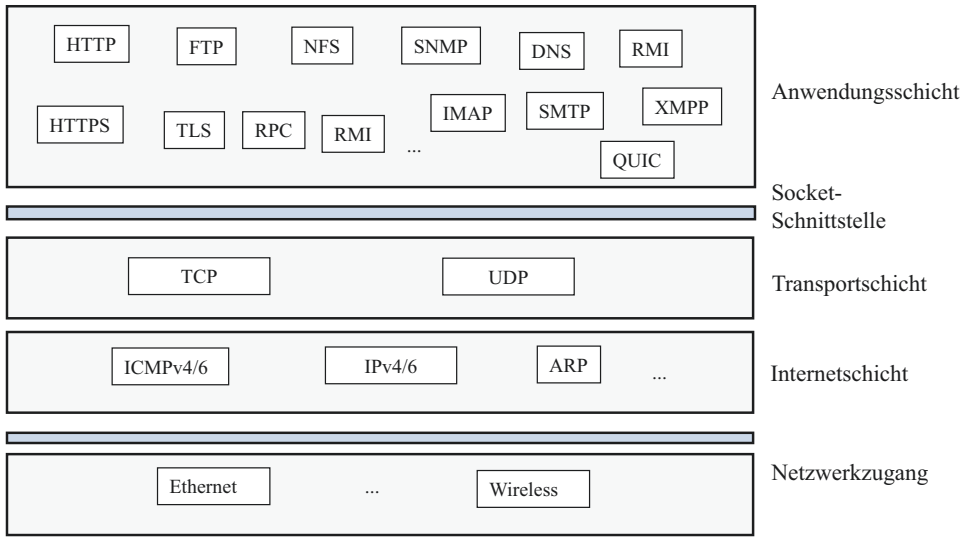


Abb. 1.6 Protokollbeispiele der TCP/IP-Welt

[Vermittlungsschicht im Internet und Internetprotokoll (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 ein Webserver bzw. ein ganzer Cluster von Servern auf der Betreiberseite. Durch die Eingabe einer Adresse im Browser wird eine Betreiberseite adressiert. Dabei wird zunächst eine Verbindung zwischen dem Webbrowser und dem Webserver aufgebaut, über die dann die Kommunikation mithilfe des HTTP-Protokolls abgewickelt wird. Bis zur Ver-

sion 2 von HTTP wurde für die verbindungorientierte Kommunikation das Transportprotokoll TCP verwendet. In der Version 3 wird das neue QUIC-Protokoll für die Verbindung verwendet. Dies ist eigentlich ein Protokoll der Anwendungsschicht, das auf dem Transportprotokoll UDP aufsetzt.

[HTTP/1.0, HTTP/1.1, HTTP/2 und HTTP/3]
HTTP ist das Standardanwendungsprotokoll für die Kommunikation im Web. Es wurde 1991 in der Version 1.0 im RFC 1945 standardisiert. In dieser Version musste für jede Anfrage zum Lesen eines Objekts 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 Jahr 2015 (RFC 7540) wurde das Protokoll nochmals u. a. um Push-Nachrichten, die vom Server initiiert werden können, erweitert. Auch das Übertragen mehrerer Objekte über parallele Streams wurde mit der Version 2 möglich; allerdings blockieren alle Streams, wenn einer ein Übertragungsproblem hat. Abhilfe dafür schaffte HTTP/3, das nicht mehr auf TCP, sondern auf QUIC und das wiederum auf UDP aufsetzt.

HTTP ist also ein Anwendungsprotokoll. Bis zur Version 2 wird der TCP-Port 80 als Standardport verwendet, ab Version 3 wird das Verbindungsmanagement über QUIC und UDP abgewickelt. Die zu übertragenden 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 (URLs) oder über sogenannte Uniform Resource Identifiers (URIs). 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

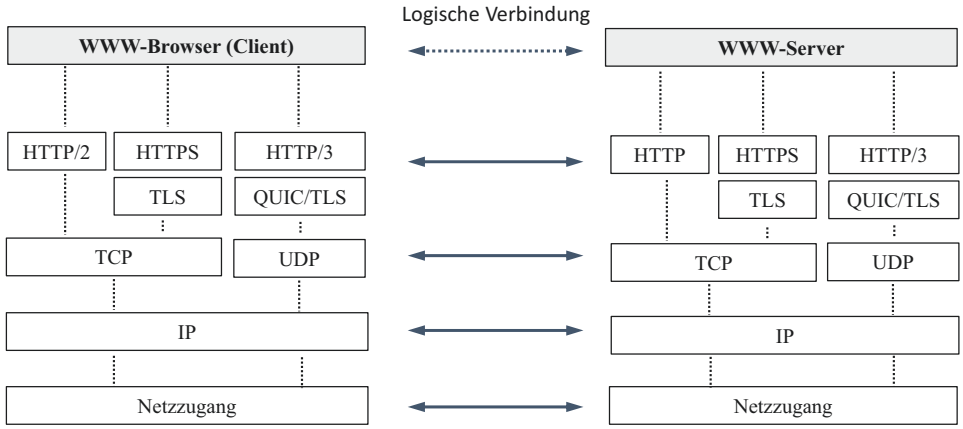


Abb. 1.7 Protokollstack für die webbasierte Kommunikation

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 sind die drei Protokollstacks skizziert, die ein WWW-Browser für die Kommunikation mit dem WWW-Server nutzen kann. Entweder fällt die Wahl auf den Protokollstack HTTP/2+TCP+IP+Netzzugang, HTTP/2+TLS+TCP+IP+Netzzugang (HTTP/2 mit TLS wird auch als HTTPS bezeichnet) oder auf die neuere Variante HTTP/2+QUIC+UDP+IP+Netzzugang. Die letzten beiden Varianten nutzen einen sicheren (verschlüsselten) Kommunikationskanal. Beide Partner müssen prinzipiell denselben Protokollstack nutzen, sonst funktioniert die Kommunikation nicht.

HTTP nutzt sogenannte MIME-Bezeichner (Medientypen) für die Angabe des Inhalts der zu übertragenden Daten. Vom Webbrowser können also beliebige Dateitypen, die einen MIME-Typ besitzen, verarbeitet und über HTTP übertragen werden. Ein Webbrowser gibt im Request an, welche Formate er verarbeiten kann. Der Webserver 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 Webbrowser zum Webserver gesendet. Das Ergebnis wird mit einer HTTP-Response-PDU vom Webserver zum Webbrowser kommuniziert.

Abb. 1.8 zeigt den grundsätzlichen Ablauf der Kommunikation ohne Verschlüsselung mit HTTP/2. Man sieht, dass vor der HTTP/2-Kommunikation ein TCP-Verbindungsaufbau durchgeführt und nach dem Empfang der Response-PDU die Verbindung (vom Client)

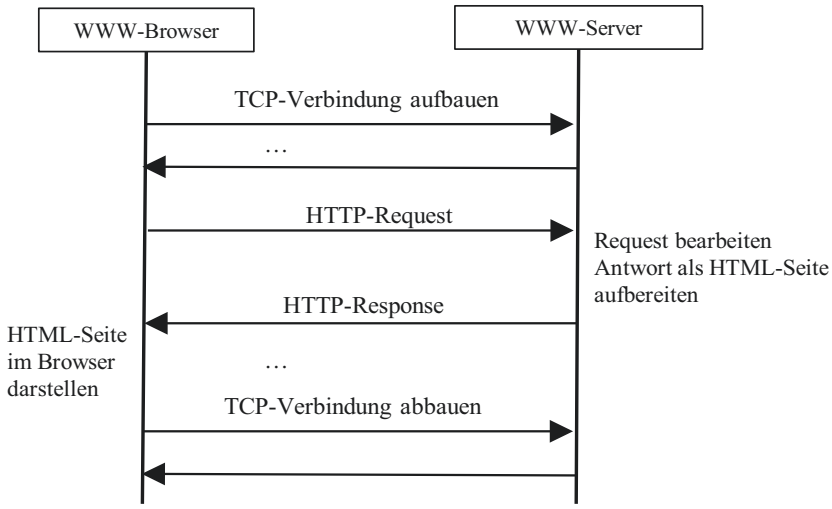


Abb. 1.8 Beispiel einer Kommunikation zwischen WWW-Browser und WWW-Server

wieder abgebaut wird. Wie der Verbindungsaufbau im Detail aussieht, werden wir noch ausführlich diskutieren.

Für sicherheitskritische Webzugriffe verwendet man heute eine Erweiterung von HTTP/2, 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. Nach dem TCP-Verbindungsaufbau ist noch ein TLS-Handshake erforderlich, um einen sicheren Kanal zwischen den Kommunikationspartnern aufzubauen. TLS ist ein Verschlüsselungsprotokoll, in dem auch ein Schlüsselaustausch erfolgt (Eckert 2014). Da bei HTTP/3 der Transport über QUIC erfolgt, wird der sichere Kanal bereits automatisch erzeugt. QUIC enthält bereits eine neuere TLS-Version (TLS 1.3). Die Transport- und Sicherheitsmechanismen werden wir in Kap. 5 und 6 noch detaillierter betrachten.

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-Server-Anwendung. 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 zu-

geordnet. Die Mailboxen werden üblicherweise in den E-Mail-Gateways von Internetprovidern verwaltet. Die E-Mail-Gateways sind Serverrechner mit speziellen Softwarebausteinen für die Mailabwicklung. 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* (MTAs) 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 wohlbekannten (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 Internetbenutzer kann sich z. B. bei einem Internetprovider (z. B. T-Online) eine Mailbox einrichten und erhält beispielsweise über das Protokoll POP3 (Post Office Protocol, Version 3) unter Nutzung des wohlbekannten TCP-Ports 110 Zugang zu seiner Mailbox. POP3 ist im RFC 1939 definiert, SMTP im RFC 5321.

Technisch gesehen muss der Internetbenutzer 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 Internetprovider herstellen. Dies erfolgt bei privaten Internetbenutzern 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 Protokolloperationen 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 dem wohlbekannten TCP-Port 143. IMAP4 ist ebenfalls ein Internetstandard. Im Gegensatz zum POP3-Protokoll verbleiben die E-Mails in der Regel auf dem Mailserver und werden nur bei Bedarf auf den Clientrechner 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.

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

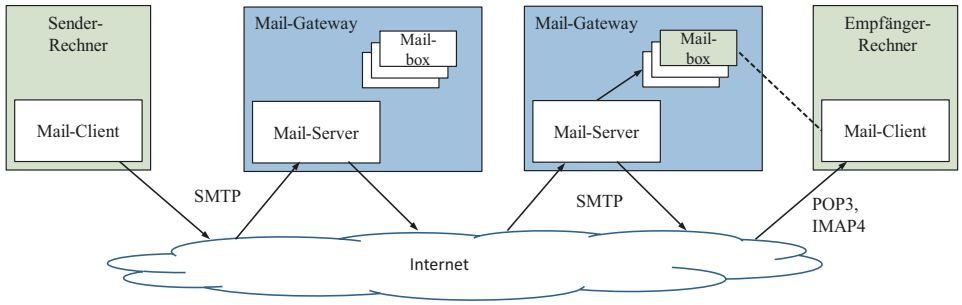


Abb. 1.9 Zusammenspiel der E-Mail-Komponenten

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 Mailbenutzer.

POP3 wird heute meist von privaten Mailbenutzern 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 Mailaustausch nach dem sogenannten Store-and-Forward-Prinzip. Beim Mail-Gateway ankommende E-Mails werden zunächst in den Mailboxen zwischengespeichert, um sie dann bei Bedarf, wenn die Verbindung zum adressierten Mailbenutzer 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

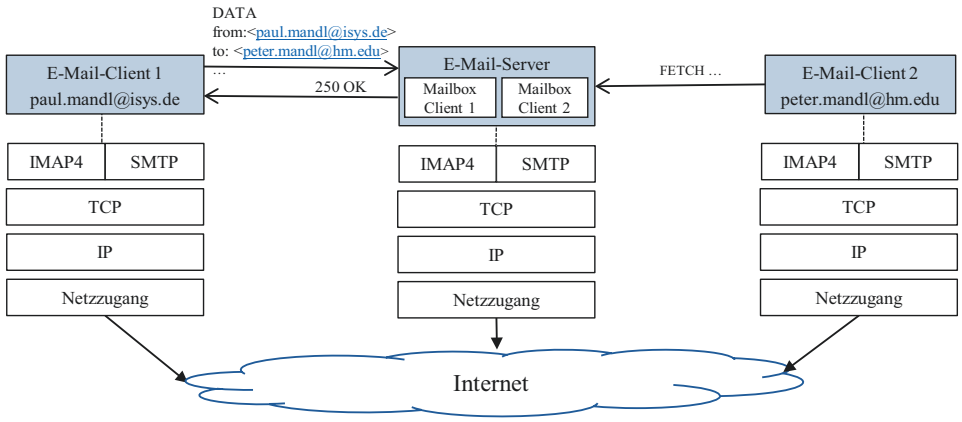


Abb. 1.10 E-Mail-Protokollstack

E-Mail-Server zusätzlich über SMTP kommunizieren. In unserem Beispiel sendet 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 Mailsystem 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, Videos, Audio) übertragen werden. Sender und Empfänger können sich über die Codierung der Daten verständigen, wobei ein Nachrichtenfeld namens *Content-Type* verwendet wird. Es sind mehrere Codierungsmethoden spezifiziert, die die Übertragung im textbasierten E-Mail-System ermöglichen. Die binären Objekte werden beim Sender codiert und beim Empfänger wieder decodiert.

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.

[6] WhatsApp gehört heute zu Facebook. Im Jahr 2014 waren bei WhatsApp alleine in Deutschland etwa 30 Mio. Teilnehmer registriert.

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 (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 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.

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 in Abb. 1.11 dargestellt 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 sowie 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 Eingabephase, 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 ge-

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

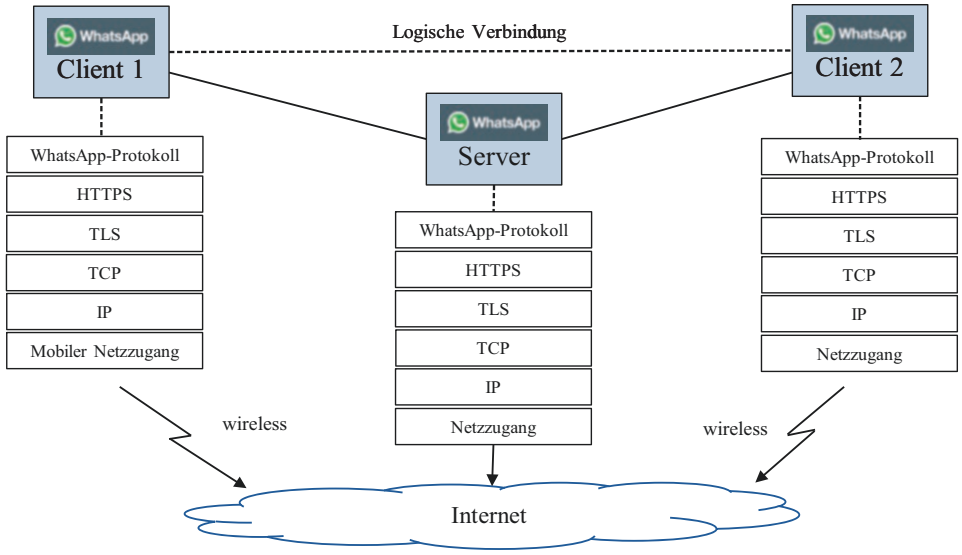


Abb. 1.11 WhatsApp-Protokollstack

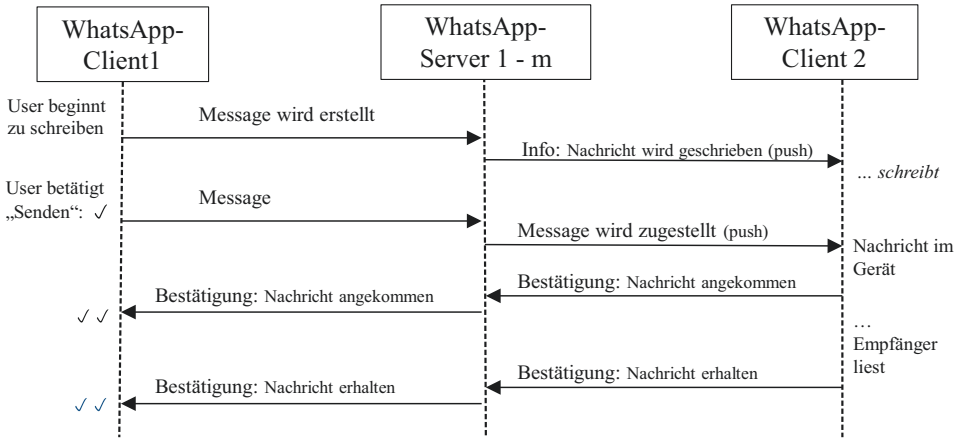


Abb. 1.12 WhatsApp-Nachrichtenfluss am Beispiel

setzt. 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 aufwendig. Bei einer Gruppenkommunikation wird sogar versucht, die Häkchen erst dann zu setzen, wenn alle Gruppenmitglieder die Nachricht auf ihren Geräten empfangen bzw. ausgelesen haben.

▶ **[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 Telefondienst, 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 Jahr 2003 von der Firma Skype Technologies entwickelt und gehört seit 2011 zu Microsoft.

Das ursprüngliche Skype-Anwendungssystem verfügte über folgende Komponenten:

- Der *Login-Server* 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* sind Anwendungen, welche die Schnittstelle zum Benutzer und auch die *Peers* in dem ursprünglichen Peer-to-Peer-Netzwerk darstellen.
- *Supernodes* oder *Superpeers* sind spezielle Skype-Clients mit Zusatzaufgaben für die Kommunikation. Jeder normale Skype-Client musste eine Verbindung zu einem Supernode aufbauen. Jeder Skype-Client konnte auch Supernode 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 Supernodes, damit ein Verbindungsaufbau zu einem Supernode in der Nähe möglich war. Supernodes wurden für verschiedene Caching- und Kommunikationsaufgaben sowie als Vermittler eingesetzt. Falls keine Verbindung über einen Supernode möglich war, nutzte ein Skype-Client fest eingebaute IP-Adressen von Bootstrap-Servern (hart codiert in der ausführbaren Datei des Skype-Clients). Wenn möglich, wurde die Kommunikation direkt zwischen den Peers ausgeführt; bei zu schlechter Verbindung wurden Supernodes dazwischengeschaltet. Supernodes 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 Supernodes mit Spezialaufgaben etwa

für das Registrieren erforderlich. In den vergangenen Jahren wurden die Supernodes 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 Supernode-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 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 Chatprotokolls oder der Header für die Webkommunikation (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 im Folgenden 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/

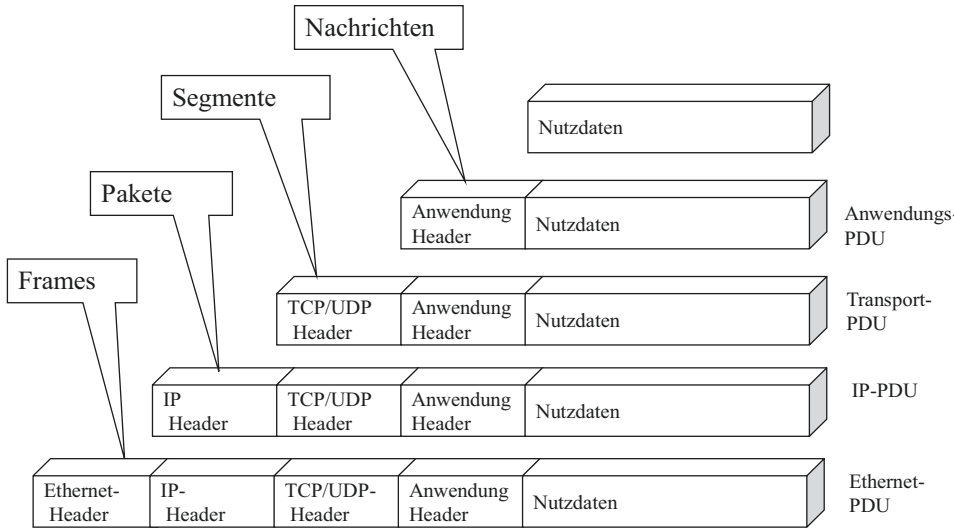


Abb. 1.13 Typischer Nachrichtenaufbau

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 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, dass 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 oberhalb der Schicht 3 nicht.

[Ethernet]
Ethernet ist eine Netzwerktechnologie für lokale Netzwerke (Local Aread Networks, LANs), welche die Funktionalität der unteren beiden Schichten gemäß ISO/OSI-Referenzmodell abdeckt. Ethernet wurde Anfang der 1970er-Jahre von R. Metcalfe[8] bei der Firma Xerox entwickelt und später von den Firmen Xerox, DEC und Intel zum Standard IEEE 802.3 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 Protokoll-Header 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.

Abb. 1.14 zeigt nochmals am Beispiel einer Nachricht zur Kommunikation zwischen WWW-Browser und WWW-Server über HTTP/2, wie beim Sender die Gesamtnachricht

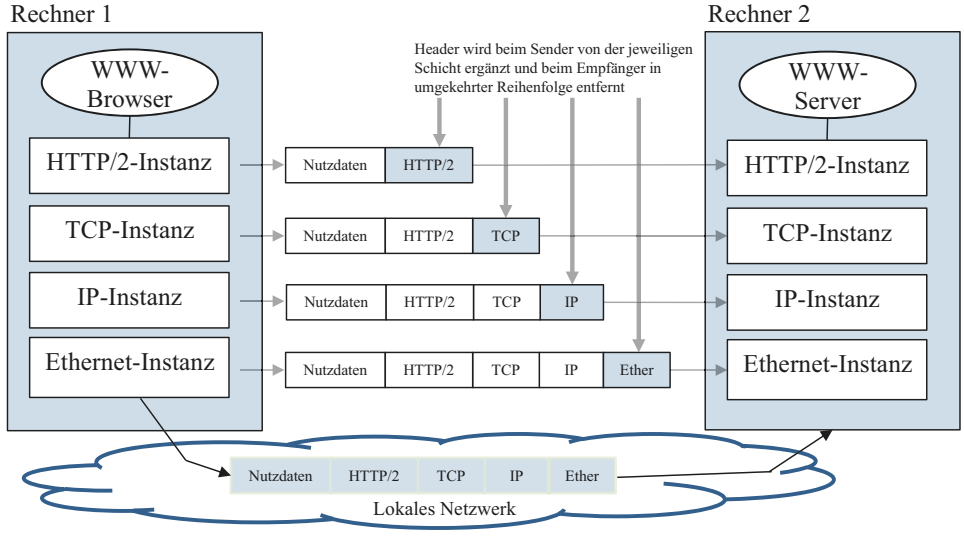


Abb. 1.14 Nachrichtenaufbau über die Schichten

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

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 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 zunächst einige wichtige Protokollfunktionen allgemein erläutert werden.

Ergänzende Information Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann [https://doi.org/10.1007/978-3-658-43988-0_2].

P. Mandl, *TCP, UDP und QUIC Internals*, https://doi.org/10.1007/978-3-658-43988-0_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 ggf. die lückenlose Zustellung sind hier anzusiedeln. Die *Quittierung* ist ebenfalls 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. Auch Mischungen der einzelnen Verfahren sind möglich. Eine Bestätigung im sogenannten Huckepack-Verfahren (piggyback 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 sowie der Überlast- und der Ratensteuerung. Die *Flusssteuerung* schützt den Empfänger vor Überlastung durch den Sender. Hierfür werden z. B. Fenstermechanismen (Sliding-Window-Verfahren) 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 es 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 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 ggf. verfälscht werden.
- Die Reihenfolge ist nicht garantiert.

Verbindungsorientierte Protokolle sind aufgrund der erweiterten Funktionalität komplexer und daher aufwendiger 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 Verzögerungsschwankungen, auch als Jitter bezeichnet.

2.1.3 Zwischenspeicherung von Nachrichten

Wenn ein sendender Prozess eine Nachricht aus der Anwendungsschicht an die Transportschicht übergibt, wird diese nicht sofort gesendet, sondern erst einmal in einen Zwischenspeicher oder Sendepuffer gelegt. Hierfür ist die Transportinstanz verantwortlich, die den Sendepuffer verwaltet. Die zuständige Transportinstanz kopiert die Nachricht üblicherweise im Adressraum des sendenden Prozesses in den eigenen Pufferbereich. Die Transportinstanz entscheidet dann nach eigenen Regeln, wann die Nachricht gesendet wird. Bei verbindungsorientierter Kommunikation kann es durchaus sein, dass das Versenden aufgrund von Flusskontroll- oder Staukontrollaspekten etwas verzögert wird.

Nachrichten, die empfangen werden, werden von der Instanz der darunterliegenden Kommunikationsschicht (bei TCP/IP ist dies die IP-Instanz) nach oben weitergereicht und in einen Empfangspuffer der Transportinstanz gelegt. Erst wenn der Empfangsprozess mit einer Receive-Operation eine Nachricht entgegennimmt, wird diese vom Empfangspuffer entfernt und an der Schnittstelle nach oben weitergereicht. In Abb. 2.1 sind die Pufferbereiche zweier kommunizierender Transportinstanzen und die Übertragungswege über die Pufferbereiche veranschaulicht. Ein Sendeaufruf (1) des Prozesses bewirkt also erst einmal, dass die Nachricht im Sendepuffer abgelegt wird, dann erfolgt entsprechend den Regeln des Protokolls die Übertragung zum Empfänger (2), und der Empfängerprozess liest die Nachricht zum Beispiel mit einer Receive-Operation aus, wenn er sie verarbeiten möchte (3). Es soll noch erwähnt werden, dass eine Ende-zu-Ende-Kommunikation auf der Transportebene bedeutet, dass die Übertragung einer Nachricht abgeschlossen ist, wenn diese im Empfangspuffer der empfangenden Transportinstanz gelandet ist. Bei verbindungsorientierter Kommunikation muss der Sender auch noch eine Bestätigung des Empfangs erhalten haben. Es ist also nicht so, dass der empfangende Prozess die Nachricht bereits gesehen hat.

Pufferspeicher werden von der Transportinstanz verwaltet und verursachen auch einen gewissen Overhead. Er wird im Adressraum des Prozesses oder je nach Implementierung in einem vom Betriebssystem verwalteten Speicherbereich reserviert, der natürlich begrenzt ist. Meist kann man die Puffergrößen in den Betriebssystemen konfigurieren. Es kommt auf den Protokolltyp und die Implementierung an, was bei einem Überlauf eines Pufferbereichs passiert. Bei verbindungsorientierten Transportprotokollen wird ein Überlauf des Empfangspuffers in der Regel durch Flusskontrollmechanismen vermieden. Wenn ein Sendepuffer voll ist, wird an der Schnittstelle ein Sendeaufruf abgelehnt, sodass es auch hier nicht zu Pufferüberläufen kommt. Bei verbindungslosen Transportprotokollen kann es durchaus dazu kommen, dass Nachrichten verworfen werden, wenn der Pufferbereich nicht ausreicht.

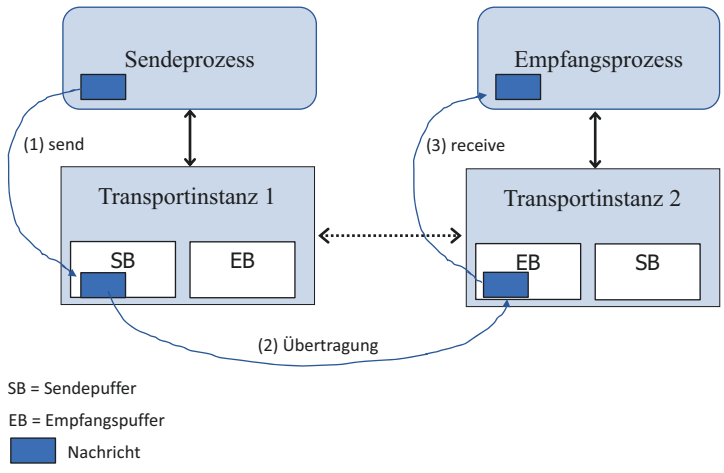


Abb. 2.1 Sende- und Empfangspuffer in der Transportinstanz

2.2 Verbindungsmanagement und Adressierung

In Abb. 2.2 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.

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* (CEPs) bezeichnet. Beim Verbindungsaufbau erfolgen eine Synchronisation zwischen den Partnern und die Einrichtung des Verbindungskontexts auf beiden Seiten. Hierfür werden Synchronisationsnachrichten ausgetauscht.

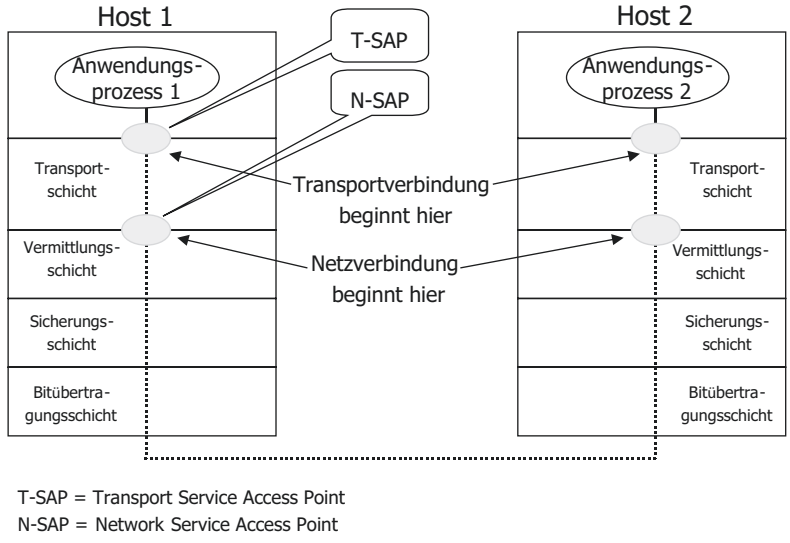


Abb. 2.2 Adressierung in der Transportschicht. (Nach Tanenbaum et al. 2021)

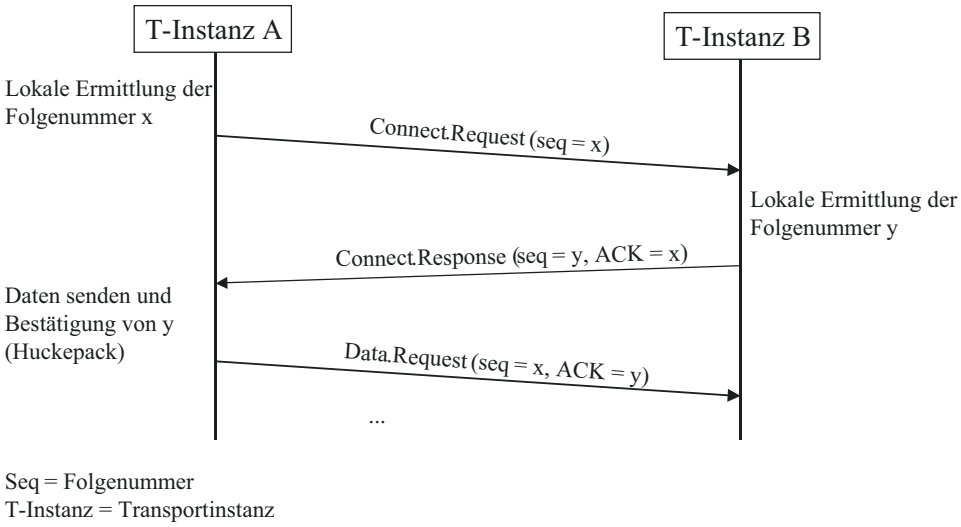


Abb. 2.3 Drei-Wege-Handshake-Protokoll für den normalen Verbindungsaufbau

Ein Verbindungsaufbau erfolgt meist über einen bestätigten Dienst, in dem Folgenummern vereinbart werden. Ein klassisches Verbindungsaufbauprotokoll ist das Drei-Wege-Handshake-Protokoll, das in Abb. 2.3 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.
- 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. 2.4 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

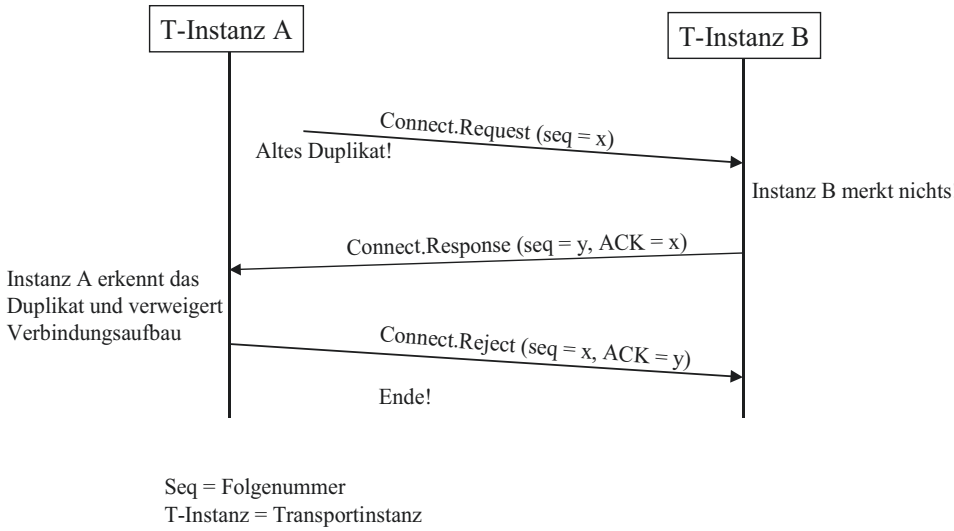


Abb. 2.4 Verbindungsaufbau, Fehlerfall 1: Connect-Request-Duplikat taucht auf

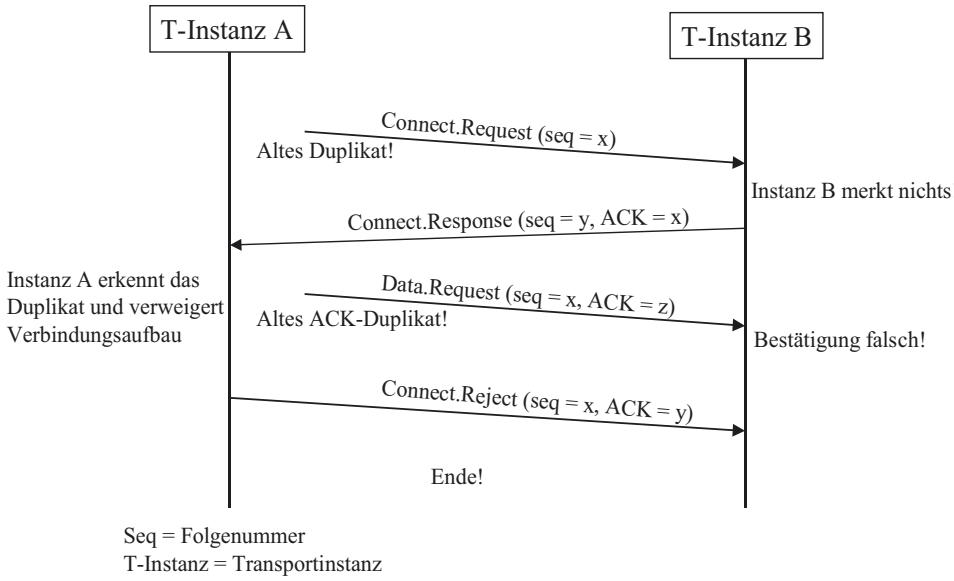


Abb. 2.5 Verbindungsaufbau, Fehlerfall 2: Connect-Request- und ACK-Duplikate tauchen auf

es sich um eine 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 (Abb. 2.5).

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 vor Erhalt der Disconnect-Request-PDU noch eine Nachricht sendet. Diese Nachricht ist dann verloren (Datenverlust). Daher ist ein anspruchsvolles Verbindungsabbauprotokoll notwendig. Auch hier verwendet man gerne einen Drei-Wege-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. 2.6). 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 Drei-Wege-Handshake jederzeit ein Disconnect-Request oder eine Bestätigung verloren gehen kann. 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 irgend-

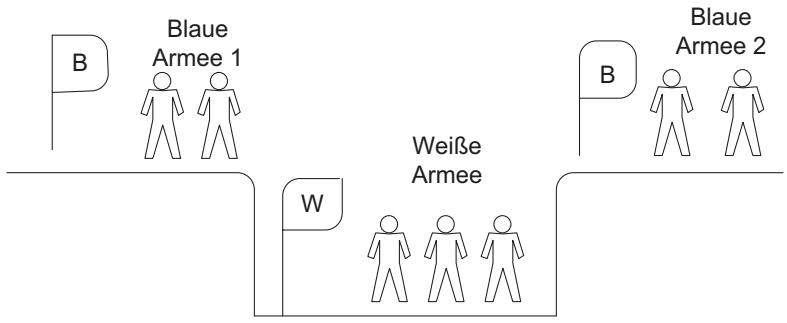


Abb. 2.6 Das Zwei-Armeen-Problem

[1] Wir gehen hier von einer Vollduplexverbindung aus.

wann wieder ab. Im Weiteren sind einige Szenarien für die Timerüberwachung beim Verbindungsabbau skizziert, die in einem verbindungsorientierten Transportprotokoll behandelt werden sollten.

Abb. 2.7 stellt den normalen Verbindungsabbau als Drei-Wege-Handshake dar. In Abb. 2.8 ist ein Szenario skizziert, in dem ein Timer abläuft. In diesem Fall trennt die Instanz B die Verbindung beim Ablauf des Timers.

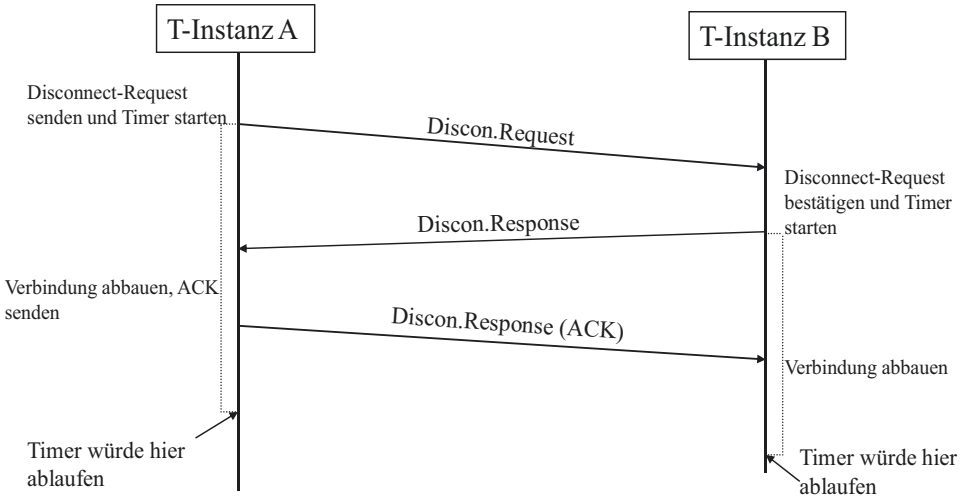


Abb. 2.7 Szenario beim Verbindungsabbau: Normaler Ablauf. (Nach Tanenbaum et al. 2021)

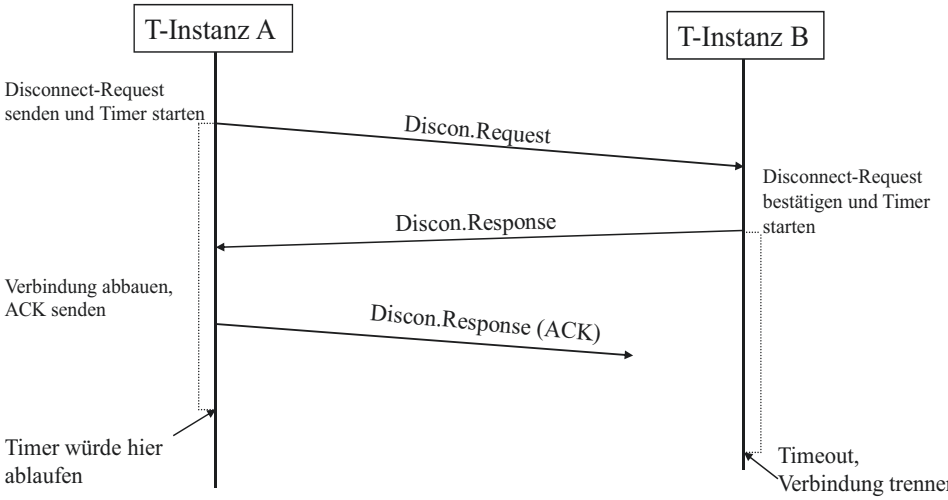


Abb. 2.8 Szenario beim Verbindungsabbau: Timerablauf

In Abb. 2.9 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.

Schließlich gehen in dem Szenario, das in Abb. 2.10 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.

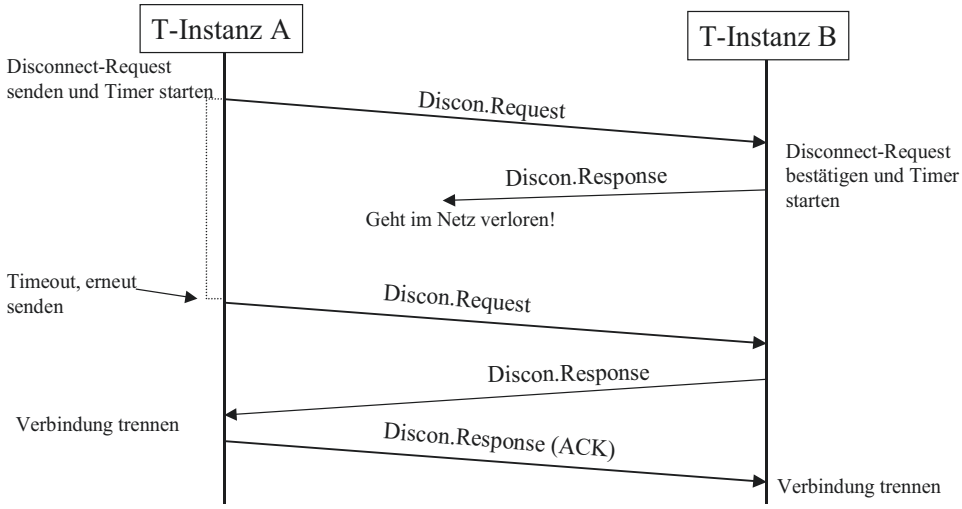


Abb. 2.9 Szenario beim Verbindungsabbau: Disconnect-Responses gehen verloren

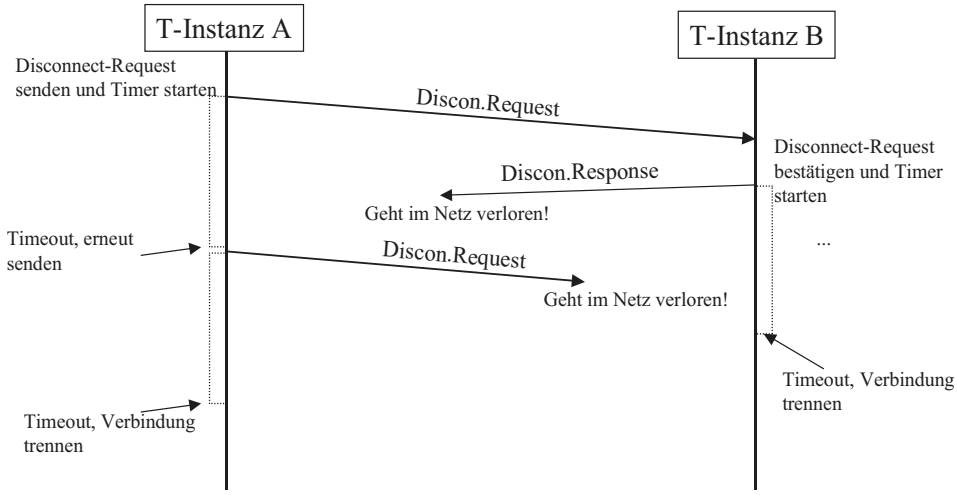


Abb. 2.10 Szenario beim Verbindungsabbau: Zwei Disconnect-PDUs gehen verloren

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 Dienste-schnittstelle nach oben weitergemeldet. 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 verlustbehafteteren 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, 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. Aufwendigere Verfahren ermöglichen das Senden mehrerer Daten und sammeln die Bestätigungen ggf. kumuliert ein. Hier spricht man auch von *Pipelining*.

Man unterscheidet folgende Quittungsverfahren:

- Beim *positiv-selektiven Quittierungsverfahren* wird vom Empfänger eine Quittung (ACK) pro empfangener Nachricht gesendet. Dies hat einen hohen zusätzlichen Nachrichtenverkehr zur Folge. Abb. 2.11 zeigt dieses Verfahren.

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

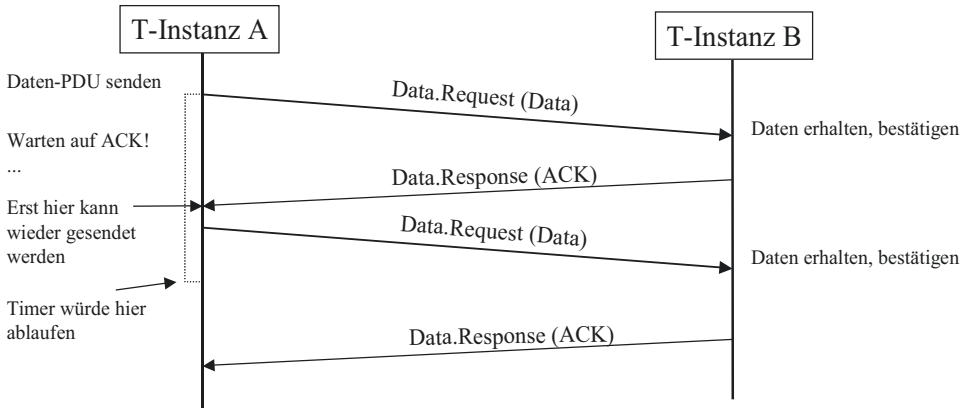


Abb. 2.11 Positiv-selektives Quittierungsverfahren

- Beim *positiv-kumulativen Quittierungsverfahren* wird eine Quittung für mehrere Nachrichten verwendet. Diese Reduzierung der Netzlast hat aber den Nachteil, dass Information über einen Datenverlust verspätet an den Sender übertragen wird.
- Beim *negativ-selektiven Quittierungsverfahren* 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.
- In Hochleistungsnetzen verwendet man eine *Kombination der Verfahren*, z. B. das positiv-kumulative Verfahren kombiniert mit dem negativ-selektiven Verfahren.

Das in Abb. 2.11 dargestellte positiv-selektive Quittierungsverfahren wird hier kombiniert mit einem Stop-and-Wait-Verfahren. Eine neue Nachricht kann in diesem Fall erst gesendet werden, wenn die Bestätigung zur vorherigen Nachricht angekommen ist. Beim positiv-selektiven Quittierungsverfahren muss dies nicht so sein, denn es können durchaus auch mehrere Nachrichten versendet werden. Entscheidend ist, dass für jede Nachricht eine Bestätigung empfangen wird.

Das positiv-kumulative Verfahren ist in Abb. 2.12 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. 2.13 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.

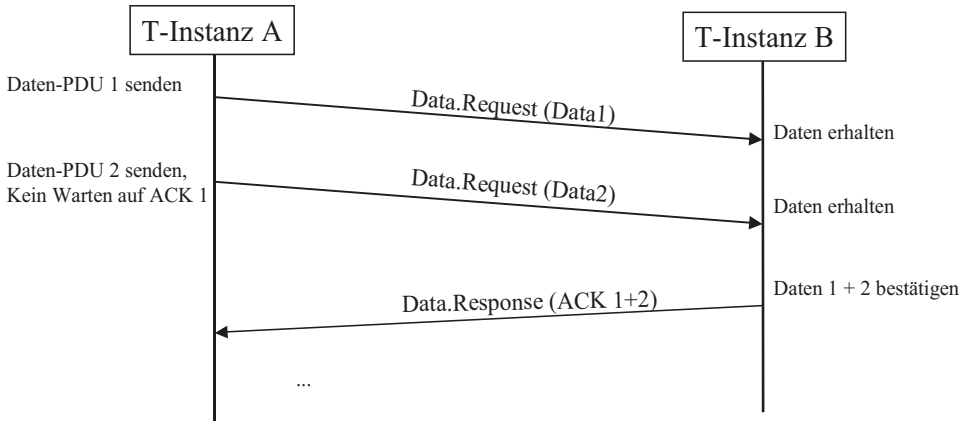


Abb. 2.12 Positiv-kumulatives Quittungsverfahren

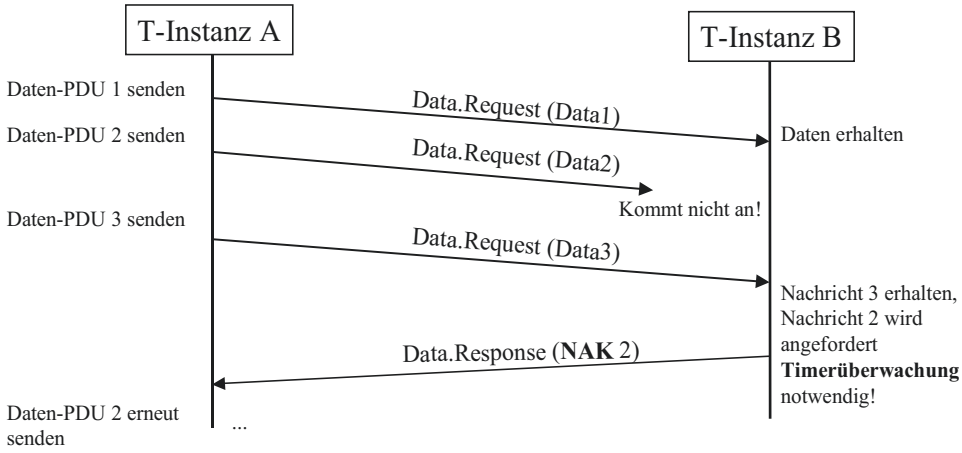


Abb. 2.13 Negativ-selektives Quittungsverfahren

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 wird. Die Übertragungswiederholung ist dabei auf zwei Arten üblich:

- Beim *selektiven Verfahren* werden nur Nachrichten wiederholt, für die keine Bestätigung empfangen oder – im negativ-selektiven Quittierungsverfahren – vom Empfänger eine Übertragungswiederholung angefordert wurde. 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 erforder-

lich 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, 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 kann eine Nachricht erst verwerfen, wenn er sicher sein kann, dass sie beim Empfänger angekommen ist. Beim positiven Quittierungsverfahren ist das unproblematisch. Sobald eine ACK-PDU empfangen wird, kann die Nachricht aus dem Sendepuffer verworfen werden. Schwieriger ist es beim negativ-selektiven Verfahren, da es hier für den Sender nicht so einfach festzustellen ist, 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. Sinnvoll ist es, wenn der Empfänger zumindest von Zeit zu Zeit auch aktiv eine Bestätigung sendet, in der angezeigt wird, welche Nachrichten schon empfangen wurden. Diese Information kann der Sender nutzen, um seinen Sendepuffer zu bereinigen.

In Abb. 2.14 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 Nummer 2 und die ebenfalls bereits gesendete (aber angekommene) PDU mit Folgenummer 3 erneut gesendet. Bei Netzen mit geringer Pfadkapazität ist mit Go-Back-N

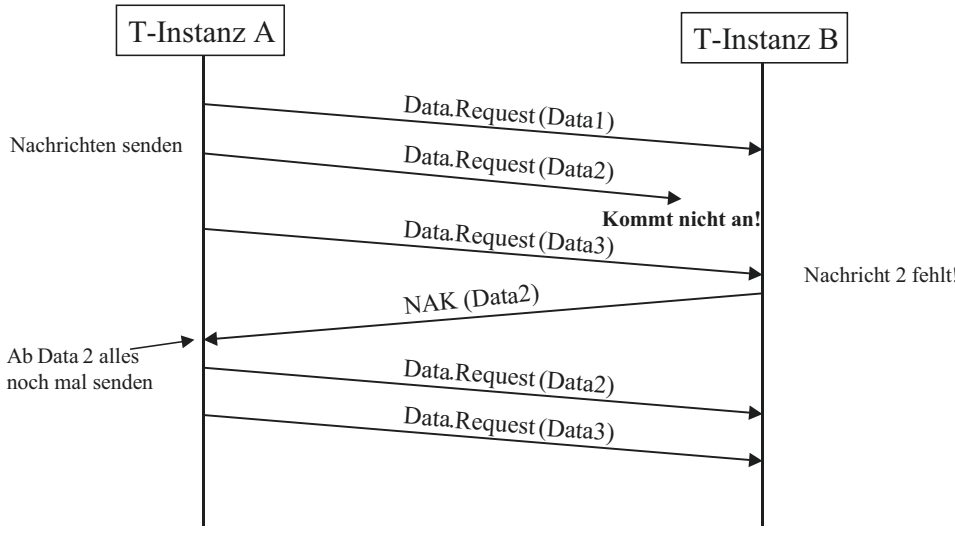


Abb. 2.14 Negativ-selektive Quittung und Go-Back-N-Verfahren

die Netzbelastung relativ gering. Anders dagegen ist dies in Netzen mit hoher Pfadkapazität, da bei entsprechender Fenstergröße (Abschn. 2.4) viele PDUs gesendet werden können, bevor die erste Negativquittung beim Sender eintrifft. Dies führt dann ggf. zur unnötigen Übertragungswiederholung vieler PDUs.

▶ **[Pfadkapazität]** Unter 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.

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.

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 ggf. 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 Flusskontrollverfahren sind:

- Das *Stop-and-Wait-Verfahren:* 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-Verfahren möglich ist.

Für die fensterbasierte Flusskontrolle werden in den Transportinstanzen für jeden Kommunikationspartner vier Folgennummernintervalle verwaltet, die grob in Abb. 2.15 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

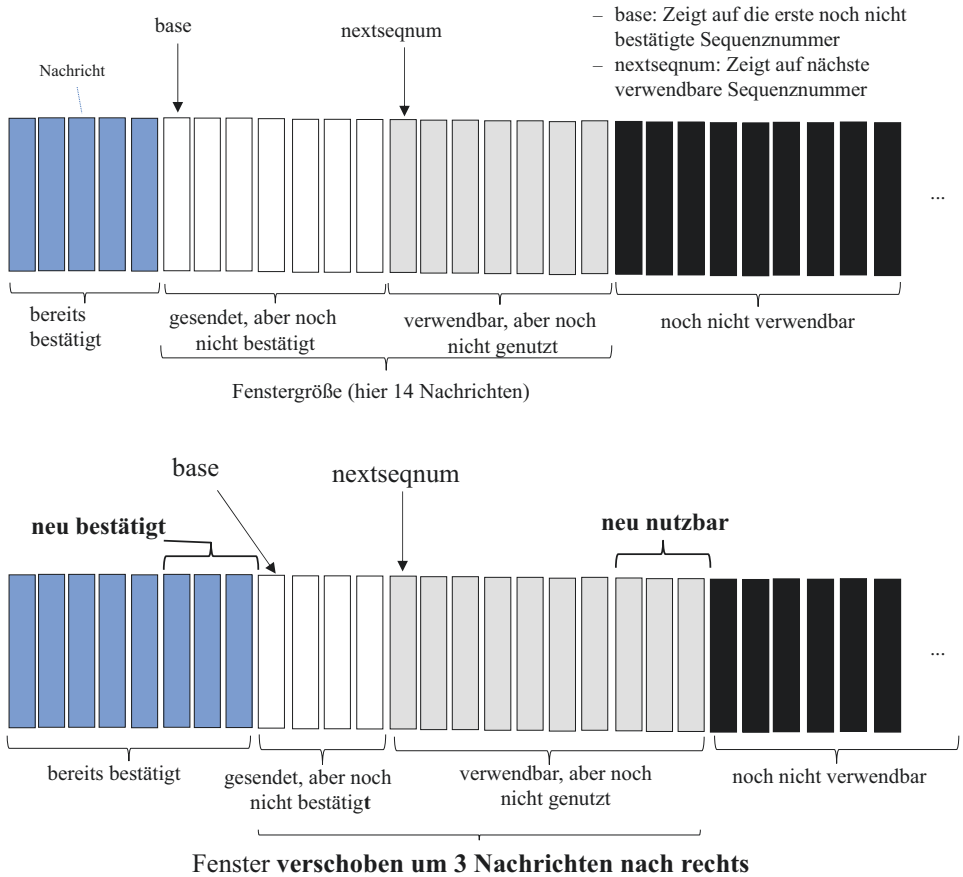
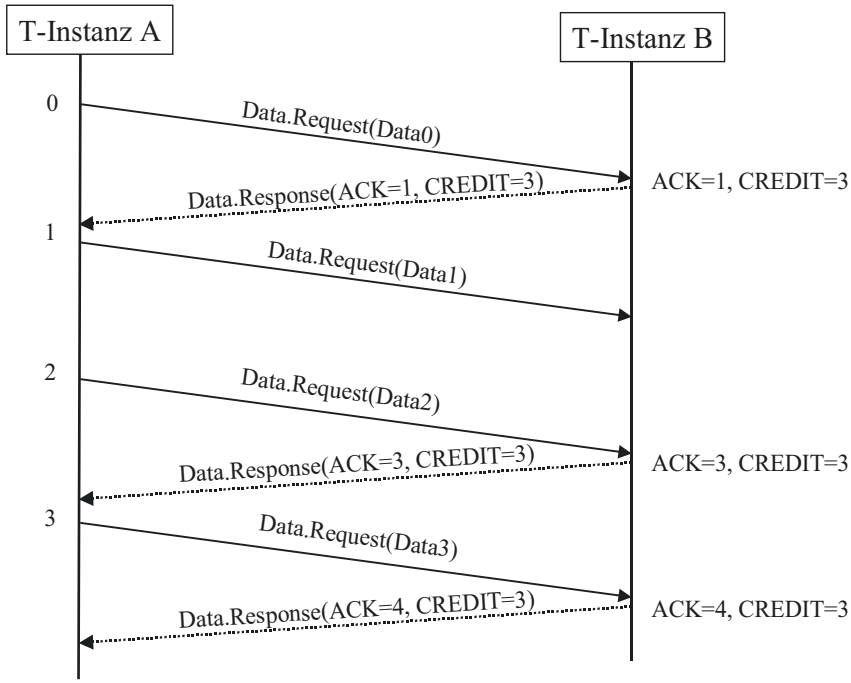


Abb. 2.15 Intervallverwaltung für die fensterbasierte Flusskontrolle

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 er entsprechend der Anzahl an benutzten Folgenummern nach rechts verschoben. In Abb. 2.15 ist auch die Situation nach drei empfangenen Bestätigungen dargestellt.

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. 2.16 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, sodass es zu keinen Wartezeiten im Sender kommt.



Anfänglicher Sendekredit ist 3
Kumulative Quittierung
Kontinuierliches Senden von Daten möglich, da immer Sendekredit verfügbar ist

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

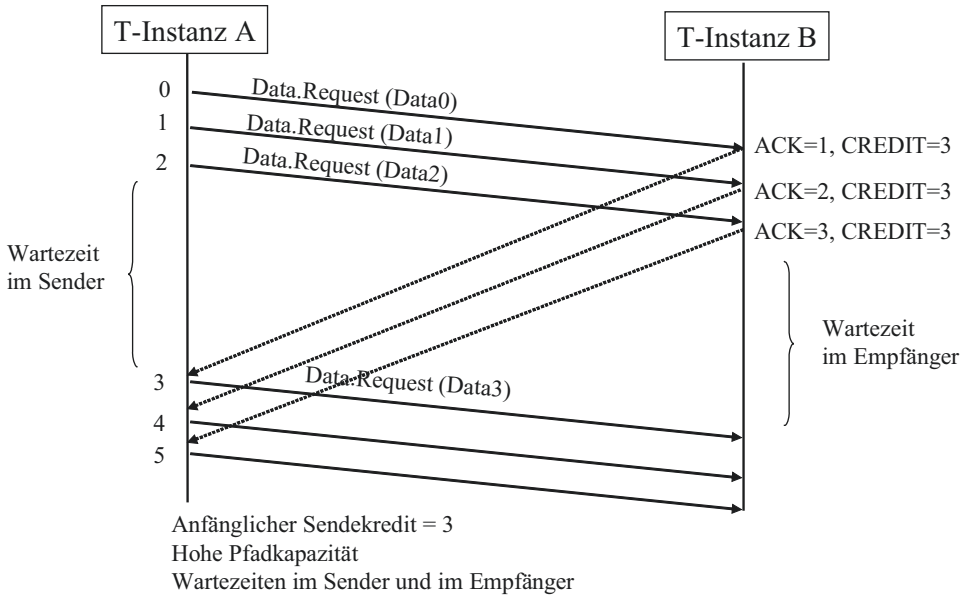


Abb. 2.17 Fensterbasierte Flusskontrolle bei hoher Pfadkapazität. (Nach Zitterbart und Braun 1996)

Bei Netzwerken mit hoher Pfadkapazität führen kleine Sendekredite häufig dazu, dass der Sender durch das Warten auf Kreditbestätigungen blockiert wird. Dies ist in Abb. 2.17 skizziert. Bei einem Sendekredit von drei Nachrichten entstehen beträchtliche Wartezeiten 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 Internetprotokoll) nicht.

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

In der Transportschicht hatte man ursprünglich in der TCP/IP-Protokollfamilie 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 (Abschn. 3.6.6). 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.

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 mithilfe der Adressinformation.

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.; Feamster, N. Wetherall, D. J. (2021) Computer Networks, Sixth Edition, Pearson Education Limited, 2021

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 wird ständig optimiert und weiterentwickelt und es gibt mittlerweile eine Fülle von RFCs, die sich mit TCP befassen. Die wichtigsten Protokollmechanismen von TCP werden in diesem Kapitel detailliert erläutert. Hierzu gehört der Verbindungsaufbau, der Verbindungsabbau und die gesicherte Datenübertragung. Die in TCP verwendeten Bestätigungs- und Wiederholungsverfahren und die Maßnahmen zur Flusskontrolle werden ebenso betrachtet wie der in der TCP-Spezifikation erläuterte Zustandsautomat für den Verbindungsauf- und -abbau im Zusammenspiel mit dem Nachrichtenaustausch. Weiterführende Konzepte und Mechanismen wie die TCP-Überlastkontrolle (Slow-Start, TCP Reno, TCP NewReno, TCP BIC, TCP CUBIC) und wichtige TCP-Protokolltimer werden diskutiert. Auch auf Sicherheitsaspekte von TCP wird in diesem Kapitel eingegangen.

Ergänzende Information Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann [https://doi.org/10.1007/978-3-658-43988-0_3].

P. Mandl, *TCP, UDP und QUIC Internals*, https://doi.org/10.1007/978-3-658-43988-0_3

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, 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[1] zu finden.

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 denselben 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 Protocol (IP). TCP garantiert weiterhin die Reihenfolge der Nachrichten und die vollständige Auslieferung, übernimmt

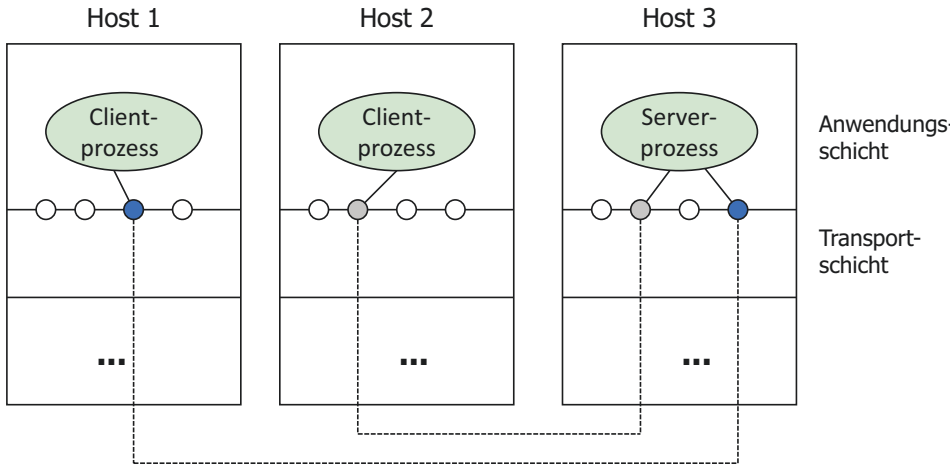


Abb. 3.1 Ende-zu-Ende-Kommunikation in TCP

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

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.

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-Window-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
- 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-

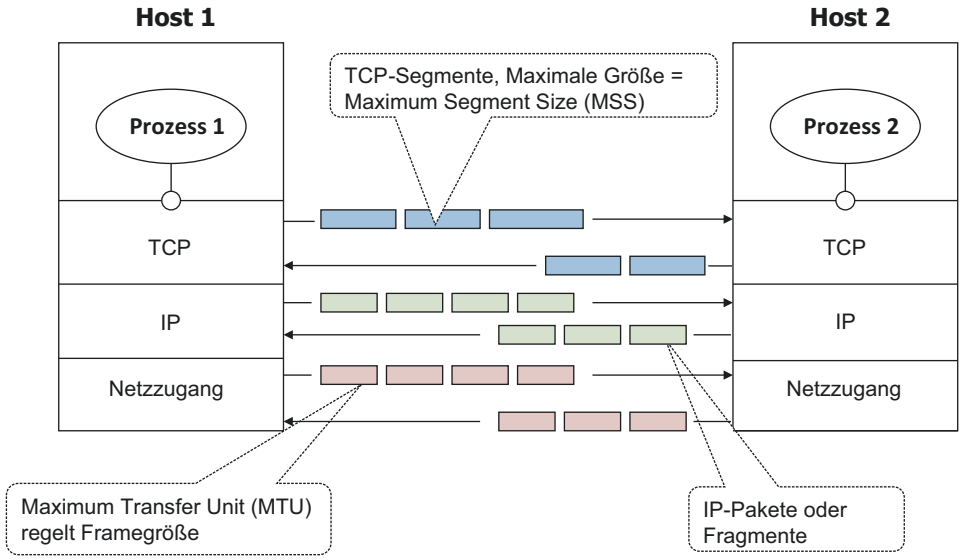


Abb. 3.2 TCP-Segmente in der Schichtenzuordnung

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 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]

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

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 ggf. auch von weiteren Protokollen wie PPP und PPPoE (bei DSL-Netzzugängen). Daraus lässt sich für IPv4-Netze Folgendes ableiten, sofern nichts über die einzelnen Teilnetze zwischen Sender und Empfänger bekannt ist:

$$\text{MTU} \geq \text{MSS} + \text{TCPHeaderL a nge} + \text{IPv4HeaderL a nge} + \text{L a nge weiterer Header}$$

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:

$$576 \geq \text{MSS} + 20 + 20$$

$$\text{MSS} \leq 576 - 40$$

$$\text{MSS} \leq 536$$

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 aufwendige 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 ggf. wiederholt werden. PMTUD wird von TCP (und auch von UDP) unterstützt und auch kontinuierlich für jede Verbindung durchgeführt, da sich Routen 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, sodass 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:

$$\text{MTUEthernet} \geq \text{MSS} + \text{TCPHeaderLänge} + \text{IPvHeaderLänge}$$

$$\text{MSS} \leq 1500 - 40 \rightarrow \text{MSS} \leq 1460$$

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 Framelänge 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 Socket-Adresse 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 (Internet Assigned Numbers Authority) verwaltet die Ports und definiert

- sogenannte wohlbekannte Ports (well-known 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 wohlbekannten 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.:[3]

- 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)
- 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 Server 1 einen Dienst mit der Portnummer 52000 anbieten und 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.

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

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 zehn 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 Bytes langen TCP-Header, sofern das Nutzdatenteil leer ist.

Der in der Regel 20 Bytes 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,

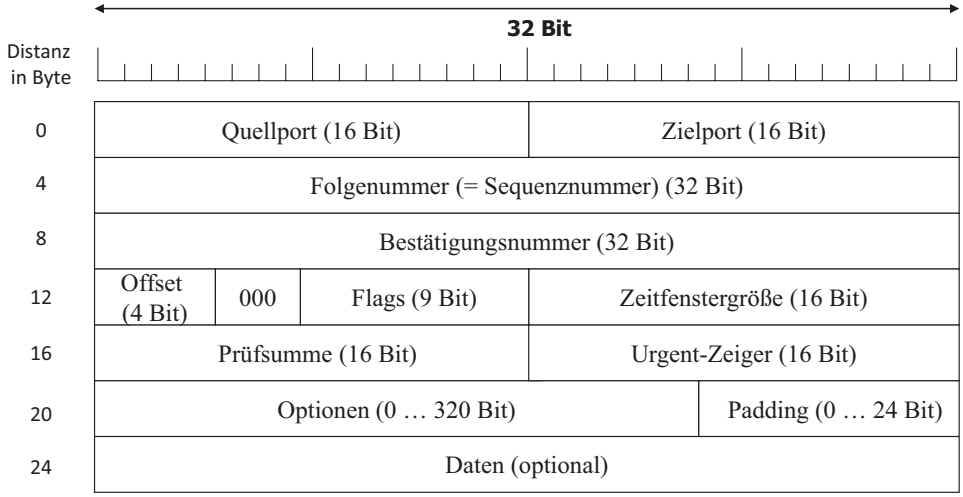


Abb. 3.3 TCP-Protokoll-Header

was dazu führt, dass der Header auch länger als 20 Bytes sein kann. Die Header-Länge 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.

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

- *Quell- und Zielport (jeweils 16 Bit):* Portnummer des Anwendungsprogramms des Senders (Quelle) und des Empfängers (Ziel).
- *Folgenummer (32 Bit):* Nächstes Byte innerhalb des TCP-Streams, das der Sender absendet.
- *Bestätigungsnummer (32 Bit):* Gibt das als Nächstes erwartete Byte im TCP-Stream des Partners an und bestätigt damit den Empfang der vorhergehenden Bytes.
- *Offset:* Gibt mit 4 Bit die Länge des TCP-Headers in 32-Bit-Worten an. Dies ist nötig, da der Header aufgrund des variablen Optionenfeldes keine fixe Länge aufweist.
- *000:* 3 Bit sind noch reserviert und haben derzeit noch keine Verwendung.
- *Flag (9 Bit):* Steuerkennzeichen für diverse Aufgaben, die im Einzelnen noch diskutiert werden.
- *Zeitfenstergröße (16 Bit):* 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 (16 Bit):* 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 (16 Bit):* 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 (0 bis 320 Bit):* Optionale Angaben zum Aushandeln bestimmter Verbindungsparameter, die noch im Einzelnen diskutiert werden.
- *Padding (maximal 24 Bit):* 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 in Abschn. 3.3.4 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.[4]

Die Flags werden bei der Erläuterung der Protokollfunktionen noch intensiv in den Abschn. 3.2 und 3.3 betrachtet. SYN, ACK, FIN und RST sind für die Verbindungsverwaltung von Interesse. ACK wird für das Bestätigungsverfahren genutzt.

Tab. 3.1 TCP-Flags

Flag (1 Bit)	Bedeutung
CWR	Dieses Flag wird für die explizite Staukontrolle (siehe Erläuterung zu Explicit Congestion Notification in Abschn. 3.6.6) genutzt
ECE	Dieses Flag wird für die explizite Staukontrolle (siehe Erläuterung zu Explicit Congestion Notification in Abschn. 3.6.6) 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	Dieses Flag wird gesetzt, um den Verbindungsaufbau anzuzeigen (siehe TCP-Verbindungsaufbau)
FIN	Dieses Flag wird benutzt, um eine Verbindung abzubauen (siehe TCP-Verbindungsabbau)

[4] 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.

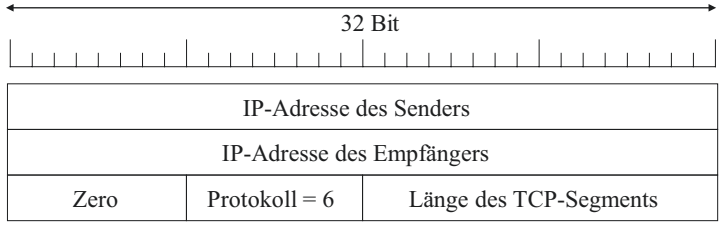


Abb. 3.4 TCP-Pseudoheader

Das PSH-Flag sollte eigentlich nur in Ausnahmefällen eingesetzt werden. Unsere Beobachtungen haben aber ergeben, dass es in heutigen TCP-Implementierungen häufig gesetzt wird, um das Ende einer logisch zusammenhängenden Nachricht anzuzeigen. Das ist nach jedem Sendeaufruf an der Socket-Schnittstelle der Fall. Damit wird bezweckt, dass alle Bytes eines Sendeaufrufs sofort, ohne Zwischenspeicherung, übertragen werden. Der Empfänger erhält damit bei Aufruf einer Receive-Operation logisch zusammengehörige Daten.

Das Feld *Prüfsumme* dient der Überprüfung des TCP-Headers und der Nutzdaten. Alle 16-Bit-Wörter werden nach einem Verfahren, das sowohl bei TCP als auch bei UDP verwendet wird, addiert und dann in ein Einerkomplement transferiert. Wir verweisen zur Bildung der Prüfsumme daher auf Kap. 4. 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 Protokoll-Headers 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 Drei-Wege-Handshake-Protokoll für den Verbindungsaufbau, wie es vereinfacht in Abb. 3.5 dargestellt ist.

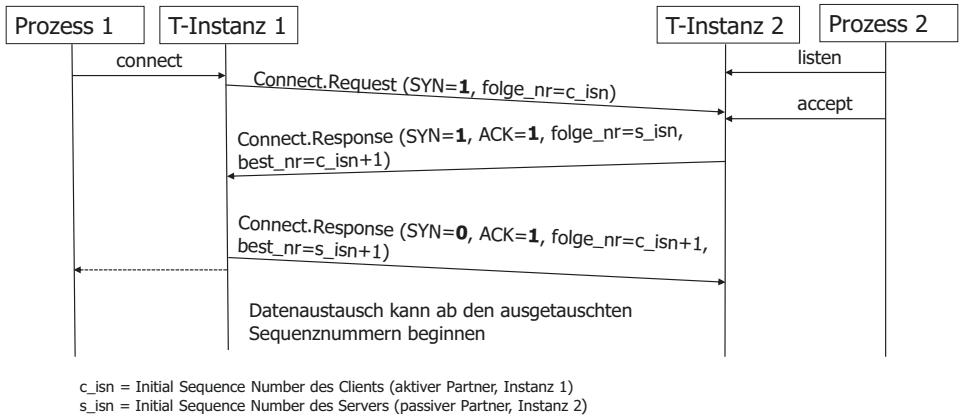


Abb. 3.5 Normaler Verbindungsaufbau in TCP

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 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*, *Folgenummer* und *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 in Abschn. 3.5.2 besprochen 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-Response-Folgen, wie dies etwa in HTTP der Fall ist, auf den Drei-Wege-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 Folgetransaktion (Request) handelt und deswegen kein Drei-Wege-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 Drei-Wege-Handshake-, bei genauerer Betrachtung sogar eine Vier-Wege-Handshake-Synchronisation.

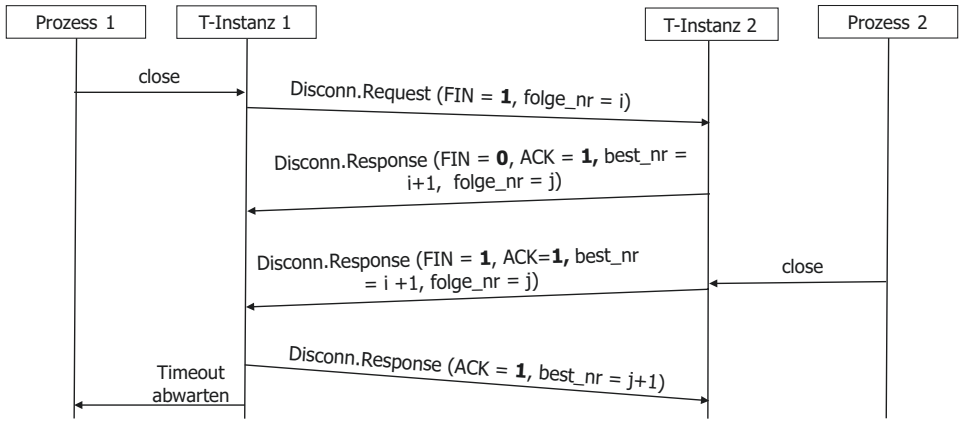


Abb. 3.6 TCP-Verbindungsabbau

Jede der beiden Verbindungsrichtungen der Vollduplexverbindung 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 Socket-Schnittstelle 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 in dem Abschn. 3.2.4 und 3.2.5 noch sehen werden, gibt es weitere 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 auf beiden Seiten ordnungsgemäß die Verbindung über die Socket-Schnittstelle mit einem *close*-Aufruf beenden. Diesen Aspekt betrachten wir in Kap. 7 noch genauer.

Eine Besonderheit des Verbindungsabbaus beim aktiven Partner ist, dass er nach allen Bestätigungen des gegenüberliegenden Partners die Verbindung noch nicht abschließt, sondern zunächst 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 in Abschn. 3.7 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 Machines) modelliert. Eine vollständige Spezifikation der Datenübertragungsphase als endlicher Zustandsautomat wurde nicht vorgenommen.

Tab. 3.2 zeigt die möglichen Zustände, die beim Verbindungsaufbau und Verbindungsabbau sowohl auf der passiven als auch 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.

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.

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

Zustand	Bedeutung
CLOSED	Keine Verbindung aktiv oder anstehend
LISTEN	Der passive TCP-Partner wartet auf 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

Zustandsautomaten in der Protokollspezifikation

Für die Spezifikation von Protokollen werden häufig deterministische endliche Automaten oder Finite State Machines (FSMs) 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-Automaten 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

$$\langle S, I, O, T, s_o, F \rangle$$

beschreiben, wobei gilt:

- S – endliche, nichtleere Menge von Zuständen
- I – endliche, nichtleere Menge von Eingaben
- O – endliche, nichtleere Menge von Ausgaben
- F – endliche, nichtleere Menge von Endzuständen, $F \subseteq S$
- $T \subseteq S \times (I \cup \{\tau\}) \times (O \cup \{\tau\}) \times S$ – eine Zustandsüberführungsfunktion; τ 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

$$\langle s, i, o, s' \rangle,$$

wobei

- $s \in S$ der aktuelle Zustand,
- $i \in (I \cup \{\tau\})$ eine Eingabe,
- $o \in (O \cup \{\tau\})$ 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* (EFSMs) oder *Communicating Extended Finite State Machines* (CEFSMs). 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 Clientseite kommuniziert eine Clientanwendung 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.

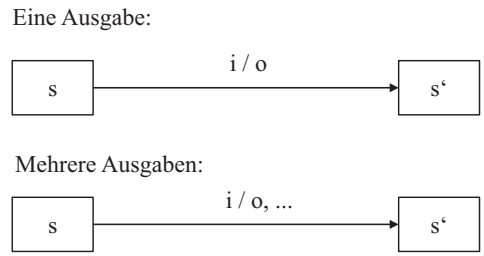


Abb. 3.7 Zustandsübergang von s nach s′ bei Eingabe i mit Ausgabe o

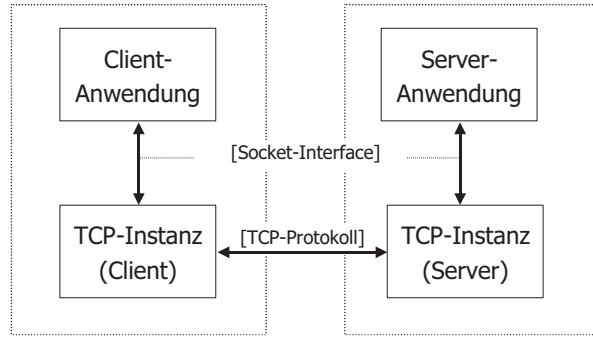


Abb. 3.8 Systemmodell aus Zustandsautomaten

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 konfigurierten Timers wird die Verbindung dann endgültig abgebaut und der Zustand CLOSED

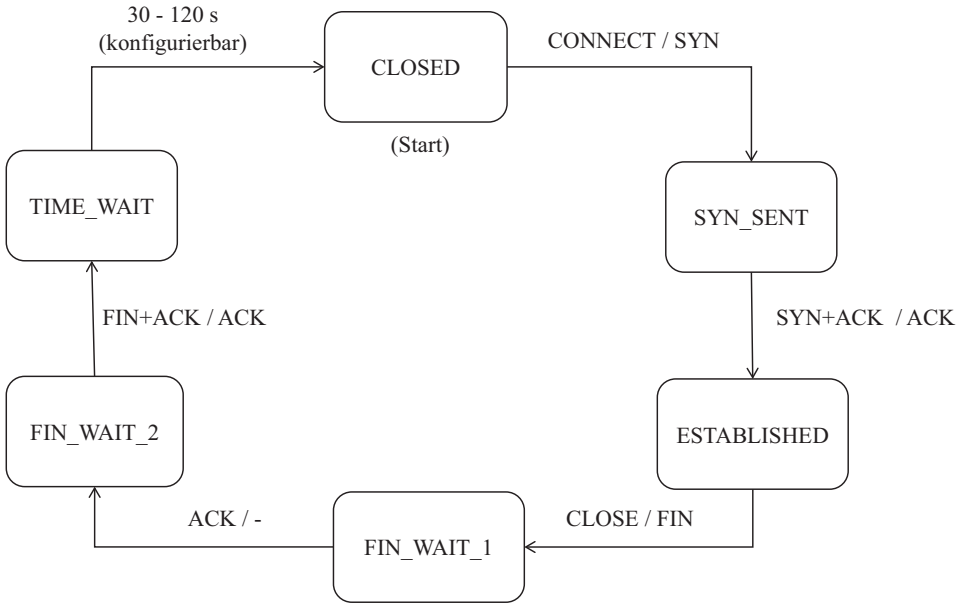


Abb. 3.9 TCP-Zustandsautomat des aktiven Partners

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.

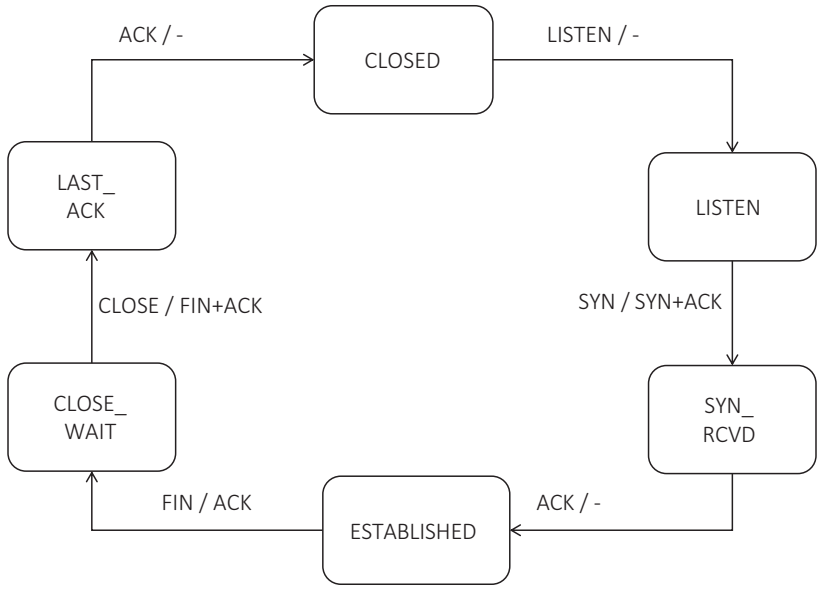


Abb. 3.10 TCP-Zustandsautomat des passiven Partners

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

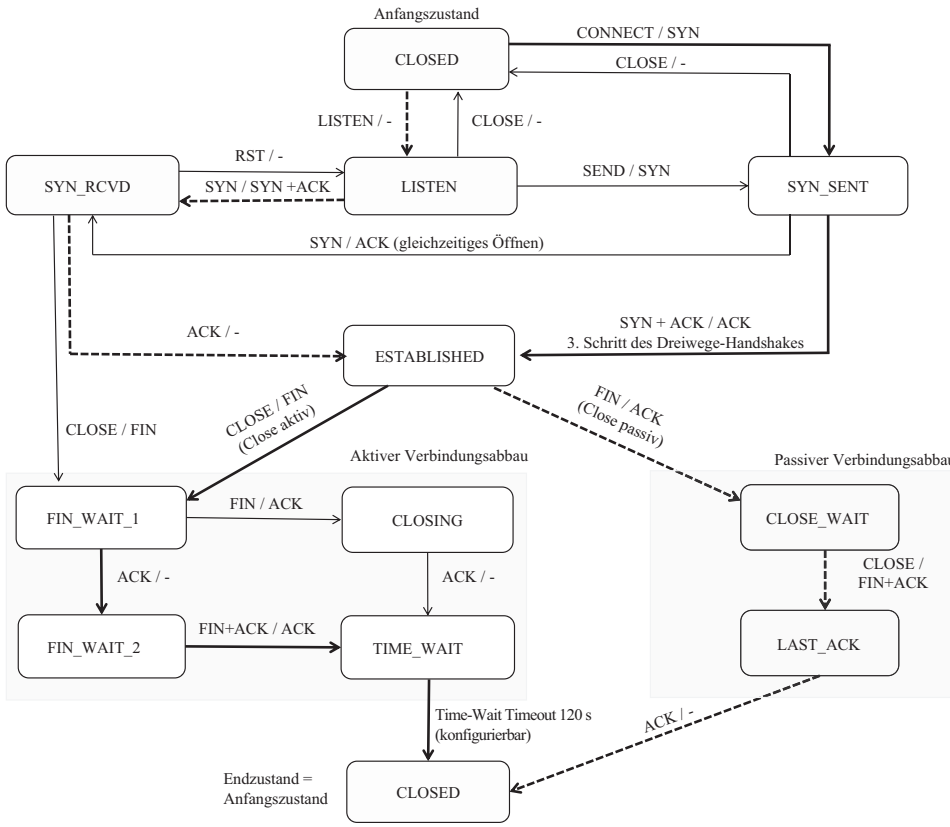


Abb. 3.11 Kompletter TCP-Protokollautomat nach RFC 793

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 Drei-Wege-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 Segments 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 Disconnect-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. In Abb. 3.10 sind neben dem standardmäßig vor-

gegebenen Timeout für den Zustand TIME_WAIT die drei Zustände FIN_WAIT2, CLOSE_WAIT und LAST_ACK markiert, bei denen in TCP-Implementierungen Timer sinnvoll, jedoch nicht in der TCP-Spezifikation vorgegeben sind.

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 werden im aktiven und im passiven Partner die Zustände FIN_WAIT_1, CLOSING, TIME_WAIT und CLOSED durchlaufen.

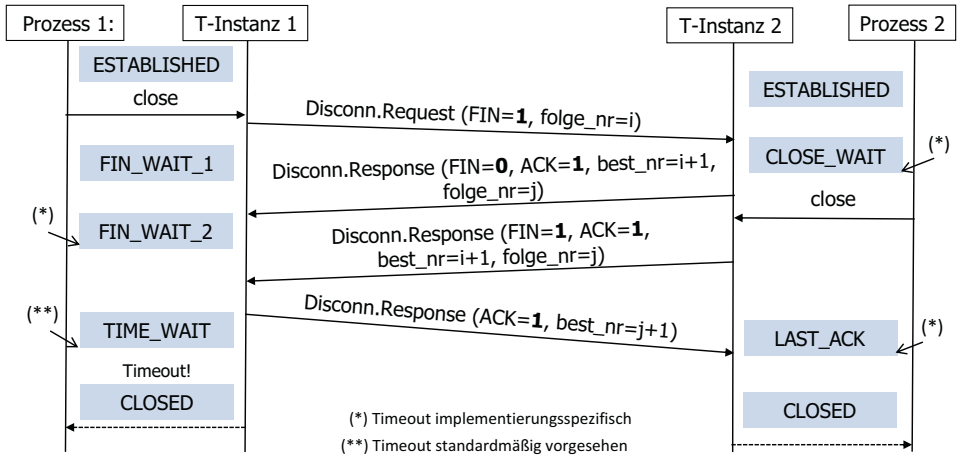


Abb. 3.12 TCP-Verbindungsabbau mit Zustandsübergängen

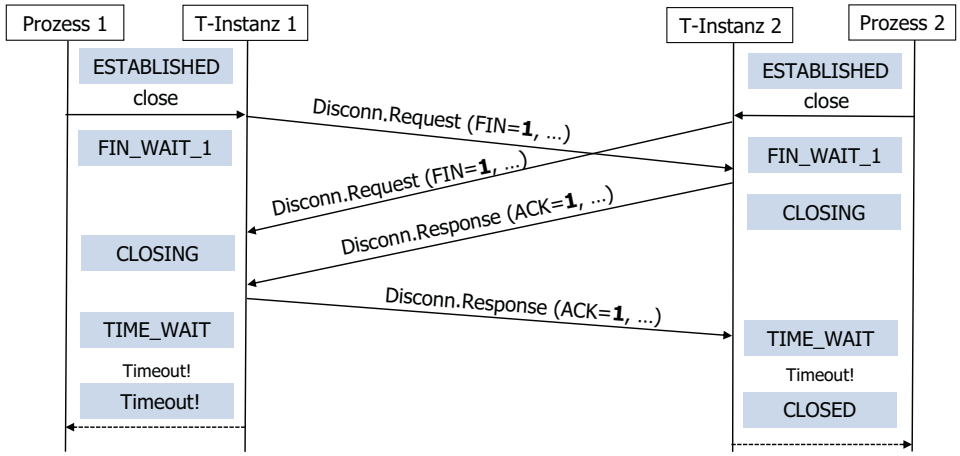


Abb. 3.13 Weitere Variante für TCP-Verbindungsabbau

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*[5] 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.

[5] Download aus www.sysinternals.com (zugegriffen am 18.07.2017).

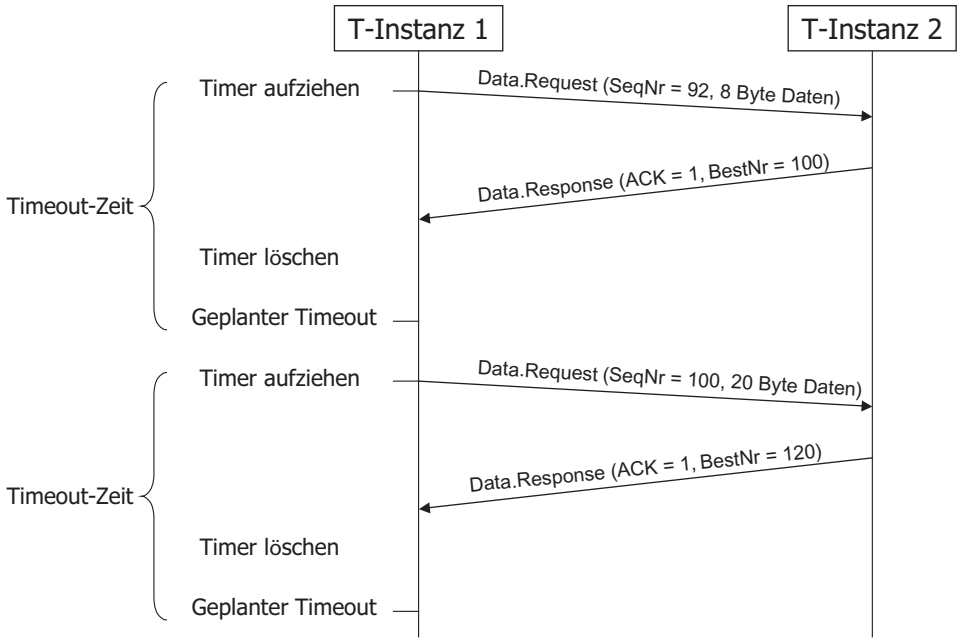


Abb. 3.14 Normale Übertragung eines TCP-Segments

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 Datenstromkonzepts 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.

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 Bytes 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

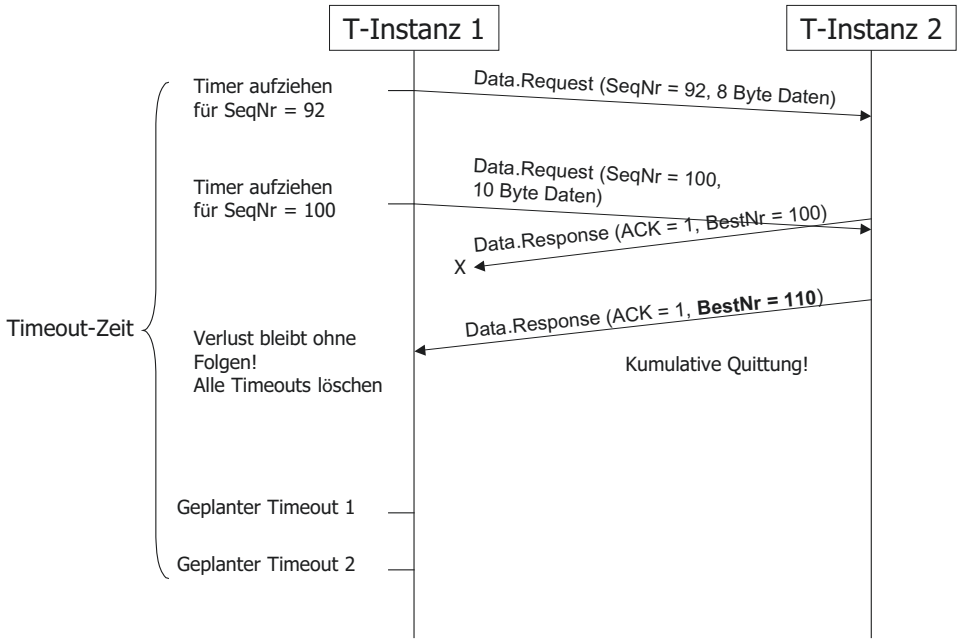


Abb. 3.15 Kumulative Quittierung in TCP

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 ggf. auch verloren gegangene ACK-PDUs durch folgende ACK-PDUs erledigt werden. Dies ist z. B. in Abb. 3.15 deutlich zu sehen. In diesem Sequenzdiagramm geht eine ACK-PDU verloren und wird durch die folgende kompensiert.

In den RFCs 1122 und 2581 werden einige Implementierungsempfehlungen 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 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 Abschn. 3.7.2 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.

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.

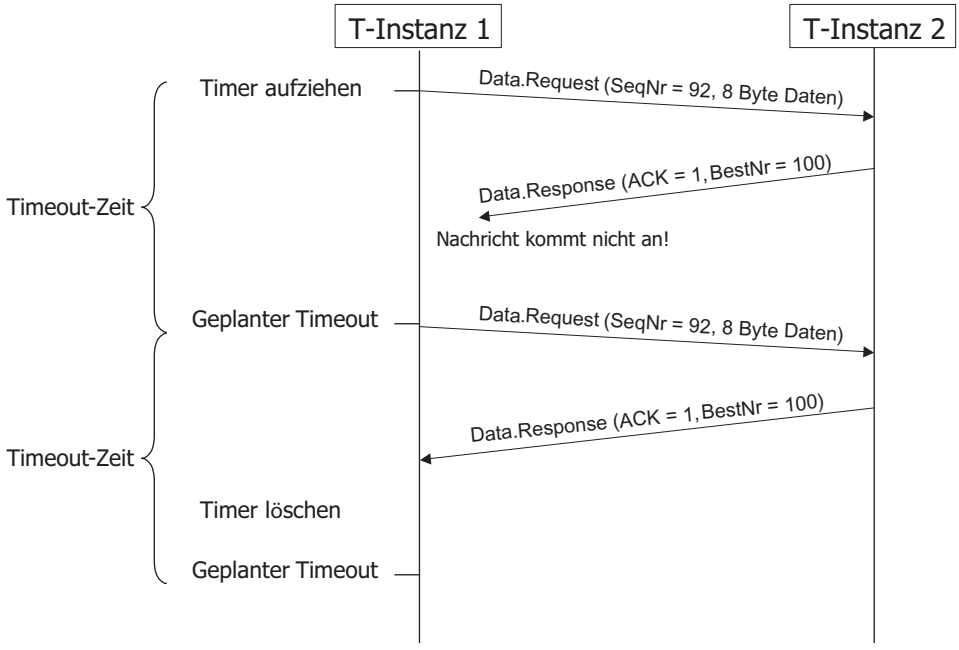


Abb. 3.16 Timerablauf bei der Übertragung eines TCP-Segments

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),[6] 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 TCP-Instanz 2 erneut eine ACK-Duplikat-PDU, in der Abbildung als *Data.Response(ACK=1, BestNr=100)* gekennzeichnet. Nach dem Empfang des erneut gesendeten Segments bestätigt die TCP-Instanz 2 alle nachfolgenden und in der Zwischenzeit gespeicherten Segmente, was in der Abbildung über die Bestätigung *Data.Response(ACK=1, BestNr=180)* angedeutet ist.

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).

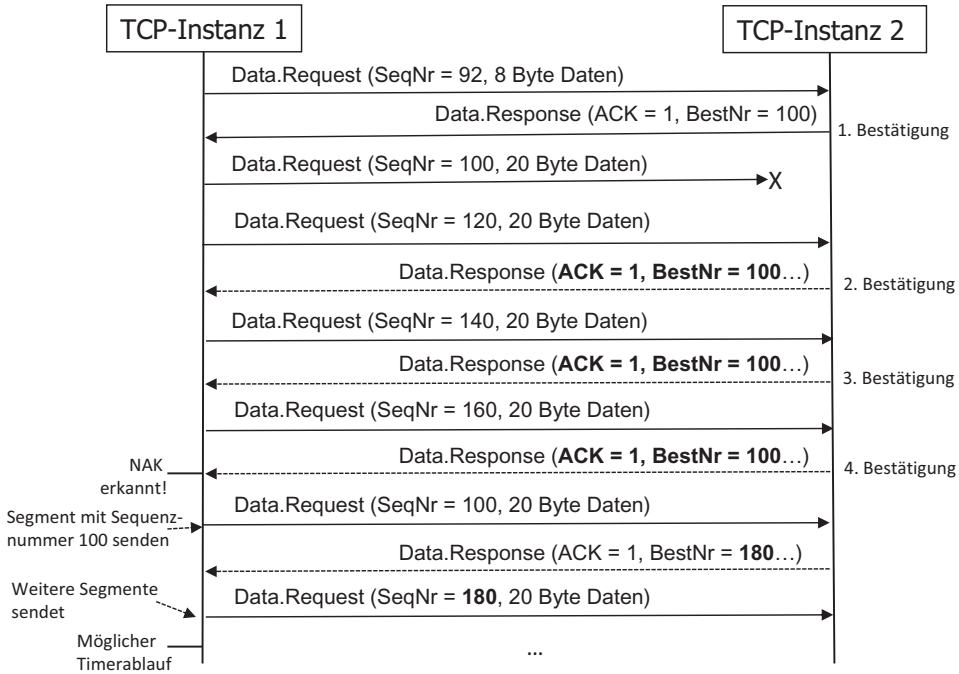


Abb. 3.17 Empfänger sendet implizites NAK

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

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 Stream-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

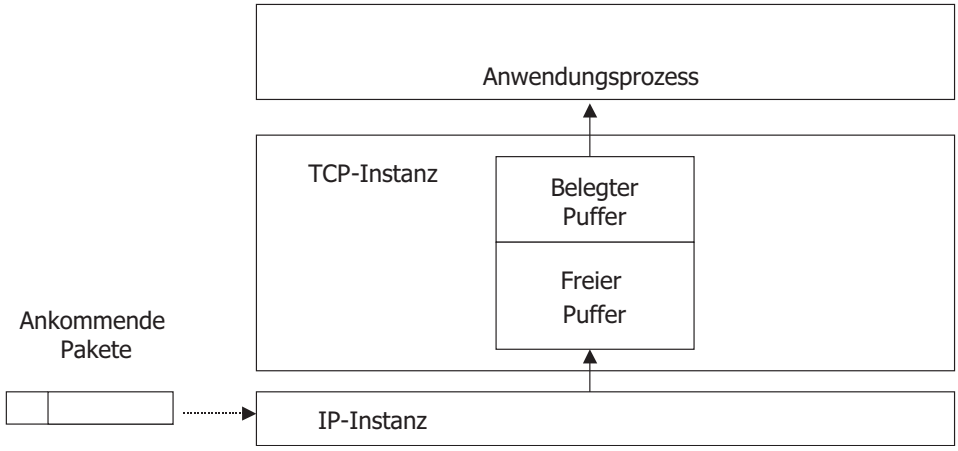


Abb. 3.18 Empfangspuffer für TCP-Verbindungen

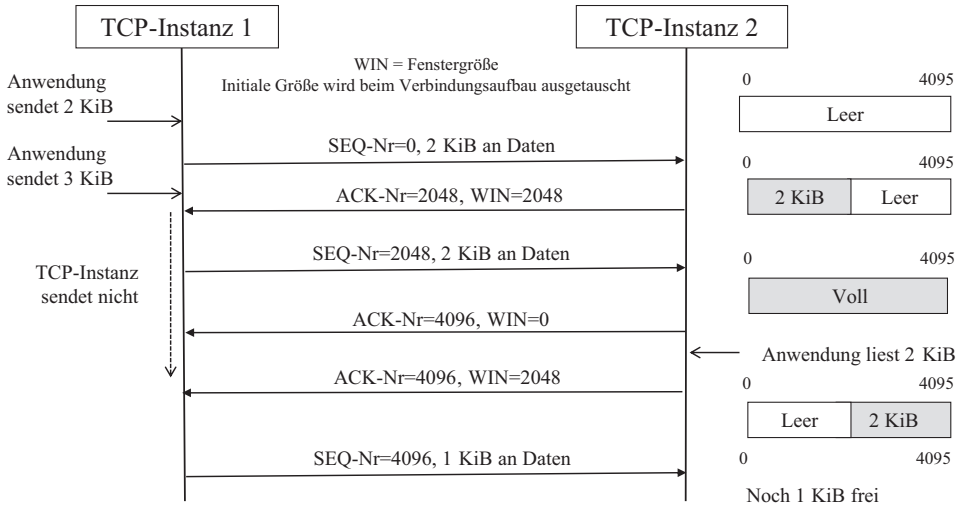


Abb. 3.19 Sliding-Window-Mechanismus bei TCP nach. (Tanenbaum et al. 2021)

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.

Die Flusskontrollmechanismen wurden mehrmals optimiert, und es gibt in der TCP-Dokumentation eine Reihe von RFCs mit Vorschlägen für Optimierungsalgorithmen. 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 Fenstergröße (siehe WIN-Parameter) im Verlauf der Kommunikation bei beiden ab.

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 zu-

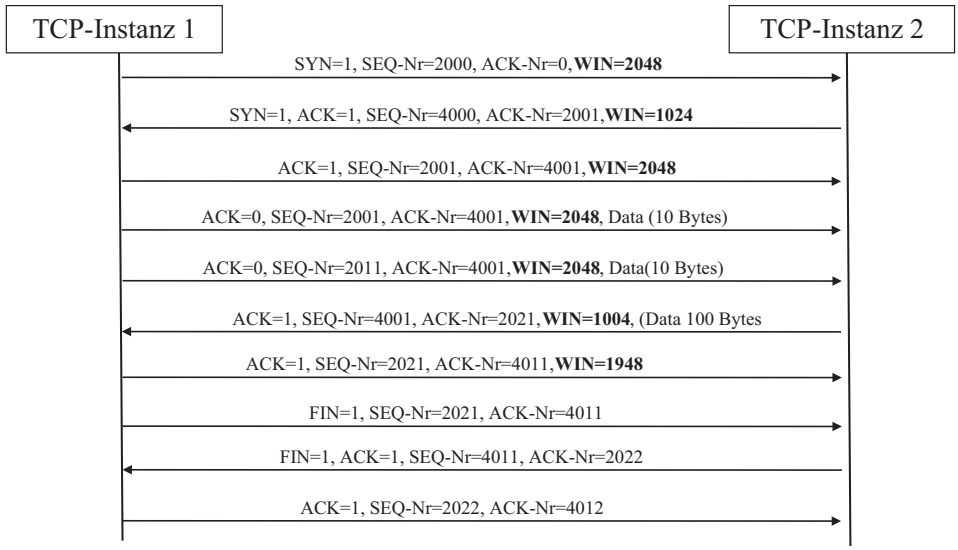


Abb. 3.20 Beispielablauf für Sliding-Window-Mechanismus bei TCP

sammengefasst. 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; erst dann wird 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 typische Ablauf bei Anwendung des Nagle-Algorithmus ist aus Sendersicht in Abb. 3.21 skizziert. Prozess 1 sendet in dem Beispiel immer einzelne Bytes. Die zuständige TCP-Instanz 1 sendet zunächst eine kleine Nachricht, sammelt dann die weiteren Bytes im Sendepuffer und sendet erst wieder, wenn die MSS erreicht ist. Bei Empfang einer ACK-PDU wird ebenfalls das Senden einer weiteren Data-PDU veranlasst.

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

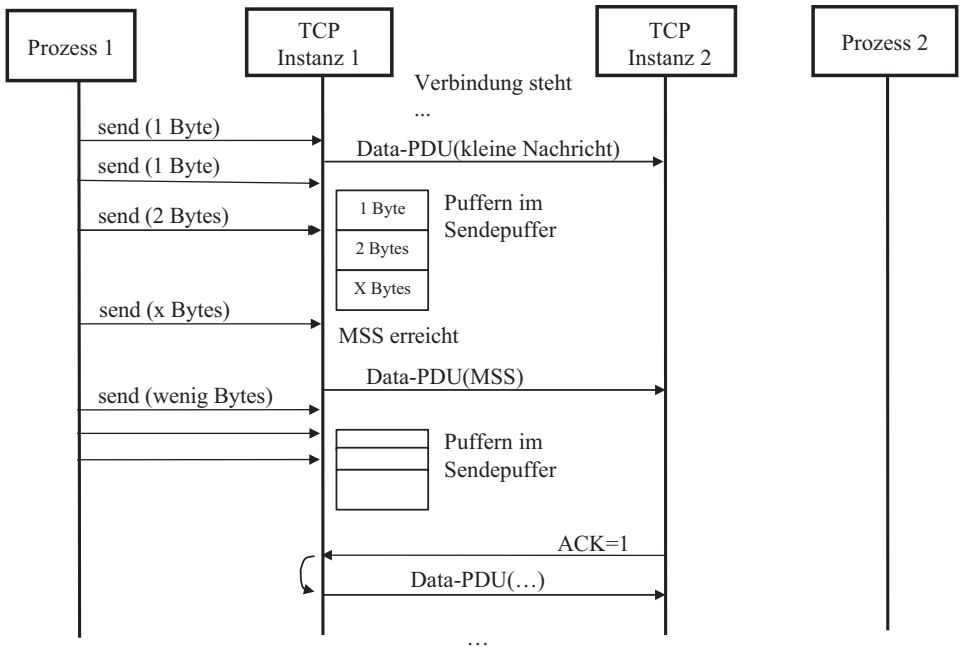


Abb. 3.21 Typischer Ablauf mit Nagle-Algorithmus aus Sendersicht

```
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 Echtzeitanwendungen, die schnelle Reaktionen erfordern, ist Nagle nur bedingt geeignet.

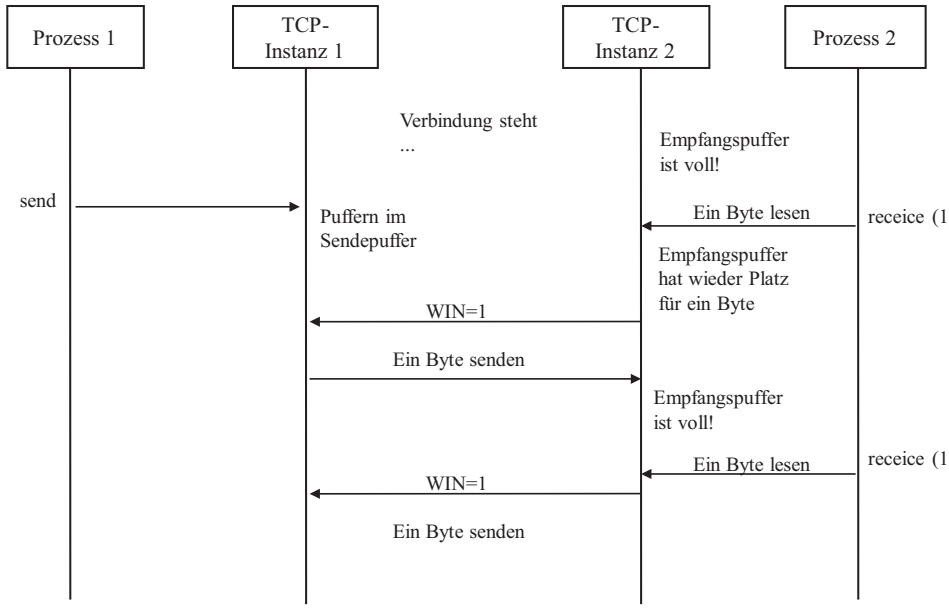


Abb. 3.22 Silly-Window-Syndrom

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 in Kap. 7) abschalten (IEEE POSIX 2017). 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 geleert, und die TCP-Instanz beim Empfänger sendet daraufhin eine ACK-PDU mit dem Hinweis, dass wieder ein Byte übertragen werden kann. In Abb. 3.22 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 Sendeinstanz 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 ist mit einer Typangabe, einem Längenfeld und den eigentlichen Optionsdaten gekennzeichnet (Abb. 3.23).

Abb. 3.24 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 u. a. 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 536+20 Bytes 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 Bytes lang.

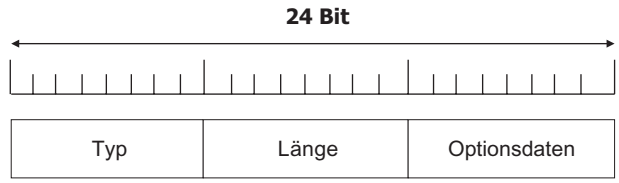


Abb. 3.23 Grundlegender Aufbau des TCP-Optionsfeldes

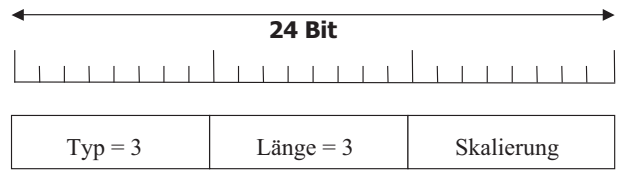


Abb. 3.24 Aufbau der WSOPT-Option

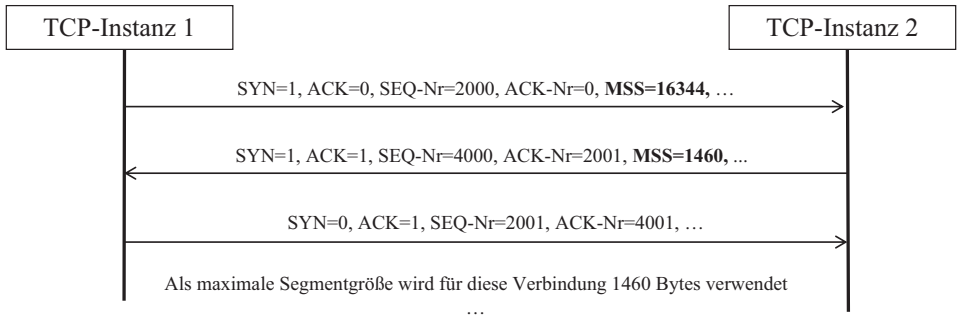


Abb. 3.25 MSS-Option beim Verbindungsaufbau vereinbaren

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 aufwendige 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. 3.25 ist eine typische MSS-Vereinbarung beim Verbindungsaufbau skizziert.

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-Verzögerungs-Produkt 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 *Long Fat Networks* (LFNs). Im RFC 1072 (TCP Extensions for Long-Delay Path, vom Oktober 1988) werden Netze auch als LFNs 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 Kommunikationskanal wesentlich. Bei TCP wird das Bandbreiten-Verzögerungs-Produkt auch über das Produkt aus RTT und Datenübertragungsrate berechnet. TCP hat mit LFNs 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 40 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 maximal 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.

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 Bytes lang. ◀

WSopt-Einstellung in Betriebssystemen

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

Im Betriebssystem macOS wird bereits ein Standardskalierungsfaktor, der je nach Betriebssystemversion bei 3 oder 5 liegt, angewendet. Die Einstellung erfolgt im Kernelparameter *net.inet.tcp.win_scale_factor* (siehe hierzu auch Hinweise zu den TCP-Parametern in Anhang A.3).

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 Bytes und bei SACK variabel lang. Wenn in einer SACK-Option n Blocks angegeben sind, ist die Option insgesamt 8n+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 in Anhang A.3) 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 Bytes, 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 TSopt hat den Typ „8“, die Optionsdaten sind 10 Bytes lang. Die TSopt wird beim Verbindungsaufbau für die Dauer der Verbindung aktiviert. Manche Implementierungen wie etwa in Windows erlauben es aber auch, die TSopt 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

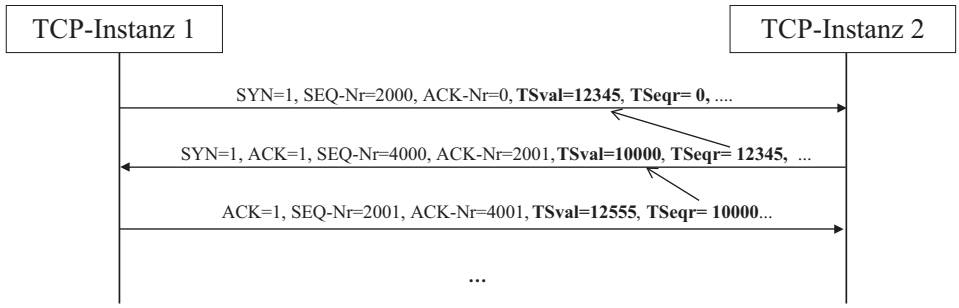


Abb. 3.26 Nutzung von TSval zur Unterstützung der RTT-Berechnung

dann mit der aktuellen Zeit und dem TSecr-Wert aus seiner lokalen Uhr die RTT berechnen (Abb. 3.26).

Die TSopt 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 Verbindungsinkarnationen zu erkennen (siehe Erläuterungen zum Thema *Protect Against Wrapped Sequences in Abschn. 3.5*).

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 Senderichtung 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. 3.27 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 Sequenznummern ankommen, die schon vergeben wurden und die bereits vorher vergebenen noch nicht beim Empfänger angekommen sind.

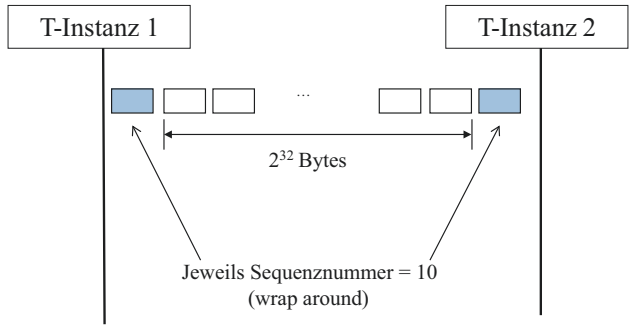


Abb. 3.27 Sequenznummernüberlauf

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 Sequenznummernkollision 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 Sequenznummernkollision zumindest möglich.

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:

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

S_{neu} entspricht der neu zu ermittelnden, S_{alt} der zuletzt benutzten Sequenznummer einer Senderichtigung und x der Anzahl an im TCP-Segment übertragenen Bytes.
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 folgende Brechnung zeigt:

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

In lokalen Netzen ist Letzteres heute schon der Normalfall.

Eine Sequenznummernkollision 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 h) 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 Sequenznummernkollision 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 Sequenznummernkollisionen

Die Maßnahmen zur Vermeidung von Sequenznummernkollisionen wurden unter dem Begriff *Protect Against Wrapped Sequences* (PAWS) zusammengefasst.

Das Problem wird zum einen durch eine Vorgabe bei der Synchronisation der Sequenznummern im Drei-Wege-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 2 min 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 Jahr 1986 wurde das Internet durch massive Stausituationen gestört. Aus diesem Grund suchte man für die Staukontrolle (Congestion Control oder Überlastkontrolle) eine Lösung, die man 1989 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 Vermittlungsschicht des Internets war ursprünglich nicht in der Lage, der TCP-Schicht irgendwelche Informationen zur Netzauslastung zu übergeben. TCP musste also selbst dafür sorgen, dass es Stausituationen möglichst 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 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 Vermittlungsschicht mit einbezieht, wird als Explicit Congestion Notification bezeichnet und in Abschn. 3.6.6) 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 ständig diskutiert und ist auch abhängig von der Implementierung. Eine maximale initiale Fenstergröße für eine TCP-Verbindung wird im RFC 3390 (Increasing TCP's Initial Window) vorgeschlagen. Die Berechnung ergibt sich aus folgender Formel (gemessen in Bytes):

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

Der in der Formel angegebene Wert von 4380 Bytes berechnet sich aus einer Paketgröße (MTU) von 1500 Bytes abzüglich der Standard-Headerlängen für TCP und IP (jeweils 20 Bytes ohne Optionen): 1460 * 3 = 4380.

Wie bereits erwähnt, baut das Staukontrollverfahren 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:

$$\text{Sendekredit} = \min\left(\text{Überlastfenstergröße}, \text{Empfangsfenstergröße}\right)$$

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 die initiale Größe 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 Schwellwertes 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. 3.28, dass das Slow-Start-Verfahren gar nicht so langsam beginnt, wie sein Name andeutet. Es beginnt zwar mit einem kleinen Überlastfenster, das aber schnell größer wird, solange alles gut geht. Als Obergrenze gilt natürlich die Empfangsfenstergröße, die auch dynamisch ermittelt wird. Die Empfangsfenstergröße kann allerdings durch Nutzung der 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

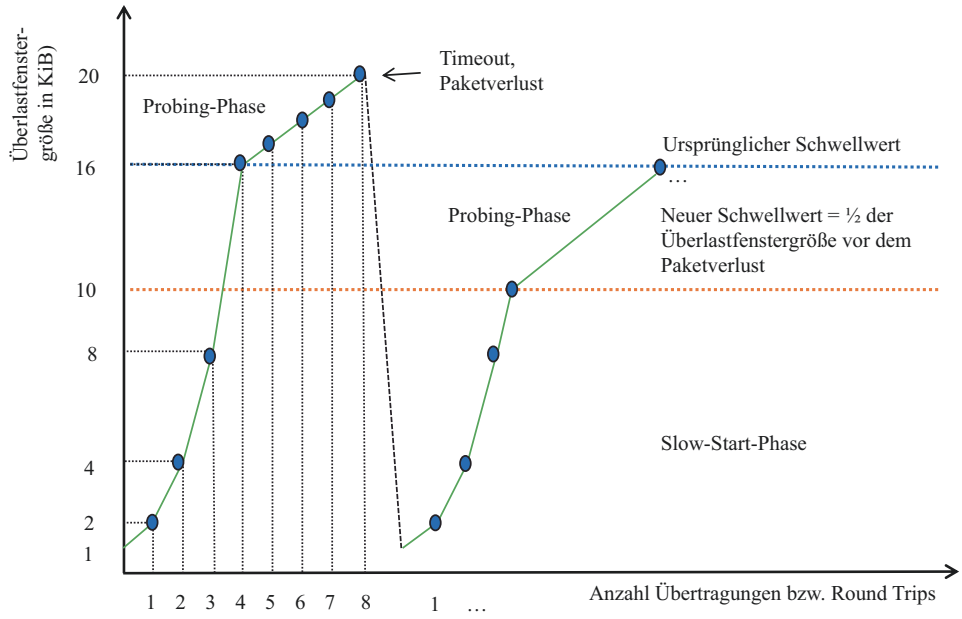


Abb. 3.28 Staukontrolle bei TCP. (Nach Tanenbaum et al. 2021)

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 vorn, 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 also aus. Der Schwellwert wird dann auf 10 KiB gesetzt, was der Hälfte der aktuellen Überlastfenstergröße entspricht, und das Verfahren beginnt von vorn. 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,

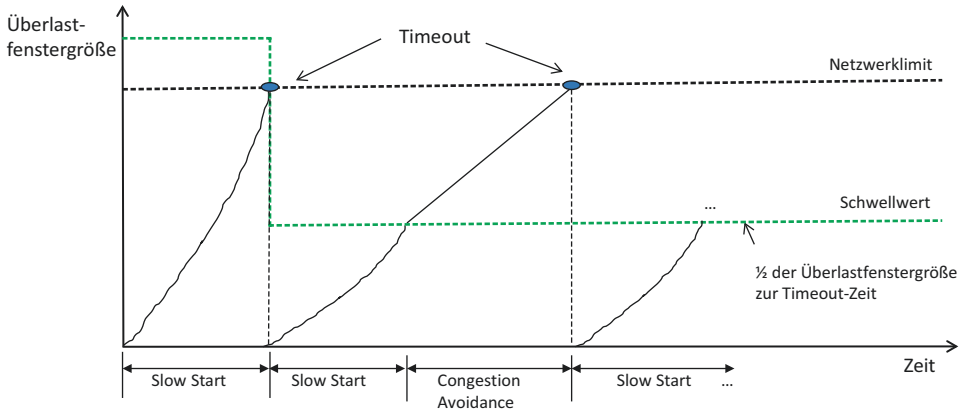


Abb. 3.29 TCP Tahoe bei fehlender Bestätigung

weil TCP die Übertragungsrate additiv erhöht, wenn der TCP-Pfad in Ordnung ist und multiplikativ senkt, wenn ein Datenverlust erkannt wird. In Abb. 3.29 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. Wenn die Anwendung natürlich nur sehr wenig sendet, wird die Bandbreite nicht ausgereizt, und es kommt kaum zu einer Überlastsituation.

3.6.2 TCP Reno und Fast-Recovery-Algorithmus

Wie bereits erläutert, sendet ein TCP-Empfänger immer dann ein ACK-Duplikat, wenn ein empfangenes TCP-Segment nicht in die Reihenfolge passt (Abb. 3.17). Der Sender nutzt den mehrfachen Empfang von Bestätigungen für das gleiche TCP-Segment (RFC 2581, 5681), um einen Paketverlust zu diagnostizieren. Wenn er 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*. TCP Tahoe nutzt diesen Mechanismus bereits, reduziert dabei aber in dieser Situation den Schwellwert auf die Hälfte des aktuellen Wertes, macht mit Slow-Start bei einem Überlastfenster von einer MSS weiter und reduziert damit die Sendelast stark.

TCP-Reno ergänzt das Fast-Retransmission-Verfahren um den *Fast-Recovery-Algorithmus.* Nach dem dreimaligen Empfang eines ACK-Duplikats wird hierbei das aktuelle Überlastfenster halbiert und der Schwellwert nicht so stark reduziert. Er wird vielmehr auf den Wert des Überlastfensters zum Zeitpunkt des Empfangs der ACK-Duplikate

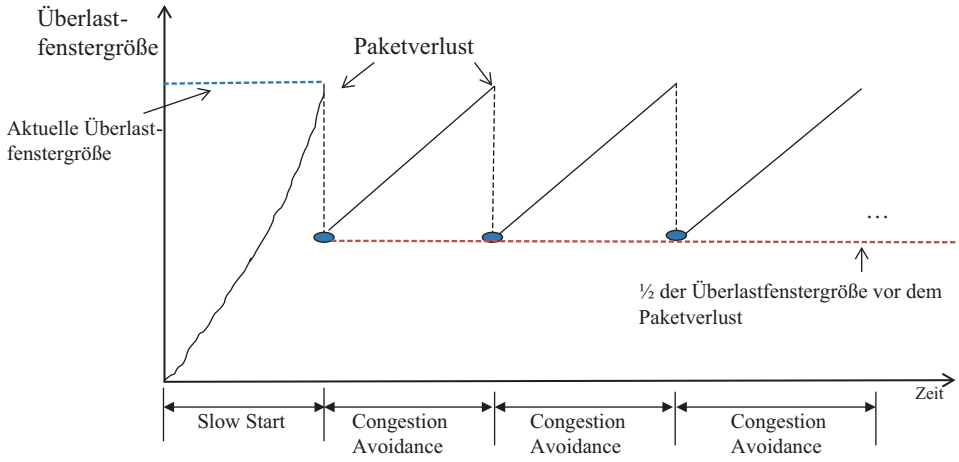


Abb. 3.30 TCP Reno

gesetzt. Das Überlastfenster wird je nach Implementierung noch um die Anzahl der empfangenen ACK-Duplikate erhöht. Die Sendeleistung wird also auch entsprechend angepasst, ein Slow-Start wird aber vermieden, da es sich ja allem Anschein nach nicht um eine Stausituation handelt. Dies wird damit begründet, dass die auf das nicht angekommene TCP-Segment gesendeten Folgesegmente beim Empfänger angekommen sind, sonst wären vom Empfänger keine ACK-Duplikate gesendet worden. Man geht also nur von einem Paketverlust und nicht von einem Stau im Netzwerk aus. Es wird in diesem Fall nur mit der Congestion-Avoidance-Phase fortgefahren. Damit wird ein zu starkes Reduzieren der Sendelast vermieden. Die Bezeichnung „Fast Recovery“ kommt daher, dass nach einem Paketverlust schneller wieder die für die Verbindung angemessene Sendeleistung erreicht werden kann, die Netzwerkkapazität also schneller ausgenutzt wird. Bei einem Retransmission Timeout wird allerdings wie bei TCP Tahoe verfahren (Abb. 3.30) und ein Slow-Start ausgeführt. Ebenso beginnt TCP Reno wie TCP Tahoe nach einem Verbindungsaufbaau erst einmal mit einer Slow-Start-Phase.

3.6.3 TCP NewReno

TCP NewReno ist im RFC 6582 beschrieben und kann als Erweiterung zu TCP Reno betrachtet werden. Bei TCP Reno sendet der Empfänger erst eine Bestätigung aller bereits empfangenen TCP-Segmente, wenn das fehlende Segment erneut vom Sender übertragen wurde und auch beim Empfänger angekommen ist. Wenn nun aber mehr als ein Segment nicht beim Empfänger ankommt, reagiert TCP Reno etwas träge. Der Empfänger wartet nämlich zunächst ab, bis alle bisher fehlenden Segmente empfangen wurden, bevor er alle bereits schon vorher angekommenen Segmente bestätigt. Fehlen mehrere Segmente müssen alle fehlenden Segmente abgewartet werden, bis der Empfänger eine kumulative Be-

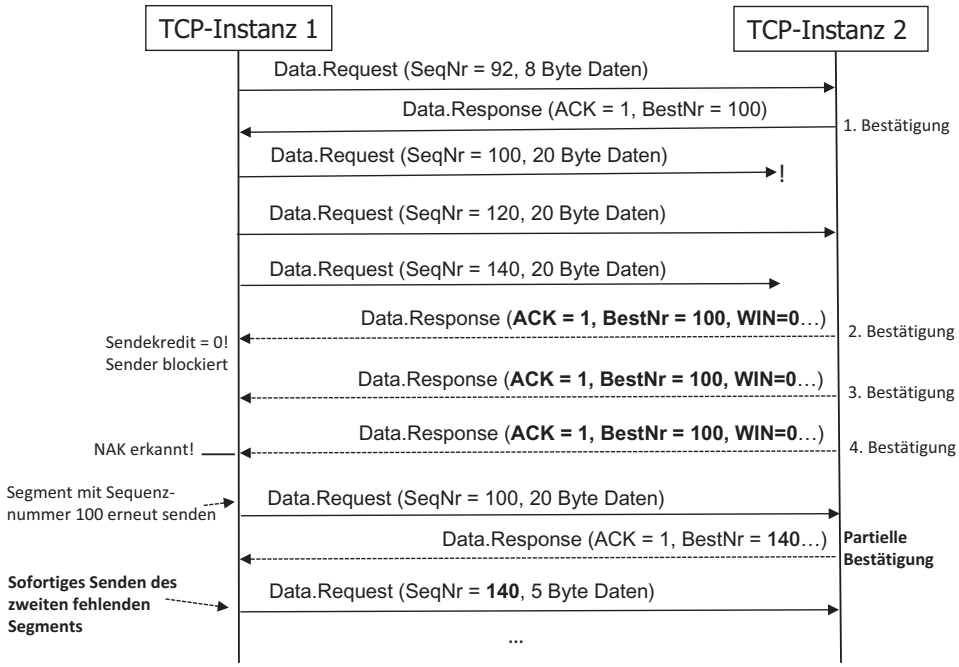


Abb. 3.31 TCP NewReno

stätigung für alle empfangenen Segmente an den Sender sendet. Dies kann dazu führen, dass der Sender bis dahin blockiert, weil sein Sendekredit aufgebraucht ist und er daher keine weitere Übertragung mehr zulässt.

TCP NewReno verbessert TCP Reno. Sobald der Empfänger ein fehlendes Segment empfangen hat, sendet er nicht nur die Bestätigung für dieses eine, sondern auch gleich für alle danach empfangenen Segmente in einer ACK-PDU. Der Sender reagiert seinerseits bei Ankunft einer Duplikatbestätigung mit dem erneuten Senden von folgenden, offensichtlich noch nicht beim Empfänger angekommenen Segmenten. Wie in Abb. 3.31 dargestellt, werden nach dem Empfang eines fehlenden Segments (im Beispiel mit Sequenznummer 100) von der TCP-Instanz 2 alle Segmente bis zum nächsten fehlenden Segment (hier bis Sequenznummer 140) kumulativ bestätigt. Dieser Vorgang wird als *partielle Bestätigung* bezeichnet. Als Antwort auf die Bestätigung sendet der Sender (TCP-Instanz 1) sofort die darauffolgenden Segmente (im Beispiel nur eines mit der Sequenznummer 140), da diese offensichtlich bei der TCP-Instanz 2 noch nicht angekommen sind. Sonst wären sie gleich kumulativ mit der vorhergehenden ACK-PDU bestätigt worden.

Ein weiterer Unterschied zu TCP Reno liegt in der Berechnung der Überlastfenstergröße (cwnd). TCP Reno erhöht den Wert von cwnd bei Ankunft einer Bestätigung (ACK-PDU) um die Anzahl der bestätigten Bytes im TCP-Stream, TCP NewReno erhöht cwnd dagegen nach folgender Gleichung:

$$\text{cwnd} += \min\left(\text{N}, \text{SMSS}\right),$$

wobei N der Anzahl der neu bestätigten Bytes und SMSS der aktuellen Segmentgröße der Verbindung entspricht.

3.6.4 TCP BIC

Ein weiterer Staukontrollmechanismus, der in Xu et al. (2004) vorgeschlagen wurde, wird als Binary Increase Congestion Control for Fast Long-Distance Networks (BIC) bezeichnet. Der Name deutet darauf hin, dass man mit TCP BIC eine Lösung für Netzwerke mit hoher Pfadkapazität im Auge hatte. Derartige Netze, auch als *Fast Long-Distance Networks* oder *Long Fat Networks* bezeichnet, verfügen über ein hohes Bandbreiten-Verzögerungs-Produkt und können somit viele Nachrichten speichern. Bei diesen Netzen ist es besonders wichtig, dass die Bandbreite auch ausgenutzt wird. TCP NewReno und auch dessen Vorgänger erhöhen ja die Überlastfenstergröße nach einer Überlastsituation nur sehr langsam. Es dauert dann gerade bei Netzwerken mit einem hohen Bandbreiten-Verzögerungs-Produkt sehr lange, bis die Pfadkapazität wieder ausgenutzt wird. TCP BIC setzt hier an und versucht die Übertragungsrate so zu erhöhen, dass das aktuelle Optimum schneller erreicht wird.

TCP BIC berechnet die Überlastfenstergröße, also die Anpassung der Übertragungsrate, als binäres Suchproblem ausgehend von der aktuellen Überlastfenstergröße W_{min} und einer Zielgröße, die der Überlastfenstergröße kurz vor der Überlastsituation entspricht und als W_{max} bezeichnet wird. In einer binären Suche wird ein Wert zwischen W_{min} und W_{max} ermittelt. Dieses Verfahren, auch als *Binary Search Increase* bezeichnet, wird genutzt, wenn die Überlastfenstergröße klein ist. Wenn in der Folge kein weiterer Timeout vorkommt, wird der Abstand zwischen W_{min} und W_{max} immer kleiner, und die Überlastfenstergröße nähert sich dem vermuteten aktuellen Maximum W_{max}. Die Entwicklung der Überlastfenstergröße ist in Abb. 3.32 anhand von drei Kommunikationsrunden skizziert.

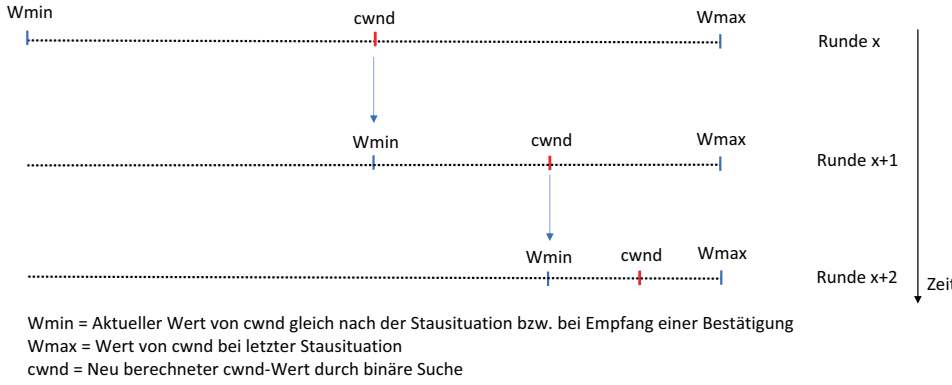


Abb. 3.32 Entwicklung des Überlastfensters bei TCP BIC

Dieses Verfahren wird mit einem weiteren Verfahren kombiniert, das als *Additive Increase* bezeichnet wird. Wenn der Abstand zwischen W_{min} und W_{max} zu groß ist, also einen maximalen Wert S_{max} übersteigt, wird die Überlastfenstergröße bei Ankunft einer Bestätigung gleich um S_{max} erhöht. Das wird so lange durchgeführt, bis der Abstand kleiner als S_{max} ist. Sobald eine Überlastfenstergröße von W_{max} erreicht wurde, wird in eine Probing-Phase wie bei Slow-Start übergegangen, um ein neues Maximum zu finden. Dieser Schritt nimmt die aktuelle Überlastfenstergröße aus Ausgangsbasis und wird auch als Limited Slow-Start bezeichnet.

3.6.5 TCP CUBIC

TCP CUBIC ist im RFC 8312 als Verbesserung zu TCP BIC beschrieben (Ha et al. 2008). Der RFC 8312 ist zwar kein Internetstandard, aber das Verfahren hat sich in der Praxis gut durchgesetzt. In Linux ist es heute das Standardverfahren, in macOS und in Microsoft Windows ist es ebenfalls implementiert. Da es sich nicht um einen Internetstandard handelt, weichen die Implementierungen in den Betriebssystemen etwas voneinander ab, das Grundprinzip ist aber überall gleich.

TCP CUBIC adressiert wie TCP BIC Netzwerke mit hoher Pfadkapazität. Im Gegensatzt zu TCP BIC berechnet TCP CUBIC die Überlastfenstergröße in Abhängigkeit der vergangenen Zeit seit der zuletzt erkannten Stausituation. Nach dem Erkennen einer Stausituation über Duplikatbestätigungen oder sonstige Mechanismen wird die neue Überlastfenstergröße (cwnd) zunächst mithilfe eines multiplikativen Reduktionsfaktors β (< 1) herabgesetzt (aktueller cwnd * β). Man spricht hier auch von *Multiple Decrease*. TCP CUBIC ermöglicht nach einer Überlastsituation in einer Phase 1 eine moderate Vergrößerung von cwnd bis zu der Überlastfenstergröße vor der Stausituation, die mit W_{max} bezeichnet wird. Nach der Überschreitung der aktuell angenommenen Netzwerkkapazität W_{max} ohne Verlust von TCP-Segmenten steigt das Überlastfenster mit der Ankunft von Bestätigungen bei einem konvexen Verlauf langsamer an (Phase 2). In dieser Phase bleibt TCP CUBIC möglichst lange, wodurch die verfügbare Bandbreite gut ausgenutzt werden soll. Wenn weiterhin keine Überlastsituation auftritt, geht das Verfahren in eine exponentielle Wachstumsphase (Phase 3) über, die dazu dient, eine aktuell im Netzwerk noch bessere Bandbreite zu nutzen. Der Verlauf der Funktion zur Berechnung des Überlastfensters entspricht also einem kubischen Polynom, woher auch der Name des Verfahrens abgeleitet ist. In Phase 1 ist die Entwicklung der cwnd konkav, in Phase 2 dann konvex und in Phase 3 exponentiell.

Jedes System kann ja frei wählen, welches Staukontrollverfahren es verwendet. Die Verfahren sollen aber insgesamt fair zueinander sein. Man spricht hier von *TCP Fairness*. Dies ist wichtig, um andere Netzwerkteilnehmer nicht zu benachteiligen. TCP CUBIC gilt gegenüber anderen Staukontrollverfahren, vor allem hinsichtlich TCP NewReno als fair, da es die Größe des Überlastfensters aufgrund des konkaven Verlaufs bis zur angenommenen Netzwerkapazität nicht aggresiv erhöht. Bei jeder Übertragungsrunde ermittelt TCP CUBIC auch die Sendefenstergröße nach dem TCP-Reno-Algorithmus. Sollte

die Berechnung nach den TCP-CUBIC-Algorithmus einen höheren Wert ergeben, verwendet TCP CUBIC den für TCP Reno berechneten Wert.

In Abb. 3.33 ist der Funktionsverlauf zur Anpassung des Überlastfensters bei TCP CUBIC angedeutet. Man erkennt die konkave Region vor und die konvexe Region nach dem angenommenen Kapazitätsmaximum.

Die neue Überlastfenstergröße wird bei Ankunft einer Bestätigung wie folgt berechnet:

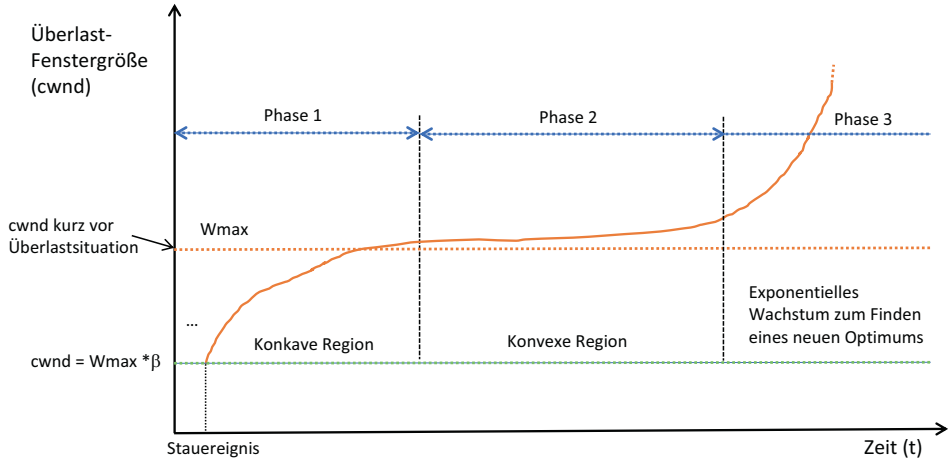
$$\text{cwnd} = C\left(t - K\right)^3 + W_{max}$$

W_{max} ist die Fenstergröße beim letzten Überlastereignis, t ist die vergangene Zeit seit dem letzten Überlastereignis, β ist ein multiplikativer Verkleinerungsfaktor und C eine Skalierungskonstante. Für K gilt dabei folgende Formel:

$$K = \sqrt[3]{\frac{W_{max}\left(1-\beta\right)}{C}}$$

Aus dieser Berechnung ergibt sich die erläuterte kubische Entwicklung des Überlastfensters, wie dies auch in Abb. 3.33 angedeutet ist. Ziel des Verfahrens ist es, dass die Überlastfenstergröße in einer konkaven Region möglichst schnell wieder bei W_{max} ist, um danach möglichst lange dort zu verweilen (konvexe Region), bevor wieder eine deutliche Vergrößerung des Überlastfensters (Phase 3) angestrebt wird.

Es sei noch erwähnt, dass TCP CUBIC genauso wie TCP BIC, Tahoe, Reno und NewReno ein Slow-Start-Verfahren unterstützen, wobei es Anpassungen für Netzwerke mit hoher Pfadkapazität gibt, die impementiert werden können (siehe hierzu beispielsweise RFC 3742, „Limited Slow-Start for TCP with Large Congestion Windows“). Weitere Details hierzu finden sich im RFC 8312.



Wmax =Überlastfenstergröße vor der Stausituation
β = multiplikativer Verkleinerungsfaktor

Abb. 3.33 Entwicklung des Überlastfensters bei TCP CUBIC

3.6.6 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 (*Echo of Congestion Encountered*) 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. 3.34 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. 3.35 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 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

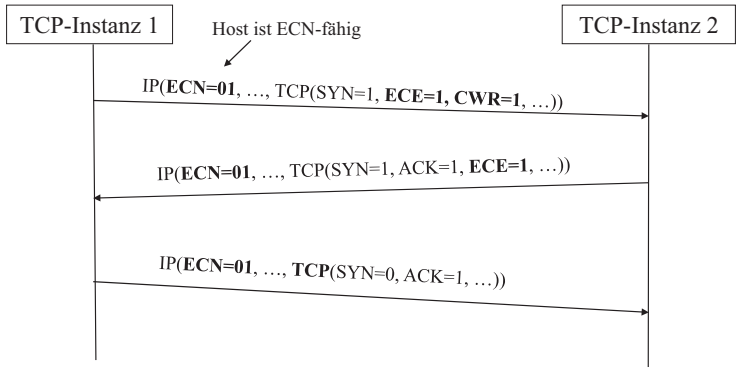


Abb. 3.34 ECN-Aktivierung im Zusammenspiel der Schichten

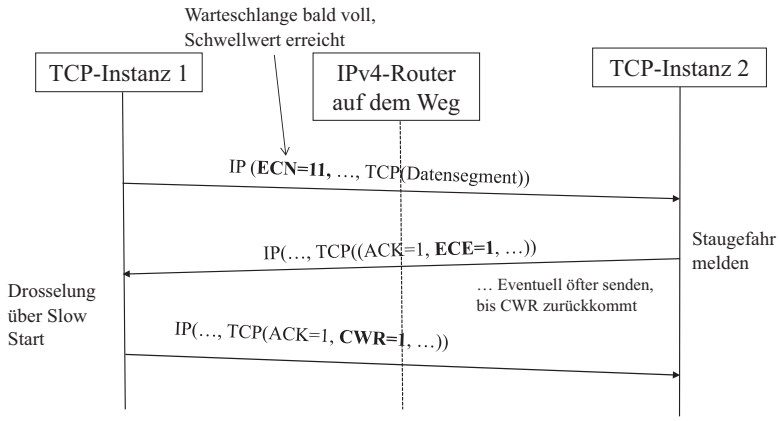


Abb. 3.35 Stausignalisierung mit ECN im Zusammenspiel der Schichten inklusive Router

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.

3.6.7 Weitere Ansätze der TCP-Staukontrolle

Man unterscheidet bei der TCP-Staukontrolle generell zwischen verlustbasierten, verzögerungsbasierten und hybriden Verfahren. Die bisher betrachteten Verfahren sind in die Kategorie „verlustbasiert“ einzuordnen. Diese Verfahren verwenden Paketverluste als Merkmal, um Stauereignisse zu erkennen. Verzögerungsbasierte Ansätze versuchen dagegen, Veränderungen in der Umlaufzeit zu identifizieren, um daraus ein Anwachsen der Puffer in den Endsystemen abzuleiten und aufgrund dieser Annahme die Überlastfenstergröße zu reduzieren. Ziel ist es dabei, durch ein frühzeitiges Reagieren Paketverluste zu vermeiden. Beispiele für derartige Verfahren sind TCP Vegas (Brakmo et al. 1994) und FAST TCP (Jin et al. 2005). Allerdings konnten sich verzögerungsbasierte Ansätze in der Praxis kaum durchsetzen, weil sie als weniger robust gelten. Hybride Verfahren sind eine Mischung beider Ansätze. In diese Kategorie fällt beispielsweise TCP Veno (Fu und Liew 2003).

Im Allgemeinen wird gefordert, dass neuere TCP-Staukontrollverfahren ältere Varianten wie TCP Reno nicht benachteiligen sollen, wenn es um die Verteilung der Bandbreite auf die konkurrierenden TCP-Verbindungen geht. Wie bei TCP CUBIC schon erwähnt, spricht man hier von TCP Fairness. Die Prinizipen hierzu werden im RFC 2914 erläutert und auch bei neueren Entwicklungen beachtet.

Aufgrund der zunehmenden Übertragungsraten und Netzwerkbandbreiten wie etwa in Long Fat Networks werden für die TCP-Staukontrollmechanismen immer wieder Optimierungen vorgeschlagen. Insbesondere für Hochgeschwindigkeitsnetze eignen sich die älteren Verfahren wie TCP Tahoe und TCP Reno, die sehr lange brauchen, um nach einem Ausfall wieder die verfügbare Kapazität auszunutzen, 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 älteren Verfahren wie TCP Tahoe und TCP Reno nur noch alle 5 Mrd. Segmente ein Segmentverlust auftreten dürfte. Ein Überblick über einige weitere Verfahren ist in Sangtae et al. (2007, 2008) zu finden. Die Forschung und Entwicklung gehen hier weiter, insbesondere um auch Lösungen für noch schnellere Netzwerke mit 100 Gbit/s und mehr zu entwickeln (Hock et al. 2019).

Moderne Betriebssysteme ermöglichen es auch, die Staukontrollmechanismen zu konfigurieren. Linux hat beispielsweise einen sehr modernen TCP/IP-Stack mit vielen Konfigurationsmöglichkeiten (siehe Beispiel unten). Die Implementierung verfügt auch über eine modulare Staukontrollarchitektur 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 verloren gehen können, was meist nichts mit einem Netzwerkstau zu tun hat. Eine Lösung ist hier beispielsweise, 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 Staukontrollalgorithmen 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 unter http://sgros.blogspot.de/2012/12/controlling-which-congestion-control.html.

Der Parameter *tcp_allowed_congestion_control* enthält die aktuell eingestellten Staukontrollverfahren. 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.

Die Kernelparameter sind beispielsweise über den Befehl *sysctl* auslesbar und können auch eingestellt werden.

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. 3.36 dargestellt, ist die optimale Länge eines Timers dann gegeben, wenn sich die Wahrscheinlichkeitsdichte der Ankunftszeiten von Bestätigungen wie in Abb. 3.36 (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 Abb. 3.36 (b) beschreibbar. Ein Timer-Intervall T1 führt hier zu unnötigen Neuübertragungen, und T2 führt zu Leistungseinbußen durch zu lange Verzögerungen.

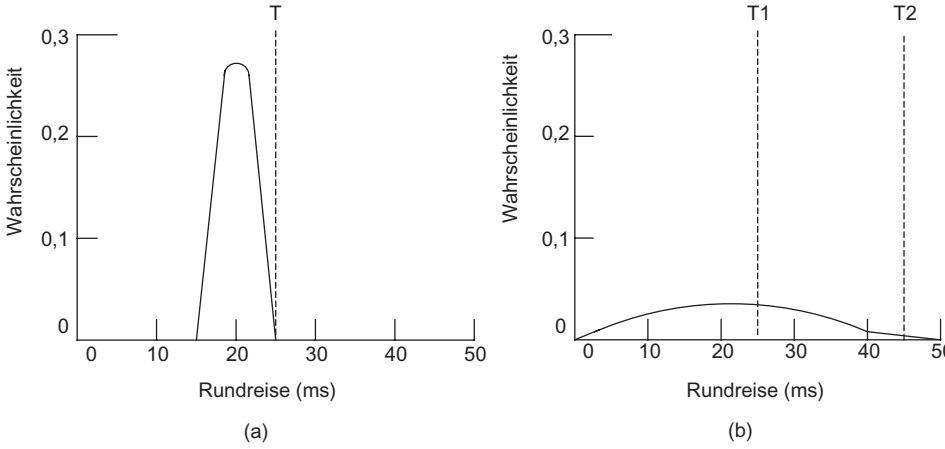


Abb. 3.36 Timerproblematik bei TCP. (Nach Tanenbaum et al. 2021)

Gehen Datenpakete verloren, so müssen sie erneut gesendet werden. Die RTT 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*. Dieser 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 er 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 Round-Trip 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 3 s, auf alle Fälle aber größer als 1 s 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:

$$SRTT = R$$

$$RTTVAR = R / 2$$

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

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:

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

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

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

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 s 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 s gesetzt werden.
- Die Zustandsvariablen SRTT und RTTVAR sind zu löschen, und der nächste RTO-Wert ergibt sich aus der nächsten Messung.

Durch diese Berechnung des 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. 3.37 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.

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 (Keepalive-Probe) an den Partner gesendet. Antwortet dieser nach mehrmaligem Wiederholen nicht, kann die Verbindung abgebaut werden.

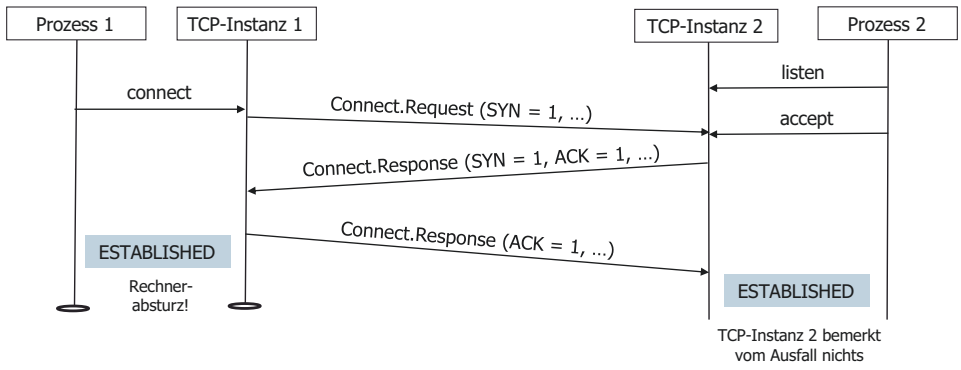


Abb. 3.37 Ausfallsituation während einer TCP-Verbindung

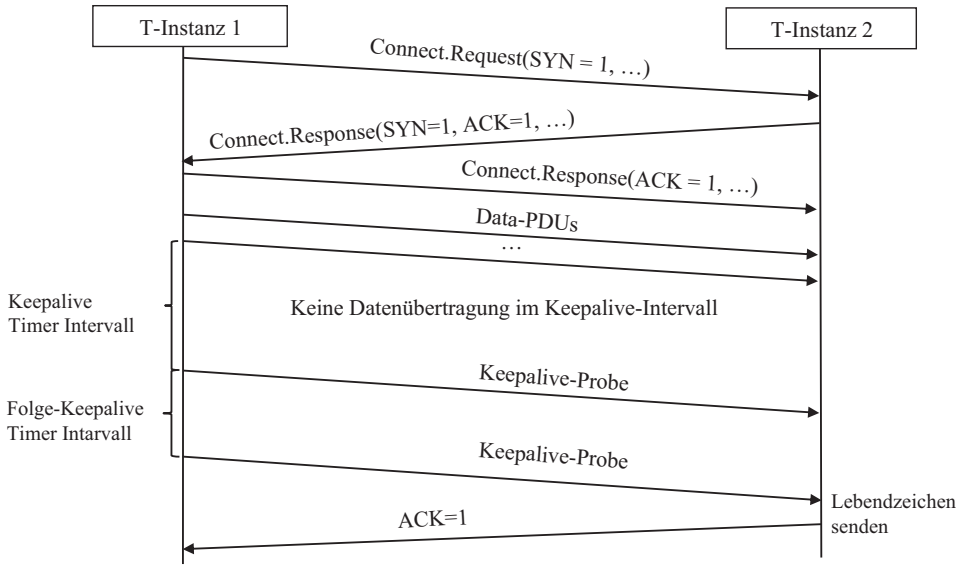


Abb. 3.38 Anwendung des Keepalive-Timers

Der Keepalive-Mechanismus ist bei TCP optional. Er ist standardmäßig ausgeschaltet und kann über eine Socket-Option beim Verbindungsaufbau aktiviert werden. Wird der Mechanismus benutzt, wird der Keepalive Timer zunächst auf 2 h 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. Der Ablauf ist in Abb. 3.38 beispielhaft skizziert. In diesem Beispiel wird die Keepalive-Probe-Nachricht einmal wieder, bevor die Transportinstanz 2 ein Lebendzeichen in Form eines TCP-Segments mit gesetztem ACK-Flag sendet.

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 mithilfe 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 s 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.

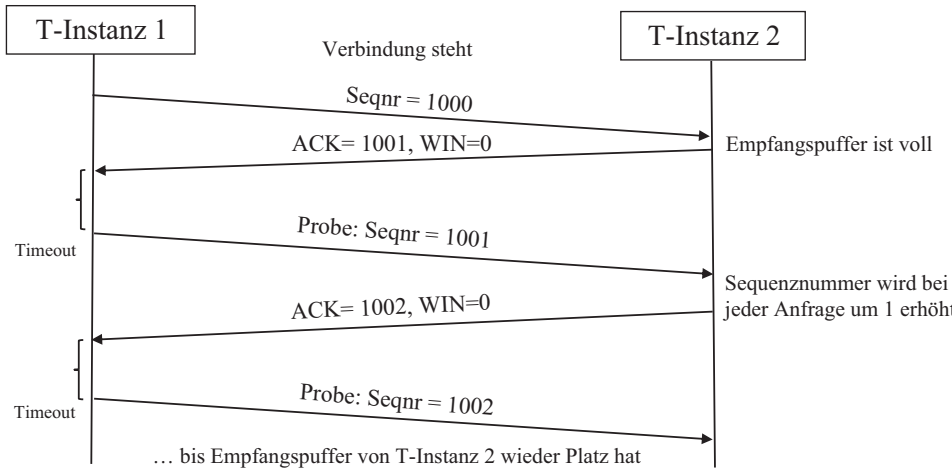


Abb. 3.39 Anwendung des Persistence Timers bei einem Sendekredit von 0

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 Partnerinstanz 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 einer Probe-Nachricht, die ein Byte an Nutzdaten enthält, gesendet. Die empfangende TCP-Instanz kann als Antwort die aktuelle Fenstergröße übermitteln. Sollte der Sendekredit wie in Abb. 3.39 immer noch 0 sein, wird erneut ein Timer aufgezogen. Auf diese Weise wird der Wartezustand der sendenden TCP-Instanz irgendwann verlassen, auch wenn eine Nachricht des Empfängers mit der Information, dass wieder Sendekredit verfügbar ist, verloren geht.

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-(DoS-)Attacke. 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 Verbindung werden nach RFC 793 ermittelt. Manchmal wird aber die Implementierung auch noch vereinfacht, und die Sequenznummern beginnen nicht zufällig. Dann kann ein Angreifer die initialen Sequenznummern auch noch leicht herausfinden. Als Gegenmaßnahmen gegen Sequenznummernangriffe dienen beispielsweise entsprechend konfigurierte 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

Brakmo L. S.; O'Malley, S. W.; Peterson L. L. (1994) Tcp vegas: New techniques for congestion detection and avoidance, in Proceedings of the Conference on Communications Architectures, Protocols and Applications, SIGCOMM '94. New York, NY, USA: ACM, 1994, pp. 24–35, doi: http://doi.acm.org/10.1145/190314.190317

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

Fu C. P.; Liew, S. C. (2003) TCP Veno: TCP enhancement for transmission over wireless access networks, in *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 2, pp. 216–228, Feb. 2003, doi: https://doi.org/10.1109/JSAC.2002.807336

Ha S., Rhee I.; Xu, L. (2008) Cubic: A new tcp-friendly high-speed tcp variant, SIGOPS Oper. Syst. Rev., vol. 42, no. 5, pp. 64–74, Jul. 2008. [Online]. Available: http://doi.acm.org/10.1145/1400097.1400105

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

Hock, M.; Veit, M.; Beumeister, F.; Bless, R.; Zitterbart, M. (2019) TCP at 100 Gbit/s – Tuning, Limitations, Congestion Control, Conference: 2019 IEEE 44th Conference on Local Computer Networks (LCN), doi: https://doi.org/10.1109/LCN44214.2019.8990842

IEEE POSIX (2017) The Open Group Base Specifications Issue 7, IEEE Std 1003.1™-2008, 2018 Edition, http://pubs.opengroup.org/onlinepubs/9699919799/, letzter Zugriff am 01.05.2023

Jin, C. et al. (2005) FAST TCP: from theory to experiments, in IEEE Network, vol. 19, no. 1, pp. 4–11, Jan./Feb. 2005, doi: https://doi.org/10.1109/MNET.2005.1383434

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. (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

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

Tanenbaum, A. S.; Feamster, N. Wetherall, D. J. (2021) Computer Networks, Sixth Edition, Pearson Education Limited, 2021

Xu L; Harfoush K.; Rhee I. (2004) Binary increase congestion control (BIC) for fast long-distance networks, IEEE INFOCOM 2004, Hong Kong, China, 2004, pp. 2514–2524 vol.4, doi: https://doi.org/10.1109/INFCOM.2004.1354672

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 als über TCP zu übertragen, 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 Internetprotokolls 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:

Ergänzende Information Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann [https://doi.org/10.1007/978-3-658-43988-0_4].

P. Mandl, *TCP, UDP und QUIC Internals*, https://doi.org/10.1007/978-3-658-43988-0_4

- Die Nachrichtenübertragung erfolgt zwischen UDP-Kommunikationsendpunkten.
- Es werden keine Empfangsbestätigungen für Pakete gesendet.
- 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 werden von UDP übernommen. Trotzdem bietet UDP – richtig eingesetzt – einige Vorteile:

- Bei UDP ist keine explizite Verbindungsaufbauphase 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 einen 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 Produzentenprozess 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 einen Socket adressierbar, dessen Adresse aus dem Tupel von Internetadresse und einem UDP-Port gebildet wird. Der Wertebereich der UDP-

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

Ports ist unabhängig vom TCP-Portbereich und geht von 0 bis 65535. Zu den wichtigen UDP-Portnummern, die auch als wohlbekannte Ports (well-known ports) definiert sind, gehören u. a.:

- 53, DNS (Domain Name Service)
- 69, TFTP (Trivial File Transfer Protocol)
- 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-Quellportnummer:* Nummer des sendenden Ports.
- *UDP-Zielportnummer:* 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:* Die Prüfsumme ist optional und prüft das ganze UDP-Segment (UDP-Header + Nutzdaten) in Verbindung mit einem Pseudoheader. Die Berechnung der Prüfsumme wird in Abschn. 4.2.2 erläutert.
- *Daten:* Nutzdaten des Datagramms.

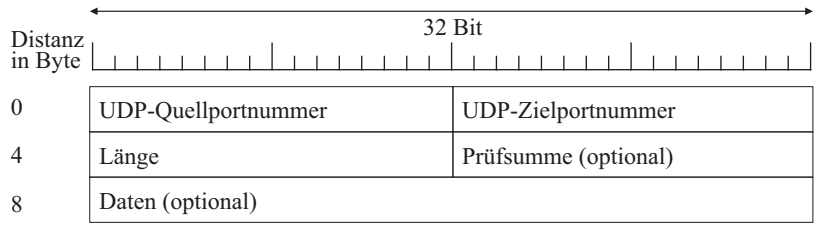


Abb. 4.1 UDP-Steuerinformation

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-Port belegt wird. Sendet ein Anwendungsprozess über einen UDP-Port eine Nachricht, wird sie 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.
- 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-Kommunikationsendpunktes, sofern die Prüfsumme fehlerfrei war.

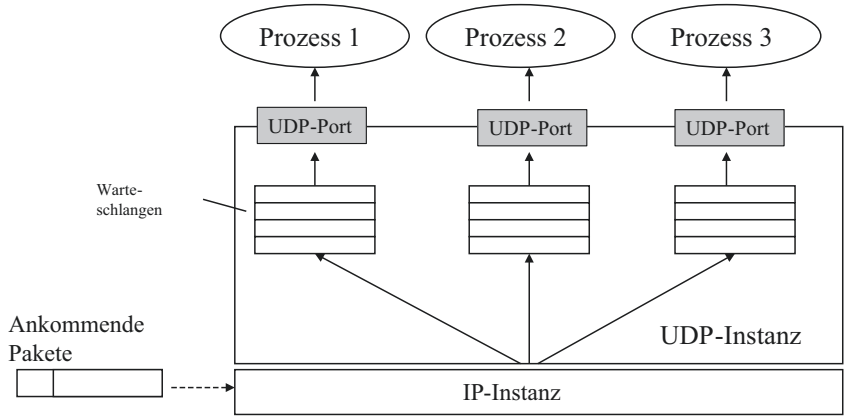


Abb. 4.2 UDP-Demultiplexing

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 einen 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 Empfängerseite interpretieren.

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

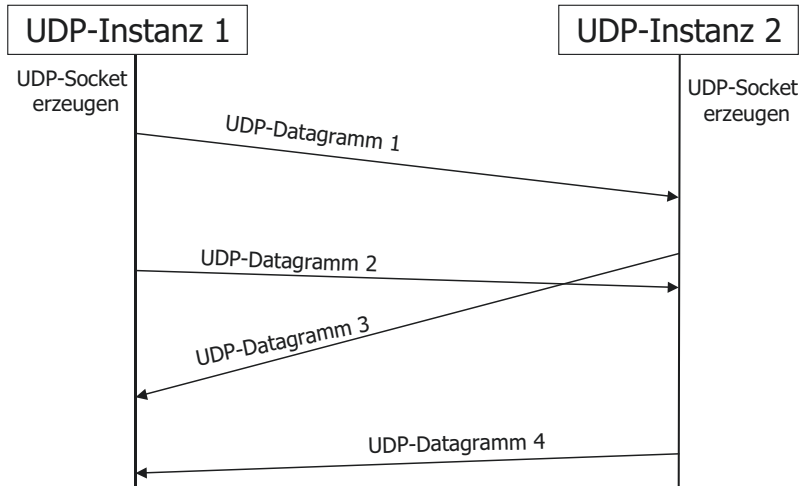


Abb. 4.3 Einfache Kommunikation über UDP

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 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 wird. Er enthält die IP-Adressen des Senders und des Empfängers, ein Nullbyte, den in IP verwendeten Protokollcode für UDP (immer den Wert „17“) und die Gesamtlänge des UDP-Segments, also die 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 Einerkomplementaddition bezeichnet. Ganz korrekt ist die Bezeichnung bei diesem Verfahren nicht, da beim Einerkomplement der Wertebereich nicht überschritten werden darf, beim UDP-Prüfsummenverfahren hingegen schon.

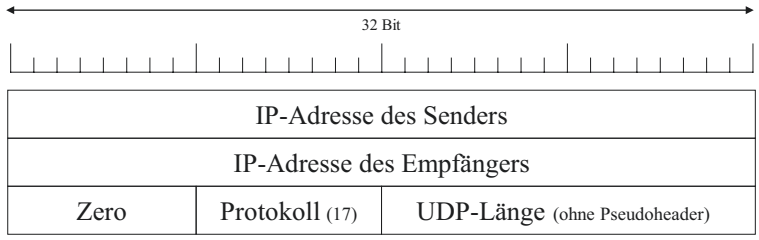


Abb. 4.4 UDP-Pseudoheader

- 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 vorn). 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 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)

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

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. ◀

Es stellt sich die Frage, welche Fehler mit dem Verfahren gefunden werden können. 1-Bit-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. 2-Bit-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-Option 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 Sequenznummern herausfinden muss. Ebenso fehlt ein Handshake für den Verbindungsaufbau, der das Einmischen eines Angreifers 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.

TLS – Transport Layer Security Protocol

5

Zusammenfassung

Das Transport Layer Security (TLS) Protocol ist ein Protokoll zur Authentifizierung, Verschlüsselung und Integritätssicherung, das vor allem in der Web-Kommunikation genutzt wird. Das QUIC-Protokoll verwendet TLS in der Version 1.3 (TLS 1.3) implizit. Im älteren HTTP/2-Protokollstack, der TCP als Transportprotokoll nutzt, wird TLS 1.2 verwendet.

In diesem Kapitel wird TLS einführend erläutert. Auf die mathematischen Grundlagen der zugrundeliegenden kryptographischen Verfahren wird in diesem Kapitel nicht eingegangen. Jedoch werden einige Hintergrundinformationen bereitgestellt, um die grundlegenden Mechanismen nachvollziehen zu können. Zu weiterführenden Protokollinformationen wird auf die RFCs 8446 (TLS 1.3) und 5246 (TLS 1.2) verwiesen, in denen die Standardprotokolle beschrieben werden.

Dieses Kapitel konzentriert sich vor allem auf TLS 1.3 und geht nur dort auf TLS 1.2 ein, wo Unterschiede bzw. Verbesserungen aufgezeigt werden sollen. Um die in QUIC (siehe Kap. 6) eingebauten Security-Ansätze im Detail zu verstehen, ist es empfehlenswert, aber nicht notwendig, dieses Kapitel vorab zu lesen.

5.1 Überblick über TLS

Die klassischen Transportprotokolle TCP und UDP bieten keine Unterstützung zur kryptografischen Absicherung der Kommunikation. Diese Aufgabe wird daher in einem weiteren Protokoll erledigt. Transport Layer Security (TLS) ist ein komplexes Protokoll, das ge-

Ergänzende Information Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann [https://doi.org/10.1007/978-3-658-43988-0_5].

P. Mandl, *TCP, UDP und QUIC Internals*, https://doi.org/10.1007/978-3-658-43988-0_5

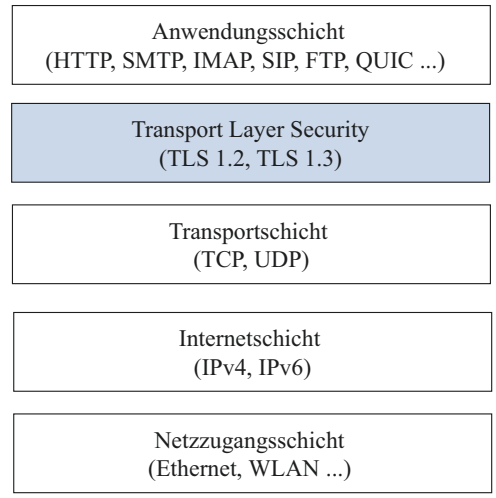


Abb. 5.1 Einordnung von TLS in die Schichtenarchitektur

legentlich auch als Cyber-Sicherheitsprotokoll bezeichnet wird. Es ist unabhängig von Anwendungen konzipiert. Im heutigen Internet wird es sehr umfangreich – insbesondere zur Absicherung der Webkommunikation zwischen Webbrowser und Webserver – verwendet, kann aber viele Anwendungsprotokolle unterstützen, wie die Einordnung in das TCP/IP-Schichtenmodell nach Abb. 5.1 zeigt. Für TLS sind auch TCP-Portnummern festgelegt. Beispielsweise nutzt das Hypertext Transfer Protokoll (HTTP) über TLS den Port 443 (als HTTPS oder HTTP Secure bezeichnet), das Simple Mail Transfer Protokoll (SMTP) den Port 465 (SMTPS), das File Transfer Protokoll (FTP) die Ports 989 und 990 (FTPS) usw.

TLS ist ein internationaler Industriestandard und in der Internet-Community in RFCs genormt. In seinen Anfangszeiten wurde das Protokoll als Secure Socket Layer (SSL) bezeichnet. SSL wird mittlerweile im RFC 6101 als historisch gekennzeichnet. Die ursprüngliche Entwicklung fand bei Netscape etwa 1994 statt, einer Firma, die sich schon früh mit Browsertechnologien befasste. Aus SSL wurde im Jahr 1999 der erste TLS-Standard in der Version 1.0 (RFC 2246) abgeleitet. Nach kleineren Anpassungen in der Version 1.1 (RFC 2817) erfolgte die Weiterentwicklung über die Version 1.2 (RFC 5246) in 2008 bis zur seit 2018 aktuellen Version 1.3 (RFC 8446). Die Version 1.2 ist nach wie vor stark im Einsatz. Zunehmend verbreitet sich mittlerweile die Version 1.3 insbesondere in Verbindung mit dem Protokoll QUIC. Ältere Versionen werden von heutigen Webbrowsern und Webservern aus Sicherheitsgründen nicht mehr unterstützt (BSI 2023).

TLS baut einen sicheren Kanal zwischen Kommunikationspartnern über einem unsicheren Netzwerk (meist IPv4 oder IPv6) auf und kümmert sich um die Authentifizierung von Kommunikationspartnern, die Erzeugung und den Austausch von geheimen Sitzungsschlüsseln für eine vertrauliche Ende-zu-Ende-Datenübertragung sowie um die Integrität der übertragenen Daten. Verschlüsselungs- und Authentifizierungsmethoden sowie Schlüssel werden in einem Handshake-Verfahren, das als TLS-Handshake-Protokoll bezeichnet wird, ausgehandelt. Mit den ausgetauschten Parametern erfolgt die verschlüsselte und authentisierte Datenkommunikation.

5.2 TLS-Protokollaufbau

Um den TLS-Handshake und die Übertragung verschlüsselter Nachrichten mit TLS zu verstehen, ist zunächst der Protokollaufbau von Bedeutung.

Die TLS-Spezifikation 1.3 beschreibt folgende Protokollbestandteile:

- Im *TLS-Handshake-Protokoll* erfolgt das Aushandeln der Protokollparameter und der kryptografischen Algorithmen für die Kommunikation innerhalb einer Sitzung (Session) zwischen den Kommunikationspartnern. Der Terminus „Session" wird hier im Sinne des ISO/OSI-Referenzmodells verstanden.
- Das *TLS-Record-Protokoll* übernimmt Anwendungsdaten, die zu übertragen sind, fragmentiert diese bei Bedarf in mehrere Blöcke (Records), verschlüsselt die Daten, ergänzt Authentifizierungsinformationen und überträgt sie. Jedes Record hat eine maximale Länge von 2^{14} Bytes und wird für sich individuell geschützt.
- *Das TLS-Alert-Protokoll* dient dazu, Notifikationen und Fehlermeldungen (HandshakeFailure, CloseNotify …) zu übertragen, die beim Verbindungsaufbau oder während der Kommunikation auftreten können.

TLS 1.2 enthält zudem noch das *TLS-Change-Cipher-Spec-* und das *Application-Data-Protokoll*. TLS-Change-Cipher-Spec hat die Aufgabe, Änderungen von kryptografischen Parametern zu kommunizieren. Das Application-Data-Protokoll dient der Übertragung der Anwendungsdaten. Diese beiden Protokolle sind in TLS 1.3 nicht mehr explizit aufgeführt, sondern als Content-Typen in das Record-Protokoll integriert. Für die Übertragung der Anwendungsdaten (Application Data) wird bei TLS 1.3 das Record-Protokoll direkt genutzt, bei TLS 1.2 wird das Application-Data-Protokoll verwendet, das seinerseits wieder das Record-Protokoll für die Verschlüsselung und Übertragung nutzt.

Die TLS-Protokollschichten für TLS 1.2 und TLS 1.3 sind zum Vergleich in Abb. 5.2 skizziert. Wie zu erkennen ist, nutzen die TLS-1.3-Protokolle der oberen Schicht, also das Handshake- und das Alert-Protokoll, das Record-Protokoll für die Übertragung. Im Record-Protokoll werden diese über Content-Typen gekennzeichnet

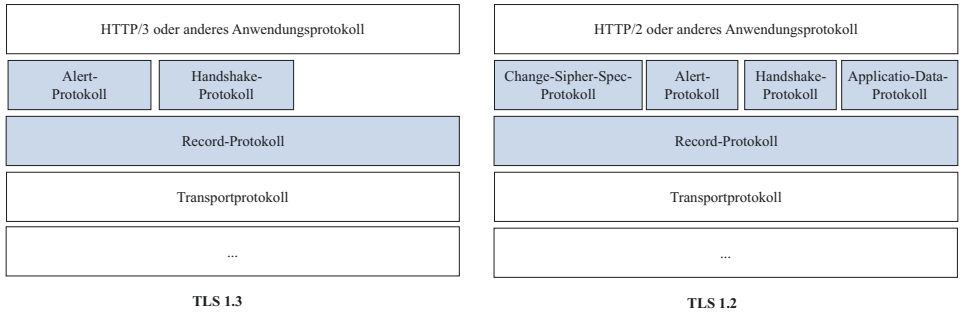


Abb. 5.2 TLS-Protokollschichten: Gegenüberstellung TLS 1.2 und TLS 1.3

Spezifikation der TLS-Nachrichten Nachrichten werden in der TLS-Spezifikation in einer eigenen Notation, die der XDR-Notation (External Data Representation Standard) nach RFC 1014 sehr ähnlich ist, beschrieben. Die Notation enthält einige grundlegende Datentypen wie *uint8*, *uint16*, *uint24* (Unsigned Integer) usw., ermöglicht die Nutzung von Enumerations (*enums*) und das Anlegen eigener Datenstrukturen (*structs*), die wiederum Datenstrukturen enthalten können.

5.3 TLS-Handshake

Beim TLS-Handshake wird eine Sitzung (Session) aufgebaut, deren ausgehandelte Parameter und Algorithmen in mehreren TLS-Verbindungen genutzt werden können. Es sei erwähnt, dass bei Nutzung des Transportprotokolls TCP vorher ein Verbindungsaufbau einer Transportverbindung anhand eines TCP-Drei-Wege-Handshakes erfolgen muss. Dagegen ist bei QUIC der TLS-Handshake bereits in den Verbindungsaufbau integriert. Im Weiteren wird auf den Aufbau der Transportverbindung nicht weiter eingegangen, der Fokus liegt vielmehr auf dem TLS-Handshake.

Das Handshake-Protokoll wird verwendet, um zwischen den Kommunikationspartnern alle Security-Algorithmen und -Parameter für eine Session auszuhandeln. Die Übertragung der Handshake-Nachrichten erfolgt wie bei allen TLS-Nachrichten über das Record-Protokoll. Die dem Handshake-Protokoll zugrunde liegende Datenstruktur ist wie folgt aufgebaut (RFC 8446):

```
enum {
    client_hello(1),
    server_hello(2),
    new_session_ticket(4),
    end_of_early_data(5),
    encrypted_extensions(8),
    certificate(11),
    certificate_request(13),
    certificate_verify(15),
    finished(20),
    key_update(24),
    message_hash(254),
    (255)
} HandshakeType;

struct {
    HandshakeType msg_type;
    uint24 length;
    select (Handshake.msg_type) {
        case client_hello:          ClientHello;
        case server_hello:          ServerHello;
        case end_of_early_data:     EndOfEarlyData;
        case encrypted_extensions:  EncryptedExtensions;
```

```
        case certificate_request:   CertificateRequest;
        case certificate:           Certificate;
        case certificate_verify:    CertificateVerify;
        case finished:              Finished;
        case new_session_ticket:     NewSessionTicket;
        case key_update:             KeyUpdate;
    };
} Handshake;
```

Wie man der Datenstruktur *Handshake* entnehmen kann, sind in Abhängigkeit des Handshake-Typs gemäß Enumeration *HandshakeType* verschiedene Nachrichtentypen für die Abwicklung eines TLS-Handshakes definiert. Eine *ClientHello*-Datenstruktur dient beispielsweise zur Anzeige des Beginns einen Handshakes, eine *Finished*-Datenstruktur wird für die Anzeige der Beendigung eines Handshakes verwendet.

Wir wollen die Felder der beiden Nachrichtentypen *ClientHello* und *ServerHello* näher betrachten (Tab. 5.1). Im RFC 8446 sind diese im Detail erläutert. Der Ablauf eines TLS-Handshakes wird weiter unten beschrieben.

Tab. 5.1 Felder der ClientHello- und der ServerHello-Datenstruktur

Feldbezeichnung	Bedeutung
legacy_version	Protokollversionsangabe für das Aushandeln der TLS-Version Fixer Wert: 0x0303
random	32 Bit lange Zufallszahl, die vom Client bzw. vom Server generiert wird; daraus wird der Sitzungsschlüssel abgeleitet
legacy_session_id	ID einer vorherigen Sitzung, für den Fall, dass der Client eine vorher bereits bestehende Sitzung weiterführen möchte (wird in TLS 1.3 nicht mehr verwendet, dient der Kompatibilität mit Vorgängerversionen)
legacy_session_id_echo	Sitzungs-ID der weitergeführten Sitzung als Bestätigung einer vom Client gewünschten Wiederaufnahme einer vorherigen Sitzung (wird in TLS 1.3 nicht mehr verwendet, dient der Kompatibilität mit Vorgängerversionen)
cipher_suites	Liste der vom Client bevorzugten Chiffrensammlungen, aus denen der Server eine auswählen kann
cipher_suite	Eine vom Server ausgewählte Chiffrensammlung
legacy_compression_methods	Liste von unterstützten Kompressionsverfahren (wird in TLS 1.3 nicht mehr verwendet, dient der Kompatibilität mit Vorgängerversionen)
legacy_compression_method	Der Server antwortet mit einem festen Wert 0, da in TLS 1.3 eine Kompression nicht mehr unterstützt wird (wird in TLS 1.3 nicht mehr verwendet, dient der Kompatibilität mit Vorgängerversionen)
extensions	Optionale Erweiterungen, die in einer eigenen Datenstruktur mit der Bezeichnung *Extension* ausgetauscht werden können. Hierüber können eine Fülle von Protokollergänzungen vereinbart werden. Beispiele hierfür sind eine maximale Fragmentlänge, ein Heartbeat-Protokoll zur gegenseitigen Überwachung, eine Vorgabe für Zertifikate, Angaben zu kryptografischen Parameter für den Schlüsselaustausch usw. Die möglichen Typen können der Enumeration *ExtensionType* entnommen werden

```
uint16 ProtocolVersion;
opaque Random[32];
uint8 CipherSuite[2];

struct {
    ProtocolVersion legacy_version = 0x0303;
    Random random;
    opaque legacy_session_id<0..32>
    CipherSuite cipher_suites<2..2^16-2>
    opaque legacy_compression_methods<1..2^8-1>
    Extension extensions<8..2^16-1>
} ClientHello;

struct {
    ProtocolVersion legacy_version = 0x0303
    Random random;
    opaque legacy_session_id_echo<0..32>
    CipherSuite cipher_suite;
    uint8 legacy_compression_method = 0;
    Extension extensions<6..2^16-1>
} ServerHello;

struct {
    ExtensionType extension_type;
    opaque extension_data<0..2^16-1>
} Extension;

enum {
    server_name(0),                             /* RFC 6066 */
    max_fragment_length(1),                     /* RFC 6066 */
    status_request(5),                          /* RFC 6066 */
    supported_groups(10),                       /* RFC 8422, 7919 */
    signature_algorithms(13),                   /* RFC 8446 */
    use_srtp(14),                               /* RFC 5764 */
    heartbeat(15),                              /* RFC 6520 */
    application_layer_protocol_negotiation(16), /* RFC 7301 */
    signed_certificate_timestamp(18),           /* RFC 6962 */
    client_certificate_type(19),                /* RFC 7250 */
    server_certificate_type(20),                /* RFC 7250 */
    padding(21),                                /* RFC 7685 */
    pre_shared_key(41),                         /* RFC 8446 */
    early_data(42),                             /* RFC 8446 */
    supported_versions(43),                     /* RFC 8446 */
    cookie(44),                                 /* RFC 8446 */
    psk_key_exchange_modes(45),                 /* RFC 8446 */
    certificate_authorities(47),                /* RFC 8446 */
```

```
    void_filters(48),                          /* RFC 8446 */
    post_handshake_auth(49),                   /* RFC 8446 */
    signature_algorithms_cert(50),             /* RFC 8446 */
    key_share(51),                             /* RFC 8446 */
    (65535)
} ExtensionType;
```

Wie aus den Datenstrukturen *ClientHello* und *ServerHello* hervorgeht, wird beim TLS-Handshake zwischen Client und Server eine Chiffrensammlung, also eine Kombination von kryptografischen Algorithmen, ausgehandelt. Der Client schlägt Varianten vor, der Server wählt eine davon aus. Die Chiffrensammlung enthält einen Verschlüsselungsalgorithmus und eine kryptografische Hashfunktion. Zudem wird ein Sitzungsidentifikator vereinbart. Für die Generierung des geheimen Sitzungsschlüssels erzeugt jeder Partner eine Zufallszahl und überträgt sie mit der Hello-Nachricht. In weiteren Extensions werden auch die öffentlichen Teile der Schlüsselpaare (Extension *key_share*), die Zertifikate und die Signaturen ausgetauscht. Die Extensions werden in sogenannten Opaque-Feldern im Binärformat übertragen. Das ist vom Konzept her sehr generisch und ermöglicht auch zukünftige Erweiterungen.

Die Zufallszahlen werden für die Herleitung des geheimen Sitzungsschlüssels mit verwendet. Bei jedem Handshake werden neuen Zufallszahlen erzeugt. Dies erschwert einem Angreifer die Generierung von Replay-Angriffen.

Die Auswahl der Verfahren ist bei TLS 1.3 deutlich eingeschränkter als bei TLS 1.2. Vorgaben gibt es für die verwendbaren Verschlüsselungsverfahren, den Schlüsselaustausch, die Schlüsselableitung für einen geheimen Schlüssel und für die Erstellung digitaler Signaturen.

Grundsätzlich sind zwei Möglichkeiten für den TLS-Handshake gegeben, die als 1-RTT und 0-RTT (Zero Roundtrip) bezeichnet werden. Die Namen geben die Anzahl der Kommunikationsrunden für den Sessionaufbau an, bis eine Anwendungsnachricht übertragen werden kann.

Der TLS-Handshake mit einem Roundtrip, auch als 1-RTT bezeichnet, ist in Abb. 5.3 mit positivem Ausgang dargestellt. Ein möglicher Ablauf sieht vereinfacht wie folgt aus:

- Zunächst sendet der Client nach der Generierung einer Zufallszahl und dem eigenen Schlüsselpaar eine *ClientHello*-Nachricht. Darin enthalten sind die Felder gemäß der *ClientHello*-Datenstruktur (Version, Zufallszahl, mögliche Cipher Suites, Extensions mit Informationen zu den kryptografischen Algorithmen und Parametern wie *key_share*).
- Der Server generiert nach dem Empfang der *ClientHello*-Nachricht auch eine Zufallszahl, berechnet den Sitzungsschlüssel (Session Key) und überträgt dem Client in einer *ServerHello*-Nachricht diese Zufallszahl, die ausgewählte TLS-Version, die ausgewählte Cipher Suite, das ausgewählte Schlüsselaustauschverfahren sowie Extensions mit Informationen zur Generierung des gemeinsamen Sitzungsschlüssels (key_share), zum Zertifikat des Servers und zu seiner digitalen Signatur. Zudem sendet er gleich eine *Finished*-Nachricht, die das Ende des Handshakes andeutet.

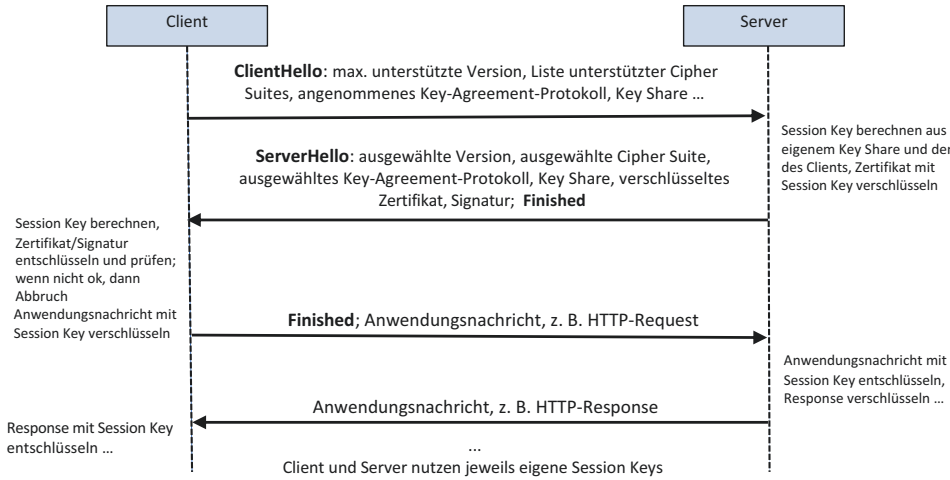


Abb. 5.3 Vereinfachter Verbindungsaufbau mit TLS 1-RTT

- Der Client erzeugt daraufhin nach dem Empfang der *ServerHello*-Nachricht ebenfalls den Sitzungsschlüssel, entschlüsselt und prüft das Zertifikat und die Signatur. Danach antwortet er ebenfalls mit einer *Finished*-Nachricht, mit der auch schon eine Anwendungsnachricht mitgeschickt werden kann (1-RTT!). Optional kann auch eine clientseitige Authentifizierung erfolgen. In diesem Fall sendet auch der Client in der dritten Nachricht zusätzlich Informationen zu seinem Zertifikat. Dies ist bei geschlossenen Anwendungen, in denen die Benutzer mit Zertifikaten ausgestattet werden können, sinnvoll, nicht so sehr bei klassischen Webanwendungen wie Online-Shops.

Beide Partner verwenden im weiteren Verlauf der Kommunikation den erzeugten Sitzungsschlüssel. Außer einer Zufallszahl, die bei der Berechnung sicherstellen soll, dass jede TLS-Verbindung einen eigenen Sitzungsschlüssel erhält, und den öffentlichen Schlüsseln wird keine Information übertragen, die für eine Kompromittierung des Sitzungsschlüssels nützlich sein könnte.

Wenn der Server die *ClientHello*-Nachricht nicht bearbeiten kann, sendet er anstelle einer *ServerHello*- eine *HelloRetryRequest*-Nachricht, um zu signalisieren, dass die TLS-Version, das angebotene Schlüsselaustauchverfahren oder sonstiges nicht unterstützt wird. Der Server teilt dem Client in seiner Antwort mit, unter welchen Bedingungen eine Verbindung möglich wäre. Die *HelloRetryRequest*-Nachricht hat den gleichen Aufbau wie eine *ServerHello*-Nachricht. Es gibt keinen eigenen Nachrichtentyp für einen *HelloRetryRequest*, in einer Extension wird lediglich der Wert 65535 angegeben, um einen *HelloRetryRequest* anzudeuten. Wenn der Client die Anforderungen des Servers nicht befriedigen kann, bricht er den Handshake-Prozess ab.

In TLS 1.3 es auch zulässig, dass beide Partner einen vordefinierten und im Vorfeld ausgetauschten geheimen Schlüssel (Pre-shared Key, PSK) nutzen. Auch dieser wird in Extensions der *ClientHello*- und der *ServerHello*-Nachricht ausgetauscht. Der PSK

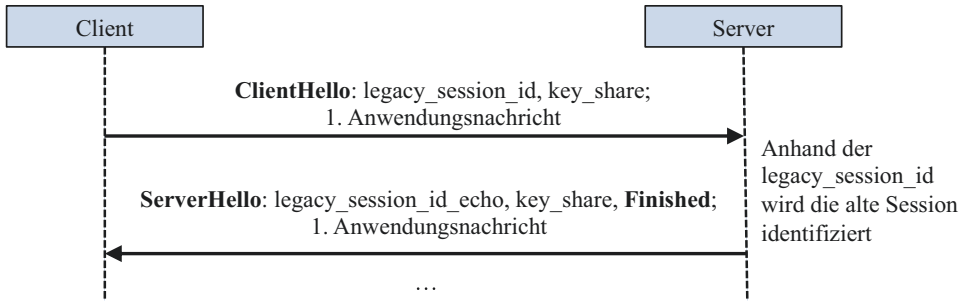


Abb. 5.4 Verbindungsaufbau mit TLS 0-RTT

kann außerhalb der Kommunikationsbeziehung vorab ausgetauscht worden sein, er kann aber auch aus einer vorherigen Verbindung stammen. Im diesem Fall ist ein 0-RTT-Verbindungsaufbau möglich, bei dem gleich verschlüsselte Daten mitgesendet werden.

Abb. 5.4 zeigt den Ablauf des TLS-Handshakes mit Zero Roundtrip (0-RTT). Bereits bekannte Schlüsselinformationen werden in der *ClientHello*-Nachricht in Extensions übertragen. Nach RFC 8446 sind die Extensions *psk_key_exchange_modes* und *pre_shared_key* erforderlich. Die erste Anwendungsnachricht wird durch die Extension *early_data* angezeigt

TLS 1.2 Resumption Key In TLS 1.2 wird mit der ClientHello-Nachricht im Feld legacy_session_id eine Session-ID übertragen werden, die der Server bereits in der letzten Sitzung an den Client übergeben hatte. Diese ID wird auch als Resumption Key bezeichnet. Der Client merkt sich die ID und nutzt sie zur Weiterführung der Verbindung. Wenn der Server die Session-ID akzeptiert, bestätigt der diese im Feld legacy_session_id_echo der ServerHello-Nachricht. Diese Felder werden in TLS 1.3 nicht mehr benutzt, aber aus Kompatibilitätsgründen noch unterstützt.

Die *ClientHello*- und die *ServerHello*-Nachricht sind bereits mit dem bekannten Geheimschlüssel verschlüsselt. Allerdings hat diese Vorgehensweise einen Nachteil: Sie ermöglicht keine *Perfect Forward Secrecy*, da Replay-Attacken nicht verhindert werden können. Serverimplementierungen sollen die Bearbeitung aufeinanderfolgender 0-RTT-Wiederholungen aber gemäß RFC 8446 limitieren, um die Auswirkungen einer Kompromittierung zu limitieren. Eine serverseitige TLS-Implementierung ist nicht verpflichtet, *ClientHello*-Anfragen mit Zero Roundtrips anzunehmen. PFS ist ein Konzept zur nachhaltigen Sicherheit verschlüsselter Daten. Dabei wird gefordert, dass ein neu benutzter Schlüssel nicht von einem älteren Schlüssel abhängen darf. Wenn ein Angreifer einen Schlüssel herausfindet, darf er keine älteren und auch keine zukünftig erzeugten Schlüssel daraus ableiten können (Alashwali et al. 2019). Mit PFS wird gewährleistet, dass Daten auch im Nachhinein nicht entschlüsselt werden können, selbst wenn ein Schlüssel bekannt wird. Um dies zu erreichen darf die Verschlüs-selung eines Sitzungsschlüssels (Session Keys) nicht nur von einem privaten Schlüssel abhängen. PFS ist mit dem Diffie-Hellman-Schlüsselaustauschverfahren realisierbar. Das Verfahren arbeitet mit flüchtigen Parametern und überträgt die Schlüssel nicht übers Netzwerk

Authentifikation
Authentifikation (oder Authentifizierung) ist das Nachweisen einer Identität eines Subjekts oder Objekts. Der Nachweis erfolgt durch sogenannte *Credentials*, wie etwa einer Benutzerkennung und einem Passwort oder einem Personalausweis. Im Internet erfolgt die Authentifikation oft über digitale Zertifikate, die über asymmetrische Verschlüsselungsverfahren ausgetauscht werden. In einem Zertifikat wird einem Subjekt oder Objekt (das kann ein Computerservice sein) ein öffentlicher Schlüssel zugeordnet. Die Zuordnung wird durch das Zertifikat verifiziert. Zertifikate werden durch eine Zertifizierungsstelle (Certificate Authority, CA) ausgestellt. Dies sind akkreditierte und unabhängige Institutionen wie beispielsweise der TÜV in Deutschland.

Authentifikation darf nicht mit Authentisierung verwechselt werden. Mit Authentisierung ist nämlich der Prozess zur Überprüfung der Identität gemeint. Auch mit dem Betriff der Autorisierung besteht Verwechslungsgefahr. Autorisierung ermöglicht das Gewähren bestimmter Privilegien oder des Zugangs zu diesen aufgrund einer nachgewiesenen Identität.

X.509-Zertifikat

X.509 ist ein Standardformat (ITU-T-Standard, ISO/IEC 9594-8) für ein Public-Key-Zertifikat. Es enthält Informationen über ein Subjekt oder Objekt, für das es ausgestellt wurde, u. a. den Namen des Subjekts oder Objekts, den Namen des Ausstellers, die Gültigkeitsdauer und den öffentlichen Schlüssel des Subjekts oder Objekts.

Extended-Validation-Zertifikate (EV-TLS/SSL-Zertifikate) sind X.509-Zertifikate, an die strenge Vergabekriterien geknüpft sind. Der Antragsteller wird beispielsweise detailliert überprüft. Seit einiger Zeit nutzen immer mehr Webseitenbetreiber diese Art von Zertifikaten. ◀

Verschlüsselung
Verschlüsselung ist die Umwandlung eines Klartextes in einen Geheimtext (Chiffretext) mithilfe eines Schlüssels. Dieser Vorgang wird auch als Chiffrierung bezeichnet. Die Umkehrung ist dann die Entschlüsselung (Dechiffrierung). Ziel ist die Geheimhaltung von Nachrichten. Man unterscheidet verschiedene Verfahren:

- Symmetrische Verschlüsselungsverfahren
- Asymmetrische Verschlüsselungsverfahren
- Hybride Verschlüsselungsverfahren

Symmetrische Verschlüsselungsverfahren verwenden den gleichen Schlüssel zum Ver- und Entschlüsseln. Der Schlüssel muss vorher vereinbart bzw. ausgetauscht werden.

Beispiele für symmetrische Verschlüsselungsverfahren sind DES (Data Encyption Standard), TripleDES (3DES) und AES_128_GCM (Advanced Encryption Standard mit 128-Bit-Schlüssel). Die mathematische Basis für DES sind Bitpermutationen und Substitution, AES_128_GCM nutzt Galois-Felder. DES ist mittlerweile weitgehend von AES abgelöst.

In TLS 1.3 sind für die Verschlüsselung nur *AEAD-Verfahren* (Authenticated Encryption with Associated Data) zugelassen. Diese Verfahren kombinieren die Verschlüsselung mit der Authentifizierung.

Erlaubt sind nach RFC 8446 die Verfahren ChaCha20-Poly1305 (siehe RFC 8439) und AES-GCM (Bellare und Tackmann 2016). ChaCha20 ist bei dieser Kombination das Verschlüsselungsverfahren und Poly1305 ein kryptografischer Hashalgorithmus.

GCM steht für Galois/Counter Mode. Das Verfahren verwendet zur Verschlüsselung einen Zähler (Counter Mode) und enthält eine Authentifizierungsfunktion, die auf der Galois-Theorie, einer mathematischen Theorie zur Lösung von algebraischen Gleichungen, basiert.

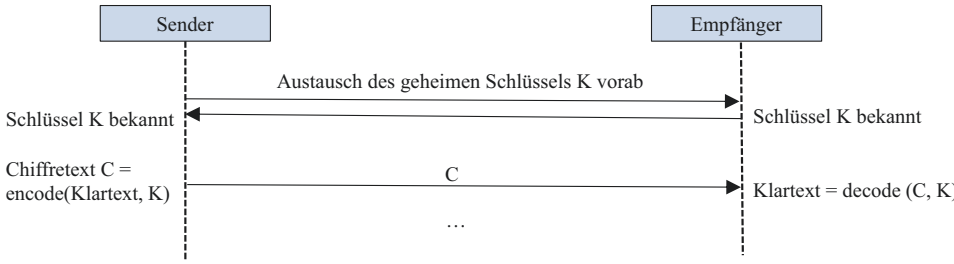


Abb. 5.5 Symmetrische Verschlüsselung

Ebenso sind in TLS 1.3 Pre-Shared Keys (PSKs) zugelassen. Das sind symmetrische Schlüssel, die im Vorfeld an die Kommunikationspartner verteilt werden.

Abb. 5.5 zeigt den prinzipiellen Ablauf der Verschlüsselung und Entschlüsselung beim symmetrischen Verfahren, wenn der geheime Schlüssel K vorher auf eine gesicherte Art und Weise ausgetauscht wurde.

Bei den *asymmetrischen Verschlüsselungsverfahren*, die auch als Public-Key-Verfahren bezeichnet werden, wird ein Schlüsselpaar {public key, private key} zum Ver- und Entschlüsseln verwendet. Der öffentliche Schlüssel (public key) wird bereitgestellt, der private Schlüssel (private key) ist geheim. Wenn die Verschlüsselung mit öffentlichem Schlüssel erfolgt, kann nur der Besitzer des privaten Schlüssels entschlüsseln. Wenn die Verschlüsselung mit privatem Schlüssel erfolgt, können alle, die den öffentlichen Schlüssel kennen, eine Nachricht entschlüsseln. Das Verfahren nutzt man auch für die Authentifizierung über ein digitales Signaturverfahren.

Asymmetrische Verschlüsselungsverfahren sind beispielsweise:

- *RSA* (Erfinder Rivest, Shamir und Adleman), dessen mathematische Basis die Faktorisierung sehr großer Primzahlen ist. Hier wird die Schwierigkeit ausgenutzt, große Zahlen in Primfaktoren zu zerlegen.
- *PGP* (Pretty Good Privacy) basiert auf gegenseitigem Vertrauen beim Austausch von öffentlichen Schlüsseln über Zertifikate.
- *ECC* (Elliptic Curve Cryptography) nutzt die mathematischen Eigenschaften von elliptischen Kurven.

Bei *hybriden Verschlüsselungsverfahren* wird ein geheimer Schlüssel K über ein asymmetrisches Verfahren ausgetauscht; die Verschlüsselung erfolgt dann über ein symmetrisches Verfahren mit dem ausgetauschten Geheimschlüssel K.

Kryptografische Hashfunktionen

Kryptografische Hashfunktionen dienen der Generierung einer meist kürzeren Zeichenkette aus beliebig langen Datensätzen. Diese Zeichenketten werden auch als digitaler Fingerabdruck, Message Digest, kryptografische Prüfsumme oder Message Authentification Code (MAC) bezeichnet. Hashfunktionen werden auch für die Erzeugung digitaler Signaturen verwendet.

Ein digitaler Fingerabdruck muss eindeutig sein. Eine Umkehrung der Berechnung sollte nicht möglich sein. Kollisionen dürfen praktisch nicht vorkommen, aus zwei verschiedenen Datensätzen darf also nicht der gleiche Fingerabdruck erzeugt werden.

SHA-256, SHA-384 und SHA-512: SHA-256, SHA-384 und SHA-512 (Secure Hash Algorithm mit 256, 384 und 512 Bit) sind Beispiele für kryptografische Hashfunktionen. Sie erzeugen Hashwerte in der angegebenen Bitlänge.

TLS 1.3 unterstützt alle drei SHAs.

Digitale Signaturen
Unter einer digitalen Signatur ist ein digitaler Ersatz für eine manuelle Unterschrift, welche die Integrität und Authentizität von digitalen Nachrichten oder allgemein von Daten mithilfe von kryptografischen Verfahren sicherstellt. Ist eine Nachricht digital signiert, geht man davon aus, dass sie einem definierten Absender zugeordnet werden kann und eine Verfälschung nicht möglich ist.

Realisiert werden digitale Signaturen meist mit asymmetrischen, kryptografischen Verfahren. Ein Sender einer digitalen Nachricht berechnet einen Hashwert und verschlüsselt diesen mithilfe eines geheimen, privaten Schlüssels (Signaturschlüssels). „Dies ist dann die digitale Signatur", die der Nachricht bei der Übertragung hinzugefügt wird. Damit ist es dem Empfänger möglich, eine zweifelsfreie Urheberschaft über den öffentlichen Schlüssel des Senders zu ermitteln.

Beispiel für Signaturverfahren

Ein bekanntes Signaturverfahren ist RSA, das nach den Erfindern Rivest, Shamir und Adleman benannt ist. Dieses Verfahren nutzt ein Schlüsselpaar bestehend aus privatem und öffentlichem Schlüssel und kann auch zum Verschlüsseln von Nachrichten verwendet werden.

TLS-Implementierungen müssen auch digitale Signaturverfahren unterstützen. In TLS 1.3 werden für die Erstellung von digitalen Signaturen die asymmetrischen Verfahren RSA nach RFC 3447, ECDSA (Elliptic Curve Digital Signature Algorithm) nach RFC 6962 und EdDSA (Edwards-curve Digital Signature Algorihm) nach RFC 8032 unterstützt. Ebenso können Pre-shared Keys verwendet werden. ◄

Schlüsselaustauschverfahren (Key Agreement)
Unter einem Schlüsselaustauschverfahren versteht man ein Verfahren, mit dem man Kommunikationspartner mit einem gemeinsamen geheimen Schlüssel, wie zum Beispiel einem Sitzungsschlüssel, für ein symmetrisches Verschlüsselungsverfahren versorgt. Dazu wird ein Schlüsselaustauschprotokoll benötigt. Im einfachsten Fall kann der Austausch manuell erfolgen, was sich aber in der Praxis als schwierig erweist. Daher verwendet man Schlüsselaustauschalgorithmen, die den Austausch von Schlüsseln über einen unischeren Kanal während der Kommunikation über asymmetrische Verfahren ermöglichen.

In TLS 1.3 muss das Diffie-Hellman-Schlüsselaustauschverfahren *ECDHE* oder ein *Pre-Shared Key* (PSK) verwendet werden. Die Parameter werden beim Handshake ausgetauscht. Eine mögliche Parameterbelegung wird mit *secp256r1* bezeichnet (RFC 5480). Als Parameter werden die x- und y-Koordinaten eines Punktes auf der elliptischen Kurve in der *KeyShare*-Extension übergeben.

Elliptic Curve Diffie-Hellman Ephemeral (ECDHE)

ECDHE ist eine Variante des Diffie-Hellmann-Schlüsselaustauschverfahrens. Der geheime Sitzungsschlüssel selbst wird nicht übertragen, sondern nur das Ergebnis einer Rechenoperation, über das man nicht auf den Schlüssel schließen kann. In der Rechenoperation wird ein gemeinsamer Punkt auf einer elliptischen Kurve berechnet. Auf diesem Punkt basiert dann die Schlüsselgenerierung jeweils lokal. Das hintere E, das für „Ephemeral" steht, bedeutet, dass der öffentliche Schlüssel der Partner temporär erzeugt wird. Ziel des Verfahrens ist es, möglichst wenige geheime Informationen über Nach-

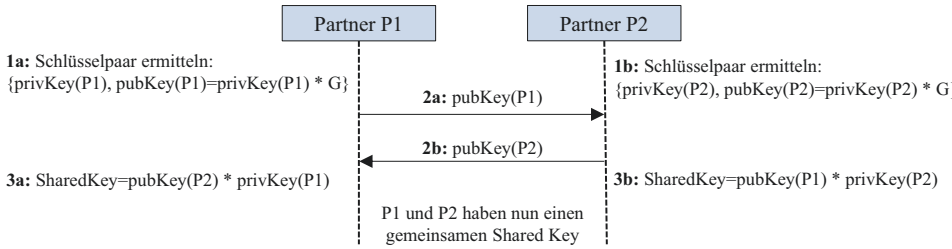


Abb. 5.6 Schlüsselaustausch mit ECDHE

richten auszutauschen. Im klassischen Diffie-Hellmann-Schlüsselaustauschverfahren wird dagegen ein vorher verteilter öffentlicher Schlüssel verwendet.

ECDHE garantiert Perfect Forward Secrecy. Beide Partner (in Abb. 5.6 als P1 und P2 bezeichnet) verwenden je eine zufällig ausgewählte geheime Zahl als private Schlüssel, die als privKey(P1) und privKey(P2) bezeichnet werden. Beide kennen auch eine ECC-Kurve (Elliptic Curve Cryptography) mit einem Generatorpunkt G auf dieser Kurve als Geheimnis. Für das Geheimnis G gilt:

$$G = pubKey(P1) * privKey(P2) = pubKey(P2) * privKey(P1)$$

Abb. 5.6 zeigt den vereinfachten Ablauf des Schlüsselaustauschs in drei Schritten: Im ersten Schritt (1a und 1b) ermitteln beide Partner ihre privaten und öffentlichen Schlüssel auf der Basis des beiden bekannten Generatorpunkts G. Im zweiten Schritt (2a und 2b) tauschen sie ihre öffentlichen Schlüssel aus. Im dritten Schritt (3a und 3b) erzeugen beide lokal einen gemeinsamen Schlüssel aus dem öffentlichen Schlüssel des Partners und dem eigenen privaten Schlüssel. ◀

Cipher Suites

Cipher Suites oder Chiffrensammlungen sind Kombinationen aus kryptografischen Algorithmen wie Schlüsselaustauschverfahren, Hashfunktionen und Verschlüsselungsalgorithmen. TLS 1.3 schreibt für eine Implementierung einige Cipher Suites vor, andere werden empfohlen. Nach BSI (2023) werden in TLS 1.3 Cipher Suites mit der Namenskonvention TLS_*AEAD*_*Hash* angegeben. *AEAD* bezeichnet ein authentisiertes Verschlüsselungsverfahren, das im Record-Protokoll verwendet wird. *Hash* gibt eine Hashfunktion an.

Beispiel für Cipher Suite

TLS_AES_128_GCM_SHA256 ist die Bezeichnung einer in TLS 1.3 zwingend erforderlichen Cipher Suite. *TLS* gibt das Protokoll an, *AES_128_GCM* den Verschlüsselungs- und Authentifizierungsalgorithmus. Hier handelt es sich um den Advanced Encryption Standard mit einem 128-Bit-Schlüssel. *SHA256* legt die Hashfunktion SHA-256 fest.

Die beiden Chiffrensammlungen mit den Bezeichnungen *TLS_AES_256_GCM_SHA384* und *TLS_CHACHA20_POLY1305_SHA256* sollten auch in TLS 1.3 implementiert sein (siehe hierzu RFC 8446). ◀

5.4 Verschlüsselte Nachrichtenübertragung

Nach dem TLS-Handshake können Nachrichten verschlüsselt übertragen werden. Hierzu wird das Record-Protokoll verwendet. Die Datenstruktur des Record-Protokolls ist im RFC 8446 wie folgt beschrieben:

```
enum {
    invalid(0),
    change_cipher_spec(20),
    alert(21),
    handshake(22),
    application_data(23),
    (255)
} ContentType;

struct {
    ContentType type;
    ProtocolVersion legacy_record_version;
    uint16 length;
    opaque fragment[TLSPlaintext.length];
} TLSPlaintext;

struct {
    opaque content[TLSPlaintext.length];
    ContentType type;
    uint8 zeros[length_of_padding];
} TLSInnerPlaintext;

struct {
    ContentType opaque_type = application_data;
    ProtocolVersion legacy_record_version = 0x0303;
    uint16 length;
    opaque encrypted_record[TLSCiphertext.length];
} TLSCiphertext;
```

In der Enumeration *ContentType* sind die möglichen Protokolltypen festgelegt, die in ein Record, das mit der Datenstruktur *TLSPlainText* beschrieben ist, eingebettet werden können. Die übertragenen Daten werden als *Fragment* oder *Block* bezeichnet. Ein Fragment hat eine maximale Länge von 2^{14} Bytes.

Bei der Verschlüsselung wird die Datenstruktur *TLSPlainText* in eine weitere Datenstruktur mit der Bezeichnung *TLSInnerPlainText* und diese wiederum in *TLSCiphertext* als verschlüsseltes Record eingebettet. Die Länge eines verschlüsselten und mit einem Hashwert versehenen Fragments darf $2^{14} + 256$ Bytes nicht überschreiten. Die Felder der verwendeten Datenstrukturen sind in Tab. 5.2 und 5.3 kurz erläutert.

Tab. 5.2 Felder des Record-Protokolls in der Datenstruktur *TLSPlainText*

Feldbezeichnung	Bedeutung
type	Content-Typ des Records
legacy_record_ version	Version von TLS Das Feld enthält den fixen Wert 0x0303
length	Länge des folgenden Fragment-Feldes in Bytes, das den Wert 16.384 nicht überschreiten darf.
fragment	Unverschlüsseltes Fragment aus den Anwendungsdaten.

Tab. 5.3 Felder der verschlüsselten Fragmente in den Datenstrukturen *TLSInnerPlainText* und *TLSCiphertext*

Feldbezeichnung	Bedeutung
content	Content des höherwertigen, eingebettenen TLS-Protokolls.
type	Content-Typ des höherwertigen, eingebettenen TLS-Protokolls.
zeros	Eine beliebig lange Folge von Nullbytes (0x00) zum Auffüllen des unverschlüsselten Fragments. Die Begrenzung ergibt sich aus der Maximallänge für ein Fragment (2^{14} Bytes)
opaque_type	Typfeld mit festem Wert 23.
legacy_record_ version	Version von TLS. Das Feld enthält den fixen Wert 0x0303.
length	Länge des folgenden Fragment-Feldes in Bytes, das den Wert 2^{14} + 256 = 16.384 + 256 = 16.640 nicht überschreiten darf
encrypted_record	Dieses Feld enthält die mit AEAD verschlüsselte Datenstruktur TLSInnerplaintext. In diesem Feld befindet sich also das mit dem Sitzungsschlüssel verschlüsselte Fragment samt Authentifizierungsinformation.

Eine Implementierung des Record-Protokolls nimmt unverschlüsselte Anwendungsnachrichten entgegen und zerlegt sie in Fragmente. Danach werden die Fragmente einzeln mit dem ausgehandelten Sitzungsschlüssel verschlüsselt, wobei die Information zur Authentifizierung gleich mit erzeugt wird. Schließlich wird jedes verschlüsselte und authentifizierte Fragment nach Ergänzung der restlichen Headerfelder der TLS-PDU an die Partnerinstanz übertragen. Diese packt jedes Fragment aus und entschlüsselt es. Nach einer Zusammenführung der Fragmente einer Anwendungsnachricht (Defragmentierung) wird die Anwendungsnachricht an die nächsthöhere Schicht weitergereicht. Die Bearbeitung der zu übertragenden Anwendungsdaten durch das Record-Protokoll erfolgt, ohne den Inhalt zu betrachten, und ist für die Senderseite in Abb. 5.7 skizziert.

Bei TLS 1.3 wird durch die zwingend vorgegebene Anwendung von AEAD-Algorithmen mit der Verschlüsselung auch eine Authentifizierung durchgeführt. Auf eine Komprimierung der Fragmente wird bei TLS 1.3 verzichtet. Im Vergleich dazu wird bei TLS 1.2 ein Fragment zunächst komprimiert, danach wird ein Hashwert für das Fragment ermittelt, und anschließend erfolgt eine gemeinsame Verschlüsselung des komprimierten Fragments und des Hashwertes.

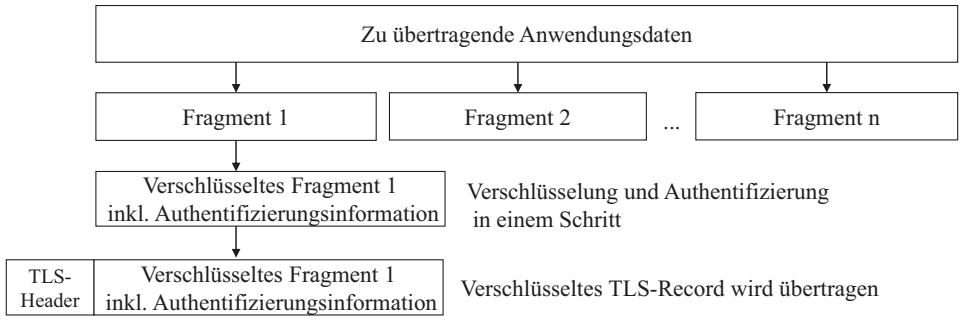


Abb. 5.7 Fragmentierung und Verschlüsselung von Anwendungsdaten im Record-Protokoll

AEAD-Algorithmen und deren Nutzung in TLS 1.3

AEAD-Verfahren kombinieren die Verschlüsselung mit der Authentifizierung und der Sicherstellung der Integrität der Nachrichten. Dabei steht AE für „Authenticated Encryption" und AD für „Associated Data". Dies sind assoziierte Daten, die in die Authentifizierung eingebunden werden. Im Unterschied zu anderen Verfahren kann mit AEAD eine verschlüsselte Nachricht zusätzlich mit einem Kontext verknüpft werden. Die eigentliche Nachricht wird mit einem Hashwert versehen und verschlüsselt. Die zusätzlichen Daten werden nicht verschlüsselt. Ein verschlüsselter Text kann nur entschlüsselt werden, wenn die bei der Verschlüsselung verwendeten zusätzlichen Daten, also der verknüpfte Kontext, beim Entschlüsseln verfügbar ist. Für den Empfänger sind die assoziierten Daten nützlich, um die Authentizität der kommunizierenden Partner zu überprüfen und die Integrität des Nachrichteninhalts sicherzustellen, also um Manipulationen zu erkennen. Ein Angreifer kann die Nachrichten nicht entschlüsseln.

Welche Daten für die Authentifizierung hinzugefügt werden, hängt auch vom Protokoll ab. Bei TLS 1.3 werden Headerfelder des Record-Protokolls als zusätzliche Daten verwendet, und zwar exakt die Felder aus der Datenstruktur *TLSCiphertext*. Nach RFC 8446 werden die zusätzlichen Daten durch Konkatenation der *TLSCiphertext*-Felder (`||`) gebildet:

```
additional_data = TLSCiphertext.opaque_type ||
    TLSCiphertext.legacy_record_version ||
    TLSCiphertext.length
```

Die Eingabe für die AEAD-Verschlüsselung besteht bei TLS 1.3 aus vier Parametern, die nach RFC 8446 aus dem Geheimschlüssel (write_key), einem Nonce-Wert (nonce), den zusätzlichen Daten (additional_data) und dem bereits verschlüsselten Nachrichtentext (plaintext) besteht. Der verschlüsselte Nachrichtentext wird der Datenstruktur *TLSInnerPlaintext* entnommen:

```
AEADEncrypted =
    AEAD-Encrypt(write_key, nonce, additional_data, plaintext)
```

Die Entschlüsselung auf der Empfängerseite nutzt ebenfalls wieder den Nonce-Wert, den Geheimschlüssel, die Zusatzdaten und den verschlüsselten AEAD-Nachrichtentext, wie der Pseudocode aus RFC 8446 zeigt:

```
plaintext of encrypted_record =
    AEAD-Decrypt(peer_write_key, nonce,
        additional_data, AEADEncrypted)
```

Der Nonce-Wert wird von einer intern verwalteten Sequenznummer je Record und vom Initialisierungsvektor (*client_write_iv* bzw. der *server_write_iv*) der jeweiligen Sendeseite durch logische XOR-Operation der beiden Werte erzeugt. Jeder Kommunikationspartner verwaltet diese eigenständig und zählt damit die innerhalb einer Verbindung empfangenen bzw. gesendeten Records. Die Werte von *client_write_iv* bzw. *server_write_iv* werden durch die TLS-Implementierungen von den ausgetauschten Schlüsselinformationen abgeleitet. Beide Partner kennen also die Werte.

5.5 Abbau der TLS-Verbindung

Bevor ein Kommunikationspartner die Schreibseite der Verbindung schließt, muss er ein *CloseNotify*-Alert senden, um die TLS-Verbindung ordentlich abzubauen. Sowohl der Client als auch der Server können diese Alert-Nachricht senden. Die Nachricht signalisiert, dass der Sender keine Nachrichten mehr über die Verbindung senden wird. Ein weiterer Sendeversuch der Anwendung wird unterbunden.

Beim Empfang eines *CloseNotify*-Alerts sollte die TLS-Implementierung der Anwendung gemäß Spezifikation mitteilen, dass keine Daten mehr von der Partneranwendung gesendet werden. Zudem sollte als Antwort ebenfalls ein *CloseNotify*-Alert gesendet werden. Damit zeigt der Partner an, dass auch von seiner Seite nichts mehr gesendet wird. Ankommende Nachrichten nach Empfang eines *CloseNotify*-Alerts sind zu ignorieren.

Wie bei anderen Protokollen ist es auch beim TLS-Verbindungsabbau sinnvoll, dass die Anwendung das ordnungsgemäße Schließen einer Verbindung initiiert. Unabhängig davon ist auch die TCP-/UDP-/QUIC-Transportverbindung zu schließen.

Ein frühzeitiger Abbruch kann auch schon beim Handshake durch beide Kommunikationspartner über ein *UserCanceled*-Alert kommuniziert werden. In diesem Fall wird der Handshake nicht zu Ende geführt.

TLS-Verbindungen können auch aufgrund von Fehlern über Error-Alerts wie *UnexpectedMessage*, *HandshakeFailure*, *BadCertificate InternalError* usw. abgebrochen werden. Diese Alerts führen zum sofortigen Schließen der Verbindung.

Die Sitzungskontexte (Sitzungs-ID, Sitzungsschlüssel) bleiben nach dem TLS-Verbindungsabbau noch eine bestimmte Zeit erhalten. Sie können dann für eine erneute Verbindungsaufnahme über eine 0-RTT-Nachricht genutzt werden, um den erneuten Handshake abzukürzen. Wie lange die Sitzungsdaten aufbewahrt werden, ist im RFC 8446 nicht festgelegt und daher implementierungsspezifisch.

5.6 Zusammenfassung und Ausblick

In diesem Kapitel wurde ein Überblick über TLS 1.3 mit einigen Rückblicken auf TLS 1.2 gegeben, wobei auf die Erläuterung der mathematischen Grundlagen zu den in TLS verwendeten kryptografischen Algorithmen verzichtet wurde. Die kryptografischen Zusammenhänge wurden in abstrakter Form beschrieben. Der Fokus des Kapitels lag bei der Darstellung des Protokolls und des Nachrichtenaustauschs, wobei auch hier viele Details, insbesondere von Spezialfällen und Fehlersituationen nicht vertieft wurden. Hierzu sei auf den RFC 8446 verwiesen.

TLS 1.3 wurde im Vergleich zu TLS 1.2 schneller und sicherer gemacht. TLS 1.2 benötigt zwei Roundtrips bis Nutzdaten gesendet werden können (1-RTT), im optimalen Fall können sogar schon mit der ersten Handshake-Nachricht Nutzdaten mitgeschickt werden (0-RTT). Ältere, nicht mehr als sicher geltende kryptografische Algorithmen wurden bei TLS 1.3 entfernt, und die unterstützten Cipher Suites wurden deutlich eingeschränkt. Bei Nutzung des 1-RTT-Handshakes wird auch Perfect Forward Secrecy garantiert. Es wird nur noch die nötigste Information über Nachrichten übertragen. Der Schlüsselaustausch erfolgt über das ECDHE-Verfahren mit fest vorgegebener Parametrierung. Parameter werden hierfür nicht mehr ausgehandelt. Durch die zwingend vorgegebene Anwendung von AEAD-Algorithmen wird mit der Verschlüsselung auch eine Authentifizierung durchgeführt. Bei TLS 1.2 sind diese Schritte noch getrennt.

Durch die Tatsache, dass QUIC (Kap. 6) das Security-Protokoll TLS 1.3 bereits integriert, ist es schon fester Bestandteil von HTTP/3. Es ist daher in den nächsten Jahren zu erwarten, dass TLS 1.3 die Vorgängerversion TLS 1.2 Zug um Zug ablösen wird. Nach einer Studie der APNIC Labs (Huston 2022) nutzten in 2022 bereits 47,7 % der Android- und 44,5 % der iOS-User HTTP/3 und damit TLS 1.3. Im mobilen Umfeld scheint HTTP/3 also stark verbreitet zu sein, unter Windows (10 %), macOS (1,5 %) und Linux (0,3 %) noch deutlich weniger. Vor allem HTTP-Requests, die von den Browsern Google Chrome (52,2 %) und Apple Safari (44,6 %) gesendet wurden, nutzten oft schon HTTP/3. Die Nutzungszahlen für HTTP/3 werden in der Statistik nach APNIC (2023) regelmäßig aktualisiert.

Empfehlung des BSI Das Bundesamt für Sicherheit in der Informationstechnik (BSI) empfiehlt derzeit die TLS-Versionen 1.2 und 1.3. In BSI (2023) sind Empfehlungen für die Auswahl der Cipher Suites, kryptografischer Algorithmen und Schlüssellängen zusammengefasst. Diese Empfehlungen werden regelmäßig aktualisiert.

Literatur

Alashwali, E. S., Szalachowski, P., Martin, A. (2019) Towards Forward Secure Internet Traffic, https://arxiv.org/abs/1907.00231v1

APNIC (2023) HTTP/3 Use by Country https://stats.labs.apnic.net/quic, https://stats.labs.apnic.net/quic, letzter Zugriff am 30.08.2023

Bellare, M. Tackmann B. (2016) The Multi-User Security of Authenticated Encryption: AES-GCM in TLS 1.3", Proceedings of CRYPTO 2016, July 2016, https://eprint.iacr.org/2016/564

BSI (2023) Technische Richtlinie TR-02102-2 Kryptographische Verfahren: Empfehlungen und Schlüssellängen, Teil 2 – Verwendung von Transport Layer Security (TLS), Version 2023-01, Bundesamt für Sicherheit in der Informationstechnik

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

Pohlmann, N. (2019) Cyber-Sicherheit – Das Lehrbuch für Konzepte, Mechanismen, Architekturen und Eigenschaften von Cyber-Sicherheitssystemen in der Digitalisierung. Springer Vieweg Verlag, Wiesbaden

Huston, G. (2022) A look at QUIC use, https://blog.apnic.net/2022/07/11/a-look-at-quic-use/, letzter Zugriff am 30.08.2023

QUIC – UDP-Based Multiplexed and Secure Transport

6

Zusammenfassung

Die Initiative für die Entwicklung von QUIC ging von Google aus. Das Unternehmen entwickelte im Jahr 2012 das Protokoll mit der Bezeichnung „Quick UDP Internet Connections“. Googles Protokoll wird seither als gQUIC bezeichnet. gQUIC ist eine Weiterentwicklung des inzwischen obsoleten Protokolls SPDY (SPeeDY), das noch auf TCP aufsetzte und ebenfalls von Google stammte. Seit 2021 ist QUIC in angepasster Form auch von IETF in den RFCs 8999, 9000, 9001 und 9002 standardisiert.

QUIC wird Transportprotokoll genannt, liegt aber eigentlich über UDP und ist somit ganz exakt als Anwendungsprotokoll zu bezeichnen. Da es aber Transportaufgaben übernimmt, bleiben wir im Weiteren bei der Bezeichnung „Transportprotokoll“. QUIC ist vor allem auf die Optimierung der Webkommunikation zugeschnitten und ist ein zuverlässiges, verbindungsorientiertes und verschlüsseltes Protokoll.

Aktuell gibt es noch keinen API-Standard, wie er aus der Socket-Programmierung her bekannt ist. Trotzdem wurde QUIC in den letzten Jahren immer beliebter. Google und Facebook nutzen es bereits fast überwiegend als Basis für ihre Dienste. Allerdings sind in Ermangelung eines API-Standards heute nur proprietäre Implementierungen in den meisten Browsern im Einsatz.

Welche TCP-Schwächen das neue Protokoll QUIC adressiert, wozu es genau konzipiert wurde und welche Protokollmechanismen es unterstützt, wird in diesem Kapitel einführend erläutert. Für eine komplette Beschreibung des Protokolls mit all seinen Feinheiten wird auf die oben genannten RFCs verwiesen. QUIC verwendet in der aktuellen Version das Transport Layer Security Protocol in der Version 1.3 (TLS 1.3) für die Authentifizierung der Kommunikationspartner und für die Verschlüsselung der Nachrichten. TLS ist in der aktuellen QUIC-Version fester Bestandteil des Protokolls.

Ergänzende Information Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann [https://doi.org/10.1007/978-3-658-43988-0_6].

P. Mandl, *TCP, UDP und QUIC Internals*, https://doi.org/10.1007/978-3-658-43988-0_6

Dieses Kapitel kann man auch ohne tiefgreifende kryptografische Kenntnisse verstehen, da auf die konkreten Maßnahmen zu Authentifizierung der Kommunikationspartner und auf die Verschlüsselungstechniken, die durch TLS eingebracht werden, nicht weiter eingegangen wird. Wer sich doch mehr dafür interessiert, sollte vorher Kap. 5 lesen.

6.1 Gründe für ein weiteres Transportprotokoll

Das bisher vorherrschende Transportprotokoll TCP ermöglicht eine verbindungsorientierte und gesicherte Kommunikation zwischen Transportinstanzen. Es geht nichts verloren, und auch die Reihenfolge der Nachrichten wird genau eingehalten. Duplikate sind ebenfalls nicht zugelassen. Wenn ein gesendetes TCP-Segment nicht beim Partner ankommt, wird es so lange angefordert, bis die Lücke geschlossen ist. So komfortabel dies auch ist, bei der Kommunikation zwischen einem Browser und einem Webserver kann dies zu ernsthaften Verzögerungen führen. Hinzu kommt, dass bei TCP immer ein Verbindungsaufbau notwendig ist, der mindestens drei Nachrichten benötigt. Das zwischen Browser und Webserver verwendete Anwendungsprotokoll ist HTTP, das auf TCP aufsetzt. Jeder HTTP-Anfrage muss also in der Regel ein aufwendiger Verbindungsaufbau vorangehen. Da die meisten Webverbindungen zumindest in wichtigen Teilen auch abhörsicher sind und daher HTTPS nutzen, ist zudem ein TLS-Handshake auszuführen, der dafür sorgt, dass geheime Schlüssel zur Verschlüsselung der Nachrichten ausgetauscht werden und eine Authentifizierung der Partner erfolgt. Dadurch werden beim Aufbau einer Verbindung zwischen einem Browser und einem Webserver zusätzlich weitere Nachrichten ausgetauscht. Die Latenzzeit beim Verbindungsaufbau ist also ziemlich hoch.

Das HTTP-Protokoll in der Version 2 (HTTP/2) leidet zudem an einem weiteren Performance-Problem. Üblicherweise werden bei einer Webverbindung parallel mehrere Streams geöffnet, in denen die Daten für die Darstellung des Webcontents im Browser nebenläufig übertragen werden. Ein Stream oder Datenübertragungsstream ist aus Sicht der Webanwendung ein Kommunikationskanal zwischen zwei Endpunkten – in diesem Fall zwischen einem Browser und einem Webserver –, über den verschiedenste Objekttypen wie Text, Bilder, Audioinhalte, Videos usw. gesendet werden können. Wenn alle Streams aber über eine TCP-Verbindung übertragen werden und ein einziger Stream aufgrund eines Nachrichtenverlusts blockiert, werden auch alle anderen Streams an der Übertragung gehindert. Hier spricht man vom Head-of-Line-Blocking (HoL-Blocking).

Unter anderem um die hohe Latenzzeit zu reduzieren, wurde QUIC entwickelt. Da QUIC nicht mehr auf TCP aufsetzt, sondern UDP als Transportprotokoll verwendet, kann HOL-Blocking weitgehend verhindert werden. Allerdings wird die Lösung der Aufgabe nun in die nächsthöhere Protokollschicht verschoben. In QUIC wird ein Stream-Multiplexing durchgeführt, d. h., mehrere Datenströme (Streams) sind über eine einzige QUIC-Verbindung möglich und können sich gegenseitig kaum mehr behindern. Zudem sind bereits gemäß RFC 9001 Security-Maßnahmen über die direkte Integration des Protokolls TLS in der Version 1.3 vorhanden. Damit muss kein eigener TLS-Handshake mehr

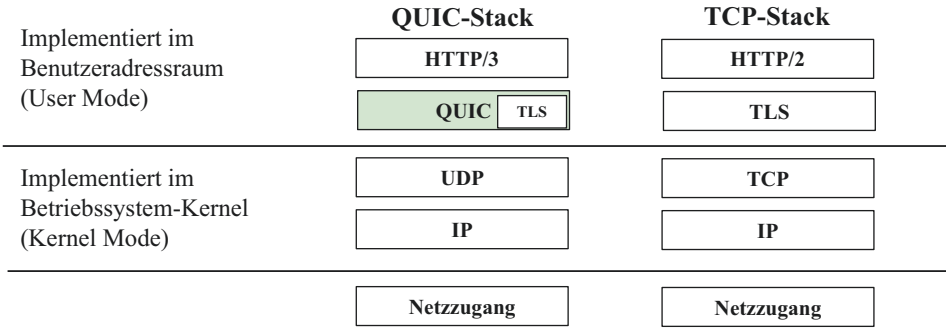


Abb. 6.1 QUIC-Stack für HTTP im Vergleich zum TCP-Stack

durchgeführt werden, was beim Verbindungsaufbau zu einer Nachrichtenreduzierung im Vergleich zu TCP führt. Kryptografische Funktionen zur Authentifizierung, Verschlüsselung und Gewährleistung der Integrität sind also bereits in QUIC enthalten. Insgesamt soll damit ein schnellerer Verbindungsaufbau mit geringer Latenz erreicht werden, ohne dass dadurch sogenannte Middleboxes wie Router verändert werden müssen.

QUIC stellt die Basis für die HTTP-Weiterentwicklung, die als HTTP/3, früher als HTTP over QUIC, bezeichnet wird, dar. In Abb. 6.1 werden die Protokollstacks von HTTP/2 und HTTP/3 gegenübergestellt. Der QUIC-Stack für HTTP/3 zeigt die Integration von TLS direkt ins Protokoll, während beim TCP-Stack für HTTP/2 das TLS_Protokoll noch separat genutzt wird.

Erwähnt werden soll noch, dass QUIC im Gegensatz zu den Standardtransportprotokollen TCP und UDP im User Mode und nicht im Kernel Mode eines Betriebssystems implementiert ist.

Obwohl QUIC formal in der Anwendungsschicht liegt, erfüllt es die Aufgaben eines verbindungsorientierten Transportprotokolls. Die Aufgaben von QUIC sind im Einzelnen:

- Im Sinne der Nachrichtenübertragung gesicherte Ende-zu-Ende Verbindung zwischen zwei Kommunikationspartnern mit garantierter Auslieferung in der richtigen Reihenfolge und unter Vermeidung von Duplikaten
- Multiplexing von QUIC-Streams über UDP (Stream Multiplexing), wobei mehrere nebenläufige Streams (eigenständige Datenströme) über eine QUIC-Verbindung als nicht blockierende, uni- oder bidirektionale Verbindungen unterstützt werden (siehe RFC 9000)
- Security-Maßnahmen mit kryptografischen Funktionen zur Authentifizierung, zur Verschlüsselung und zur Gewährleistung der Integrität über TLS 1.3 (RFC 9001)
- Integrierte Flusskontrolle für einzelne QUIC-Verbindungen und für jeden einzelnen Stream innerhalb einer QUIC-Verbindung
- Integrierte Staukontrolle

Die wichtigsten Aufgaben und Protokollmechanismen werden im Weiteren erläutert.

6.2 QUIC-Nachrichtenaufbau

QUIC unterscheidet Pakete (Packages) und Frames. Ein QUIC-Paket ist eine vollständige Nachricht, die in einem UDP-Datagramm übertragen wird. In einem UDP-Datagramm können auch mehrere QUIC-Pakete übertragen werden. Ein QUIC-Frame ist eine strukturierte Protokollinformation, die in einem QUIC-Paket übertragen wird. In einem QUIC-Paket können auch mehrere Frames unterschiedlichen Typs enthalten sein. In QUIC-Frame werden unter anderem QUIC-Streams übertragen. Eine Übersicht über alle definierten QUIC-Pakete und -Frames findet sich in Tab. 6.2 und 6.5.

6.2.1 QUIC-Nachrichten in UDP-Datagrammen

In der QUIC-Spezifikation wird allgemein für einen Kommunikationspartner der Begriff *Endpunkt* oder *Peer* verwendet. Die konkrete Implementierung von Endpunkten erfolgt in QUIC-Instanzen. Endpunkte agieren zudem noch in der Rolle eines Clients, Servers oder in beiden Rollen. Client und Server tauschen beim Aufbau der Kommunikationsbeziehung und während der Datenübertragungsphase QUIC-PDUs, die im Weiteren gemäß Spezifikation als QUIC-Pakete bezeichnet werden, aus.

Die QUIC-Spezifikation fordert für QUIC-Pakete eine UDP-Mindestnutzlast von 1200 Bytes und eine IP-Paketlänge von mindestens 1280 Bytes. Kleiner dürfen QUIC-Pakete nicht sein. Die IP-Paketlänge ergibt sich aus der minimalen IPv6-Paketlänge, die auch immer mehr von IPv4-Netzen bereitgestellt wird. Eine Begrenzung ist durch die MTU-Größe im Netzwerkpfad gegeben. Werden größere UDP-Datagramme versendet, so schlägt die QUIC-Spezifikation vor, beispielsweise PMTUD (Path Maximum Transmission Unit Discovery) nach RFC 1981, 820 bzw. 8899 zu verwenden, um die in einem Netzwerkpfad zulässige MTU-Größe zu ermitteln.

Abb. 6.2 zeigt die Zusammenhänge. Die QUIC-Nutzdaten, in denen die Frames enthalten sind, und der QUIC-Paket-Header umfassen mindestens 1200 Bytes. Der UDP-Header hat eine Länge von acht Bytes, sodass sich für ein UDP-Datagramm mindes-

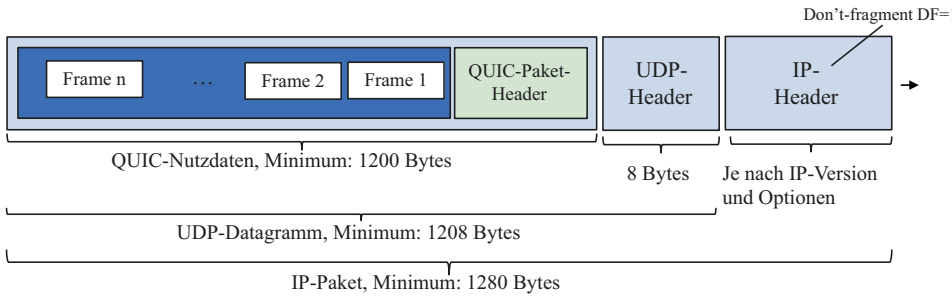


Abb. 6.2 Einbettung von QUIC-Paketen und -Frames in UDP-Datagramme

tens 1208 Bytes ergeben. Für den IP-Header werden 72 Bytes angesetzt, sodass sich für die Mindestgröße eines IP-Pakets, das ein QUIC-Paket enthält, 1280 Bytes ergibt. Eine IP-Fragmentierung ist in QUIC nicht vorgesehen (siehe DF-Bit = 1 im IP-Header; Mandl 2019).

Alle in einem Paket übertragenen Frames werden vollständig verschlüsselt (Abb. 6.3). Nach RFC 9001 werden bei der Übertragung auch Teile des QUIC-Headers wie die Länge der Nutzdaten, die QUIC-Version, die Flags, die Stream-ID, die Bestätigungsnummern und die Paketnummer verschlüsselt. Man spricht hier von Header Protection.

Der Aufbau der Paket-Header von QUIC wird im Weiteren etwas detaillierter betrachtet.

6.2.2 QUIC-Paket-Header

In QUIC sind zwei Headerformate definiert, die als Short und Long Header bezeichnet werden (Abb. 6.4 und 6.5).

QUIC nutzt verschiedenen Paketformate während des Verbindungsaufbaus und während der Übertragung. Ein Long Header wird verwendet, solange noch keine kryptografischen Parameter für die Übertragung von 1-RTT-Paketen eingerichtet sind. Nach dem Verbindungsaufbau werden nur noch Short Header verwendet, da sie weniger Overhead enthalten und daher effizienter sind. Im Long Header können spezielle Pakete, etwa zum Verhandeln der verwendeten Version (Version Negotiation), übertragen werden. Tab. 6.1 erläutert die einzelnen Felder der beiden QUIC-Headertypen, wobei die Bit-Längenangabe „(…)“ ein beliebig langes Feld und die Angabe „0…160“ ein Feld in der Länge zwischen 0 und 160 Bit angibt.

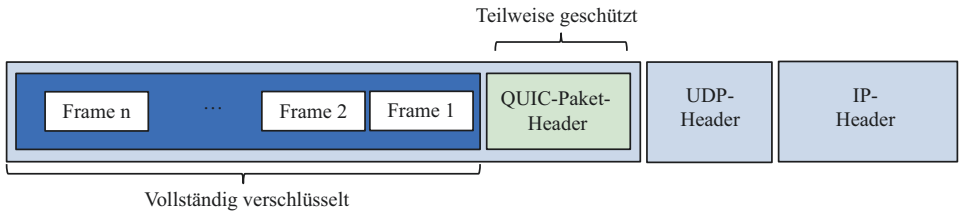


Abb. 6.3 Verschlüsselung der QUIC-Pakete

Header Form (1 Bit)	7 Bits (Fixed Bit, Spin-Bit, Reserved Bit …)
Destination Connection ID (0..160 Bit)	
Packet Number (8..32 Bit)	Packet Payload (mind. 8 Bit)

Abb. 6.4 QUIC Short Header

Header Form (1 Bit)	3 Bits (Fixed Bit, ...) 4 Type-Specific Bits
Version (32 Bit)	
Destination Connection ID Length (8 Bit)	
Destination Connection ID (0..160 Bit)	
Source Connection ID Length (8 Bit)	
Source Connection ID (0..160 Bit)	
Type-Specific Payload (..)	

Abb. 6.5 QUIC Long Header

Tab. 6.1 Felder der QUIC-Header

Feldbezeichnung	Länge in Bit	Bedeutung
Headerform	1	Das erste (höchstwertige Bit des ersten Bytes 0x80 zeigt an, ob es ein Long Header (0b01) oder ein Short Header (0b0) ist.
Bits	7	Die Bits sind wie folgt definiert: • Das Fixed Bit (0x40) gibt an, ob es ein Version-Negotiation-Paket (0b01) oder ein anderes Paket ist. Das Bit muss bei allen anderen Paketen auf 0 stehen. • Long Packet Type zeigt in zwei Bits an (Maske 0x30), um welches QUIC-Paket es sich handelt (Tab. 6.2). Die Codierung ist im RFC 9000 zu finden. • Type-Specific Bits: Die folgenden vier Bits werden individuell je nach Pakettyp genutzt.
Version	32	Angabe der verwendeten QUIC-Protokollversion
Destination Connection ID Length	8	Länge der Connection-ID der Partnerinstanz in Bytes
Destination Connection ID	0..160	Connection-ID der Partnerinstanz
Source Connection ID Length	8	Länge der eigenen Connection-ID in Bytes
Source Connection ID	0..160	Eigene Connection-ID
Packet Number	8..32	Die Paketnummer wird nur im Short Header verwendet und identifiziert die Pakete innerhalb einer Verbindung eindeutig. Paketnummern werden auch für die Bestätigung verwendet.
Typspezifische oder Packet Payload	(...)	Nutzdaten

Tab. 6.2 QUIC-Paketformate und deren Nutzung

QUIC-Paket	Headertyp	Beschreibung	Benutzt durch
Version Negotiation Packet	Long Header Packet	Ermöglicht einem Server anzuzeigen, dass er die QUIC-Version des Clients nicht unterstützt	Server
Initial Packet	Long Header Packet	Überträgt den ersten CRYPTO-Frame vom Client zum Server zum Schlüsselaustausch	Client
0-RTT Packet	Long Header Packet	Überträgt erste Daten vom Client zum Server vor dem Beenden des Handshakes	Client
Handshake Packet	Long Header Packet	Überträgt kryptografische Handshake-Nachrichten und Bestätigungen vom Server zum Client	Server
Retry Packet	Long Header Packet	Dient der Anzeige des Servers an den Client, eine Adressvalidierung durchzuführen, wobei ein Retry Token übertragen wird, das der Client im Folgenden benutzen muss	Server
1-RTT Packet	Short Header Packet	Dient u. a. der Übertragung von Stream-Frames und ACK-Frames (Bestätigungen)	Client, Server

QUIC-Paketformate

In QUIC sind verschiedene Paketformate definiert. In Tab. 6.2 werden sie zusammen mit deren Nutzung in der Kommunikation kurz erläutert. Je nach Paketformat wird entweder ein Short oder ein Long Header benutzt. In der Tabelle wird auch angegeben, ob die QUIC-Paketformate durch den Client oder den Server oder durch beide Endpunkte der Kommunikation verwendet werden.

Beispielpaket: 1-RTT

Der Aufbau des 1-RTT-Pakets soll im Folgenden exemplarisch so gezeigt werden, wie er im RFC 9000 notiert ist. Die Zahlen in Klammern geben jeweils die Feldlängen in Bit an:

```
1-RTT Packet {
    Header Form (1) = 0,
    Fixed Bit (1) = 1,
    Spin Bit (1)
    Reserved Bits (2),
    Key Phase (1),
    Packet Number Length (2),
    Destination Connection ID (0..160),
    Packet Number (8..32),
    Packet Payload (8..),
}

Packet Payload {
    Frame (8..) ...,
}

Frame {
    Frame Type (i), Type-Dependent Fields (..),
}
```

Tab. 6.3 Felder des 1-RTT-Pakets

Feldbezeichnung	Typ	Bedeutung
Header Form	Bit	Fester Wert 0b0 gibt an, dass hier ein Short Header verwendet wird
Fixed Bit	Bit	Nur, wenn in diesem Bit 0b1 steht, ist das Paket gültig
Spin Bit	Bit	Latenz-Spin-Bit: Wenn es auf 0b1 gesetzt ist, darf die Latenz der Verbindung überwacht werden
Reserved Bit	Bit	Zwei reservierte Bits
Packet Number Length	Bit	Länge der Paketnummer
Key Phase	Bit	Gibt an, dass ein Empfänger die Paketschutzschlüssel verwenden darf (TLS)
Destination Connection ID	Bit	Eindeutige Verbindungsidentifikation
Packet Number	Bit	In der Verbindung eindeutige Paketnummer
Packet Payload	Bit	Nutzdaten, die im Paket als QUIC-Frames übertragen werden (siehe Datenstruktur *Packet Payload*)

In Tab. 6.3 werden die Felder eines 1-RTT-Pakets beschrieben.

Die Nutzdaten im Feld *Packet Payload* enthalten beispielsweise STREAM-Frames. Ein Frame beginnt mit einem *Frame Type* gefolgt von framespezifischen Inhalten und hat eine Mindestlänge von 8 Bit. ◀

QUIC-Transportparameter

Über definierte Transportparameter ist es in QUIC möglich, die Kommunikation zwischen QUIC-Endpunkten zu beeinflussen. Sie können während des Verbindungsaufbaus, also des TLS-Handshakes, der noch erläutert wird, eingestellt werden.

Transportparameter werden jeweils als Tripel in der folgenden Form übertragen:

```
Transport Parameter {
    Transport Parameter ID (i),
    Transport Parameter Length (i),
    Transport Parameter Value (..),
}
```

In Tab. 6.4 sind einige dieser Parameter exemplarisch beschrieben. Die Ids geben die hexadezimale Schreibweise der bei der Übertragung genutzten Identifikatoren an. ◀

QUIC-Frames

In Tab. 6.5 sind alle derzeit im RFC 9000 spezifizierten QUIC-Frames mit jeweils einer kurzen Beschreibung und den bei der Übertragung verwendeten Identifikationscodes (Ids) in Hexadezimalschreibweise zusammengefasst. Aus den Bezeichnungen kann die Funktion der einzelnen Frames abgeleitet werden. Beispiele:

- Der PING-Frame dient dem Überprüfen, ob ein Partnerendpunkt noch verfügbar ist.
- Mit dem STREAM-Frame wird ein neuer Stream erzeugt, und es werden Daten über einen Stream gesendet.
- Das ACK-Frame dient der Bestätigung des Empfangs eines QUIC-Pakets.

Tab. 6.4 Beispiele für QUIC-Transportparameter

QUIC-Frame	Beschreibung	Ids in hex
original_destination_connection_id	Destination-Connection-ID des ersten Verbindungsaufbaupakets	0x00
max_idle_timeout	Maximale Idle-Zeit bis zum Timeout einer Verbindung in ms	0x01
max_udp_payload_size	Maximal erlaubte UDP-Nutzdatenlänge in Bytes, Standard: 65527 wie UDP-Datagramm-Nutzdatenlänge, Minimum: 1200 Bytes	0x02, 0x03
initial_max_data	Initiale maximale Paketlänge für eine Verbindung	0x04
initial_max_stream_data_*	Initialer Sendekredit für Streams (Flusskontrolle), je nach Streamtyp, z. B. *initial_max_stream_data_uni* für unidirektionalen Stream	0x05
max_ack_delay	Maximale Verzögerungszeit für eine Bestätigung, Standard: 25 ms	0x06
disable_active_migration	Die Migration einer Verbindung wird nicht unterstützt	0x0c

Tab. 6.5 QUIC-Frames

QUIC-Frame	Beschreibung	Ids in hex
PADDING-Frame	Keine Semantik, dient nur zum Auffüllen von Paketen, falls es erforderlich, die Mindestlänge zu erreichen Beispiel: Initial Packet	0x00
PING-Frame	Prüfen, ob Partnerendpunkt noch lebt oder erreichbar ist, Empfänger muss bestätigen	0x01
ACK-Frame	Bestätigung des Empfangs von Paketen in sog. Ranges (mehrere Pakete auf einmal bestätigen)	0x02, 0x03
RESET_STREAM-Frame	Sofortige Beendigung einer Sendeseite eines Streams	0x04
STOP_SENDING-Frame	Endpunkt (auch Peer genannt) beendet den Empfang in einem Stream und verwirft ankommende Pakete	0x05
CRYPTO-Frame	Übertragung kryptografischer Handshake-Pakete	0x06
NEW_TOKEN-Frame	Serverendpunkt sendet dem Clientendpunkt ein Token für zukünftige Verbindungen	0x07
STREAM-Frame	Erzeugen eines Streams (beim ersten Senden) und Übertragen von Stream-Daten	0x08..0x0f
MAX_DATA-Frame	Partnerendpunkt wird informiert, wie viele Daten er über die Verbindung senden darf	0x10
MAX_STREAM_DATA-Frame	Partnerendpunkt wird informiert, wie viele Daten er über einen Stream senden darf	0x11
MAX_STREAMS-Frame	Endpunkt informiert einen Partnerendpunkt, wie viele Streams er während einer Verbindung öffnen darf	0x12, 0x13
DATA_BLOCKED-Frame	Endpunkt möchte Daten senden, kann aber nicht, weil Verbindungsflusskontrolle es nicht zulässt	0x14
STREAM_DATA_BLOCKED-Frame	Endpunkt möchte Daten senden, kann aber nicht, weil Flusskontrolle des Streams es nicht zulässt	0x15
STREAMS_BLOCKED-Frame	Endpunkt möchte einen Stream öffnen, hat aber das Maximum erlaubter Streams erreicht	0x186, 0x17

(Fortsetzung)

Tab. 6.5 (Fortsetzung)

QUIC-Frame	Beschreibung	Ids in hex
NEW_ CONNECTION_ID-Frame	Der Endpunkt sendet eine alternative Connection-ID für die Migration von Verbindungen	0x18
RETIRE_ CONNECTION_ID-Frame	Endpunkt signalisiert, dass eine Connection-ID nicht länger genutzt wird	0x19
PATH_CHALLENGE-Frame	Endpunkt überprüft die Erreichbarkeit seines Partnerendpunktes (Pfadüberprüfung)	0x1a
PATH_RESPONSE-Frame	Antwort auf den PATH-Challenge Frame zur Erreichbarkeitsprüfung	0x1b
CONNECTION_ CLOSE-Frame	Endpunkt signalisiert einen Verbindungsabbau, alle Streams der Verbindung werden geschlossen	0x1c, 0x1d
HANDSHAKE_ DONE-Frame	Serverendpunkt bestätigt den Handshake an den Clientendpunkt	0x1e

6.3 QUIC-Verbindungen, Streams und Multiplexing

Bei TCP ist eine Verbindung im Netzwerk eindeutig über zwei Socket Pairs definiert. Bei QUIC erfolgt die Verbindungsidentifikation eindeutig über Connection-IDs (CIDs) je Endpunkt, wobei die CID-Länge beim Verbindungsaufbau vereinbart wird und standardmäßig acht Bytes lang ist. Ein Wechsel der CIDs und Nutzung mehrerer CIDs während einer Verbindung sind möglich. Durch die Einführung einer CID wird eine Entkopplung von den darunterliegenden Adressen (UDP-Port, IP-Adresse) erreicht.

Bei QUIC ist es sogar möglich, dass die Verbindung eines Clients (Browsers) zu einem Server unterbrochen und mit einem anderem Server aufgebaut wird, ohne dass die Anwendung den Verbindungswechsel bemerkt. Man spricht hier von einem „Reestablishment“ der Verbindung.

Anwendungsprotokolle, die QUIC nutzen, tauschen ihre Informationen innerhalb von Verbindungen über QUIC-Streams (kurz: Stream) aus. Ein Stream ist aus Sicht der Anwendung ein uni- oder bidirektionaler und geordneter Bytestrom. Unidirektionale Streams ermöglichen das Senden in eine Richtung, bidirektionale Streams erlauben beiden Endpunkten das Senden von Daten – initiiert entweder durch den Client oder durch den Server. Innerhalb der Verbindung ist der Stream eindeutig durch eine Stream-ID gekennzeichnet. Für einen einzelnen Stream wird die richtige Reihenfolge garantiert, gesendete Bytes kommen also beim Empfänger genau so an, wie sie abgesendet wurden. Dies gilt aber nicht streamübergreifend.

In einer QUIC-Verbindung können mehrere, völlig unabhängige Streams erzeugt werden, in denen Daten nebenläufig übertragen werden. Die Position im Stream wird durch sogenannte Offsets markiert. Für jeden Stream wird auch eine Flusskontrolle durchgeführt. Im Vergleich dazu unterstützt TCP nur einen einzigen bidirektionalen Datenstrom über eine Verbindung. Durch die unabhängige Übertragung der Streams einer Verbindung

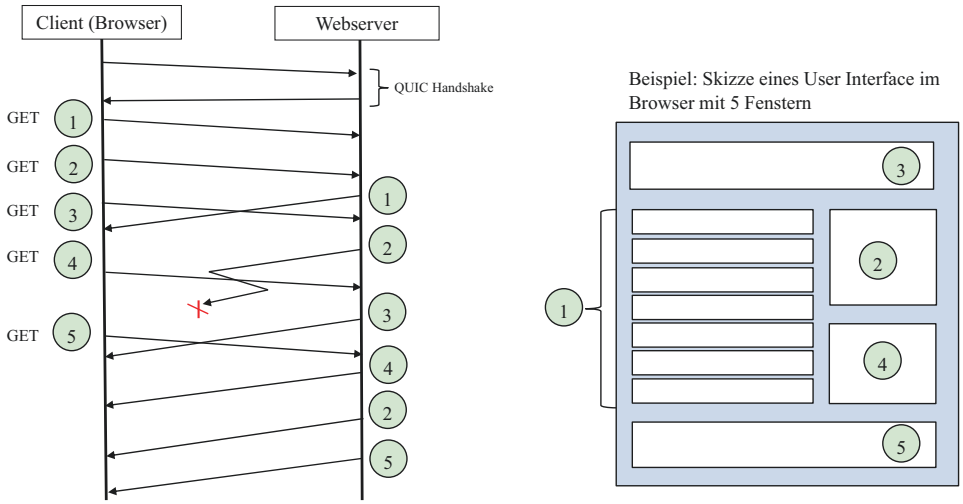


Abb. 6.6 Beispiel zur Nutzung von QUIC-Streams in HTTP/3

kann es nicht wie bei TCP-basierter Kommunikation dazu kommen, dass sich Streams gegenseitig blockieren. Das bereits erwähnte HoL-Blocking kann also nicht vorkommen.

Der Zusammenhang ist in Abb. 6.6 skizziert: In einem browserbasierten User Interface werden nach einem Verbindungsaufbau zwischen einem Browser und einem Webserver parallel fünf Fenster angezeigt, die z. B. Text, Audio-, Videoströme oder Bilder enthalten können. Jedem dieser Fenster kann mit QUIC ein eigener Stream zugeordnet werden. Wenn der Browser das Anwendungsprotokoll HTTP/3, das auch QUIC aufsetzt, verwendet, wird für jedes Fenster ein eigener Stream genutzt. Die Daten werden jeweils über einen eigenen HTTP-Request vom Webserver angefordert. Bei Stream 2 geht eine Nachricht verloren. Da die Streams parallel übertragen, hat dies keine Auswirkungen auf die anderen Streams. Das bedeutet, dass zwar das Fenster 2 etwas langsamer mit Daten versorgt wird, aber die anderen Fenster ohne Verzögerung befüllt werden.

Wie in QUIC HoL-Blocking vermieden wird, wir in Abb. 6.7 und 6.8 nochmals veranschaulicht. Abb. 6.7 zeigt das HoL-Blocking-Problem bei HTTP/2. Eine TCP-Verbindung zwischen zwei der beteiligten TLS/TCP-Instanzen unterstützt zwar bei HTTP/2 parallele Streams, aber die Daten mehrerer Streams werden in TCP-Segmenten (TCP-PDUs) zusammengefasst. Bei einem TCP-Segmentverlust blockieren dann alle Streams der Verbindung, bis eine erneute Übertragung eines fehlenden TCP-Segments erfolgreich war. Um dies zu vermeiden, kann man bei HTTP/2 mehrere TCP-Verbindungen für eine logische Kommunikation zwischen Browser und Webserver nutzen, für jeden Stream auf der HTTP-Ebene eine eigene. Dieses Vorgehen ist aber nicht sehr ressourcenschonend.

Bei QUIC werden – wie in Abb. 6.8 gezeigt – Streams parallel und unabhängig voneinander in einer QUIC-Verbindung übertragen. Wie man in der Abbildung erkennen kann, werden die QUIC-PDUs als QUIC-Pakete bezeichnet, die wiederum aus QUIC-

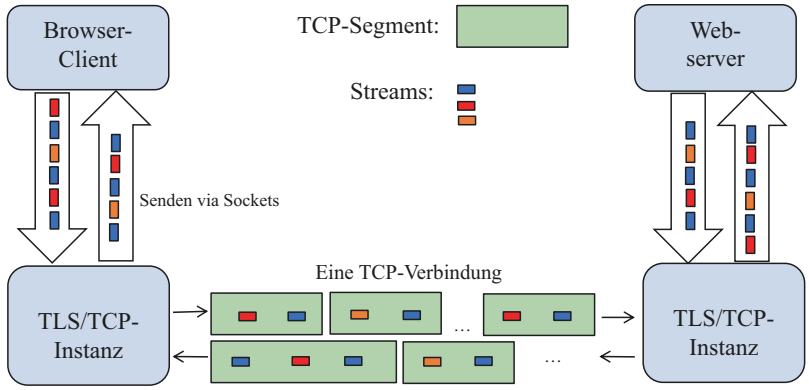


Abb. 6.7 HoL-Blocking bei HTTP/2

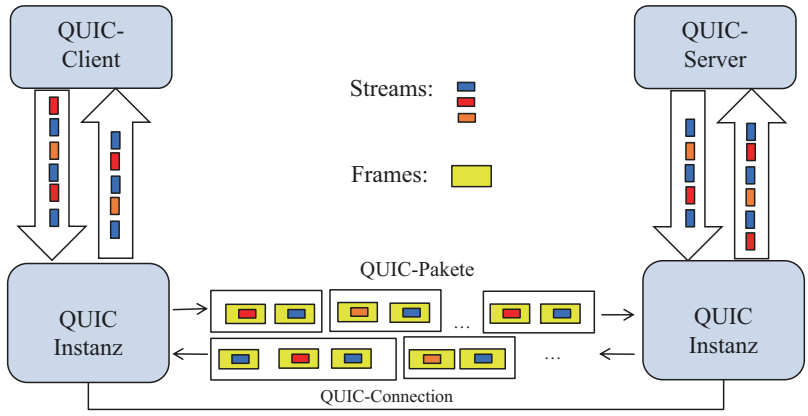


Abb. 6.8 Vermeidung von HoL-Blocking bei HTTP/3

Frames aufgebaut sind (Tab. 6.2 und 6.5). Die Stream-Daten werden in speziellen Stream-Frames übertragen. Wenn ein QUIC-Paket verloren geht, blockieren nur die Streams, für die in dem Paket Stream-Frames übertragen wurden.

6.4 Verbindungsmanagement

Im Folgenden betrachten wir die Besonderheiten beim QUIC-Verbindungsauf- und abbau.

6.4.1 Verbindungsaufbau

QUIC baut eine gesicherte und auch verschlüsselte Ende-zu-Ende-Verbindung zwischen den beteiligten Kommunikationspartnern auf. Nutzdaten und auch Teile des QUIC-Paket-Headers werden verschlüsselt übertragen. In der QUIC-Version 1 ist TLS 1.3 bereits in

QUIC integriert und sorgt für die Authentifizierung der Serverseite und die Verschlüsselung der Nachrichten. Eine Authentifizierung der Clientseite ist optional in der QUIC-Spezifikation verankert (RFC 9000). Ob zukünftige QUIC-Versionen auch TLS nutzen, ist nicht festgelegt.

Beim Verbindungsaufbau über den QUIC-Handshake werden die folgenden kryptografischen Daten ausgetauscht:

- Serverauthentifizierung
- Clientauthentifizierung (optional)
- Eindeutige Schlüssel je Verbindung

Der Austausch von Nachrichten erfolgt über QUIC-Pakete, die wiederum fest definierte Frames enthalten (Tab. 6.2 und 6.5).

Vergleicht man QUIC mit TCP, so ist beim QUIC-Verbindungsaufbau eine deutliche Nachrichteneinsparung zu erkennen. Beim TCP-Verbindungsaufbau mit anschließendem TLS-Handshake sind drei Roundtrips erforderlich, bei QUIC mit TLS 1.3 ist nur noch maximal ein Roundtrip notwendig, bevor Nutzdaten gesendet werden können. Abb. 6.9 zeigt den vereinfachten QUIC-Verbindungsaufbau im Vergleich zum TCP-Verbindungsaufbau mit anschließendem TLS-Handshake. Während QUIC mit höchstens zwei Nachrichten auskommt, bevor Nutzdaten gesendet werden können, benötigt TCP mit TLS (TCP-TLS) sechs Nachrichten.

Beim QUIC-Handshake sendet der Client zunächst ein Initial-Paket mit einem Crypto-Frame, der kryptografische Daten für den TLS-Handshake enthält. Der Server bestätigt mit einem Initial- und einem Handshake-Paket, die jeweils auch kryptografische Daten (Zertifikate, Signaturen, gemeinsamer Sitzungsschlüsssel = Key Share)

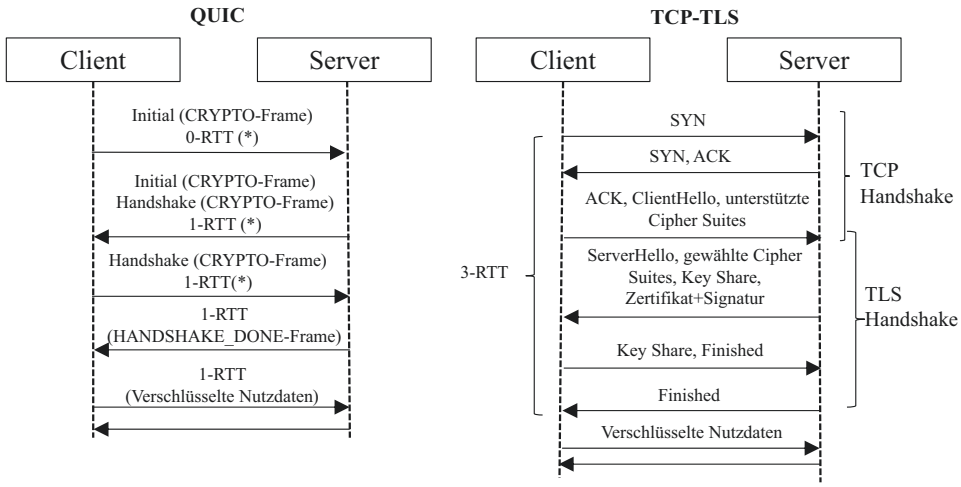


Abb. 6.9 Vereinfachter Verbindungsaufbau mit 1-RTT im Vergleich zum TCP-TLS-Verbindungsaufbau

enthalten. Der Austausch der kryptografischen Parameter und der Ablauf des TLS-Handshakes sind in der TLS-Spezifikation erläutert (RFC 8446). Eine kurze Erläuterung befindet sich auch in Kap. 5.

Man spricht von Zero Roundtrip (0-RTT), wenn der Client bereits in seiner ersten Nachricht Nutzdaten sendet. Auch der Server kann in seiner ersten Nachricht bereits ein 1-RTT-Paket mit Nutzdaten übertragen. In der dritten Nachricht des Handshakes sendet der Client nochmals ein Handshake-Paket zur Bestätigung der kryptografischen Daten. Daraufhin schließt der Server den Handshake mit einem 1-RTT-Paket, in dem ein HANDSHAKE-DONE-Frame übertragen wird, ab. Danach erfolgt der reguläre Austausch der verschlüsselten Nutzdaten über 1-RTT-Pakete. In Abb. 6.9 sind die Stellen, in denen bereits Nutzdaten gesendet werden können, mit (*) gekennzeichnet. Die Bezeichnungen *1-RTT* und *0-RTT* stammen aus der Spezifikation für TLS 1.3. 0-RTT (Zero Roundtrip) bedeutet, dass vor den ersten Nutzdaten kein weiterer Roundtrip erforderlich ist. 1-RTT erfordert einen Roundtrip vor dem Senden der Nutzdaten.

Bei TCP-TLS ist der Verbindungsaufbau aufwendiger. Zunächst erfolgt der klassische TCP-Drei-Wege-Handshake, wobei in der dritten Nachricht aus TCP-Sicht bereits die ersten Nutzdaten übertragen werden, womit der TLS-Handshake eingeleitet wird. Der Austausch und die Vereinbarung der zu verwendenden kryptografischen Daten erfordern danach drei weitere Nachrichten, bevor mit der eigentlichen Übertragung von verschlüsselten Nutzdaten begonnen werden kann.

Der Ablauf mit Zero Roundtrip ist dann möglich, wenn Verschlüsselungsparameter aus einer vorherigen Verbindung (in TLS auch Session genannt) bekannt sind, die wiederverwendet werden. Allerdings ist damit keine vollständige Forward Secrecy gewährleistet (siehe Hintergrundinformation zu *Forward Secrecy* in Kap. 5). 0-RTT muss in QUIC-Implementierungen nicht zwingend umgesetzt werden.

6.4.2 Verbindungsabbau

Im RFC 9000 spricht man von einem Immediate Close, wenn ein Endpunkt eine Verbindung zügig schließen möchte. In diesem Fall sendet ein Endpunkt einen CONNECTION_CLOSE-Frame an den Partner. Das Schließen der Verbindung wird üblicherweise durch die Anwendung initiiert, sinnvollerweise nachdem vorher im Anwendungsprotokoll ein geordneter Abschluss der Kommunikation (graceful shutdown) erfolgte.

Der Endpunkt nimmt den Zustand *closing* ein. Alle noch offenen Streams werden implizit geschlossen. Das bedeutet, die Anwendung kann über die Verbindung nichts mehr senden. Der Endpunkt wartet noch eine bestimmte Zeit auf Pakete, die eventuell noch unterwegs sind, und soll laut RFC 9000 auch das Senden von Antworten auf diese Nachrichten etwa durch Verzögerungen einschränken. Antworten auf noch ankommende Pakete enthalten einen CONNECTION_CLOSE-Frame, um den Partner darüber zu informieren, dass die Verbindung bereits abgebaut wird. Nach einer ausreichenden Wartezeit wird die Verbindung schließlich geschlossen und der Verbindungszustand gelöscht.

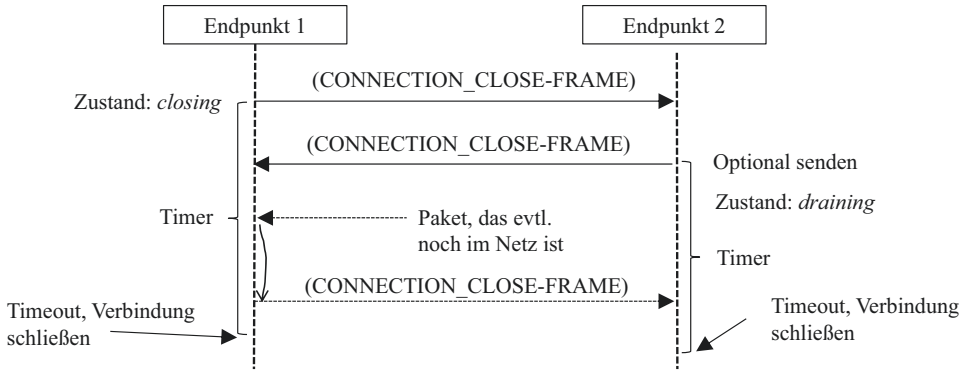


Abb. 6.10 Vereinfachter QUIC-Verbindungsabbau

Wenn ein Endpunkt einen CONNECTION_CLOSE-Frame empfängt, kann er ebenfalls mit einem CONNECTION_CLOSE-Frame antworten, aber auch noch anstehende Nachrichten senden. Der Endpunkt geht in den Zustand *draining* über, schließt alle Streams sowie nach einer Wartezeit die Verbindung und verwirft die Kontextinformationen dazu. Der Ablauf des Verbindungsabbaus ist vereinfacht in Abb. 6.10 skizziert. Hier empfängt der initiierende Endpunkt vor dem endgültigen Abschließen der Verbindung noch eine Nachricht des Partners aus dem Netz, die er wiederum mit einem CONNECTION_CLOSE-Frame beantwortet.

Wenn beide Endpunkte ein CONNECTION_CLOSE-Frame empfangen, können auch beide in den Zustand *draining* übergehen, Die Wartezeit bis zum endgültigen Schließen der Verbindung ist aber dann wie im Zustand *closing*. In diesem Zustand darf ein Endpunkt keine weiteren Pakete mehr senden. Abschließend verwerfen die Endpunkte die Kontextinformationen der Verbindung.

Es ist auch möglich, dass eine Verbindung bereits während eines Handshakes beim Verbindungsaufbau durch das Senden eines CONNECTION_CLOSE-Frames durch einen Endpunkt geschlossen werden kann (siehe RFC 9000).

Neben dem regulärem Abbau kann die Verbindung auch sofort abgebaut werden, wenn von keinem der beiden Endpunkte über eine bestimmte Zeit Nachrichten übertragen wurden und auch keine Bestätigungen mehr fehlen. In diesem Fall spricht man von einem Idle Timeout. Für diese Zwecke wird ein Timer verwaltet, der jeweils nach dem Empfang und der Verbreitung einer Nachricht jeweils wieder zurückgesetzt wird. Der Wert des Timers ist in einem QUIC-Transportparameter mit der Bezeichnung *max_idle-timeout* (Tab. 6.4), der zwischen den Partnern vereinbart wird, festgelegt.

Um Idle Timeouts und damit unnötige Beendigungen von Verbindungen möglichst zu vermeiden, können Endpunkte PING-Frames an die Partner senden, um festzustellen, ob sie noch aktiv sind. PING-Frames werden vom Partner ebenfalls mit einem PING-Frame beantwortet.

6.4.3 Verbindungsmigration

Eine Endpunktadresse ist in internetbasierten Netzen üblicherweise durch das Tupel (IP-Adresse, Portnummer) gegeben. Die Nutzung einer Connection-ID entkoppelt im Gegensatz zu TCP die Verbindungsidentifikation von den Endpunktadressen. Diese Entkopplung ermöglicht die Migration der QUIC-Verbindung derart, dass ein Endpunkt während einer bestehenden Verbindung auf einen anderen Host wechseln kann. Eine Migration kann auch durch Einflüsse erforderlich sein, die ein Client nicht zu verantworten hat. Dies kann insbesondere dann erfolgen, wenn eine Middlebox wie ein NAT-Server eine Adressänderung durchführt.

Ebenso ist es in QUIC möglich, dass ein Server nach dem Verbindungsaufbau eine neue bevorzugte Adresse bekannt gibt. Der Client entscheidet aber, ob er sie annimmt oder nicht. Die Entscheidung für eine Verbindungsmigration trifft also nicht der Server, er kann nur einen Vorschlag für einen neue Adresse machen. Die Partner müssen die neue Adresse bei einer Migration stets überprüfen (validieren).

Eine Migration kann beim Verbindungsaufbau auch per Transportparameter verboten werden (*disable_active_migration*; Tab. 6.4).

6.5 Datenübertragungsphase

Für die Übertragung von Nutzdaten verwendet QUIC eindeutige Paketnummern in der Form von monoton aufsteigenden Zählern je Verbindung. Paketnummern werden in einer Verbindung nie wiederholt, es gibt daher im Unterschied zu TCP keine Zweideutigkeit der Paketidentifikation bei Wiederholungen. In Bestätigungsnachrichten (ACK) wird der Empfang von QUIC-Paketen unter Angabe der zugehörigen Paketnummern kommuniziert.

ACK Ambiguity bzw. ACK Confusion

Bei TCP kann es durchaus Zweideutigkeiten bei der Kennzeichnung von TCP-Segmenten geben, da die Bestätigung von empfangenen TCP-Segmenten byteweise unter Nutzung von Sequenznummer und Bestätigungsnummer (siehe TCP-Header) erfolgt und TCP die Sequenznummer des ursprünglichen Segments bei erneutem Senden wiederholt.

Bei TCP kann beim Sender daher nicht unterschieden werden, ob eine Bestätigung für das ursprüngliche TCP-Segment oder für das im Sinne der Nachrichtenwiederholung erneut gesendete TCP-Segment empfangen wird. Dies hat eventuell Auswirkungen auf die Berechnung des Retransmission Timouts (RTOs).

Anwendungsnachrichten werden in QUIC als Streams innerhalb vom QUIC-Paketen übertragen. Als Streams werden geordnete Byte-Ströme bezeichnet.

6.5.1 Erzeugen und Schließen von Streams

Streams, die von einem Endpunkt erzeugt werden, sind über eine eindeutige Stream-ID mit einem 62 Bit langem Integerwert gekennzeichnet. In den ersten beiden Bits der Stream-ID ist codiert, ob ein Stream vom Client oder vom Server erzeugt wurde und ob er bidirektional oder unidirektional ist. Der binäre Wert 0b00 bedeutet z. B., dass der Stream vom Client initiiert wurde und bidirektional ist, 0b01 ist ein bidirektionaler Stream, den der Server erzeugt hat. Eine Stream-ID darf in einer Verbindung nicht mehrfach oder erneut genutzt werden.

Während bei TCP je Verbindung nur ein bidirektionaler Byte-Stream erzeugt wird, können bei QUIC mehrere Streams in einer Verbindung parallel genutzt werden. Für die Nutzung von Streams sind Funktionen zum Senden, Abschließen, Zurücksetzen, zum Lesen aus einem Stream und zum Abbruch eines Streams vorgesehen. Eine exakte API-Vorgabe für die Implementierung ist nicht verfügbar. In der QUIC-Spezifikation ist keine API wie die Socket API definiert, über die Verbindungen aufgebaut und abgebaut sowie Daten über einen Stream gesendet und empfangen werden können.

Für die Erzeugung von Streams, das Senden von Daten über Streams und für den Austausch von Stream-Parametern werden spezielle Frames wie STREAM, MAX_STREAMS, MAX_STREAM_DATA, STOP_SENDING und RESET_STREAM genutzt. Alle zum Anlegen und Entfernen von Streams verwendeten Frames enthalten die dem Stream zugeordnete Stream-ID zur eindeutigen Kennzeichnung.

Streams werden beim ersten Senden von Daten über einen STREAM-Frame implizit erzeugt. Die Endpunkte richten daraufhin die erforderlichen Empfangspuffer ein. Bei unidirektionalen Streams legt nur der Empfänger einen Empfangspuffer an, bei bidirektionalen Streams beide Endpunkte.

Innerhalb einer Verbindung können die beiden Kommunikationspartner über die Nutzung eines MAX_STREAMS-Frames vereinbaren, wie viele Streams maximal erzeugt werden dürfen, und über einen weiteren Frame mit der Bezeichnung MAX_STREAM_DATA, wie viele Nutzdatenbytes maximal in einem Stream gesendet werden dürfen. Ebenso kann vereinbart werden, wie viele Nutzdatenbytes maximal über die gesamte Verbindung gesendet werden dürfen. Diese und weitere Parameter werden als Transportparameter (Tab. 6.4) bezeichnet. Sie können beim Verbindungsaufbau oder während einer aktiven Verbindung ausgehandelt werden.

Beide Endpunkte können einen Stream auch jederzeit beenden, indem sie – ähnlich wie beim TCP-Verbindungsabbau – ein FIN-Flag im STREAM-Frame mitsenden. Mit dem STOP_SENDING-Frame kann ein Empfänger auch bekannt geben, dass er die Daten des Streams nicht mehr benötigt, wobei hier sogar Pakete verworfen werden können. Der Partner antwortet mit einem RESET_STREAM-Frame und beendet das Senden über den Stream sofort. Dieser Frame kann auch benutzt werden, wenn die Sendeseite ihrerseits das Senden über einen Stream beenden möchte.

6.5.2 Nachrichtenübertragung, Bestätigung und Sendungswiederholung

Die gesicherte Übertragung von Nutzdaten erfolgt bei QUIC also über Streams. Hierzu werden STREAM-Frames für das Senden und ACK-Frames für die Empfangsbestätigung verwendet. Eine Verzögerung der Bestätigungen ist wie bei TCP zugelassen. Damit sind auch kumulative Bestätigungen möglich.

Der Aufbau eines STREAM-Frames soll exemplarisch diskutiert werden:

```
STREAM Frame {
    Type (i) = 0x08..0x0f
    Stream ID (i),
    [Offset (i)],
    [Length (i)],
    Stream Data (..),
}
```

Die Notation in RFC 9000 sieht vor, dass Integerfelder mit (i) und beliebig lange Felder ohne Typangabe mit (..) gekennzeichnet werden. Die Felder des STREAM-Frames sind in Tab. 6.6 zusammengefasst. Mit dem Feld *Offset* wird die aktuelle Position im Stream angezeigt. Sender und Empfänger tauschen hierüber die Positionsinformation über die im Feld *Stream Data* gesendeten Daten aus. Das erste Byte im Stream hat den Offsetwert 0.

Die Empfangsbestätigung erfolgt über das Senden von ACK-Frames. Wie bei TCP sind kumulative Bestätigungen und auch verzögerte Bestätigungen möglich. Es werden nicht nur Stream-Daten bestätigt, sondern generell alle QUIC-Pakete. Exemplarisch soll die Bestätigung von Stream-Daten diskutiert werden. Pakete werden, wie bereits beim QUIC-Paketaufbau erläutert, in einer Verbindung über monoton aufsteigende Paketnummern identifiziert. Entsprechend erfolgt die Bestätigung anhand der Paketnummern.

Tab. 6.6 Felder des STREAM-Frames

Feldbezeichnung	Typ	Bedeutung
Type	Integer	0x08..0x0f, binäre Form: 0b00001XXX. Die letzten drei Bit geben Folgendes an: • Das OFF-Bit (0x04) bedeutet, dass das Offset-Feld einen gültigen Wert hat. • Das LEN-Bit (0x02) zeigt an, dass das Length-Feld mit einen gültigen Wert belegt ist • Das FIN-Bit (0x01) zeigt an, dass der Stream beendet wird
Stream ID	Integer	Variabel langes Feld, das eine eindeutige Stream-ID enthält
Offset	Integer	Variabel langes und optionales Feld, das die aktuelle Position des zuletzt gesendeten Bytes im Stream angibt
Length	Integer	Variabel langes und optionales Feld zur Angabe der Länge der im Stream gesendeten Daten
Stream Data	Bytes	Beliebig lange Nutzdaten (begrenzt durch MTU usw.), die über den Stream-Frame gesendet bzw. ausgeliefert werden

Tab. 6.7 Felder des ACK-Frames

Feldbezeichnung	Typ	Bedeutung
Type	Integer	0x02..0x03, mit 0x03 = Kumulative Quittierung
Largest Acknowledged	Integer	Variabel langes Feld, das die höchste Paketnummer angibt, die mit diesem ACK-Frame bestätigt wird
ACK Delay	Integer	Variabel langes Feld, das eine Bestätigungsverzögerung in Mikrosekunden angibt
ACK Range Count	Integer	Variabel langes Feld zur Angabe der ACK-Bereiche, die im Folgenden angegeben werden.
First ACK Range	Integer	Variabel langes Feld, das die erste Paketnummer in dem Bestätigungsbereich anzeigt. Damit wird in Verbindung mit dem Feld *Largets Acknowledged* ein zusammenhängender Paketbereich angegeben, der über den ACK-Frame bestätigt wird (Largest Acknowledged – First ACK Range).
ACK Range		Weitere Bereiche von bestätigten Paketen oder Paketbereichen, die von der Bestätigung explizit ausgenommen werden (Gaps)
ECN Counts	Bytes	Die Zähler geben an, wie viele Pakete sog. ECN-Markierungen erhalten haben. Diese Markierungen deuten an, dass die Pakete möglicherweise Informationen über mögliche Netzwerkengpässe aus dem Netzwerk erhalten haben. Diese Informationen können für die Staukontrolle verwendet werden. IP-Router können die Überlastinformationen im IP-Header in zwei Bits übertragen. Hier spricht man von ECT(0) und ECT(1) Codepoints. Wenn beide Codepoints gesetzt sind, spricht man auch von ECT-CE (Congestion Experienced). Die genaue Nutzung der Bits ist nicht immer gleich und soll hier nicht weiter vertieft werden.

Der nachfolgende Programmcode zeigt den Aufbau eines ACK-Frames mit den darin verwendeten Datenstrukturen. Tab. 6.7 fasst die Feldbeschreibung des ACK-Frames zusammen. Die Bestätigungen erfolgen paketweise. Gesendete Bestätigungen sind nicht mehr umkehrbar. Im ACK-Frame können mehrere Pakete, sogar ganze Bereiche von Paketen, die bestätigt werden sollen, angegeben werden. Im Feld ACK Range wird mit der Notation … angezeigt, dass beliebig viele Bereiche angegeben werden können. Eine Begrenzung ist nur durch die MTU-Größe gegeben. Es können auch Paketbereiche explizit von der Bestätigung ausgeschlossen werden. Hier spricht man von *Gaps*, die über die Datenstruktur *ACK Range* beschrieben werden können.

```
ACK Frame {
    Type (i) = 0x02..0x03,
    Largest Acknowledged (i),
    ACK Delay (i),
    ACK Range Count (i),
    First ACK Range (i),
    ACK Range (..), ...,
    [ECN Counts (..)],
}
```

```
ACK Range {
    Gap (i),
    ACK Range Length (i),
}

ECN Counts {
    ECT0 Count (i),
    ECT1 Count (i),
    ECN-CE Count (i),
}
```

Nutzung eines ACK-Frames

Das Zusammenspiel beim Senden und Bestätigen von Stream-Daten ist in Abb. 6.11 in einem einfachen Beispiel angedeutet. Zwei Pakete mit Stream-Daten werden vom Endpunkt 1 an den Endpunkt 2 gesendet und vom Endpunkt 2 gemeinsam (kumulativ) bestätigt. Hier handelt es sich um das Paket mit der höchsten Paketnummer 11 (LargestACK) gefolgt von einem Paket 10 (FirstACKRange). ◄

Nutzung von ACK-Bereichen (ACK Range)

Abb. 6.12 zeigt die Bestätigung von ganzen Bereichen und auch die Anzeige von Paketnummernbereichen, die nicht bestätigt werden. Im ACK-Frame werden in absteigender Reihenfolge die Paketnummern 10, 9, 6, 5, 4 und 3 bestätigt. Nicht bestätigt werden explizit die Pakete mit den Nummern 8, 7, 2 und 1.
Die Listenelemente zählen die Paketnummern von der höchsten bestätigten (LargestACK) bis zum Paket mit der Paketnummer, die in FirstACKRange angegeben ist, abwärts. Da FirstACKRange den Wert 9 hat, werden die Pakete 10 und 9 bestätigt. Ebenso

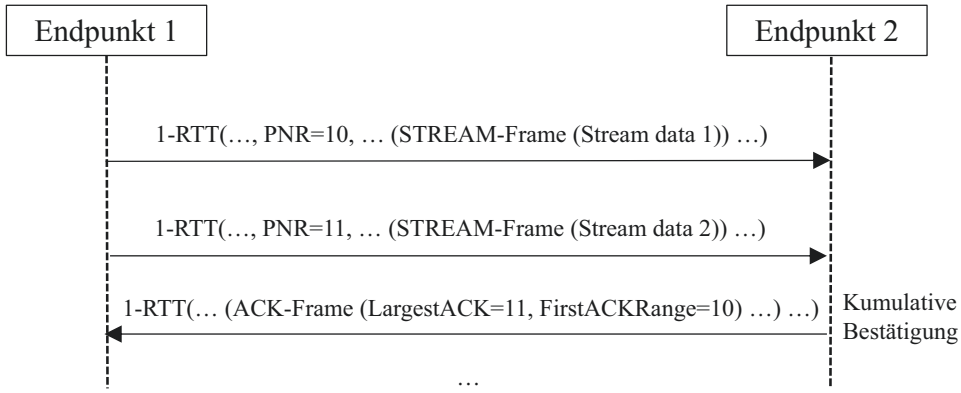


Abb. 6.11 Nachrichtenübertragung mit einfacher Bestätigung

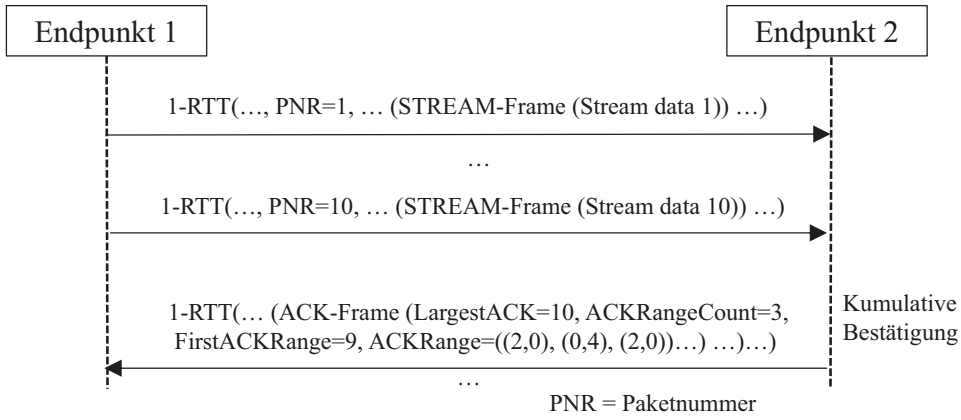


Abb. 6.12 Nachrichtenübertragung mit Bestätigung über ACK-Bereiche

werden die Bereiche, die in ACKRange angegeben sind, bestätigt bzw. explizit nicht bestätigt (sog. Gaps bzw. Lücken).

Drei Bereiche werden in der Abbildung angegeben (ACKRangeCount=3), zwei davon sind Gaps. Das Listenelement (2,0) in ACKRange zeigt beispielsweise ein Gap für die Paketnummern 8 und 7 an, das Listenelement (0,4) eine Bestätigung der vier Paketnummern 6, 5, 4 und 3. ◄

Wenn QUIC-Pakete verloren gehen, werden sie nicht grundsätzlich komplett neu gesendet. Es werden auch keine alten Paketnummern wiederverwendet, vielmehr werden bei erneutem Senden verloren gegangener Pakete stets neue Paketnummern generiert. Paketnummern wiederholen sich also in QUIC nie. Endpunkte sollen verloren gegangene Stream-Daten zeitlich vor neuen Frames senden.

Verloren gegangene Stream-Daten werden in neuen Stream-Frames gesendet, es sei denn, der Partner sendet vorher einen RESET_STREAM-Frame für den Stream, der anzeigt, dass der Stream beendet wird.

Nicht alle Frames erfordern eine Bestätigung, in jedem Fall werden aber Stream-Daten bestätigt. Um Paketverluste zu erkennen und zu behandeln, ist gemäß RFC 9002 eine Kombination der TCP-Verfahren Fast Retransmission (RFC 6681), Early Retransmission (RFC 5827), Forward Acknowledgment und Selectives Recovery (RFC 6675) vorgesehen. Diese kombinierte Funktionsweise soll grob skizziert werden:

- Ein Paket wird als verlustig betrachtet, wenn es nicht bestätigt wurde, Folgepakete aber bereits bestätigt wurden. Die Anzahl der abzuwartenden Folgepakete wird als *Packet Threshold* bezeichnet. Standardmäßig sollten mindestens drei Bestätigungen für Folgepakete gesendet worden sein, um ein unnötiges Neuversenden zu vermeiden. Da in diesem Fall kein Timer abläuft, bezeichnet man dieses Vorgehen auch als frühe erneute Übertragung (Early Retransmission).

- Wenn ein Paket schon längere Zeit nicht bestätigt wurde, wird es ebenfalls erneut gesendet. Die Wartezeit, die mindestens verstreichen sollte, wird als *Time Threshold* bezeichnet. Der Wert bzw. die Berechnung des Thresholds wird in der Spezifikation nicht exakt festgelegt.
- Eine weitere Zeitüberwachung wird über den *Probe Timeout* (PTO) erreicht, dessen Pendant bei TCP als Retransmission Timeout (RTO) bezeichnet wird. Die Berechnung des PTO ist ähnlich wie bei TCP adaptiv und für QUIC-Implementierungen nicht exakt festgelegt. Es wird aber die Berechnung als exponentiell gewichteter Durchschnitt unter Einbeziehung der mittleren Abweichung und des gleitenden Mittelwertes der Paketumlaufzeit (RTT) empfohlen.

Nachrichtenwiederholung wegen Packet Threshold

Abb. 6.13 zeigt, dass der Endpunkt 1 vier Pakete an Endpunkt 2 sendet. Endpunkt 2 bestätigt die letzten drei Pakete, nicht aber das zuerst gesendete. Da der Packet Threshold auf 3 gesetzt ist, erkennt der Endpunkt 1 beim Empfang des ACK-Frames, dass eine Bestätigung für Paket 10 fehlt, und sendet daher vor Ablauf eines Timers die Stream-Daten *Stream data 1* in einem neuen QUIC-Paket. ◄

Die Sende- und die Empfangsseite sind im RFC 9000 auch mithilfe von Zustandsautomaten als Anregung für eine QUIC-Implementierung beschrieben. In Abb. 6.14 wird der Zustandsautomat eines Streams auf der Sendeseite skizziert, in Abb. 6.15 der Zustandsautomat eines Streams auf der Empfangsseite. Bei unidirektionalen Streams werden die Zustandsautomaten nur auf einer Kommunikationsseite benötigt, bei bidirektionalen Streams in beiden Endpunkten.

Ein Stream, über den gesendet wird, ist nach seiner Erzeugung über den ersten STREAM-Frame im Anfangszustand *Ready* und geht mit weiteren Sende-Requests der

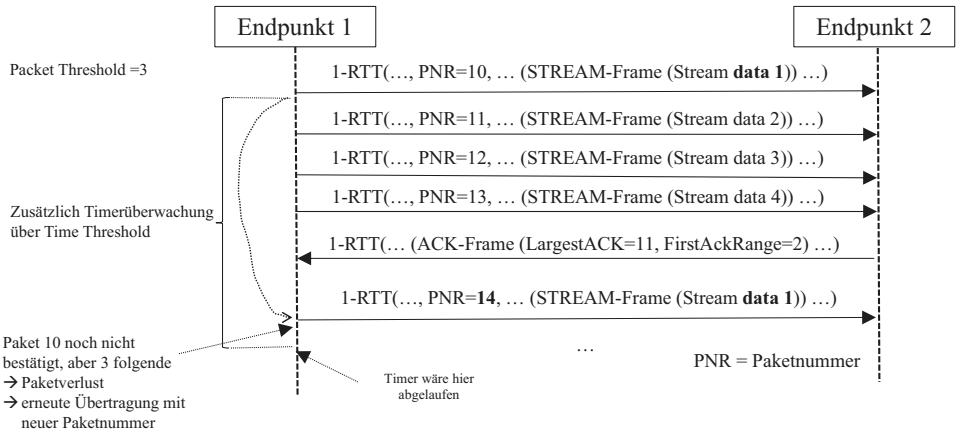


Abb. 6.13 Nachrichtenwiederholung aufgrund des Packet Thresholds

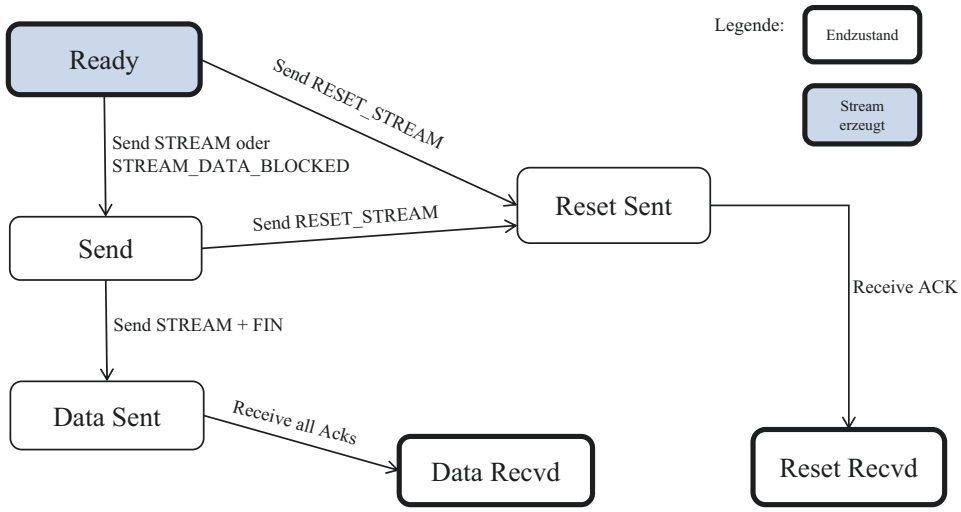


Abb. 6.14 Zustandsautomat eines Streams auf der Sendeseite

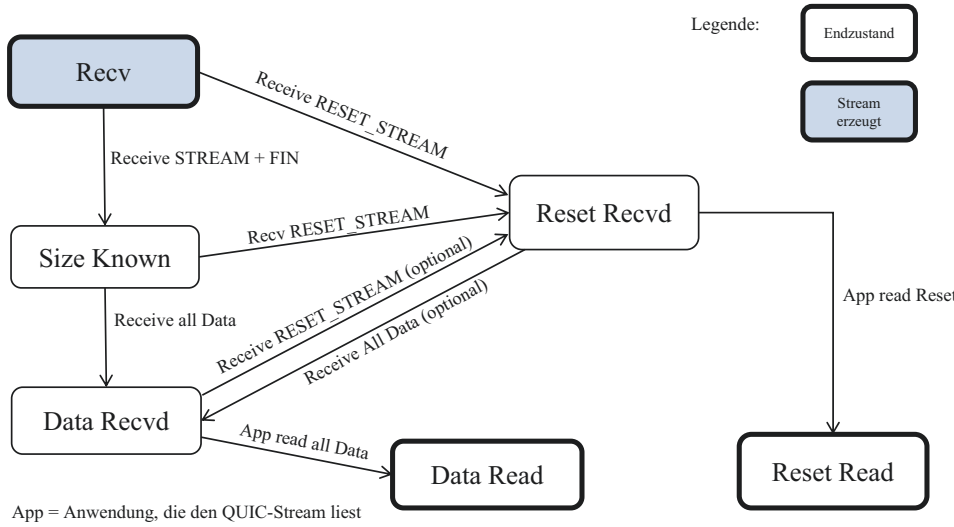


Abb. 6.15 Zustandsautomat eines Streams auf der Empfangsseite

Anwendung bzw. des darüberliegenden Anwendungsprotokolls in den Zustand *Send* über (Abb. 6.14). Auch wenn QUIC der Partnerinstanz über einen STREAM_DATA_BLOCKED-Frame anzeigt, dass aufgrund gefüllter Empfangspuffer auf der Partnerseite aktuell nichts gesendet werden kann, wird dieser Zustand eingenommen. Erst wenn der Stream beendet wird (FIN-Flag im Stream-Frame), wird der Zustand *Data Sent* eingenommen. Sind alle Bestätigungen empfangen, wird in den Endzustand *Data Recvd* gewechselt. In den Zuständen Ready und Send kann auch ein RESET_STREAM-Frame ge-

sendet werden, wodurch zunächst der Zustand *Reset Sent* eingenommen und nach Empfang einer Bestätigung in den Endzustand *Reset Recvd* gewechselt wird.

Entsprechend sieht es auf der Empfangsseite aus (Abb. 6.15). Als Anfangszustand ist *Recv* definiert. In diesem Zustand werden Daten in STREAM-Frames empfangen. Wenn die Senderinstanz den Stream abschließt (FIN-Bit) geht die Empfängerinstanz in den Zustand *Size Known* über. Dieser Zustand besagt, dass die Gesamtlänge des im Stream übertragenen Datenstromes nun der Empfängerinstanz bekannt ist. Wenn alle Daten der Senderinstanz empfangen wurden, wird in den Zustand *Data Recvd* übergegangen. Sobald die Empfängerinstanz alle Stream-Daten an die lokale Empfängeranwendung übertragen hat, wird der Endzustand *Data Read* eingenommen. Ein RESET_STREAM-Frame kann in den einzelnen Zuständen auch empfangen werden. In diesem Fall wird in den Zustand *Reset Recvd* und nach der Bekanntgabe an die Empfängeranwendung in den Endzustand *Reset Read* übergegangen.

6.5.3 Flusskontrolle

QUIC sorgt wie TCP dafür, dass Empfangspuffer nicht überlaufen. Hierfür wird ein eigener Flusskontrollmechanismus je Verbindung verwendet. Der aktuelle Sendekredit wird im Parameter Connection-Limit des Verbindungskontexts verwaltet. Ebenso wird je Stream eine eigene Flusskontrolle unterstützt. Der Sendekredit für einen Stream wird über Stream-Limits (= Sendekredit) verwaltet. Limits dürfen nicht überschritten werden. Das Aushandeln der Sendekredits (Limitparameter) erfolgt beim Verbindungs- bzw. Streamaufbau. Initiale Sendekredite für die Verbindung und die Streams werden über Transportparameter gesetzt (Tab. 6.4). Die Flusskontrolle erfolgt also bei QUIC-Verbindungen unabhängig je Stream. Das Ausbremsen eines Streams hat somit keine Auswirkung auf einen parallelen Stream.

Die Größe der Empfangspuffer für die Verbindung und je Stream sind in den Endpunkten limitiert. Die Implementierung legt die maximale Größe fest. Initiale Puffergrößen können auch über Transportparameter beim Handshake festgelegt werden, beispielsweise über den Transportparameter *initial_max_stream_data_uni* für unidirektionale Streams (Tab. 6.4).

Abb. 6.16 zeigt eine Verbindung zwischen einem Client und einem Server mit einem bidirektionalen Stream 0, für den beide Endpunkte einen Empfangspuffer verwalten und zwei unidirektionale Streams 1 und 2. Bei letzteren verwaltet nur jede Empfangsseite einen Empfangspuffer.

Das QUIC-Verfahren zur Flusskontrolle entspricht dem TCP-Sliding-Window-Verfahren. Bei TCP wird der noch verfügbare Sendekredit durch die Empfängerinstanz im TCP-Header (Feld *Zeitfenstergröße*) an die Senderinstanz übertragen (Kap. 4). Bei QUIC sendet die Senderinstanz spezielle Frames wie SEND_DATA_BLOCKED, wenn ein Stream nicht mehr senden kann, oder SEND_BLOCKED, wenn das Limit für die ganze Verbindung erreicht ist.

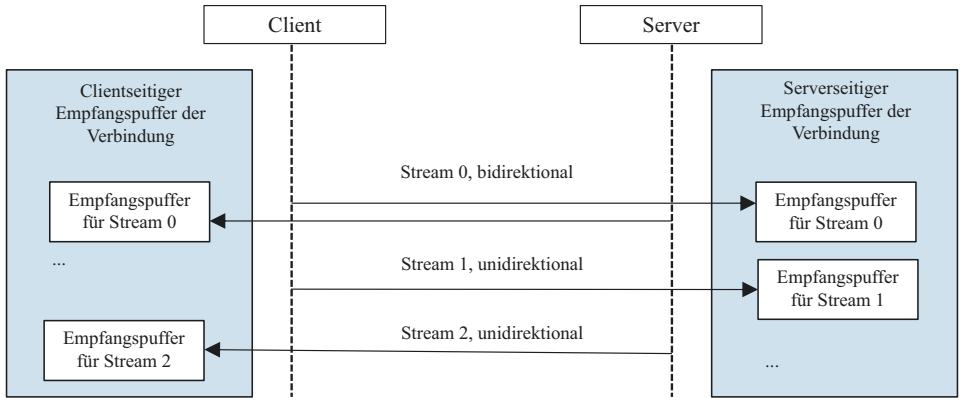


Abb. 6.16 Beispiel zur Flusskontrolle und Pufferverwaltung

6.5.4 Staukontrolle

QUIC sieht auch Mechanismen zur Staukontrolle vor und greift dabei auf die im TCP-Umfeld erprobten Verfahren zurück. Im RFC 9002 wird für QUIC-Implementierungen das NewReno-Verfahren nach RFC 6582 mit den drei Phasen Slow Start, Recovery und Congestion Avoidance empfohlen.

Die Erkennung von Paketverlusten erfolgt dabei aufgrund fehlender Bestätigungspakete und, falls es im Netzwerk unterstützt wird, über ECN-Signalisierung nach RFC 3168.

QUIC beginnt mit Slow Start, wobei die Überlastfenstergröße auf einen initialen Wert gesetzt wird. Hierfür wird ein Wert aus der Berechnung

$$10^{*} \, max\_datagram\_size \, [Bytes]$$

empfohlen, wobei nach RFC 6928 mindestens

$$max(2^{*} \, max\_datagram\_size, 14.720 \, [Bytes]$$

für die Überlastfenstergröße eingestellt werden sollte.

Andere Verfahren wie CUBIC (nach RFC 8312) können ebenfalls verwendet werden, die Regeln in RFC 9002 müssen aber eingehalten werden.

QUIC unterstützt auch in Verbindung mit dem IP-Protokoll *Explicit Congestion Notification* (ECN), sofern dieses Verfahren im Netzwerkpfad genutzt wird. Wenn beide Endpunkte ECN unterstützen, können ECN-Markierungen in den ACK-Frames übertragen werden. Damit können mögliche Stausituationen auf dem Pfad zwischen den QUIC-Partnern frühzeitig erkannt und Gegenmaßnahmen zur Lastreduzierung eingeleitet werden. Mehr zu dem Thema findet sich im RFC 9000 sowie in Mandl (2019).

6.6 Zusammenfassung und Ausblick

In diesem Kapitel wurden wichtige, aber längst nicht alle Feinheiten des QUIC-Protokolls beschrieben. Das Protokoll wurde speziell für die Webkommunikation in Verbindung mit HTTP/3 entwickelt. Aufgrund der Leistungsvorteile besonders beim Verbindungsaufbau und der integrierten Sicherheitsmechanismen hat die Nutzung von QUIC in den letzten Jahren deutlich zugenommen. Facebook und Google nutzen das Protokoll schon überwiegend für ihre Internetdienste. Auch die gängigsten Browser wie Google Chrome, Mozilla Firefox, Microsoft Edge, Opera und Apple's Safari unterstützen QUIC.

Für die Implementierung von Anwendungen stehen mehrere Referenzimplementierungen wie *Chromium*, die Implementierung des Browsers Google Chrome (Chromium Project 2023) oder die QUIC Library *mvfst* von Meta (mvfst 2023) zur Verfügung, allerdings schränkt ein fehlender API-Standard die Verbreitung für beliebige Anwendungen ein.

Nach Elmenhorst (2022) ist zu beobachten, dass QUIC wohl in einigen Ländern wie Russland, China und Indien zensiert wird, da verschlüsselte QUIC-Nachrichten nicht mehr so leicht zu überwachen sind. Eine Zensur erschwert die weitere Verbreitung zudem. Die Zukunft wird zeigen, ob sich QUIC auch über die Webkommunikation hinaus bei anderen Anwendungen durchsetzen kann.

Literatur

Mandl, P. (2019) Internet Internals – Vermittlungsschicht, Aufbau und Protokolle. Springer Vieweg
Chromium Project (2023) https://www.chromium.org/Home/, letzter Zugriff am 15.08.2023
mvfst (2023) https://github.com/facebook/mvfst, letzter Zugriff am 15.08.2023
Elmenhorst, K. (2022) "A Quick Look at QUIC Censorship", https://www.opentech.fund/news/a-quick-look-at-quic/, letzter Zugriff am 15.08.2023

Programmierung von TCP- und UDP-Anwendungen

7

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 gibt es 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 Multithreading, 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 feingranulare 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 Ereigniswartepunktes für viele Ereignisquellen wie TCP-Verbindungen realisiert werden kann. Im Folgenden wird die Programmie-

Ergänzende Information Die elektronische Version dieses Kapitels enthält Zusatzmaterial, auf das über folgenden Link zugegriffen werden kann [https://doi.org/10.1007/978-3-658-43988-0_7].

P. Mandl, *TCP, UDP und QUIC Internals*, https://doi.org/10.1007/978-3-658-43988-0_7

rung 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 Framework zur Erleichterung der Programmierung wird vorgestellt, und die Programmierung von Servern für sehr leistungsstarke Anwendungen wird andiskutiert.

Auf die Entwicklung von QUIC-basierten Anwendungen wird in Ermangelung einer Standard-API nicht weiter eingegangen. Es besteht noch Hoffnung, dass eine API, mit der portierbare Anwendungen beispielsweise in Java auf Basis eines Standards ähnlich wie die TCP-Sockets auch in QUIC programmiert werden können; eine Aussage der Internet-Community hierzu (August 2023) gibt es aber nicht.

7.1 Überblick

Die Entwicklung von Kommunikationsanwendungen ist in den vergangenen 30 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 ggf. 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.

7.2 Grundkonzepte der Socket-Programmierung

Wir beginnen mit einer grundlegenden Einführung in das Programmiermodell, das der Socket-Programmierung zugrunde liegt.

7.2.1 Einführung und Programmiermodell

Die Socket-Schnittstelle ist eine klassische Transportzugriffsschnittstelle. Sie stellt eine 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.

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. 7.1 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.

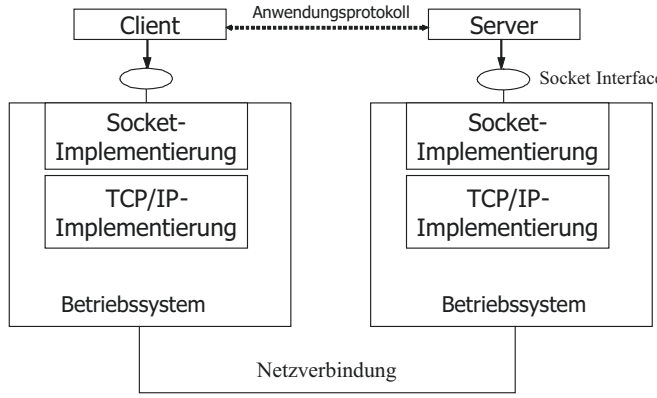


Abb. 7.1 Einbettung der Socket-Implementierung

7.2.2 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 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. 7.2 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 wohlbekannten 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.
- 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.

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

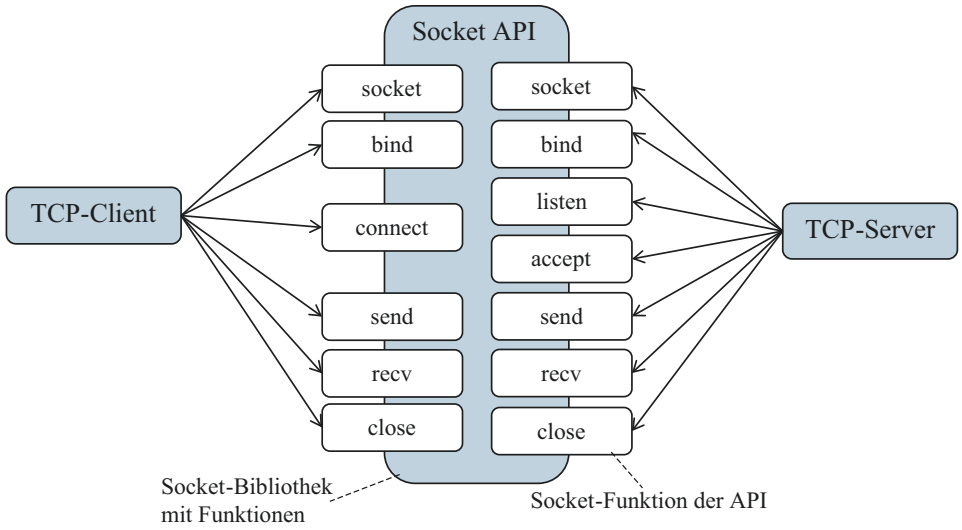


Abb. 7.2 TCP-Socket-Funktionen in der Client-Server-Beziehung

- Die Serveranwendung nimmt den Verbindungswunsch über einen *accept*-Aufruf entgegen und beantwortet ihn mit einer Connect-Response-PDU. Implizit wird ein Drei-Wege-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. 7.3 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-Drei-Wege-Handshake einleitet. Der zweite Partner muss auch einen *close*-Aufruf absetzen.

Der Empfang von Nachrichten mit der Funktion *receive* kann sowohl synchron als auch asynchron erfolgen. Mit synchron ist gemeint, dass die *receive*-Methode so lange blockiert, bis die zuständige TCP-Instanz eine Nachricht empfangen und vom Empfangspuffer 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 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 blo-

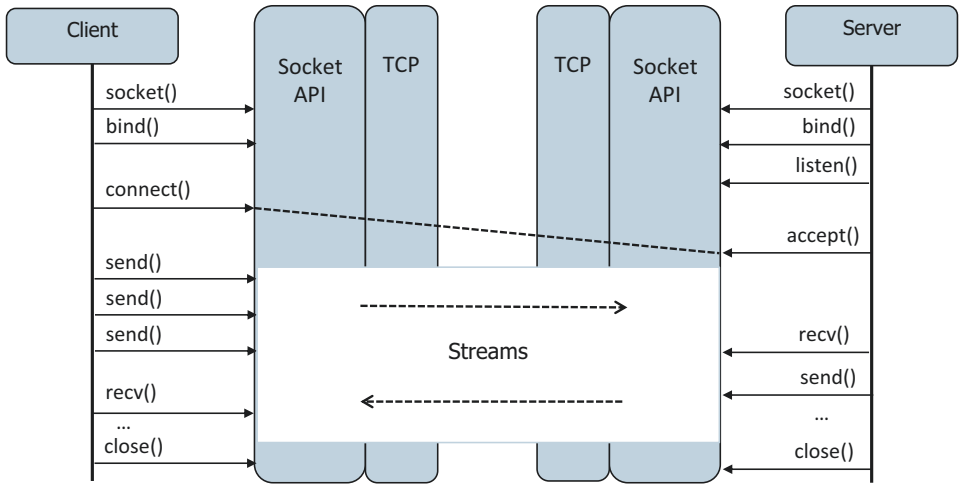


Abb. 7.3 Ablauf einer verbindungsorientierten Kommunikation

ckiert, und danach wird die Kontrolle an den Aufrufer ggf. 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 Betriebssystem-Kernel, 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 zuvorkommen.

7.2.3 Datagramm-Sockets

Im Falle einer UDP-Socket-Kommunikation fällt der Verbindungsaufbau im Vergleich zur TCP-Socket-Kommunikation weg. In Abb. 7.4 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. 7.5 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*).

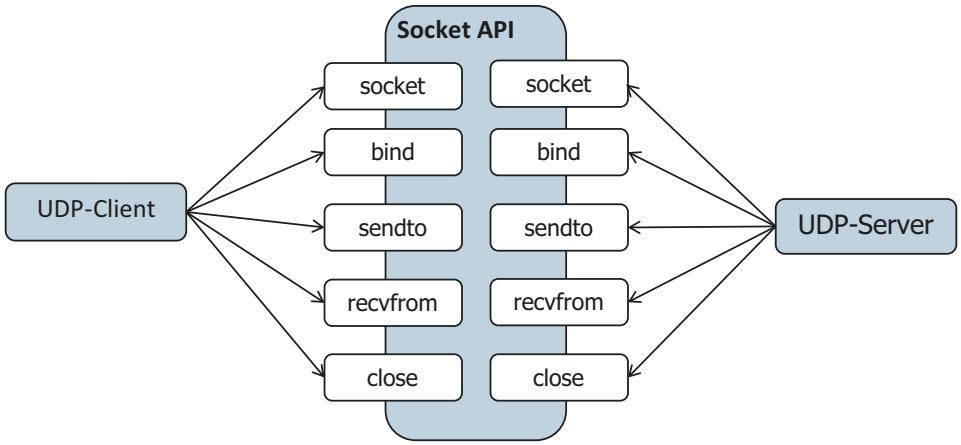


Abb. 7.4 Datagramm-Socket-Funktionen in der Client-Server-Beziehung

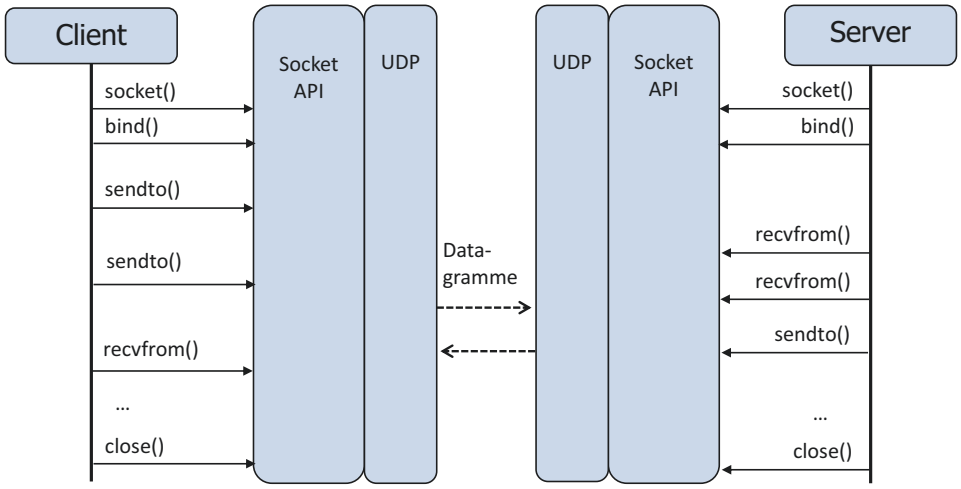


Abb. 7.5 Ablauf einer verbindungslosen Kommunikation

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.

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.

7.3 Socket-Programmierung in C

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

7.3.1 Die wichtigsten Socket-Funktionen im Detail

Die C-Socket-Schnittstelle wird in nahezu jedem Betriebssystem in einer Funktionsbibliothek bereitgestellt. Auf Basis der Funktionsbibliothek lässt es sich, verglichen mit anderen Sprachen wie Java, relativ aufwendig 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 Socket-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. 7.1).
type:	Der Socket-Typ (bei TCP: SOCK_STREAM) (Tab. 7.2)
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 werden hier IP-Adresse und Port angegeben) zur Beschreibung der lokalen Adresse
addrlen:	Länge der Serveradressstruktur

Tab. 7.1 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. 7.2 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 Internetschicht

[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).

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 Drei-Wege-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 werden hier die IP-Adresse und der Port angegeben), in der die Partneradresse eingetragen ist
addrlen:	Länge der Serveradressstruktur

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 Betriebssystem-Kernel) 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 welcher die Clientadresse der Verbindung gespeichert wird
addrlen:	Größe der Socket-Adressstruktur des Client

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 neu 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.

7.3.2 Nutzung von Socket-Optionen

Die in den vorhergehenden Kapiteln 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 Protokolllevel sind in Tab. 7.3 aufgeführt, mögliche Optionen in Tab. 7.4

Tab. 7.3 Protokolllevel, 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. 7.4 Mögliche Optionen (Auszug)

Optionsname	Beschreibung
SO_BROADCAST	Broadcast-Nachrichten sind erlaubt (Integerwert als Boolean: 0 oder 1).
SO_REUSEADDR	Bei einem *bind*-Aufruf kann eine lokale Adresse sofort wiederverwendet werden (Integerwert als Boolean: 0, 1).
SO_KEEPALIVE	Periodisches Senden von Nachrichten zur Lebendüberwachung (Integerwert, 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 (Integerwert gibt Puffergröße in Bytes an).
SO_RCVBUF	Setzen der Empfangspuffergröße des Sockets (Integerwert gibt Puffergröße in Bytes an).
SO_NO_CHECK	Aktivieren oder Deaktivieren der UDP-Prüfsummenberechnung für ein UDP-Socket.

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");
}
…
```

◀

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]

7.3.3 Nutzung von TCP-Sockets in C

Im folgenden Beispielcode (nach Stevens 2000) 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];
```

[3] Siehe z. B. IEEE POSIX (2016) oder http://man7.org/linux/man-pages/man2/fcntl.2.html (zugegriffen am 24.08.2023).

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

```
  // 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));
  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 Parallelbearbeitung 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 aufwendig 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);
  }
```

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

```
  // 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);
}
```

7.3.4 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. Beispiels-

weise kann es sein, dass die beiden Rechner unterschiedliche Integerformate nutzen. Aus diesem Grund stellt die Socket-Schnittstelle Konvertierungsroutinen bereit. Beispiele für Konvertierungsroutinen 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 (BERs)* bekannt, die über sogenannte ASN.1-Compiler unterstützt werden.

Heute verwendet man in Anwendungsprotokollen oft eine textorientierte Nachrichtencodierung 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.

7.4 Socket-Programmierung in Java

Im Folgenden wird auf die Socket-Programmierung in der Programmiersprache Java eingegangen.

7.4.1 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. 7.6 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 Internetadressen
- *Socket* zur clientseitigen Socket-Nutzung
- *ServerSocket* zur serverseitigen Socket-Nutzung

[6] http://www.json.org (zugegriffen am 01.05.2023).

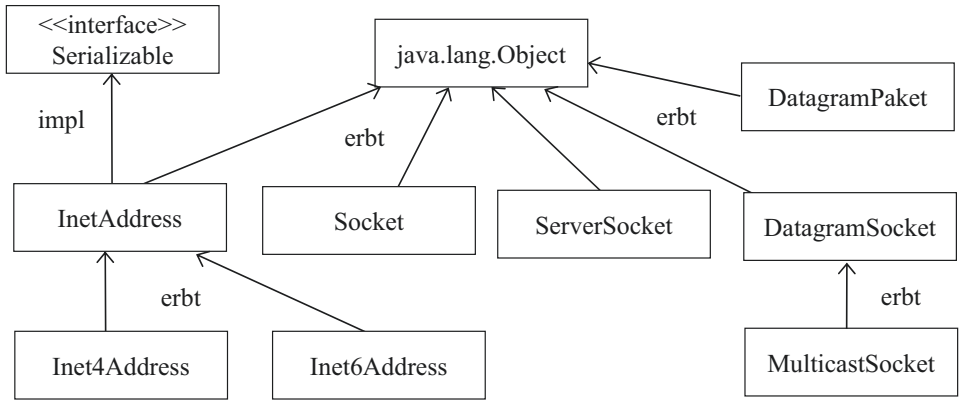


Abb. 7.6 Objektklassen für Sockets aus java.net

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

- *DatagramSocket* für die Nutzung von UDP-Sockets
- *DatagramPacket* 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 Rechner bzw. sogar einzelne Netzwerkschnittstellen in der Internetwelt 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*. Sie repräsentieren IP-Adressen in den Versionen IPv4 und IPv6. 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.

Für die Entwicklung von TCP-Anwendungen werden die Klassen *Socket* und *ServerSocket* sowie *InetAddress* verwendet. Da aus Sicht von Java das Senden und Empfangen von Daten wie das Schreiben und Lesen von Dateien behandelt werden, 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 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 Abb. 7.7 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 ServerSocket-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.

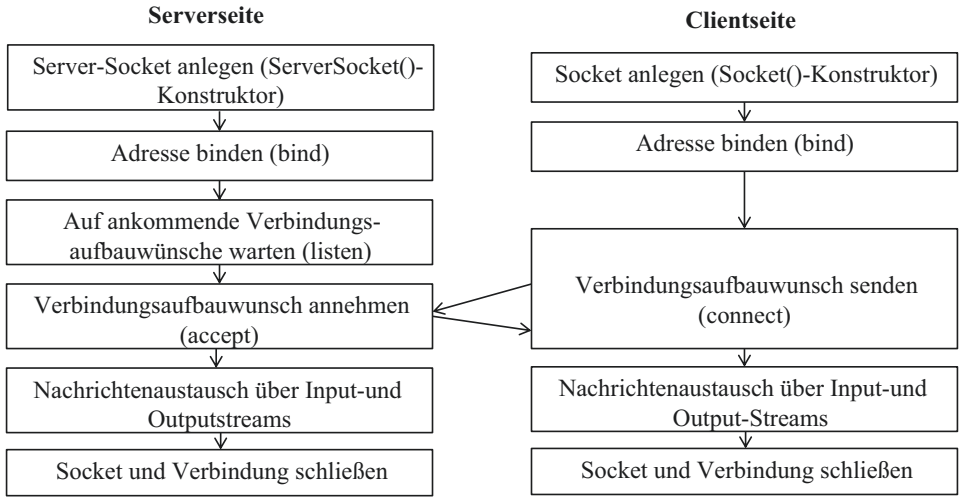


Abb. 7.7 Schematische Darstellung einer einfachen TCP-Socket-Kommunikation in Java

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-Drei-Wege-Handshake-Verbindungsaufbau ausgeführt.

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, sodass 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 Abb. 7.8 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 ihr Socket 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:

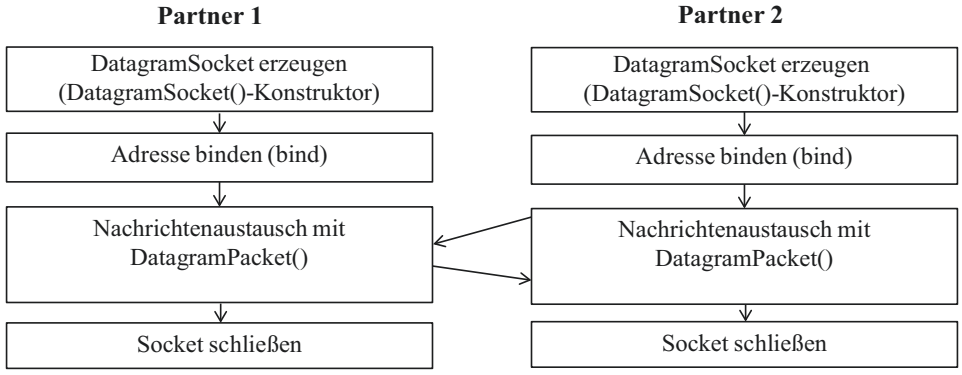


Abb. 7.8 Schematische Darstellung einer UDP-Datagramm-Kommunikation in Java

[7] https://devdocs.io/openjdk~18/java.base/java/net/socket (zugegriffen am 02.05.2023).

```
/* 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 ggf. bei einem Methodenaufruf erzeugt (geworfen) und müssen entsprechend behandelt werden.

7.4.2 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 (JVM), 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);
    ObjectOutputStream out =
      new ObjectOutputStream(client.getOutputStream());
    ObjectInputStream in =
      new ObjectInputStream(client.getInputStream();
…
} …
```

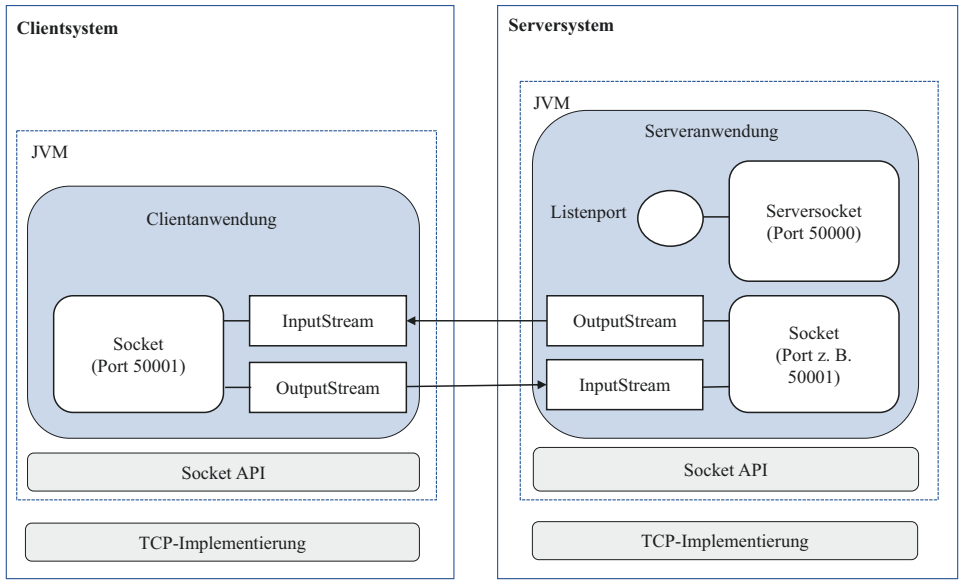


Abb. 7.9 Sockets und Streams in der JVM

Es ist zu beachten, dass für ein TCP-Socket mindestens auf einer Kommunikationsseite der ObjectOutputStream zunächst erzeugt werden muss, bevor der ObjectInput-Stream angelegt wird. Der Grund dafür ist, dass nach der Erzeugung des ObjectInputStreams auf eine Nachricht gewartet wird, die einen Serialisierungsheader enthält. Wenn keine Seite diesen sendet, blockieren beide Kommunikationspartner.[8]

In Abb. 7.9 ist die Einbettung von Java-Sockets in die virtuellen Maschinen (JVMs) von kommunizierenden Anwendungen skizziert. Jedem Socket ist also ein Stream-Paar 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.

[8] Dies ist auch in der Java-API-Beschreibung beim Konstruktor für ObjectInputStream nachzulesen: https://docs.oracle.com/en/java/javase/18/docs/api/java.base/java/io/ObjectInputStream.html.

7.4.3 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. 7.5 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. 7.6 nur die wichtigsten erwähnt werden.

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. 7.7 kurz beschrieben sind.

Mit einer konkreten Socket-Instanz kann man verschiedene Methoden nutzen, von denen in Tab. 7.8 einige beschrieben sind.

Tab. 7.5 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. 7.6 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. Ebenso wird damit die Wartezeit bei einem accept-Aufruf begrenzt.
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.

Tab. 7.7 Clientseitige Socket-Konstruktoren

Konstruktor	Beschreibung
Socket()	Erzeugt ein Socket, das noch keine Adresse zugeordnet bekommt. Es wird noch kein Verbindungsaufbau vorgenommen.
Socket(InetAddress 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. 7.8 Wichtige Methoden für clientseitige Sockets

Methode	Beschreibung
bind(SocketAddress bindpoint)	Binden einer Adresse an ein Socket als Alternative zur Angabe der Adresse bereits bei der Socket-Konstruktion. Wird kein Port angegeben, wird ein gerade freier Port ausgewählt.
connect(SocketAddress endpoint)	Baut eine Verbindung zu einem Server auf.
connect(SocketAddress endpoint, int timeout)	Baut eine Verbindung zu einem Server auf, wartet aber darauf nur die in Millisekunden angegebene Zeit.
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.
…	

Wenn bei der Erzeugung eines Sockets oder beim bind-Aufruf keine lokale IP-Adresse angegeben wird, wird die Kommunikation über alle verfügbaren Netzwerkzugänge des Hosts ermöglicht. Bei Angabe einer IP-Adresse wird nur über die der IP-Adresse zugeordnete Netzwerkkarte kommuniziert.

7.4.4 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

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 mithilfe der oben beschriebenen DatagramSocket-Methode *send* gesendet wird, wird durch alle Gruppenmitglieder empfangen, die ihrerseits den Empfang über das lokale Datagramm-Socket vornehmen.

7.5 Einfache Java-Beispiele

7.5.1 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);
    ObjectOutputStream out =
      new ObjectOutputStream(client.getOutputStream());
    ObjectInputStream in =
      new ObjectInputStream(client.getInputStream();

    // 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.

7.5.2 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.

```
package UDPEchoExample;
import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;

public class EchoUdpServer {

  protected DatagramSocket socket;

  public EchoUdpServer(int port) throws IOException {
    socket = new DatagramSocket(port);
  }

  public static void main(String[] args) {
    final int serverPort = 56000;
    try {
      EchoUdpServer echo = new EchoUdpServer(serverPort);
      System.out.println("UDP Echo Server started");
      echo.execute();
    } catch (IOException e) {
      System.out.println("Port " + serverPort +
                         " is already in use");
       System.out.println("UDP Echo Server not started");
    }
  }
```

```
public void execute() {
  boolean running = true;
  while (running) {
    try {
      System.out.println("Waiting for messages ...");
      DatagramPacket packet = receivePacket();
      sendEcho(packet.getAddress(), packet.getPort(),
               packet.getData(), packet.getLength());
    } catch (IOException e) {
        running = false;
        closeSocket();
    }
  }
}

protected DatagramPacket receivePacket() throws IOException {
  byte[] buffer = new byte[65535];
  DatagramPacket packet = new DatagramPacket(buffer, buffer.length);

  try {
     socket.receive(packet);
  } catch (IOException e) {
     System.out.println("Exception in receive");
     throw e;
  }

  String receivedMessage = new String(packet.getData(), 0,
                                      packet.getLength());
  System.out.println("Message received: " + packet.getLength() +
              " Bytes >" + receivedMessage + "<");
  return packet;
}

protected void sendEcho(InetAddress address, int port, byte[] data,
                        int length) throws IOException {
  DatagramPacket packet = new DatagramPacket(data, length,
                                             address, port);

  try {
    socket.send(packet);
    String sendMessage = new String(packet.getData(), 0,
                                    packet.getLength());
    System.out.println("Response sent:    " + length + " Bytes >" +
                       sendMessage + "<");
  } catch (IOException e) {
```

```
      System.out.println("Exception in send");
      throw e;
    }
  }

  protected void closeSocket() {
    try {
     socket.close();
    } catch (Exception e) {
      System.out.println("Error in close");
    }
  }
}
```

Der zugehörige Client kann vereinfacht wie folgt aufgebaut sein:

```
package edu.hm.dako.echoUdpSocketApp;


import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;

public class EchoUdpClient {


  protected DatagramSocket socket;
  protected InetAddress serverAddress;
  protected int serverPort;

  public EchoUdpClient(int serverPort) throws IOException {
    try {
      socket = new DatagramSocket();
      serverAddress = InetAddress.getLocalHost();
      this.serverPort = serverPort;
     System.out.println("Serverhost: " + serverAddress.getHostName() +
                        ", Serverport: " + this.serverPort);
    } catch (IOException e) {
      socket.close();
      throw e;
    }
  }
```

```
public static void main(String[] args) {
  final int serverPort = 56000;
  try {
    EchoUdpClient echoClient = new EchoUdpClient(serverPort);
    System.out.println("UDP Echo Client started");
    echoClient.execute();
    System.out.println("UDP Echo Client finished");
  } catch (IOException e) {
    System.out.println("Error in Creating a datagram socket");
    System.out.println("UDP Echo Server not started");
  }
}

public void execute() throws IOException {
  String myMessage = "Des is de Nachricht, die zruck kemma soi";

  try {
    endPacket(myMessage, myMessage.length(), serverAddress,
              serverPort);
    receivePacket();
  } catch (IOException e) {
    closeSocket();
  }

  try {
    Thread.sleep(1000);
  } catch (InterruptedException e) {
    System.out.println("Error in sleep");
  }
}

protected void sendPacket(String message, int length, InetAddress
              serverAddress, int serverPort) throws IOException {
  byte[] buffer;
  buffer = message.getBytes();

  DatagramPacket packet = new DatagramPacket(buffer, length,
                          serverAddress, serverPort);
  try {
    socket.send(packet);
    String sendMessage = new String(packet.getData(), 0,
                         packet.getLength());
    System.out.println("Message sent : " + length + " Bytes >" +
                       sendMessage + "<");
  } catch (IOException e) {
```

```
      System.out.println("Exception in send");
      throw e;
    }
  }

  protected void receivePacket() throws IOException {
    byte[] buffer = new byte[65535];
    DatagramPacket packet = new DatagramPacket(buffer, buffer.length);
    try {
      socket.receive(packet);
      String receivedMessage = new String(packet.getData(), 0,
                                          packet.getLength());
      System.out.println("Echo received: " + packet.getLength() +
                         " Bytes >" + receivedMessage + "<");
    } catch (IOException e) {
      System.out.println("Exception in receive");
      throw e;
    }
  }

  protected void closeSocket() {
    try {
      socket.close();
    } catch (Exception e) {
      System.out.println("Error in close");
    }
  }
}
```

7.5.3 Ein Java-Beispiel für Multicast-Sockets

Das folgende Beispielprogramm soll aufzeigen, wie Multicast über Java funktioniert. Ein Multicast-Receiver und ein Multicast-Sender verbinden sich über die Methode *joinGroup* über eine vorgegebene IPv4-Multicast-Adresse (hier 224.10.1.1) und einen UDP-Port (hier 7000) in einer Multicast-Gruppe. Der Sender sendet eine Nachricht an die Gruppe und beendet sich danach. Der Empfänger liest alle Nachrichten so lange, bis ein Fehler auftritt.

Die Methode *joinGroup* (ab Java-Version14) erfordert die Angabe des Netzwerk-Interface, über das Multicast-Nachrichten empfangen werden sollen. Dies wird im Beispiel statisch auf „en0“ eingestellt. Diese Bezeichnung ist bei macOS für LAN-Interfaces üblich. Besser ist es, das Netzwerk-Interface dynamisch zu ermitteln, was über die Klasse *NetworkInterface* möglich ist. Hierauf wird aber im Beispiel nicht weiter eingegangen.

Der Programmcode des Empfängers sieht folgendermaßen aus:

```
package edu.hm.dako.udpMulticast;

import java.io.IOException;
import java.net.DatagramPacket;
import java.net.InetAddress;
import java.net.InetSocketAddress;
import java.net.MulticastSocket;
import java.net.NetworkInterface;

public class UdpMulticastReceiver {


  public final static int MY_MULTICAST_PORT = 7000;
  public final static int MY_LOCAL_PORT = 7000;
  public final static String MY_MULTICAST_ADDRESS = "224.10.1.1";

  public static void main(String[] args) {

    InetAddress myMulticastAddress;
    InetSocketAddress group;
    MulticastSocket s = null;
    NetworkInterface networkInterface;

    try {
      // Verwendung des Ports 7000 und der IP-Klasse-D-Adresse
      // 224.10.1.1 für die Multicast-Gruppe
      myMulticastAddress = InetAddress.getByName(MY_MULTICAST_ADDRESS);
      group = new InetSocketAddress(myMulticastAddress, MY_LOCAL_PORT);
      s = new MulticastSocket(MY_MULTICAST_PORT);
      System.out.println("Multicast socket created with port " +
                         s.getLocalPort());

      // Zur Multicast-Gruppe beitreten
      networkInterface = NetworkInterface.getByName("en0");
      s.joinGroup(group, networkInterface);
      System.out.println("Multicast group joined");

    } catch (IOException e) {
      System.out.println("Error in creating multicast socket");
      System.exit(1);
    }
```

```
    byte[] block = new byte[1024];
    DatagramPacket packet = new DatagramPacket(block, block.length);
    boolean running = true;

    while (running) {
      try {
        System.out.println("Waiting for multicast message ...");

        // Nachricht empfangen
        s.receive(packet);

        // ... und ausgeben
        String receivedMessage = new String(packet.getData(), 0,
            packet.getLength());
        System.out.print("Packet received in Multicast group: >");
        System.out.println(receivedMessage + "<");

      } catch (IOException e) {
        System.out.println("Error in receiving Multcast packet");
        running = false;
        s.close();
      }
    }
  }
}
```

Der Programmcode des Senders kann beispielsweise wie folgt programmiert werden:

```
package edu.hm.dako.udpMulticast;

import java.io.IOException;
import java.net.DatagramPacket;
import java.net.InetAddress;
import java.net.MulticastSocket;

public class UdpMulticastSender {

  public final static int MY_MULTICAST_PORT = 7000;
  public final static String MY_MULTICAST_ADDRESS = "224.10.1.1";

  public static void main(String[] args) {

    InetAddress group = null;
    MulticastSocket s = null;
```

```
    try {
      group = InetAddress.getByName(MY_MULTICAST_ADDRESS);
      s = new MulticastSocket();
      System.out.println("Multicast socket created");
      // Sender muss nicht in Multicast-Gruppe sein
    } catch (IOException e) {
      System.out.println("Error in creating multicast socket");
      System.exit(1);
    }

    // Nachricht füllen, lokalen Port hinzugeben
    byte[] buffer;
    String message = "Dies ist ein Multicast von Port ";
    String messageToSend = message.concat(((Integer)
    s.getLocalPort()).toString());

    buffer = messageToSend.getBytes();
    DatagramPacket packet = new DatagramPacket(buffer,
            buffer.length, group, MY_MULTICAST_PORT);

    // ... und senden
    try {
      s.send(packet);
      System.out.println("Packet sent to Multicast group "
              + group.getHostAddress() + ":" + MY_MULTICAST_PORT
              + " >" + messageToSend + "<");
    } catch (IOException e) {
      System.out.println("Error in sending Multcast packet");
    }
    s.close();
  }
}
```

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. 224.0.0.0 ist allerdings eine reservierte Adresse, die nicht als öffetliche oder private IP-Adresse zugelassen ist. Der Adressbereich 224.0.0.1 bis 224.0.0.255 wird beispielsweise in lokaler Umgebung genutzt. Nachrichten an diese Gruppenadressen verlassen ein lokales Netzwerk nicht.

IP-Router benötigen für das Routing dieser Nachrichten an alle Beteiligten IPv4-Rechner spezielle Multicast-Routing-Protokolle wie *DVMRP* oder *MOSPF* (Tanenbaum et al. 2021).

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-(MLD-)Funktion.

7.5.4 Zusammenspiel von Socket API und Protokollimplementierung

Es wurde bereits 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, das 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 bekannt gemacht. 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 Drei-Wege-Handshake wird also vollständig ausgeführt. Mit dem *connect*-Aufruf auf der aktiven Seite (Clientseite) kann aktiv ein Drei-Wege-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, d. h., 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 Drei-Wege-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

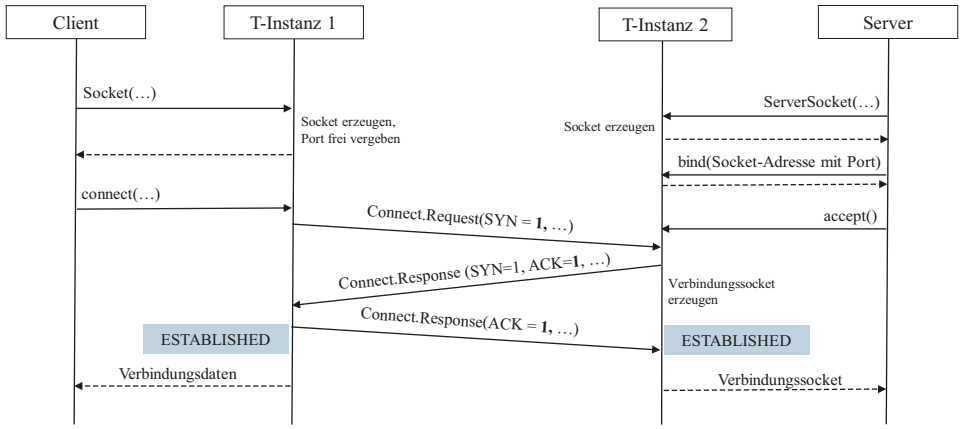


Abb. 7.10 Zusammenspiel Java Socket API und TCP-Instanz beim Verbindungsaufbau

zur etablierten Verbindung ist in Abb. 7.10 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 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. 7.11 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

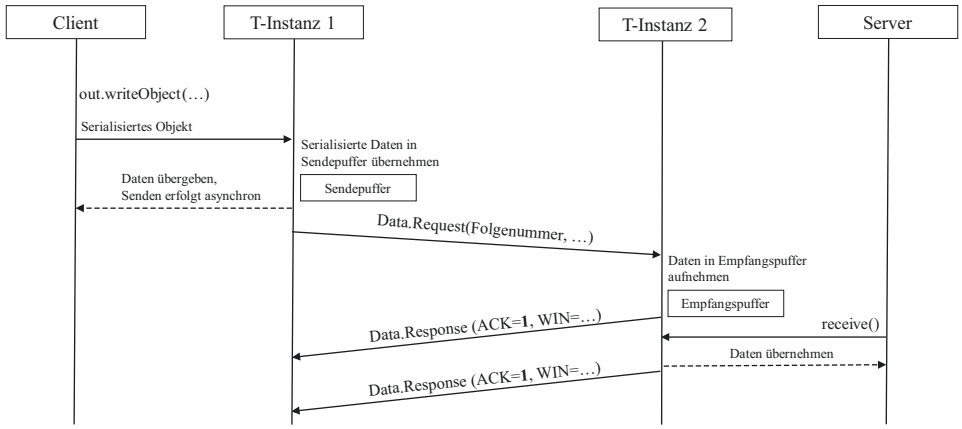


Abb. 7.11 Zusammenspiel Java Socket API und TCP-Instanz beim Datenaustausch

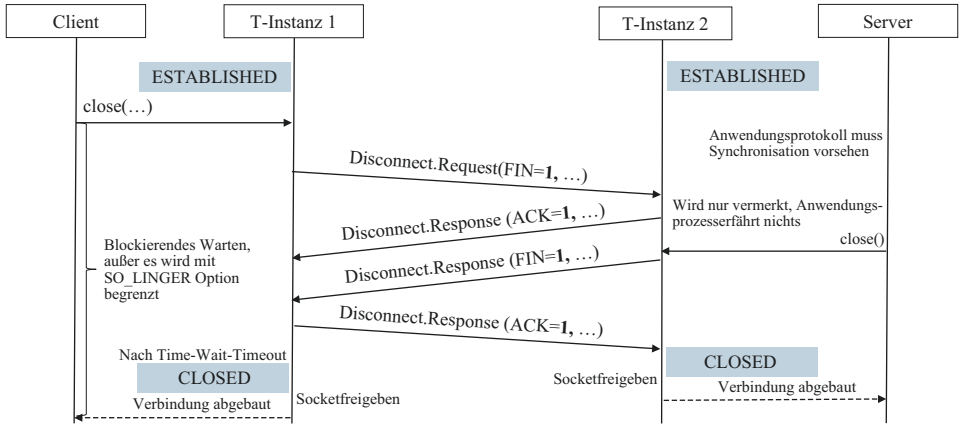


Abb. 7.12 Zusammenspiel Java Socket API und TCP-Instanz beim Verbindungsabbau

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. 7.12).

7.6 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 Abschnitt stellen wir ein Mini-Framework zur Nutzung von TCP-Sockets in der Programmiersprache Java vor, das einige Erleichterungen ver-

schafft. 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 Verbindungskontexts 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.[9]

7.6.1 Überblick über vordefinierte Schnittstellen und Objektklassen

Das in Abb. 7.13 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, weshalb eine Java-Objektserialisierung der Anwendungs-PDUs möglich ist. Man muss sich nicht um eigene Mechansimen zur Serialisierung und Deserialisierung kümmern.

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

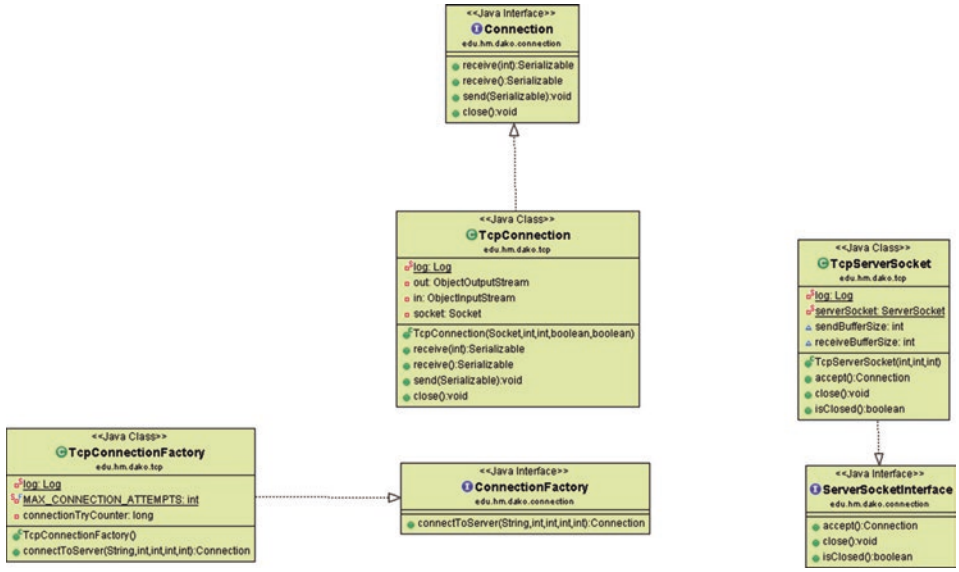


Abb. 7.13 Klassenmodell des Frameworks für die TCP-Kommunikation

```
/* 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();
}
```

7.6.2 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 vewendeten Java-Objekte müssen lediglich serialisierbar sein, was durch die Implementung des Java-Standard-Interface *Serializable* gegeben ist. Die *receive*-Methode kann auch zeitlich über eine Zeitangabe begrenzt werden, sodass 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, EndOfFileException {
    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();
    }
  }
}
```

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 u. a. 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;

  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 {
       attempts++;
        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 e3) {
         throw new IOException();
      }
      if (attempts >= MAX_CONNECTION_ATTEMPTS) {
        throw new IOException();
      }
    }
    return connection;
  }
}
```

Die Objektklasse *TCPServerSocket* stellt eine konkrete Implementierung des Interfac *ServerSocket* dar. Bei Ankunft eines Verbindungsaufbauwunsches wird in der *accept*-Methode ein lokales Socket für die Verbindung erzeugt.

```
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();
  }
}
```

7.6.3 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. 7.14 skizziert.

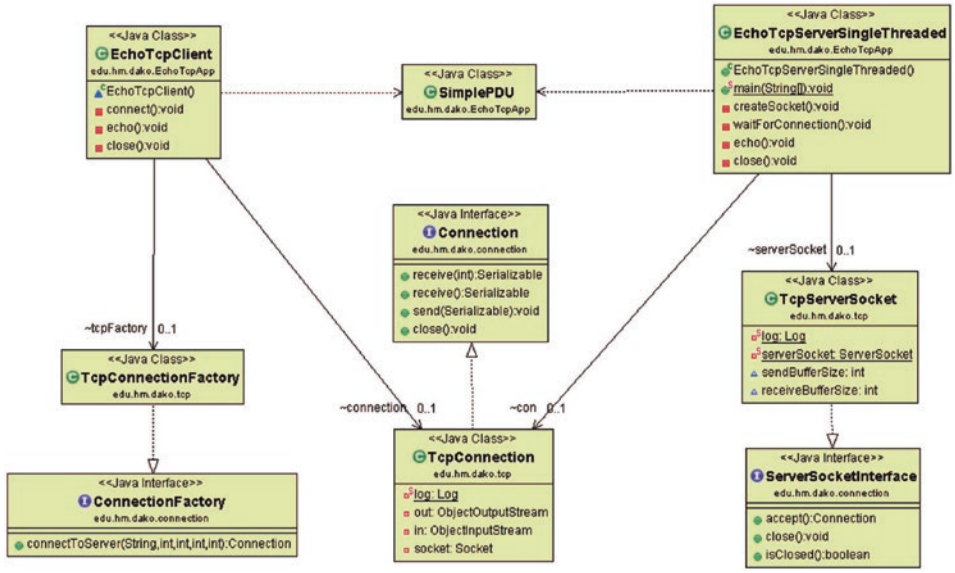


Abb. 7.14 Klassenmodell für die Echo-Beispielanwendung (Single-threaded Server)

```
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 *TCPConnectionFactory*, 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.

```
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) {
      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");
      }
    }
}
```

7.6.4 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. 7.15 ist das Klassendiagramm für diese Anwendung skizziert.

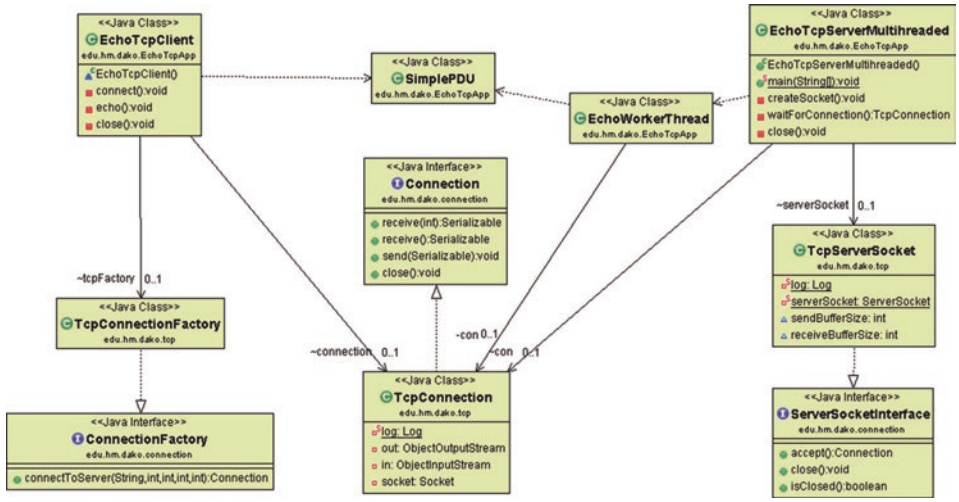


Abb. 7.15 Klassenmodell für die Echo-Beispielanwendung (Multi-threaded Server)

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.

```
package edu.hm.dako.EchoTcpApp;
import …

  public class EchoTcpServerMultihreaded {
    TcpServerSocket serverSocket = null;
      TcpConnection con = null;

    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 selbstständig 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.

```
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();
      }
    }
}
```

7.7 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. 7.16).

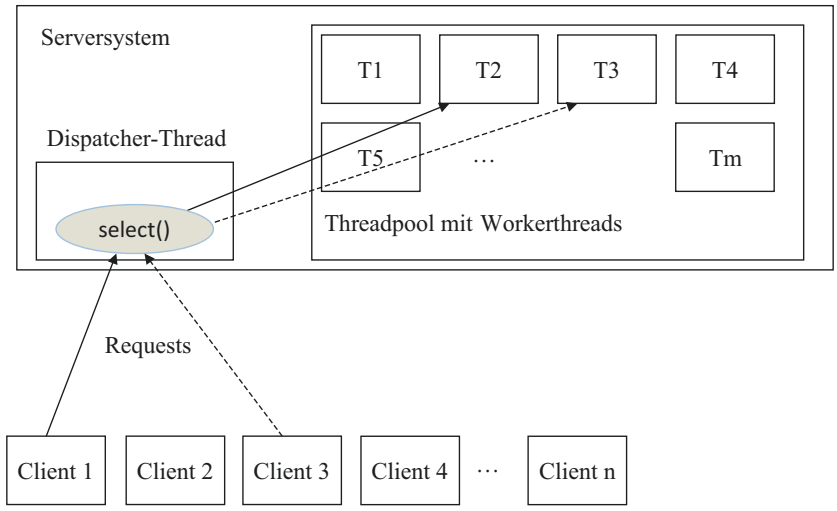


Abb. 7.16 Dispatching über zentralen Ereigniswartepunkt

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 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-Server-Kommunikation kapseln die gesamte Socket-Schnittstelle und bieten komfortable Mechanismen für die Request-Bearbeitung. 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 am01.05.2023

Mandl, P. (2009) Masterkurs Verteilte betriebliche Informationssysteme – Prinzipien, Architekturen und Technologien, Springer-Vieweg Verlag, 2009

Mandl, P. (2014) Grundkurs Betriebssysteme, 4. Auflage, Springer-Vieweg Verlag, 2014

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.; Feamster, N. Wetherall, D. J. (2021) Computer Networks, Sixth Edition, Pearson Education Limited, 2021

Schlussbemerkung

8

Zusammenfassung

Dieses Buch sollte einen tieferen Einblick in die Funktionsweise und Nutzung der 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 sie 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 mithilfe der Socket API eingeführt, um zu zeigen, wie man verteilte Anwendungen entwickeln kann, die den Nachrichtenaustausch über TCP und UDP organisieren. Das auf UDP aufsetzende QUIC-Protokoll verbreitet sich seit seiner Standardisierung immer weiter und ist mittlerweile, insbesondere für die Webkommunikation mit HTTP/3, eine Alternative zum klassischen TCP-basierten HTTP-Stack. Daher wird in dieser Auflage auch auf die QUIC-Protokollmechanismen eingegangen. Programmierschnittstellen für QUIC sind allerdings noch nicht standardisiert, auf proprietäre Implementierungen von APIs wurde daher nicht weiter eingegangen.

Bei der Entwicklung und Erprobung neuer Transportmechanismen gibt es aufgrund der wachsenden Anforderungen weiterhin viel zu tun. Die Internet-Community diskutiert und forscht hier rege weiter. Allerdings können sich Erweiterungen und Verbesserungen

P. Mandl, *TCP, UDP und QUIC Internals*, https://doi.org/10.1007/978-3-658-43988-0_8

aufgrund der sehr starken Verbreitung von TCP und auch von UDP nur langsam durchsetzen. Die Entwicklung bleibt aber weiterhin sehr interessant.

Auf die anderen Kommunikationsschichten wie etwa auf die Vermittlungsschicht und die Netzzugangsschicht wurde in diesem Buch nur so weit eingegangen, wie es für das Verständnis der Funktionen des Transportsystems von Bedeutung ist. Vor allem die Vermittlungsschicht mit ihren vielfältigen Funktionen soll an anderer Stelle näher betrachtet werden (z. B. Mandl et al. 2010; Mandl 2019).

Literatur

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

Mandl, P. (2019) Internet Internals – Vermittlungsschicht, Aufbau und Protokolle. Springer Vieweg

Anhang: TCP/IP-Konfiguration in Betriebssystemen

TCP/IP-Stacks sind fester Bestandteil im Betriebssystem-Kernel 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.

Linux

Unter Linux gibt es eine Fülle von Kernel-Parametern, die sowohl dynamisch für die Laufzeit bis zum nächsten Neustart des Systems als auch dauerhaft eingestellt werden können.

Alle Kernel-Parameter sind unter Linux im Verzeichnis */proc/sys/net* zu finden. Kernel-Parameter 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:

$$cat\,/proc/sys/net/ipv4\,/tcp\_eepalive\_time > 7200\,(Angabe\ in\ Sekunden)$$

Eine temporäre Veränderung für die Systemlaufzeit ist z. B. über das Kommando *echo* möglich:

$$echo\ \ 600 > /proc/sys/net/ipv4\,/\,tcp\_leepalive\_time$$

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, UDP und QUIC Internals*, https://doi.org/10.1007/978-3-658-43988-0

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:

$$net.ipv4.tcp\_keepalive\_time = 600$$

Die aktuell eingestellten TCP-Parameter kann man sich beispielsweise mit folgendem Kommando anzeigen lassen:

$$sysctl\text{ - }a\ grep \mid ipv4.tcp$$

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
...
```

In Tab. A.1 sind einige TCP- und UDP-Kernel-Parameter 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 in der entsprechenden Linux-Kernel-Dokumentation nachgelesen werden (Linux-TCP 2023).

Der Sourcecode der TCP-Implementierung für Linux kann in Linux-TCP (2023) studiert werden.

Tab. A.1 Wichtige Linux-Kernel-Parameter 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
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 Drei-Wege-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.)

(Fortsetzung)

Tab. A.1 (Fortsetzung)

Parameter	Bedeutung
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

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, jeweils im entsprechenden Netzwerkzugang (Interface) im Windows Registry, hat folgende Bezeichnung:

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\Tcpip\Parameters

Ein anderer wichtiger Pfad ist:

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\AFD\Parameters.

Einige TCP/IP-Parameter sind in Tab. A.2 zusammengefasst.

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 die Anzahl der ausstehenden Bestätigungen (ACKs) beim Empfänger fest, die dazu führen, dass keine verzögerte Bestätigung nach Clarks Algorithmus verwendet wird. Mit dem Standardwert von 2 führt das zweite empfangene, aber noch nicht bestätigte Segment zum Senden einer Bestätigung. Wird der Wert auf 1 gesetzt, wird jedes empfangene TCP-Segment sofort bestätigt
SackOpts	Kennzeichen zum Aktivieren von Selective Acknowledgement (SACK) gemäß RFC 2018
Tcp1323Timestamps	Der Parameter legt fest, ob die Timestamps Option verwendet wird (RFC1323)
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

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 die Network Shell *netsh* (ältere Windows-Versionen) oder über die *Microsoft Powershell*[2] (aktuell empfohlene Variante) als Administrator möglich. netsh

[1] Informationen hierzu findet man unter https://technet.microsoft.com/en-us/library/cc957548.aspx (zugegriffen am 02.05.2023).

[2] https://learn.microsoft.com/en-us/powershell/ (zugegriffen am 02.05.2023).

wird in Zukunft wohl nicht mehr gepflegt. Ein Anzeigen der aktuellen TCP-Parametereinstellungen kann in der Kommandoeingabe z. B. mit dem Kommando

netsh 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 mit dem Powershell-Kommando

Set-NetTCPSetting -SettingNameInternetCustom -EcnCapability Enabled

ECN zugelassen werden. Über netsh lautet der entsprechende Befehl

netsh int *tcp set globalecncapability = enabled*.

macOS

Auch in macOS können Kernel-Parameter für TCP eingestellt werden. Die Parameter können zur Laufzeit über das Kommando *sysctl* angesehen und verändert werden.

Ein Auflisten aller TCP-Systemparameter in der Shell ist wie folgt möglich:

sysctl net.inet.tcp

Die Ausgabe des Kommandos ist im Folgenden angedeutet:

```
net.inet.tcp.mssdflt: 512
net.inet.tcp.keepidle: 7200000
net.inet.tcp.keepintvl: 75000
net.inet.tcp.sendspace: 131072
net.inet.tcp.recvspace: 131072
net.inet.tcp.v6mssdflt: 1024
…
```

Die Veränderung eines Parameters erfolgt über die Zuordnung eines Wertes, wie hier am Beispiel des Parameters *mssdflt* gezeigt wird:

sudo sysctl -w net.inet.tcp.mssdflt = 1452

Mit diesem Kommando wird die Standard-MSS auf 1452 Bytes gesetzt.

Tab. A.3 Wichtige TCP/IP-Kernel-Parameter in macOS

Parameter	Bedeutung
blackhole	Legt fest, was passiert, wenn ein TCP-Segment bei einem geschlossenen Port ankommt, also ob zusätzlich zum Verwerfen des Segments eine RST-PDU gesendet wird oder nicht
slowstart_ flightsize	Anzahl der zulässigen noch nicht bestätigten TCP-Segmente während der Slow-Start-Phase bei Verbindungen im globalen Internet
local_slowstart_ flightsize	Anzahl der zulässigen noch nicht bestätigten TCP-Segmente während der Slow-Start-Phase im lokalen Netzwerk
sendspace	Maximale Fenstergröße für die Flusskontrolle beim Senden in einer TCP-Verbindung
receivespace	Maximale Fenstergröße beim Empfangen für die Flusskontrolle in eine TCP-Verbindung
mssdflt	Standardgröße für die Maximum Segment Size (MSS) bei Nutzung von IPv4
v6mssdflt	Standardgröße für die Maximum Segment Size (MSS) bei Nutzung von IPv6
msl	Wartezeit in Millisekunden im Zustand TIME_WAIT (Standard: 15000)
delayed_ack	Anzahl der Bestätigungen (ACK), die ein Empfänger verzögern kann, bevor er eine kumulative Bestätigung senden muss
autorcvbufmax	Maximale Größe des Empfangspuffers einer TCP-Verbindung in Bytes
autosndbufmax	Maximale Größe des Sendepuffers einer TCP-Verbindung in Bytes
always_keepalive	Verwendung des Keepalive Timers (SO_KEEPALIVE) bei jeder TCP-Verbindung (Standard: 0 = nicht genutzt)
keepidle	Der Parameter gibt die Zeit zwischen dem zuletzt gesendeten TCP-Segment und der ersten Keepalive-Nachricht an, sofern always_keepalive eingeschaltet ist
keepintvl	Mit diesem Parameter wird die Zeit zwischen zwei aufeinanderfolgenden Keepalive-Nachrichten eingestellt, wenn der Partner nicht antwortet
sack	Kennzeichen, ob Selective Acknowledgement anstelle von Go-back-N (SACK) aktiviert ist. Standardwert=1 (aktiviert)

Zur dauerhaften Veränderung der Standardeinstellungen werden die Parameter in die Datei */etc/sysctl.conf* eingetragen.

Einige dieser Parameter werden in Tab. A.3 beschrieben. Weiterführende Erläuterungen finden sich in der sysctl-Kommando-Beschreibung.[3]

[3] Siehe zum Beispiel unter https://developer.apple.com/library/archive/documentation/System/Conceptual/ManPages_iPhoneOS/man3/sysctl.3.html (letzter Zugriff am 03.06.2023).

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, letzter Zugriff am 25.08.2017

Comer, D. 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 (2023) https://github.com/torvalds/linux/blob/master/net/ipv4/tcp.c, letzter Zugriff am 02.05.2023

Salman, A.; Schulzrinne, H. (2004) An Analysis of the Skype Peer-to-Peer Internet Telephony Protocol, 2004, http://arxiv.org/abs/cs/0412017v1. Zugegriffen: 03.05.2021

Schill, A.; Springer, T. (2012) Verteilte Systeme, Springer Verlag, 2012

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, Spektrum Akademischer Verlag, 1998

Zitterbart, M. (1995) Hochleistungskommunikation Band 1: Technologie und Netze, Oldenbourg Verlag, 1995

Stichwortverzeichnis

P. Mandl, *TCP, UDP und QUIC Internals*, https://doi.org/10.1007/978-3-658-43988-0