4 Transportprotokolle TCP, UDP, SCTP und QUIC

Notwendigkeit der Transportprotokolle

Das Protokoll IP garantiert keine zuverlässige Übermittlung von Daten zwischen den Endsystemen. Deshalb sind Funktionen notwendig, um Verluste bzw. Verfälschungen der in Form von IP-Paketen übertragenen Daten zu entdecken und eventuell zu veranlassen, dass der Quellrechner deren Übermittlung wiederholt. Diese Funktionen gehören der Transportschicht an. Die Protokolle dieser Schicht werden *Transportprotokolle* genannt.

Die TCP/IP-Protokollfamilie beinhaltet zwei klassische Transportprotokolle: UDP (*User Datagram Protocol*) und TCP (*Transmission Control Protocol*). Seit Oktober 2000 steht SCTP (*Stream Control Transmission Protocol*) als zusätzliches Transportprotokoll zur Verfügung, in dem versucht wurde, die Vorteile von UDP und TCP in Bezug auf die Übermittlung von Daten und Nachrichtenströmen zu vereinen.

Unterschiede zwischen UDP, TCP und SCTP

UDP stellt einen verbindungslosen und unzuverlässigen Dienst dar, mit dem voneinander unabhängige Nachrichten bzw. digitalisierte Echtzeitmedien wie Sprache, Audio und Video zwischen der Datenquelle (dem sendenden Rechner) und dem Datenziel (dem Empfänger) übermittelt werden. Zwischen den Kommunikationspartnern wird beim UDP keinerlei Vereinbarung hinsichtlich des Verlaufs der Kommunikation getroffen. Dagegen sind TCP und SCTP verbindungsorientierte Transportprotokolle. Der *Ablauf* der Kommunikation bei TCP bzw. bei SCTP ist durch das Protokoll geregelt, sodass sich die kommunizierenden Rechner gegenseitig laufend über den *Zustand* der Kommunikation und des Datenaustauschs verständigen. Hierdurch wird zwischen den Kommunikationspartnern eine *virtuelle Verbindung* etabliert und gepflegt.

QUIC

Ähnlich, aber wesentlich komplizierter geht das Protokoll QUIC vor, das eine virtuelle, verschlüsselte Verbindung zwischen den Kommunikationspartnern über UDP ermöglicht. Dieses Protokoll wird vornehmlich von Webbrowsern gesprochen und dient als Transportprotokoll für HTTP/3.

Überblick über das Kapitel

Nach einer kurzen Erläuterung der Aufgaben der Transportschicht in Abschnitt 4.1 beschreibt Abschnitt 4.2 das Konzept und den Einsatz von UDP. Die an die Anforderungen der Echtzeitkommunikation angepasste und neue Version von UDP wird als *UDP-Lite* bezeichnet, worauf wir ebenfalls eingehen.

Die Funktionen von TCP, insbesondere die Fehlerkontrolle und Flusskontrolle, erläutert Abschnitt 4.3 und die Implementierungsaspekte von TCP präsentiert Abschnitt 4.4. Die Überlastkontrolle bei TCP nach dem Prinzip ECN (*Explicit Congestion Notification*) stellt Abschnitt 4.5 dar. Die Erweiterung *Multipath-TCP* wird in Abschnitt 4.6 ausführlich dargestellt

Den neuen Verbindungsprotokollen SCTP ist Abschnitt 4.7 gewidmet, während in Abschnitt 4.8 das Protokoll QUIC besprochen wird.

4.1 Grundlagen der Transportprotokolle

Wozu Transportschicht?

Wir greifen die bereits in Abschnitt 1.4 gezeigte Darstellung des Transports von IP-Paketen auf und betrachten die ersten drei Schichten in einem IP-Netz als *IP-Übermittlungsdienst*. Da dieser Dienst über keine Mechanismen verfügt, um u.a. Verluste der übertragenen Daten zu entdecken und zu veranlassen, dass der Quellrechner die Übermittlung wiederholt, ist die Transportschicht nötig. Damit können bestimmte Mechanismen für die Garantie der zuverlässigen Datenübermittlung zur Verfügung gestellt werden. Abb. 4.1-1 illustriert die Bedeutung der Transportschicht in IP-Netzen und die Aufgabe ihrer Protokolle.

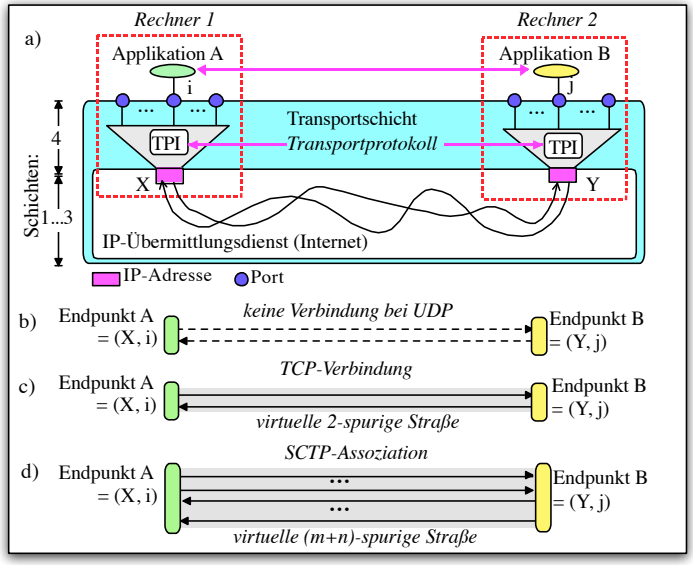


Abb. 4.1-1: Bedeutung der Transportschicht in IP-Netzen und ihre Protokolle: a) Logische Interpretation der Transportschicht, b) Kommunikation mit UDP, c) Interpretation einer TCP-Verbindung, d) Interpretation einer SCTP-Assoziation
TPI: Transportprotokollinstanz; X, Y: IP-Adressen; i, j: Ports

Sockets

Wie bereits in Abschnitt 1.4 [Abb. 1.4-9] gezeigt wurde, werden die Endpunkte einer Kommunikationsbeziehung als Paar (*IP-Adresse, Portnummer*) dargestellt und *Socket* genannt. Sockets können sowohl die Quelle als auch die Senke von Daten repräsentieren. Falls eine Applikation Daten sendet bzw. empfängt, ist sie an einen Port angebunden. Daher kann ein Socket als *Lokationsangabe* einer Applikation – d.h. auf welchem Rechner sie 'läuft' und welchen Port sie nutzt – angesehen werden.

Da mehrere Ports über eine IP-Adresse an das IP-Übermittlungsnetz 'angebunden' werden können, kann eine *Transportprotokollinstanz* als *logischer Multiplexer* angesehen werden [Abb. 1.4-7 und Abb. 1.4-8].

Transportprotokolle

Die wichtigsten Transportprotokolle in IP-Netzen sind:

- **UDP** (*User Datagram Protocol*),

- **TCP** (*Transmission Control Protocol*). sowie
- Google's **QUIC** (*Quick UDP Internet Connections* und weit abgeschlagen das
- **SCTP** (*Stream Control Transmission Protocol*).

UDP

UDP ist ein verbindungsloses und unzuverlässiges Transportprotokoll, nach dem hauptsächlich voneinander unabhängige Nachrichten zwischen den kommunizierenden Rechnern ausgetauscht werden, ohne dass eine explizite Vereinbarung hinsichtlich des Ablaufs der Übertragung vorgenommen wird. Es wird daher keine virtuelle (logische) Verbindung zwischen den Kommunikationsinstanzen aufgebaut [Abb. 4.1-1b]. UDP wird in Abschnitt 4.2 näher dargestellt.

TCP

TCP ist ein verbindungsorientiertes und zuverlässiges Transportprotokoll. Vor dem eigentlichen Datenaustausch werden Vereinbarungen hinsichtlich des Verlaufs der Kommunikation zwischen den Rechnern getroffen. Zentraler Bestandteil der Vereinbarung sind der Aufbau, die Überwachung der Korrektheit der übertragenen Daten (Byte) und der Abbau einer virtuellen Verbindung, d.h. einer *TCP-Verbindung* [Abb. 4.1-1c], die unabhängig (d.h. vollduplex) für beide Kommunikationsrichtungen erfolgt. Eine letzten Abschluss findet TCP im aktuellen RFC 9293, der die Ergebnisse des nun viele Dekaden umfassenden Erfahrungen des Einsatzes von TCP zusammen fasst.

SCTP

SCTP ist ein neues verbindungsorientiertes Transportprotokoll. Im Gegensatz zu TCP kann sich eine *SCTP-Verbindung*, die auch *SCTP-Assoziation* genannt wird, aus einer Vielzahl von entgegen gerichteten Kommunikationspfaden zusammensetzen [Abb. 4.1-1d]. SCTP wird in Abschnitt 4.7 detailliert dargestellt.

Zu berücksichtigen ist ferner, dass die Protokolle UDP, TCP und SCTP unabhängige Implementierungen der Transportschicht sind, sodass es hier prinzipiell keine Überschneidungen gibt.

Mehrere IP-Adressen

Ferner ist zu bedenken, dass alle aktuellen TCP/IP-Implementierungen mehrere IP-Adressen bereitstellen, und zwar die IPv4-Adressen `0.0.0.0` für die IP-Instanz (d.h. den Rechner) selbst [vgl. Tab. 3.3-2], `127.0.0.1` für das erste Netzwerk-Interface und die jeweils zugewiesenen privaten oder offiziellen IP-Adressen. Unterstützt der Rechner gleichzeitig IPv6, kommen die entsprechenden IPv6-Adressen hinzu.

Da die Portnummer als Angabe im TCP-, im UDP- und im SCTP-Header 16 Bit lang ist [Abb. 4.2-1, Abb. 4.3-2 und Abb. 4.7-3], kann ein Rechner pro verfügbare IP-Adresse gleichzeitig bis zu 65535 Ports organisieren. Die Ports mit den Nummern 0 bis 1023 können in der Regel nur von privilegierten, systemnahen Anwendungen benutzt werden.

Well-known Ports

Standarddienste, wie z.B. FTP oder HTTP, nutzen *Well-known Ports*, die typischerweise in der Datei `etc/services` deklariert sind. Hierdurch kann ein Client Dienste nicht nur über deren Portnummern, sondern auch über ihre zugeordneten Namen ansprechen.

Freie Ports

Die Ports mit den Nummern im Bereich von 49152 bis 65535 sind frei. Sie können den Applikationen in Rechnern zugeteilt werden und stehen insbesondere für Client-Anwendungen zur Verfügung.

Registered Ports und IANA

Die Vergabe von Portnummern und deren Zuordnung zu Applikationen wird durch die IANA koordiniert. Viele Portnummern aus dem Bereich zwischen 1024 und 49151 sind als *Registered Ports* vergeben und spezifischen Applikationen zugewiesen.

Eine aktuelle Auflistung von belegten Portnummern mit der Angabe zu jedem Port, welches Transportprotokoll der Port nutzt (UDP, TCP oder SCTP), findet man bei der IANA. Für weitere Details ist zu verweisen auf die Web-Adresse `https://www.iana.org/assignments/port-numbers`.

QUIC-Protokoll

Einen vollkommen neuen Weg geht das QUIC-Protokoll. Geplant als *User Space Implementierung* liefert QUIC alle Dienste aus einer Hand: Verlässliche Voll-Duplex-Verbindungen über UDP, Verschlüsselung und Datenflusskontrolle. Dies alles unter Verzicht auf variable Portnummern: QUIC bietet ausschließlich Kommunikation über UDP-Port 443 an, was hauptsächlich für HTTP/3 genutzt wird. Wir wollen die Eigenschaften dieses neuen Protokolls, das mittlerweile in verschiedenen RFCs (vor allem RFC 9000) spezifiziert ist, in Abschnitt 4.8 vorstellen.

4.2 Konzept und Einsatz von UDP

Besonderheiten von UDP

Mit UDP wird ein einfacher verbindungsloser, unzuverlässiger Dienst zur Verfügung gestellt. UDP wurde bereits 1980 von der IETF in RFC 768 spezifiziert. Mittels UDP können Anwendungen ihre Daten als selbstständige *UDP-Pakete* senden und empfangen. UDP bietet – ähnlich wie IP – keine zuverlässige Übertragung und keine Flusskontrolle. Ergänzend kann UDP mittels einer Prüfsumme prüfen, ob die Übertragung eines UDP-Pakets fehlerfrei erfolgt ist. Enthält ein UDP-Paket einen Übertragungsfehler, wird es beim Empfänger verworfen, allerdings ohne dass der Absender des Pakets darüber informiert wird. UDP garantiert daher keine zuverlässige Kommunikation. Eine derartige Kommunikation liegt vielen Applikationen zugrunde. Hierzu gehören besonders jene, bei denen einzelne Nachrichten zwischen Rechnern ausgetauscht werden.

Einsatz von UDP

Der wichtigste Einsatz von UDP liegt in der Übermittlung von Netzwerkkonfigurationsnachrichten und unterstützt das

- DHCP (*Dynamic Host Configuration Protocol*) und
- DNS (*Domain Name System*).

Ein weiteres Einsatzgebiet von UDP besteht in der Unterstützung der Echtzeit- und Multimedia-Kommunikation (z.B. Skype):

- UDP wird vom RTP (*Real-time Transport Protocol*) für die Audio- und Video-Übermittlung und vom Signalisierungsprotokoll SIP (*Session Initiation Protocol*) verwendet.
- Bei *Voice over IP* dient UDP als primäres Transportprotokoll [Bad22].

Neben dem Protokoll RADIUS [Abschnitt 6.6] nutzen ferner die Anwendungen TFTP (*Trivial File Transfer Protocol*) und RPC/NFS (*Remote Procedure Call* beim *Network File System*) UDP als Transportprotokoll.

4.2.1 Aufbau von UDP-Paketen

Den Aufbau von UDP-Paketen zeigt Abb. 4.2-1 mit folgenden Angaben im UDP-Header:

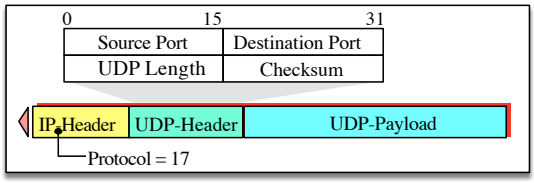


Abb. 4.2-1: Aufbau von UDP-Paketen

- `Source Port` (Quellport)
  Angabe der Nummer des Ports als Kommunikationspuffer der Applikation im Quellrechner [Abb. 4.1-1a].
- `Destination Port` (Zielport)
  Die Nummer des Ports als Identifikation der Applikation im Zielrechner.
- `UDP Length` (UDP-Paketlänge)
  Hier wird die Länge des UDP-Pakets angegeben.
- `Checksum` (Prüfsumme)
  Diese Prüfsumme ermöglicht, sowohl im UDP-Paket als auch in einigen Angaben im IP-Header, die den *IP-Pseudo-Header* bilden, Übertragungsfehler zu entdecken [Abb. 4.2-2].

Mit der Angabe `Source Port` und `Destination Port` im UDP-Header und der Angabe von `Source IP Address` und `Destination IP Address` im IP-Header werden die beiden Endpunkte (d.h. die Sockets) festgelegt, zwischen denen der Datenaustausch stattfindet [Abb. 4.1-1b].

UDP-Prüfsumme

Die Berechnung der Prüfsumme wird in RFC 1071, RFC 1141 und RFC 1624 detailliert spezifiziert, dessen Nutzung in der UDP-Spezifikation RFC 768 jedoch nicht gefordert.

IP-Pseudo-Header

Die Prüfsumme im UDP-Header deckt auch einige Angaben im IP-Header ab, die den *IP-Pseudo-Header* bilden. Abb. 4.2-2 zeigt dies.

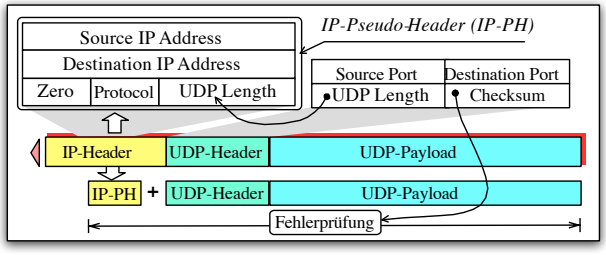


Abb. 4.2-2: Prüfsumme im UDP-Header und Angaben im IP-Pseudo-Header (IP-PH)

Die Angaben im IP-Pseudo-Header sind:

- `Source IP Address` (*IP-Quelladresse*), d.h. Quelladresse aus dem IP-Header,

- `Destination IP Address` (*IP-Zieladresse*), d.h. Zieladresse aus dem IP-Header,
- `Zero`: Der Wert 0 wird hier eingetragen.
- `Protocol`: Es wird die Protokollnummer 17 von UDP angegeben [Tab. 3.1-1].
- `UDP Length` (UDP-Paketlänge): Die Länge des UDP-Pakets wird mitgeteilt. Dieser Wert ist aus dem Feld `UDP Length` im UDP-Header übernommen.

Die Konstruktion des IP-Pseudo-Header kann nur als historisch betrachtet werden. Einerseits verstößt sie elementar gegen das Schichtenprinzip für den Kommunikationsablauf, andererseits kann sie keines der angestrebten Ziele wie beispielsweise Identifikation von 'fremden' UDP-Paketen in einer Kommunikationsbeziehung garantieren. Daher wird dieses Konstrukt bereits bei UDP-Lite aufgegeben.

UDP-Paketgröße 512 Byte (IPv4)

Eine zentrale Einschränkung bei der Kommunikation über UDP besteht in der Notwendigkeit, die maximale Größe der UDP-Pakete auf 512 Byte zu beschränken. Diese Anforderung entstammt nicht der Definition von UDP, sondern hat einen praktischen Hintergrund: RFC 1122 verlangt bei der IP-Kommunikation als minimale Größe des Datenpuffers 576 Byte. Zieht man den (minimal) 20 Byte großen IP-Header ab, bleiben für das UDP-Paket 556 Byte und abzüglich des UDP-Header (8 Byte) noch maximal 548 Byte für Nutzdaten. Diese historisch festgelegte Größe, die Unabhängigkeit einzelner UDP-Pakete voneinander sowie das Fehlen einer Flusskontrolle (wie bei TCP) führen dazu, dass UDP-Pakete in der Regel auf eine Größe von 512 Byte beschränkt sind.

PMTU → UDP-Paketgröße 1280 Byte

Mit dem Einzug von IPv6 und dem Verzicht auf Netzwerke, die lediglich relativ kleine Framegrössen erlauben, ist diese Vorgabe jedoch entfallen und der neue Wert für die *Path Maximum Transmission Unit* Größe – die das gesamte IP-Paket samt Header umfasst – liegt bei mittlerweile 1280 Byte.

4.2.2 Protokoll UDP-Lite

UDP verwendet man für die Übermittlung von Echtzeitdaten wie z.B. Audio oder Video über das Internet bzw. über andere IP-Netze. Im Gegensatz zur Datenkommunikation haben Bitfehler bei Audio- und Videokommunikation oft nur eine geringe negative Auswirkung auf die Qualität der Kommunikation. Ein Bitfehler bei Voice over IP ist sogar für das menschliche Ohr direkt nicht bemerkbar.

Wozu UDP-Lite?

Um die empfangenen UDP-Pakete mit einzelnen Bitfehlern in Audio- bzw. in Videodaten nicht verwerfen zu müssen und damit die Häufigkeit der Verluste von UDP-Paketen mit Audio- und Videodaten zu reduzieren, war eine Modifikation von UDP nötig. Die UDP-Fehlerkontrolle wurde so verändert, dass nur die Bereiche in UDP-Paketen überprüft werden, in denen Übertragungsbitfehler eine große Auswirkung auf die Qualität der Kommunikation haben. Dies hat zur Entstehung von UDP-Lite geführt [RFC 3828], dessen Paketaufbau in Abb. 4.2-3 gezeigt ist.

Der Header von UDP-Lite hat fast die gleiche Struktur wie der von UDP [Abb. 4.2-1]. Im Unterschied zu UDP enthält der Header von UDP-Lite `Checksum Coverage` anstelle der Angabe `UDP Length`. Die Nummern der *Well-known Ports* sind bei UDP-Lite die gleichen wie bei UDP. Damit können sowohl UDP als auch UDP-Lite die gleichen Applikationen nutzen.

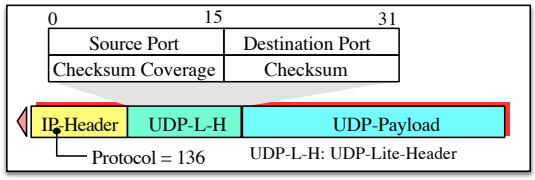


Abb. 4.2-3: Aufbau von UDP-Lite-Paketen

Idee von UDP-Lite

Die Idee von UDP-Lite besteht darin, dass im Header mit `Checksum Coverage` angegeben werden kann, welcher Teil der UDP-Payload bei der Fehlerprüfung berücksichtigt wird. Abb. 4.2-4 illustriert die Idee von UDP-Lite.

Mit `Checksum Coverage` informiert der Sender den Empfänger darüber, welcher Teil von UDP-Payload durch die Prüfsumme (Checksum) abgedeckt wird. `Checksum Coverage` stellt daher die *Reichweite der Prüfsumme* dar.

Falls als UDP-Payload ein RTP-Paket übermittelt wird, kann mit `Checksum Coverage` angegeben werden, dass die Fehlerprüfung beispielsweise nur im RTP-Header stattfinden soll. Daher deckt die Prüfsumme in diesem Fall nur den IP-Pseudo-Header, den UDP-Header und den RTP-Header ab.

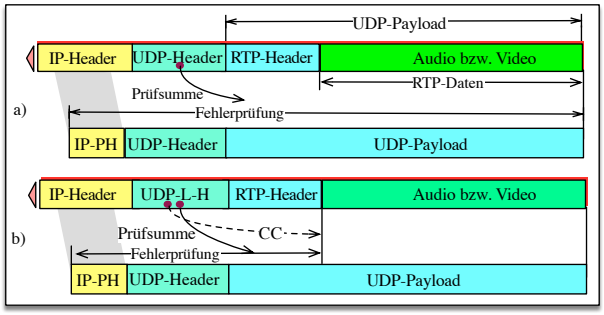


Abb. 4.2-4: Überprüfung von Übertragungsbitfehlern: a) bei UDP, b) bei UDP-Lite
CC: Checksum Coverage, IP-PH: IP-Pseudo-Header, UDP-L-H: UDPLite Header, RTP: Real-time Transport Protocol

Der Wert von `Checksum Coverage` wird von der Applikation auf der Sendeseite bestimmt. Diese Angabe besagt, wie viele Byte, beginnend vom ersten Byte im UDP-Lite-Header, die Prüfsumme abdeckt. Der UDP-Lite-Header muss immer mit der Prüfsumme abgedeckt werden. Ist nur der UDP-Lite-Header mit der Prüfsumme abgedeckt, wird als `Checksum Coverage` der Wert `0` eingetragen. Daher ist `Checksum Coverage` entweder gleich `0` oder größer als `8` (d.h. größer als die Anzahl von Byte im UDP-Header). Ein IP-Paket mit `Checksum Coverage` von 1 bis 7 wird einfach verworfen.

Checksum Coverage

Vergleicht man Abb. 4.2-2 und Abb. 4.2-4, so ist ersichtlich, dass die Angabe `UDP Length` im UDP-Header durch `Checksum Coverage` bei UDP-Lite ersetzt wurde. Dies ist möglich, da die Angabe `UDP Length` im UDP-Header redundant ist und die

Länge des UDP-Pakets aus den Angaben im IP-Header (d.h. `UDP Length = Total Length minus Header Length`) berechnet werden kann.

Das Feld `Checksum` in UDP-Paketen weist lediglich eine Größe von 16 Bit auf. Dies bedeutet Einschränkungen im Hinblick auf den Umfang der erkannten Bitfehler und der möglichen Korrekturen. Daher ist es sinnvoll, nur diejenigen Daten per `Checksum` zu sichern, die von besonderer Bedeutung sind.

4.3 Funktion des Protokolls TCP

TCP ist ein verbindungsorientiertes, zuverlässiges Transportprotokoll, das bereits in RFC RFC 793 spezifiziert wurde und seither ständig weiterentwickelt wird. Die aktuelle Version des Protokolls einschließlich der vielen Neuerung wurden nun in RFC 9293 zusammen gefasst und damit auch weitgehend abgeschlossen.

Seine Bedeutung kommt in Abb. 4.3-1 zum Ausdruck, wobei deutlich wird, dass mittels TCP mehrere Applikationen auf die Dienste von IP gleichzeitig zugreifen können. Daher kann TCP auch als *logischer Multiplexer* von Applikationen angesehen werden.

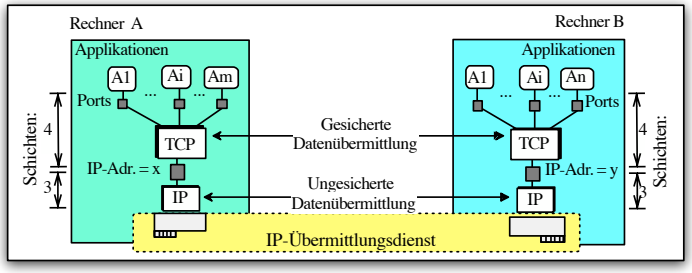


Abb. 4.3-1: TCP als Sicherungsprotokoll zwischen zwei entfernten Rechnern

TCP als Kontrollprotokoll

Im Gegensatz zum UDP-Paket – das als reiner Datencontainer aufgefasst werden kann – beinhalten TCP-Pakete umfangreiche Steuerungsinformationen, die zum geregelten Ablauf der Verbindung eingesetzt werden, aber teilweise auch für die auf TCP aufbauenden Applikationen genutzt werden können. Die Steuerungsinformationen werden bei Bedarf im TCP-Header im Optionsfeld (`Options`) untergebracht. TCP hat sich im Laufe seiner Entwicklung durch den Einsatz neuer sowie die Umdefinition bestehender Optionen wie ein Chamäleon an veränderte Gegebenheiten angepasst.

TCP-Verbindungen

Die Zuverlässigkeit von TCP wird dadurch gewährleistet, indem sich die TCP-Instanzen gegenseitig über ihren jeweiligen Zustand (*Status*) informieren, also eine *virtuelle Verbindung* unterhalten. Um die Datenübermittlung nach TCP zu gewährleisten, muss daher zunächst eine *TCP-Verbindung* aufgebaut werden. Wenn der Nutzdatenfluss ausschließlich von *A* nach *B* geht, existiert immer auch eine Kontrollverbindung von *B* nach *A*. Typischerweise hat aber auch die Gegenseite etwas mitzuteilen, wodurch auch Nutzdaten von *B* nach *A* übertragen werden. In diesem Fall werden die Kontrollinformationen in den (beidseitigen) Nutzpaketen untergebracht. TCP realisiert eine *Vollduplex-Verbindung* zwischen den Kommunikationspartnern

mit einem ausgeklügelten und effizienten *In-Band-Steuerungsverfahren* [Abb. 4.1-1]. Im Vergleich hierzu stellt beispielsweise ICMP für IPv4 [Abschnitt 3.7] einen Out-of-Band-Kontroll- und Fehlermechanismus bereit; bei UDP existiert hingegen gar keiner.

4.3.1 Aufbau von TCP-Paketen

Arten von TCP-Paketen

Über eine TCP-Verbindung werden die Daten in Form von festgelegten Datenblöcken – von nun an *TCP-Pakete* bzw. *TCP-PDUs* genannt – ausgetauscht. Typischerweise treten bei einer TCP-Verbindung zwei Arten von Paketen auf:

- TCP-Pakete mit Nutzdaten – als *TCP-Nutzlast* – und mit Steuerungsinformationen im TCP-Header [Abb. 1.4-2]. Zusätzlich zur eigentlichen Übermittlung von Nutzdaten sind Angaben zur Realisierung der Flusskontrolle nach dem *Sliding-Window-Prinzip* [Abb. 4.4-2] im TCP-Header dieser Pakete enthalten.
- TCP-Pakete ausschließlich mit Steuerungsinformationen. Deren Aufgabe ist das Regeln des Auf- und Abbaus von TCP-Verbindungen sowie die Überprüfung ihrer Fortdauer in Form von *Keep-Alives*, falls längere Zeit keine Datenübermittlung erfolgte.

Nummerierung von Byte

Ist eine TCP-Verbindung aufgebaut, besteht die zentrale Aufgabe von TCP darin, die zu übertragenden Daten bzw. Nachrichten vom Quellrechner *byteweise zu nummerieren* und diese in Form von *Datensegmenten* zu übermitteln. Die Nummer des ersten Nutzbyte im TCP-Datencontainer ist die laufende *Sequenznummer* und bezeichnet somit die Anzahl aller gesendeten Nutzdatenbyte der vorigen Segmente (ohne Wiederholungen); eine Methode, die als *Byte-stream* bezeichnet wird [Abb. 4.3-8].

Flusskontrolle

Die Partnerinstanz quittiert die erhaltenen Daten, indem sie als *Quittung* (*Acknowledgement*) den letzten Wert der *Sequenznummer* (`SEQ`) plus 1 – also `SEQ + 1` – zurückgibt [Abb.4.3-8]. Damit dieser Mechanismus funktioniert, wird vor Beginn einer Übermittlung zwischen den TCP-Instanzen im Quell- und im Zielrechner entsprechend der anliegenden Datenmenge die maximale Segmentgröße vereinbart (z.B. 1000 Byte). Diese und weitere Angaben werden im *TCP-Header* eingetragen.

Garantie der Reihenfolge

Die TCP-Instanz im Zielrechner setzt die empfangenen Datensegmente mittels der übertragenen Sequenznummern in der richtigen Reihenfolge in die ursprünglichen Daten zurück. Erreicht ein TCP-Paket den Zielrechner nicht, wird die Wiederholung der Übertragung der fehlenden Datensegmente durch die TCP-Partnerinstanz veranlasst.

TCP-Header

Abb. 4.3-2 zeigt den Aufbau des TCP-Header.

Die einzelnen Angaben im TCP-Header haben folgende Bedeutung:

- `Source Port` (Quellport)
  Hier wird die Nummer des Ports der Applikation im Quellrechner angegeben, die die TCP-Verbindung initialisiert hat.
- `Destination Port` (Zielport)
  Hier wird die Nummer des Ports dieser Applikation im Zielrechner angegeben, an die die Daten adressiert sind.

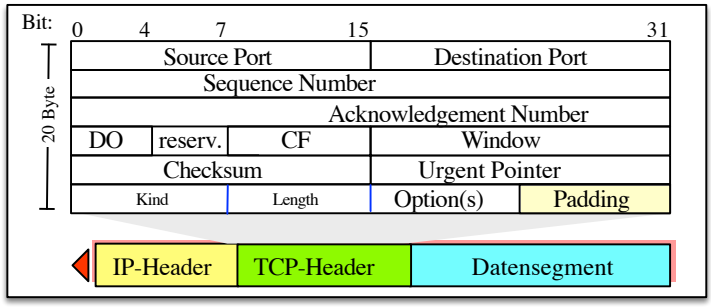


Abb. 4.3-2: Aufbau des TCP-Header
CF: Control Flags, DO: Data Offset

Bedeutung von ISN

- `Sequence Number` (Sequenznummer)
  Die Sequenznummer bezieht sich auf den Sender und dient diesem zur Byte-weisen Nummerierung der gesendeten Daten. Beim Aufbau der TCP-Verbindung generiert jede TCP-Instanz eine Anfangssequenznummer `ISN` (*Initial Sequence Number*). Diese Nummern werden ausgetauscht und gegenseitig bestätigt [Abb. 4.3-4]. Um die gesendeten TCP-Pakete eindeutig am Zielrechner zu identifizieren, muss der Fall ausgeschlossen werden, dass sich zu einem Zeitpunkt im Netz mehrere Segmente mit der gleichen Sequenznummer befinden. Aus diesem Grund darf sich die Sequenznummer innerhalb der festgelegten *Lebenszeit* (*Time to Live*) für die IP-Pakete nicht wiederholen [Abb. 2.2-1]. Im Quellrechner wird die Sequenznummer immer jeweils um die Anzahl bereits gesendeter Byte erhöht.
  Die `ISN` wird in der Regel nicht von 0 hochgezählt, sondern bei aktuellen TCP-Implementierungen zufällig erzeugt. Hierdurch kann in gewissem Umfang gewährleistet werden, dass ein Kommunikationsendpunkt zum Zeitpunkt der Initialisierung über eine weitgehend einmalige ISN verfügt, die somit neben dem Socket (*IP-Adresse, Portnummer*) ein zusätzliches Verbindungsmerkmal darstellt.
- `Acknowledgement Number` (Quittungsnummer)
  Die Quittungsnummer wird vom Empfänger vergeben und dient zur Bestätigung der empfangenen Daten, indem dem Quellrechner mitgeteilt wird, bis zu welchem Byte (*Sequenznummer*) die Daten korrekt empfangen wurden.
- `Data Offset` (Datenabstand)
  Das vier Bit große Feld gibt die Länge des TCP-Header in 32-Bit-Worten an und damit die Stelle, ab der die Daten beginnen. Dieses Feld kann auch als `Header Length` interpretiert werden.
- Reserved (Reserviert)
  Diese drei Bit sind reserviert und üblicherweise auf `0` gesetzt.
- `Control Flags` (Kontrollflags)
  Die Kontrollflags legen fest, welche Felder im Header gültig sind und dienen zur Verbindungssteuerung. Zurzeit sind 9 Bit vorgesehen. Ist das entsprechende Bit gesetzt, gilt Folgendes:
  - `NS` (*ECN-nonce Concealment Protection*) [RFC 3540]: Unterstützung des ECN-Verfahrens [Abschnitt 4.5] mit einem Nonce-Bit.
  - `CWR` (*Congestion Window Reduced*): Einsatz bei der Überlastkontrolle nach `ECN` (*Explicit Congestion Notification*) [RFC 3168].

- ECE (*ECN-Echo*): Einsatz bei der Überlastkontrolle nach ECN [RFC 3168].
- URG: Der *Urgent Pointer* (Zeiger im Urgent-Feld) ist vorhanden und gültig.
- ACK: Die *Quittungsnummer* ist vorhanden und gültig.
- PSH (*Push-Funktion*): Die Daten sollen sofort an die nächsthöhere Schicht weitergegeben werden, ohne nächste Segmente abzuwarten. UNIX-Implementierungen senden PSH immer dann, wenn sie mit dem aktuellen Segment gleichzeitig alle Daten im Sendepuffer übergeben.
- RST (*Reset*): Die TCP-Verbindung soll zurückgesetzt werden.
- SYN: Aufbau einer TCP-Verbindung und ist mit einem ACK zu quittieren [Abb. 4.3-4].
- FIN: Einseitiger Abbau einer TCP-Verbindung und das Ende des Datenstroms aus dieser Richtung, das mit einem ACK quittiert werden muss [Abb. 4.3-5].

- **Window** (Fenstergröße)

  Window für Flusskontrolle

  Diese Angabe dient der Flusskontrolle nach dem Fenstermechanismus [Abb.4.3-6]. Das Feld Window gibt die Fenstergröße an, d.h. wie viele Byte – beginnend ab der Quittungsnummer – der Zielrechner in seinem Aufnahmepuffer noch aufnehmen kann. Empfängt der Quellrechner eine TCP-PDU mit der Fenstergröße gleich 0, muss der Sendevorgang gestoppt werden. Wie die Fenstergröße die Effizienz der Übermittlung beeinflussen kann, ist aus Abb. 4.4-1b ersichtlich (Senderblockade).

- **Checksum** (Prüfsumme)

  Diese Prüfsumme erlaubt es, den TCP-Header, die Daten und einen Auszug aus dem IP-Header (u.a. Quell- und IP-Zieladresse), der an die TCP-Instanz zusammen mit den Daten übergeben wird, auf das Vorhandensein von Bitfehlern zu überprüfen. Bei Berechnung der Prüfsumme wird dieses Feld selbst als null angenommen.

- **Urgent Pointer** (Urgent-Zeiger)

  TCP ermöglicht es, wichtige (dringliche) und meist kurze Nachrichten (z.B. Interrupts) den gesendeten normalen Daten hinzuzufügen und an Kommunikationspartner direkt zu übertragen. Damit können außergewöhnliche Zustände signalisiert werden. Derartige Daten werden als *Urgent-Daten* bezeichnet. Ist der Urgent-Zeiger gültig, d.h. URG = 1, so zeigt er auf das Ende von Urgent-Daten im Segment. Diese werden immer direkt im Anschluss an den TCP-Header übertragen. Erst danach folgen normale Daten.

- **Options**

  TCP erlaubt es, Service-Optionen anzugeben. Das erste Byte in jeder Option legt den *Optionstyp* (*Kind*) fest, das zweite Byte die Länge (*Length*) der entsprechenden Option. Hierdurch können mehrere Optionen zugleich angegeben werden. Der experimentelle RFC 6994 geht noch einen Schritt weiter und erlaubt in den nächsten 2 Byte die Angabe einer *Experimental-ID* (ExID). Der Einsatz der Message-Digest und Authentication Option wird in Abschnitt 11.4 besprochen.

  Das Options-Feld ist entsprechend Tab. 4.3-1 zu interpretieren. In den RFCs 1323 und 2018 wurde die Bedeutung des Option-Feldes folgendermaßen festgelegt:

Options-Type	Option-Feldlänge [Byte]	Bedeutung
0	nicht vorgesehen	Ende der Optionsliste
1	nicht vorgesehen	No-Operation
2	4	Maximum Segment Size (MSS)
3	3	Window Scale (WSopt)
4	2	SACK erlaubt
5	variabel	SACK
8	10	TimeStamp (TSOpt)
11*)	6	Connection Count CC (bei T/TCP)
12*)	6	CC.NEW (T/TCP)
13*)	6	CC.ECHO (T/TCP)
19	18	TCP Message Digest
29	≥ 4	TCP-Authentication

Tab. 4.3-1: Verwendung möglicher Optionen im `Options`-Feld des TCP-Headers https://www.iana.org/assignments/tcp-parameters; *) sind *historische* Optionen, die nicht mehr genutzt werden

- `Maximum Segment Size` (`MSS`): Diese Option wird beim Aufbau einer TCP-Verbindung genutzt. Der Client teilt dem Kommunikationspartner im `<SYN>`-Paket und der Server dies `<SYN,ACK>`-Paket mit. Die gewählte, maximale Segmentgröße hängt von der `MTU` des Links ab, und es gilt:

  ```
  MSS = MTU - IP-Header-Länge - TCP-Header-Länge
  ```

- `Window Scale` (`WSopt`): Mit `WSopt` können die Kommunikationspartner während des Aufbaus einer TCP-Verbindung (also im `SYN`-Paket) festlegen, ob die Größe des 16 Bit `Window` um einen konstanten Skalenfaktor multipliziert wird. Dieser Wert kann unabhängig für den Empfang und das Versenden von Daten ausgehandelt werden. Als Folge dessen wird nun die Fenstergröße von der TCP-Instanz nicht mehr als 16 Bit-, sondern als 32 Bit-Wert aufgefasst. Der maximale Wert für den Skalenfaktor von `WSopt` beträgt 14, was einer neuen oberen Grenze für `Window` von 1 GByte entspricht.
- `Timestamps Option` (`TSopt`):
  Dieses Feld besteht aus den Teilen `Timestamp Wert` (`TSval`) und `Timestamp Echo Reply` (`TSecr`), die jeweils eine Länge von 4 Byte aufweisen. Eingetragen wird hier der *Tickmark*, ein interner Zähler, der pro Millisekunde um eins erhöht wird. `Timestamps Option` kann nur im ersten `SYN`-Paket angezeigt werden, das Feld ist nur bei `ACK`-PDUs erlaubt. Genutzt wird `TSopt` zur besseren Abschätzung der `RTT` (*Round Trip Time*) und zur Realisierung von `PAWS` (*Protect Against Wrapped Sequences*) [RFC 1323].
- Mit RFC 2018 wurde das Verfahren *Selective Acknowledgement* (`SACK`) eingeführt, das sich wesentlich auf das Optionsfeld `SACK` stützt und es zulässt, dieses Feld variabel zu erweitern. Auf dieses Verfahren wird später näher eingegangen.
- Ergänzt werden die Optionen um `Connection Count` (`CC`) sowie `CC.NEW` und `CC.ECHO`, die bei der Implementierung von *T/TCP* anzutreffen sind [Abschnitt 4.4].
- Bei Verwendung von *Multipath TCP* [Abschnitt 5.6] werden noch weitere TCP-`Options` genutzt [Abb. 4.6-6].

- `Padding` (Füllzeichen)
  Die Füllzeichen ergänzen die Optionsangaben auf die Länge von 32 Bit.

TCP Timeouts

TCP verhindert den gleichzeitigen Aufbau einer TCP-Verbindung seitens der beiden Instanzen, d.h. nur eine Instanz kann den Aufbau initiieren. Des Weiteren ist es nicht möglich, einen mehrfachen Aufbau einer TCP-Verbindung durch den Sender

aufgrund eines *Timeout* des ersten Verbindungsaufbauwunsches zu generieren. Der Datenaustausch zwischen zwei Stationen erfolgt erst nach dem Verbindungsaufbau. Registriert der Sender, dass der Empfänger nach Ablauf eines *Timeout* die übertragenen Daten nicht bestätigt hat, wird eine Wiederholung der nicht-quittierten Segmente gestartet. Aufgrund der Sequenznummer ist es prinzipiell möglich, $2^{32} - 1$ Datenbyte (4 Gigabyte) pro bestehender TCP-Verbindung zu übertragen, innerhalb dessen doppelt übertragene Daten erkannt werden. Bei den meisten Implementierungen ist die Sequenznummer aber als *signed Integer* deklariert, sodass nur die Hälfte, d.h. 2 Gigabyte möglich sind.

TCP-Window

Die Flusskontrolle nach dem Fenstermechanismus (*Window*) erlaubt es einem Empfänger, dem Sender mitzuteilen, wie viel Pufferplatz zum Empfang von Daten zur Verfügung steht. Ist der Empfänger zu einem bestimmten Zeitpunkt der Übertragung einer höheren Belastung ausgesetzt, signalisiert er dies dem Sender über das `Window`-Feld, sodass dieser die Senderate reduzieren kann.

4.3.2 Konzept der TCP-Verbindungen

Three-Way Handshake

Eine TCP-Verbindung wird mit dem Ziel aufgebaut, einen zuverlässigen Datenaustausch zwischen den kommunizierenden Anwendungsprozessen in entfernten Rechnern zu gewährleisten. Die TCP-Verbindungen sind vollduplex. Man kann eine TCP-Verbindung als ein Paar von gegenseitig gerichteten unidirektionalen Verbindungen zwischen zwei Sockets interpretieren [Abb. 4.1-1c]. Der Aufbau einer TCP-Verbindung erfolgt immer mittels des *Three-Way Handshake* (*3WHS*)-Verfahrens, das für eine Synchronisation der Kommunikationspartner sorgt und gewährleistet, dass die TCP-Verbindung in jede Richtung korrekt initialisiert wird. An dieser Stelle ist hervorzuheben, dass die Applikation im Quellrechner mit TCP über einen wahlfreien Port kommuniziert, der dynamisch zugewiesen wird.

TCP-Zustandsmodell

Das TCP-Modell geht von einer *Zustandsmaschine* aus. Eine TCP-Instanz befindet sich immer in einem wohldefinierten Zustand. Die Hauptzustände sind `Listen` und (Verbindung-) `Established`. Zwischen diesen stabilen Zuständen gibt es gemäß Abb.4.3-3 eine Vielzahl zeitlich befristeter (Zwischen-)Zustände. Mittels der Kontroll-Flags `ACK`, `FIN`, `SYN` und ggf. auch `RST` wird zwischen den Kommunikationspartnern der Wechsel zu bzw. der Verbleib in einem Zustand signalisiert.

Fenstermechanismus

Um die Menge der auf einer TCP-Verbindung übertragenen Daten zwischen den kommunizierenden Rechnern entsprechend abzustimmen, was man als Flusskontrolle bezeichnet, kommt der *Fenstermechanismus* (Window) zum Einsatz. Zur effizienten Nutzung des Fenstermechanismus stehen zwei Parameter zur Verfügung, die zwischen den TCP-Instanzen im Verlauf der Kommunikation dynamisch angepasst werden. Es handelt sich hierbei um: `Windowsize` und `Maximum Segment Size`.

Interpretation von Windowsize

`Windowsize` (`WSIZE`) ist als die Größe des TCP-Empfangspuffers in Byte zu interpretieren. Aufgrund des maximal 16 Bit großen Feldes im TCP-Header kann dieses maximal einen Wert von $2^{16} - 1 = 65535$ Byte (d.h. rund 64 KByte) aufweisen. Moderne TCP-Implementierungen nutzen allerdings die im TCP-Header vorgesehene Option des *WSopt*, sodass nun Werte bis 2^{30}, also rund 1 GByte, möglich sind.

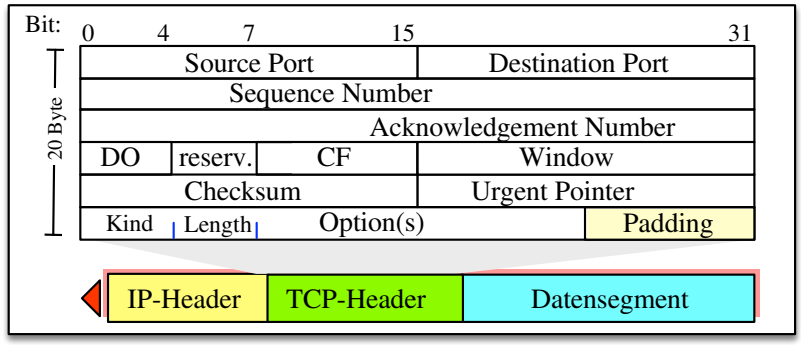


Abb. 4.3-3: TCP-Zustandsdiagramm und gestattete Übergänge zwischen den Zuständen
E: Empfänger, S: Sender, A: Applikation

`Windowsize` als Konfigurationsparameter der TCP-Implementierung ist üblicherweise auf einen Wert von 4, 8, 16 oder 32 KByte initialisiert. Beim Verbindungsaufbau teilt die TCP-Empfängerinstanz dem Sender ihre *Windowsize* mit, was als *advertised Windowsize* (*advWind*) bezeichnet wird.

Interpretation von Maximum Segment Size

`Maximum Segment Size` (`MSS`) stellt das Gegenstück zu `WSIZE` dar, ist also die Größe des TCP-Sendepuffers für die zu übertragenden Daten. Für übliche TCP-Implementierungen gilt die Ungleichung `MSS < WSIZE`. Die dynamische Aushandlung dieser Parameter zusammen mit der Methode der Bestimmung der *Round Trip Time* (`RTT`) begründet ursächlich das gute Übertragungsverhalten von TCP in sehr unterschiedlichen Übermittlungsnetzen (siehe Abschnitt 4.4).

4.3.3 Auf- und Abbau von TCP-Verbindungen

Den Aufbau einer TCP-Verbindung zeigt Abb. 4.3-4. Hier soll insbesondere zum Ausdruck gebracht werden, dass eine TCP-Verbindung vollduplex ist und als ein Paar von zwei unidirektionalen logischen Verbindungen gesehen werden kann [Abb. 4.3-1c]. Die Kommunikationspartner befinden sich zum Anfang der Übertragung immer in folgenden Zuständen [Abb. 4.3-3]:

- *Passives Öffnen* (`Listen`): Eine Verbindung tritt in den Empfangsstatus ein, wenn eine Anwendungsinstanz TCP mitteilt, dass sie Verbindungen für eine bestimmte Portnummer annehmen möchte.
- *Aktives Öffnen* (`SYN sent`): Eine Applikation teilt dem TCP mit, dass sie eine Verbindung mit einer bestimmten IP-Adresse und Portnummer eingehen möchte, was mit einer bereits abhörenden Applikation korrespondiert.

Beispiel: FTP

In Abb. 4.3-4 wird die TCP-Verbindung im Rechner *A* mit der IP-Adresse `x` durch die Applikation FTP initiiert. Hierbei wird ihr für die Zwecke der Kommunikation beispielsweise die Portnummer 3028 zugewiesen. Die TCP-Instanz im Rechner *A* generiert ein TCP-Paket, in dem das Flag `SYN` gesetzt ist. Somit wird dieses Paket hier als `<SYN>`-Paket bezeichnet. Der Verbindungsaufbau beginnt damit, dass jeder der beiden Kommunikationspartner zunächst einen Anfangswert für die jeweilige

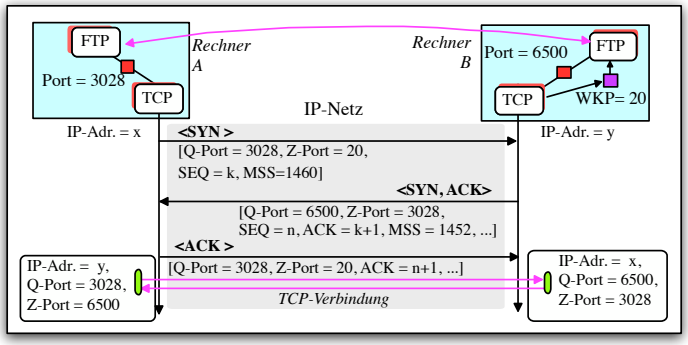


Abb. 4.3-4: Beispiel für den Aufbau einer TCP-Verbindung
Q: Quell, Z: Ziel, WKP: Well-known Port

Sequenznummer wählt. Dieser Anfangswert für eine Verbindung wird als *Initial Sequence Number* (ISN) bezeichnet.

<SYN>-Paket

Die TCP-Instanz im Rechner *A* sendet dazu an den Rechner *B* ein <SYN>-Paket, in dem u.a. folgende Informationen enthalten sind:

- SYN-Flag im TCP-Header wird gesetzt. Damit ist der Name <SYN>-Paket zu begründen.
- frei zugeteilte Nummer des Quellports; hierzu werden besonders die Ports[1] zwischen 5000 und 64000 benutzt.
- Zielport als *Well-known Port*, um die richtige Anwendung 'anzusprechen'. Ein *Well-known Port* ist ein Kontaktport (*Contact Port*) einer Applikation und kann als ihr Begrüßungsport angesehen werden.
- SEQ: *Sequenznummer* der Quell-TCP-Instanz (hier SEQ = k).

Das gesetzte SYN-Bit bedeutet, dass die Quell-TCP-Instanz eine Verbindung aufbauen (synchronisieren) möchte. Mit der Angabe des Zielports (als *Well-known Port*) wird die gewünschte Applikation im Rechner *B* gefordert.

Reaktion des Zielrechners auf <SYN>-Paket

Die Ziel-TCP-Instanz befindet sich im *Listenmodus*, sodass sie auf ankommende <SYN>-Pakete wartet. Nach dem Empfang eines <SYN>-Pakets leitet die Ziel-TCP-Instanz ihrerseits den Verbindungswunsch an die Ziel-Applikation (hier FTP) gemäß der empfangenen Nummer des Zielports weiter. Falls die Applikation im Zielrechner die ankommende TCP-Verbindung akzeptiert, wird ihr ein Port als Puffer für zu sendende und zu empfangende Daten eingerichtet. Diesem Port wird eine große Nummer (z.B. 6500) zugeteilt. Danach wird eine eigene ISN in Richtung Rechner *A* generiert.

Im zweiten Schritt des Verbindungsaufbaus wird ein TCP-Paket im Rechner B mit folgendem Inhalt an den Rechner *A* zurückgeschickt:

<SYN,ACK>-Paket

- Die beiden Flags SYN und ACK im TCP-Header werden gesetzt, sodass man von einem <SYN,ACK>-Paket spricht.

[1] Ports unter 1000 sind für Server-Anwendungen reserviert; die hohen Portnummern stehen für Client-Anwendungen zur Verfügung.

- Die beiden Quell- und Zielportnummern werden angegeben. Als Quellport wird der neu im Rechner *B* eingerichtete Port 6500 angegeben. Als Zielport wird der Quellport im Rechner *A* eingetragen.
- Die Sequenznummer der Ziel-TCP-Instanz (hier `SEQ = n`) wird mitgeteilt.

Reaktion auf <SYN,ACK>-Paket

Das `ACK`-Bit signalisiert, dass die Quittungsnummer (hier kurz `ACK`) im `<SYN,ACK>`-Paket von Bedeutung ist. Die *Quittungsnummer* `ACK` enthält die nächste, von der TCP-Instanz im Rechner *B* erwartete Sequenznummer. Die TCP-Instanz im Rechner *A* bestätigt noch den Empfang des `<SYN,ACK>`-Pakets mit einem `<ACK>`-Paket, in dem das `ACK`-Flag gesetzt wird. Mit der *Quittungsnummer* `ACK = n + 1` wird der TCP-Instanz im Rechner *B* bestätigt, dass die nächste Sequenznummer `n + 1` erwartet wird.

`MSS` in <SYN>-Paketen

Zusätzlich teilt Rechner *A* die maximale Größe eines TCP-Segments Rechner *B* mit einem Wert von `MSS = 1460` (Byte) mit, wohingegen Rechner *B* in der Gegenrichtung einen Wert von `MSS = 1452` Byte vorschlägt, da *B* im TCP-Header noch (nicht gezeigte) Optionen übermittelt.

Aus Abb. 4.3-4 geht außerdem hervor, dass sich eine TCP-Verbindung aus zwei *unidirektionalen Verbindungen* zusammensetzt. Jede dieser gerichteten Verbindungen wird im Quellrechner durch die Angabe der IP-Zieladresse und der Quell- und Zielports eindeutig identifiziert.

Wurde eine TCP-Verbindung aufgebaut, so kann der Datenaustausch zwischen den kommunizierenden Applikationen erfolgen, genauer gesagt zwischen den mit der TCP-Verbindung logisch verbundenen Ports. Bevor wir aber auf die Besonderheiten der Datenübermittlung nach dem Protokoll TCP eingehen, soll zunächst der Abbau einer TCP-Verbindung kurz erläutert werden.

Abbau einer TCP-Verbindung

Den Abbau einer TCP-Verbindung illustriert Abb. 4.3-5. Im Normalfall kann der Abbau von einer der beiden kommunizierenden Applikationen initiiert werden. Da jede TCP-Verbindung sich aus zwei gerichteten Verbindungen zusammensetzt, werden diese gerichteten Verbindungen quasi nacheinander abgebaut. Jede TCP-Instanz koordiniert den Abbau ihrer gerichteten Verbindung zu ihrer Partner-TCP-Instanz und verhindert hierbei den Verlust von übertragenen, aber noch nicht quittierten Daten.

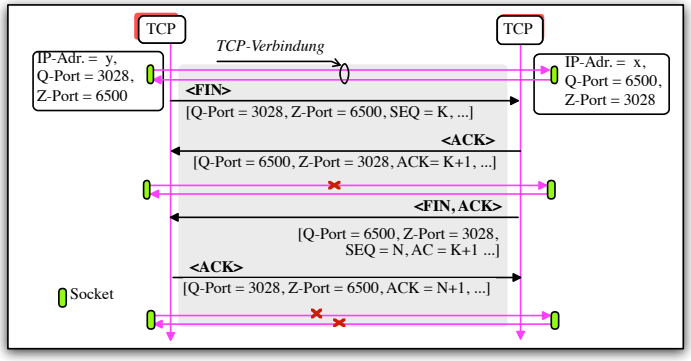


Abb. 4.3-5: Beispiel für den Abbau einer TCP-Verbindung

Maximum Segment Lifetime

Der Abbau wird von einer Seite mit einem TCP-Paket initiiert, in dem das FIN-Flag im Header gesetzt wird, sodass man dieses TCP-Paket als <FIN>-Paket bezeichnet. Dies wird von der Gegenseite durch ein <ACK>-Paket mit dem gesetzten ACK-Flag positiv bestätigt. Die positive Bestätigung erfolgt hier durch die Angabe der Quittungsnummer ACK = K + 1, d.h. der empfangenen Sequenznummer SEQ = K plus 1. Damit wird eine gerichtete Verbindung abgebaut. Der Verbindungsabbau in der Gegenrichtung wird mit dem TCP-Paket begonnen, in dem die beiden FIN- und ACK-Flags gesetzt sind, d.h. mit dem <FIN,ACK>-Paket. Nach der Bestätigung dieses <FIN,ACK>-Pakets durch die Gegenseite wird der Abbauprozess beendet.

Beim Abbau einer Verbindung tritt u.U. ein zusätzlicher interner *Time-out-Mechanismus* in Kraft. Die TCP-Instanz geht in den Zustand *Active-Close*, versendet ein abschließendes <ACK>-Paket und befindet sich dann im Status *Time-Wait* [Abb.4.3-3]. Dessen Zeitdauer beträgt 2 * MSL (*Maximum Segment Lifetime*), bevor die TCP-Verbindung letztlich geschlossen wird. TCP-Pakete, die länger als die MSL-Dauer im Netz unterwegs sind, werden verworfen. Der Wert von *MSL* beträgt bei heutigen TCP-Implementierungen in der Regel 120 Sekunden. Anschließend wird der Port freigegeben und steht (mit einer neuen ISN) für spätere Verbindungen wieder zur Verfügung.

4.3.4 Flusskontrolle bei TCP

Sliding-Window-Prinzip

Bei der Datenkommunikation muss die Menge der übertragenen Daten an die Aufnahmefähigkeit des Empfängers angepasst werden. Sie sollte nicht größer sein als die Datenmenge, die der Empfänger aufnehmen kann. Daher muss die Menge der übertragenen Daten zwischen den kommunizierenden Rechnern entsprechend abgestimmt werden. Diese Abstimmung bezeichnet man oft als *Datenflusskontrolle* (*Flow Control*). Sie erfolgt beim TCP nach dem Prinzip *Sliding Window*. In Abschnitt 1.2 wurde bereits das Window-Prinzip (*Fensterprinzip*) bei der Nummerierung nach dem Modulo-8-Verfahren kurz erläutert. Bevor auf die Besonderheiten der Flusskontrolle beim TCP eingegangen wird, soll das allgemeine *Sliding-Window-Prinzip* näher veranschaulicht werden.

Für die Zwecke der Flusskontrolle nach dem Sliding-Window-Prinzip dienen folgende Angaben im TCP-Header:

- *Sequence Number* (Sequenznummer),
- *Acknowledgement Number* (Quittungs- bzw. Bestätigungsnummer),
- *Window* (Fenstergröße).

Sequenznummer

Mit der Sequenznummer SEQ werden die zu sendenden Daten fortlaufend byteweise nummeriert. Sie besagt, mit welcher Nummer die Nummerierung der im TCP-Paket gesendeten Byte beginnen soll [Abb. 4.3-8].

Quittungsnummer und Window

Mit der Quittungsnummer teilt der Empfänger dem Sender mit, welche Sequenznummer als nächste bei ihm erwartet wird. Hierbei wird die Angabe Window wie folgt interpretiert:

- *Seitens des Senders* stellt die Window-Größe (Fenstergröße) bei TCP die maximale Anzahl von Daten in Byte dar, die der Sender absenden darf, ohne auf eine Quittung vom Empfänger warten zu müssen.
- *Seitens des Empfängers* ist `Window` als Anzahl der Daten in Byte zu verstehen, die von diesem auf jeden Fall immer aufgenommen werden können.

Wird die maximale Länge von Datensegmenten in TCP-Paketen festgelegt, so kann die Menge von Daten unterwegs mit den erwähnten drei Parametern (Sequenz- und Quittungsnummer sowie Window) jederzeit kontrolliert werden.

Flusskontrolle

Abb. 4.3-6 veranschaulicht die Flusskontrolle nach dem Sliding-Window-Prinzip mit der `Window-Größe = 4`. Wie hier ersichtlich ist, lässt sich das Window als Sendefenster interpretieren.

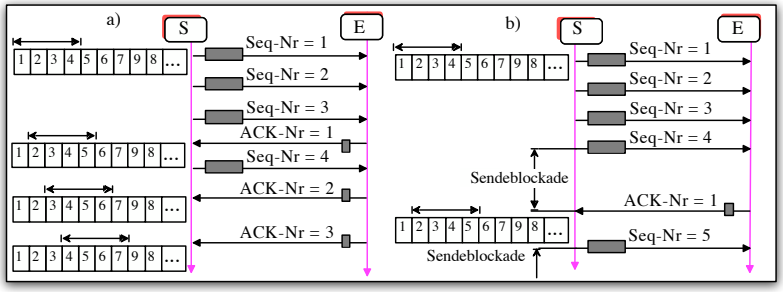


Abb. 4.3-6: Sliding-Window-Prinzip bei Window-Größe = 4:
a) fehlerfreie Übertragung, b) Bedeutung der Sende-Blockade
ACK: Quittungsnummer (Acknowledgement Number), Seq-Nr.: Sequenznummern;
E: Empfänger, S: Sender

Sendeblockade

Betrachten wir zunächst das Beispiel in Abb. 4.3-6a. Da die Window-Größe 4 beträgt, darf der Sender nur 4 Byte absenden, ohne auf eine Quittung warten zu müssen. Dies bedeutet, dass er die Byte mit den Nummern 1, 2, 3 und 4 absenden darf. Nach dem Absenden der ersten drei Byte ist eine Quittung eingetroffen, mit der das erste Byte quittiert wird. Dadurch verschiebt sich das Fenster (*Window*) mit den zulässigen Sequenznummern um eine Position nach rechts. Da maximal 4 Byte unterwegs sein dürfen, kann der Sender nun die nächsten Byte mit den Nummern 4 und 5 senden. Nach dem Absenden des Byte mit Sequenznummer 5 wird das Byte mit der Sequenznummer 2 durch den Empfänger positiv quittiert. Dadurch verschiebt sich das Fenster nach rechts um eine Position weiter.

Abb. 4.3-6b zeigt die Situation, in der der Sendeprozess blockiert werden muss (*Sendeblockade*). Sie kommt oft dann vor, wenn einerseits die Verzögerungszeit im Netz groß und andererseits die Window-Größe zu klein ist. Mit großen Verzögerungszeiten ist immer zu rechnen, wenn eine Satellitenstrecke als ein Übertragungsabschnitt eingesetzt wird. Wie hier ersichtlich ist, muss der Sender nach dem Absenden von Byte mit den Sequenznummern 1, 2, 3 und 4 auf eine Quittung warten. Hier wurden die Daten aus dem Sendefenster abgesendet, und deren Empfang wurde noch nicht bestätigt. Bevor einige Byte quittiert werden, darf der Sender keine weiteren Byte senden. Nach dem Eintreffen der Quittung für das Byte mit Sequenznummer 1 verschiebt sich das Sendefenster um eine Position nach rechts. Das Byte mit der Sequenznummer 5 darf nun

gesendet werden. Anschließend muss der Sendevorgang wiederum bis zum Eintreffen der nächsten Quittung blockiert werden.

4.3.5 TCP Sliding-Window-Prinzip

Bei TCP erfolgt die Flusskontrolle nach dem *Sliding-Window-Prinzip*. Hierbei legt die Window-Größe die maximale Anzahl von Byte fest, die der Quellrechner absenden darf, ohne auf eine Quittung vom Zielrechner warten zu müssen. Abb. 4.3-7 illustriert die Interpretation von Window bei TCP.

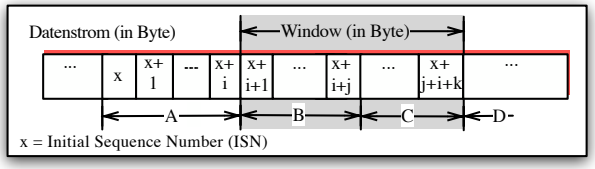


Abb. 4.3-7: Interpretation von Window bei TCP

Byte in Flight

Mit dem Parameter *Window* wird bei TCP ein Bereich von Nummern markiert, die den zu sendenden Datenbyte zuzuordnen sind. Dieser Bereich kann als *Sendefenster* angesehen werden. Im Strom der Datenbyte sind vier Bereiche zu unterscheiden:

- *A*: `i` Datenbyte, die abgesendet und bereits positiv quittiert wurden,
- *B*: `j` Datenbyte, die abgesendet und noch nicht quittiert wurden (’*Data in flight*’),
- *C*: `k` Datenbyte, die noch abgesendet werden dürfen, ohne auf eine Quittung warten zu müssen,
- *D*: Datenbyte außerhalb des Sendefensters. Diese Datenbyte dürfen erst dann abgesendet werden, wenn der Empfang von einigen vorher abgeschickten Datenbyte bestätigt wird.

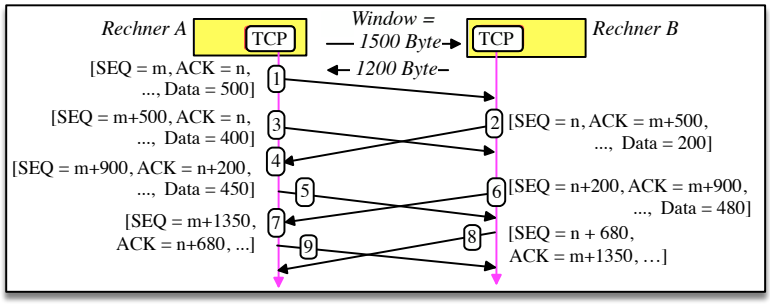


Abb. 4.3-8: Beispiel für den TCP-Ablauf bei der fehlerfreien Datenübermittlung
SEQ: Sequenznummer, ACK: Quittungsnummer

Sliding Window mit fehlerfreier Datenübermittlung

Im Folgenden wird der Datenaustausch nach TCP verdeutlicht und damit auch das Sliding-Window-Prinzip näher erläutert. Abb. 4.3-8 zeigt ein Beispiel für eine fehlerfreie Datenübermittlung.

Die hier dargestellten einzelnen Ereignisse sind wie folgt zu interpretieren:

1. Das erste TCP-Paket von Rechner *A* zu Rechner *B* besitzt die Sequenznummer `SEQ = m`. Dieses Paket enthält die ersten 500 Datenbyte. `SEQ = m` verweist darauf, dass den einzelnen übertragenen Datenbyte die Nummern `m, m+1, ..., m+499` zugeordnet sind.
2. Von Rechner *B* wird mit einem TCP-Paket, in dem 200 Datenbyte enthalten sind und bei dem das `ACK`-Flag gesetzt wurde, bestätigt, dass das erste TCP-Paket von Rechner *A* fehlerfrei aufgenommen wurde und das nächste Datenbyte mit der Nummer `m+500` erwartet wird. `SEQ = n` besagt, dass die Nummern `n, n+1, ..., n+199` den hier übertragenen Datenbyte zuzuordnen sind.
3. Das zweite TCP-Paket von Rechner *A* zu Rechner *B* mit den nächsten 400 Datenbyte und mit `SEQ = m+500`. Somit enthält dieses TCP-Paket die Datenbyte mit den Nummern `m+500, m+501, ..., m+899`. Damit wurden bereits 900 Datenbyte von Rechner *A* abgeschickt und vom Zielrechner *B* noch nicht quittiert. Da die Window-Größe in Richtung zum Rechner *B* 1500 Byte beträgt, können vorerst nur noch `1500 - 900 = 600` Datenbyte abgesendet werden.
4. Nach dem Empfang dieses TCP-Pakets werden 500 Datenbyte vom Zielrechner *B* positiv quittiert. Somit verschiebt sich das Sendefenster im Rechner *A* um 500. Da der Empfang von 400 Byte (das zweite TCP-Paket) noch nicht bestätigt wurde, können noch `1500-400 = 1100` Datenbyte gesendet werden.
5. Das dritte TCP-Paket von Rechner *A* zu Rechner *B* mit `SEQ = m+900` und mit 450 Datenbyte. Dieses Paket enthält die Datenbyte mit den Nummern von `m+900` bis `m+1349`. Somit sind bereits 850 Datenbyte abgeschickt, die noch nicht quittiert wurden. Darüber hinaus können nur noch weitere 650 (d.h. `1500 - 850`) Datenbyte abgesendet werden. Mit diesem TCP-Paket wird dem Rechner *B* auch mitgeteilt, dass die nächsten Datenbyte ab Nummer `n+200` erwartet werden.
6. Rechner *B* quittiert mit einem TCP-Paket, in dem das `ACK`-Flag gesetzt wird, das dritte TCP-Paket von Rechner `A` und sendet zu Rechner *A* die nächsten 480 Datenbyte. Diesen Datenbyte sind die Nummern `n+200, ..., n+679` zuzuordnen.
7. Nach dem Empfang dieses TCP-Pakets werden die an Rechner *A* abgeschickten Datenbyte mit den Nummern `m+900-1` positiv quittiert. Damit verschiebt sich in Rechner *A* das Sendefenster entsprechend.
8. Rechner *B* bestätigt mit einem TCP-Paket, in dem das `ACK`-Flag gesetzt wird, die Datenbyte einschließlich bis zur Nummer `m+1350-1`. Auf diese Weise wurden alle zu Rechner *B* abgeschickten Daten quittiert. Rechner *A* kann nun an Rechner *B* die durch die Window-Größe festgelegte Datenmenge (d.h. 1500 Byte) unmittelbar weitersenden, ohne vorher auf eine positive Quittung von Rechner *B* warten zu müssen.
9. Rechner *A* quittiert mit `ACK = n+680` positiv Rechner *B*, alle Datenbyte bis zur Nummer `n+680-1` empfangen zu haben. Zugleich wird mit `SEQ = m+1350` signalisiert, dass bereits `m+1350` Datenbyte geschickt wurden.

Sendeblockade bei der Datenübermittlung

Fehlerhaftete Datenübertragung

Den Ablauf des Protokolls TCP bei einer fehlerbehafteten Datenübermittlung illustriert Abb. 4.3-9.

Die einzelnen Ereignisse sind hier folgendermaßen zu interpretieren:

1. Auch hier besitzt das erste TCP-Paket von Rechner *A* zu Rechner *B* die `SEQ = m` und liefert die ersten 500 Datenbyte. Diesen Datenbyte sind daher die Nummern `m,`

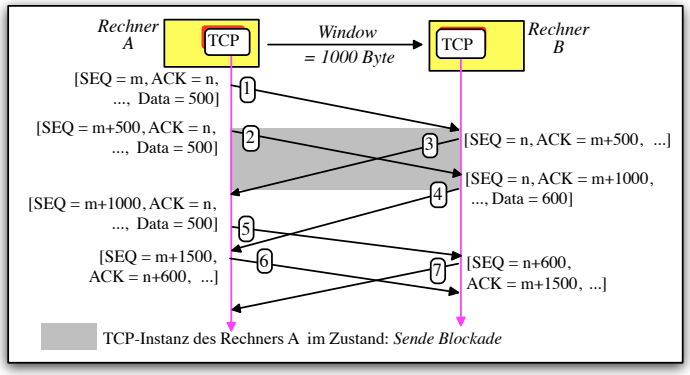


Abb. 4.3-9: Beispiel für das TCP-Verhalten bei einer fehlerbehafteten Datenübermittlung

`m+1, ..., m+499` zuzuordnen. Mit `ACK = n` wird dem Rechner *B* mitgeteilt, dass das nächste Datenbyte mit der Nummer *n* von ihm erwartet wird.

2. Das zweite TCP-Paket von Rechner *A* zu Rechner *B* mit den nächsten `500` Datenbyte und mit `SEQ = m+500`. Somit enthält dieses TCP-Paket die Datenbyte mit den Nummern `m+500, ..., m+999`. Mit dem Absenden dieser 500 Datenbyte wurden die Nummern zur Vergabe der zu sendenden Datenbyte 'verbraucht' (⇒ `Window = 1000` Byte). Aus diesem Grund muss der Sendeprozess blockiert werden.
3. Es werden `500` Datenbyte vom Zielrechner *B* positiv quittiert. Damit verschiebt sich das Sendefenster in Rechner *A* entsprechend, sodass weitere `500` Datenbyte gesendet werden dürfen.
4. Rechner *B* sendet `600` Datenbyte und quittiert dem Rechner *A* alle Datenbyte bis einschließlich Nummer `m+1000-1`.
5. Das dritte TCP-Paket von Rechner *A* zu Rechner *B* mit den nächsten `500` Datenbyte und mit der Sequenznummer `SEQ = m+1000`.
6. Rechner *A* quittiert Rechner *B* alle Datenbyte bis einschließlich Nummer `n+600-1`.
7. Rechner *B* quittiert Rechner *A* alle Datenbyte bis einschließlich Nummer `m+1500-1`.

Fehlerbehaftete Datenübermittlung

Da IP zu den unzuverlässigen Protokollen gehört, muss TCP über Mechanismen verfügen, der es in die Lage versetzt, mögliche Fehler auf den unteren Protokollschichten (z.B. Verlust von IP-Paketen, Verfälschung der Reihenfolge usw.) zu erkennen und zu beheben. Der von TCP verwendete Mechanismus ist bemerkenswert einfach:

> Ist die Wartezeit für *ein* abgesendetes TCP-Segment überschritten, innerhalb derer eine Bestätigung erfolgen sollte (*Maximum Segment Lifetime*), wird die Übertragung *aller* Datenbyte wiederholt, für die bis dahin noch keine Quittungen vorliegen.

Im Unterschied zu anderen Methoden der Fehlerkontrolle kann hier der Empfänger zu keinem Zeitpunkt eine wiederholte Übertragung erzwingen. Dies liegt zum Teil daran, dass kein Verfahren vorhanden ist, um negativ zu quittieren, sodass keine wiederholte Übertragung einzelner TCP-Pakete direkt veranlasst werden kann. Der Empfänger muss einfach abwarten, bis das von vornherein festgelegte Zeitlimit *MSL*

(*Maximum Segment Lifetime*) auf der Sendeseite abgelaufen ist und infolgedessen bestimmte Daten nochmals übertragen werden.

MSL Timer

Das Funktionsweise des MSL-Timer bei TCP zeigt Abb. 4.3-10. Um die Darstellung zu vereinfachen, werden hier nur jene Angaben gezeigt, die nötig sind, um dieses Prinzip zu erläutern.

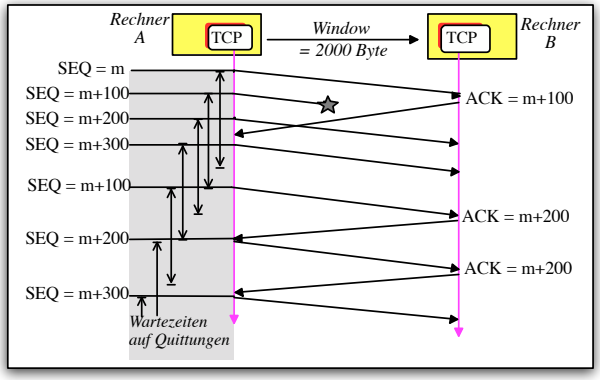


Abb. 4.3-10: Gesteuerte Segmentwiederholung über den TCP MLS Timer
ACK: Quittungsnummer, SEQ: Sequenznummer

Nutzung des MSL-Timers

> Wie in Abb. 4.3-10 dargestellt, wird der MSL-Timer nach dem Absenden jedes TCP-Segments neu gestartet. Mit diesem *Timer* wird eine maximale Wartezeit (*Timeout*) auf die Quittung angegeben. Kommt innerhalb dieser festgelegten maximalen Wartezeit keine Quittung an, wird die Übertragung des betreffenden TCP-Pakets wiederholt. Darin besteht das eigentliche Prinzip der Fehlerkontrolle bei TCP.
>
> Das TCP-Paket mit der Sequenznummer `m+100` hat den Empfänger nicht erreicht, obwohl die später abgeschickten TCP-Pakete (mit den Sequenznummern `m+200` und `m+300`) dort ankamen. Die TCP-Instanz im Rechner *B* sendet keine Bestätigung für den Empfang des TCP-Pakets mit der Sequenznummer `m+200`, da die Datenbyte mit den Nummern `m+100, ..., m+199` noch nicht empfangen wurden. Das nächste TCP-Paket mit `SEQ = 300` wird ebenfalls nicht bestätigt. Dies hat zur Folge, dass das Zeitlimit für die Übertragung des TCP-Pakets mit der Sequenznummer `x+100` abläuft und dieses Paket infolgedessen erneut übertragen wird.

Bestätigung

Der Empfang der TCP-Pakete wird nur dann bestätigt, wenn ihre Reihenfolge vollständig ist. Somit kann die Situation eintreten, dass eine Reihe von TCP-Paketen (soweit die Window-Größe dies zulässt) sogar dann wiederholt übertragen werden muss, wenn sie bereits fehlerfrei beim Zielrechner ankamen. Wie dem Beispiel in Abb. 4.3-10 zu entnehmen ist, betrifft dies in unserem Fall die TCP-Pakete mit den Sequenznummern `m+200` und `m+300`. Diese müssen nochmals übertragen werden.

Round Trip Time

Die maximale Wartezeit auf die Quittung ist ein zentraler Parameter von TCP. Er hängt von der zu erwartenden Verzögerung im Netz ab. Die Verzögerung im Netz kann durch die Messung der Zeit, die bei der Hin- und Rückübertragung zwischen dem Quell- und dem Zielrechner auftritt, festgelegt werden. In der Literatur wird diese Zeit als *Round Trip Time* (`RTT`) bezeichnet. In Weitverkehrsnetzen, in denen

auch Satellitenverbindungen eingesetzt werden, kann es einige Sekunden dauern, bis eine Bestätigung ankommt.

Im Laufe einer Verbindung kann die RTT aufgrund der Netzbelastung schwanken. Daher ist es nicht möglich, einen festen Wert für die maximale Wartezeit auf die Quittung einzustellen. Wird ein zu kleiner Wert gewählt, läuft die Wartezeit ab, bevor eine Quittung eingehen kann und das Segment unnötig erneut gesendet. Wird ein zu hoher Wert gewählt, hat dies lange Verzögerungspausen zur Folge, da die gesetzte Zeitspanne abgewartet werden muss, bevor eine wiederholte Übertragung stattfinden kann. Durch den Verlust eines Segments kann der Datendurchsatz erheblich sinken.

4.4 Implementierungsaspekte von TCP

TCP wurde in der Vergangenheit den sich ändernden Gegebenheiten der Netze (LANs und WANs) angepasst. Dies betrifft nicht die Protokollparameter, die über die Jahre unverändert geblieben sind, sondern vielmehr die Implementierung der Algorithmen in den TCP-Instanzen, d.h. den *TCP-Stack* als Bestandteil der Kommunikationssoftware in Betriebssystemen und Routern. Der TCP-Stack ist so zu optimieren, dass er unter den heute gegebenen Netzen und Anwendungen eine maximale Performance und eine hohe Übertragungssicherheit gewährleistet [RFC 1323, 2001 und 2018].

Die Einsatzgebiete von TCP haben sich durch die Popularität des Internet und den Entwicklungen der lokalen Netze stark erweitert, und es sind insbesondere folgende Netze zu unterstützen:

1. Schnelle LANs wie z.B. Gigabit-Ethernet.
2. DSL-Anbindungen (*Digital Subscriber Line*) von Heimarbeitsplätzen ans Internet mittels PPPoE (*PPP over Ethernet*) [RFC 2516, Abschnitt 11.2] und Datenraten bis zu 8 Mbit/s, aber u.U. mit merklichen Verzögerungen bei der Datenübertragung bedingt durch Packet-Interleaving.
3. Ausgedehnte, große Netzstrukturen (WAN) mit unterschiedlichen Übermittlungsnetzen wie ATM oder Frame-Relay und Satellitenverbindungen mit z.T. signifikanten Verzögerungszeiten bei der Übertragung auf *Long Fat Networks* (LFNs) [RFC 2488].
4. Zu übertragende Datenvolumen, die im Bereich von Gigabyte liegen (z.B. beim Download von Videodateien) und den Bereich der einfach-adressierbaren Sequenznummern bei TCP überschreiten.

4.4.1 Klassische TCP-Implementierungen

Für TCP existieren eine Reihe von Implementierungsvorschlägen – konkret die Realisierung des TCP/IP-Stack –, die bekannte Schwächen beheben und zudem die Effizienz steigern sollen [RFC 1122]:

- *Nagle-Algorithmus*
  Er nimmt Bezug auf das Problem, dass die TCP-Instanz auf Anforderung der Anwendungsschicht sehr kleine Segmente sendet. Zur Reduzierung der Netzlast und damit zur Verbesse-

Nagle-Algorithmus

rung des Durchsatzes sollten die pro Verbindung von der Anwendungsschicht ankommenden Daten möglichst konkateniert, d.h. in einem Segment zusammen gesendet werden. Dies hat nicht nur zur Folge, dass der Protokoll-Overhead verringert wird, sondern auch, dass die TCP-Instanz beim Empfänger nicht jedes (kleine) Segment per ACK bestätigen muss, was wiederum Auswirkungen auf die Gesamtlaufzeit hat. In den Nagle-Algorithmus gehen vier Faktoren ein:

- die *TCP-Haltezeit* für das Zusammenführen von Applikationsdaten in Segmente,
- der verfügbare *TCP-Pufferbereich* für Applikationsdaten,
- die *Verzögerungszeiten* im Übermittlungsnetz (z.B. LAN oder WAN),
- der *Applikationstyp*: In diesem Zusammenhang sind die 'interaktiven' Protokolle wie TELNET, RLOGIN, HTTP und speziell auch X-Windows besonders kritisch, da hier z.T. jedes einzelne Zeichen (Tasteneingabe bzw. Mausklick) für die Bildschirmanzeige 'geechot' wird.

Silly Window Syndrom

- *Silly Window Syndrome*

Mit diesem Begriff wird der Zustand gekennzeichnet, wenn ein TCP-Empfänger sukzessive mit der Erhöhung des zunächst kleinen internen TCP-Puffers beginnt und dies der sendenden TCP-Instanz durch ein weiteres `<ACK>`-Paket mit der neuen `Windowsize` umgehend mitteilt. Hierdurch kann es vorkommen, dass sich bei der Übertragung großer Datenmengen Sender und Empfänger hinsichtlich von *Windowsize* nicht mehr vernünftig abstimmen und der Sender nur noch sehr kleine Datensegmente übermittelt. Dieser Fehler im Fenstermanagement ist dadurch zu vermeiden, dass der Empfänger mit der Sendung des *ACK-Pakets* wartet, bis er hinlänglich TCP-Puffer allokieren kann. Es ist zu beachten, das dies ein zum *Nagle-Algorithmus* komplementärer Effekt ist.

Zero Window Probe

- *Zero Window Probe*

Ist eine TCP-Verbindung aufgebaut [Abb. 4.3-4], kann eine TCP-Instanz der anderen durch Setzen von *Windowsize* auf `0` mitteilen, dass sie ihren TCP-Empfangspuffer auf Null reduziert hat. Dies kann z.B. eine Folge davon sein, das die TCP-Instanz die bereits anstehenden Daten nicht mehr an die Applikation weiterreichen kann, was typischerweise der Fall ist, wenn sich in einem Netzwerkdrucker kein Papier mehr befindet. In Anschluss daran ist es nach Ablauf von *Timeout* Aufgabe des Senders, mit einer *Zero Window Probe* festzustellen, ob der Empfänger wieder aufnahmebereit ist. Hintergrund hierfür ist, dass `<ACK>`-Pakete ohne Daten nicht verlässlich übertragen werden. Sollte der Empfänger hierauf nicht antworten, wird der *Retransmission-Algorithmus* in Gang gesetzt.

TCP Keep-Alives

- *TCP Keep-Alives*

TCP verzichtet in der Regel auf ein *Keep-Alive-Verfahren*, ohne ein solches jedoch ausdrücklich auszuschließen. TCP-Keep-Alive-Informationen werden in *<ACK>-Paketen* mit einem bedeutungsfreien Datenbyte oder völlig ohne Daten eingeschlossen. Sie dürfen aber nur versendet werden, wenn keine anderen regulären Daten zwischen den TCP-Instanzen ausgetauscht werden. Entscheidend ist, das die generierte Sequenznummer [Abb. 4.4-2] dem obersten Wert des Sendefensters abzüglich eines Byte entspricht. Dieser Wert liegt außerhalb des ausgehandelten Sendefensters, was die TCP-Partnerinstanz veranlasst, mit einem `<ACK>`-PDU zu antworten.

4.4.2 Abschätzung der Round Trip Time

Eine konzeptionelle Eigenheit von TCP besteht darin, auch ohne positives ACK nach einer bestimmten Zeit (*Timeout*) die Daten erneut zu versenden. Hierzu bedient sich TCP einer Abschätzung der *Round Trip Time* (RTT), die möglichst präzise erfolgen sollte. Abb. 4.4-1 illustriert die Möglichkeiten der RTT-Abschätzung.

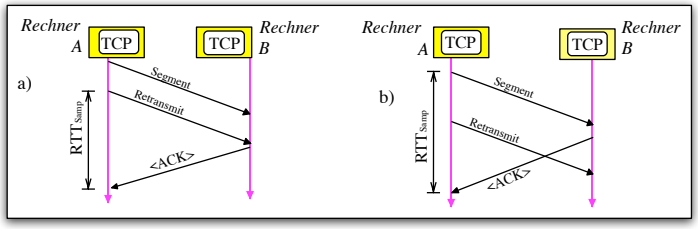


Abb. 4.4-1: Abschätzung von RTT nach: a) original TCP-Implementierung, b) Karn/Partridge-Algorithmus

Drei unterschiedliche Methoden für die RTT-Abschätzung sind sowohl in Endstationen als auch in Routern gebräuchlich:

- Die *Originalimplementierung* [Abb. 4.4-1a] sieht vor, RTT für jedes einzelne TCP-Paket zu ermitteln und hieraus ein gewichtetes Mittel rekursiv zu berechnen:

  $$RTT_{Est} = a * RTT_{Est} + (1 - a) * RTT_{Samp}$$

  Hierbei stellt RTT_{Samp} den für ein <ACK>-Paket gemessenen Wert von RTT dar. Da am Anfang RTT_{Est} nicht bestimmt ist, wird $RTT_{Est} = 2s$ angenommen. Hieraus ergibt sich auch eine Abschätzung für Timeout:

  $$\text{Timeout} = b * RTT_{Est}$$

  Für die Parameter wird üblicherweise $a = 0,9$ und $b = 2$ angenommen. Problematisch an diesem Ansatz ist, dass sich einerseits ein 'verloren gegangenes' Paket in einer Unterschätzung von RTT auswirkt und damit andererseits keine Korrelation mit ACKs des Empfängers gegeben ist.

- Die *Karn/Partridge-Implementierung* [Abb. 4.4-1b] umgeht die letzte Einschränkung, da erst das ACK-Paket den Datenempfang bestätigt, wobei wiederholte TCP-Pakete nicht in den Algorithmus einbezogen werden. Zudem wird die Dauer von *Timeout* nach jedem Empfang angehoben:

  Karn/Partridge-Implementierung

  $$\text{Timeout} = 2 * \text{Timeout}$$

- Die *Jacobsen/Karel-Implementierung* verfeinert den Karn/Partridge-Algorithmus durch Einbeziehen der Varianz in RTT_{Samp}, d.h. der Schwankungen dieser Werte:

  Jacobsen/Karel-Implementierung

  $$\delta(RTT) = RTT_{Samp} - RTT_{Est} \quad \text{und} \quad RTT_{Est} = RTT_{Est} + g_0 * \delta(RTT)$$

Hierbei beträgt der Wert $g_0 = 0,125$. Auch wird die Berechnung des Timeout angepasst. Zunächst wird eine Hilfsgröße definiert:

$$\text{Abweichung} = \text{Abweichung} + g_1 * \delta(RTT)$$

mit $g_1 = 0,25$. Somit wird das neue Timeout berechnet nach:

$$\text{Timeout} = p * RTT_{Est} + q * \text{Abweichung}$$

wobei für p und q die Erfahrungswerte $p = 1$ und $q = 4$ gewählt sind.

TSopt und RTT

Statt dieser heuristischen Verfahren kann die RTT auch über die TCP-Option RTT (*Round Trip Timer*) bestimmt werden. Abb. 4.4-2 illustriert dies.

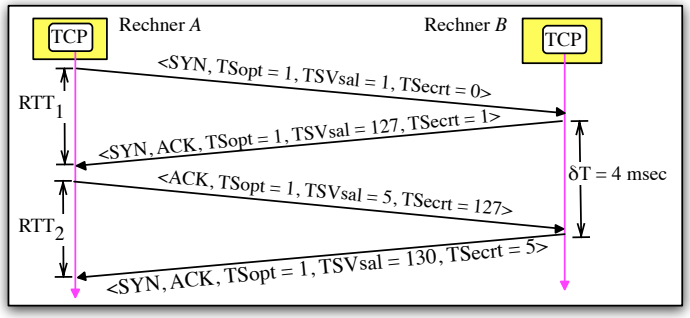


Abb. 4.4-2: Bestimmung der Round Trip Time mittels der Option TSopt, bei der beide Kommunikationspartner die jeweiligen Werte für TSVal und TSecrt mitteilen

Der Client setzt zunächst im ersten <SYN>-Paket die Option TSopt und befüllt das 4 Byte große TCP-Feld TSVal mit dem eigenen Zeitstempel (*Tickmark*). Sofern die angesprochene TCP-Instanz TSopt unterstützt, trägt diese wiederum ihre Tickmark im Timestamp des TSecr-Felds des <ACK>-Pakets ein, wobei immer auf das vorausgegangene <SYN>-Paket des Partners Bezug genommen wird.

Aus den gemittelten Werten lässt sich so eine relativ präzise Abschätzung der effektiven Round Trip Time ableiten, sofern statistische Ausreißer eliminiert werden [RFC 1323]. Es ist zu beachten, das die TimeStamp-Option nur einmal – und zwar genau zu Beginn der Verbindung – ausgehandelt werden darf und dann bis zum Abschluß der TCP-Konversation benutzt werden muss. Ein Wechsel in den 'TSopt-Betrieb' während der laufenden TCP-Sitzung ist eigentlich verboten, wird jedoch von z.B. Windows-Betriebssystemen stillschweigend unterstützt. Wollen beide Kommunikationspartner TSopt benutzen, ist dies jeweils unabhängig anzuzeigen.

4.4.3 Verbesserung der Effizienz von TCP

Das Erkennen und das Beheben möglicher Durchsatzprobleme beinhalten mehrere Aspekte, die den Aufbau der Verbindung, ihre Unterhaltung und die Reaktion auf

Fehler betreffen. Um die Effizienz von TCP zu verbessern, wurden folgende Verfahren vorgeschlagen:

- *TCP Slow Start* sagt aus [RFC 1323], dass eine TCP-Instanz die Übermittlung von TCP-Paketen mit einem kleinen *Congestion Window* (`cwnd`) beginnt, i.d.R. `cwnd = 1` Segment (Defaultwert: 536 Byte). Anschließend wird cwnd mit jedem empfangenen `<ACK>`-Paket quadratisch vergrößert, d.h. `cwnd` = `cwnd` $*$ `cwnd` und die Anzahl der übertragenen TCP-Pakete entsprechend erhöht, bis der Empfänger bzw. ein zwischengeschalteter Router Paketverluste signalisiert. Dies teilt dem Sender mit, dass er die Kapazität des Netzwerks bzw. des Empfängers überschritten hat, was eine Reduktion von `cwnd` zur Folge hat. TCP Slow Start
- *Congestion Avoidance* [RFC 2581, 2001 sowie 5681] geht von der Annahme aus, dass Datenpakete in Netzen kaum mehr verloren gehen, sondern im Falle von *Timeouts* und doppelt empfangenen `<ACK>`-Paketen (`dACKs`) eine Überlast im Netzwerk aufgetreten ist. Eine TCP-Instanz kann dem vorbeugen, indem sie neben `cwnd` und der Grösse des *advertised Windows* (`advWin`) eine Variable *Slow Start Threshold* (`ssthresh`) nach folgendem Schema nutzt: Congestion Avoidance

  1. Initialisierung: `cwnd` = 1 Segment, `ssthres` = max. Windowsize (64 KByte)
  2. Maximale zu sendende Datenmenge: $min(\texttt{cwnd}, \texttt{advWin})$
  3. Beim Empfang von `dACKs` wird `ssthresh` neu berechnet:

     $$\texttt{ssthresh} = max(2, min(\texttt{cwnd}/2, \texttt{advWin}))$$

     Falls Timeouts registriert werden, gilt: `cwnd` = 1
  4. Beim Empfang eines neuen `ACK-Pakets` wird `cwnd` nach *Slow Start* oder nach *Congestion Avoidance* wieder erhöht.
- *Fast Retransmit* [RFC 2581/2001] nimmt an, dass mehrere empfangene (double ACK) `dACKs` den Verlust lediglich eines TCP-Pakets bedeuten, welches neu gesendet werden muss. Falls die Anzahl von `dACKs` einen Schwellenwert überschreitet (z.B. 3 `dACKs`), tritt ein *Fast Retransmit* in Kraft, indem nicht der TCP-Timeout abgewartet wird, sondern sofort das letzte Segment zu wiederholen ist. Im Anschluss geht die TCP-Instanz in ein *Fast Recovery* über, was auf der Annahme basiert, dass ein (noch) anhaltender Datenfluss vorliegt. Es wird die *Congestion Avoidance* statt des *Slow Start* eingesetzt. Fast Retransmit
- *Selective Acknowledgement* (*SACK*): Mit dem in RFC 2018 vorgestellten SACK-Verfahren wird dem Problem begegnet, dass mit hoher Wahrscheinlichkeit beim Registrieren von kumulativen `dACKs` nur ein Paket neu übertragen werden muss. *Fast Retransmit* würde hingegen mehrere u.U. fehlerfrei – aber mit Verzögerung – empfangene TCP-Pakete wiederholen. Um dieser Situation zu entgehen, muss der Empfänger dem Sender in einem `<ACK>`-Paket den Beginn und das Ende derjenigen Datenblöcke mitteilen, die er als Letzte zusammenhängend in seinem Empfangsfenster (Datenpuffer) verarbeitet hat. Den Aufbau der *SACK*-Option zeigt Abb. 4.4-3. Selective Acknowledgement

  Die *SACK*-Option erweitert das Optionsfeld um bis zu maximal 40 Byte. Wie aus Abb. 4.4-3 ersichtlich ist, können höchstens 4 Blöcke gebildet werden, in denen

Informationen über vier unterschiedliche Pufferbereiche einfließen. Wird zusätzlich die Option *Timestamp* eingesetzt, sind lediglich drei Blöcke möglich.

PAWS

- *Protection Against Wrapped Sequence Number*: Ein Problem, das bei der Übertragung großer Datenmengen auftritt, ist der Überlauf des Sequenzzählers `SEQ` (*Sequence Number*) für TCP-Segmente. Wie in Abb. 4.3-2 dargestellt, ist `SEQ` ein 32-Bit-Wert, durch den normalerweise ein TCP-Segment – und damit die maximale Größe einer zu übertragenden Datei – auf maximal 4 GByte beschränkt ist. Ist die Datenmenge größer, muss der Zähler neu von 1 initialisiert werden. Wie soll die TCP-Instanz entscheiden, ob ein eventuell verlorengegangenes und zu wiederholendes TCP-Segment aus der aktuellen 'Runde' oder aus einer früheren stammt? Die Lösung hierfür ist die in Abschnitt 4.3 vorgestellte *Timestamp*-Option. Zusätzlich zu `SEQ` enthält das TCP-Segment eine monoton steigende, 4 Byte große Zeitinformation vom Kommunikationspartner, dessen jeweils aktueller Wert zu speichern ist. Ist das Zeitintervall eines 'Uhrticks' nun 1 ms, reicht dies aus, die Datenübertragung über nahezu 25 Tage zu monitoren. Durch den Vergleich des *Timestamp* eines alten und eventuell wiederholten TCP-Pakets mit der aktuell entgegengenommenen Zeitmarke kann das Erstere getrost verworfen werden.

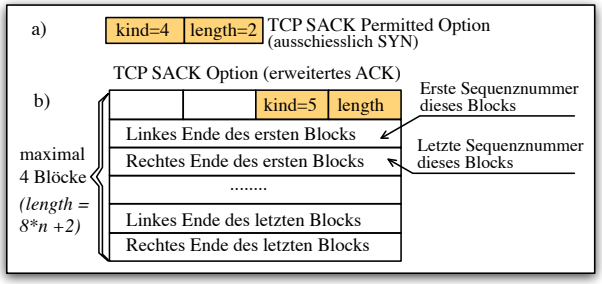


Abb. 4.4-3: Aufbau des SACK-Optionsfelds: a) für `<SYN>`-Pakete beim Aushandeln der Option b) bei der Übermittlung fehlerhafter oder fehlender Daten in `<ACK>`-Paketen

Hervorzuheben sind weitere interessante (z.T. experimentelle) Implementierungsvorschläge zur Flusskontrolle wie z.B.:

dACKs

- `dACKs`, d.h. die Erkennung von 'Duplicate ACKs' [RFC 2883], sowie die Ergänzung in Form des
- *Eifel-Algorithmus* für TCP [RFCs 3522 und 4015] für 'verlorene' ACKs zusammen mit dem
- 'New Reno' Fast *Recovery Algorithmus* [RFC 3782] sowie der Einsatz von
- `ECN` (*Explicit Congestion Notification*) [RFCs 3168/2481 und 2884] für TCP.

4.4.4 Datendurchsatz beim TCP

Bei der UDP-Übertragung ergibt sich der theoretisch maximale (Nutz-)Datendurchsatz einfach unter Berücksichtigung des Protokoll-Overheads und den Abständen, die die Frames auf der Datensicherungsschicht einhalten müssen,

wobei als Faustformel davon ausgegangen werden kann, dass der Protokoll-Overhead etwa 3% ausmacht. Wir bezeichnen den maximal möglichen Durchsatz auf der Schicht 2 auch als *Wirespeed* bzw. *Leitungsdurchsatz*. Hierbei ist zu beachten, dass die 'Leitungsgeschwindigkeit' in der Regel als `Bit/s` angegeben, der 'Datendurchsatz' aber in `Byte/s`.

Beim verbindungsorientierten TCP-Protokoll hängt der Durchsatz von der Bestätigung durch den Empfänger ab. Somit müssen nicht nur die Nutzdaten beim Empfänger angekommen sein, sondern auch die Quittung muss beim Sender angekommen sein. Daher bestimmt die *Round Trip Time* der Nachrichten den Durchsatz bei TCP.

Bandbreiten-Delay-Produkt

Maßgeblich für den TCP-Datendurchsatz ist das 'Bandbreiten-Delay'-Produkt. Der maximal erzielbare Durchsatz ergibt sich durch:

```
Durchsatz [KByte/s] = TCP Windowsize [KByte] / RTT [s]
```

Signallaufzeiten

Damit hängt der Datendurchsatz nicht nur von der *Leitungsschwindigkeit* also dem 'Wirespeed' ab, sondern insbesondere auf längeren Strecken merklich von der Laufzeiten der Signale auf der Übertragungsstrecke. Auf Kupfer- und Glasfaserkabel können wir hier von einer physikalischen Signalgeschwindigkeit von rund 200 000 km/s ausgehen; also etwa 2/3 der Lichtgeschwindigkeit. Hinzu kommen noch Verzögerungen in den aktiven Komponenten wie Router und Switches. Deren Einfluss ist unter normalen Umständen zu vernachlässigen, sofern keine Paketverluste, *Congestion* bzw. ein gewolltes *Traffic-Shaping* hinzukommen.

Performance-relevant sind folgende Faktoren:

- Der 'Wirespeed' der Datenverbindung, die den maximalen Datendurchsatz festgelegt.
- Die *Round Trip Time* der Datenpakete, wobei hier die Signallaufzeiten zwischen den kommunizierenden Rechnern, aber auch statistische Ausreißer durch Paketverluste beitragen.
- Die *Windowsize* des Empfängers, die festlegt, wie viele Byte vom Sender verschickt werden können, ohne auf ein `<ACK>`-Paket des Empfängers zu warten. Entsprechend RFC 3390 wird die *initiale Windowsize* gemäß folgender Relation

  ```
  Windowsize = min (4*MSS, max (2*MSS, 4380 bytes)),
  ```

  gewählt, wobei `MSS` die *Maximum Segment Size* darstellt. Diese entspricht in der Regel dem Wert der MTU, bei Ethernet also typischerweise 1500 Byte.
- Die Größe der zu übertragenden Nachricht.

Datendurchsatz beim Three-Way Handshake

Aus Abb. 4.4-4a ist zu entnehmen, dass bereits der notwendige *Three-Way Handshake* zu einem merklichen Effekt beiträgt, da zunächst die Zeit von `1,5 * RTT` abgewartet werden muss, bevor Nutzdaten gesendet werden können.
Bei dem in Abb. 4.4-4b illustrierten Beispiel wird angenommen, dass der Empfänger *B* eine *Windowsize* von 1500 Byte besitzt. Die zu übertragende Datenmenge beträgt 2500 Byte. Hierbei sendet Rechner *A* drei TCP-Segmente mit jeweils 500 Byte. Diese kann er in einem Schwung absetzen, da er auf keine `<ACK>`-Pakete warten muss. Der Durchsatz beträgt damit näherungsweise `Windowsize/RTT`. Nach Erhalt des `<ACK>`-Pakets vom Rechner *B* kann die restliche Datenmenge übertragen werden. Obwohl jetzt

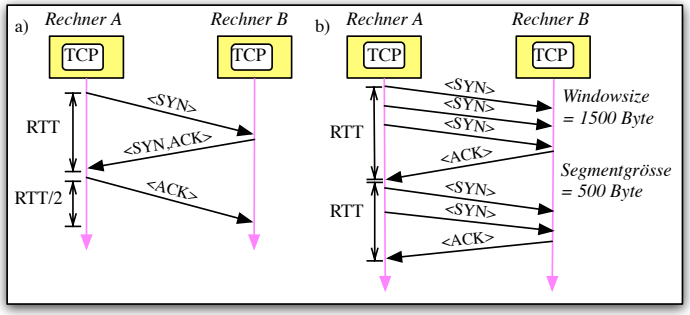


Abb. 4.4-4: Einflussgrößen für den TCP-Datendurchsatz a) der 3WHS, b) die Round Trip Time zusammen mit der Windowsize

eine geringe Anzahl Nutzbyte gesendet wurde, muss dennoch praktisch die gleiche RTT abgewartet werden.

Die zentrale Stellschraube zur Verbesserung des Datendurchsatzes bei TCP ist die *Windowsize*. Während das Standard-TCP-Fenster 64 KByte beträgt, kann mittels der *Window Scale* Option dieses Fenster bis zu 1 GByte vergrößert werden. Aktuelle TCP-Implementierungen nehmen ein *Autotuning* der *Windowsize* vor. Hierbei wird zunächst der Wert der RTT ermittelt und die *Windowsize* dem Bedarf der zu sendenden Datei angepasst. Ist die Größe der zu übertragenden Datei z.B. 100 MByte, wird auch nur ein Fenster von maximal dieser Größe angefordert. Daraus ergibt sich (vgl. Abb. 4.4-4b), dass der erzielbare Datendurchsatz für kleine Dateien immer schlechter ist als für große, und zwar unabhängig von der 'Leitungsgeschwindigkeit'.

TCP Fast Open

Aus Abb. 4.4-4 geht hervor, dass das Öffnen einer TCP-Verbindung durch den 3WHS viel Zeit kostet. Dies trifft besonders auf Anwendungen zu, die viele TCP-Verbindungen (auf dem gleichen Server) öffnen. Typische Beispiele sind:

- HTTP-Zugriffe auf Webseiten mit vielen (lokal) eingebetteten Bildern, für deren Download jeweils eine TCP-Session benötigt wird.
- Zugriffe auf Dateien, z.B. auf einen Samba-Server mittels des Protokolls CIFS (*Common Internet File System*), wo pro Verzeichnis und Datei ebenfalls eine TCP-Sitzung verlangt wird.
- Mail-Server per IMAP4 [RFC 3501], bei dem jeder Zugriff auf einen abonnierten Ordner ebenfalls mit einer TCP-Sitzung einhergeht.

Unter der Annahme, dass die TCP-Verbindung mit dem gleichen Server stattfindet, kann der 3WHS durch das *TCP Fast Open* (TCP/FO) Verfahren [https://tools.ietf.org/html/draft-ietf-tcpm-fastopen-05] deutlich beschleunigt werden, was aber sowohl vom Client als auch vom Server unterstützt werden muss [Abb. 4.4-5]:

1. Der Client generiert bei der ersten Kommunikation zum Server eine spezielle <SYN>-Sequenz, die im TCP Option-Feld einen Cookie-Request beinhaltet (*Fast Open Cookie Request Option*).
2. Der Server generiert hierauf ein Cookie und sendet dies im Folgenden <SYN,ACK>-Paket im Option-Feld an den Client, der dieses pro TCP-Verbindung cached.
3. Wird die TCP-Verbindung zum selben Server erneut angefordert, findet ein verkürzter Handshake statt, indem bereits im ersten <SYN>-Paket neben dem Cookie Nutzdaten untergebracht werden können, was somit eine RTT-Zeit spart.

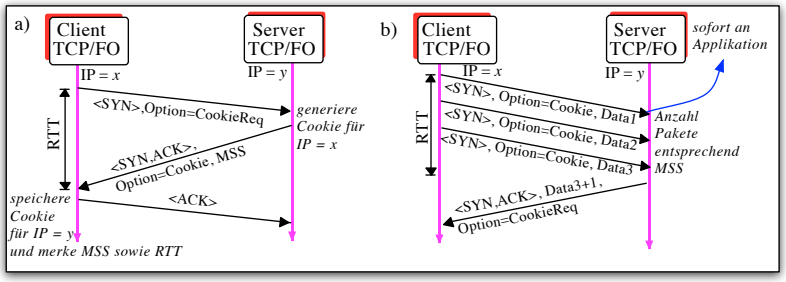


Abb. 4.4-5: Ablauf des TCP/FO-Verfahrens: a) Anforderung des FastOpen-Cookies im 3WHS b) TCP FastOpen falls gültiges Cookie vorhanden

Das Cookie *authentisiert* hierbei den Client und ist im Grunde genommen statisch. Server und Client müssen sich das Cookie merken, d.h. im Cache behalten. Allerdings ist die Gültigkeitsdauer des Cookies beschränkt.

Man kann also sagen, dass das übliche TCP-Protokoll mit seiner zustandslosen Verbindungsaufnahme aber zustandsbehaftetem Sitzungsgestaltung nun durch eine zustandsbehaftete Verbindungsaufnahme bei TCP/FO und anschließend durch einen quasi zustandslosen Sitzungsablauf ersetzt wird. Dies erfordert zunächst neue Parameter für den Socket-Aufruf (siehe nächster Abschnitt) und verlangt, dass die Applikation die Daten bereits vom ersten <SYN>-Paket entgegen nehmen kann. Ein weiteres Problem stellen herkömmliche Firewalls dar, die auf dieses TCP-Verhalten nicht vorbereitet sind und die Verbindungsaufnahme unterbinden würden. Sollte die TCP/FO-fähige Applikation dies feststellen (nach *Timeout*), muss sie in den normalen TCP-Modus zurück fallen.

4.4.5 TCP Socket-Interface

Socket-Schnittstellen

Die Entwicklung von TCP/IP vollzieht sich in engem Zusammenhang mit der Programmiersprache C und ihrem Einsatz insbesondere unter dem Betriebssystem Unix. Die Implementierung von TCP liegt in Form des TCP-Stacks vor, der sowohl eine Schnittstelle für die Applikation als auch für das darunterliegende IP-Interface bereitstellen muss. Software-technisch wird dies über das Socket-Interface realisiert, was in der Windows-Welt als WinSock-Schnittstelle, in der Unix-Welt als BSD- bzw. Unix-System-V Socket-Schnittstelle gewährleistet wird.

Während für IP und UDP die Realisierung der entsprechenden Schnittstelle relativ einfach gehalten werden kann, verlangt TCP – aufgrund seiner Zustandsmaschine – große Sorgfalt bei der Gestaltung. Zudem verfügt TCP über eine Reihe von optionalen Schaltern, die bei Unix mittels Kernelparameter und bei den Windows-Betriebssystemen über *Registry-Einträge* gesetzt werden können.

TCP-API

Für die aufrufende Anwendung verfügt TCP über eine Programmschnittstelle [RFC 793] für den Auf- und Abbau von Verbindungen sowie die Steuerung der Datenübermittlung. Diese Programmschnittstelle wird TCP-API (*Application Program Interface*) genannt.

Sie beinhaltet folgende Funktionen:

Open

- *Open*: Öffnen von Verbindungen mit den Parametern:
  - Aktives/Passives Öffnen,
  - Entfernter Socket, d.h. Portnummer und IP-Adresse des Kommunikationspartners,
  - Lokaler Port,
  - Wert des Timeouts (optional). Als Rückgabewert an die Applikation dient ein lokaler Verbindungsname, mit dem diese Verbindung referiert werden kann.

Send

- *Send*: Übertragung der Benutzerdaten an den TCP-Sendepuffer und anschließendes Versenden über die TCP-Verbindung. Optional kann das *URG-Bit* bzw. *PSH-Bit* gesetzt werden.

Receive

- *Receive*: Daten aus dem TCP-Empfangspuffer werden an die Applikation weitergegeben.

Close

- *Close*: Beendet die Verbindung, nachdem zuvor alle ausstehenden Daten aus dem TCP-Empfangspuffer zur Applikation übertragen und ein TCP-Paket mit dem FIN-Bit versandt wurde.

State

- *State*: Gibt Statusinformationen über die Verbindung aus wie z.B. lokaler und entfernter Socket, Größe des Sende- und des Empfangsfensters, Zustand der Verbindung und evtl. lokaler Verbindungsname. Diese Informationen können z.B. mittels des Programms *netstat* ausgegeben werden.

Abort

- *Abort*: Sofortiges Unterbrechen des Sende- und Empfangsprozesses und Übermittlung des `RST`-Bit an die Partner-TCP-Instanz.

Das Zeitverhalten der TCP-Instanz wird im Wesentlichen durch die Implementierung und natürlich durch die Reaktionszeiten des Peer-Partners und durch das Netzwerk bestimmt. Ausnahmen entstehen durch den Einsatz folgender Socket-Optionen:

- `TCP_NODELAY`: gibt vor, das die in der TCP-Instanz anstehenden Daten unverzüglich gesendet werden. Hiermit wird der Nagle-Algorithmus außer Kraft gesetzt.
- *LINGER*: Die Socket-Option *LINGER* (Verweilen) ermöglicht der Applikation zu kontrollieren, ob die ausgesandten Daten bei einem TCP-Close auch angekommen sind. Beim Aufruf wird angegeben, ob diese Option benutzt wird und welche Zeitdauer das 'Linker' (d.h. de facto die Blockierung des Sockets) bis zum Empfang der Bestätigung maximal überschreiten soll. Die Implementierung dieser Option ist nicht eindeutig.

- *KEEPALIVE*: Mittels dieser für die TCP-Instanz allgemeingültigen Einstellung kann festgelegt werden, ob überhaupt TCP-Keep-Alives verwendet werden sollen.

TCB

Sowohl für aktiv geöffnete Ports wie auch für Ports, die ein `<SYN>`-Paket zum Verbindungsaufbau empfangen haben, wird in der Regel ein *TCP Control Block* (TCB) geöffnet, über den die Daten weitergegeben werden und der für den Ablauf der Kommunikation verantwortlich zeichnet.

4.4.6 Angriffe gegen den TCP-Stack

Wie in Abb. 4.1-1 bereits gezeigt, wird eine TCP-Verbindung durch den Socket im Rechner *A* – z.B. als Client – und den Socket im Rechner *B* – z.B. als Server – festgelegt. Da die Datenübertragung unverschlüsselt erfolgt, kann die Kommunikation jederzeit mit geeigneten technischen Möglichkeiten überwacht bzw. auch verfälscht werden, indem 'irreguläre' Datenpakete eingeschleust werden.

Spoofing

Da die IP-Adresse des Empfängers (Servers) und die Portnummern der Standardapplikation in der Regel über *Portscans* sehr leicht ermittelt werden können (oder einfach auch nur ausprobiert werden), kann ein Serversystem sehr leicht mit unvorhergesehenen Paketen kompromittiert werden (*TCP-Spoofing*). Im ungünstigsten Falle kann dies zu einer Ressourcenblockade bzw. zu einer *Denial of Service-Attacke* (kurz *DoS-Attacke*) führen, durch die u.U. die gesamte TCP-Instanz oder der Rechner selbst blockiert wird.

Zustandstabellen

Daher sollte man vorsichtig vorgehen und nur solche TCP-Pakete akzeptieren, die entsprechende Kriterien erfüllen. TCP-Pakete mit Nutzdaten sind z.B. nur dann sinnvoll, wenn zuvor eine TCP-Verbindung aufgebaut wurde. Zur Kontrolle von TCP-Paketen kann eine Zustandstabelle gepflegt werden, die den Status der einzelnen Verbindungen protokolliert. Das Kontrollieren von TCP-Paketen aufgrund von Zustandsinformationen wird *Stateful Inspection* genannt und ist ein Verfahren, das typischerweise in *Firewalls* eingesetzt wird.

Firewalls

Firewalls überprüfen in der Regel den Inhalt der Daten (z.B. in Form von 'malicious Code' oder Viren) nicht, sondern können bestenfalls die Validität und die Authentizität einer Verbindung feststellen. Die Authentizität wird mittels *TCP-Wrapper* ermittelt, die einen Abgleich zwischen der IP-Adresse des Clients und des Eintrags im DNS (und vice versa) durchführen – in Zeiten sog. NAT'ed Clients ein kaum mehr wirkungsvolles Verfahren. Typischerweise ist es Aufgabe der Applikation, die Authentizität der Verbindung sicherzustellen, teilweise auch mittels vorgeschalteter Programme wie unter Unix der *tcpserver* [`https://cr.yp.to/ucspi-tcp.html`].

Stateful Inspection

Die Validität eines TCP-Pakets ergibt sich aus dem Zustandsdiagramm [Abb. 4.3-3]. Zusätzlich zur Registrierung der jeweiligen Kommunikations-Tupel können als sekundäre Kriterien `ISN` und ggf. `RTT` und bei T/TCP auch `CC` herangezogen werden. Tertiäre Kriterien ergeben sich aufgrund der in TCP-Paketen möglichen Optionen und ihrer Kopplung an die übertragenen Daten.

TCP-Backlog

TCP-Instanzen führen Buch über die Anzahl der Verbindungsversuche, die aktuellen Verbindungen sowie die in Abbau befindlichen Verbindungen. Für die aktuell bestehenden Verbindungen ist bereits ein TCB-Puffer (*TCP Control Block*) reserviert,

für die abzubauenden Verbindungen wird er wieder freigegeben; aber für alle neuen Verbindungen muss er erneut allokiert werden. Da die Zeitspanne zwischen dem (empfangenen) ersten `<SYN>`-Paket (d.h. dem Verbindungsaufbauwunsch) und dem gesendeten `<SYN,ACK>`-Paket sowie der ersten Datenübertragung durchaus beträchtlich sein kann ($< 2^n * MSL$), wird dies bei der TCP-Instanz in einem *Backlog-Puffer* protokolliert: Die TCP-Instanz befindet sich im Zustand `SYN-Received`. In diesem Backlog-Puffer werden die *halb offenen Verbindungen* festgehalten. Erfolgt nach dem Absenden des `<SYN,ACK>`-Pakets des Servers keine Reaktion des Clients, wird der Verbindungswunsch nicht abgelehnt, sondern die TCP-Instanz wiederholt die Aussendung dieses Pakets in der Regel nach einem Algorithmus, der quadratisch in der Zeit aufgesetzt ist [Abb. 4.4-6].

SYN-Flooding

Durch zu viele offene TCP-Verbindungen kann dieser Puffer überlaufen und so die Arbeitsfähigkeit der TCP-Instanz insgesamt beeinträchtigen. In diesem Zusammenhang spricht man von *SYN-Flooding*. Leider sind besonders Firewalls und LoadBalancer für SYN-Flooding anfällig, da typischerweise Verbindungswünsche für `N` Endsysteme (z.B. `HTTP-Requests`) über lediglich einen oder zwei Firewall-Rechner bzw. den *LoadBalancer* (bzw. deren öffentliche IP-Adressen) abgewickelt werden.

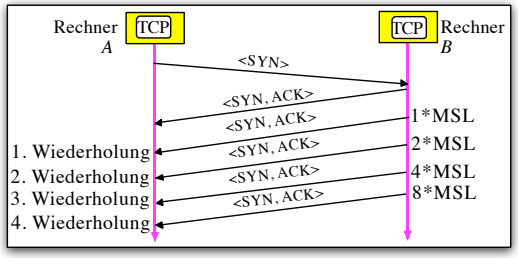


Abb. 4.4-6: `SYN`-Attacke mit anschließender Wiederholung der `<SYN,ACK>`-Pakete

SYN-Cookies

Eine Möglichkeit, die Anzahl der offenen TCP-Verbindungen zu kontrollieren, besteht darin, ab einem gewissen Schwellenwert *SYN-Cookies* einzusetzen [`https://cr.yp.to/syncookies.html`, 4987]. Hierbei lehnt die TCP-Instanz bei Überschreitung des Schwellenwerts neue Verbindungen nicht ab, sondern antwortet zunächst mit der Aussendung eines SYN-Cookies in einem normalen `<SYN,ACK>`-Paket. Im Gegensatz zu einer normalen Antwort wird die `ISN` nicht zufällig, sondern aus den Werten IP-Adresse, Portnummer, einem Zeitstempel sowie der `MSS` (kryptografisch) als Hashwert gebildet. Zusätzlich wird der TCB erst dann aufgebaut, wenn von der Gegenseite ein `<ACK>`-Paket empfangen wird, dessen Acknowledge-Nummer identisch ist mit dem Wert von `ISN + 1`. Der Charme dieser Lösung besteht darin, dass der Server sich keine `ISN` pro Verbindung merken muss; vielmehr kann die Validität der Verbindung algorithmisch aus dem ersten empfangenen `<ACK>`-Paket berechnet werden.

RST-Angriffe

Gelingt es einem Angreifer, den TCP-Datenverkehr zu belauschen oder einige Steuerungsangaben zu erraten, kann er in bestimmten Situationen mit einer 'brute force'-Attacke die TCP-Verbindung zum Abbruch zwingen, wenn er den Socket – also das Tupel (IP-Adresse, Portnummer) – vom Client sowie die aktuelle Sequenznummer

kennt bzw. errät. Mittels dieser Informationen kann ein <RST>-Paket zum Server verschickt werden, der daraufhin die Verbindung ebenfalls abbricht [Abb. 4.3-5].

4.4.7 Socket Cloning und TCP-Handoff

Weiterschalten von TCP-Verbindungen

Eine TCP-Verbindung kann mit einer 'Telefonverbindung' zwischen Applikationen verglichen werden. In Telefonnetzen wird *Weiterschalten von Verbindungen* als selbstverständlich angenommen. Auch in IP-Netzen ist es manchmal nötig, eine TCP-Verbindung weiter zu schalten. In diesem Zusammenhang spricht man von *TCP-Handoff* und von *Socket Cloning*. Abb. 4.4-7 illustriert dies. Hier greift z.B. der Rechner eines Benutzers aus Deutschland auf den Webserver S_0 in den USA zu, der als Eingang zu einem E-Commerce-System mit Streaming-Media dient. Zwischen diesem Rechner und dem Server S_0 wird eine TCP-Verbindung aufgebaut. Der Server S_0 leitet aber diese Webanfrage (d.h. *HTTP-Request*) zu dem von ihm ausgewählten Server S_k z.B. in Frankfurt weiter, um das Streaming von dort auszuliefern. Dieses Prinzip liegt den *Content Delivery Networks* zugrunde [BRS03].

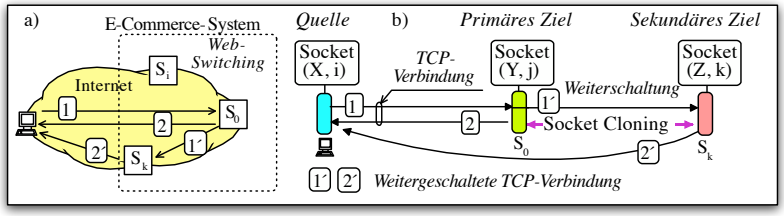


Abb. 4.4-7: Handoff als Weiterschaltung einer TCP-Verbindung
Appli.: Applikation; X, Y, Z: IP-Adressen; i, j, k: Portnummern

In diesem Fall wird die TCP-Verbindung zwischen dem Rechner des Benutzers und dem Server S_0 (primäres Ziel) zum sekundären Ziel weiter geschaltet. Das Weiterschalten einer TCP-Verbindung zum sekundären Ziel könnte man sich so vorstellen, als ob der Zustand des Sockets, das dem primären Ziel entspricht, an ein anderes Socket übertragen wäre. In Abb. 4.4-6b wurde der Zustand von Socket (`Y,j`) auf Socket (`Z,k`) übertragen. Man könnte sich dies aber auch so vorstellen, als wäre Socket (`Z,k`) mit den Eigenschaften von Socket (`Y,j`) erzeugt (geklont). Daher spricht man hierbei von *Socket Cloning*.

TCP Handoff kommt u.a. in Web-Switching-Systemen vor, die in E-Commerce-Systemen mit zeitkritischem Content (z.B. Streaming-Media) eingesetzt werden.

4.4.8 MSS Clamping

Beim Aufbau einer TCP-Verbindung wählt der Client eine Segmentgröße `MSS` entsprechend der MTU des sendenden Interfaces – unter Abzug des IP- und TCP-Headers [Abb. 2.2-1, Abb. 4.3-2] – bei Ethernet (und einer MTU von 1500 Byte) mit 1460 Byte, sofern in der Routing-Tabelle kein anderer Wert hinterlegt ist.

Befindet sich der Client in einem per NAT eingerichteten Netzwerk, ist häufig die `MTU-Path-Discovery` [Abschnitt 3.7] aufgrund von Firewall-Regeln nicht möglich,

sodass die Router die IP-Pakete fragmentieren müssten, sofern das Transitnetzwerk diese IP-Paketgröße nicht unterstützt. IP-Fragmentierung ist aber sowohl aufwändig als auch performance-reduzierend. Einige Router ermöglichen daher, den Wert der MSS im <SYN>-Paket mit der MTU des Transfernetzes zu 'verzahnen', was sich als *MSS Clamping* oder – frei übersetzt – als *MSS Klempnern* bezeichnen lässt.

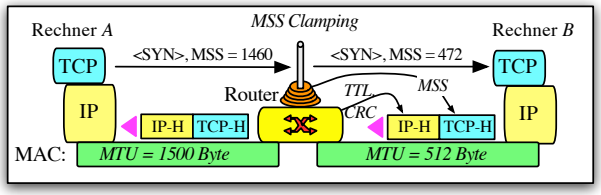


Abb. 4.4-8: MSS Clamping beim Aufbau einer TCP-Verbindung
MSS: Maximum Segment Size, MTU: Maximum Transfer Unit, IP-H: IP-Header, TCP-H: TCP-Header

MSS Clamping

Abb. 4.4-8 zeigt das Konzept des *MSS Clamping*. Ein Router, der zwei Netze mit unterschiedlichen MTUs (hier: 1500 und 512 Byte) verbindet, müsste normalerweise die vom Rechner *A* ausgehenden IP-Pakete bei der Übertragung zum Rechner *B* fragmentieren. Zusätzlich zu den Anpassungen der *TimeToLive* und des *CRC* im IPv4-Paket modifiziert er beim TCP-Verbindungsaufbau im TCP <SYN>-Paket den Wert von MSS = 1460 Byte, der im Options-Feld des TCP-Header hinterlegt ist und reduziert diesen auf 472 Byte. Somit greift der Router auf Informationen der Schicht 4 (Transport) zu und ändert diese, was eigentlich im Widerspruch zum Schichtenmodell ist. Dieses 'Klempnern' ist vor allem dann notwendig, falls ICMP-Nachrichten durch die Firewall blockiert sind. Für die Rechner *A* und *B* ist das MSS Clamping transparent, d.h. wird weder von diesen registriert noch besitzt es Einfluss auf die anschließende Datenübertragung.

4.5 Explicit Congestion Notification

Der TCP-Fenstermechanismus stellt eine effiziente Möglichkeit bereit, die zu übertragende Datenrate zwischen Sender und Empfänger abzustimmen, indem sich die Kommunikationspartner ihren Zustand gegenseitig mitteilen.

Congestion

Kommunikations- und Übertragungsengpässe (*Congestions*) können aber auch auf der Transitstrecke selbst, d.h. in den Routern auftreten. Abgesehen von der Möglichkeit, eine ICMP *Source Quench* Nachricht [Abschnitt 3.7] an das sendende System zu übermitteln, kann der Router nicht das Senden der Daten beeinflussen, auch wenn er selbst die Ursache ist. Wie wir bereits in Abschnitt 1.2 dargestellt haben, wirken sich Überlastsituationen sowohl hinsichtlich des erzielbaren Durchsatzes als auch im Hinblick auf die Verzögerung der Datenmittlung aus [Abb.1.2-5]. Ohne eine geeignete Überlastkontrolle kann das Netzwerk praktisch zusammenbrechen. Zur Verbesserung dieser Situation wurde ein spezielles Überlastsignalisierungsprotokoll in Form der *Explicit Congestion Notification* (ECN) entwickelt, das in RFC 3168 spezifiziert ist[2].

ECN verwendet zusätzliche Angaben im TCP- als auch im IP-Header in den zu übertragenden IP-Paketen und ist daher wohl der Schicht 3 als auch der Schicht 4 zuzuordnen.

[2] Ergänzende RFC sind 3540 und 4774 bzw. auch 6040.

Allerdings verlangt ECN, dass die für die Übertragung relevanten Knoten es auch unterstützen, was somit sowohl für die beteiligten Endsysteme als auch die Router gilt. Ein ECN-fähiger Router, über den eine TCP-Verbindung verläuft, kann dem Upstream-Rechner durch eine Verzahnung von IP- und TCP-Informationen *indirekt* signalisieren, dass eine Überlastsituation aufgetreten ist, sodass dieser die Menge der zu sendenden Daten reduzieren kann. Somit verbessert ECN nicht unmittelbar den TCP-Datendurchsatz (dieser ist durch das *Bandbreiten/Delay-Produkt* gegeben [Abschnitt 4.4]); bei Einsatz von ECN wird aber die Anzahl der ansonsten zu verwerfenden IP-Pakete verringert.

4.5.1 Anforderungen an ECN-fähige Netzknoten

Beim Routing von zu übertragenden IP-Paketen werden sie in der Regel temporär in einem Router (einem Netzknoten) gepuffert. Der Router organisiert seinen Paketspeicher in Warteschlangen (*Queues*), die z.B. entsprechend den Zieladressen, dem Payload (ICMP, TCP, UDP ...) und ggf. den Flusslabel (ToS/DS bei IPv4 [Abb. 3.2-2], Traffic Class und Flowlabel bei IPv6 [Abb. 8.2-1] 'bedient' werden: *Queue Management* [RFC 2309]. Abb. 4.5-1a zeigt den prinzipiellen Aufbau eines Routers mit Queue Management.

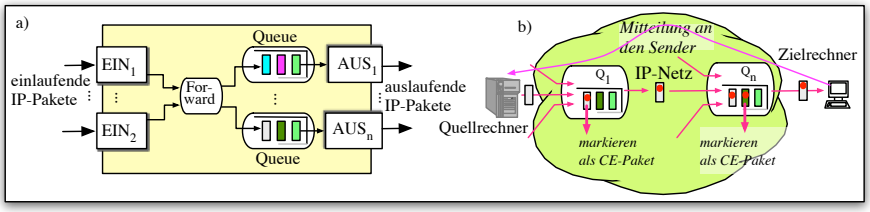


Abb. 4.5-1: Router Queue-Management: a) Aufbau eines Router mit AQM, b) markieren von IP-Paketen mit Flag *Congestion Experienced*
AQM: Active Queue Management

Bemerkung: Funktional stellt ein Router somit neben den Eingangsports (üblicherweise pro logischer oder physikalischer Schnittstelle) einen *Forwarder* zur Verfügung, der die Pakete an die Ausgangsports weiter leitet. Hierbei genügt es, die Pakete entsprechend der im IP-Header vorliegenden Informationen weiter zu leiten, weshalb auch der Begriff *IP-Switching* gebräuchlich ist, obwohl es sich eigentlich um ein Routing handelt. Die zum Versand bereit stehenden Pakete werden in unterschiedliche Queues eingereiht, die per Ausgangsport (entsprechend der Zieladresse) und Priorität organisiert sind.

FIFO-Queue

Die Queue stellt einen Puffer dar, der Pakete aufnehmen kann, solange hierin noch Platz vorhanden ist. Üblicherweise ist die Queue nach dem *FIFO-Prinzip* (*First-In/First-Out*) organisiert. Ist der Puffer schließlich voll, können die ankommenden Pakete nicht mehr gespeichert werden. Dies hat den gleichen Effekt wie das Verwerfen der Pakete.

AQM

Bei einer Überlastsituation [Abb. 1.2-5] ist die 'Leitung' nahezu vollständig ausgelastet, und entsprechend werden auch die Wartezeiten zum Versenden der IP-Pakete sehr lang. *Aktives Queue Management* (AQM) kommt dann zum Zuge, falls die Länge der Warteschlangen einen definierten Grenzwert überschreitet.

Congestion Experienced CE

Erst in diesem Fall werden die IP-Pakete einer TCP-Verbindung mit dem Flag *Congestion Experienced* (CE) markiert und in die Warteschlange eingereiht [Abb. 4.5-1b]. In der Regel befinden sich auf einer Strecke zwischen Quelle und Ziel mehrere Router mit jeweils eigenen solchen Warteschlangen. Durch Einsatz von ECN kann die mittlere Wartezeit *für alle Pakete in der Queue* reduziert werden, indem die Senderate der IP-Pakete beim Quellrechner bzw. beim vorausgehenden Router mit ECN-Befähigung an den Zustand der Queue angepasst wird.

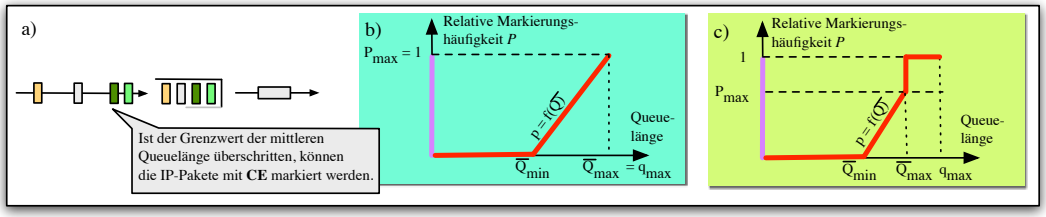


Abb. 4.5-2: Aktives Queue Management: a) zeitlicher Verlauf ankommender Pakete, b) AQM mit einem Grenzwert $\bar{Q}_{min}$, c) AQM mit den Grenzwerten $\bar{Q}_{min}$ und $\bar{Q}_{max}$

Congestion Experienced

Üblicherweise wird ein Grenzwert $\bar{Q}_{min}$ (für den mittleren Füllstand $\bar{Q}$) der Warteschlange (Queue) definiert, der angibt, ab wann von ECN Gebrauch gemacht wird. In Abb. 4.5-2a ist zunächst das zeitliche Verhalten der ankommenden IP-Pakete zu sehen. Diese füllen den Ausgangspuffer (Queue), dessen Füllstand vom Router beständig überwacht wird. Wird die Grenze $\bar{Q}_{min}$ erreicht kommt ECN zum Zuge. Anstatt aber alle Pakete mit `CE` zu markieren (vgl. Abb. 4.5-2b), wird zunächst nur jedes n-te Paket markiert, wobei der Anteil der markierten Pakete *P* eine lineare Funktion der Länge der Warteschlange ist, die ihren Maximalwert `1` bei einer Länge der Queue von $\bar{Q}_{max}$ erreicht, was der Größe des Puffers q_{max} entspricht.

Im Fall von Abb. 4.5-2c wurde ein anderer Algorithmus verwendet. Der Grenzwert für $\bar{Q}_{max}$ wird kleiner als die tatsächliche Größe des Puffers gewählt (um zusätzlichen Spielraum zu erhalten), und zusätzlich zur linearen Funktion $p(\bar{Q})$ werden ab diesem Grenzwert (der p_{max} entspricht) alle IP-Pakete als `CE` markiert.

RED

Dieses Verhalten von Routern, Überlastsituation quasi präventiv zu vermeiden, wird auch als *Random Early Detection/Discarding* (RED) bezeichnet.

Ursache von Überlastsituationen

Überlastsituation treten [Abb. 4.5-2] häufig bei der Kopplung 'großer' Netze über eine Punkt-to-Punkt Verbindung oder aber auch beim Aufbau einer vernetzen IP-Infrastruktur mittels eines VPN auf, sofern das Transitnetzwerk eine zu geringe Bandbreite besitzt, d.h. vor allem da, wo bereits gewisse Engpässe bereits existieren. ECN kann natürlich den Durchsatz einer Verbindung nicht verbessern, aber die bereit stehenden Ressourcen gleichmäßiger und gerechter verteilen.

4.5.2 Überlastkontrolle mit ECN

Wann ist der Einsatz von ECN sinnvoll?

Das zentrale Einsatzgebiet von ECN besteht darin, dass ein Router, über den die IP-Pakete übertragen werden, die Überlastsituation bei einer TCP-Verbindung erkennt und mittels ECN diese dem Empfänger (*downstream*) mitteilt, sodass dieser dem Sender die Reduktion der Datenrate bereits an der Quelle (*upstream*) nahelegt. Hierzu

werden bestimmte ECN-Angaben in IP-Paketen benötigt, die mittels zweier kleiner Erweiterungen sowohl im IP-Header als auch im TCP-Header geschaffen wurden. Um ECN zu nutzen, müssen die Netzwerkkomponenten die ECN-Erweiterungen verstehen und bei Überlastsituationen einsetzen, um Letztere zu signalisieren. Der gemischte Betrieb von Netzknoten mit und ohne ECN-Erweiterung ist hierbei gewährleistet.

Da ECN auf Informationen der Schicht 3 (IP) und Schicht 4 (TCP) angewiesen ist, muss dessen Aktivierung bei den Knoten in der Regel speziell vorgenommen werden. Bei den aktuellen Unix-Betriebssystemen (einschließlich Linux) verfügt der Network-Stack des Kernels bereits über ECN-Erweiterungen[3].

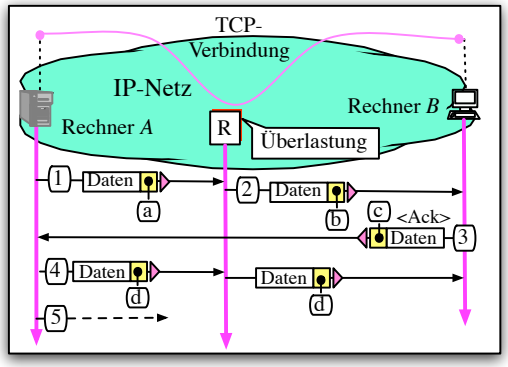


Abb. 4.5-3: Idee der Überlastvermeidung mit ECN
R: Router

Grundlegende Idee von ECN

Abb. 4.5-3 illustriert den prinzipiellen Ablauf der Überlastkontrolle bei ECN. Hierbei wurde angenommen, dass die TCP-Verbindung über einen überlasteten Router verläuft.

Überlastvermeidung

Wie aus Abb. 4.5-3 hervorgeht, sind folgende Schritte zu unterscheiden:

1. Rechner *A* hat ein IP-Paket abgeschickt, in dessen IP-Header mittels der Angabe '`a`' signalisiert wird, dass beide Kommunikationspartner (Rechner *A* und *B*) die Überlastkontrolle nach ECN unterstützen, d.h. beide sind ECN-fähig.
2. Das IP-Paket hat der überlastete Router empfangen und will nun diesen Zustand den Rechnern *A* und *B* anzeigen. Hierzu nimmt er einen Eintrag im IP-Header vor, indem er dieses durch Angabe '`b`' als `CE`-Paket (*Congestion Experienced*) markiert und so *B* seinen Überlastzustand signalisiert.
3. Rechner *B* hat das IP-Paket empfangen. Um die in diesem IP-Paket empfangenen Daten zu bestätigen, sendet er ein entsprechendes IP-Paket an Rechner *A*, in dem er diesem signalisiert, dass das Netz überlastet ist. Dazu macht er die Angabe '`c`' nun im TCP-Header. Dieses IP-Paket kann – muss aber nicht – über den überlasteten Router übertragen werden.
4. Nach dem Empfang des Pakets mit der Überlastungsanzeige im TCP-Header halbiert Rechner *A* seine Window-Größe und signalisiert dies Rechner *B* im nächsten übertragenen TCP-Segment. Dafür ergänzt er den TCP-Header um die Angabe '`d`'. Hat

[3] Eine Übersicht liefert: `https://www.icir.org/floyd/ecn.html`

Rechner *B* das IP-Paket empfangen, haben sich die beiden Rechner darauf verständigt, dass die Window-Größe auf dieser gerichteten TCP-Verbindung reduziert wurde.

5. Rechner *A* muss daraufhin seinen ins Netz gesendeten Datenstrom bewusst drosseln, z.B. nach dem im RFC 5681 spezifizierten Verfahren *Slow start and congestion avoidance algorithm* [Abschnitt 4.4]. Dies führt dazu, das die Datenrate des Rechners *A* sofort sinkt und der Router die Möglichkeit hat, die Belastung seines Puffers zu reduzieren. Empfängt der Rechner *A* über eine bestimmte Zeitdauer kein IP-Paket mit der Anzeige *Netz überlastet*, vergrößert er die Window-Größe wieder auf den alten Wert.

In Abb. 4.5-3 wurden die ECN-Angaben im IP- bzw. TCP-Paket beispielhaft als `a`, `b`, `c` und `d` bezeichnet. Im Weiteren wird gezeigt, um welche realen Angaben es sich handelt und wie sie eingesetzt sind.

4.5.3 Signalisierung von ECN in IP- und TCP-Headern

ECN-Angaben im IP-Header

Um den Routern die Möglichkeit zu verschaffen, ihre Überlastung den Endsystemen zu signalisieren, werden bestimmte ECN-Angaben im IP-Header gemacht, was Abb. 4.5-4a für IPv4 und Abb. 4.5-4b für das Protokoll IPv6 illustriert.

Differentiated Services Code Point

Das 8-Bit-Feld `ToS` (*Type of Service*) im IP-Header dient seit RFC 791 dazu, Prioritäten an die IP-Pakete vergeben zu können, was zunächst nur hypothetische Bedeutung hatte. Der Bedarf wurde aber in RFC 4594 aufgegriffen und das ToS-Feld umdefiniert, sodass es nun ein 6-Bit-Feld DSCP (*Differentiated Services Code Point*) sowie zwei weitere Bit enthält. Der DSCP-Wert dient zur Festlegung der Priorität des IPv4-Pakets, und die beiden zusätzlichen Bit ermöglichen die Signalisierung von ECN. Daher werden diese zwei Bit auch als *ECN-Feld* bezeichnet. Wie die Bit `x` und `y` im ECN-Feld belegt und interpretiert werden, zeigt Abb. 4.5-3c.

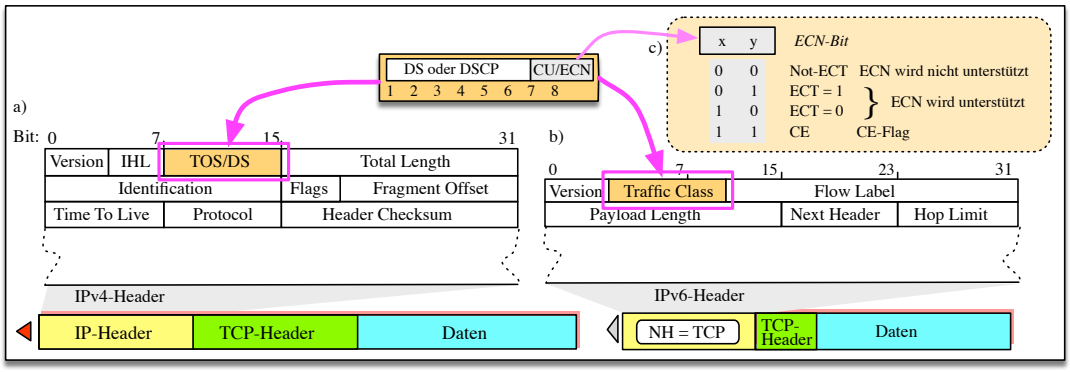


Abb. 4.5-4: ECN-Angaben im a) IPv4-Header, b) IPv6-Header und mit c) Interpretation der ECN-Bit
CE: Congestion Experienced, CU: Currently Unused, DS: Differentiated Services (DiffServ), DSCP: Differentiated Services Code Point, ECT: ECN-Capable Transport, IHL: Internet Header Length, ToS: Type of Service

Sind beide Bit `x` und `y` auf `0` gesetzt, so wird signalisiert, dass ECN nicht unterstützt wird. `ECT(0)` und `ECT(1)` können von ECN-fähigen Endsystemen als *Nonce* eingesetzt werden, mit denen sie sich dem jeweiligen Partner mitteilen. Durch Setzen der

Bit x und y auf 1 kann ein Router seine Überlastung den Endsystemen signalisieren. In diesem Fall spricht man von einem *CE-Paket*. Wurde das ECN-Feld von einem Router auf 1 gesetzt, so gilt das IP-Paket bereits als CE-Paket, und dieser Eintrag im ECN-Feld wird von nachfolgenden Routern nicht modifiziert.

ECN-Angaben im TCP-Header

Die ECN-Angaben, die zwischen den Endsystemen – also zwischen den kommunizierenden Rechnern, d.h. den *TCP-Peers* – zu übermitteln sind, werden im TCP-Header eingetragen [Abb. 4.5-5]. Die TCP-Peers zeigen daher an, dass sie ECN-fähig sind. Hat ein Rechner ein IP-Paket empfangen, das bereits ein Router als CE-Paket markierte und durch dass der Router seine Überlastung signalisiert, muss er dies dem Quellrechner mitteilen.

Der sendende Rechner ist darüber zu informieren, dass ein Router überlastet ist. Im Gegenzug teilt der Quellrechner nun auch dem Zielsystem mit, dass er dessen Warnung – d.h. dass ein Router überlastet ist – wahrgenommen und seine Senderate bereits verringert hat. Wie Abb. 4.5-6 illustriert, werden hierfür die beiden Flags ECE (*ECN-Echo*) und CWR (*Congestion Window Reduced*) im TCP-Header verwendet, die Teil der *Control Flags* CF sind.

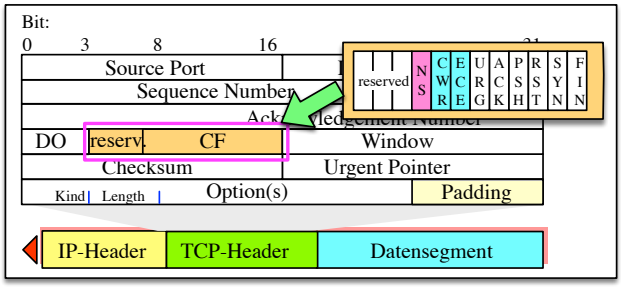


Abb. 4.5-5: ECN-Angaben im TCP-Header
ACK: Acknowledgement, CF: Control Flags, CWR: Congestion Window Reduced, DO: Data Offset, ECE: ECN-Echo, FIN: Finish, PSH: Push, RST: Reset, SYN: Synchronize, URG: Urgent Pointer, NS: Nonce Sum

Die Flags ECE und CWR haben folgende Bedeutung:

- **ECE** (*ECN-Echo*)
  Das Flag wird für zwei Zwecke verwendet. Einerseits dient es dazu anzuzeigen, dass ein Rechner ECN-fähig ist. Dies geschieht während des Aufbaus einer TCP-Verbindung mit der Überlastkontrolle (vgl. Abb. 4.5-6). Andererseits zeigt es an, dass das IP-Paket, in dem sich das TCP-Segment befindet, von einem Router als CE-Paket markiert wurde.
- **CWR** (*Congestion Window Reduced*)
  Das Flag wird vom sendenden Rechner – also dem Quellrechner – gesetzt und zeigt an, dass dieser ein TCP-Segment mit gesetztem ECE-Flag im TCP-Header empfangen und infolgedessen die Window-Größe und Senderate bereits reduziert hat [Abb. 4.5-8].
- **NC**: (*Nonce Sum*)
  Die Nonce Sum kann als ergänzende Paritätsinformation für die empfangenen TCP-Segmente mit ECN betrachtet werden.

Hat ein Router das ECN-Feld im IP-Header eines IP-Pakets auf 1 gesetzt, d.h. wenn er das Paket als CE-Paket markiert und damit signalisiert, dass er überlastet ist, weiß nur der Zielrechner dieses CE-Pakets, dass die TCP-Verbindung überlastet ist; der Quellrechner des IP-Pakets besitzt hierüber noch keine Informationen. Hierzu dient das Flag ECE, mit dessen Hilfe der Empfänger dies dem Quellrechner signalisiert.

4.5.4 Ablauf des ECN-Verfahrens

Aufbau einer TCP-Verbindung mit der Überlastkontrolle

Um die Überlast während einer TCP-Verbindung zu kontrollieren, müssen die beiden kommunizierenden Rechner sich zunächst hierauf verständigen. Abb. 4.5-6 zeigt den Ablauf des *ECN-setup*. Wie hier ersichtlich ist, erfolgt der Aufbau der TCP-Verbindung nach dem bekannten Prinzip, also dem 3-Way-Handshake-Verfahren, wie es beim TCP normalerweise der Fall ist.

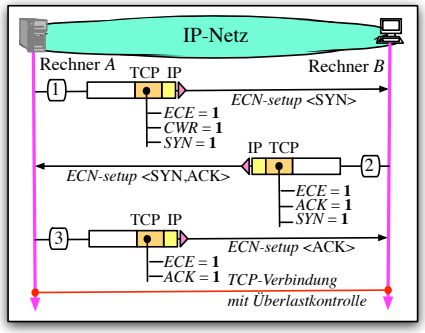


Abb. 4.5-6: Aufbau einer TCP-Verbindung mit Überlastkontrolle
ACK: Acknowledgement, CWR: Congestion Window Reduced, ECE: ECN-Echo, SYN: Synchronize

ECN-Verfahren

Wir betrachten die beiden ECN-fähigen Rechner *A* und *B* in Abb. 4.5-6.

- Der ECN-fähige Rechner *A* initiiert eine TCP-Verbindung zu Rechner *B* mittels eines <SYN>-Segments. Im TCP-Header des <SYN>-Pakets werden zusätzlich die beiden ECN-Flags, d.h. ECE und CWR, auf 1 gesetzt. Ein derartiges <SYN>-Paket wird *ECN-setup-SYN-Paket* genannt. Die Angaben ECE = 1 und CWR = 1 signalisieren Rechner *B*, dass der Quellrechner (hier *A*) ECN-fähig ist.
- Ist Rechner *B* ebenso ECN-fähig, signalisiert er dies Rechner *A* durch Setzen des Flag ECE auf 1. Somit wird das ECN-setup-SYN-Paket mit dem Paket <SYN,ACK>, quittiert, in dem ebenfalls im TCP-Header neben SYN = 1 und ACK = 1 auch ECE = 1 gesetzt ist, was als *ECN-setup-SYN-ACK-Paket* bezeichnet wird.
- Den Empfang des *ECN-setup-SYN-ACK-Pakets* bestätigt Rechner *A* mit dem *ECN-setup-ACK-Paket*, d.h. mit einem <ACK>-Paket und gesetztem ECE-Flag im TCP-Header. Damit wurde eine TCP-Verbindung zwischen den Rechnern *A* und *B* mit der Möglichkeit aufgebaut, im Bedarfsfall der Überlastung des Netzes entgegenzuwirken.

Entdeckung eines ECN-unfähigen Rechners

Bisher wurde vorausgesetzt, dass die beiden kommunizierenden Rechner ECN-fähig sind. In der Praxis aber ist das nicht immer der Fall. Daher muss der Rechner, der

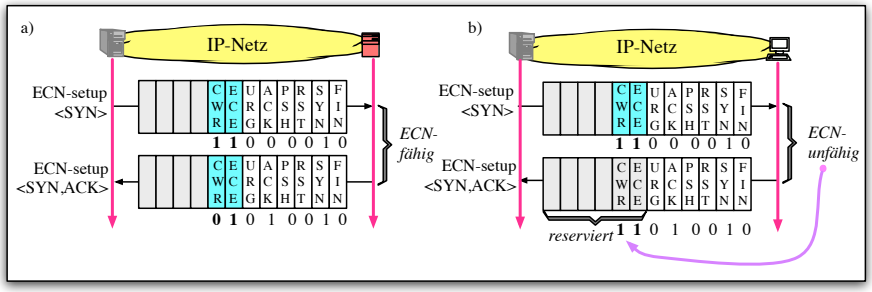


Abb. 4.5-7: Feststellung der ECN-Befähigung: a) ECN-fähiger, b) ECN-unfähiger Rechner
ACK: Acknowledgement, CWR: Congestion Window Reduced, ECE: ECN-Echo, FIN: Finish, PSH: Push, RST: Reset, SYN: Synchronize, URG: Urgent Pointer

eine TCP-Verbindung mit der Überlastkontrolle initiiert, erkennen können, ob die Router-Interfaces in Zielrichtung ECN unterstützten. Hierbei sendet der Quellrechner zunächst ein *ECN-setup-SYN-Paket* mit den Angaben `ECE = 1` und `CWR = 1`. Damit teilt der die TCP-Verbindung initiierende Rechner mit, dass er ECN-fähig ist.

Entdeckung eines ECN-unfähigen Rechners

Entsprechend Abb. 4.5-7 belegt der ECN-unfähige Rechner die beiden Flags `ECE = 1` und `CWR = 1` im zurückzusendenden TCP-Paket um. Ist an der CRW-Stelle der Wert 1, erkennt Rechner *A*, dass Rechner *B* nicht ECN-fähig ist. Ist ein Rechner nicht ECN-fähig, wird zwischen den Rechnern nur eine normale TCP-Verbindung aufgebaut.

Vermeidung der Überlast auf einer TCP-Verbindung

Die grundlegende Idee der Überlastvermeidung auf einer TCP-Verbindung wurde bereits in Abb. 4.5-4 dargestellt. Dort wurden aber die ECN-Angaben im IP- und TCP-Header nur kurz erwähnt, ohne auf die Details einzugehen. Abb. 4.7-8 illustriert die gleiche Situation unter Angabe der ECN-Flags im IP- und TCP-Header.

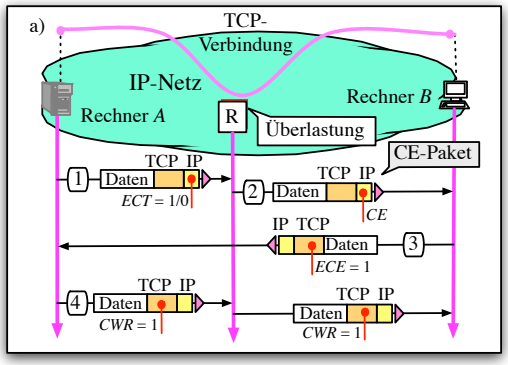


Abb. 4.5-8: Vermeiden der Überlast auf einer TCP-Verbindung durch ECN-Signalisierung
R: Router, CE: Congestion Experienced; weitere Abkürzungen siehe Abb. 4.5-7

Überlastsituation bei ECN

Abb. 4.5-8 illustriert den Ablauf im Falle einer Überlastsituation:

1. Der ECN-fähige Rechner *A* sendet ein IP-Paket, in dem der IP-Header im ECN-Feld die Bitkombination `01` oder `10` enthält. Damit wird auf `ECT(0)` (`ECT=0`) oder

`ECT(1)` (ECN-capable Transport; `ECT=1`) verwiesen, also darauf, dass es sich um eine TCP-Verbindung mit ECN-Überlastkontrolle handelt.

2. Ein ECN-fähiger Router, der bereits ausgelastet ist, setzt das ECN-Feld im IP-Header auf `1`. Damit wird das IP-Paket als CE-Paket markiert, und der Router zeigt so an, dass er überlastet ist.
3. Rechner *B* sendet nun ein IP-Paket – entweder mit den Daten oder nur als Quittung (`ACK`) – an Rechner *A*. In diesem IP-Paket signalisiert Rechner *B* Rechner *A*, indem er das `ECE`-Flag (ECN-Echo) im TCP-Header auf `1` gesetzt hat, dass eine Überlastsituation aufgetreten ist.
4. Hat Rechner *A* das IP-Paket mit `ECE = 1` empfangen, erfährt er damit von der Überlastsituation. Er reduziert dann die Window-Größe um die Hälfte, führt den '*Slow start and congestion avoidance algorithm*' aus und setzt im TCP-Header des nächsten abgeschickten IP-Pakets das Flag `CWR` auf `1`. Damit signalisiert Rechner *A* Rechner *B*, dass er auf dessen Warnung vor der Überlastsituation bereits entsprechend reagiert hat.

ECN mit Nonce Sum

In Abb. 4.5-6 wurde gezeigt, dass die Rückkopplung über eine mögliche Überlastsituation mittels der TCP-`<ACK>`-Segmente erfolgt, die der Zielrechner dem Quellsystem zukommen lässt. Wie kann aber nun der Zielrechner sicher sein, dass der Quellrechner sich auch an die Spielregeln hält, d.h. eine mögliche Überlastsituation per CE-Paket auch korrekt verarbeitet? Die Lösung bietet die Protokollerweiterung *Robust Explicit Congestion Notification* entsprechend RFC 3540.

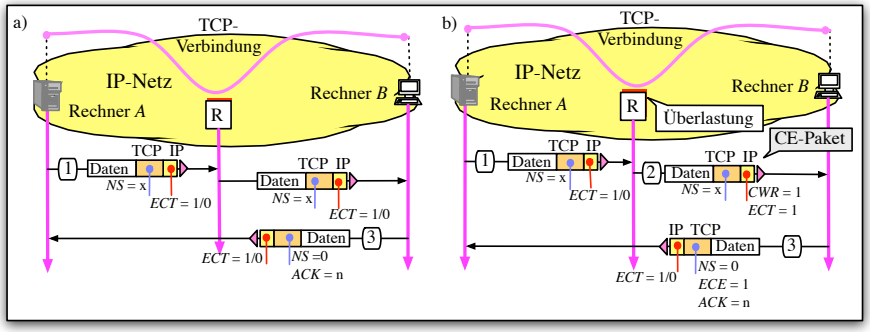


Abb. 4.5-9: Robust ECN: a) ohne Überlast, b) bei Überlast
NS: Nonce Sum, CE: Congestion Experienced; weitere Abkürzungen siehe Abb. 4.5-7

Robust Explicit Congestion Notification

Hierzu betrachten wir zunächst in Abb. 4.5-9a den Gutfall:

- Der Rechner *A* fügt zu den bereits diskutierten ECN-Bit im IP- und TCP-Header hier noch ein '*Nonce Sum*'-Bit (`NS`) hinzu, das aus dem Feld 'Reserviert' im TCP-Header entnommen wird [Abb. 4.5-5].
- Wird das IP-Paket fehlerfrei übertragen, ist der Zielrechner angehalten, dieses Bit per `XOR` mit dem Feld `ECT(0)` bzw. `ECT(1)` zu verknüpfen und in den Header des `<ACK>`-Segments einzubetten. Dieses Verfahren findet iterativ für jedes `<ACK>`-Paket statt, weshalb auch von einer *Nonce Sum* gesprochen wird.
- Ist die Angabe korrekt, kann der Quellrechner erkennen, dass der TCP-Partner ECN aktiv unterstützt.

Die Interpretation der *Nonce Sum* ändert sich im Fall einer Überlast [Abb. 4.5-9b]:

- Nach dem Versand des IP-Pakets und bei Vorlage einer Überlast ändert der Router die Bedeutung der ECN-Bit im IP-Paket, indem er diese auf '11' setzt (CE-Paket).
- Wertet das Zielsystem diese Information korrekt aus, wird im anschließenden <ACK> das NS-Bit im TCP-Header auf '0' (oder auf den vorigen Wert) gesetzt.

Zwar bietet das 'robuste' ECN nur eine 50% Chance, den Zustand korrekt zu erkennen und verlangt zudem von beiden Systemen neben dem (zufällig gewählten) Anfangswert des NS-Bit auch den Zustand der Verbindungen im Auge zu behalten. Da aber <ACK>-Segmente auch von Paketverlusten betroffen sein können, stellt das robuste ECN einen zusätzlichen – wenn auch nicht starken – Schutz dar.

4.6 Multipath TCP

Idee von MPTCP

MPTCP (*Multipath TCP)* beschreibt ein Konzept, nach dem das klassische verbindungsorientierte Transportprotokoll TCP um die Funktionalität *Multipathing*, d.h. um die Fähigkeit, Daten über parallele Paths transportieren zu können, erweitert wird. Zwei Rechner mit MPTCP können untereinander eine virtuelle MPTCP-Verbindung (*MPTCP Connection*) einrichten. Diese wird aus mehreren, über verschiedene Pfade verlaufende, als *Subflows* bezeichnete *TCP-Subverbindungen* gebildet. Die Anzahl von Subflows kann sich sogar während einer bestehenden MPTCP-Verbindung ändern; ein Subflow kann dynamisch hinzugefügt oder entfernt werden. Durch diese Möglichkeiten lässt sich eine beachtliche Verbesserung der Effizienz für die Internet-Kommunikation erzielen.

Use Case von MPTCP

Während Desktop-PCs und speziell natürlich auch Server in der Regel eine feste und im Wesentlichen dauerhafte Anbindung an das lokale Netz besitzen, sind besonders mobile Endgeräte wie Smartphones und Tablets häufig von stark schwankenden Netzanbindungen betroffen. Zudem haben diese Geräte mehrere aktive Interfaces für den Netzzugang: Eins für das WLAN (bevorzugt) und ein anderes für den Zugang über das Mobilfunknetz UMTS (*Universal Mobile Telecommunications System*) bzw. LTE (*Long Term Evolution*). Mobile Endgeräte sind also nicht nur *Multihomed*, sondern auch *Multilinked*.

Gerade für die Echtzeitkommunikation, so wie sie in den bisherigen Abschnitten dargestellt wurde, besteht der Bedarf, die Anwendung auch dann fortsetzen zu können,

- wenn ein Data-Link abbricht bzw. oder die TCP-Verbindung über IP neu aufgebaut werden muss, um z.B. von einem Provider zum anderen zu wechseln,
- den verfügbaren maximalen Datendurchsatz zu erzielen, falls beide Interfaces gemeinsam auf eine Ressource im Internet zugreifen können.

Die Herausforderung besteht nun darin, die Multipath TCP-Kommunikation so zu konstruieren, die weder für den Client noch für den Server eine Änderung des Socket-Interface notwendig zu machen. Der MPTCP-Dienst soll quasi transparent von den Applikationen genutzt werden. Dies wird durch eine zusätzliche *Aggregation-Layer* realisiert, der auf der Betriebssystemseite über ein *logisches Interface* bereitgestellt wird [Abb. 4.6-2].

Die Transportschicht mit MPTCP stellt sich für die Applikationsschicht wie eine übliche TCP-Implementierung dar, die über einen Standard-Socket-Aufruf angesprochen werden kann. MPTCP ist eine Erweiterung von TCP, sodass ein Rechner mit MPTCP auch in der Lage ist, über eine klassische TCP-Verbindung zu kommunizieren, sodass sich MPTCP hierbei wie 'normales' TCP verhält. MPTCP kann somit als eine Erweiterung von TCP angesehen werden, um Daten parallel über mehrere Pfade zu transportieren.

Entwicklung und Implementierung von MPTCP

An der Entwicklung von MPTCP wird schon seit ca. 2009 intensiv gearbeitet, und alle Aktivitäten werden von der Working Group MPTCP der IETF koordiniert. Das Konzept und die Einsatzmöglichkeiten von MPTCP sind bereits in mehreren RFC spezifiziert; dem Konzept von MPTCP widmet sich insbesondere RFC 6824. Eine MTCP Version 1 findet sich in RFC 8624, die allerdings zum vorherigen Entwurf nicht rückwärtskompatibel ist und unseres Wissens auch noch nicht unterstützt wird.

Eine Implementierung von MPTCP für Linux ist seit längerem vorhanden [https://www.multipath-tcp.org/]; ebenso eine experimentelle für FreeBSD [https://freebsdfoundation.org/project/multipath-tcp-for-freebsd/].

Wir wollen nun das Konzept des Protokolls MPTCP erläutern und dessen Bedeutung sowie Einsatzmöglichkeiten aufzeigen. Dabei präsentiert der folgende Abschnitt, wie verschiedene, aus parallelen Pfade bestehende MPTCP-Verbindungen auf- und abgebaut werden und wie die Übermittlung von Daten über diese Verbindungen erfolgt.

4.6.1 Typischer Einsatz von MPTCP

Wann ist MPTCP von Bedeutung?

MPTCP bringt große Vorteile in Rechnern bzw. anderen, insbesondere mobilen Endeinrichtungen, die über mehrere Interfaces zur Datenkommunikation verfügen. Abb. 4.6-1 illustriert die Bedeutung von MPTCP in mobilen Endeinrichtungen sowie in Servern in Datacentern. Abb. 4.6-1a zeigt den Einsatz von MPTCP in mobilen Endeinrichtungen, in mobilen Clients, mit zwei Interfaces für den Zugang zum Internet: Ein WLAN-Interface und ein 3G/4G-Interface[4]. In solchen mobilen Endeinrichtungen ist MPTCP von großer Bedeutung, wie wir dies nun näher erläutern möchten.

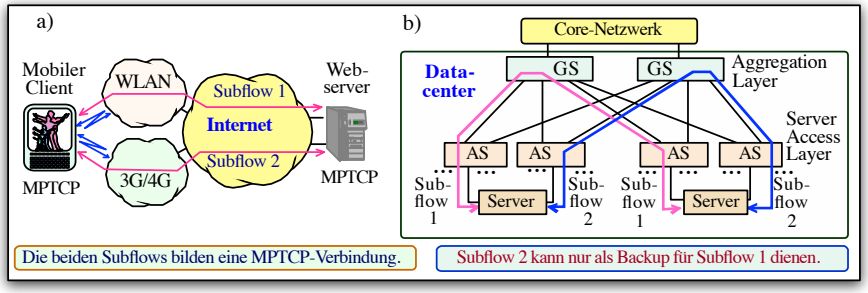


Abb. 4.6-1: Einsatz von MPTCP: a) in mobilen Endeinrichtungen, b) in Servern im Datacenter
3/4G: 3te/4te Generation der Mobilfunknetze, WLAN: Wireless Local Area Network

[4] Als 3G bezeichnet man die dritte Generation von Mobilfunknetzen – und dementsprechend als 4G die vierte Generation. Zur 3G gehören die Netzstandards UMTS und HSPA (*High Speed Packet Access*). 4G stellt den Netzstandard LTE dar.

MPTCP in mobilen Endeinrichtungen

1ter Subflow über WLAN und 2ter über Mobilfunknetz

Ein mobiler Client initiiert eine 'normale' TCP-Verbindung zum Server in der Regel zuerst über ein WLAN – also über den kostengünstigeren Internetzugang. Dabei zeigt der Client dem Server mittels der TCP-Option `MP_CAPABLE` an, dass er MPTCP-fähig ist [Abb. 4.6-7]. Der Server teilt dem Client ebenso mit, dass er MPTCP-fähig ist; MPTCP kann also zum Einsatz kommen. Die 1te TCP-Verbindung über das WLAN wird als *Subflow* angesehen – d.h. die TCP-Subverbindung einer MPTCP-Verbindung. Danach wird die 2te, vom Client initiierte und über das Mobilfunknetz (3G/4G) verlaufende TCP-Verbindung zum Server aufgebaut und diese stellt den 2ten Subflow dar. Die beiden Subflows werden nun entsprechend 'gebündelt' – und es entsteht auf diese Weise eine MPTCP-Verbindung zwischen Client und Server.

> **Anmerkung**: In der Regel kennt der Client, bevor er eine TCP-Verbindung zum Server initiiert, nur seinen Hostnamen. Aufgrund dessen kann er beim DNS nur die IP-Adresse vom Server abfragen und nicht die Anzahl dessen Internet-Interfaces. Beim Initiieren einer TCP-Verbindung weiß der Client also noch nicht, ob der Server über mehrere Interfaces verfügt. Der Server kann aber dem Client schon während des Aufbaus des 1ten Subflow in der TCP-Option `ADD_ADDR` mitteilen, dass er noch über eine weitere IP-Adresse und somit auch über ein weiteres Interface verfügt.

Nutzung von Subflows

Die folgenden Möglichkeiten kommen in Abb. 4.6-1a in Frage:

- Die transportierten Daten werden gleichmäßig auf parallele Subflows verteilt, was zur Erhöhung der Transportkapazität von Daten führt.
  Subflows → Load Balancing
- Die Daten werden nur über ein WLAN transportiert, und der über ein Mobilfunknetz G3/G4 verlaufende Subflow dient nur als Backup des anderen Subflow. Eine solche Lösung führt zur Reduzierung der für den Transport von Daten über ein Mobilfunknetz anfallenden Gebühren. Hat der mobile Client den vom WLAN versorgten Bereich verlassen, geht die über WLAN verlaufende TCP-Verbindung 'verloren' und die MPTCP-Verbindung reduziert sich zu einer TCP-Verbindung.
  Backup-Subflow

MPTCP in Datacentern

Lastverteilung im Aggregation Layer

In der Regel werden Server in Datacentern über zwei Ethernet-Interfaces an einen *Access Switch* angeschlossen [Abb. 16.5-2]. Wie Abb. 4.6-1b illustriert, kann zwischen zwei Servern jeweils eine aus zumindest zwei Subflows bestehende MPTCP-Verbindung aufgebaut werden. Auf diese Weise lässt sich Multipathing in Datacentern verwirklichen, ohne komplexe Lösungen auf der Basis von TRILL oder SPB realisieren zu müssen [Abschnitt 14.5]. Eine solche relativ einfache Multipathing-Lösung hat eine große Bedeutung, falls einer dieser beiden Server dem anderen als Backup dienen soll. Mithilfe von MPTCP in Servern innerhalb von Datacentern kann eine gleichmäßige Lastverteilung im *Aggregation Layer* erreicht werden.

Subflow-Nutzungsstrategien

Mehrere Subflows einer MPTCP-Verbindung kann man unterschiedlich nutzen, und man spricht dabei von *Subflow Policies*, also von *Subflow-Nutzungsstrategien*:

- Zur Erhöhung des Datendurchsatzes (also der Transportkapazität) werden die zu übertragenen Daten auf mehrere Subflows verteilt, d.h. es wird Load Balancing realisiert. Dies ist in Datacentern von großer Bedeutung [Abb. 4.6-1b].
  Load Balancing

Backup-Strategie

- Die Verbesserung der Verfügbarkeit einer Verbindung ist vor allem für mobile Endeinrichtungen mit zwei Interfaces, wie z.B. WLAN und 3G/4G verfolgt [Abb. 4.6-1a]. Eine mobile Endeinrichtung baut zuerst den 1ten Subflow zum Zielrechner über ein WLAN auf und dann einen 2ten Subflow über ein Mobilfunknetz (3G/4G). Der 2te Subflow dient aber nur als Backup für den 1ten, über das WLAN verlaufenden Subflow. Diese Nutzungsweise von Subflows wird zwischen den kommunizierenden Rechnern vereinbart. Hierfür nutzen sie das Flag-Bit `B` in der TCP-Option `MP-JOIN`.

  Hat die mobile Endeinrichtung aber das WLAN verlassen, ist der 1te Subflow nicht mehr verfügbar und erst dann kommt der 2te Subflow zum Einsatz. Demzufolge fällt die MPTCP-Verbindung auf eine normalen TCP-Verbindung zurück, aber die mobile Endeinrichtung hat weiterhin Internetzugang. Dank dieser Lösung können mobile Endeinrichtungen mit zwei Interfaces WLAN und 3G/4G flächendeckend Internetzugang haben und dabei auch die Möglichkeit, ein WLAN zu nutzen, um die anfallenden Datentransportkosten zu reduzieren.

Entlastungs-strategie

- Die Minderung der Überlastspitzen kann insbesondere in Rechnern mit zwei Interfaces praktiziert werden. Normalerweise werden Daten nur über den 1ten Subflow (also über den 1ten Path) transportiert. Sollte der 1te Subflow aber überlastet sein, dann wird der 2te Subflow dazugeschaltet, um den 1ten zu entlasten. Findet eine Überlastkontrolle nach ECN statt, dann ist diese Strategie einfach realisierbar.

4.6.2 Transportschicht mit MPTCP

Wir möchten jetzt ein vereinfachtes Modell der Rechnerkommunikation beim MPTCP zeigen und hierbei insbesondere die Struktur der Transportschicht mit MPTCP näher erläutern. Abb. 4.6-2 zeigt ein solches Modell[5].

Aufteilung der Transportschicht

Logisch betrachtet wird die Transportschicht in Rechnern mit MPTCP auf zwei Teilschichten aufgeteilt, nämlich auf eine Teilschicht *Multipath Management* (MPM), zu der die MPTCP-Instanzen gehören, und auf eine Teilschicht mit TCP-Instanzen, welche der herkömmlichen Transportschicht mit TCP entspricht.

Applikationen 'sehen' nur TCP

Das zentrale Konstruktionsmerkmal von MPTCP besteht somit darin, dass zwei kommunizierenden Applikationen die Transportschicht als eine normale, klassische Transportschicht 'sehen' und folglich 'glauben', zwischen ihnen verlaufe eine normale TCP-Verbindung. Dies ist dank der oberen Teilschicht MPM mit MPTCP-Instanzen möglich.

Grundlegende Idee von MPTCP

Um die grundlegende Idee von MPTCP zu erläutern, nehmen wir in Abb. 4.6-2 an, dass eine Client-Applikation im Quellrechner `A` eine Standard-TCP-Verbindung zu einer Server-Applikation auf Zielrechner `B` aufbauen möchte.

Eine 'normale' TCP-Sitzung wird gekennzeichnet durch

- den *Session-Identifier*, gegeben durch den Wert der `ISN` [Abschnitt 4.3] sowie

[5] siehe z.B. https://inl.info.ucl.ac.be/system/files/networking-mptcp.pdf

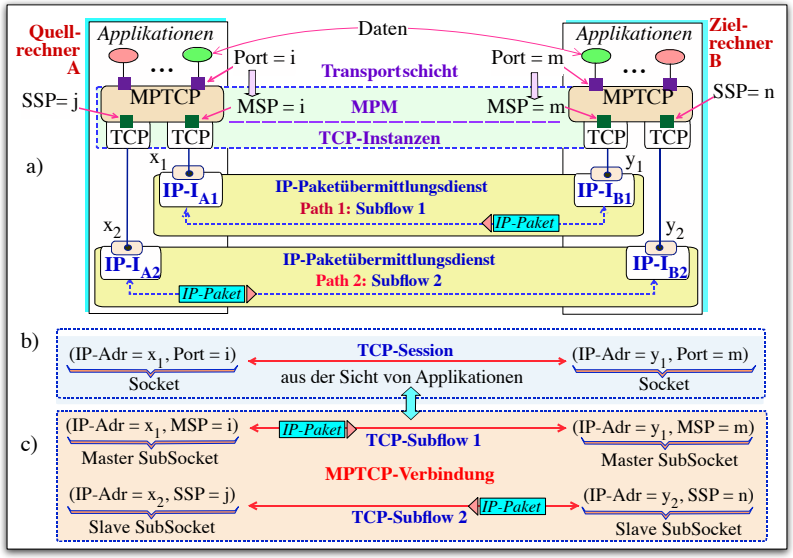


Abb. 4.6-2: Rechnerkommunikation mit MPTCP: a) Struktur der Transportschicht mit MPTCP, b) TCP-Sitzung aus der Sicht von Applikationen, c) MPTCP-Verbindung bilden zwei TCP-Subflows
IP-I: IP-Instanz, MPM: Multipath-Management, M/SSP:Master/Slave SubPort

- den *Verbindungs-Identifier* – also den beiden TCP-Socket – der aus den Tupeln `(Source-Port,Sender-IP-Adresse)` und `(Zielport,Ziel-IP-Adresse)` besteht. Da ein Socket entweder über eine IPv4- oder IPv6-Adresse genutzt werden kann, spricht RFC 6824 in diesem Zusammenhang auch von einem *4-Tupel*.

Feststellung der MPTCP-Fähigkeit des Partners

Diese Situation gilt bei einer Verbindungsaufnahme per MPTCP unverändert, nun jedoch für jeden einzelnen Subflow. Daher wird ein zusätzliches, für TCP-Subflows vorhandene Sitzungsmerkmal benötigt, das gleichzeitig genutzt wird, die MPTCP-Fähigkeit des Partners zu bestimmen:

- Sind beide Kommunikationspartner MPTCP-fähig, erzeugen diese bei der TCP-Verbindungsaufnahme einen 64 Bit langen kryptographischen Schlüssel und teilen sich diese in den Nachrichten `<SYN>` bzw. `<SYN,ACK>` in der Option des `MP_CAPABLE` als Key-A bzw. Key-B gegenseitig mit [Abb. 4.6-7], die beide abschließend in der finalen Nachricht `<ACK>` des *Three-Way-Handshakes* bestätigt werden.
- Hierdurch sind sich die Partner nicht nur sicher, dass sie gegenseitig MPTCP unterstützen und authentisiert sind, sondern auch, dass die notwendigen TCP-Protokollerweiterungen erfolgreich über das IP-Netz übermittelt werden können.

Registrierung der Subflows

Ist die primäre MPTCP-Verbindung initiiert, sodass die 1te TCP-Verbindung über den Path zwischen den IP-Adressen x_1 und y_1 in diesen Rechnern verläuft, wird die 2te TCP-Verbindung aufgebaut, und diese stellt den Subflow *2* dar. Das notwendige Binden von Subflow und IP-Adresse geschieht über die Mitteilung von `IP Address`. Die kommunizierenden Applikationen registrieren die zwischen ihnen bestehende,

zwei Subflows (als TCP-Subverbindungen) enthaltende MPTCP-Verbindung als normale TCP-Verbindung zwischen Socket (x1,i) im Rechner A und Socket (y1,m) im Rechner B (siehe hierzu auch Abb. 4.6-2b und Abb. 4.6-3a).

SubSockets adressieren einen Subflow

Es sei hervorgehoben, dass man in diesem Zusammenhang von *Socket*, *Master SubSocket* und von *Slave SubSocket* spricht[6] [Abb.4.6-2b, Abb.4.6-2c]. Diese Bezeichnungen basieren auf Folgendem: Normalerweise adressiert ein Socket einen Endpunkt einer Kommunikationsbeziehung, u.a. einer TCP-Verbindung. Dementsprechend adressiert ein SubSocket einen Endpunkt einer TCP-Subverbindung. Da bei MPTCP eine TCP-Subverbindung einen Subflow darstellt, repräsentiert ein SubSocket folglich einen Endpunkt eines Subflows.

Die in Abb. 4.6-2 gezeigte MPTCP-Verbindung bilden die zwei folgenden *Subflows* – nämlich: Subflow *1* zwischen Master SubSockets (x1,i) und (y1,m) sowie Subflow *2* zwischen Slave SubSockets (x2,j) und (y2,n). Diese Subflows können quasi als 'normale' TCP-Verbindungen angesehen werden – und zwar aus folgendem Grund: Die zuerst aufgebaute TCP-Verbindung, also die zwischen den Sockets (x1,i) und (y1,m), stellt die 1te TCP-Subverbindung des MPTCPs dar – d.h. den Subflow *1*.

Master und Slave SubSockets

Die SubSockets des zuerst aufgebauten Subflow – also des Subflows *1* – nennt man *Master SubSockets*, und als *Slave SubSockets* bezeichnet man die Endpunkte von weiteren, später aufgebauten Subflows. Die Ports in SubSockets nennt man *Subports*, entsprechend *Master Subports* in *Master!SubSockets* und *Slave Subports* in *Slave SubSockets*.

Aufgabe der Teilschicht MPM

Unter Bezug auf Abb. 4.6-2 möchten wird jetzt die Aufgabe der Teilschicht MPM kurz zusammenfassen: Eine Applikation im Quellrechner A initiiert (ohne Kenntnis zu besitzen, dass MPTCP zum Einsatz kommt) eine TCP-Verbindung zu einer Applikation im Zielrechner B. Für die initiierte TCP-Verbindung wird der Port i im Rechner A – oberhalb Schicht MPM – bereitgestellt; dieser Port wird als *Master Port* bezeichnet. Danach verläuft die TCP-Verbindung im Rechner A über die IP-Instanz 1 mit der IP-Adresse x1 und Interface 1. Demzufolge stellt der Socket (IP-Adresse x1, Port i) den Beginn der im Rechner A initiierten TCP-Verbindung dar.

Nutzung der Option MP_CAPABLE beim Aufbau 1tes Subflows

Vom Quellrechner A wird nun an den Zielrechner B ein TCP-Paket <SYN> gesendet. In diesem TCP-Paket verweist der Quellrechner mit der Option MP_CAPABLE darauf, dass er MPTCP-fähig ist [Abb. 4.6-7]. Für die 'ankommende' TCP-Verbindung wird im Rechner B der Port m – oberhalb Schicht MPM – bereitgestellt und ebenso als *Master Port* bezeichnet. Der Zielrechner B antwortet dem Quellrechner A mit dem TCP-Paket <SYN,ACK> und verweist durch der Option MP_CAPABLE, dass auch er MPTCP-fähig ist. Auf diese Weise haben sich die Rechner darüber verständigt, dass beide MPTCP einsetzen können.

Nachdem die beiden Rechner den Aufbau der 1ten TCP-Verbindung abgeschlossen und sich während des Aufbaus darüber verständigt haben, dass sie MPTCP-fähig sind, betrachten sie diese TCP-Verbindung als Subflow *1*. Wie Abb. 4.6-2a zum Ausdruck

[6] Die Begriffe *Master SubSocket* und *Slave SubSocket* sind entnommen aus: 'MultiPath TCP: From Theory to Practice' [https://inl.info.ucl.ac.be/system/files/networking-mptcp.pdf].

bringt, enthalten die TCP-Instanzen in beiden Rechnern des Subflows *1* de facto 'Kopien' des Master Ports. Nachdem der Subflow *1* eingerichtet wurde, wird die 2te TCP-Verbindung aufgebaut und als Subflow *2* angenommen.

4.6.3 Multipath-Kommunikation mit MPTCP

Abb. 4.6-3 zeigt ein allgemeines Modell der Kommunikation mit MPTCP. Mit dessen Hilfe möchten wir zum Ausdruck bringen, dass die kommunizierenden Applikationen letztlich eine TCP-Verbindung 'sehen' und nicht, dass ihnen eine aus zwei Subflows bestehende MPTCP-Verbindung bereitgestellt wird. Darüber hinaus wird gezeigt, dass auch jeder Rechner mit nur einem Interface MPTCP-fähig sein kann. Die in Abb. 4.6-3 dargestellten drei Fälle a), b) und c) lassen sich wie folgt charakterisieren [Abb. 4.6-2a]:

Dual Link

a) Die beiden Rechner verfügen über zwei Interfaces. Hier ist direkt ersichtlich, dass eine MPTCP-Verbindung ein 'Bündel' von TCP-Subverbindungen – Subflows genannt – darstellt. Es sei hervorgehoben, dass die mit einem 'x' markierten Ports oberhalb der MPTCP-Instanz, den die Applikationen 'sehen', und der Master SubPort (MSP) im Subflow 1 die gleiche Nummer haben[7]. Deswegen wurden diese beiden Ports mit einem 'x' markiert.

Rechner mit einem Interface MPTCP-fähig

b) Ein Rechner mit nur einem Interface und TCP über IPv4 oder über IPv6. Voraussetzung für MPTCP ist lediglich, dass der Rechner über mehrere ansprechbare IP-Adressen verfügt. Hier kann man entweder das Internetprotokoll IPv4 oder IPv6 verwenden. In Abb. 4.6-4 ist das Verhalten von Linux für die Vergabe mehrerer IP-Adressen für ein Interface dargestellt, wobei hierbei vom Konzept *virtueller Interfaces* Gebrauch gemacht wird. Schaut man sich Abb. 4.6-4 zudem genauer an, so stellt man fest, dass die Kommunikationspuffer `send-queue` und `ofo-queue` einzelner Subflows auch oberhalb einer TCP-Instanz entsprechend im Master SubPort und Slave SubPort untergebracht werden können. Die beiden Subflows nutzen hier im Rechner `B` die Ressourcen (NIC, IP-Instanz und TCP-Instanz) gemeinsam.

MPTCP-Verbindung mit IPv4 und IPv6

c) Ein Rechner mit nur einem Interface und TCP über IPv4 und IPv6. Auch ein Rechner mit nur einem Interface und mit den beiden Internetprotokollen IPv4 und IPv6 kann MPTCP-fähig sein. Dabei kann ein Subflow das IPv4 und der andere das IPv6 nutzen. Theoretisch betrachtet können somit die beiden Protokolle IPv4 und IPv6 in einer MPTCP-Verbindung gleichzeitig zum Einsatz kommen.

Modell der Rechnerkommunikation mit MPTCP

Zweistufige Datenflusskontrolle

Da TCP ein byte-orientiertes Protokoll ist, d.h. die übertragenen Daten werden bei TCP quasi byteweise nummeriert, muss der Fluss von Daten bei MPTCP auf den beiden Levels der Transportschicht, d.h. auf dem Level MPM und dem Level TCP, kontrolliert werden. Dementsprechend muss bei MPTCP eine *zweistufige Datenflusskontrolle* realisiert werden.

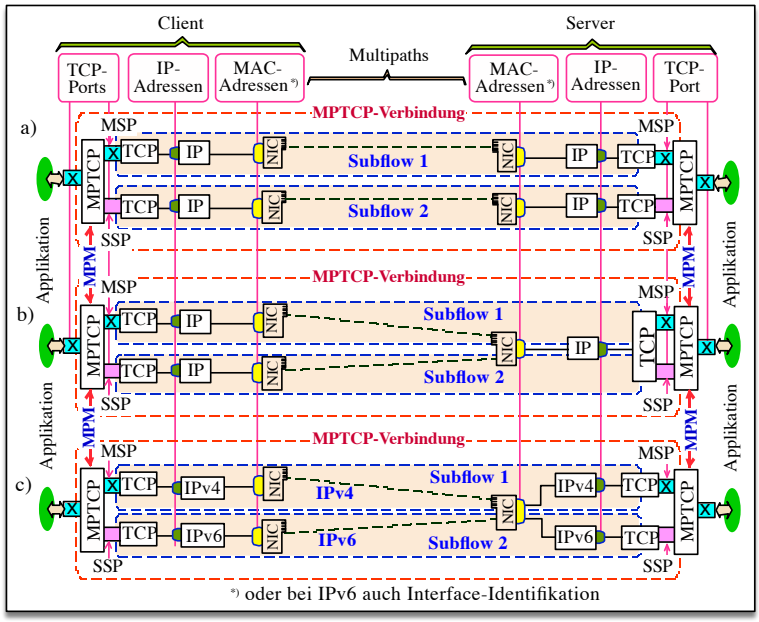


Abb. 4.6-3: Kommunikation mit MPTCP: a) beide Rechner mit zwei Interfaces; ein Rechner mit einem Interface und TCP über: b) IPv4 oder IPv6, c) IPv4 und IPv6
MPM: Multipath Management (u.a. Bündelung von Subflows), M/SSP: Master/Slave Sub-Port [Abb. 4.6-2]), NIC: Network Interface Controller (oft Ethernet-Adapterkarte)

Um dies erläutern zu können, illustriert Abb. 4.6-4 ein Modell der Rechnerkommunikation mit MPTCP und soll zum Ausdruck bringen, dass ein Rechner mit einer IP-Adresse auch MPTCP-fähig sein kann (vgl. hierzu Abb. 4.6-3c).

Falls die Übermittlungszeit der Datenblöcke – als *End-to-End Delay* bezeichnet – auf einem Path viel größer als auf einem anderen ist, kann dies, wie Abb. 4.6-5 zeigt, dazu führen, dass die Datenblöcke den Zielrechner nicht in der Reihenfolge erreichen, in der sie am Quellrechner abgeschickt wurden. Um eine solche Situation 'in den Griff' zu bekommen, müssen die Datenblöcke, deren Reihenfolge eventuell unkorrekt ist, zuerst in einem Zwischenempfangspuffer abgespeichert und anschließend in der korrekten Reihenfolge in einen Endempfangspuffer 'verlegt' werden. Erst von dort aus können sie an eine Applikation übergeben werden.

Zwischenempfangspuffer (`ofo-sende-queue`)

Die Datenblöcke zum Senden werden im Quellrechner entsprechend mithilfe eines sog. *Schedulers* auf einzelne Subflows verteilt und in die Sendepuffer `sende-queue` innerhalb der TCP-Instanzen einzelner Subflows eingereiht. Von dort werden die Datenblöcke über verschiedene Paths also folglich über verschiedene Transportwege an den Zielrechner übermittelt. In mehreren Subflows ankommende Datenblöcke werden im Zielrechner in Empfangspuffern innerhalb der TCP-Instanzen einzelner Subflows zwischengespeichert. Da die Datenblöcke in einzelnen IP-Paketen übermittelt werden und unterschiedliche Wege 'laufen' können, ist zu erwarten, dass die

[7] Die zuerst aufgebaute, also 1te TCP-Verbindung – TCP-Subverbindung 1 – wird Subflow 1 genannt.

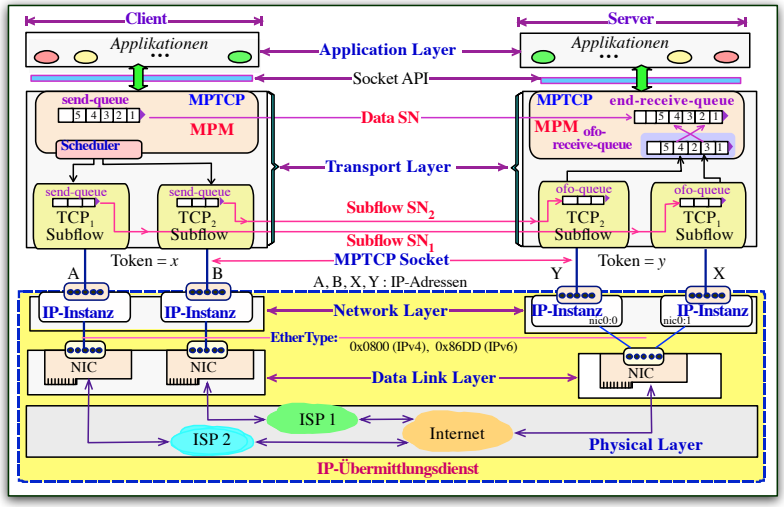


Abb. 4.6-4: Modell der Kommunikation mit MPTCP – im Hinblick auf die Datenflusskontrolle
API: Application Programming Interface, D/SSN: Data/Subflow Sequence Number, MPM: Multipath Management (MPTCP Level), ofo: out-of-order (siehe ofo-queue), TPNr: Transportprotokollnummer (TCP-Nummer = 6), NIC: Network Interface Controller
nic0:x: Instanz `x` des NIC `nic0`

Reihenfolge der auf jedem Subflow empfangenen Datenblöcke anders ist als die, in der sie im Quellrechner abgeschickt wurden. Demzufolge können die Datenblöcke in Empfangspuffern von Subflows, also auf dem Level TCP, Warteschlangen in unkorrekten Reihenfolgen bilden – also Warteschlangen vom Typ `ofo-receive-queue`[8].

Endempfangspuffer (`ofo-receive-queue`)

Damit man jede eventuell unkorrekte Reihenfolge der im Zielrechner empfangenen Datenblöcke immer korrigieren kann, sind, wie Abb. 4.6-4 zeigt, auf dem Level MPM zwei Empfangspuffer nötig: Ein Zwischenempfangspuffer mit der Warteschlange `ofo-receive-queue` von Datenblöcken in einer eventuell unkorrekten Reihenfolge und ein Endempfangspuffer mit der Warteschlange `end-receive-queue` von Datenblöcken in einer korrekten Reihenfolge. Bevor man aber die Daten an eine Applikation übergibt, werden sie von der Warteschlange `ofo-receive-queue` in die `end-receive-queue` kopiert [Abb. 4.6-5].

Zweistufige Datenflusskontrolle

Warum zweistufige Datenflusskontrolle?

Bei MPTCP werden – genauso wie bei TCP – Daten zwischen Applikationen in zwei kommunizierenden Rechnern als eine Folge von Byte so übermittelt, dass sie zuerst zu Datenblöcken einer bestimmten Größe zusammengefasst und aus den Datenblöcken dann TCP-Dateneinheiten gebildet werden. Anschließend werden die Dateneinheiten in IP-Paketen eingekapselt und über mehrere Paths einer MPTCP-Verbindung zum Ziel verschickt. Da die Datenblöcke in IP-Paketen über mehrere Paths und sogar auf einem Path über verschiedene Wege zum Ziel laufen können, muss bei MPTCP,

[8] Eine ofo-Receive-Queue repräsentiert einen Kommunikationspuffer, in dem Daten in einer – noch – unkorrekten Reihenfolge vorliegen können.

wie Abb. 4.6-5 illustriert, eine besondere zweistufige Datenflusskontrolle realisiert werden.

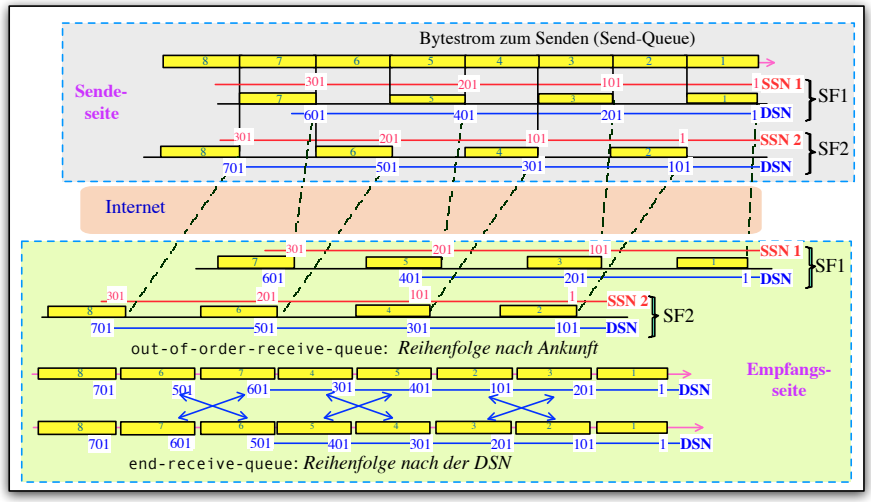


Abb. 4.6-5: Bedeutung von DSN (Data SN) und SSN (Subflow SN)
SN: Sequence Number, SSNi: SN auf dem Subflow i (i = 1, 2)

Zweistufige Nummerierung

Um den Datenfluss bei MPTCP kontrollieren zu können, müssen die als Folge von Byte zu übermittelnden Daten zweistufig nummeriert werden. In Abb. 4.6-5 wurde zwecks einer Vereinfachung der Darstellung angenommen, dass die Daten in Blöcken von je 100 Byte übermittelt werden. Die zu übermittelnden Byte werden auf eine besondere Art und Weise fortlaufend nummeriert – sowohl *global* auf dem Level MPM [Abb. 4.6-4] mit `DSN` (*Data Sequence Number*) als auch *lokal* auf dem Level TCP mit `SSN` (*Subflow Sequence Number*). Es sei aber angemerkt, dass die Daten auf jedem Subflow unabhängig von anderen Subflows (also lokal) mit `SSN` fortlaufend nummeriert werden – und zwar wird mit der `SSN` die Nummer des 1ten Byte in jedem Datenblock angegeben (ebenso wie mit der *Sequence Number* bei TCP). Zur Kontrolle der Datenübermittlung auf jedem Subflow dient daher die lokale Nummerierung mit der `SSN`.

Da jeder Subflow in der Tat eine 'normale' TCP-Verbindung darstellt, entspricht die `SSN` auf einem Subflow vollkommen der *Sequence Number* (`SN`) auf einer TCP-Verbindung. Demzufolge wird, wie Abb. 4.6-9 illustriert, die `SSN` anstelle der `SN` im TCP-Header eingetragen und gibt an, mit welcher Nummer die im betreffenden TCP-Paket übertragenen Datenbyte beginnen.

DSN in jedem TCP-Paket

Wie in Abb. 4.6-8 zum Ausdruck gebracht wurde, muss – um die Reihenfolge von Datenblöcken am Ziel in die richtige Reihenfolge zu 'bringen' – die `DSN` als globale Nummer von Datenbyte auch in jeden TCP-Paket enthalten sein. Die Angabe `DSN` wird in der TCP-Option `DSS` (*Data Sequence Signal*) übermittelt.

Abb. 4.6-5 zeigt zudem, dass die Daten in Blöcken an den Sendepuffer `sende-queue` des Quellrechners übergeben werden. Hierbei soll die globale Nummerierung mit der

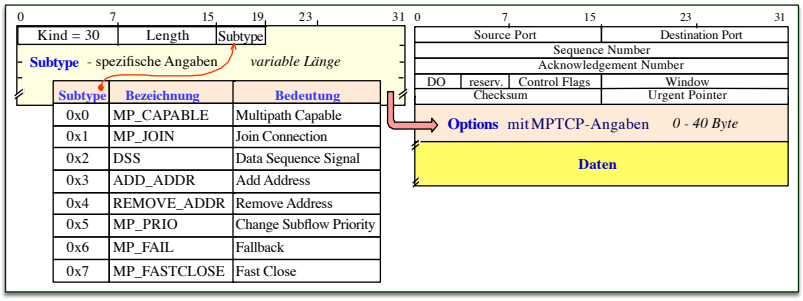
DSN auf dem Level MPM garantieren, dass die im Zielrechner empfangenen Datenblöcke in der korrekten Reihenfolge zum Endempfangspuffer (end-receive-queue) eingereiht werden können.

Bestätigung von empfangenen Daten

Entsprechend der zweistufigen Nummerierung der Datenbyte bei der Sendeseite mit SSN und DSN, sind die empfangenen Datenbyte beim Empfang zweistufig zu bestätigen. Als globale Quittung dient die Angabe Data ACK in der TCP-Option DSS und als 'lokale' Quittung auf jedem Subflow verwendet man die Subflow Acknowledgement Number im TCP-Header [Abb. 4.6-9]. Mit Data ACK wird die sog. kumulative ACK (Quittung) realisiert, genauso wie mit ACK bei TCP. Um die Effizienz des Datentransports zu verbessern, ist empfohlen, mit der SSN die sog. *Selective ACKs* nach RFC 2883 zu realisieren [Abschnitt 4.4].

4.6.4 MPTCP-Angaben im TCP-Header

Die MPTCP-Angaben werden im TCP-Header als TCP-Optionen übermittelt. Abb. 4.6-6 zeigt, welche neuen TCP-Optionen für MPTCP definiert und zu welchen Zwecken sie verwendet werden. Die einzelnen, zwischen Rechner A (als Initiator einer MPTCP-Verbindung) und Rechner B im TCP-Header übermittelten MPTCP-Angaben sind:



Subtype	Bezeichnung	Bedeutung
0x0	MP_CAPABLE	Multipath Capable
0x1	MP_JOIN	Join Connection
0x2	DSS	Data Sequence Signal
0x3	ADD_ADDR	Add Address
0x4	REMOVE_ADDR	Remove Address
0x5	MP_PRIO	Change Subflow Priority
0x6	MP_FAIL	Fallback
0x7	MP_FASTCLOSE	Fast Close

Abb. 4.6-6: Typen von TCP-Optionen mit MPTCP-Angaben im TCP-Header
DO: Data Offset (Angabe der Stelle, an der die Daten beginnen)

Aufbau des 1ten Subflow

- MP_CAPABLE: Mit dieser Option signalisiert ein Rechner seine MPTCP-Fähigkeit (*Multipath Capable*) [Abb. 4.6-7]. MP_CAPABLE wird nur beim Aufbau des 1ten Subflow verwendet und enthält zwei Arten von Angaben: die Angaben, die es dem Quellrechner A ermöglichen, zu überprüfen, ob der Zielrechner B auch MP-fähig ist, und die Angaben zu Sicherheitszwecken in Form eines 64 Bit-langen Senderschlüssels (Key-A) und einer 32 Bit langen Nonce, welche man als Key-B bezeichnet[9].

Aufbau von weiteren Subflows

- MP_JOIN: Mit dieser Option wird ein weiterer Subflow eingerichtet. MP_JOIN wird statt MP_CAPABLE beim Aufbau nächster Subflows verwendet [Abb.4.6-7] und enthält sowohl Angaben zur gegenseitigen Authentisierung beider Rechner, als auch Angaben, die dem Quellrechner A beim Aufbau des 2ten Subflow – also de facto

[9] Die Angaben *Keys* und *Nonces* beim Aufbau des 1ten Subflow sollen u.a. als 'Mittel' gegen verschiedene Spoofing-Angriffe dienen – in der Tat gegen TCP-Spoofing, also gegen eine Vortäuschung des wahren Initiators von nachfolgenden Subflows.

beim Aufbau einer MPTCP-Verbindung – ermöglichen, zusätzlich dem Zielrechner B die Identifikation der MPTCP-Verbindung als *Token* anzugeben. MP_JOIN enthält auch ein Flag-Bit B (*Backup-Flag*), damit der Quellrechner A dem Zielrechner B mitteilen kann, dass der 2te Subflow nur als Backup des 1ten Subflow dienen soll.

Fehler- und Datenflusskontrolle

- DSS (*Data Sequence Signal*): In der Option DSS werden Angaben zur Fehler- und Datenflusskontrolle übermittelt und zwar: Data ACK zur Bestätigung von Daten auf dem Level Multipath Management (MPM), Data Sequence Number (DSN) zur Nummerierung von Datenbyte auf dem Level MPM und Subflow Sequence Number (SSN) zur Nummerierung von Datenbyte auf dem Level TCP, ergo einem Subflow.

Angabe weiterer IP-Adresse

- ADD_ADDR: Mit dieser Option teilt ein Rechner einem anderen Rechner seine weitere IP-Adresse mit, sodass er an ein Interface mit dieser IP-Adresse einen weiteren Subflow initiieren kann [Abb. 4.6-7] .

Entfernen einer IP-Adresse

- REMOVE_ADDR: Mit dieser Option gibt ein Rechner einem anderen eine seiner IP-Adressen mit der Aufforderung an, den zu dieser Adresse führenden Subflow aus der zwischen ihnen bestehenden MPTCP-Verbindung zu entfernen.

Aufbau einer MPTCP-Verbindung (1)

> Zwischen einem Smartphone mit den zwei Interfaces WLAN und 3G/4G und einem Webserver bestand eine MPTCP-Verbindung mit zwei Subflows: Subflow *1* über ein WLAN und Subflow *2* über ein Mobilfunknetz. Der mobile Benutzer hat das WLAN mit seinem Smartphone verlassen. Demzufolge ist der Subflow *1* 'unbrauchbar' und er kann auch über WLAN nicht abgebaut werden. Über die noch aus dem Subflow *2* bestehende, über das Mobilfunknetz verlaufende MPTCP-Verbindung wird der Webserver vom Smartphone mit REMOVE_ADDR dazu aufgefordert, den noch zur in REMOVE_ADDR angegebenen IP-Adresse des WLAN-Interfaces im Smartphone führenden Subflow aus der MPTCP-Verbindung zu entfernen. Nachdem der Webserver dies 'erledigt' hat, ist aus der MPTCP-Verbindung zum Smartphone eine 'normale', lediglich über das Mobilfunknetz verlaufende TCP-Verbindung geworden.

Backup-Subflow ist zu verwenden

- MP_PRIO: Diese Option kann dann verwendet werden, falls eine MPTCP-Verbindung aus zwei Subflows besteht, wobei ein Subflow nur als Backup-Subflow des anderen 'lasttragenden', regulären Subflow dient. Ist der reguläre Subflow aber abgebrochen, muss dessen Aufgabe der Backup-Subflow übernehmen. Hierfür kann ein Rechner mit der Option MP_PRIO dem anderen Rechner mitteilen, dass der Backup-Subflow nun den regulären Subflow ersetzen soll; die Priorität des Backup-Subflow soll folglich geändert werden.

Aufbau einer MPTCP-Verbindung (2)

> Zwischen einem Smartphone und einem Webserver bestand – wie in Beispiel 1 – eine MPTCP-Verbindung mit zwei Subflows: Subflow *1* über ein WLAN und Subflow *2* über ein Mobilfunknetz. Der zweite dient nur als Backup des Subflow *1*. Der mobile Benutzer hat mit seinem Smartphone WLAN aber verlassen und demzufolge ist der Subflow *1* über WLAN 'abgebrochen'. Das Smartphone übermittelt an den Webserver nun die Option MP_PRIO, um ihm mitzuteilen, dass der Backup-Subflow *2* jetzt als 'lasttragender', regulärer Subflow fungieren soll. Folglich reduziert sich die MPTCP-Verbindung zu einer 'normalen' über das Mobilfunknetz verlaufenden TCP-Verbindung.

- `MP_FAIL`:Während einer MPTCP-Verbindung kann ein Path – und dadurch ein Subflow – gravierend gestört bzw. plötzlich abgebrochen werden[10]. Die MPTCP-Verbindung muss somit in einen vorherigen, noch normalen Zustand zurückgesetzt werden. Man spricht hierbei von *Fallback*. In einer solchen Situation wird der betreffende, also gestörte, Subflow aus der MPTCP-Verbindung 'entfernt' und die 'letzten', noch über den gestörten Subflow abgeschickten, vom Zielrechner noch nicht quittierten Daten müssen folglich über einen intakten Subflow erneut gesendet werden. Mit der `DSN` (*Data Sequence Number*) in der Option `MP_FAIL` signalisiert ein Rechner dem anderen, welche Daten (mit DNS beginnend) wiederholt über einen anderen intakten Subflow gesendet werden müssen.
- `MP_FASTCLOSE`: Diese Option dient dazu, eine ganze, aus mehreren Subflows bestehende MPTCP-Verbindung schnell schließen (beenden) zu können. `MP_FASTCLOSE` würde der Funktion nach dem `<RST>`-Paket entsprechen, d.h. dem TCP-Paket mit dem auf 1 gesetztem `RST`-Flag (Reset-Flag), über den der Abbau einer TCP-Verbindung in Fehlersituationen initiiert wird.

MPTCP-Verbindung schnell schließen

4.6.5 Aufbau einer MPTCP-Verbindung

In Abb.4.6-7 wird der Aufbau einer MPTCP-Verbindung gezeigt. Der Aufbau des 1ten Subflows [Abb. 4.4-3] folgt den gleichen Prinzipien wie der einer TCP-Verbindung insgesamt.

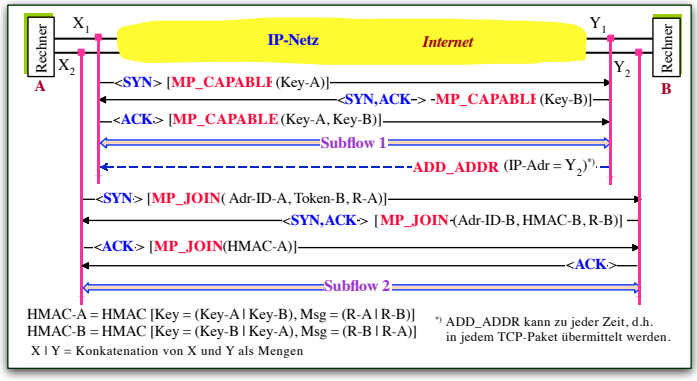


Abb. 4.6-7: Aufbau einer MPTCP-Verbindung [RFC 6824]
X_1, X_2, Y_1, Y_2: IP-Adressen, HMAC: Hash Message Authentication Code

Annahme: Rechner `A` kennt bereits den Hostnamen `www.abc.de` des Rechners `B` und möchte als Quellrechner eine MPTCP-Verbindung zu diesem initiieren. Hierfür fragt Rechner `A` beim DNS nach der IP-Adresse des Hostnamens `www.abc.de` und bekommt als Antwort dessen IP-Adresse Y_1 eines seiner Interfaces. Der Rechner `B` hat aber noch ein zweites Interface und demzufolge auch eine ihm entsprechende zweite IP-Adresse Y_2. Diese kennt der Rechner `A` noch nicht.

Um eine MPTCP-Verbindung zu Rechner `B` zu initiieren, sendet Rechner `A` – auf dem Interface mit der IP-Adresse X_1 – ein TCP-Paket `<SYN>`. In diesem verweist Rechner `A`

Aufbau des 1ten Subflow

[10] Dies kann vorkommen, falls eine MPTCP-spezifische Option im TCP-Header in einer *Middlebox* verändert bzw. entfernt wird [Abb. 4.6-10].

mit der Option MP_CAPABLE darauf, dass er MPTCP-fähig ist und folglich auch darauf, dass er eine MPTCP-Verbindung aufbauen möchte. Der Zielrechner B antwortet dem Rechner A – auf dem Interface mit der IP-Adresse Y_1 – mit dem TCP-Paket <SYN,ACK> und verweist mit MP_CAPABLE genauso darauf, dass er MPTCP-fähig ist. Somit haben sich die beiden Rechner darüber verständigt, dass sie beide MPTCP-fähig sind und eine MPTCP-Verbindung zwischen ihnen aufgebaut werden kann. Der Rechner A bestätigt den Empfang des TCP-Pakets <SYN,ACK> mit dem TCP-Paket <ACK>. Nachdem der Rechner B <ACK> empfangen hat, besteht zwischen ihnen bereits eine TCP-Verbindung – und diese wird dann in den 1ten Subflow 'umgewandelt'.

Sicherheitsangaben

In MP_CAPABLE werden die TCP-Verbindungs-Identifier, die sog. *Keys*, zur gegenseitigen Authentisierung übermittelt. Wie Abb.4.6-7 zeigt, ist in <SYN> nur der Schlüssel des Quellrechners <SYN> (Key-A) und in <SYN,ACK> dementsprechend nur der des Zielrechners B (Key-B) enthalten. Der Quellrechner A übermittelt an den Zielrechner B noch in <ACK>: den eigenen Schlüssel Key-A und den aus <SYN,ACK> kopierten Key-B. Die beim Quellrechner A 'gemachte' Kopie des Key-B soll beim Zielrechner B ein Echo seines Key-B darstellen. Stimmt das Echo des Key-B mit dessen Original beim Rechner B überein, kann dieser sicher sein, dass <ACK> vom wahren Absender kommt, d.h. vom Rechner A. Die beiden Keys beim Aufbau des 1ten Subflow werden als Klartext übermittelt.

Bedeutung der Option ADD_ADDR

Es wurde bereits darauf hingewiesen, dass der Rechner A die 2te IP-Adresse Y_2 des Rechners B noch nicht kennt. Das MPTCP wurde so konzipiert, dass der Zielrechner, falls er über mehrere Interfaces und demzufolge über mehrere IP-Adressen verfügt, jede weitere IP-Adresse zu der IP-Adresse, unter dem 1te Subflow bei ihm endet, mit der Option ADD_ADDR signalisieren/anzeigen kann. Dies kann er jederzeit 'tun', d.h. während des Aufbaus oder nach dem Aufbau des 1ten Subflow, also in der Praxis mit ADD_ADDR in jedem TCP-Paket. Abb. 4.6-7 bringt dies zum Ausdruck.

Aufbau des 2ten Subflow

Nachdem der Rechner A aus den Angaben in der Option ADD_ADDR 'erfahren' hat, dass der Rechner B noch über die 2te IP-Adresse Y_2 verfügt, initiiert Rechner A den 2ten Subflow an diese 2te IP-Adresse. Es sei angemerkt, dass der Aufbau des 2ten Subflow weitgehend nach dem gleichen Schema wie beim 1ten Subflow verläuft.

Token als Subflow-Identifier

Folgende Unterschiede sind beim Aufbau des 2ten Subflow [Abb. 4.6-7] hervorzuheben:

1. MP_JOIN <SYN>-Nachricht: Es wird die Option MP_JOIN verwendet und hierin ein den Subflow identifizierendes *Token* erzeugt, das ausgehend vom Key-B des Empfängerrechners zunächst als SHA-1 Hashwert berechnet und anschließend auf eine Länge von 32 Bit gekürzt wird. Ergänzt wird die Nachricht mit einem *Nonce* sowie mit der Angabe Address ID zur Unterstützung des Verfahrens bei NAT und einigen weitere Flags.
2. MP_JOIN <SYN,ACK>-Nachricht: Der Empfänger versendet in der Nachricht <SYN,ACK> ebenfalls die Option MP_JOIN, generiert aber nun einen HMAC (*Keyed-Hash Message Authentication Code*, vgl. Abschnitt 2.3), der aus den übertragenen Sitzungsschlüsseln sowie der ausgetauschten Nachricht(en) besteht und auf 64 Bit gekürzt wird. Auch hier wird ein *Nonce* mit 32 Bit eingefügt.

3. `MP_JOIN <ACK>`-Nachricht: Abschließend wird im Handshake wieder ein HMAC wie beim `<SYN,ACK>` berechnet, nun aber mit der Gesamtlänge von 160 Bit übermittelt.

Dieses Verfahren findet für alle Subflows Anwendung, wobei jeweils das 32 Bit *Token* den Subflow identifiziert. Das Token stellt somit eine Abstraktion des Sockets-Paars dar und ermöglicht die Aufteilung (und Zusammenführung der Subflows) nicht nur über unterschiedliche Links und IP-Adressen, sondern auch unabhängig davon, ob IPv4 oder IPv6 genutzt wird.

4.6.6 Anpassung des TCP-Headers für MPTCP

Ein Subflow innerhalb einer MPTCP-Verbindung stellt eine Sonderform einer TCP-Verbindung dar. Beim Transport von Daten über einen Subflow müssen somit einige dem Subflow und somit auch dem MPTCP entsprechenden Angaben im TCP-Header gemacht werden. Abb. 4.6-8 illustriert, um welche Angaben es sich dabei handelt, und zeigt auch, wie der TCP-Header an den MPTCP-Bedarf angepasst werden kann.

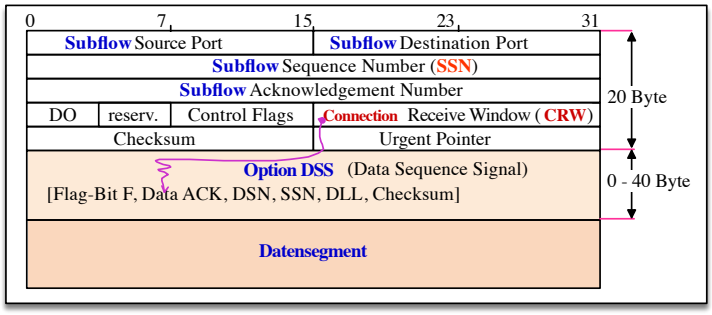


Abb. 4.6-8: Aufbau des TCP-Header mit MPTCP-relevanten Angaben
DO: Data Offset, DLL: Data-Level Length, DSN: Data Sequence Number

Zu Abb. 4.6-8 sei angemerkt, dass die Angaben `Source Port`, `Destination Port`, `Sequence Number`, `Acknowledgement Number` im TCP-Header nur einen Subflow bei MPTCP betreffen und dadurch anders zu interpretieren sind, nun zwar nun als `Subflow Source Port`, `Subflow Destination Port`, `Subflow Sequence Number`, `Subflow Acknowledgement Number`.

Angabe `Receive Window`

Dem 'Sender' bei TCP besagt die Angabe `Receive Window` in einem bei ihm empfangenen TCP-Paket, wie viele Daten (bei TCP in Byte ausgedrückt) er an den 'Empfänger' abschicken darf, ohne auf eine Quittung von diesem warten zu müssen. Mit `Receive Window` wird der Sender darüber informiert, wie groß der Puffer für die Aufnahme der von ihm kommenden Daten ist. Bei MPTCP bezieht sich die Angabe `Receive Window` auf alle Subflows einer MPTCP-Verbindung. Es handelt sich bei MPTCP um `Connection Receive Window` (CRW), d.h. die Puffergröße für eine MPTCP-Connection/Verbindung.

TCP-Option `DSS`

Abb. 4.6-8 bringt zum Ausdruck, dass im TCP-Header die Option `DSS` (*DataSequence Signal*) enthalten ist und somit die den Datenfluss auf der MPTCP-Verbindung be-

treffende Angabe `Data ACK` mit `CRW` zusammenhängt. Mit `Data ACK` zeigt der Empfänger dem Sender an, bis zu welchem Byte die Daten auf der MPTCP-Verbindung bei ihm korrekt empfangen wurden. Auf diese Weise erfährt der Sender, dass er an den Empfänger Daten bis zur Byte-Nummer `CRW+(Data ACK)` senden darf und, dass diese bei ihm im Puffer aufgenommen werden. So wird die Datenflusskontrolle auf dem MPTCP-Level (Schicht MPM, [Abb. 4.6-4a]) mittels der Angaben `CRW` und `Data ACK` realisiert [Abb. 4.6-5].

Die TCP-Option `DSS` wird verwendet, um einerseits die auf der MPTCP-Verbindung gesendete Folge von Byte mit einer Sendefolgenummer, als Datensequenznummer (`Data Sequence Number`, `DSN`) bezeichnet, zu nummerieren und andererseits die empfangene Folge von Byte mit einer *Quittungssequenznummer* (*Data Acknowledgement Number*, `Data ACK`) bestätigen/quittieren zu können.

Flag-Bit `F`

Eine besonders wichtige Bedeutung besitzt das Flag-Bit `F` (`Finish`) in der Option `DSS`. Mit `F = 1` wird signalisiert, dass nach dem Ende von Daten ein fiktives Datenbyte `DATA_FIN` [Abb. 4.6-9] vorhanden ist und dass damit der Abbau einer MPTCP-Verbindung initiiert wird.

4.6.7 Abbau einer MPTCP-Verbindung

Eine MPTCP-Verbindung muss auch abgebaut werden, damit die kommunizierenden Rechner reservierte Ressourcen freigeben können. Der Abbau könnte im Grunde so erfolgen, dass alle zur MPTCP-Verbindung gehörenden Subflows, als 'normale' TCP-Verbindungen, individuell und unabhängig voneinander abgebaut werden. In der Praxis kann aber ein Subflow auf einer MPTCP-Verbindung 'abgebrochen' – quasi 'aufgehängt' – werden, bevor man mit dem Abbau dieser beginnt. Eine solche Situation kommt oft vor, z.B. falls ein Smartphone während einer MPTCP-Verbindung, welche aus einem Subflow über ein WLAN und einem Subflow über ein G3/4G-Mobilfunknetz besteht, das WLAN plötzlich verlässt, ohne den Subflow vorher über ein WLAN abzubauen; folglich wird dieser Subflow dann einfach abgebrochen.

Bedeutung von `DATA_FIN`

Es muss bei MPTCP also möglich sein, eine MPTCP-Verbindung mit einem 'abgebrochenen' Subflow vollständig abbauen zu können. Hierfür wurde das fiktive Datenbyte `DATA_FIN` eingeführt. Mit dessen Hilfe kann ein Rechner den Abbau einer MPTCP-Verbindung, d.h. den Abbau aller Subflows auf einen 'Schlag', initiieren. Das fiktive Databyte `DATA_FIN` bei MPTCP hat im Grunde eine ähnliche Bedeutung wie das Flag-Bit `FIN` bei TCP. Abb. 4.6-9 illustriert den Abbau einer MPTCP-Verbindung – und somit auch die Bedeutung von `DATA_FIN`. Es sei hervorgehoben, dass `DATA_FIN` auf jedem Subflow gesendet werden kann und mit `Data ACK` bestätigt werden muss

Abbau einer MPTCP-Verbindung mittels `DATA_FIN`

> Der Abbau der MPTCP-Verbindung als Punkt am Satzende auf dem Subflow 1 wird mit einem `<FIN;DATA_FIN>`-Paket initiiert, d.h. mit einem TCP-Paket, in dem das Flag-Bit `FIN` (im TCP-Header) und das Flag-Bit `F` in der Option `DSS` auf 1 gesetzt sind, um das Zeichen `DATA_FIN` zu signalisieren. Das Zeichen `DATA_FIN` wird als ein fiktives Byte betrachtet, und dessen Empfang muss vom Zielrechner extra bestätigt werden, um sicher zu gehen, dass alle Daten bei ihm angekommen sind. Daher repräsentiert `DATA_FIN` ein fiktives Byte am Ende aller Daten – wie ein Punkt am Satzende. Im hier gezeigten Beispiel enthält das TCP-Paket den letzten Datenblock *(end-data)*: Dieser beginnt mit

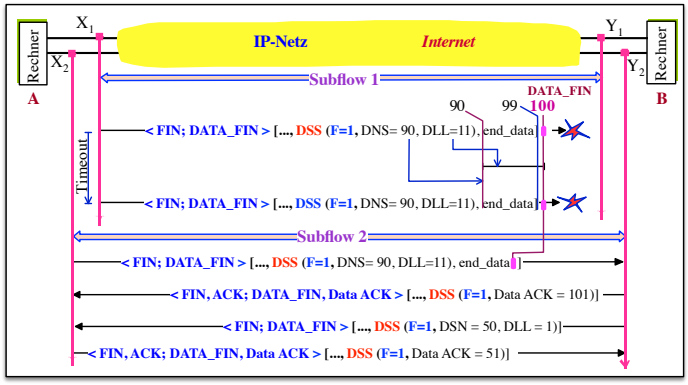


Abb. 4.6-9: Abbau einer MPTCP-Verbindung und Bedeutung der Angabe `DATA_FIN`
DSN: Data Sequence Number (globale Datenfolgenummer),
DLL: Data Level-Length (die Länge des Datenblocks [Abb. 4.6-8])

dem Datenbyte 90 (Angabe `DSN = 90`) und enthält zusammen mit dem fiktiven Byte `DATA_FIN` 11 Byte (DLL = 11); folglich hat `DATA_FIN` die Nummer 100.

In Abb. 4.6-9 wird muss bestätigt werden das `<FIN;DATA_FIN>`-Paket nach einem Timeout erneut gesendet. Da der Subflow *1* 'abgebrochen' ist, kommt nach dem *Timeout* keine Antwort und deswegen wird der Abbau der MPTCP-Verbindung auf dem Subflow *2* genauso wie vorher mit `<FIN;DATA_FIN>` initiiert. Die Gegenseite (Rechner `B`) antwortet darauf mit dem `<FIN,ACK;DATA_FIN,Data ACK>`-Paket, d.h. mit dem Paket, in dem zusätzlich das Flag-Bit `ACK` (im TCP-Header) auf 1 gesetzt ist und `Data ACK = 101` in der Option `DSS` enthalten ist [Abb. 4.6-8]. Mit `Data ACK = 101` signalisiert die Gegenseite, dass sie jetzt das Datenbyte mit Nummer 101 erwartet. Auf diese Weise bestätigt sie den Empfang von `DATA_FIN`. Dadurch ist der Rechner `A` sicher, dass alle von ihm abgeschickten Daten das Ziel (den Rechner `B`) erreicht haben und dort auch aufgenommen wurden.

Genauso wie beim Abbau jeder TCP-Verbindungmuss beim Abbau einer MPTCP-Verbindung jede Seite den Abbau initiieren, um sicher sein zu können, dass alle von ihr abgeschickten Daten das Ziel erreicht haben und dort aufgenommen wurden. Daher wird der gleiche Vorgang seitens des Rechners `B` wiederholt, d.h, das `<FIN;DATA_FIN>`-Paket wird an den Rechner `A` geschickt. Da der Rechner `B` keine Daten mehr zum Senden hat, und das letzte, an den Rechner `A` geschickte Datenbyte die Nummer 49 aufweist, enthält `<FIN;DATA_FIN>` die Datensendefolgenummer `DSN = 50` und `DLL = 1`. Die Gegenseite (Rechner `A`) antwortet darauf mit dem `<FIN,ACK;DATA_FIN,Data ACK>`-Paket, in dem die Anzeige `DATA_FIN` mit `Data ACK = 51` bestätigt wird. Dadurch erfährt Rechner `B`, dass alle von ihm abgeschickten Daten den Rechner `A` erreicht haben und bei ihm aufgenommen wurden.

4.6.8 Middleboxen als Störfaktoren bei MPTCP

Was ist eine Middlebox?

MPTCP ist ein Transportprotokoll und operiert innerhalb der Transportschicht (Transport Layer). In Netzwerken werden aber – irgendwo unterwegs, in der Mitte (Middle), zwischen kommunizierenden Rechnern – verschiedene Komponenten (quasi als *Bo-*

xen) mit dem Ziel installiert, besondere und nicht standardmäßige Funktionen zu erbringen. Allgemein werden solche Komponenten, wie z.B. verschiedene Arten von Proxies und Firewalls, als *Middleboxen* bezeichnet [RFC 3234].

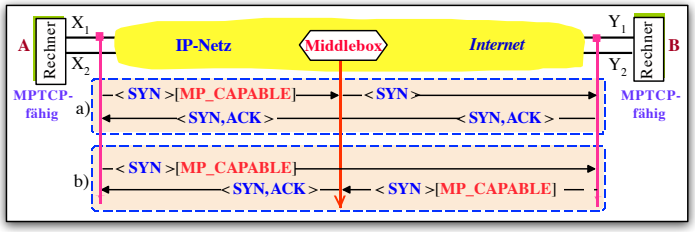


Abb. 4.6-10: Middleboxen als Störfaktoren bei MPTCP: Die MPTCP-Fähigkeit wird nicht erkannt: a) des Quellrechners A, b) des Zielrechners B

Eine Middlebox kann innerhalb von nur einer Schicht oder aber auch gleichzeitig innerhalb mehrerer Schichten agieren, beispielsweise innerhalb der Netzwerk-, Transport- und Applikationsschicht. Dabei interpretiert eine Middlebox die Angaben im Header des Protokolls der entsprechenden Schicht. Folglich kann sie auch einige Angaben im Header ändern. Eine innerhalb der Transportschicht agierende Middlebox kann also die Angaben im TCP-Header ändern und demzufolge die Funktion von MPTCP negativ beeinflussen, einschränken etc. Middleboxen gelten daher als Störfaktoren bei MPTCP. Abb. 4.6-10 soll dies näher zum Ausdruck bringen.

Middlebox-Störung

> Hier wurde angenommen, dass die beiden Rechner MPTCP-fähig sind und dass die Middlebox die ihr unbekannten TCP-Optionen aus dem TCP-Header entfernt. Dies kann u.a. dazu führen, dass die MPTCP-Fähigkeit eines Rechners nicht erkannt wird; z.B. die des Quellrechners, falls die Middlebox `MP_CAPABLE` in `<SYN>` entfernt, oder die des Zielrechners, falls `MP_CAPABLE` in `<SYN,ACK>` entfernt wird.

4.7 Konzept und Einsatz von SCTP

Werden IP-Netze auch für die Sprachübermittlung genutzt, so müssen sie mit den öffentlichen TK-Netzen (wie z.B. mit ISDN, Mobilfunknetzen GSM und UMTS) entsprechend integriert werden. Hierfür müssen u.a. die Nachrichten des *Signalisierungssystems Nr. 7* (SS7) über IP-Netze transportiert werden [Bad22]. Diese Integration stellt besondere Anforderungen an das Transportprotokoll innerhalb der Protokollfamilie TCP/IP. TCP und UDP können diese Anforderungen nicht vollständig erfüllen.

Zweck von SCTP

Daher wurde u.a. das Transportprotokoll SCTP (*Stream Control Transsmission Protocol*) entwickelt und in RFC 4960 spezifiziert. SCTP ermöglicht eine zuverlässige Übertragung von Nachrichten in mehreren unabhängigen *SCTP-Streams*. Ein Stream kann als eine unidirektionale, virtuelle Verbindung interpretiert werden. Jeder Stream kann hierbei unterschiedliche Segmentgrößen unterstützen, um somit z.B. Bytestrom- oder auch Nachrichten-orientierte Übertragung optimal zu bedienen. Diese Eigen-

schaft macht SCTP besonders geeignet für einen Mix unterschiedlicher Datenströme für eine Verbindung, wie dies bei WebRTC (*Web Real Time Connection*) der Fall ist.

4.7.1 SCTP versus UDP und TCP

SCTP wurde entwickelt, um einige Schwächen der beiden klassischen Transportprotokolle UDP und TCP auszugleichen. Auf diese Schwächen wird nun kurz eingegangen.

Schwächen vom UDP

UDP ist ein Protokoll, das einen schnellen, verbindungslosen Dienst zur Verfügung stellt. Dadurch ist es zwar für die Übertragung einzelner Nachrichten geeignet, die empfindlich gegenüber Verzögerungen sind; es bietet jedoch keinen zuverlässigen Transportdienst. Sicherung gegen Übertragungsfehler wie das Erkennen duplizierter Nachrichten, das wiederholte Übertragen verloren gegangener Nachrichten, Reihenfolgesicherung und Ähnliches, müssen durch die jeweilige UDP-Anwendung erfolgen.

Schwächen vom TCP

TCP realisiert sowohl eine Fehlersicherung als auch eine Flusssteuerung, aber es hat auch eine Reihe von Nachteilen. TCP ist *Bytestrom-orientiert*, sodass die einzelnen zu sendenden Byte nummeriert werden. TCP ist nicht effektiv bei der Übermittlung einer Folge von zusammenhängenden Nachrichten. Bei TCP werden alle Nachrichten als Strom von Byte gesehen und die einzelnen Byte fortlaufend nummeriert [Abb. 4.3-7]. Sollte eine Nachricht während der Übertragung verfälscht werden, so ist es bei TCP nicht möglich, nur diese einzige Nachricht wiederholt zu übermitteln. Außerdem macht TCP eine strikte Sicherung der Reihenfolge von Datenbyte. Viele Anwendungen erfordern jedoch lediglich eine teilweise Sicherung der Reihenfolge von Nachrichten. Durch die Sicherung der Reihenfolge bei TCP kann eine unnötige Blockierung bereits angekommener TCP-Pakete durch fehlende Teile von Nachrichten anderer Prozesse oder Transaktionen auftreten.

Was bringt SCTP?

SCTP ist ein verbindungsorientiertes und *nachrichtenbasiertes* Protokoll, das eine zuverlässige Übermittlung von Nachrichten in mehreren unabhängigen SCTP-Streams bietet. Innerhalb einer *SCTP-Assoziation* [Abb. 4.7-2], die in etwa einer TCP-Verbindung entspricht, findet eine TCP-ähnliche Flusssteuerung statt. SCTP kann sowohl Nachrichten segmentieren als auch mehrere Nachrichten in den SCTP-Paketen transportieren.

SCTP vereint die Vorteile von UDP und TCP

SCTP versucht, die Vorteile von UDP und TCP in Bezug auf die Übermittlung von Nachrichtenströmen zu vereinen. SCTP erweitert einerseits den UDP-Dienst um Fehlersicherung und Multiplexing und realisiert andererseits TCP-Konzepte. Somit eignet sich SCTP nicht nur zum Transport von Nachrichtenströmen, sondern kann sich neben UDP und TCP als ein drittes, wichtiges Transportprotokoll für den Transport verschiedener Datenströme etablieren.

4.7.2 SCTP-Assoziationen

Was ist eine SCTP-Assoziation?

SCTP ist ein verbindungsorientiertes Transportprotokoll, nach dem eine SCTP-Assoziation zwischen zwei SCTP-Endpunkten aufgebaut wird. Abb. 4.7-1 illustriert eine SCTP-Assoziation. Sie ist als eine Vereinbarung zwischen zwei SCTP-Endpunkten in Bezug auf den Verlauf der Kommunikation zwischen ihnen zu verstehen. Einen SCTP-Endpunkt stellt das folgende Paar dar:

```
SCTP-Endpunkt = (IP-Adresse, SCTP-Portnummer)
```

Ein SCTP-Endpunkt kann auch als ein (*SCTP-*)Socket betrachtet werden. Im Allgemeinen kann ein Endsystem mit SCTP mehrere IP-Adressen besitzen, d.h. es kann ein Multihomed-IP-Endsystem sein.

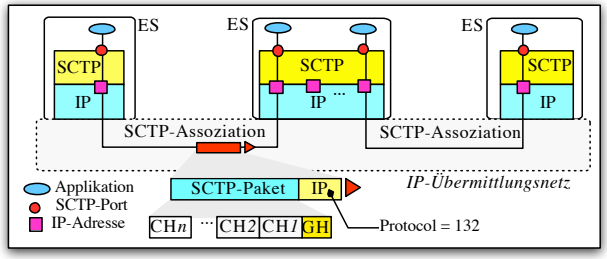


Abb. 4.7-1: Veranschaulichung von SCTP-Assozationen
ES: Endsystem, CH n: Chunk n; GH: Gemeinsamer Header

Was ist ein Stream?

Eine SCTP-Assoziation kann als virtuelle SCTP-Verbindung angesehen werden, die sich aus einer Vielzahl von *SCTP-Streams* zusammensetzen kann. Ein Stream kann wiederum als eine unidirektionale, (gerichtete) virtuelle Verbindung interpretiert werden [Abb. 4.7-2], auf der entweder Nachrichten fester Größe oder Datenströme (unbekannter Länge) transportiert werden können.

Mehrere Nachrichten in einem SCTP-Paket

Die Nachrichten und Bytesströme werden in SCTP-Paketen transportiert [Abb. 4.7-3]. Die Nummer des Protokolls SCTP im IP-Header ist 132. Wie aus Abb. 4.7-1 ersichtlich ist, setzt sich ein SCTP-Paket aus einem gemeinsamen Header und einer Reihe von *Chunks* zusammen. Ein Chunk stellt eine Art Container dar und kann eine Signalisierungsnachricht, normale Daten bzw. bestimmte Steuerungsangaben enthalten. In einem SCTP-Paket können somit mehrere Nachrichten bzw. mehrere Datenblöcke aus den unterschiedlichen Datenströmen transportiert werden.

Über eine SCTP-Assoziation können parallel mehrere SCTP-Streams übermittelt werden. Dies veranschaulicht Abb. 4.7-2. Ein Stream kann eine Folge von zusammenhängenden Nachrichten in eine Richtung darstellen.

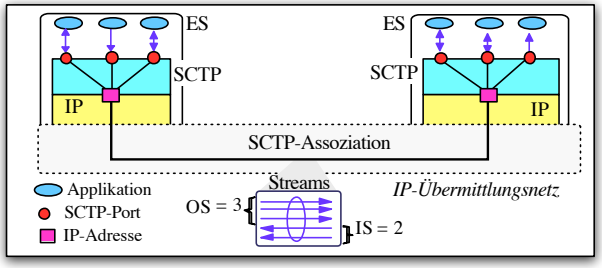


Abb. 4.7-2: Mehrere Streams innerhalb einer SCTP-Assoziation
ES: Endsystem, IS: Inbound Stream , OS: Outbound Stream

Arten von Streams

Man unterscheidet zwischen ausgehenden Streams (*Outbound Streams*) und ankommenden Streams (*Inbound Streams*). Beim Aufbau einer Assoziation gibt die initiie-

rende SCTP-Instanz die Anzahl von Outbound Streams als Parameter OS (Number of Outbound Streams) und die zulässige Anzahl von Inbound Streams als Parameter `MIS` (*Maximum of Inbound Streams*) in der von ihr initiierten Assoziation an.

SCTP-Verbindung als virtuelle Autobahn

Da mehrere Streams innerhalb einer SCTP-Assoziation verlaufen, müssen die einzelnen Streams entsprechend gekennzeichnet werden. Hierfür dient der Parameter *Stream Identifier*. Anders als bei einer TCP-Verbindung (vgl. Abb. 4.3-4), die als virtuelle Straße mit zwei entgegen gerichteten Spuren interpretiert werden kann, kann man sich eine SCTP-Verbindung als virtuelle Autobahn mit einer beliebigen Anzahl von Spuren in beiden Richtungen vorstellen.

Die Nachrichten (bzw. andere Daten) werden in *DATA-Chunks* transportiert [Abb. 4.7-5]. Mit dem Parameter *Stream Identifier* im Chunk DATA wird markiert, zu welchem Stream die übertragene Nachricht gehört. Damit ist es möglich, in einem SCTP-Paket mehrere Chunks `DATA` mit den Nachrichten aus verschiedenen Streams zu übermitteln. Dies bezeichnet man beim SCTP als *Chunk Bundling* (*Chunk-Bündelung*).

4.7.3 Struktur der SCTP-Pakete

Wie Abb. 4.7-3 illustriert, besteht ein SCTP-Paket aus einem gemeinsamen Header (*Common Header*) und einer Reihe von festgelegten Chunks [Tab. 4.7-1]. Ein Chunk kann als Container für Nachrichten, Daten bzw. SCTP-Steuerungsangaben angesehen werden. Die Anzahl von Chunks und deren Reihenfolge im SCTP-Paket ist nicht festgelegt.

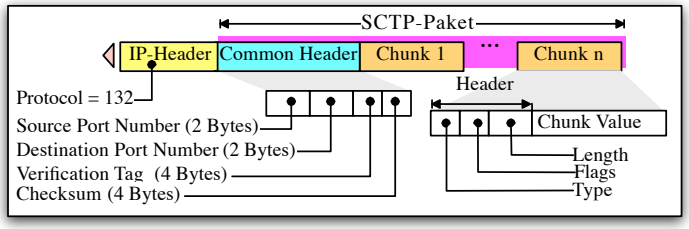


Abb. 4.7-3: Struktur der SCTP-Pakete

Die Angaben im *Common Header* beinhalten folgende Informationen:

- `Source Port Number`: Angabe des SCTP-Quellports.
  Die Nummer des Quellports und die IP-Quelladresse im IP-Header stellen den ersten Endpunkt der SCTP-Assoziation dar.
- `Destination Port Number`: Angabe des SCTP-Zielports.
  Die Nummer des Zielports und die IP-Zieladresse stellen den zweiten Endpunkt der SCTP-Assoziation dar.
- *Verification Tag* (`Veri-Tag`): `Veri-Tag` stellt eine Zufallszahl dar, die dem Empfänger eines SCTP-Pakets die Prüfung ermöglicht, ob das empfangene Paket zur aktuellen Assoziation gehört.
- *Checksum*: Die Prüfsumme dient zur Entdeckung von Bitfehlern im SCTP-Paket.

SCTP-Chunks

Jedes Chunk setzt sich aus einem Header und dem Inhalt zusammen. Die Flags im Header stellen die Bit dar, deren Werte vom Chunk-Typ abhängig sind. Im Feld *Chunk Type* wird die Chunk-ID (Identifikation) angegeben, d.h. welcher Inhalt (Nachrichten/Daten bzw. Steuerungsangaben) das betreffende Chunk hat. Eine Auflistung von ausgewählten Chunks zeigt Tab. 4.7-1.

Chunk-ID	Bedeutung
0	Payload Data (`DATA`)
1	Initiation (`INIT`)
2	Initiation Acknowledgement (`INIT ACK`)
3	Selective Acknowledgement (`SACK`)
7	Shutdown (`SHUTDOWN`)
8	Shutdown Acknowledgement (`SHUTDOWN ACK`)
10	State Cookie (`COOKIE ECHO`)
11	Cookie Acknowledgement (`COOKIE ACK`)
14	Shutdown Complete (`SHUTDOWN COMPLETE`)

Tab. 4.7-1: Chunk-Typen und ihre IDs

Es gibt zwei Klassen von Chunks. Zu der ersten Klasse gehört das Chunk `DATA` (`Chunk-ID = 0`). Dieses Chunk kann als Container interpretiert werden, in dem sowohl die normalen Daten als auch verschiedene Nachrichten transportiert werden [Abb. 4.7-5]. Zur zweiten Klasse gehören die restlichen Chunks, die zur Realisierung von SCTP-Funktionen dienen, d.h. sie enthalten bestimmte Steuerungsangaben und werden im Weiteren als *Kontrollchunks* bezeichnet.

4.7.4 Aufbau und Abbau einer SCTP-Assoziation

Den Aufbau und den Abbau einer SCTP-Assoziation zeigt Abb. 4.7-4. Eine Assoziation wird hier vom SCTP-Endpunkt A initiiert. Dies erfolgt durch das Absenden eines Chunk `INIT`, in dem Folgende Parameter enthalten sind:

- `Initiate Tag` (`I-Tag`),
- `Advertised Receiver Window Credit` (`a_rwnd`),
- `Number of Outbound Streams` (`OS`),
- `Maximum Number of Inbound Streams` (`MIS`),
- `Initial TSN` (`I-TSN`, *Transmission Sequence Number*).

Der Parameter `I-Tag` ist eine Zufallsvariable aus dem Bereich zwischen 1 und 4294967295. Mit dem Parameter `a_rwnd` wird der Gegenseite die Größe des reservierten Speichers in Byte für die Zwischenspeicherung von ankommenden Daten mitgeteilt. Die Anzahl von ausgehenden Streams innerhalb der Assoziation wird mit `OS` angegeben. Die zulässige Anzahl von eingehenden Streams gibt `MIS` an. Gesendete Chunks `DATA` mit Datensegmenten bzw. anderen Nachrichten werden während einer Assoziation fortlaufend nummeriert. Hierfür wird `TSN` im `DATA-DATA` verwendet. Daher gibt jede Seite beim Aufbau einer Assoziation an, mit welcher Nummer (d.h. `I-TSN`) sie die Nummerierung von `DATA-Chunks` beginnt.

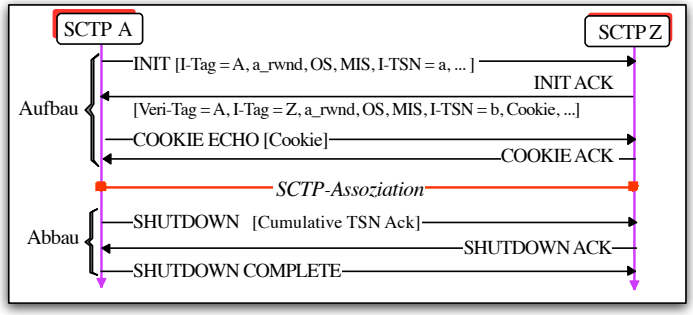


Abb. 4.7-4: Aufbau und Abbau einer SCTP-Assoziation

Das empfangene Chunk `INIT` wird mit dem Chunk `INIT ACK`, in dem die Parameter `I-Tag`, `a_rwnd`, `OS`, `MIS`, `I-TSN` und *Cookie* enthalten sind, bestätigt. Im `INIT ACK` wird der Wert von `I-Tag` aus `INIT` als `Veri-Tag` im SCTP-Header übermittelt. Dadurch wird sichergestellt, dass `INIT` und `INIT ACK` zur gleichen Assoziation gehören.

Cookie gegen DoS-Angriffe

Um sich gegen DoS-Angriffe (*Denial of Service*) zu wehren, wird das Cookie-Konzept verwendet. Ein Cookie stellt den Wert einer Hashfunktion dar und muss für eine Assoziation eindeutig sein. RFC 2522 schlägt vor, wie ein Cookie zu berechnen ist. In `INIT ACK` wird ein Cookie übermittelt. Das empfangene Cookie in `INIT ACK` wird im `COOKIE ECHO` zurückgeschickt. Auf diese Art und Weise erfolgt eine gegenseitige Authentisierung des Kommunikationspartners. `COOKIE ECHO` wird mit `COOKIE ACK` bestätigt. Mit dem Empfang von `COOKIE ACK` wird der Aufbau einer SCTP-Assoziation beendet. Nun kann ein Austausch von Datensegmenten bzw. Nachrichten zwischen den beiden Kommunikationspartnern erfolgen.

Abbau einer SCTP-Assoziation

Abb.4.7-4 zeigt auch einen normalen Abbau einer SCTP-Assoziation. Der Abbau wird mit dem Chunk `SHUTDOWN` initiiert und von der Gegenseite mit `SHUTDOWN ACK` bestätigt. Der Empfang von `SHUTDOWN ACK` wird anschließend mit `SHUTDOWN COMPLETE` quittiert.

4.7.5 Daten- und Nachrichtenübermittlung nach SCTP

DATA-Chunk

Die Daten bzw. Nachrichten werden in `DATA-Chunk` transportiert, die als Container dienen. Abb. 4.7-5 zeigt, welche Angaben im Header eines `DATA-Chunk` enthalten sind.

Die innerhalb einer Assoziation gesendeten Chunks werden fortlaufend nummeriert und die Nummer wird als `TSN` geführt. Die Zugehörigkeit eines `DATA-Chunks` zum Stream wird mit dem Parameter `S` (*Stream Identifier*) markiert. Die zu einem Stream gehörenden *DATA-Chunks* werden ebenfalls fortlaufend mit dem Parameter `n` (*Stream Sequence Nr*) nummeriert. Der Teil `User Data` kann eine ganze Nachricht, einen Teil einer Nachricht oder normale Daten darstellen.

In einem `DATA-Chunk` können enthalten sein:

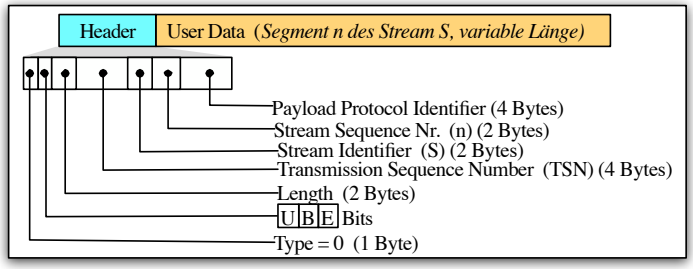


Abb. 4.7-5: Aufbau von DATA-Chunks als Container von Nutzdaten
B (E): Beginning (Ending) fragment bit, U: Unordered Chunk

- Eine vollständige, unnummerierte Nachricht (`U = 1`), indem das Bit `U` auf `1` gesetzt wird. In diesem Fall wird das Feld `Stream Sequence Nr` nicht interpretiert.
- Eine vollständige, nummerierte Nachricht bzw. ein Segment aus einer nummerierten Nachricht (`U = 0`), wobei die beiden Bit `BE` gemäß Tab. 4.7-2 zu interpretieren sind.

B	E	Bedeutung
1	0	Erstes Segment einer fragmentierten Nachricht
0	0	Inneres Segment einer fragmentierten Nachricht
0	1	Letztes Segment einer fragmentierten Nachricht
1	1	Unfragmentierte Nachricht

Tab. 4.7-2: Interpretation von Bit `BE` im Chunk-Header
B: Beginning Chunk, E: Ending Chunk

Das Zusammenspiel von Parameter `n` (*Stream Sequence Nr*), `TSN` und dem `BE` Bit veranschaulicht Abb. 4.7-6.

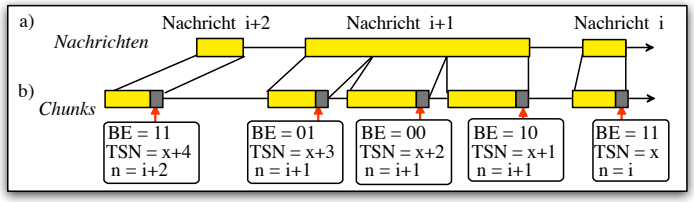


Abb. 4.7-6: Bedeutung des Bit `BE` sowie der Parameter `n` und `TSN`: a) die zu übertragenden Nachrichten, b) Folge von DATA-Chunks

BE-Bit-Nutzung

Es wurde hier angenommen, dass drei Nachrichten `i`, `i+1` und `i+2` als Stream übermittelt werden. Die lange Nachricht `i+1` wurde hier aufgeteilt und in drei `DATA-Chunks` übermittelt. Alle `DATA-Chunks` werden hier fortlaufend mit dem Parameter `TSN` nummeriert. Die Nachricht `i` wird vollständig in einem `DATA-Chunk` transportiert und sein Header enthält:

- `n = i` (Nachricht `i` in einem Stream),
- `TSN = x` (beispielsweise) und
- `BE = 11` (unfragmentierte Nachricht).

Die Nachricht `i+1` wird aufgeteilt und in mehreren `DATA`-Chunks transportiert. In allen Chunks mit den Segmenten dieser Nachricht ist `n = i+1`. Das erste Segment wird mit `BE = 10` markiert. Das letzte Segment enthält `BE = 01`. Die inneren Segmente werden mit `BE = 00` markiert.

Selektive Bestätigung von DATA-Chunks mit SACK und GAB

Zur Übermittlung unstrukturierter Daten und Nachrichten dienen folgende Chunks:

- `DATA` als Container, in dem die Nachrichten eingebettet werden,
- `SACK` (*Selective Acknowledgement*), mit dem die fehlerfrei empfangenen `DATA-Chunks` bestätigt werden. Diese Feld wird im Besonderen zur Mitteilung des *Gap Acknowledge Block* Zählers `GAB` genutzt.

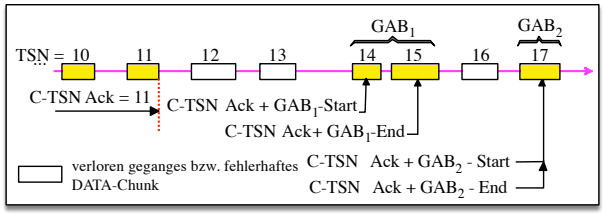


Abb. 4.7-7: Interpretation von Parametern: *C-TSN Ack*, *GAB$_i$-Start* und *GAP$_i$-End*

SACK und GAB Einsatz

Abb. 4.7-7 illustriert das Prinzip der selektiven Bestätigung von empfangenen `DATA-Chunks`. Es wurde hier angenommen, dass die Ziel-SCTP-Instanz die `Data-Chunks` mit den Sequenznummern `TSN = 10`, `TSN = 11`, `TSN = 14`, `TSN = 15` und `TSN = 17` fehlerfrei empfangen hat. Es ist hervorzuheben, dass es sich hierbei um die `DATA-Chunks` einer Assoziation mit mehreren Streams handelt. In der Folge von `DATA-Chunks` sind zwei Lücken (*Gaps*) entstanden. Die erste Lücke ist dadurch aufgetreten, dass die `DATA-Chunks` mit den Sequenznummern `TSN = 12` und `TSN = 13` unterwegs verlorengegangen sind bzw. mit Fehlern empfangen und verworfen wurden. Die zweite Lücke entsteht dadurch, dass das fehlerfreie `DATA-Chunk` mit `TSN = 16` noch nicht vorliegt.

SACK Chunk

Nun soll der Empfang von fehlerfreien Chunks bestätigt werden. Hierfür dient das Kontroll-Chunk `SACK` mit den folgenden Parametern:

- `Cumulative TSN Ack` (*C-TSN Ack*)
  Mit *C-TSN Ack* werden alle bis zur ersten Lücke (Gap) empfangenen `DATA-Chunks` bestätigt. In Abb. 4.7-7 ist `C-TSN Ack = 11`.
- `Number of Gap ACK Blocks` (*Number of GABs*)
  Mit diesem Parameter wird die Anzahl der Blöcke (d.h. der lückenlosen Gruppen) von `DATA-Chunks` angegeben. In Abb. 4.7-7 ist `N = 2`.
- `Gap ACK Block # n Start` (`GAB`$_n$`-Start`)
  Mit diesem Parameter wird der Beginn des n-ten Blocks (d.h. der n-ten lückenlosen Gruppe) von `DATA-Chunks` folgendermaßen bestimmt:
  `Beginn des n-ten Blocks = C-TSN Ack + GAB`$_n$`-Start`

- `Gap ACK Block # n Stop` (GAB_n-Stop)
  Dieser Parameter bestimmt das Ende des n-ten Blocks von `DATA-Chunks`:
  `Ende des n-ten Blocks = C-TSN Ack + GAB`$_n$`-Stop`
- `Number of Duplicate TSNs (X)`
  Mit diesem Parameter wird die Anzahl der Duplikate, d.h. von Chunks `DATA` mit der gleichen Sequenznummer `TSN`, die z.B. wiederholt übermittelt wurden, angegeben. In Abb. 4.7-7 ist X = 0.

Fehlerfreie SCTP-Übermittlung

Bevor Daten bzw. Nachrichten gesendet werden, muss zuerst eine SCTP-Assoziation aufgebaut werden [Abb.4.7-4]. Den SCTP-Verlauf bei einer fehlerfreien Übermittlung von Daten illustriert Abb. 4.7-8.

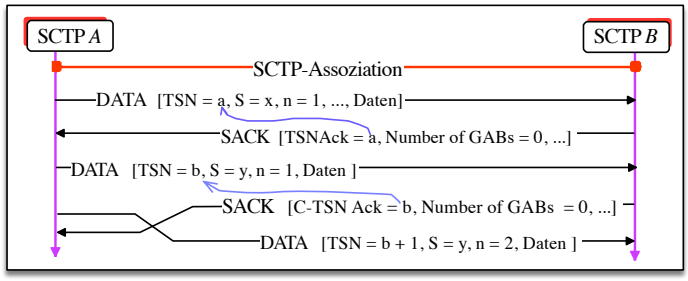


Abb. 4.7-8: Fehlerfreie Übermittlung von Daten nach SCTP
GAB: Gap Ack Block [Abb. 4.7-7], n: Stream Sequence Number, S: Stream Identifier, TSN: Transmission Sequence Number, C-TSN Ack: Cumulative TSN Acknowledgement

Fehlerfreie SCTP-Übermittlung

Zunächst sendet der SCTP-Endpunkt *A* ein `Chunk DATA` mit der Sendefolgenummer `TSN = a`. Innerhalb der Assoziation gehört dieses Chunk zu dem Stream `x` (`S = x`), und dessen Folgenummer im Stream `x` ist `1` (`n = 1`). Dieses Chunk `DATA` bestätigt der SCTP-Endpunkt *B* mit `SACK`, in dem u.a. angegeben wird:

- `TSN Ack = a`: Die fehlerfreie Empfang von `DATA` mit `TSN = a` wird bestätigt.
- `Number of GABs = 0`: Damit wird mitgeteilt, dass es keine Lücke in der empfangenen Folge von `DATA-Chunks` gibt (Es handelt sich hierbei nur um ein Chunk `DATA`).

Daraufhin sendet der SCTP-Endpunkt *B* die zwei Chunks `DATA` innerhalb des Streams `y` (`S = y`). Diese Chunks haben die Sendefolgenummern `TSN = b` und `TSN = b+1` innerhalb der Assoziation und die Sequenznummern im Stream `y` entsprechend `n = 1` und `n = 2`. Das erste Chunk `DATA` wurde vom SCTP-Endpunkt *A* bereits mit `SACK` bestätigt. Der Empfang des zweiten Chunk `DATA` bleibt vom SCTP-Endpunkt *B* hier noch unbestätigt.

Bei TCP enthalten die TCP-Pakete mit den Daten auch die Quittungen als `Acknowledgement Number`. Im Gegensatz zum TCP werden in den *DATA-Chunks* keine Quittungen übermittelt. Für die Quittungen werden die Chunks `SACK` verwendet.

Fehlerhafte SCTP-Übermittlung

Den SCTP-Verlauf bei einer fehlerhaften Übermittlung von Daten zeigt Abb. 4.7-9.

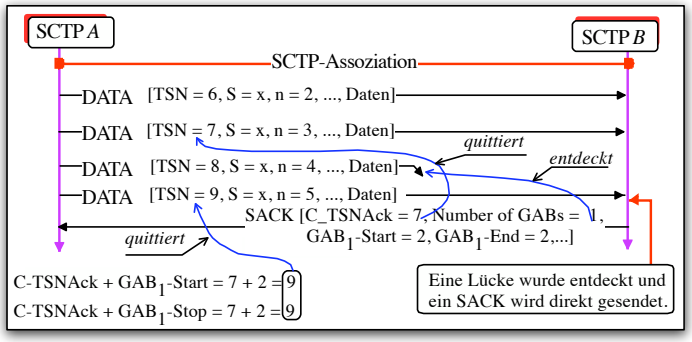


Abb. 4.7-9: Fehlerhafte Übermittlung von Daten nach SCTP
GABn-Start: Gap ACK Block # n Start, GABn-End: Gap ACK Block # n End, Weitere Abkürzungen wie in Abb. 4.7-8

Fehlerbehaftete SCTP-Übermittlung

Der SCTP-Endpunkt *A* sendet hier vier Nachrichten in den `DATA-Chunks` mit den Sendefolgenummern `TSN` von 6 bis 9 innerhalb einer Assoziation. Diese Nachrichten gehören zum Stream `x` und haben innerhalb dieses Streams die Nummern 2, 3, 4 und 5. Das dritte Chunk `DATA` ist unterwegs verloren gegangen. Nach Eintreffen des darauf folgenden Chunk `DATA` entdeckt die Empfangsseite die Lücke (*Gap*) in der empfangenen Folge von `DATA-Chunks`. Als Reaktion darauf sendet sie unmittelbar die Quittung als Chunk `SACK` mit den Angaben:

- `C-TSN Ack = 7`: Damit wird der fehlerfreie Empfang von `DATA`-Chunks bis zur Nummer `TSN = 7` einschließlich bestätigt.
- `Number of GABs = 1`: Damit wird mitgeteilt, dass es eine Lücke in der empfangenen Folge von `DATA`-Chunks gibt (Chunk `DATA` mit `TSN = 8` fehlt).
- $\texttt{GAB}_1\texttt{-Start = 2}$: Mit dieser Angabe wird der Beginn des 1-ten Blocks von `DATA`-Chunks nach der 1-ten Lücke bestimmt [Abb. 4.7-7].
- $\texttt{GAB}_1\texttt{-End = 2}$: Damit wird das Ende des 1-ten Blocks von `DATA`-Chunks nach der 1-ten Lücke bestimmt.

Da der 1-te Block von `DATA`-Chunks nach der 1-ten Lücke nur das Chunk mit `TSN = 9` beinhaltet, wird mit `SACK` auch dieses Chunk bestätigt (quittiert). Das verloren gegangene Chunk mit `TSN = 8` muss vom SCTP-Endpunkt *A* wiederholt übertragen werden.

Für detaillierte Informationen über das SCTP ist insbesondere auf die Ergebnisse der IETF-Arbeitsgruppe `sigtran` (Signalling Transport) zu verweisen.

4.8 Das QUIC-Protokoll

Analysiert man den Datenverkehr im Internet, so kommt man zu dem Schluss, dass der Datenverkehr über die Ports 443 und 80 (`https` und `http`) den dominierenden Traffic im Internet darstellt. Dies ist auch kaum verwunderlich, da die meisten Applikationen

Web-basiert sind, einschließlich geschäftlicher Transaktionen per SOAP-Nachrichten (*Simple Object Access Protocol*).

4.8.1 Ziele von QUIC

Content Delivery Networks

Der Web-Datenverkehr vollzieht sich heute vielfach über ein *Content Delivery Network* (CDN), bei der die Webseite, die man aufruft, Links zu vielen anderen Seiten aufweist, um z.B. Schrift-Fonts, JavaScript oder weiteren Content (kaskadenförmig) nachzuladen. Dies ist für den Nutzer häufig nicht nachvollziehbar; außer über die langen Ladezeiten in seinem Webbrowser und damit eine reduzierte '*User Experience*' (UX). Der Grund liegt darin, dass jede externe Ressource eine neue TCP-Verbindung (einschließlich Handshake) benötigt. Diese kann nicht immer parallelisiert werden, sondern die Webseite wird im Grunde genommen wie ein logischer Baum aufgebaut.

Roaming Users

Dies gilt um so mehr, je weniger performant die eigene Netzwerkverbindung ist, deren Qualität beim Einsatz mobiler Geräte wie Smartphones nicht abgeschätzt werden kann. Zudem wechselt der mobile Benutzer häufig zwischen Netzwerken bzw. Access-Punkten (vgl. Abschnitt 13.3) mit der Folge unterschiedlicher Übertragungscharakteristika und IP-Adressen im Laufe einer 'Verbindung'.

TLS 1.3 Verschlüsselung

Ein dritter Punkt kommt hinzu: Smartphones und der hierdurch verursachte Internetverkehr, der mittlerweile mehr als 50 % ausmacht[11], werden nicht nur für das Surfen auf Webseiten und Telefonieren (mittlerweile auch über VoIP; siehe Abschnitt 7.3 und Abschnitt 7.4), sondern auch für geschäftliche Vorgänge, wie z.B. Banküberweisungen genutzt. Smartphones fungieren zudem als 'Trust-Anchor' für die Multifaktor- bzw. Multikanal-Authentisierung. Der Absicherung der Datenübertragung ist Google bei QUIC umfassend entgegengekommen, indem moderne Verschlüsselungstechnologien auf Grundlage der ECC-Kryptographie [Abschnitt 2.6] hierin Einzug gefunden haben.

Quick UDP Internet Connections

Die Probleme hat Google mit dem QUIC-Protokoll an der Wurzel angepackt. Aus den eigenen Erfahrungen mit dem Abhören des Datenverkehrs durch die NSA wurde ein Protokoll geschaffen, das versucht, diese unterschiedliche Anforderungen unter einen Hut zu bringen, wobei auf bewährte Technologien, wie das TLS-Protokoll 1.3 (Abschnitt 7.2) und natürlich die Erfahrungen der bestehenden Netzwerkprotokolle UDP und TCP zurückgegriffen wurde. Dies wurde mittlerweile in den Internetstandards RFC 8999, RFC 9000, RFC 9001 und RFC 9002 spezifiziert [Abb. 4.8-1]. Derzeit fungieren allerdings nur die Applikationsprotokolle HTTP/3 sowie DNS/oQuic als konkrete Implementierungen für QUIC.

Bei der Spezifikation hat Google jedoch das gesamte ISO/OSI- und auch TCP/IP-Kommunikationsmodell über Bord geworfen und die hier in den oberen vier (bzw. zwei) Schichten verteilten Aufgaben für die Netzwerkkommunikation quasi in der Schicht 4 – dem Transportlayer in Form von UDP – integriert, das auf den bestehenden

[11] Eine Übersicht liefert z.B. https://www.statista.com/markets/424/topic/538/mobile-internet-apps/

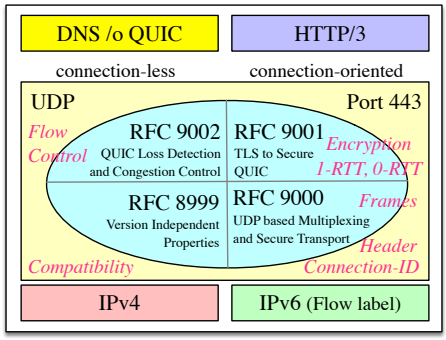


Abb. 4.8-1: Definitionsbestandteile des QUIC-Protokolls

IPv4- und IPv6-Strukturen aufbaut. Deren Dienste werden weiterhin benötigt, um das Routing in den IP-Netzen zu gewährleisten.

QUIC im User Space

Während TCP/IP als Teil des Kernels der Betriebssysteme fungiert (dies gilt sowohl für Unix/Linux als auch Windows) und auch die Tendenz besteht, Verschlüsselungsroutinen auch vom Betriebssystem zu realisieren, um sie performanter zu machen, ist QUIC eine sogenannte User-Space-Implementierung. D.h., der Webbrowser, der QUIC benutzt, nutzt die QUIC-Libraries auf der Benutzerebene, während die IP/UDP-Datagramme vom Betriebssystem über die Netzwerkkarte verarbeitet werden. Gleiches gilt auf der Serverseite: Webserver wie Apache oder Nginx müssen mit den QUIC-Bibliotheken gebunden werden, um diesen Dienste bereitzustellen.

Wir wollen hier den Versuch unternehmen, dieses noch recht frische Kommunikationsprotokoll nicht in Form seiner abstrakten Dienste, sondern auf Grundlage der übertragenen Informationen für den Kommunikationspartner vorzustellen.

4.8.2 QUIC-Pakete in UDP und Transport über IP-Netze

Mit der Einführung von IPv6 wurde die Paketgröße für UDP-Nachrichten von den ursprünglichen 512 Byte of 1280 Byte hochgeschraubt: Netzwerke dürfen IP-Pakete (einschließlich des IP-Headers) unterhalb dieser Größe nicht fragmentieren, sondern müssen in der Lage sein, diese unversehrt zu transportieren.

Diese Eigenschaft wird vom QUIC-Protokoll vorausgesetzt (und ist auch mittlerweile gängige Praxis in anderen Protokollen, die auf UDP aufsetzen). Somit liegt der Payload for UDP bei 1232 Byte auf IPv4-Netzen und 1252 Byte über IPv6 (mit Standardheader). QUIC limitiert allerdings von vornherein die Datagrammgröße auf 1200 Byte, was noch etwas Luft für IPv6 Extension Header lässt.

PMTUD

Zur Unterstützung der *Path Maximum Transmission Unit Discovery* wird das erste QUIC-Datenpaket für eine Verbindung auf die erlaubte maximale Größe von 1200 Byte expandiert. Wir erinnern uns daran, dass die MAC-Frames bei Ethernet bis zu 1480 Byte Payload aufweisen können, bei Jumbo-Frames, die auch von IPv6 unterstützt werden, deutlich mehr. QUIC-Implementierungen können dies ausnutzen und die MTU-Größe entsprechend 'nach oben' anpassen. Hier gilt es aber, ICMP-

Datenpakete auszuwerten, die z.B. für IPv6 '*Packet Too Big*' (PTB) signalisieren können.

QUIC kultiviert die *Babuschka-Methode*: Der eigentliche Payload kann erst nach umfangreichen Decodierungs-Runden entnommen werden [Abb. 4.8-2].

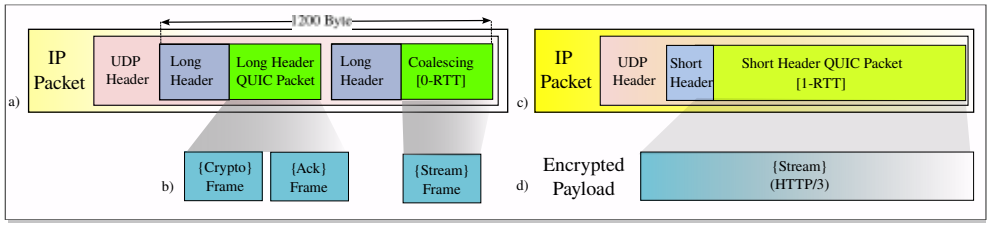


Abb. 4.8-2: Enkapsulierung beim QUIC-Protokoll: a) 'Long Header Packets' im UDP-Datagramm mit b) unterschiedlichen Frametypen (in geschweiften Klammern), c) UDP-Datagramme 'Short Header Packets', die ihrerseits d) die verschlüsselten Daten in Stream-Frames enthalten (entsprechend Tab. 4.8-1)

1. Auf der Netzwerkschicht kann IPv4 oder IPv6 eingesetzt werden.

Coalescing Packets

2. Als Transportprotokoll wird das verbindungslose UDP genutzt, wobei hier ausschließlich von Port 443 (also HTTPS) ausgegangen wird. Mehrere QUIC-Pakete können hier u.U. untergebracht werden, falls sie aus einem zusammengehörigen Datenstrom stammen: *Coalescing Packets*[12]. Dies erfordert notwendigerweise die Kenntnis der Paketlänge des ersten Paketes, was daher die *Short Header*-Pakete hier ausschließt. Des Weiteren ist dieser Mechanismus für Pakete vom Typ *Version Negotiation* und *Retry* untersagt.
3. Das QUIC-Protokoll kennt Pakete mit 'langem' und 'kurzem' Paket-Header:

Verbindungsaufbau

   - Pakete mit 'langem' Header werden zur Initialisierung einer Sitzung und für weitere Protokollnachrichten verwendet. Die Endpunkte der Kommunikation werden über eine *Connection ID* identifiziert, die eine Datenstruktur im Hauptspeicher des Rechners beschreibt, in der die Zustände für diese Verbindung, aber auch die zugehörigen *Traffic Secrets* gehalten werden.

Verschlüsselte Nachrichten

   - Pakete mit 'kurzem' Header besagen, dass verschlüsselte Nutzlast auf Grundlage von TLS 1.3 übertragen wird.

Frames

4. Der ursprünglich unverschlüsselte Payload in einem Paket wird als *Frame* bezeichnet. Beim aktuellen Stand der Entwicklung von QUIC sind dies HTTP/3-Daten. An der Unterstützung weiterer Protokolle, wie *DNS over QUIC* und der SIP-Nachfolger *Real Time Internet Peering for Telephony* (RIPT) wird gearbeitet.

4.8.3 Aufbau von QUIC-Nachrichten und der Payload

Während die Protokolle wie IP, TCP und UDP einen einfachen und einheitlichen Nachrichtenkopf vorsehen, ist dieser beim QUIC-Protokoll entsprechend der Aufgabenstellung variabel aufgestellt:

[12] also sinngemäß eine 'Paketkoalition'

- *Long Header Packets*: Diese werden zum Aufbau der Kommunikation eingesetzt und übertragen Informationen für die Peer-Identifizierung, den Sitzungsaufbau sowie die Ablaufsteuerung einschließlich der Einleitung der Verschlüsselung.
- *Short Header Packets*: Diese werden auch `1-RTT` (*Round Trip Time*) Pakete genannt und dienen zur Übertragung des verschlüsselten Payloads.

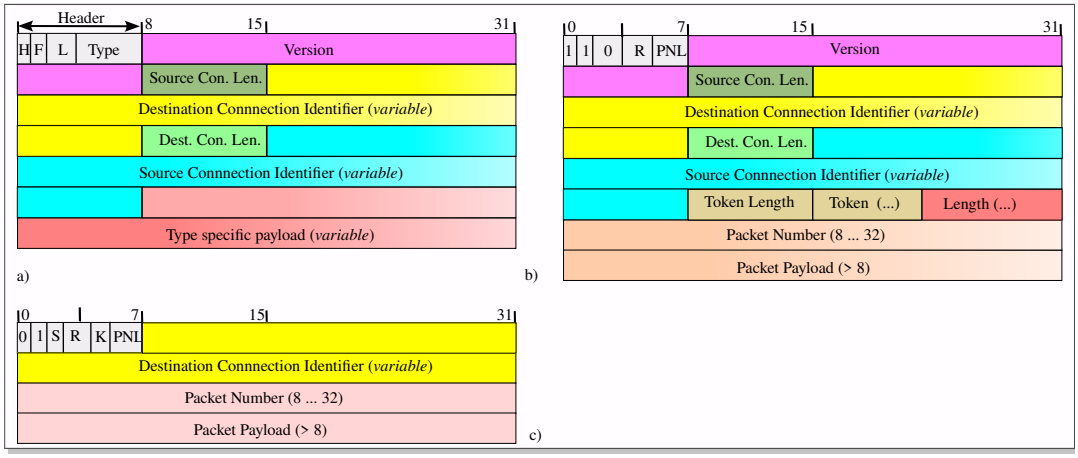


Abb. 4.8-3: Drei Nachrichtenformate bei QUIC: a) Allgemeiner Aufbau des *Long Header Packets*, b) das *Initial Packet*, sowie c) das *Short Package*-Format; H: Header Form, F: Fixed Bit, S: Spin Bit, K: Key Phase, L: Long Packet Type, R: Reserved Bits, PNL: Packet Number Length

Header Form

Das erste Byte im QUIC-Header wird *Header Form* genannt. Aus dem Encoding dieses Feldes ergeben sich die verschiedenen Pakettypen von QUIC, die recht unterschiedlich sind. Im Gegensatz zur typischen Header-Struktur bei den üblichen TCP/IP-Paketen, ist die Länge einiger Felder nicht initial spezifiziert: Deren Werte ergeben sich erst während des Verbindungsaufbaus, liegen also im Speicher der jeweils kommunizierenden Knoten und werden Dritten somit nicht mitgeteilt.

Variable-Length Integer Encoding

Beim Entwurf des QUIC-Protokolls bestand eine weitere Anforderung darin, die zu übertragende Datenmenge zu reduzieren. Dies realisiert QUIC auf der Verbindungsebene[13], indem keine festen Bytelängen für nicht-negative Ganzzahlen genutzt werden, sondern diese werden 'variabel' mit den Längen von 1 (MSB = `00`), 2 (`01`), 4 (`10`) oder 8 (`11`) Bytes dargestellt, wobei die Zahlen in Klammern die *most-significant Bits* zur Bestimmung der Enkodierung bzw. Dekodierung in den entsprechenden 'Klassen' genutzt werden. Dieses Verfahren wird sowohl für die QUIC-Header als auch den Payload (mit Ausnahme des *Frame Type*-Feldes) eingesetzt.

Long Header-Pakete

Bei den *Long Header* QUIC-Paketen lassen sich mittels des `Long Packet Type`-Feldes folgende Pakettypen differenzieren, die den Aufbau und die Steuerung von QUIC-Verbindungen signalisieren:

Packet Type 0x00
- *Initial Packet* sowie *Abandoning Initial Packet*: Bei QUIC wird jeder Endpunkt der Kommunikation mit einem *Connection Identifier* versehen, der eine nichtspezifizierte Länge aufweisen kann. Diese *Connection Identifier* sind praktisch die logischen Adressen bei QUIC-Verbindungen und unabhängig von IP-Adressen und Port-Nummern, wie ansonsten bei TCP/IP. Um *Address Spoofing* zu unterbinden und einen schnelleren (zukünftigen) Verbindungsaufbau zu erzielen, kann der Server an den Client ein Token (`NEW_TOKEN`) übermitteln, das als Verbindungsmerkmal aufzufassen ist. Der Wert des Tokens kann mittels eines *Retry Packets* neu eingestellt werden.

Packet Type 0x01
- *0-RTT Packet*: Dieses wird vom Client an den Server geschickt und beinhaltet im Anschluss an das *Initial Packet* nun erste kryptographisch relevante Daten für den Server.

Packet Type 0x02
- *Handshake Packet*: Dieses wird vom Server zu Client übertragen und beantwortet die kryptographischen Vorgaben des Clients. Hiermit startet die Verbindung im verschlüsselten Modus und beinhaltet entsprechend gesicherte Daten. Aus diesem Grund wird die Paketnummer auf den Wert '0' zurückgesetzt.

Packet Type 0x03
- *Retry Packet*: Das *Retry Packet* kommt immer dann ins Spiel, falls Verbindungsprobleme und Datenverluste bei einem der Kommunikationspartner zu verzeichnen sind. Im Grunde genommen hat es die gleiche Protokollfunktion wie ein *Alert*-Protokoll bei TLS (Abschnitt 7.3), und dessen Inhalt ist daher auch nicht kryptographisch gesichert. Allerdings werden hier die Connection IDs für Validierungszwecke benutzt, und es können *Retry Token* mitgeteilt werden.

IPv6 Flow Label

Das Konzept des Token für eine Verbindung – also einen Ende-zu-Ende-Datenstrom – findet sich auch im Protokoll IPv6 als *Flow Label* [Abb. 8.2-1]. QUIC erwartet, dass dieses beim Transport über IPv6-Netze von den Endteilnehmern genutzt wird.

Short Header Packet

Packet Type 0x80

Die QUIC-Pakete mit einem *Short Header* und seinem Wert von `0x80` sind für die eigentliche Nutzdatenübertragung vorgesehen. Die hierin eingebetteten Daten liegen verschlüsselt vor, sofern RFC 9001 angewandt wird, wovon im Folgenden ausgegangen wird. Dies setzt somit den Einsatz des TLS-Protokolls in der Version 1.3 voraus. Für die Steuerung des Datenaustauschs ist der Header im *Short Packet* maßgeblich. Hier können zwei interessante Bits genutzt werden:

1. *Key Phase Bit*: Dieses ist im ersten 1-RTT Paket auf '0' gesetzt und für alle weiteren auf '1', wobei natürlich der TLS-Handshake abgeschlossen sein muss, und das Schlüsselmaterial liegt vor. QUIC lässt es nun zu, dass während der Datenübertragung zwischen den Endknoten auf neue Schlüssel (*Traffic Secret*) umgeschaltet wird, indem dieses Bit wieder auf '0' gesetzt wird. Wie in Abschnitt 7.3 erläutert, können nun die beiden Beteiligten den nachfolgenden Schlüsselwert generieren und einsetzen.
2. *Spin Bit*: 'Geschwindigkeit' ist bei QUIC ein Muss, und sie sollte auch gemessen werden können, wozu das *Spin Bit* als optionales Feature dient: Wird beim Client das Spin Bit gesetzt, beantwortet der Server dies mit ebenfalls gesetztem

[13] im OSI-Siebenschichtenmodell wäre dies Aufgabe der 'Präsentationsschicht'

Bit. Anschließend setzt der Client dies wieder zurück, was der Hin- und Rücklaufzeit entspricht, also genau 1-RTT. Unabhängig vom (verschlüsselten) Inhalt, kann nun ein externer Monitor dieses 'Spin Toggle' feststellen und hieraus die RTT-Zeit bestimmen.

Diese *Short Packet Header*-Pakete sind somit die QUIC-Container für den verschlüsselten Nachrichteninhalt, den Frames. Jedes Paket wird enummeriert, wobei hier ein Maximalwert von $2^{62} - 1$ zur Verfügung steht (vgl. die *Sequence Number* bei TCP [Abb. 4.3-2]).

Protected Header

Bei der Architektur des QUIC-Protokolls wurde Wert auf *Confidentiality* und *Correctness* der übertragenen Informationen gelegt: Wichtige Teile des QUIC-Paketkopfes sind daher kryptographisch gesichert. Entsprechend dem CAR-Modell [Abb. 2.8-2], geschieht dies durch eine *Längsparität* [Abb. 2.8-1] nach dem *ChaCha-Poly1305*-Verfahren und nicht mehr durch eine Querparität, wie beim FCS.

Paketnummer als NONCE

Durch Hinzufügen der Paketnummer sichert der AEAD-Modus, der auch bei *ChaCha-Poly1305* zum Einsatz kommt (vgl. Abb. 2.4-6), die Einmaligkeit der Verschlüsselung pro Paket. Zudem kann dies auch als Maßnahme gegen *Replay-Attacken* beim ansonsten verbindungslosen Nachrichtenaustausch über UDP betrachtet werden.

Die Frage der *Authentizität* löst das QUIC-Protokoll mit der *Connection ID*, deren Nutzen und Einsatz wir im Anschluss an diesen Abschnitt besprechen möchten.

Frames

Frametypen

Die Frames sind die eigentlichen Datenträger der Information, die entsprechend ihrem Informationsgehalt in unterschiedliche Kategorien eingeteilt werden können: *Frame Types*. Im Groben können wir hier unterscheiden zwischen

- Protokollinformationen wie Verbindungsinitiierung und den TLS-Handshake.
- 0-RTT-Daten mit 'frühen' Verbindungsdaten für den TLS-Handshake.
- 1-RTT-Daten, die im Wesentlichen die Nutzinformation tragen.

Wir sehen, diese Einteilung entspricht weitgehend den Pakettypen. Allerdings ist der Tisch für die Nutzung von Frames deutlich reicher gedeckt, und RFC 9000 (Abschnitt 12) spezifiziert folgende Typen:

Jeder Frametyp besitzt einen angepassten Aufbau, der häufig als erstes Feld einen weiteren `Type` aufweist. Somit besitzt das QUIC-Protokoll auch ein umfangreiches Frame-Inventar sowohl für die Verbindungssteuerung als auch für die Flusskontrolle, die in RFC 9002 diskutiert wird, und deren Nutzung wir uns anschauen wollen.

Multiple Streams

Die Nutzdatenübertragung vollzieht das QUIC-Protokoll im Stream-Modus mit einem *Acknowledgement* der eingetroffenen Daten, vergleichbar also TCP bzw. STCP, wobei hier zwischen den Kommunikationsendpunkten mehrere unterschiedliche Datenströme aufgebaut werden können.

Packet Acknowledgement

Das Acknowledgement bezieht sich aber nicht auf die Anzahl der übertragenen Bytes, sondern auf die der Pakete. Eine weitere Eigenheit des QUIC-Protokolls besteht darin, einen potentiellen Mithorcher nicht wissen zu lassen, wie groß die übertragene Datenmenge ist:

Frametyp	Bezeichnung	Pakettyp	Verwendung
0x00	Padding	IH01	Auffüllen von Long Header Packets
0x01	Ping	IH01	Verbindungstests
0x02, 0x03	Ack	IH_1	Acknowledgements
0x04	Reset_Stream	__01	Rücksetzen des Datenstroms
0x05	Stop_Sending	__01	Verbindungsunterbrechung
0x06	Crypto	IH_1	Übertragung von Crypto-Material
0x07	New_Token	___1	Anforderung neues Token
0x08 - 0x0f	Stream	__01	Nutzdatenübertragung
0x10	Max_Data	__01	Flusskontrolle
0x11	Max_Stream_Data	__01	Flusskontrolle
0x12, 0x13	Max_Streams	__01	Flusskontrolle
0x14	Data_Block	__01	Flusskontrolle
0x15	Stream_Data_Blocked	__01	Flusskontrolle
0x16, 0x17	Streams_Blocked	__01	Flusskontrolle
0x18	New_Connection_Id	__01	Anfordern einer neuen Connection ID
0x19	Retire_Connection_Id	__01	Verwerfen einer alten Connection ID
0x1a	Path_Challenge	__01	UDP Pfad-Ermittlung
0x1b	Path_Response	__1	UDP Pfad-Ermittlung
0x1c, 0x1d	Connection_Close	ih01	Abbau der virtuellen Verbindung
0x1e	Handshake_Done	___1	TLS-Handshake erfolgreich durchgeführt

Tab. 4.8-1: Frametypen (in hexadezimaler Nummerierung) des QUIC-Protokolls; der zugehörige Pakettyp ergibt sich wie folgt: I=Initial, H=Handshake, 0=0-RTT, 1=1-RTT, ih=nur beim `Connection_Close`

- Nur die *Long Header Packets* teilen in der Verbindungsaufnahme mit, wie viele Bytes im QUIC-Paket übertragen werden.
- Bei der laufenden (verschlüsselten) Datenübertragung mittels *Short Header Packets* müssen sich die beteiligten Endknoten merken, wie viele Daten gesendet bzw. empfangen wurden.

4.8.4 QUIC-Verbindungen und Datenströme

Zentraler Anker beim QUIC-Protokoll, ist eine Verbindung, die zwischen zwei Knoten eröffnet, genutzt überwacht, sowie letztlich wieder geschlossen wird, und das Ganze aufbauend auf dem verbindungslosen UDP-Protokoll.

QUIC-Socket

Bei den bisher vorgestellten Protokollen ist der *Socket* ein Endpunkt der Kommunikation. Dieser besteht aus dem Triple `{IP-Adresse,UDP|TCP|SCTP,Port-Nummer}`. Dieses Konzept wird bei QUIC über den Haufen geworfen, da QUIC immer das UDP-Port 443 nutzt. An die Stelle des Sockets, der vom Betriebssystem gemanagt werden muss, und der bei TCP und STCP eine Zustandstabelle umfasst [Abb. 4.3-3], tritt eine Datenstruktur, die sich um das Management von QUIC-Verbindungen, der Verschlüsselung und der Flusskontrolle kümmert, hier als *QUIC-Socket* bezeichnet.

Eine Verbindung bei QUIC wird durch das Tupel `{Source Connection ID,Destination Connection ID}` festgemacht, und es wird eine QUIC-Zustandstabelle hierfür (auf beiden Seiten) im Speicher unterhalten.

Connection IDs

Aus Abb. 4.8-2 können wir entnehmen, dass die `Connection IDs`, die in den QUIC-Paketen übertragen werden, im Klartext vorliegen. Daher schreibt das QUIC-Protokoll vor, dass diese keine auswertbaren Eigenschaften aufweisen, die auf den tatsächlichen Kommunikationsendpunkt schließen lassen: Es sind QUIC-Artefakte.

Der Client wählt sich eine `Connection ID` und übergibt diese dem Server im ersten Long Header Packet (`0x00`), das die Sequenznummer `0` trägt. Um eine neue `Connection ID` bekannt zu geben, kann der Frametype `New_Connection_ID` im laufenden Datenstrom (über Short Header-Pakete) genutzt werden. Die beim QUIC-Protokoll notwendige Verbindungstabelle umfasst somit nicht nur die eigenen `Connection IDs`, sondern auch die des Kommunikationspartners. Nicht mehr aktive `Connection IDs` können aus dieser Tabelle entfernt und diese per Frametype `Retire_Connection_Id` dem Gegenüber mitgeteilt werden.

Destination Connection ID als NONCE

Beim 'Erstkontakt' vom Client mit dem Server wird der Wert der `Destination Connection ID` nicht etwa auf '0' gesetzt, sondern durch einen Zufallswert gebildet, der eine Mindestlänge von 8 Byte aufweisen muss. Dieses wird nun als NONCE beim Server eingesetzt. Somit kennt dieser die `Connection ID` des Clients und erzeugt seine eigene, die im folgenden Long Header-Paket mitgeteilt wird.

Verbindungsaufbau

Neben den Connection IDs umfasst der Verbindungsaufbau von QUIC eine beachtliche Komplexität, da umfangreiches Verbindungs- und Kryptomaterial ausgetauscht werden muss. Zwei Punkte sind hierbei beachtenswert:

Packet Stuffing

1. QUIC nutzt das Konzept, in Long Header Packets mehrere QUIC-Pakete mit Nachrichten in unterschiedlichen Frametypes in einem UDP-Datagramm unterzubringen. Diese werden als coalesced Pakete bezeichnet und beschleunigen den Verbindungsaufbau. Das Konzept hat QUIC dem TLS-Protokoll (vgl. Abb. 7.2-2) entlehnt.

1-RTT und 0-RTT Handshake

2. Während bei TCP das erste Datagramm leer ist und nur zur Signalisierung des Sitzungsaufbaus dient, kennt QUIC das Konzept eines 1-RTT- und 0-RTT-Verbindungsaufbaus. Im letzteren Fall lassen sich schon im ersten UDP-Datagramm – nun als (*coalesced*) Short Header-Paket – Nutzinformationen übertragen, die für die *Session Resumption* genutzt werden können

1-RRT-Verbindungsaufnahme

Der 1-RTT-Verbindungsaufbau stellt den Normalfall dar und wird immer dann genutzt, wenn sich Client und Server noch nicht kennen.

1. Zu Anfang der Kommunikation muss der Server auf dem Port 443 lauschen; eine Verbindung besteht zunächst nicht, wird aber mit dem ersten UDP-Datagramm mit einem (Long Header) `INIT`-Paket (`0x00`) mit Sequenznummer `0` eingeleitet, in das der Client ein *Random* im `Crypto`-Frame einbettet.
2. Die Antwort des Servers besteht aus einem UDP-Datagramm, das mehrere QUIC-Pakete beinhaltet:
   - Ein `INIT`-Paket (`0x00`) mit ebenfalls Sequenznummer `0` und dem *Server Random* im `Crypto`-Frame sowie ein `Ack`-Frame für das empfangene `Crypto`-Frame des Clients.

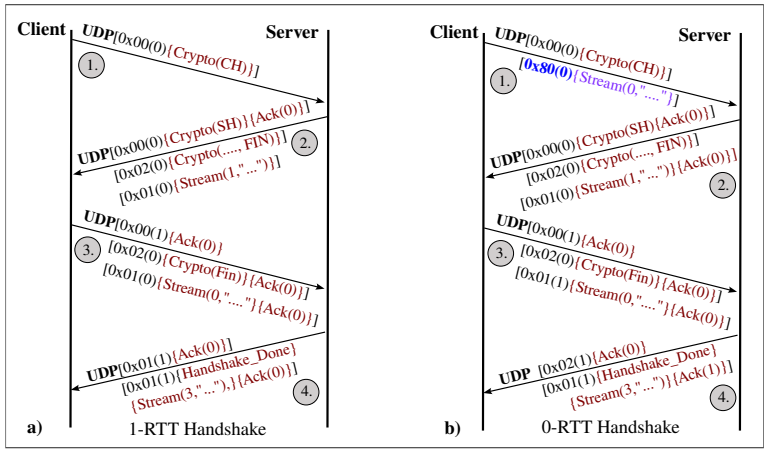


Abb. 4.8-4: Verbindungsaufbau bei QUIC: 1) 1-RTT und b) 0-RTT; QUIC-Pakete sind in rechteckigen Klammern eingeschlossen; QUIC-Frames [Abb. 4.8-1] in geschweiften. CH: Client Handshake Random, SH: Server Handshake Random

- Ein erstes *coalesced* `Handshake`-Paket (`0x02`) mit kryptographischem TLS-Material und einem *Finished* hierfür.
- Ein abschließendes coalesced `0-RTT`-Paket (`0x01`) mit weiteren Daten im `Stream`-Frame.

3. Hierauf antwortet der Client mit folgenden Angaben im UDP-Datagram:
   - Als Erstes ein `INIT`-Paket (`0x00`) mit Sequenznumer `1` und einem `Ack`-Frame auf das Server `Crypto`-Frame.
   - Ein `Handshake`-Paket (`0x02`), das kryptographische Informationen des Clients im `Crypto`-Frame mitteilt und das vorige Frame des Servers mit einem `Ack`-Frame bestätigt.
   - Ein finales `0-RTT`-Paket (`0x01`) mit ergänzenden Daten im `Stream`-Frame und einem `Ack`-Frame auf die im `Stream`-Frame gesendeten Daten des Servers.
4. Die Verbindungsaufnahme wird abgeschlossen durch ein Server UDP-Datagramm
   - mit einem `0-RTT`-Paket (`0x02`) und einem `Ack`-Frame auf das vorherige Client `0-RTT`-Paket und
   - mit einem abschließenden `1-RTT`-Paket (`0x80`) mit Sequenznummer `1`, das ein `Handshake_Done`-Frame, ein `Stream`-Frame und ein `Ack`-Frame für die vom Server gesendeten Daten im `Stream`-Frame beinhaltet.

Getrennte Zahlenräume für QUIC-Pakete

Wie aus Abb. 4.8-4a und Abb. 4.8-4b zu entnehmen ist, werden die QUIC-Pakete enummeriert. In diesen Abbildungen erfolgt die Angabe in runden Klammern – in Ergänzung zum hexadezimal dargestellten Pakettyp. Bei der Einbettung des Zahlenwertes der Sequenznummer in Abb. 4.8-3 wird vom *Variable Length Encoding* Gebrauch gemacht wird. Jeder Pakettyp verfügt über einen eigenen Zahlenraum und muss somit spezifisch im Speicher der Kommunikationspartner zusammen mit der `Connection ID` geführt werden.

Als notwendiger Bestandteil der Verbindungsaufnahme und der Aushandlung von Verbindungsparametern, wie z.B. die gegenseitige Nutzung von Flusskontrollmechanismen, müssen beide Kommunikationspartner sich auf das eigentliche Applikationsprotokoll über UDP-Port 443 einigen. Hiermit erhält QUIC die Möglichkeit, über Port 443 ein Multiplexing unterschiedlicher Anwendungen durchzuführen.

0-RTT Verbindungsaufnahme

Beim 0-RTT Verbindungsaufbau ergibt sich die Neuerung, dass bereits im ersten UDP-Datagramm Nutzdaten in Form eines *coalesced* `0-RTT`-Paketes (*Short Header*) untergebracht werden können. Hiermit ergibt sich die Möglichkeit – wie bei TLS 1.3 –, auf *Pre-shared Keys* zurückzugreifen [Abb. 4.8-4b]. Ansonsten folgt der Verbindungsaufbau weitgehend dem in Abb. 4.8-4a gezeigten, sodass auf eine weitere Betrachtung verzichtet werden soll.

Die Möglichkeit, Nutzdaten bereits in der Verbindungsphase zu übertragen, ist aber kein Alleinstellungsmerkmal für den 0-RTT-Verbindungsaufbau: Auch bereits beim 1-RTT-Mechanismus können diese als `0-RTT`-Pakete per `Stream`-Frames mitgegeben werden.

4.8.5 Verbindungsmanagement bei QUIC

Das QUIC-Protokoll nutzt intern (virtuelle) Verbindungen zwischen zwei Connection IDs, die unterhalten werden wollen: Ein Stream. Angelehnt an TCP stehen auch Mechanismen bereit, diese Verbindung zu monitoren und den Durchsatz besonders dann zu optimieren, wenn Paketverluste eintreten.

Eine weitere Spezialität von QUIC ist, die Verbindung unabhängig von der bestehenden Netzwerkstruktur zu betreiben; ein Feature, das wir bei TCP unter dem Begriff *Multipath TCP* [Abschnitt 4.6] kennen gelernt haben.

Stream ID

Obwohl QUIC auf das verbindungslose UDP aufsetzt, erfolgt der Datenaustausch zwischen zwei Connection IDs, die die Endpunkte bei QUIC darstellen, über einen (virtuellen) Datenstrom, auf Grundlage des Stream-Frames [Tab. 4.8-1]. Während die Stream-Frames für die QUIC-Pakete die Kennungen `0x08` bis `0x0f` besitzen, wird jedem Stream eine 62-Bit-Ganzzahl als Stream-Identifier (`Stream ID`) zugewiesen. Hierbei werden die beiden geringstwertigsten Bits genutzt, die Art des Datenstroms zu charakterisieren:

- `0x00`: Vom Client ausgehend; bi-direktional
- `0x01`: Vom Server ausgehend: bi-direktional
- `0x02`: Vom Client ausgehend; uni-direktional
- `0x03`: Vom Server ausgehend: uni-direktional

Eine besondere Priorisierung von Datenströmen ist nicht vorgesehen; die Verschlüsselung wird TLS überlassen [RFC 9001].

Stream-Frame-Aufbau

Ein Stream-Frame besitzt folgenden Aufbau:

1. Ein `Type`-Feld mit der Binärstruktur '`00001xxx`', bei der die letzten drei Bits Folgendes mitteilen:
   - `0x04` – Ein Offset-Feld ist vorhanden (Bit 2 gesetzt).

   - `0x02` – Die Länge des Frames wird mitgeteilt (Bit 1 gesetzt).
   - `0x01` – Das Ende des Datenstroms ist gegeben (`FIN` – Bit 0 gesetzt).
2. Die `Stream ID`.
3. Der *Offset* (falls `0x04` vorliegt).
4. Die *Länge* (falls `0x02` gegeben ist).
5. Die eigentlichen *Streamdaten*.

Acknowledegments

Nach Etablierung des Datenstroms kann gesendet werden, wobei der Sender zu diesem Moment eine 'Zustandsmaschine' instantiieren muss, die den (erfolgreichen) Datenaustausch mittels der empfangenen `ACK`s und bis zum abschließenden `FIN` (-Bit) begleitet. Jedes QUIC-Paket muss durch ein von der Gegenseite erzeugtes `ACK`-Frame bestätigt werden. Für das Acknowledgement sieht QUIC wiederum einen Mechanismus vor, mit dem die Verzögerung per `max_ack_delay` im `ACK`-Frame gegenseitig mitgeteilt (und wiederum bestätigt) wird.

Frames können mittels `Resent_Frames` erneut verschickt werden; der Sender muss also die Daten im Sendepuffer halten, bis ein zugehöriges `ACK`-Frame empfangen wurde.

Kontrolle des Stream-Datenflusses

Im Gegenzug verfügt der Empfänger über einen Empfangspuffer, in dem die Daten gehalten werden, bis diese zur Applikation durchgereicht werden können. Von diesen Empfangspuffern gibt es pro Verbindung zumindest einen, und das QUIC-Protokoll bietet die Möglichkeit, deren Anzahl und Größe auf der Empfangsseite einzugrenzen:

- Mittels `Max_stream_data` kann ein einzelner *Datenstrom* auf die hierin angegebene Datenmenge beschränkt werden.
- Ergänzend kann die gesamte maximal akzeptierte Datenmenge eines *Senders* per `Max_data` limitiert werden.
- Wie viele Datenströme pro Verbindung hierbei parallel geöffnet werden dürfen, legt der Empfänger implizit mit seiner vergebenen `Connection ID` fest. Hierbei gilt: Max. Concurrency = $\texttt{Max\_stream} * 4 + \texttt{first\_stream\_id\_of\_type}$.

Connection Migration

Bei der Nutzung mobiler Endgeräte oder nach einer Neuvergabe einer IP-Adresse z.B durch DHCP (siehe Abschnitt 6.2 und Abschnitt 9.4) muss die Datenverbindung neu etabliert werden, da sich bei der UDP/IP-Kommunikation der lokale Socket geändert hat. Das QUIC-Protokoll abstrahiert aber von der IP-Adresse, indem es an dieser Stelle eine `Connection ID` führt.

Probing Frames

Die `Connection ID` bleibt (für die bestehende Verbindung) auch bei wechselnden IP-Adressen gleich. Stellt z.B. der Client fest, dass sich seine lokale IP-Adresse geändert hat, kann er dies mittels der QUIC-Frames `Path_Challenge`, `Path_Response` sowie `New_Connection_Id`, unterstützt durch `Padding`, mitteilen und ein *Re-Binding* auf der IP/UDP-Schicht anfordern.

None Probing Frames

Im Gegenzug kann er aber auch feststellen, dass sich die IP-Adresse des Clients geändert hat, wenn er für eine bestehende `Connection ID` plötzlich neue Verbindungsdaten erhält. Wird dies für *None Probing Frames* (also z.B. *Stream-Frames*) registriert,

muss er seinerseits eine Pfad-Validierung durchführen und sicherstellen, dass die IP-Adresse des Client gültig, d.h. nicht '*gespooft*' ist.

4.9 Schlussbemerkungen

In diesem Kapitel wurden die Protokolle UDP, TCP und SCTP der Transportschicht in IP-Netzen in fundierter Form dargestellt. Aus Platzgründen konnten nicht alle Aspekte der Transportprotokolle präsentiert werden. Abschließend ist noch Folgendes hervorzuheben:

Unterstützung der Echtzeitkommunikation

- Das Internet wird zunehmend für die Internet-Telefonie und für die Übermittlung von Streaming-Medien (Audio und Video) unter Einsatz des Protokolls RTP (*Real-time Transport Protocol*) eingesetzt. Dies stellt ganz neue Herausforderungen an die Transportschicht in IP-Netzen, weil man durch die zu erwartende Flut von audiovisuellen Medien zukünftig mit der Überlastung des Internet rechnen muss. Da RTP heute das verbindungslose UDP nutzt und UDP über keine Mechanismen für die Überlastkontrolle (*Congestion Control*) verfügt, ist ein neues verbindungsloses Transportprotokoll nötig, mit dem die Echtzeitkommunikation und die Überlastkontrolle möglich sind.

Neues Protokoll DCCP

- Ein derartiges Transportprotokoll DDCP (*Datagram Congestion Control Protocol*) wurde bereits in RFC 4340 (2006) veröffentlicht. Wie bei UDP werden die DCCP-Pakete ebenso als selbstständige Datagramme unquittiert übermittelt. Im Gegensatz zu TCP kann DCCP aber als *Paketstrom* – und nicht als *Bytestrom* – angesehen werden. Für die Überlastkontrolle nutzt DCCP das Verfahren ECN (*Explicit Congestion Notification*) [RFC 3168], das ursprünglich für TCP entwickelt wurde. Das Konzept für RTP-over-DCCP – als Grundlage für die multimediale Kommunikation über IP-Netze – ist bereits spezifiziert [RFC 5762]. Es ist zu erwarten, dass DCCP zukünftig eine wichtige Rolle spielen wird. Für detaillierte Informationen über die DCCP-Entwicklung sei auf die IETF-Arbeitsgruppe `https://www.ietf.org/html.charters/dccp-charter.html` verwiesen.

SCTP → QUIC

- Das neue QUIC-Protokoll von Google wird natürlich in erster Linie vom Webbrowser *Chrome* auf der Client-Seite unterstützt. Da die hierauf aufbauende Webkit-Applikation es selbst bis zum Microsoft-Browser *Edge* geschafft hat, ist ein Siegeszug des QUIC-Protokolls für Web-Anwendungen somit gar nicht mehr zu verhindern. Es tritt damit die Nachfolge von SCTP an. Inwieweit weitere Applikationen wie *DNS over QUIC* (DoQ [`https://datatracker.ietf.org/doc/html/draft-huitema-dprive-dnsoquic-00`] – geplant für RFC 9250) folgen werden, ist noch nicht abzusehen.

SCTP bei WebRTC

- WebRTC ist eine richtungsweisende Idee, um multimediale Echtzeitkommunikation, d.h. Datenkommunikation mit einem Webserver und gleichzeitig eine *audiovisuelle Echtzeitkommunikation* mit Webbrowsern untereinander realisieren zu können, ohne dafür zusätzliche Software-Module installieren zu müssen. Zur Echtzeitkommunikation wird bei WebRTC das Transportprotokoll SCTP zwischen zwei 'kommunizierenden' Webbrowsern genutzt, damit zwischen ihnen mehrere Datenströme parallel übermittelt werden können. Zur Garantie der Sicherheit soll das Sicherheitsprotokoll DTLS eingesetzt werden – also SCTP over DTLS.

4.10 Verständnisfragen

1. Müssen UDP- bzw. TCP-Sockets für einen gleichen Service die gleiche Protokollnummer tragen (vgl. `/etc/services`)?
2. Es wurden zwei verbindungslose und unzuverlässigen Protokolle vorgestellt: UDP und UDP-Lite. Wie stellen diese Protokolle die Integrität der transportierten Nutzdaten sicher?
3. Welche Optionen können im TCP-Header mitgeteilt werden?
4. Wie funktioniert der '3-Way Handshake' bei TCP?
5. Was wird mit der 16 Bit 'Sequence Number' im TCP-Paket gezählt?
6. Wie wird dieser Wert bei der Verbindungsaufnahme initialisiert?
7. Was passiert bei TCP, falls dieser Zähler überlaufen sollte?
8. Was wird über die 'Acknowledgement Number' mitgeteilt?
9. Das in Abb. 4.3-4 und Abb. 4.3-5 als Beispiel gezeigte FTP-Verfahren stellt eine absolute Ausnahme dar, da hier Kommandos und Daten über unterschiedliche Sockets übertragen werden, was speziell beim Einsatz von FTP über Firewalls zu großen Schwierigkeiten führt. Wie kann dies praktisch umgangen werden?
10. Wozu werden die TCP-Optionen 'Window Size' und 'Segment Size' genutzt?
11. Werden die TCP-Erweiterungen ECN bzw. SCTP auch in der Praxis eingesetzt?
12. Können Sie in Ihrem Webbrowser die Nutzung des QUIC-Protokolls ein- bzw. ausschalten? (Tipp: In URL-Zeile `<browser>/flags` aufrufen; <browser> ist Name Ihres Browsers)