# 4 Transport-Protokolle TCP und UDP

Die TCP/IP-Protokollfamilie stellt für die Übertragung der Nutzdaten zwei sehr unterschiedliche Transportprotokolle auf IP-Basis zur Verfügung:

- Das *Transmission Control Protocol TCP*, das einen gesicherten, verbindungsorientierten Vollduplex-Datenstrom zwischen Sender und Empfänger ermöglicht.
- Das *User Datagram Protocol UDP*, über das das eine ungesicherte, verbindungslose Kommunikation zwischen Sender und Empfänger abgewickelt werden kann.

Neben diesen beiden Protokollen umfaßt die TCP/IP-Protokollfamilie noch weitere Transportprotokolle, die jedoch auf TCP und UDP aufsetzen und sich deren jeweilige Eigenschaften zunutze machen:

- Übertragung der *ISO-Protokolle TP0* bis *TP4* (=> RFC 905, 1240 und 2126)
- *NetBIOS* über TCP/UDP (=> RFC 1001/1002)
- *Real Time Protocol* (*RTP*) über UDP (=> RFC 1889)
- *System Network Architecture* Protokoll *SNA* über UDP (=> RFC 1538)

Während TCP im Laufe der Entwicklung mit seinen Kontrollmechanismen zusehends verfeinert wurde, ist das UDP Protokoll aufgrund seines sehr schmalen Funktionsumfangs im wesentlichen unverändert geblieben, sieht man von den *Portmapper*-Diensten ab, die seinen Einsatz stark vereinfachen.

## 4.1 Funktion des Protokolls TCP

Das Protokoll TCP wurde bereits in Kapitel 2 kurz vorgestellt. Nun gehen wir auf die Besonderheiten von TCP-Verbindungen sowie auf die Prinzipien der Datenübermittlung näher ein. Die Bedeutung des Protokolls TCP kommt in Abbildung 4.1-1 zum Ausdruck. Die Protokolle IP und TCP sind im Schichtenmodell des Endsystems entsprechend den Schichten 3 und 4 zuzuordnen. Die Aufgabe des IP-Protokolls besteht in der Übermittlung von IP-Paketen über ein privates IP-Netz bzw. das weltweite Internet als selbständige Datagramme. Die IP-Adresse in einem Schichtenmodell kann der Grenze zwischen den Schichten 3 und 4 zugeordnet und als ein Kommunikationspuffer interpretiert werden. Die Datenkommunikation zwischen zwei Rechnern kann somit als ein Austausch von IP-Paketen zwischen Kommunikationspuffern in diesen Rechnern gesehen werden. Da das Protokoll IP keine Fehlerkontrolle bei der Übermittlung dieser Pakete realisiert, ist der Datenaustausch unsicher. Beispielsweise werden die Verluste bzw. Verfälschungen (Verletzung) von Daten während der Übermittlung mit dem Protokoll IP nicht entdeckt. Das Protokoll TCP ist als ein Transportprotokoll zu bezeichnen, dessen Aufgabe es ist, den Datenaustausch zwischen zwei kommunizierenden Rechnern auf einer sicheren Basis zu gewährleisten, d.h. Datenverluste und -verfälschungen während der Übertragung zu entdecken und dementsprechend eine wiederholte Übertragung zu veranlassen.

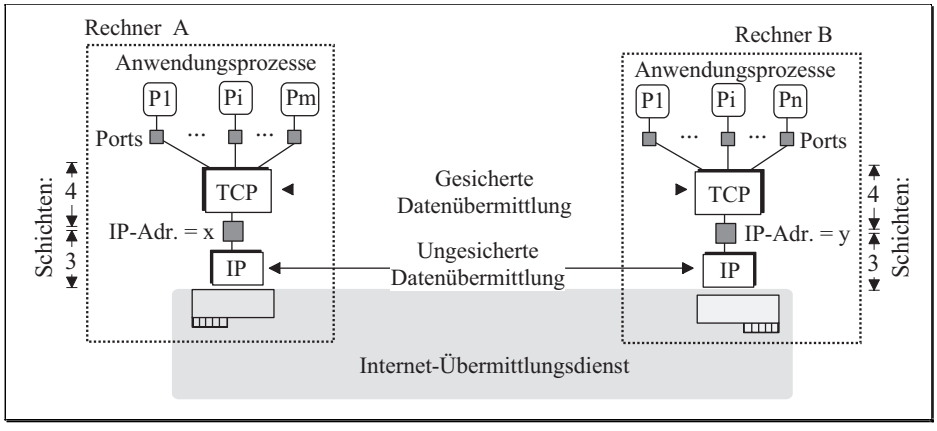


**Abbildung 4.1-1:** TCP als Sicherungsprotokoll zwischen zwei entfernten Endsystemen

*Ports* Wie bereits im Abschnitt 2.2 gezeigt wurde, werden sogenannte Ports den aktiven TCP/IP-Anwendungen als Anwendungsprozessen zugeordnet. Diese Ports stellen die individuellen Kommunikationspuffer einzelner Anwen-

dungsprozesse dar. Die Ports werden den Anwendungsprozessen nach Bedarf (auch dynamisch) zugeordnet. Da die Port-Nummern 16 Bits lang sind, könnte ein Rechner theoretisch gleichzeitig bis zu 65 535 Ports organisieren. Die Ports mit den Nummern 0 bis 1023 sind weltweit eindeutig für Standarddienste (sog. *Well Known Services*) wie z.B. TELNET, FTP, HTTP von vornherein reserviert. Die reservierten Ports von Standardanwendungen werden auch als *Well Known Ports* bezeichnet. Unter den Nummern im Bereich von 1 024 bis 65 535 können sie im Rechner den Anwendungsprozessen frei zugeteilt werden.

*TCP als Sicherungsprotokoll*

Abbildung 4.1-1 demonstriert überdies, daß das Protokoll TCP auch als ein logischer Multiplexer von Anwendungsprozessen gesehen werden kann. Das Protokoll TCP ermöglicht es, daß mehrere Anwendungsprozesse auf die Dienste des Protokolls IP gleichzeitig zugreifen können. Jedes Sicherungsprotokoll ist verbindungsorientiert (=> z.B. HDLC). Dies setzt voraus, daß eine „Beziehung“ zwischen zwei entsprechenden Kommunikationspuffern in den kommunizierenden Rechnern zustande kommen muß. Um die Datenübermittlung nach dem Protokoll TCP sicher zu machen, muß ebenfalls eine logische Verbindung aufgebaut werden. Diese logische Verbindung wird im folgenden als *TCP-Verbindung* bezeichnet. Es stellt sich nun die Frage, wie man TCP-Verbindungen identifiziert?

### 4.1.1 Aufbau von TCP-Paketen

Über eine TCP-Verbindung werden die Daten in Form von festgelegten Datenblöcken – von nun an „TCP-Pakete“ genannt – ausgetauscht. Aufgabe von TCP ist es, die zu übertragenden Daten im Quellrechner in numerierte Segmente zu zerlegen, was man als *Segmentierung* bezeichnet. Dabei versieht TCP jedes Datensegment mit einem TCP-Header. Vor Beginn einer Übertragung wird zwischen den TCP-Instanzen im Quell- und im Zielrechner die maximale Segmentgröße vereinbart (z.B. 15 000 Byte). Die TCP-Pakete werden dann vom IP-Protokoll als zusammenhanglose IP-Pakete (*Datagramme*) übertragen. Die TCP-Instanz des Zielrechners setzt die empfangenen IP-Pakete in der richtigen Reihenfolge in die ursprünglichen Daten (*Nachricht*) zurück. Erreicht ein IP-Paket den Zielrechner nicht, wird die Wiederholung der Übertragung eines entsprechenden Datensegments von TCP veranlaßt.

Abbildung 4.1-2 zeigt den Aufbau des TCP-Headers.

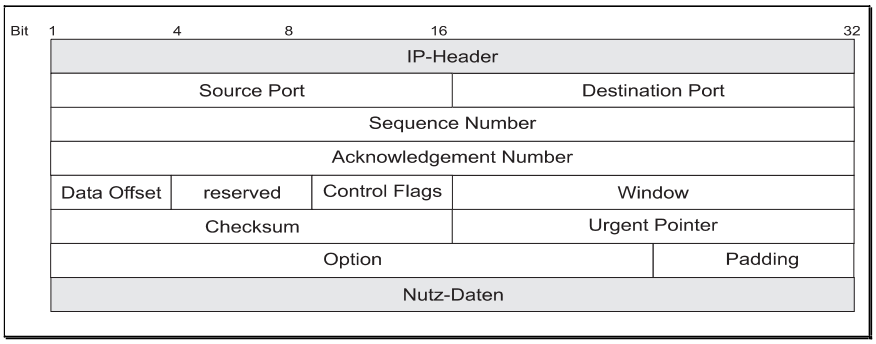


**Abbildung 4.1-2:** Aufbau des TCP-Headers

Die Steuerungsinformationen im TCP-Header haben folgende Bedeutung:

- *Source Port (Quell-Port):*
  Der Quell-Port enthält die Portnummer des Anwenderprozesses im Quellrechner, der die Verbindung initialisiert hat.
- *Destination Port (Ziel-Port):*
  Der Ziel-Port enthält die Portnummer des Anwenderprozesses im Zielrechner, an den die Daten adressiert sind. Die Bedeutung der Portnummer wurde bereits in Abschnitt 2.2 erläutert.
- *Sequence Number (Sequenznummer):*
  Die Sequenznummer gilt in der Senderichtung und dient der Numerierung von gesendeten Datensegmenten. Beim Aufbau einer virtuellen Ende-zu-Ende-Verbindung generiert jedes TCP-Modul eine Anfangs-Sequenznummer. Diese Nummern werden ausgetauscht und gegenseitig bestätigt. Um die gesendeten TCP-Pakete eindeutig am Zielrechner zu identifizieren, muß der Fall ausgeschlossen werden, daß sich zu einem Zeitpunkt – im Netz – mehrere TCP-Pakete mit der gleichen Sequenznummer befinden. Aus diesem Grund darf sich die Sequenznummer innerhalb der festgelegten Lebenszeit für die IP-Pakete nicht wiederholen. Im Quellrechner wird die Sequenznummer immer jeweils um die Anzahl bereits gesendeter Bytes erhöht.
- *Acknowledgement Number (Quittungsnummer):*
  Die Quittungsnummer gilt in der Empfangsrichtung und dient der Bestätigung von empfangenen Datensegmenten. Diese Nummer wird vom Zielrechner gesetzt, um dem Quellrechner mitzuteilen, bis zu welchem Byte die Daten korrekt empfangen wurden.

- *Data Offset (Datenabstand):*
  Das vier Bit große Feld gibt die Länge des TCP-Headers in 32-Bit-Worten an und damit die Stelle, ab der die Daten beginnen.
- *Control Flags (Kontroll-Flags):*
  Die Kontroll-Flags legen fest, welche Felder im Header gültig sind, und steuern die Verbindung. Die Struktur sieht 6 Bits vor, und wenn das entsprechende Bit gesetzt ist, gelten die folgenden Bedingungen:
  - *URG*: Der Urgent Pointer (Zeiger im Urgent-Feld) ist gültig.
  - *ACK*: Die Quittungsnummer ist gültig.
  - *PSH*: Push-Funktion: Die Daten sollen sofort an die nächsthöhere Schicht weitergegeben werden. UNIX-Implementierungen senden PSH immer dann, wenn sie mit dem Segment gleichzeitig alle Daten im Sendepuffer übergeben.
  - *RST* (Reset): Die Verbindung soll zurückgesetzt werden.
  - *SYN*: Verbindungsaufbauwunsch; es muß quittiert werden.
  - *FIN*: Einseitiger Verbindungsabbau und das Ende des Datenstroms aus dieser Richtung. Es muß quittiert werden.
- *Window (Fenstergröße):*
  Mit dieser Angabe, die der Flußkontrolle nach dem Fenster-Mechanismus dient (=> Abbildung 4.2-1), steuert der Zielrechner den an ihn gesendeten Datenstrom im Quellrechner. Das Feld gibt die Fenstergröße an, d.h. wie viele Bytes – beginnend ab der Quittungsnummer – der Zielrechner in seinem Aufnahme-Puffer noch aufnehmen kann. Empfängt der Quellrechner ein TCP-Paket mit der Fenstergröße gleich 0, muß der Sendevorgang gestoppt werden. Wie die Fenstergröße die Effizienz der Übermittlung beeinflussen kann, ist aus Abbildung 4.2-4 ersichtlich (Senderblockade). Die Ermittlung einer optimalen Fenstergröße gehört zu den wichtigsten Aufgaben bei der TCP/IP-Implementierung.
- *Checksum (Prüfsumme):*
  Diese Prüfsumme erlaubt es, den TCP-Header, die Daten und einen Auszug aus dem IP-Header (u.a. Quell- und Ziel-IP-Adresse), der an das TCP-Protokollmodul zusammen mit den Daten übergeben wird, auf das Vorhandensein von Fehlern (Bitfehler, Datenverlust) zu überprüfen. Bei Berechnung der Prüfsumme wird dieses Feld selbst als null angenommen.
- *Urgent Pointer (Urgent-Zeiger):*
  Das Protokoll 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 hierbei als Urgent-Daten bezeichnet. Ist der Urgent-Zeiger gültig, d.h. URG = 1, so zeigt er auf das Ende von Urgent-Daten. Sie werden immer direkt nach dem TCP-Header übertragen. Erst danach folgen „normale“ Daten.

- *Option:*
  Das TCP erlaubt es, Service-Optionen anzugeben. Das erste Byte im Optionsfeld legt den Optionstyp (*kind*) fest, wobei das Optionsfeld nun entsprechend Tabelle 4.1-1 zu interpretieren ist. Mit den RFC 1323 und 2018 wurde das Optionsfeld in seiner Bedeutung wesentlich erweitert und besitzt folgende Bedeutung:
  - *Maximum Segment Size* (*MSS*): Diese Option wird beim Verbindungsaufbau genutzt, um dem Kommunikationspartner mitzuteilen, welche maximale Segmentgröße verarbeitet werden kann.
  - *Window Scale* (*WSopt*): Mittels der Option *WSopt* können die Kommunikationspartner optional während der Initialisierung (also beim SYN-Segment) festlegen, ob die Größe des 16 Bit *Window* (siehe oben) um einen konstanten Skalenfaktor multipliziert wird. Dieser Wert kann unabhängig für den Empfang (*R*) und das Versenden (*S*) von Daten ausgehandelt werden. Als Konsequenz dieses Verfahrens 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. Auf Grundlage des Skalenfaktors von *WSopt* wird auch der übertragene Wert von *Window* im TCP-Segment neu berechnet.
  - *Timestamps Option* (*TSopt*): Dieses Feld besteht aus den Teilen *Timestamp Wert* (*TSval*) und *Timestamp Echo Reply* (*TSecr*). Letzteres Feld ist nur bei ACK-Segment erlaubt. Mit diesen Informationen informieren sich die TCP-Instanzen über die sog. *Round Trip Zeit*, was im nächsten Abschnitt genauer erläutert ist.
  - Mit RFC 2018 wurde das Verfahren *Selective Acknowledgement* (*SACK*) eingeführt, das sich wesentlich auf das Optionsfeld stützt und es zuläßt, dieses Feld variabel zu erweitern. Auch auf dieses Verfahren wollen wir später im Detail eingehen.
  - Ergänzt werden die Optionen schließlich noch um den *Connection Count CC*, sowie *CC.NEW* und *CC-ECHO*, die bei der Implementierung T/TCP (=> RFC 1644) anzutreffen sind und die wir in Abschnitt 4.2.3 diskutieren.

| Optionstyp [kind] | Länge des Optionsfelds [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 | Times-Stamp-Abgleich |
| 11 | 6 | Connection Count CC (T/TCP) |
| 12 | 6 | CC.NEW (T/TCP) |
| 12 | 6 | CC.ECHO (T/TCP) |

**Tabelle 4.1-1:** Belegung des Optionsfeldes im TCP-Header

- *Padding (Füllzeichen):*
  Die Füllzeichen ergänzen die Optionsangaben auf die Länge von 32 Bits.

*TCP Timeouts*

TCP verhindert den gleichzeitigen Verbindungsaufbau zwischen zwei Stationen, d.h. nur eine Station kann den Aufbau initiieren. Des weiteren ist es nicht möglich, einen mehrfachen Aufbau einer Verbindung durch den Sender aufgrund eines Timeouts des ersten Verbindungsaufbauwunsches zu generieren. Der Datenaustausch zwischen zwei Stationen erfolgt nach dem Verbindungsaufbau. Gehen Daten bei der Übertragung verloren, wird nach Ablauf eines Timeouts die Wiederholung der fehlerhaften Segmente gestartet. Durch die Sequenznummer werden doppelt übertragene Pakete erkannt. Aufgrund der Sequenznummer ist es möglich, $2^{32}$-1 Daten (8 Gigabyte) pro bestehender Verbindung zu übertragen.

*Fenster-Mechanismus*

Die Flußkontrolle nach dem Fenster-Mechanismus (*Window*-Feld) erlaubt es einem Empfänger, dem Sender mitzuteilen, wieviel Pufferplatz zum Empfang von Daten zur Verfügung stehen. Ist der Empfänger zu einem bestimmten Zeitpunkt der Übertragung einer höheren Belastung ausgesetzt, kann er dies dem Sender über das Window-Feld bekanntgeben.

*Retransmission Time*

Jedes übertragene TCP-Paket unterliegt einer Zeitüberwachung (*Retransmission Time*); das bedeutet, daß ein Empfänger nach einer bestimmten Zeitdauer eine Quittung über die erhaltenen Frames aussenden muß. Da diese Zeitdauer stark von der aktuellen Belastung des Netzes abhängt, muß der Retransmission Timer für jedes TCP-Paket neu berechnet und eingestellt werden.

### 4.1.2 Konzept der TCP-Verbindungen

Eine TCP-Verbindung wird mit dem Ziel aufgebaut, einen sicheren Datenaustausch zwischen den kommunizierenden Anwendungsprozessen in entfernten Rechnern zu gewährleisten. Als Beispiel nehmen wir an, daß die Kommunikation zwischen zwei FTP-Anwendungen (*File Transfer Protocol*) stattfinden soll. Die FTP-Anwendung ist eine TCP/IP-Standardanwendung und hat den *Well Known Port* Nummer 20. Aus Sicht der Kommunikation kann die Nummer des Well Known Ports auch als die weltweit eindeutige Identifikation einer TCP/IP-Standardanwendung gesehen werden.

*Three Way Handshake*

Die TCP-Verbindungen sind vollduplex. Man kann eine TCP-Verbindung als ein Paar von gegenseitig gerichteten unidirektionalen Verbindungen interpretieren. Der Verbindungsaufbau zwischen zwei Rechnern, die das TCP-Protokoll benutzen, erfolgt immer mit Hilfe des *Three Way Handshake* (*3WHS*)-Verfahrens, das für eine Synchronisation der Kommunikationspartner sorgt und sicherstellt, daß die TCP-Verbindung in jede Richtung korrekt initialisiert wird (=> vollduplex). An dieser Stelle ist hervorzuheben, daß der Anwendungsprozeß im Quellrechner mit dem TCP über einen wahlfreien Port kommuniziert, der dynamisch (aber nur im Quellrechner!) zugewiesen wird.

*TCP-Zustands-modell*

Das TCP-Modell geht von einer sog. *Zustandsmaschine* aus. Eine TCP-Instanz befindet sich immer in einem wohldefinierten Zustand. Die Hauptzustände sind hierbei (engl.) *Listen* und (Verbindung) *Established*. Zwischen diesen stabilen Zuständen gibt es gemäß Abbildung 4.1-3 eine Vielzahl zeitlich befristeter (Zwischen-)Zustände. Mittels der TCP-Kontroll-Flags ACK, FIN, SYN und ggf. auch RST wird zwischen den Kommunikationspartnern der Wechsel bzw. der Verbleib in einem Zustand signalisiert.

Zur effizienten Nutzung des Fenstermechanismus stehen zwei Parameter zur Verfügung, die zwischen den TCP-Instanzen im Verlauf der Kommunikation dynamisch angepaßt werden:

*Windowsize (WSIZE)*

Die *Windowsize* (*WSIZE*) ist zu interpretieren als die Größe des TCP-Empfangspuffers in Byte. Aufgrund des maximal $2^{16}$-1 großen Feldes im TCP-Header kann dieses maximal einen Wert von 65 535 Byte, d.h. rund 64 KByte (K = 1024) aufweisen. Moderne TCP-Implementierungen nutzen allerdings die in den TCP-Segmenten vorgesehene Option des *WSopt*, so daß nun Werte bis $2^{30}$, also rund 1 GByte möglich sind.

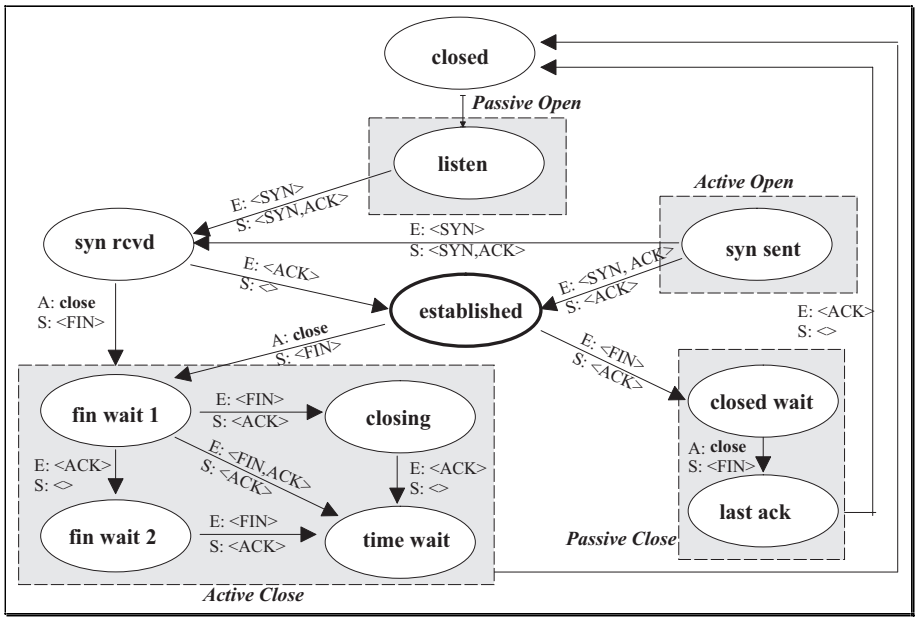


**Abbildung 4.1-3:** TCP-Zustandsdiagramm
E: Empfänger, S: Sender, A: Applikation

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

*Maximum Segment Size (MSS)*

Die *Maximum Segment Size MSS* stellt das Gegenstück zu *WSIZE* dar, ist also der maximale Wert des TCP-Sendepuffers. Für übliche TCP-Implementierungen gilt die Ungleichung *MSS* < *WSIZE*. Die dynamische Aushandlung dieser Parameter zusammen mit der Methode der Bestimmung der sog. *Round Trip Time* begründet ursächlich das gute Übertragungsverhalten von TCP auf sehr unterschiedlichen Trägernetzen (=> Abschnitt 4.2.2).

### 4.1.3 Auf- und Abbau von TCP-Verbindungen

Den Aufbau einer TCP-Verbindung illustriert Abbildung 4.1-4. 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. Die Kommunikationspartner befinden sich

zum Anfang der Übertragung immer in folgenden Zuständen (=> Abbildung 4.1-3):

- *Passives Öffnen:*
  Eine Verbindung tritt in den Abhörstatus ein, wenn eine Anwendungsinstanz TCP mitteilt, daß sie Verbindungen für eine bestimmte Portnummer annehmen würde.
- *Aktives Öffnen:*
  Eine Anwendungsinstanz teilt TCP mit, daß es eine Verbindung mit einer bestimmten IP-Adresse und Portnummer eingehen möchte, was mit einer bereits abhörenden Anwendungsinstanz korrespondiert.

*Beispiel: FTP* Im vorliegenden Beispiel wird die TCP-Verbindung im Rechner *A* mit der IP-Adresse *x* durch den FTP-Prozeß initiiert. Hierbei wird diesem FTP-Prozeß für die Zwecke der Kommunikation die Port-Nummer (beispielsweise 3028) zugewiesen. Die TCP-Protokoll-Instanz im Rechner *A* generiert ein TCP-Segment, in dem das Flag SYN gesetzt ist. Somit wird dieses Segment hier als <SYN>-Segment bezeichnet. Der Verbindungsaufbau beginnt damit, daß die beiden Kommunikationspartner einen Anfangswert für die jeweiligen Sequenznummern festlegen. Dieser Anfangswert für eine Verbindung wird als *Initial Sequence Number* (*ISN*) bezeichnet.

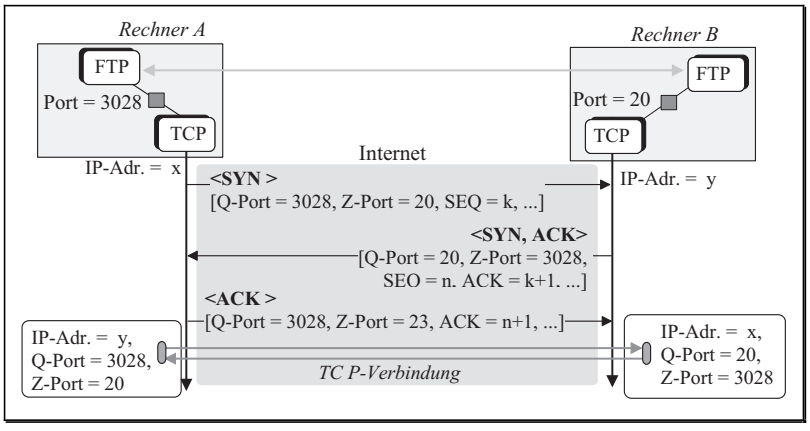


**Abbildung 4.1-4:** Beispiel für den Aufbau einer TCP-Verbindung
Q: Quell, Z: Ziel

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

- SYN-Flag im TCP-Header wird gesetzt (=> <SYN>-Segment),
- frei zugeteilte Nummer des Quell-Ports,

- Zielport als Well Known Port,
- SEQ: Sequenznummer der Quell-TCP-Instanz (hier SEQ = k).

*SYN-Bit*

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 Standardanwendung TCP im Rechner *B* gefordert.

Die Ziel-TCP-Instanz befindet sich im sogenannten Listenmodus, so daß sie auf ankommende <SYN>-Segmente wartet. Nach dem Empfang eines <SYN>-Segments leitet die Ziel-TCP-Instanz ihrerseits den Verbindungswunsch an den Ziel-Anwendungsprozeß (hier FTP-Prozeß) gemäß der empfangenen Nummer des Zielports weiter und generiert eigene Initial Sequence Number (ISN) für die Richtung zum Rechner *A*. Im zweiten Schritt des Verbindungsaufbaus wird ein TCP-Segment im Rechner *B* mit folgendem Inhalt an den Rechner *A* zurückgeschickt:

- Die beiden Flags SYN und ACK im TCP-Header werden gesetzt (=> <SYN+ACK>-Segment).
- Die beiden Quell- und Ziel-Port-Nummern werden angegeben.
- Die Sequenznummer SEQ der Ziel-TCP-Instanz (hier SEQ = n) wird mitgeteilt.

*ACK-Bit*

Das ACK-Bit signalisiert, daß die Quittungsnummer (hier kurz ACK) in diesem <SYN,ACK>-Segment von Bedeutung ist. Die Quittungsnummer ACK enthält die nächste von der TCP-Instanz im Rechner *B* erwartete Sequenznummer (SEQ).

Die TCP-Instanz im Rechner *A* bestätigt den Empfang des <SYN,ACK>-Segments mit einem <ACK>-Segment, in dem das *ACK*-Flag gesetzt wird. Mit der Quittungsnummer ACK = n +1 wird der TCP-Instanz *i* Rechner *B* bestätigt, daß die nächste Sequenznummer n+1 erwartet wird.

Aus Abbildung 4.1-4 geht außerdem hervor, daß sich eine TCP-Verbindung aus zwei unidirektionalen Verbindungen zusammensetzt. Jede dieser gerichteten Verbindungen wird im Quellrechner durch die Angabe der Ziel-IP-Adresse und von beiden Quell- und Zielports eindeutig identifiziert.

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

*Aufbau einer TCP-Verbindung*

Den Abbau einer TCP-Verbindung illustriert Abbildung 4.1-5.

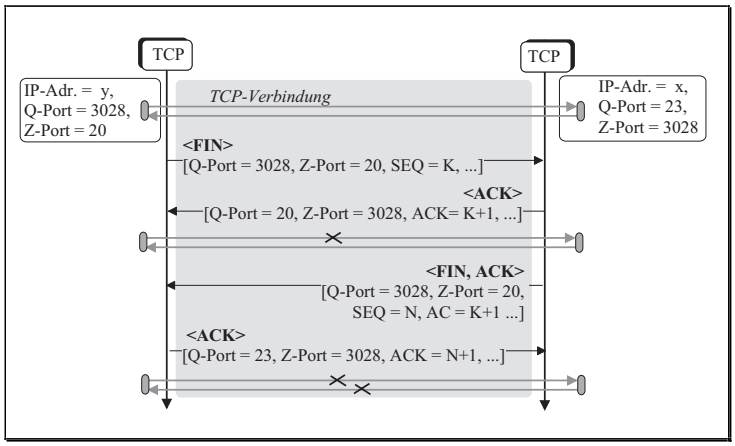


**Abbildung 4.1-5:** Beispiel für den Aufbau einer TCP-Verbindung

Im Normalfall kann der Abbau einer TCP-Verbindung von beiden kommunizierenden Anwendungsprozessen 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 seiner gerichteten Verbindung zu seiner Partner-TCP-Instanz und verhindert hierbei den Verlust von noch unquittierten Daten.

*Abbau einer TCP-Verbindung*

Der Abbau wird von einer Seite mit einem TCP-Segment initiiert, in dem das FIN-Flag im Header gesetzt wird (=> <FIN>-Segment bezeichnet). Dies wird von der Gegenseite durch das ACK-Segment 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 Segment, in dem die beiden FIN- und ACK-Flags gesetzt sind, begonnen (=> <FIN,ACK>-Segment). Nach der Bestätigung dieses <FIN+ACK>-Segments durch die Gegenseite wird der Abbauprozeß beendet.

*Maximum Segment Lifetime*

Beim Abbau einer Verbindung tritt u.U. ein zusätzlicher interner Time-Out-Mechanismus in Kraft. Die TCP-Instanz geht in den Zustand Aktive-Close, versendet ein abschließendes ACK und befindet sich dann im Status Time-Wait (=> Abbildung 4.1-3). Dessen Zeitdauer beträgt zweimal die maximale Segment-Lebensdauer (*Maximum Segment Lifetime MSL*), bevor die TCP-Verbindung letztlich geschlossen wird. TCP-Segmente, die länger als die

MSL-Zeit 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 über eine neue ISN für spätere Verbindungen wieder zur Verfügung.

*TCP-API*

Das TCP-Protokokoll verfügt über nur wenige Programmschnittstellen (=> RFC 793), mit denen Applikationen den Auf- und Abbau von Verbindungen sowie die Datenübertragung beeinflussen können. Diese Programmschnittstellen werden *TCP Application Program Interface API* genannt und beinhalten die folgenden Funktionen:

- *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* – Übertragung der Benutzerdaten an den TCP-Sendepuffer und anschließendes Versenden über die TCP-Verbindung. Optional kann das URG- bzw. PSH-Bit gesetzt werden.
- *Receive* – Daten aus dem TCP-Empfangspuffer werden an die Applikation weitergegeben.
- *Close* – Beendet die Verbindung, nachdem zuvor alle ausstehenden Daten aus dem TCP-Empfangspuffer zur Applikation übertragen und ein TCP-Segment mit dem FIN-Bit versandt wurde.
- *Status* – Gibt Statusinformationen über die Verbindung aus, wie z.B. lokaler und entfernter Socket, Größe des Sende- und Empfangsfensters, Zustand der Verbindung und evtl. lokaler Verbindungsname. Diese Informationen können z.B. mittels des Programms *netstat* ausgegeben werden.
- *Abort* – Sofortiges Unterbrechen des Sende- und Empfangsprozesses und Übermittlung des RST-Bits an die Partner-TCP-Instanz.

## 4.2 Flußkontrolle beim Protokoll TCP

Bei der Datenkommunikation entsteht das Problem, daß die Menge der übertragenen Daten an die Aufnahmefähigkeiten des Empfängers angepaßt

werden muß. Die übertragene Datenmenge sollte nicht größer sein als die Datenmenge, die der Empfänger aufnehmen kann. Die Menge der übertragenen Daten muß zwischen den kommunizierenden Rechnern entsprechend abgestimmt werden. Diese Abstimmung bezeichnet man oft als Flußkontrolle (*Flow Control*). Die Flußkontrolle bei der Datenübermittlung über eine TCP-Verbindung erfolgt nach dem Prinzip des Sliding Windows. In Abschnitt 1.7 wurde bereits das Window-Prinzip (*Fensterprinzip*) bei der Numerierung nach dem Modulo-8-Verfahren kurz erläutert. Bevor wir auf die Besonderheiten der Flußkontrolle beim Protokoll TCP eingehen, wollen wir noch das allgemeine Sliding-Window-Prinzip näher veranschaulichen.

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-Größe*.

*Sequenznumerierung*

Mit der Sequenznummer werden die zu sendenden TCP-Segmente fortlaufend numeriert. Die Sequenznummer im TCP-Header eines Datensegments stellt dessen laufende Nummer in der gesendeten Segmentreihe dar. Mit der Quittungsnummer teilt der Empfänger dem Sender mit, welche Sequenznummer als nächste bei ihm erwartet wird. Seitens des Senders stellt die Window-Größe die maximale Anzahl der Datensegmente dar, die der Sender absenden darf, ohne auf eine Quittung vom Empfänger warten zu müssen. Seitens des Empfängers kann die Window-Größe als die maximale Anzahl der Datensegmente gesehen werden, die beim Empfänger immer aufgenommen werden. Wird die maximale Länge von TCP-Datensegmenten festgelegt, so kann die übertragene Datenmenge, bzw. die Menge von Daten unterwegs, mit den erwähnten drei Parametern (Sequenz- und Quittungsnummer sowie Window-Größe) immer kontrolliert werden.

*Flußkontrolle*

Abbildung 4.2-1 veranschaulicht die Flußkontrolle nach dem Sliding-Window-Prinzip mit der Window-Größe = 4. Wie hier ersichtlich ist, läßt sich das Window als ein Sendefenster interpretieren.

> ***Beispiel:*** Betrachten wir zunächst das Beispiel in Abbildung 4.2-1a. Da die Window-Größe 4 beträgt, darf der Sender 4 Datensegmente absenden, ohne auf eine Quittung warten zu müssen. Dies bedeutet, daß er die Segmente mit den Nummern 1, 2, 3 und 4 absenden darf. Nach dem Absenden der ersten drei Segmente ist eine Quittung eingetroffen, mit der das erste Segment quittiert wird. Dadurch verschiebt sich das Fenster (*Window*) mit den zulässigen Sequenznummern um eine Position nach rechts. Da maximal 4 Segmente unterwegs sein dürfen, kann der Sender nun die nächsten beiden Segmente mit

den Nummern 4 und 5 senden. Nach dem Absenden des Segments mit Sequenznummer 5 wird das Segment mit der Sequenznummer 2 durch den Empfänger positiv quittiert. Dadurch verschiebt sich das Fenster nach rechts um eine Position weiter usw.

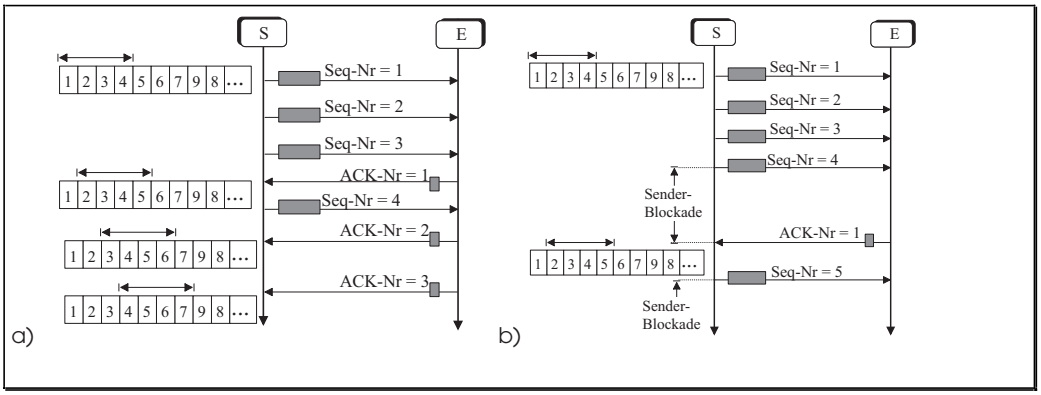


**Abbildung 4.2-1:** Sliding-Window-Prinzip bei Window-Größe = 4:
a) fehlerfreie Übertragung
b) Bedeutung der Sender-Blockade
ACK: Quittungsnummer (Acknowledgement Number),
Seq-Nr.: Sequenznummern; E: Empfänger, S: Sender

*Sende-blockade*

Abbildung 4.2-1b zeigt die Situation, in der der Sendeprozeß blockiert werden muß (*Sendeblockade*). Sie kommt dann oft 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 der Segmente mit Sequenznummer 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 Segmente quittiert werden, darf der Sender keine weiteren Segmente senden. Nach dem Eintreffen der Quittung für das Segment mit Sequenznummer 1 verschiebt sich das Sendefenster um eine Position nach rechts. Das Segment mit der Sequenznummer 5 darf nun gesendet werden. Anschließend muß der Sendevorgang wiederum bis zum Eintreffen der nächsten Quittung blockiert werden.

### 4.2.1 TCP Sliding-Window-Prinzip

Beim Protokoll TCP wird eine modifizierte Version des *Sliding-Window-Prinzips* verwendet. Diese Modifikation besteht darin, daß die Window-Größe in Bytes (und nicht in der Anzahl der Segmente) angegeben wird.

Damit legt die Window-Größe die maximale Anzahl von Bytes fest, die die Sendeseite absenden darf, ohne auf eine Quittung vom Ziel warten zu müssen. Die Interpretation von Window beim Protokoll TCP illustriert Abbildung 4.2-2.

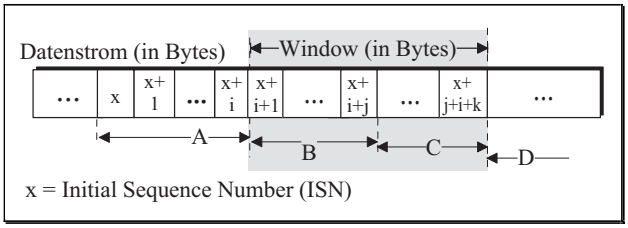


**Abbildung 4.2-2:** Interpretation von Window beim Protokoll TCP

Mit dem Parameter *Window* wird ein Bereich von Nummern markiert, die den zu sendenden Datenbytes zuzuordnen sind. Dieser Bereich kann als Sendefenster gesehen werden. Es sind vier Bereiche *A*, *B*, *C* und *D* im Strom von Datenbytes zu unterscheiden:

- *A*: i Datenbytes, die abgesendet und bereits vom Zielrechner positiv quittiert wurden.
- *B*: j Datenbytes, die abgesendet und vom Zielrechner noch nicht quittiert wurden.
- *C*: k Datenbytes, die noch abgesendet werden dürfen, ohne auf eine Quittung warten zu müssen.
- *D*: Datenbytes außerhalb des Sendefensters. Diese Datenbytes dürfen erst dann abgesendet werden, wenn der Empfang von einigen vorher abgeschickten Daten bestätigt wird.

*Fehlerfreie Datenübermittlung*

Im folgenden wird der Datenaustausch nach dem Protokoll TCP verdeutlicht und damit auch das Sliding-Window-Prinzip näher erläutert. Abbildung 4.2-3 zeigt ein Beispiel für eine fehlerfreie Datenübermittlung.

Die hier dargestellten einzelnen Ereignisse sind wie folgt zu interpretieren:

1: Das erste TCP-Segment von Rechner *A* zu Rechner *B* mit der Sequenznummer SEQ = m. Dieses Segment enthält die ersten 500 Datenbytes. Die *SEQ* = m verweist darauf, daß den einzelnen übertragenen Datenbytes die Nummern m, m+1, ..., m+499 zuzuordnen sind.

2: Von Rechner *B* wird mit einem TCP-Segment, in dem 200 Datenbytes enthalten sind und das ACK-Flag gesetzt wurde, bestätigt, daß das erste Datensegment von Rechner *A* fehlerfrei aufgenommen wurde und das nächste Datenbyte mit der Nummer m+500 erwartet wird. Die Sequenz-

nummer *SEQ* ist SEQ = n, so daß die Nummern n, n+1, ..., n–199 den hier übertragenen Datenbytes zugeordnet werden sollen.

3: Das zweite TCP-Segment von Rechner *A* zu Rechner *B* mit den nächsten 400 Datenbytes und mit der Sequenznummer *SEQ* = m+500. Somit enthält dieses Segment die Datenbytes mit den Nummern m+500, m+501, ..., m+899. Damit wurden bereits 900 Datenbytes beim Rechner *A* abgeschickt und vom Ziel-Rechner *B* noch nicht quittiert. Da die Window-Größe in Richtung zum Rechner *B* 1500 Bytes beträgt, können weiter nur noch 1500–900 = 600 Datenbytes abgesendet werden.

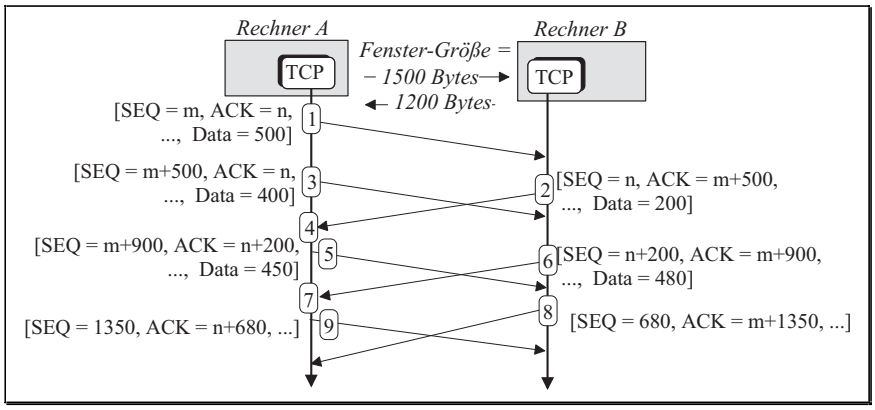


**Abbildung 4.2-3:** Beispiel für den TCP-Ablauf bei der fehlerfreien Datenübermittlung
SEQ: Sequenznummer, ACK: Quittungsnummer

4: Nach dem Empfang dieses TCP-Segments werden 500 Datenbytes vom Zielrechner *B* positiv quittiert. Somit verschiebt sich das Sendefenster im Rechner *A* um 500. Da der Empfang von 400 Bytes (=> das 2-te Segment) noch nicht bestätigt wurde, können noch 1500–400 = 1100 Datenbytes gesendet werden.

5: Das dritte TCP-Segment von Rechner *A* zu Rechner *B* mit der Sequenznummer SEQ = m+900 und mit 450 Datenbytes. Dieses Segment enthält die Datenbytes mit den Nummern von m+900 bis m+1349. Somit sind bereits 850 Datenbytes abgeschickt, die noch nicht quittiert wurden. Darüber hinaus können nur noch weitere 650 (d.h. 1500–850) Datenbytes abgesendet werden. Mit diesem TCP-Datensegment wird dem Rechner *B* auch mitgeteilt, daß die nächsten Datenbytes ab Nummer n+200 erwartet werden.

6: Rechner *B* quittiert mit einem TCP-Segment, in dem das ACK-Flag gesetzt wird, das dritte Datensegment von Rechner *A* und sendet zu Rech-

ner *A* die nächsten 480 Datenbytes. Diesen Datenbytes sind die Nummern n+200, ..., n+679 zuzuordnen.

7: Nach dem Empfang dieses TCP-Segments werden die an Rechner *A* abgeschickten Datenbytes mit den Nummern m+900–1 positiv quittiert. Damit verschiebt sich in Rechner *A* das Sendefenster entsprechend.

8: Rechner *B* bestätigt mit einem TCP-Segment, in dem das ACK-Flag gesetzt wird, die Datenbytes einschließlich bis zur Nummer m+1350–1. Auf diese Weise wurden alle zu Rechner *B* abgeschickten Daten quittiert. Rechner A kann nun zu Rechner *B* die durch die Window-Größe festgelegte Datenmenge (d.h. 1500 Bytes) unmittelbar weitersenden, ohne vorher auf eine positive Quittung von Rechner *B* warten zu müssen.

9: Rechner *A* quittiert positiv Rechner *B* alle Datenbytes bis zur Nummer n+680–1.

*Sende-blockade bei der Daten-übermittlung*

Den Ablauf des Protokolls TCP bei einer fehlerbehafteten Datenübermittlung illustriert Abbildung 4.2-4.

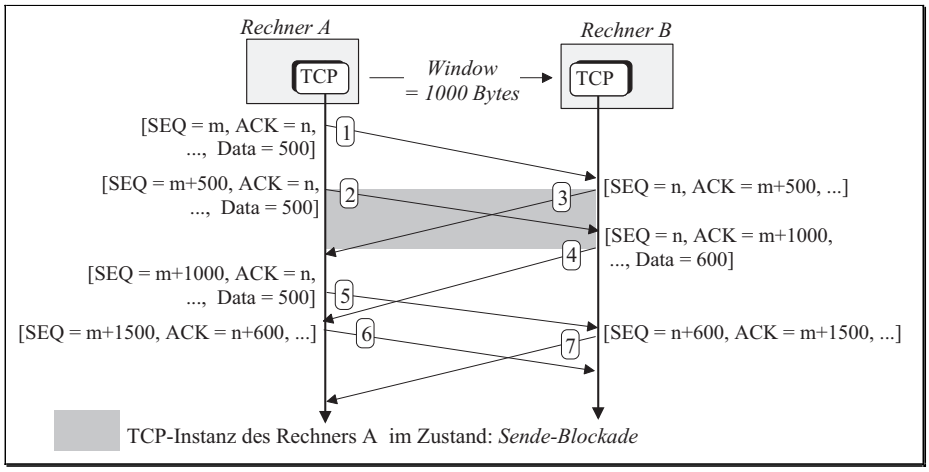


**Abbildung 4.2-4:** Beispiel für den TCP-Ablauf bei einer fehlerbehafteten Datenübermittlung

Die einzelnen Ereignisse sind hier folgendermaßen zu interpretieren:

1: Das erste TCP-Segment von Rechner *A* zu Rechner *B* mit der Sequenznummer SEQ = m. Dieses Segment enthält die ersten 500 Datenbytes und die Sequenznummer SEQ = 500 und verweist darauf, daß diesen Datenbytes die Nummern m, m+1, ..., m+499 zuzuordnen sind. Mit der Quittungsnummer ACK = n wird dem Rechner *B* mitgeteilt, daß das nächste Datenbyte mit der Nummer n von ihm erwartet wird.

2: Das zweite TCP-Segment von Rechner *A* zu Rechner *B* mit den nächsten 500 Datenbytes und mit der Sequenznummer SEQ = m+500. Somit enthält dieses Segment die Datenbytes mit den Nummern m+500, ..., m+999. Mit dem Absenden dieser 200 Datenbytes wurden die Nummern zur Vergabe der zu sendenden Datenbytes „verbraucht“ (=> Window = 1000 Bytes). Aus diesem Grund muß der Sendeprozeß blockiert werden.
3: Es werden 500 Datenbytes vom Zielrechner *B* positiv quittiert. Damit verschiebt sich das Sendefenster in Rechner *A* um 500, so daß die weiteren 500 Datenbytes gesendet werden dürfen.
4: Rechner *B* sendet 600 Datenbytes und quittiert Rechner *A* alle Datenbytes bis zur Nummer m+1000–1 einschließlich.
5: Das dritte TCP-Segment von Rechner *A* zu Rechner *B* mit den nächsten 500 Datenbytes und mit der Sequenznummer SEQ = m+1000.
6: Rechner *A* quittiert Rechner *A* alle Datenbytes bis einschließlich Nummer n+600–1.
7: Rechner *B* quittiert Rechner *A* alle Datenbytes bis einschließlich Nummer m+1500–1.

*Fehlerbehaftete Datenübermittlung*

Da IP zu den ungesicherten Protokollen gehört, muß TCP über Mechanismen verfügen, die in der Lage sind, mögliche Fehler (z.B. Verlust von IP-Paketen, Verfälschung der Reihenfolge usw.) zu erkennen und zu beheben. Der Mechanismus der Fehlerkorrektur von TCP ist einfach: Wenn für ein abgesendetes Datensegment nicht innerhalb eines bestimmten Zeitraums eine Bestätigung eingeht, wird die Übertragung des Segments wiederholt. Im Unterschied zu anderen Methoden zur 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, so daß keine wiederholte Übertragung von einzelnen Segmenten direkt veranlaßt werden kann. Der Empfänger muß einfach abwarten, bis das von vornherein festgelegte Zeitlimit (*Maximum Segment Lifetime MSL*) auf der Sendeseite abgelaufen ist und infolgedessen bestimmte Daten nochmals übertragen werden.

*TCP Fehlerkorrektur*

Das Prinzip der Fehlerkorrektur beim Protokoll TCP demonstriert Abbildung 4.2-5. Um die Darstellung zu vereinfachen, wurden hier nur jene Angaben gezeigt, die nötig sind, um das Prinzip der Fehlerkontrolle während der Datenübermittlung zu erläutern.

Wie hier ersichtlich ist, wird der MSL-Timer nach dem Absenden jedes TCP-Segments neu gestartet. Mit diesem Timer wird eine maximale Wartezeit (*Time Out*) auf die Quittung angegeben. Kommt innerhalb dieser festgelegten maximalen Wartezeit keine Quittung an, wird die Übertragung des

betreffenden Segments wiederholt. Darin besteht das eigentliche Prinzip der Fehlerkontrolle beim Protokoll TCP.

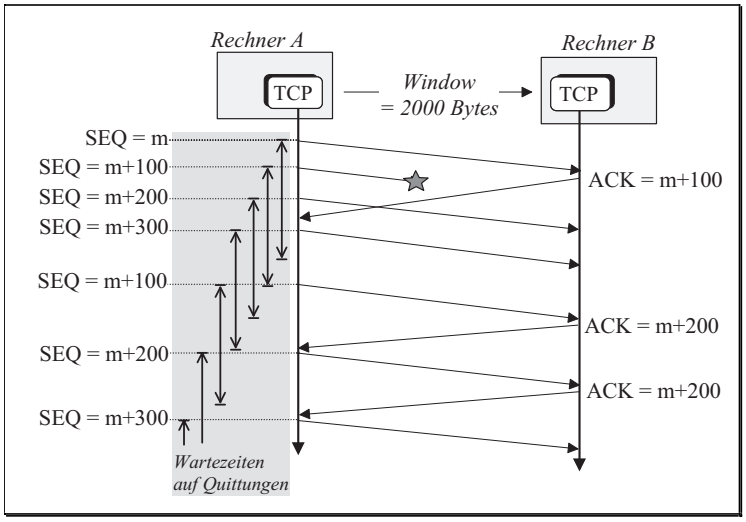


**Abbildung 4.2-5:** Prinzip der Fehlerkorrektur beim TCP
ACK: Quittungsnummer, SEQ: Sequenznummer

Das Segment mit der Sequenznummer m+100 hat den Empfänger nicht erreicht, obwohl die später abgeschickten Segmente (mit den Sequenznummern m+200 und m+300) dort ankamen. Die TCP-Instanz im Rechner B sendet somit keine Bestätigung für den Empfang des Segments mit der Sequenznummer m+200, da die Datenbytes mit den Nummern m+100, ..., m+199 noch nicht empfangen wurden. Das nächste Segment mit SEQ = 300 wird ebenfalls nicht bestätigt. Dies hat zur Folge, daß das Zeitlimit für die Übertragung des Segments mit der Sequenznummer x+100 abläuft und dieses Segment infolgedessen erneut übertragen wird.

*Bestätigung*

Die TCP-Segmente werden von der Empfangsseite nur dann bestätigt, wenn ihre Reihenfolge vollständig ist. Somit kann die Situation eintreten, daß eine Reihe von TCP-Segmenten (sofern die Window-Größe dies zuläßt) sogar dann wiederholt übertragen werden müssen, wenn sie bereits fehlerfrei beim Zielrechner ankamen. Wie dem Beispiel in Abbildung 4.2-5 zu entnehmen ist, betrifft dies die Segmente mit den Sequenznummern m+200 und m+300, die in diesem Fall nochmals zu übertragen sind.

*Round Trip Time*

Die maximale Wartezeit auf die Quittung ist ein wichtiger Parameter des Protokolls 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 Zielrechner auftritt, festgelegt werden. In der Literatur wird diese Zeit als *Round Trip Time* (*RTT*) bezeichnet. In Weitverkehrsnetzen, in denen die Satellitenverbindungen eingesetzt werden, kann es einige Sekunden dauern, bis eine Bestätigung ankommt. Es ist schwierig, die Verzögerungsrate eines Netzes im vorhinein zu wissen.

Im Laufe einer Verbindung können zudem durch Netzbelastung bedingte Schwankungen von RTT auftreten. Daher ist es nicht möglich, einen festen Wert für die maximale Wartezeit auf die Quittung einzustellen. Wenn ein zu kleiner Wert gewählt wurde, läuft die Wartezeit ab, bevor eine Quittung eingehen kann. Infolgedessen wird 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 Übertragungsüberholung stattfinden kann. Der Verlust eines Segments kann in diesem Fall den Datendurchsatz erheblich senken.

### 4.2.2 Implementierungsaspekte von TCP

Das TCP-Protokoll 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 sog. TCP-Stack als Bestandteil der Kommunikations-Software in Betriebssystemen und Routern. Ziel ist es, den TCP-Stack so zu optimieren, daß er unter den heute gegebenen Netzen und Anwendungen eine maximale Performance und eine hohe Übertragungssicherheit gewährleistet (=> RFC 1323, 2001, 2018).

*Klassische TCP-Algorithmen*

Zuvor wollen wir jedoch einige klassische TCP-Problemfälle betrachten (=> RFC 1122):

- *Nagle Algorithmus*
  Der *Nagle Algorithmus* nimmt Bezug auf das Problem, daß die TCP-Instanz auf Anforderung der Anwendungsschicht sehr kleine Segmente sendet. Zur Reduzierung der Netzlast und damit zur Verbesserung des Durchsatzes sollten die pro Verbindung von der Anwendungsschicht ankommenden Daten möglichst konkatiniert, d.h. in einem Segment zusammen gesendet werden. Dies hat nicht nur zur Folge, daß auf diese Weise der Protokoll-Overhead verringert wird, sondern daß die TCP-Empfänger-Instanz 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 unterliegenden Netz (z.B. LAN oder WAN) => *Round Trip Time*.
  - 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 Syndrome*
  Das *Silly Window Syndrome* (*SWS*) kennzeichnet den Zustand, wenn ein TCP-Empfänger sukzessive mit der Erhöhung des zunächst kleinen internen TCP-Puffers dies der sendenden TCP-Instanz durch ein weiteres ACK-Segment mit der neuen *Windowsize* umgehend mitteilt. Hierdurch kann es bei der Übertragung großer Datenmengen vorkommen, daß sich Sender und Empfänger hinsichtlich der *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 wartet, bis er hinlänglich TCP-Puffer allokieren kann.
  Es ist zu beachten, daß dies ein zum *Nagle Algorithmus* komplementärer Effekt ist.
- *Zero Window Probe*
  Ist eine TCP-Verbindung etabliert (=> 4.1-3), kann eine TCP-Instanz der anderen durch Setzen der Windowsize auf 0 mitteilen, daß es seinen TCP-Empfangspuffer auf null reduziert hat. Dies kann z.B. eine Folge davon sein, daß die TCP-Instanz die bereits anstehenden Daten nicht mehr an die Applikation weiterreichen kann, was typischerweise der Fall ist, wenn sich in einem Netzwerk-Drucker kein Papier mehr befindet. In Anschluß daran ist es nach Ablauf des *Timeouts* Aufgabe des Senders, mit einer *Zero Window Probe* festzustellen, ob der Empfänger wieder aufnahmebereit ist. Hintergrund hierfür ist, daß ACK-Segmente ohne Daten nicht verläßlich übertragen werden. Sollte der Empfänger hierauf nicht antworten, wird der *Retransmission-Algorithmus* in Gang gesetzt.
- *TCP Keep-Alives*
  TCP verzichtet in der Regel auf ein sog. *Keep-Alive Verfahren*, ohne ein solches jedoch ausdrücklich auszuschließen. TCP-Keep-Alive-Informationen werden in ACK-Segmenten mit einem bedeutungsfreien Daten-

byte oder völlig ohne Daten eingeschlossen. Sie dürfen aber nur versendet werden, wenn keine anderen regulären Daten zwischen den TCP-Instanzen ausgetauscht werden. Entscheidend ist, daß die generierte Sequenz-Nummer (=> Abbildung 4.2-2) dem obersten Wert des Sendefensters abzüglich einem Byte entspricht. Dieser Wert liegt außerhalb des ausgehandelten Sendefensters, was den TCP-Partner veranlaßt, mit einem ACK zu antworten.

Das Einsatzgebiet von TCP hat sich mit der Erweiterung des Internets und den Entwicklungen der lokalen Netze stark erweitert. Folgende Aspekte spielen dabei eine zentrale Rolle:

- Schnelle, d.h. durchsatzstarke und latenzzeitarme LANs, wie z.B. Gigabit-Ethernet bzw. geswitchte Fast-Ethernet Netze.
- Ausgedehnte, große Netzstrukturen (WAN) mit unterschiedlichen Trägernetzen wie ATM oder Frame-Relay mit z.T. signifikanten Verzögerungszeiten bei der Übertragung sog. *Long Fat Networks* (*LFN*).
- Zu übertragende Datenvolumen, die im Bereich von Gigabyte liegen, d.h. außerhalb den Bereich der einfach-adressierbaren TCP-Sequenznummern überschreiten.

*Round Trip Time Measurement*

Eine konzeptionelle Eigenheit von TCP besteht darin, auch ohne positives ACK nach einer gewissen Zeit (dem Timeout) die Daten erneut zu versenden. Hierzu bedient sich TCP einer Abschätzung der sog. *Round Trip Time* (*RTT*). Diese Abschätzung sollte möglichst präzise erfolgen, da dies speziell für die oben bezeichneten LFNs den maximalen Durchsatz bestimmt. Drei unterschiedliche Methoden sind sowohl in Endstationen wie in Routern gebräuchlich:

- *Originalimplementierung*
  Sieht vor, die RTT für jedes einzelnen TCP-Paket zu ermitteln und hieraus ein gewichtetes Mittel rekursiv berechnen:

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

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

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

  Für die Parameter wird üblicherweise $a = 0{,}9$ und für $b = 2$ angenommen.
  Problematisch an diesem Ansatz ist, daß sich einerseits ein „verlorenge-

gangenes“ Paket in einer Unterschätzung des RTT auswirkt und damit andererseits keine Korrelation mit den ACKs des Empfängers gegeben ist (=> Abbildung 4.2-6).

- *Karn/Partridge-Implementierung*
  Umgeht die letzte Einschränkung, da erst das ACK den Datenempfang bestätigt, wobei wiederholte TCP-Pakete nicht in den Algorithmus einbezogen werden. Zudem wird die Timeout-Zeit nach jedem Empfang angehoben:

  $$Timeout = 2 * Timeout$$

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

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

  Hierbei beträgt der Wert $g_0 = 0{,}125$. Auch wird die Berechnung des Timeouts angepaßt. Zunächst wird eine Hilfsgröße definiert:

  $$Abweichung = Abweichung + g_1 * \delta(RTT)$$

  mit $g_1 = 0{,}25$. Somit wird der neue Timeout berechnet nach:

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

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

*Congestion Control*

Das Erkennen und Beheben möglicher Durchsatzprobleme (*Congestion*) beinhaltet mehrere Aspekte, die den Aufbau der Verbindung, ihre Unterhaltung und die Reaktion auf Fehler betreffen: *Slow Start*, *Congestion Avoidance*, *Fast Retransmit/Fast Recovery* sowie *Selective Acknowledgements* (*SACK*).

- *TCP Slow Start*
  sagt aus, daß eine TCP-Instanz die Datenübermittlung von TCP-Segmenten mit einem kleinen sog. *Congestion Window cwnd* beginnt, i.d.R. *cwnd = 1 Segment* (Defaultwert: 536 Byte, => RFC 1122). Anschließend wird *cwnd* mit jedem empfangenen ACK-Segment exponentiell vergrößert, d.h. *cwnd = cwnd * cwnd* und die Anzahl der übertragenen TCP-Segmente 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.

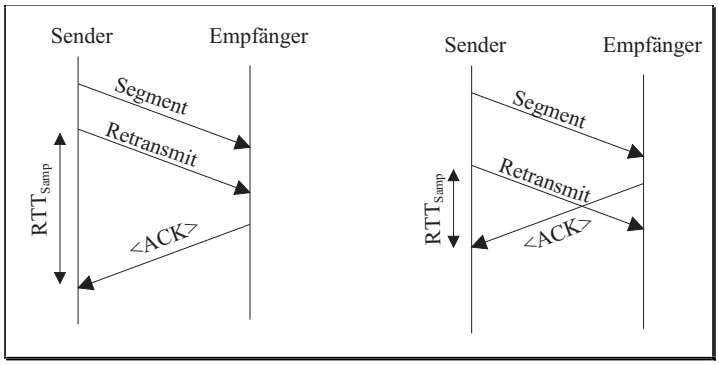


**Abbildung 4.2-6:** Abschätzung von RTT nach
a) original TCP-Implementierung
b) Karn/Partridge-Algorithmus

- *Congestion Avoidance*
  geht von der Annahme aus, daß Datenpakete in heutigen Netzen kaum mehr verlorengehen, sondern im Falle von *Timeouts* und doppelt empfangener ACKs (sog. *dACKs*) eine Überlast im Netzwerk aufgetreten ist. Eine TCP-Instanz kann dem vorbeugen, indem sie neben der Congestion Window *cwnd* eine Variable *Slow Start Threshold ssthresh* nach folgendem Schema nutzt:

  *1.* Initialisierung:
  *cwnd = 1 Segment, ssthresh = max. Windowsize (64 kByte)*
  *2.* Maximale zu sendende Datenmenge:
  *min (cwnd, advWin)*
  *3.* Beim Empfang von *dACKs* wird *ssthresh* neu berechnet:
  *ssthresh = max (2, min(cwnd/2,advWin))*
  Falls Timeouts registriert werden, gilt:
  *cwnd = 1*
  *4.* Beim Empfang neuer ACKs
  wird *cwnd* nach Slow Start oder nach Congestion Avoidance wieder erhöht.

- *Fast Retransmit/Fast Recovery*
  geht davon aus, daß mehrere empfangene *dACKs* den Verlust lediglich *eines* TCP-Segment bedeuten, welches neu gesendet werden muß. Falls die Anzahl der *dACKs* einen Schwellwert 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 Starts* eingesetzt.

- *Selective Acknowledgement*
  Mit dem in RFC 2018 vorgestellten *Selective Acknowledgement* (*SACK*)-Verfahren wird dem Problem begegnet, daß mit hoher Wahrscheinlichkeit beim Registrieren von kumulativen *dACKs* nur ein Paket neu übertragen werden muß. Der *Fast Retransmit* würde hingegen mehrere u.U. fehlerfrei – aber mit Verzögerung – empfangene TCP-Segmente wiederholen. Um dieser Situation zu entgehen, muß der Empfänger dem Sender in einem ACK-Segment den Beginn und das Ende derjenigen Datenblökke mitteilen, die er als letzte zusammenhängend in seinem Empfangsfenster (Datenpuffer) verarbeitet hat.
  Die SACK-Option erweitert das Optionsfeld um bis zu maximal 40 Bytes. In Abbildung 4.2-7 ist dargestellt, daß höchstens 4 Blöcke gebildet werden können, in denen Informationen über vier unterschiedliche Pufferbereiche einfließen. Wird zusätzlich die *Timestamp* Option eingesetzt, sind lediglich 3 Blöcke möglich.

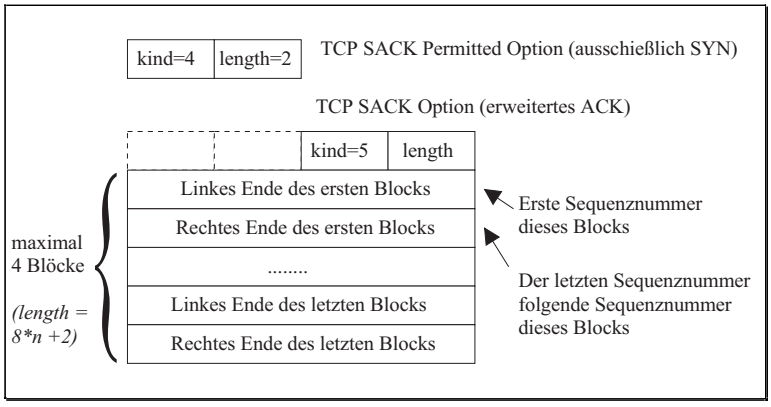


**Abbildung 4.2-7:** Aufbau des SACK-Optionsfelds:
a) für SYN-Segmente beim Aushandeln der Option
b) bei der Übermittlung fehlerhafter oder fehlender Daten in ACK-Segmenten

- *Protection Against Wrapped Sequence Number*
  Ein Problem, das bei der Übertragung großer Datenmengen auftritt, ist der Überlauf des Sequenzzählers SEQ (Sequence Number) für die TCP-Segmente. Wie in Abbildung 4.1-2 dargestellt, ist SEQ ein 32-Bit-Wert, mit dem TCP-Segmente von maximal 4 GByte adressieren kann. Ist die Datenmenge größer, muß der Zähler neu von 1 initialisiert werden. Wie

soll die TCP-Instanz entscheiden, ob ein evtl. verlorengegangenes und zu wiederholendes TCP-Segment aus der aktuellen „Runde“ oder aus einer früheren stammt? Die Lösung hierfür ist die in Abschnitt 4.1.1 vorgestellte Timestamp-Option. Zusätzlich zu SEQ enthält das TCP-Segment eine monotone steigende 4 Byte große Zeitinformation vom Kommunikationspartner, dessen jeweils aktueller Wert zu speichern ist. Ist das Zeitintervall eines „Uhrtick“ nun 1 ms, reicht dies aus, die Datenübertragung über nahezu 25 Tage zu monitoren. Durch den Vergleich des Timestamps eines alten und evtl. wiederholten Segments mit der aktuell entgegen genommenen Zeitmarke, kann ersteres getrost verworfen werden.

### 4.2.3 Transaction TCP T/TCP

Eine Weiterentwicklung und Ergänzung von TCP liegt (=> RFC 1644) in Form des Protokolls Transaction TCP (T/TCP) vor. Wie wir bereits dargestellt haben, arbeitet TCP symmetrisch zwischen den Kommunikationspartnern und stellt den Anwendungen eine virtuelle Verbindung zur Verfügung. Viele Anwendungen – wie z.B. das in Abschnitt 2.3.4 dargestellte HTTP – besitzen jedoch einen asymmetrischen Kommunikationsablauf: auf einen Request (kleine Anfrage) erfolgt ein Response (umfangreiche Antwort). Dieses Frage/Antwort-Schema soll als Transaktion bezeichnet werden.

*Transaktionen*

Eine virtuelle TCP-Verbindung ist durch die Phasen Verbindungsaufbau (*Three Way Handshake,* 3WHS), Datentransfer und Verbindungsabbau gekennzeichnet. Dies stellt einen erheblichen Protokolloverhead dar. Wie schon bei der Vorstellung von HTTP diskutiert, sind zwei Szenarien möglich:

1. Die Requests werden von der TCP-Instanz als unabhängig betrachtet, d.h. es wird jeweils eine neue TCP-Verbindung auf- und abgebaut. Dies entspricht dem Verhalten von HTTP 1.0.
2. Die Applikation nutzt eine persistente TCP-Verbindung, d.h. in einem Kommando-Streaming-Modus werden mehrere Antworten/Anfragen über eine bestehende TCP-Verbindung abgewickelt, vgl. von HTTP 1.1.

*Transmission Control Blocks*

Beide Verfahren sind nicht optimal. Im ersten Fall treten für jeden Request Verzögerungen aufgrund des 3WHS auf. Die sequentielle Abarbeitung der Anfragen und Antworten wird der TCP-Instanz aufgebürdet, indem sie jeweils eine neue virtuelle Verbindung aufbaut. Hierdurch werden entsprechend viele TCP-Puffer (Transmission Control Blocks TCB) verbraucht, die

im Fall des Verbindungsabbaus unter Umständen sehr lange im Time-Wait-Zustand hängen.

Im zweiten Fall ist es Aufgabe der Server-Applikation, zu entscheiden, ob ein verzögert einlaufender Request eine wirkliche Neuanfrage ist – die es erneut zu bearbeiten gilt – oder ob es sich lediglich um die Wiederholung eines TCP-Segments handelt (*duplicate request*), der noch über den Cache der TCBs bedient werden kann.

*T/TCP-Erweiterungen*

T/TCP bietet für die effiziente Abwicklung von Transaktionsprozessen zwei TCP-Erweiterungen:

- Einführung eines *Connection Counts CC*, d.h. eines unabhängigen Transaktionszählers für jeden Request bzw. Response, der als Option in jedem TCP-Segment mitgegeben wird. Dieser Transaktionszähler wird bei jedem TCP-Open vom Sender monoton hochgezählt.
- Nutzung des *TCP Accelerated Open* (*TAO*), d.h. der Möglichkeit, den 3WHS zu umgehen, indem die Partner als Bestandteil des TCB den letzten Wert des empfangenen und gesendeten CC mitführen. Beim Empfang eines initialen <SYN>-Segments mit einem CC-Wert höher als dem im Cache befindlichen, geht die Empfänger-TCP-Instanz sofort in die Datentransferphase, da sie sicher sein kann, daß der TCP-Sender sich auf die vorige Transaktion bezieht. Sollte dies nicht der Fall sein, unternimmt der TCP-Empfänger den normalen Three Way Handshake.

Zur Synchronisation des CC-Zählers zwischen Empfänger und Sender werden die zusätzlichen TCP-Optionen (=> Tabelle 4.1-1) *CC.NEW* sowie *CC.ECHO* eingesetzt. Mit der gesetzten CC.NEW-Option im TCP-Segment fordert der Sender den Empfänger auf, mit dem normalen Three Way Handshake zu beginnen, dabei jedoch den in seinem Cache befindlichen Wert von CC durch CC.NEW zu überschreiben. Dies kann z.B. notwendig sein, wenn das 32-Bit-Wertefeld überschritten wurde. Die Option CC.ECHO wird hingegen vom Sender in einem <SYN,ACK>-Segment genutzt, um somit die CC-Werte zwischen beiden T/TCP-Instanzen zu validieren.

T/TCP stützt sich hierbei auf ein kompliziertes Zustandsdiagramm, das neben den in Abbildung 4.1-3 gezeigten noch spezielle TAO-Zustände kennt, die mit dem Empfang bzw. dem Senden der CC-Optionen verknüpft sind. Im besonderen unterscheiden sich die Zustände TAO-Syn-Sent und TAO-Syn-Received von dem des TCP, da ein zusätzliches FIN-Bit im TCP-Segment gesetzt sein muß.

*TAO-Mechanismus*

Abbildung 4.2-8a demonstriert den TAO-Mechanismus unter der Voraussetzung, daß der T/TCP-Empfänger (d.h. der Server) den CC-Wert des Senders noch im Cache hat. In Abbildung 4-2.8b wird schließlich eine einfache

Transaktion gezeigt. Hierbei ist zu beachten, daß die Empfänger-T/TCP-Instanz bereits nach dem einlaufenden ersten TCP-Segment mit gesetztem FIN-Bit in den Zustand Close-Wait geht, dies über das ausgesendete Segment quittiert und unmittelbar nach Empfang des <ACK,CC>-Segments den Port schließt.

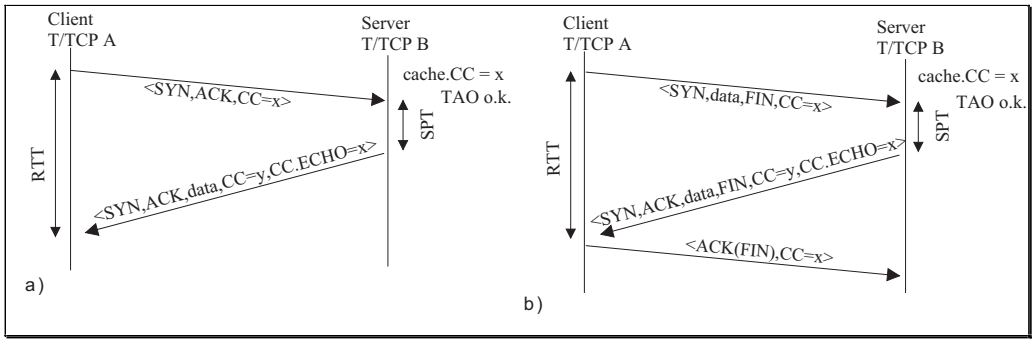


**Abbildung 4.2-8:** T/TCP-Mechanismus:
a) TAO-Verfahren mit initaler CC-Nummer x
b) einfaches Transaktionsmuster mit initialem FIN-Bit im TCP-Segment
RTT: TCP Round Trip Time, SPT: Server Processing Time

## 4.3 Aufbau und Arbeitsweise von UDP

Bei *UDP (User Datagram Protocol)* handelt es sich um den einfachsten verbindungslosen Dienst, der auf IP aufsetzt. Durch UDP können Anwendungen Datagramme (selbständige Datenblöcke) senden und empfangen. UDP bietet – ähnlich dem IP – keine gesicherte Übertragung und keine Flußkontrolle. Ob die IP-Pakete auch wirklich beim Empfänger landen, ist nicht sichergestellt. UDP ist wie TCP der Schicht 4 zugeordnet, was für einen Anwenderprozeß bedeutet, daß er entweder mit TCP oder UDP Daten empfangen bzw. senden kann.

Den Aufbau von UDP-Paketen zeigt Abbildung 4.3-1. Der IP-Header und Teile des UDP-Header werden auch häufig zusammengefaßt und als UDP *Pseudo-Header* bezeichnet. Im Gegensatz zum TCP erfordert das UPD-Protokoll (=> RFC 768) nicht notwendigerweise die Berechnung einer Prüfsumme; auch wenn dies aktuelle UDP-Implementierungen in der Regel leisten.

*UDP-Aufgaben*

Die Bedeutung von UDP liegt in seinem Transportdienst für andere wichtige Internet-Protokolle. Hierzu zählen u.a.:

- Trivial File Transfer Protocol (TFTP),
- Remote Procedure Calls (RPC),
- Network File System (NFS),
- Domain Name Services (DNS),
- Simple Network Management Protocol (SNMP),
- BOOT Protocol,
- Lightweight Directroy Access Protocol (LDAP),
- TIME und DAYTIME Nachrichten sowie
- das Versenden von Broadcast-Nachrichten.

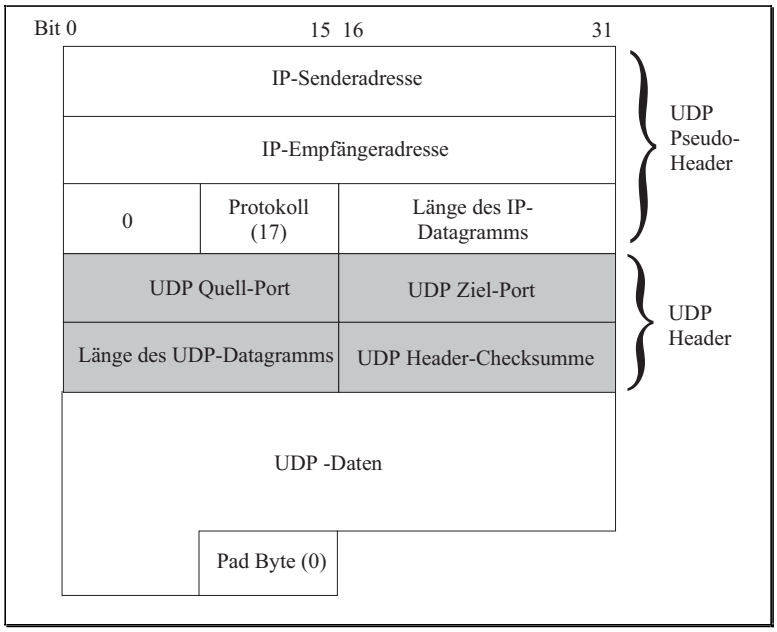


**Abbildung 4.3-1:** Aufbau von UDP-Paketen