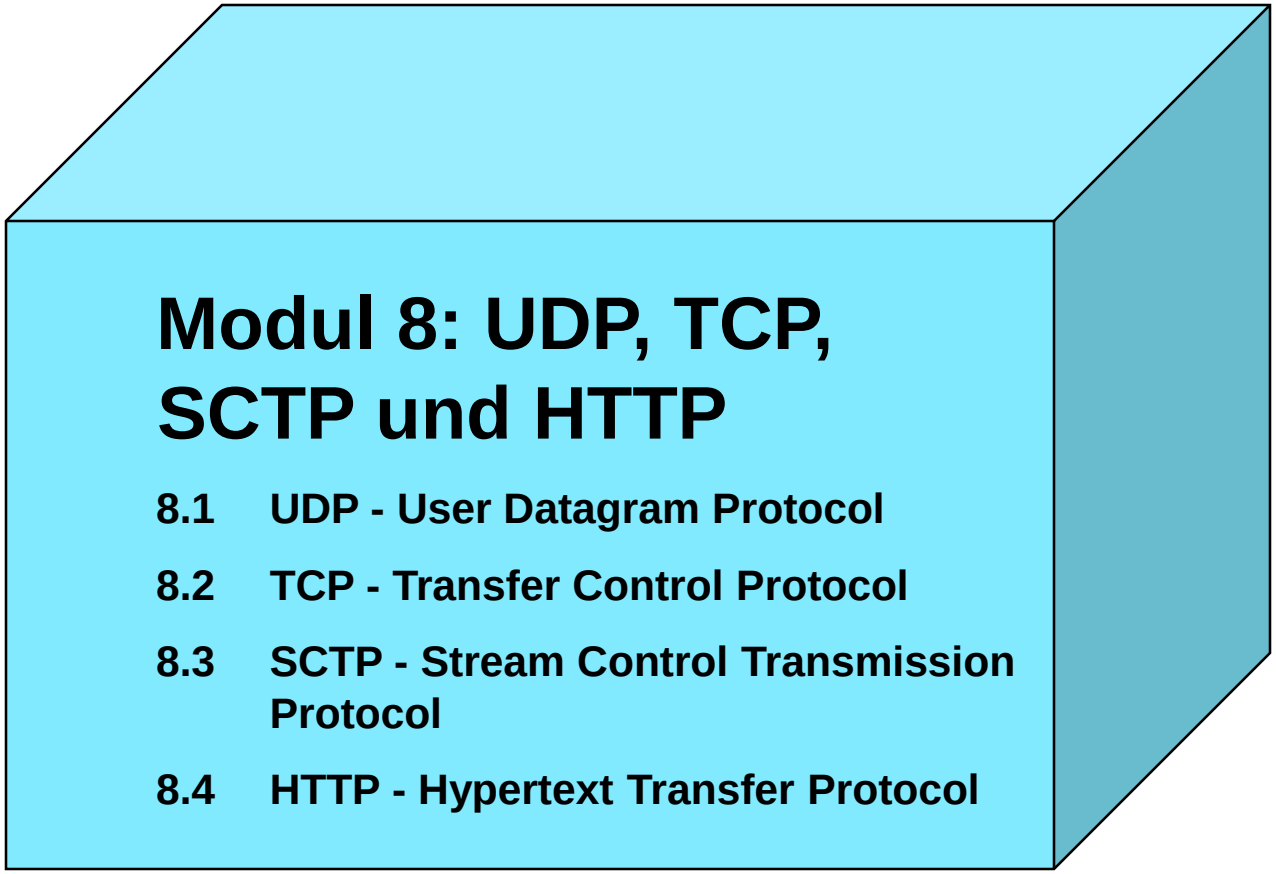


Modul 8: UDP, TCP, SCTP und HTTP

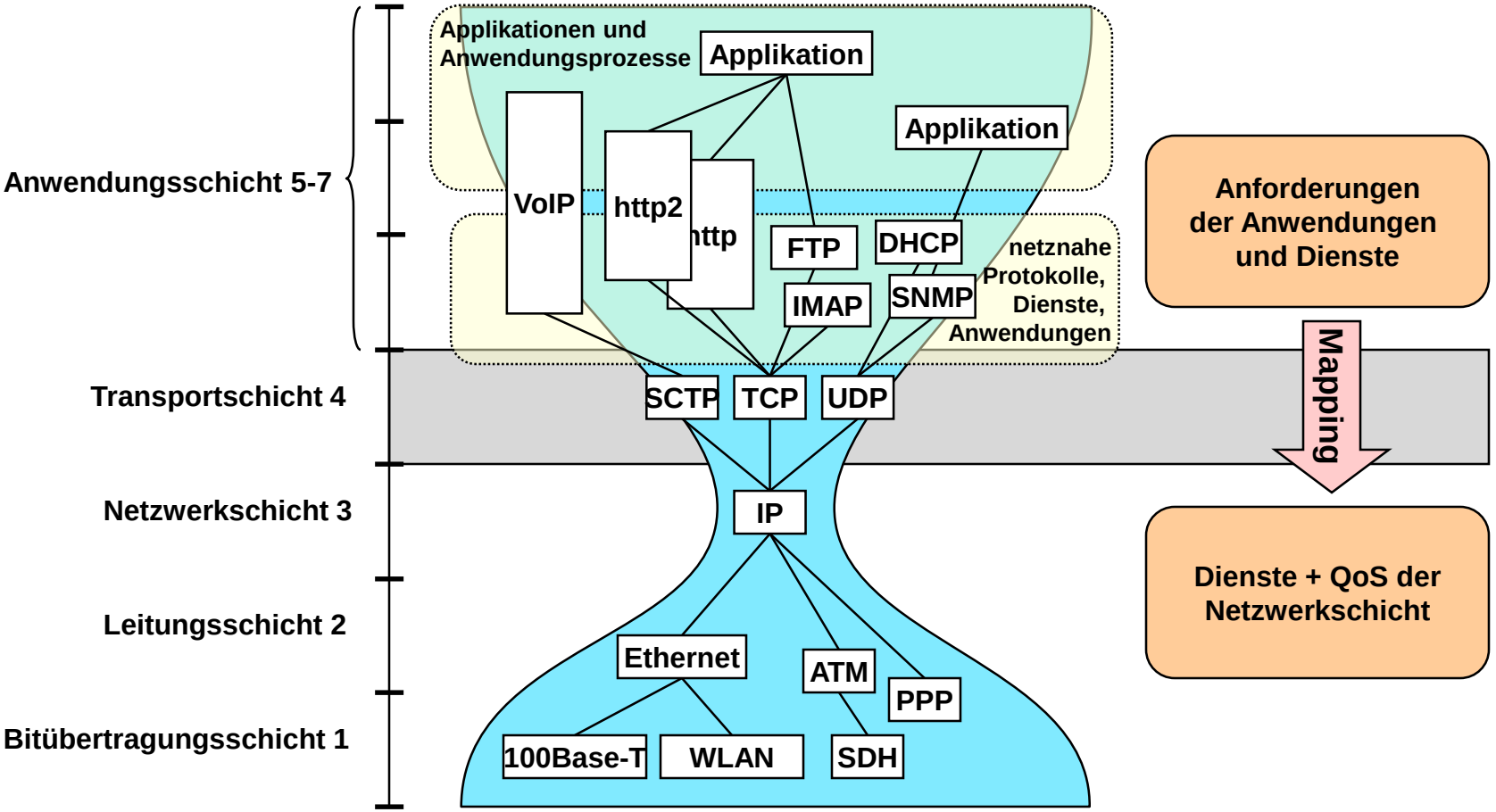
8.1 UDP - User Datagram Protocol

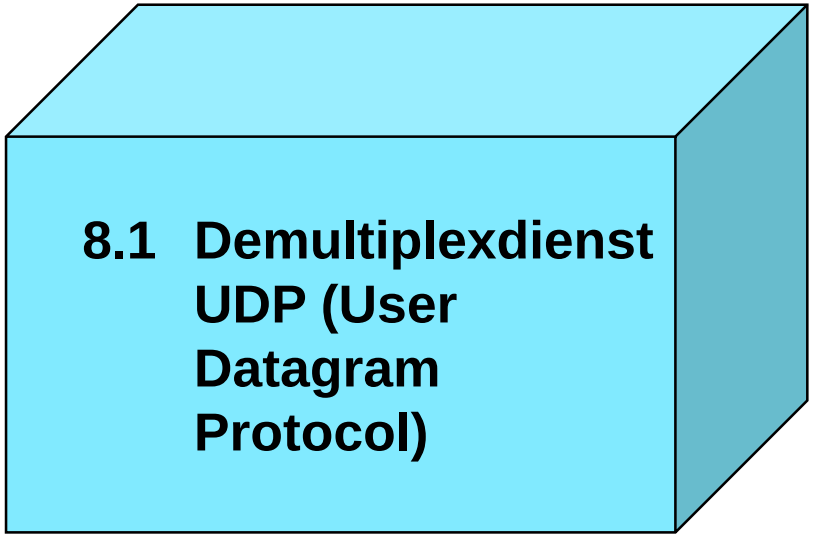
8.2 TCP - Transfer Control Protocol

8.3 SCTP - Stream Control Transmission Protocol

8.4 HTTP - Hypertext Transfer Protocol

End-to-End Protokolle: Einordnung + Aufgabenstellung



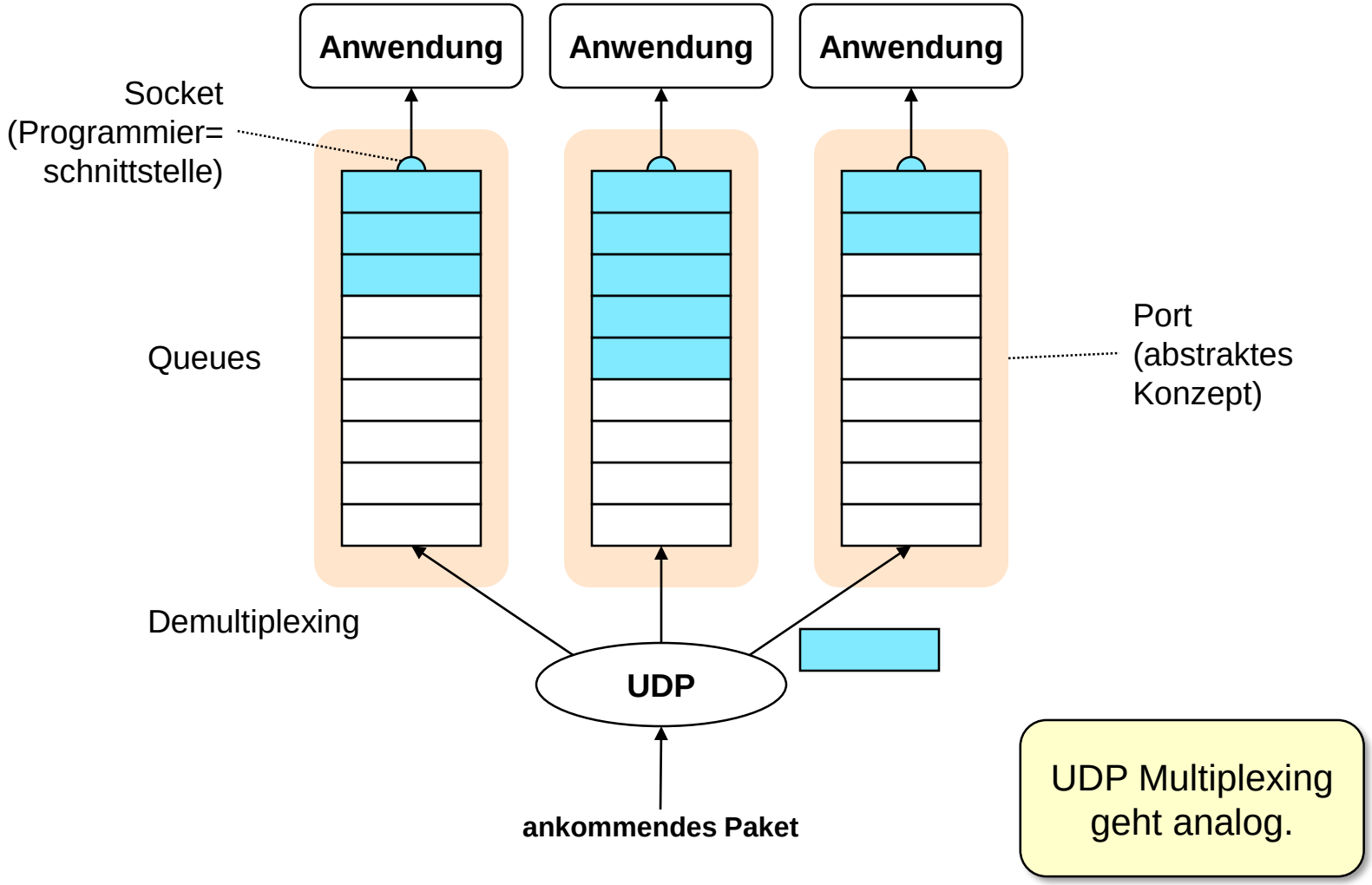


8.1 Demultiplexdienst UDP (User Datagram Protocol)

Lernziele:

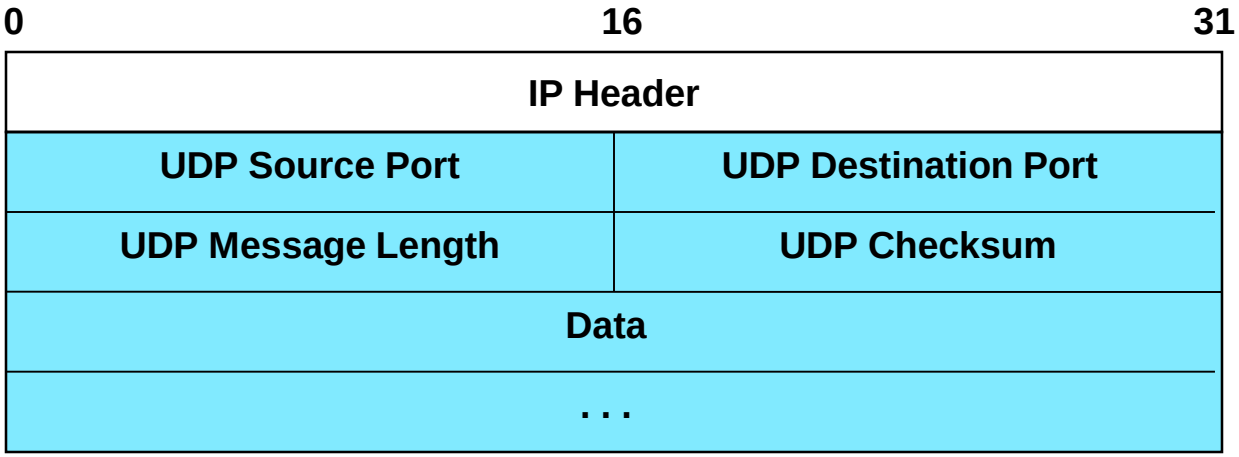
Nach Durcharbeiten dieses Teilkapitels sollen Sie das Grundkonzept von UDP darstellen können.

UDP Demultiplexing (Ports und Sockets)



UDP Multiplexing geht analog.

UDP Header

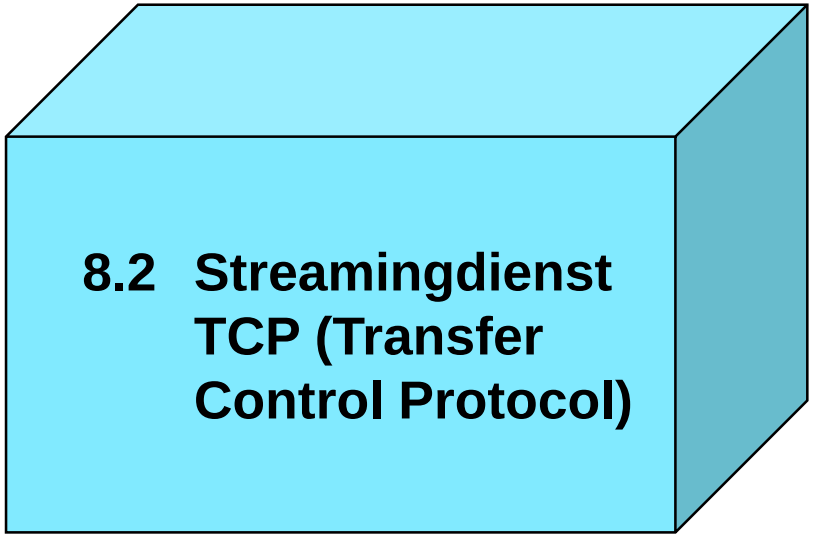


Source Port: Enthält den Port des versendenden Hosts; optionale Angabe, d.h. wenn angegeben, ist dies der Port, an den eine evtl. Antwort gehen soll

Destination Port: Zielport, wohin das Datagramm geschickt wird

Message Length: Gibt die Gesamtlänge des Datagramms an

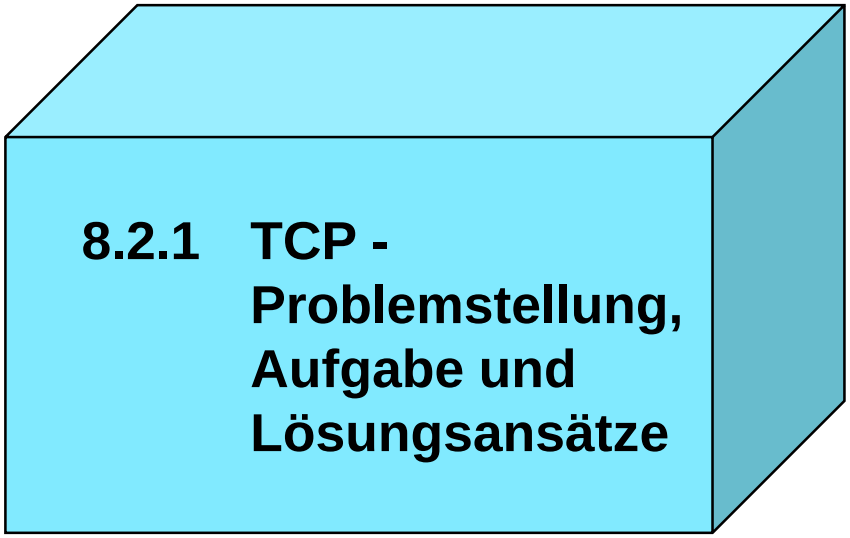
Checksum: Prüfsumme über UDP-Header sowie IP-Pseudoheader (Protokollnummer+IP-Adressen)
Stellt sicher, dass der richtige Port und der richtige Host erreicht worden sind.



8.2 Streamingdienst TCP (Transfer Control Protocol)

Lernziele:

Nach Durcharbeiten dieses Teilkapitels sollen Sie Grundelemente, Grundkonzepte und grundlegende Funktionen von TCP darstellen können.



Problemstellung: Zuverlässige Transportleistung über unzuverlässige Netzwerkschicht

Eigenschaften der IP-Netzwerkschicht:

- Aufteilung der Information in Pakete
- Maximale Paketgrößen
- Paketverluste bei der Übertragung
- Empfang fehlerhafter Pakete
- Mehrfaches Übertragen von Paketen
- Umordnung der Paketreihenfolge
- Wechselnde Paketübertragungszeiten
- Speicherfähigkeit des Netzes

IP ist (nur) paketorientiertes Best-Effort-Netz

Was soll TCP leisten?

TCP stellt einen zuverlässigen bidirektionalen Bytestrom zur Verfügung

Für TCP gibt es viel zu tun

Überblick TCP

Grundeigenschaften:

wichtige Folie!

- **Punkt-zu-Punkt-Verbindung**
- **Streaming-Schnittstelle**
  - **Byteorientiert**
  - **keine Fragment-/Segmentgrenzen**
- **Zuverlässige, verbindungsorientierte Übertragung**
  - **Drei Phasen: Verbindungsaufbau, Nutzdatenübertragung, Verbindungsabbau**
  - **Zuverlässiger Verbindungsaufbau, Initialisierung des Empfänger/Sender-Kontexts (→ *Three-Way-Handshake*)**
  - **Zuverlässige Nutzdatenübertragung (→ *Retransmission*)**
  - **zuverlässiger Verbindungsabbau**
- **Eine Vollduplex-Verbindung**
  - **bidirektionaler Datenfluss in derselben Verbindung**
  - **Optimale maximale Segmentgröße (512-1500 Bit) beachten**
- **Flusskontrolle und Überlastkontrolle**

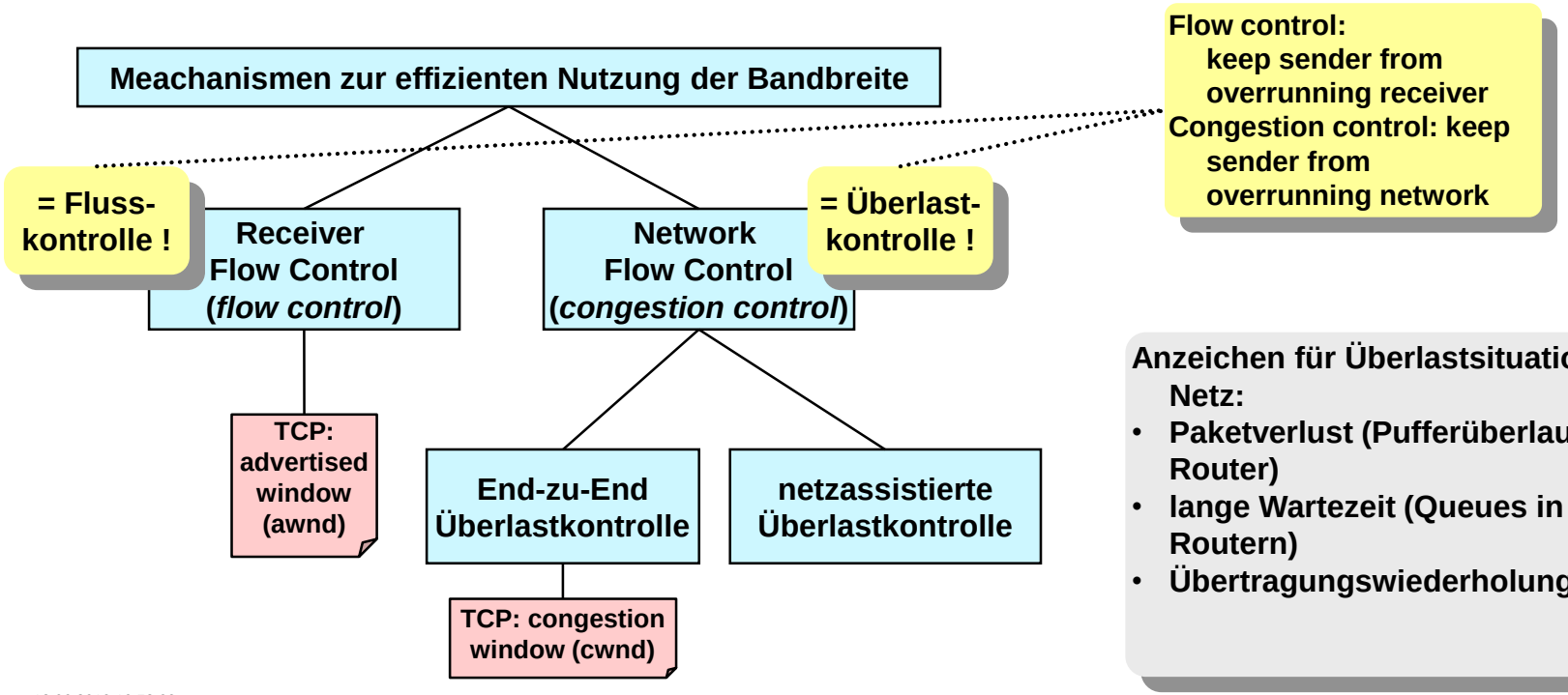
Begriffsklärung: Fluss- und Überlastkontrolle

Ziel der Kontrollmechanismen:

Effiziente Nutzung der Bandbreite eines verbindungslosen Netzes durch Sender und Empfänger .

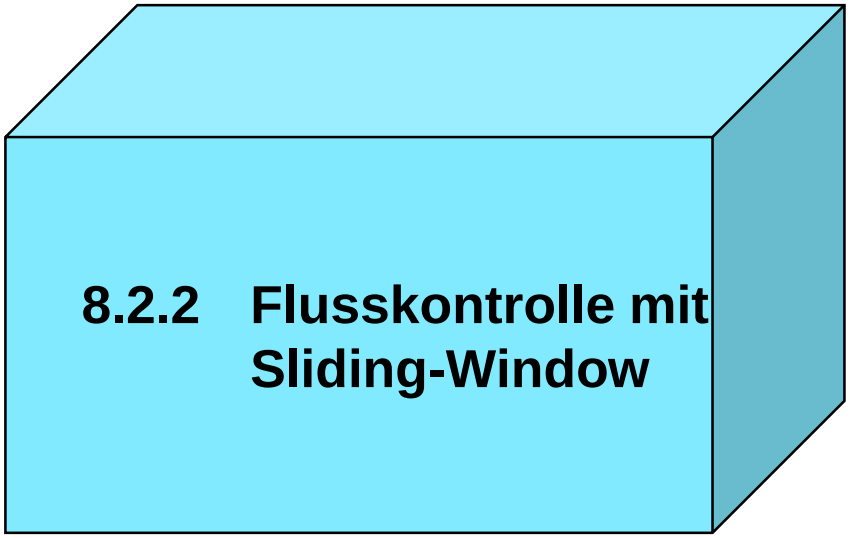
Fenster-Flusskontrolle:

Kontrollmechanismen, die auf Fenstergrößen (=Puffergrößen) beruhen.

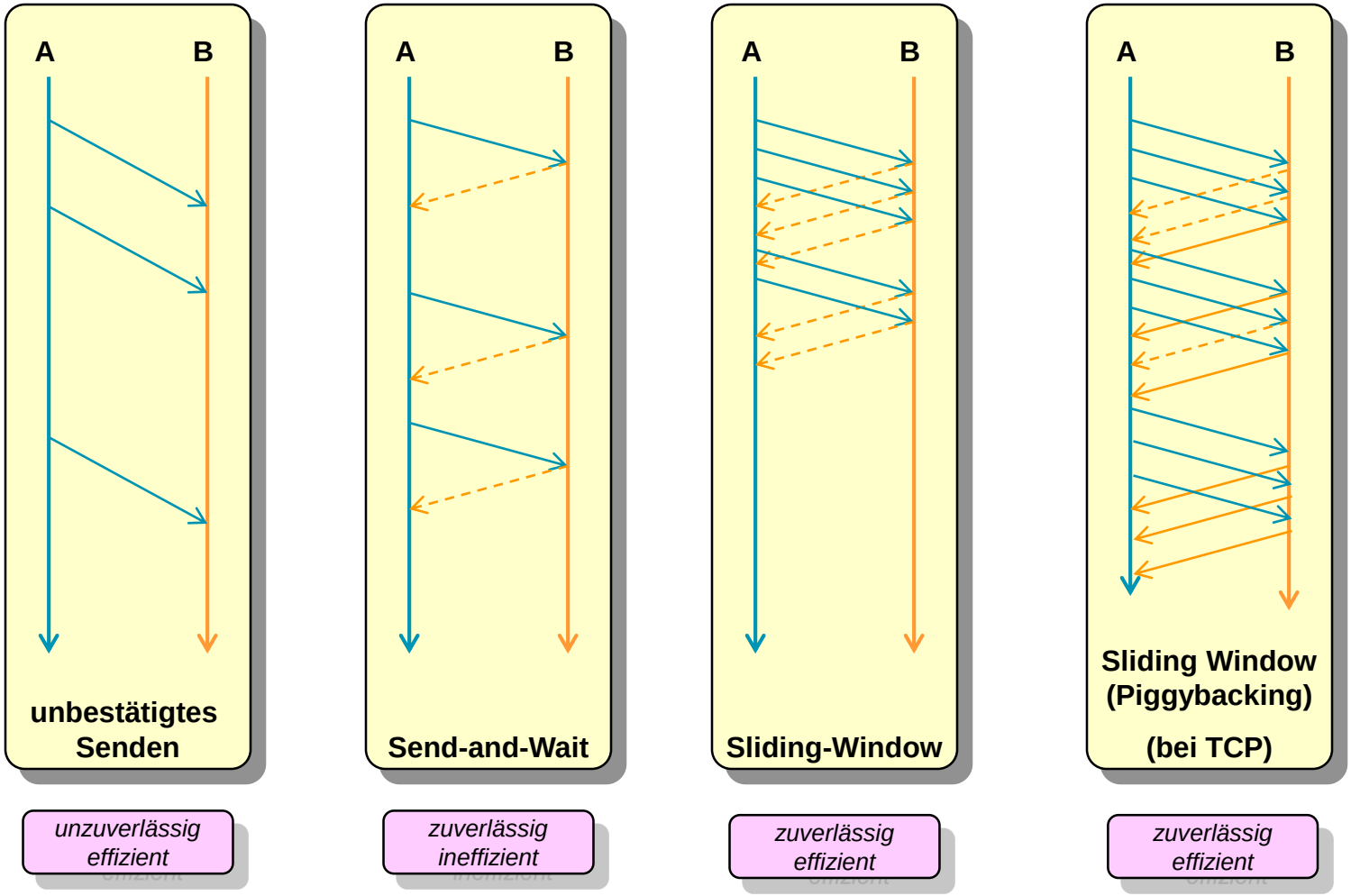


Anzeichen für Überlastsituationen im Netz:

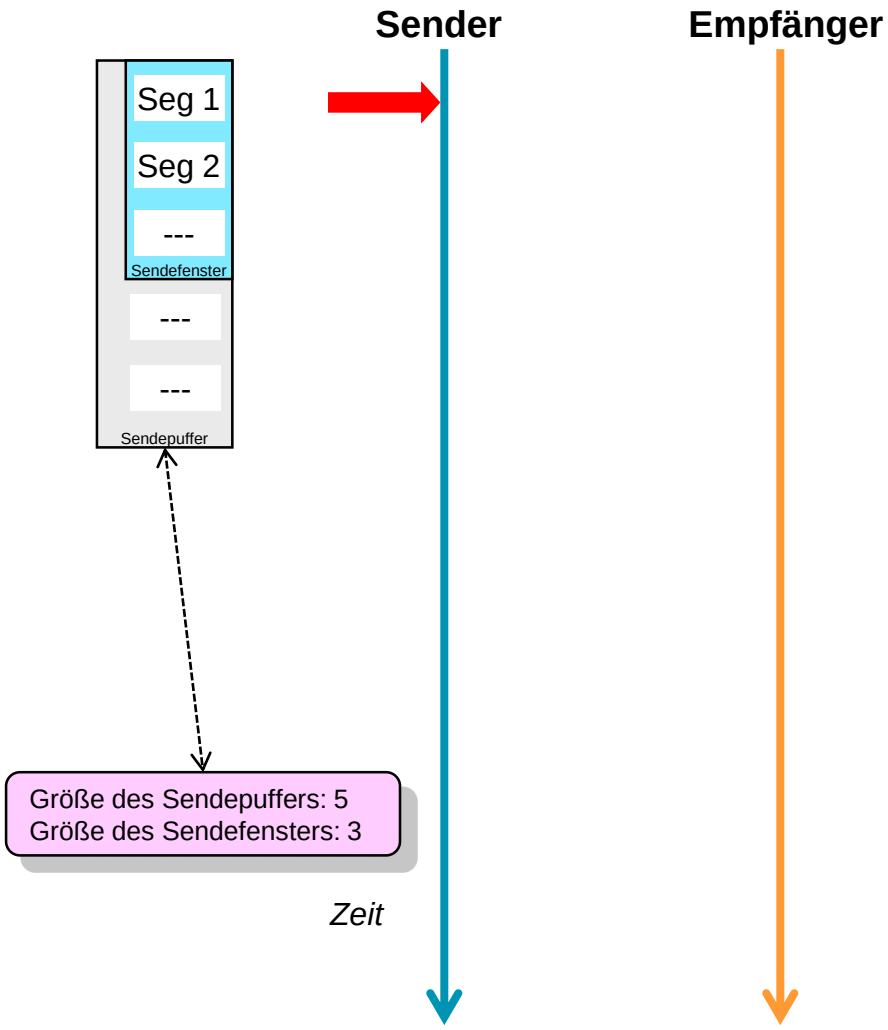
- **Paketverlust (Pufferüberlauf Router)**
- **lange Wartezeit (Queues in den Routern)**
- **Übertragungswiederholungen**



Prinzip des Sliding-Window: Zuverlässigkeit + Effizienz



Sliding-Window - detaillierter

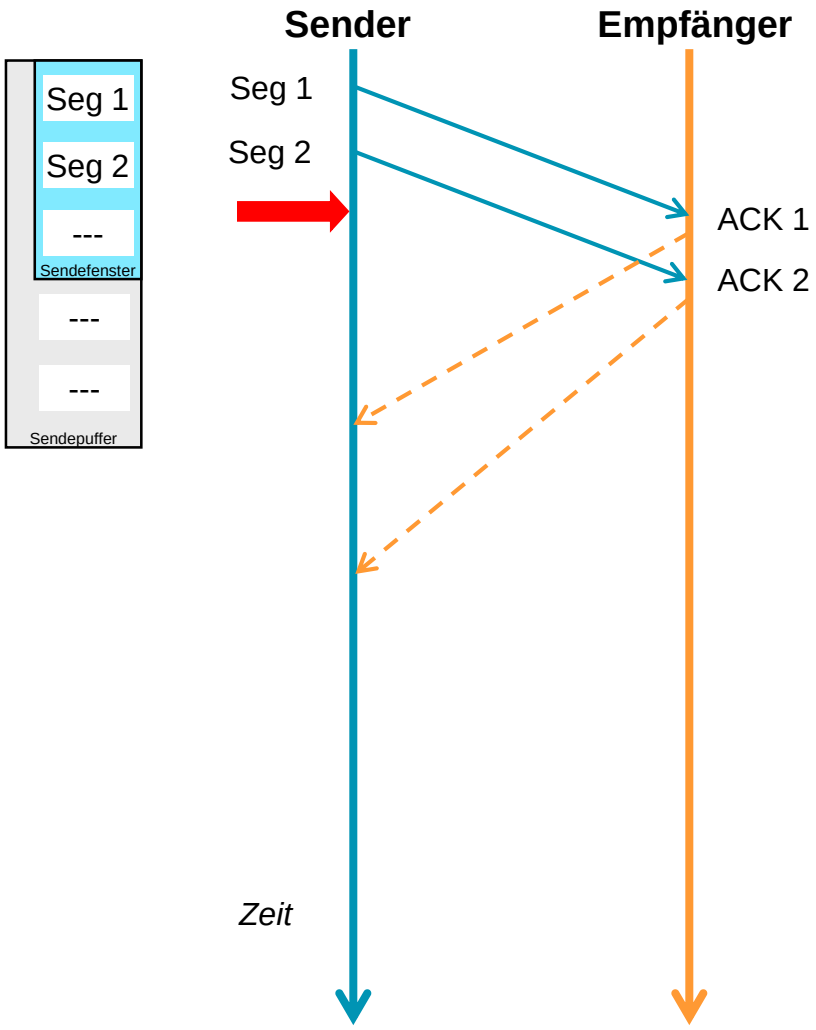


- **Anwendung produziert 2 Segmente**
- **Diese Segmente liegen im Sendepuffer und im Sendefenster**

Was ist der Unterschied zwischen Sendepuffer und Sendefenster?

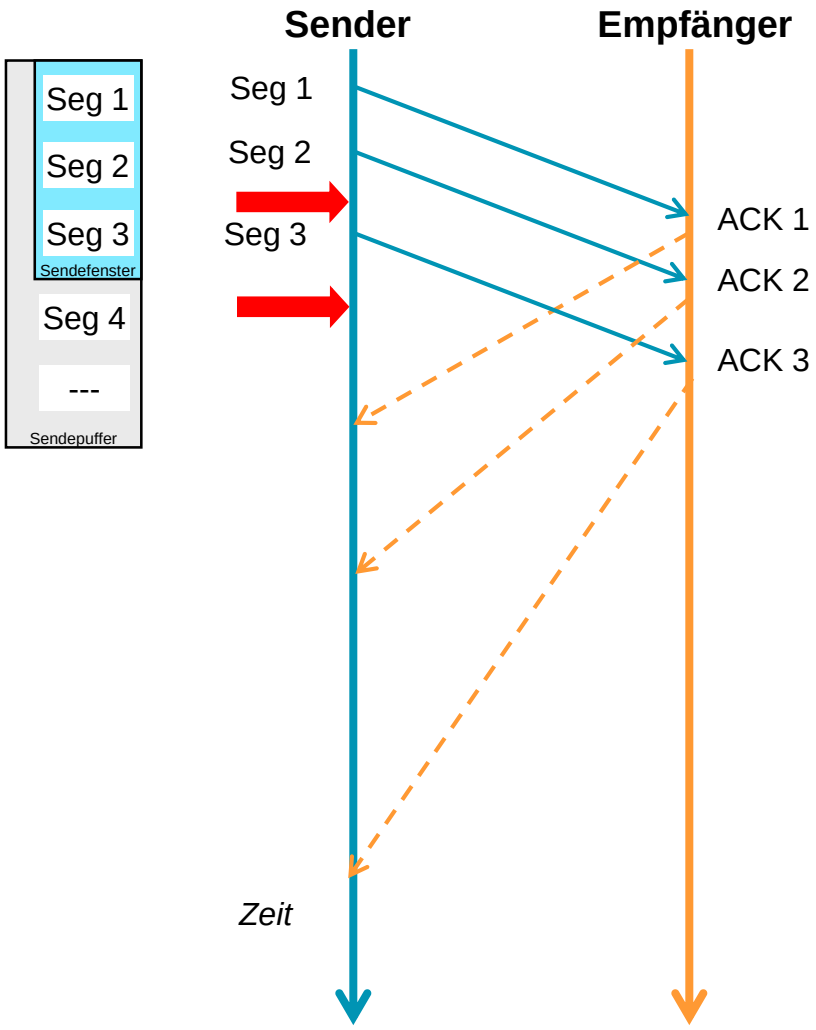
Was ist die Aufgabe von Sendepuffer und Sendefenster?

Sliding-Window - detaillierter



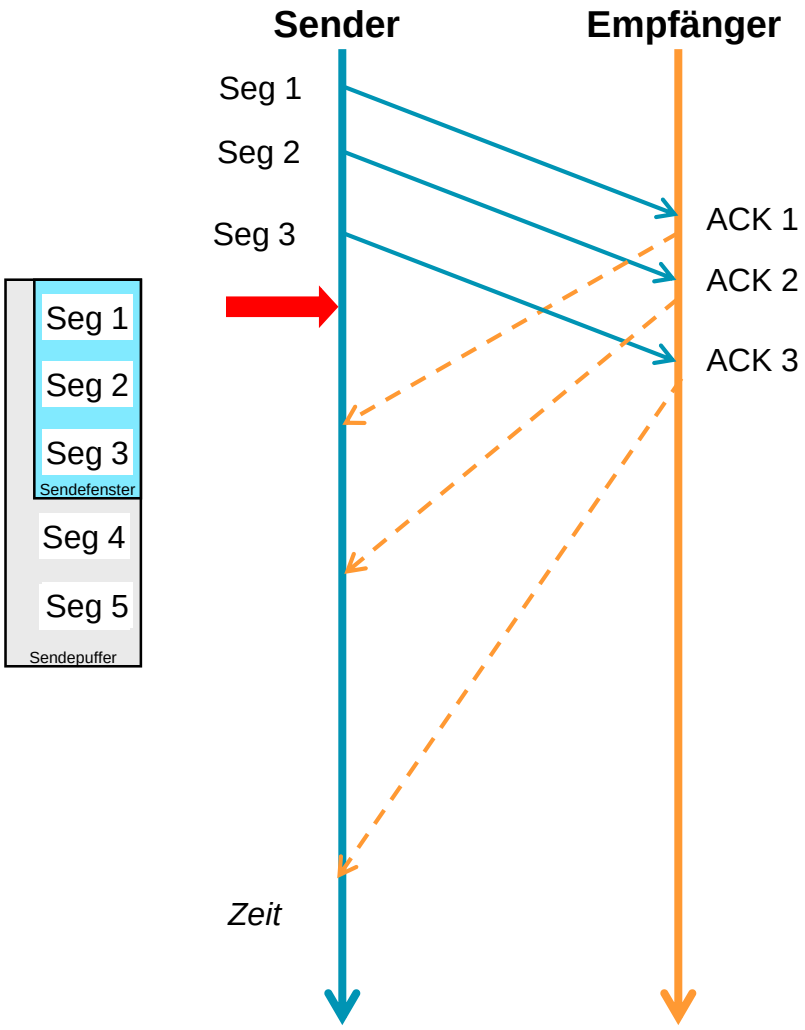
- **Was passiert jetzt??**
- **Beide Segmente können gesendet werden**
- **Auswirkungen auf den Sendepuffer und das Sendefenster?**

Sliding-Window - detaillierter



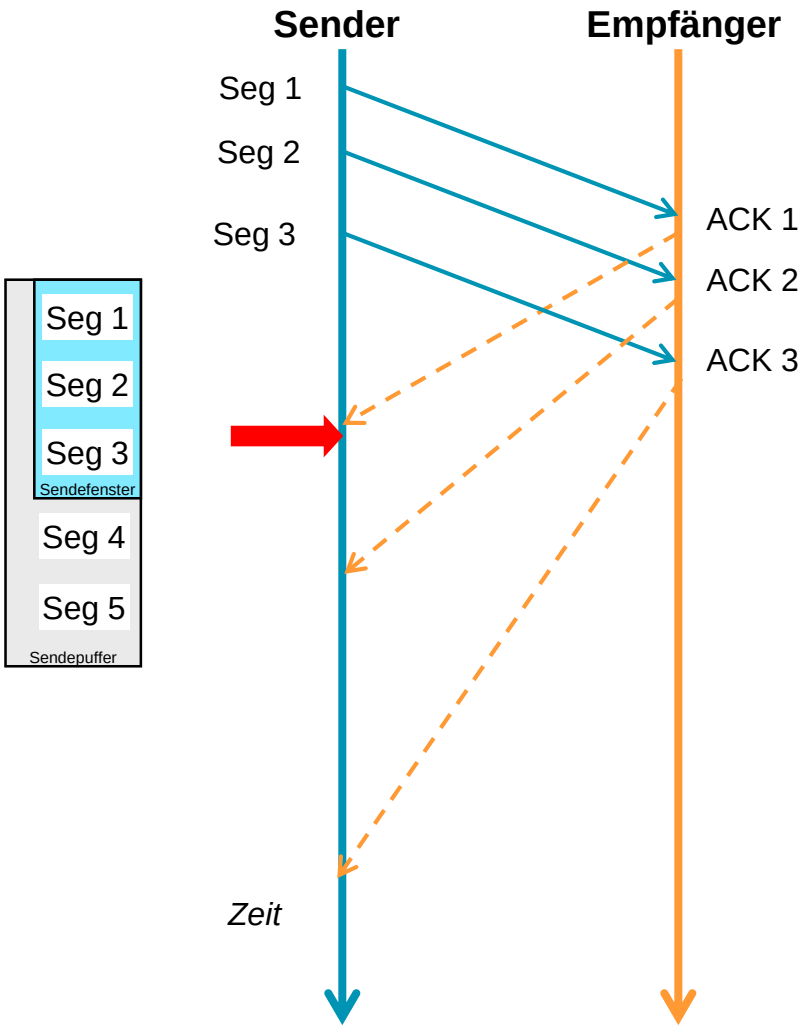
- **Anwendung produziert zwei weitere Segmente**
- **Was macht der Sender?**

Sliding-Window - detaillierter



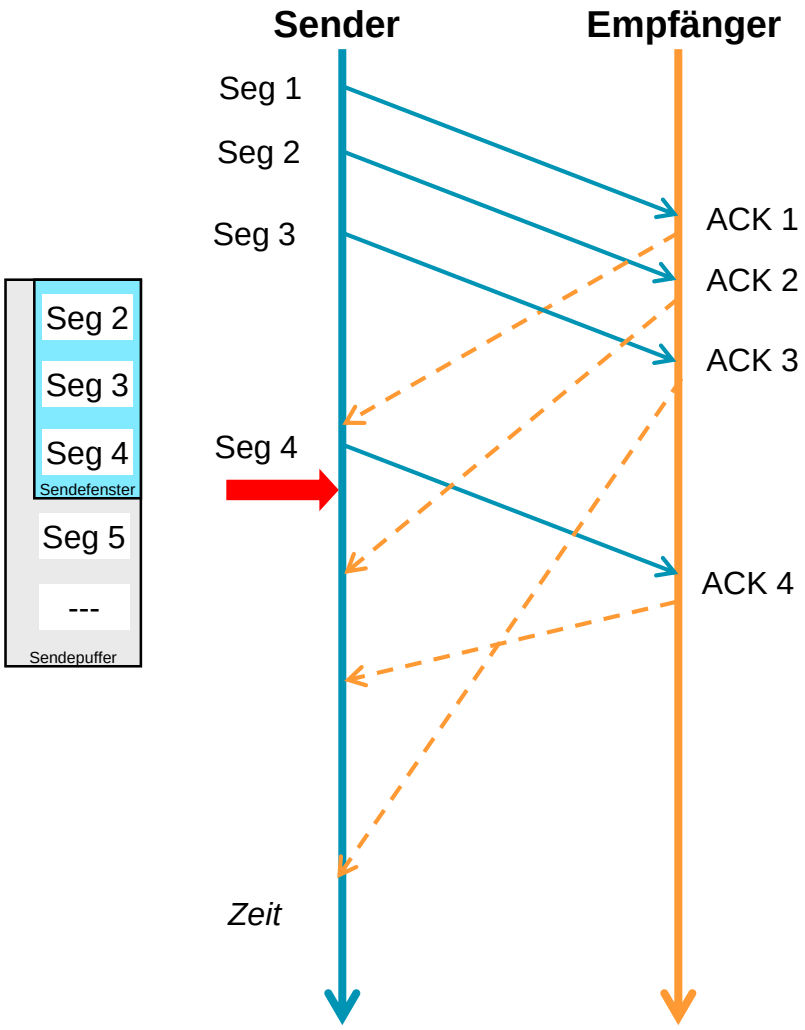
- **Anwendung produziert ein weiteres Segment**
- **Was passiert?**
- **Was wäre, wenn sie zwei weitere Segmente produziert?**

Sliding-Window - detaillierter



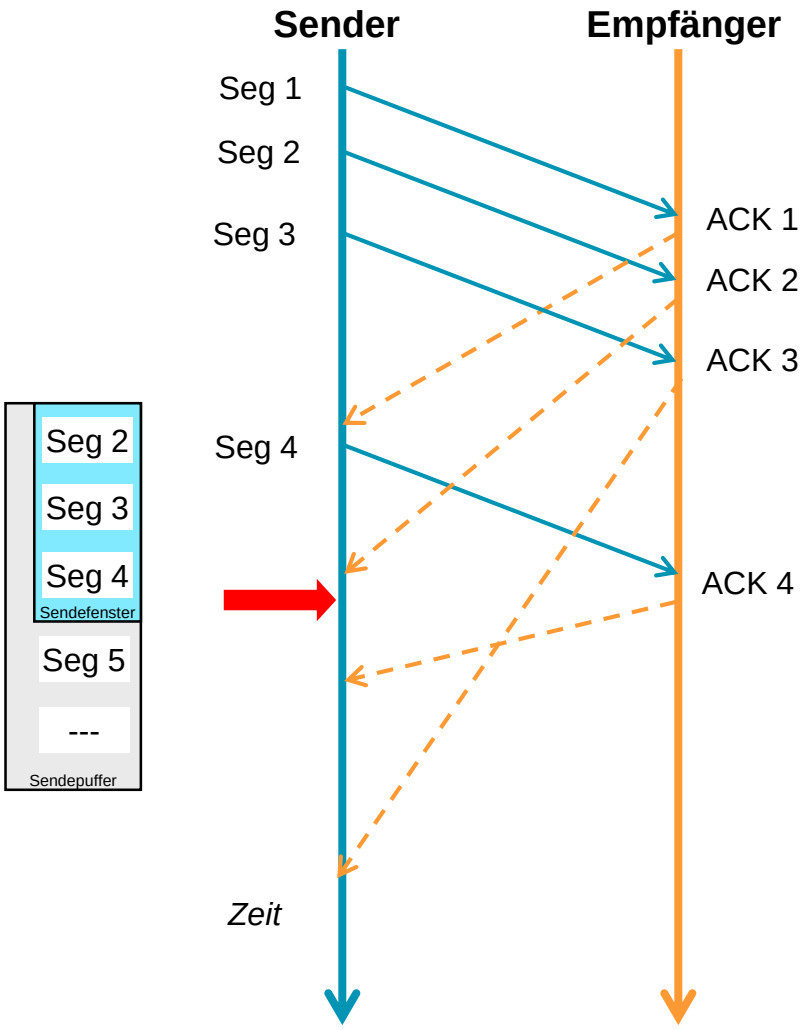
- **ACK 1 kommt nun beim Sender an**
- **Was passiert?**

Sliding-Window - detaillierter



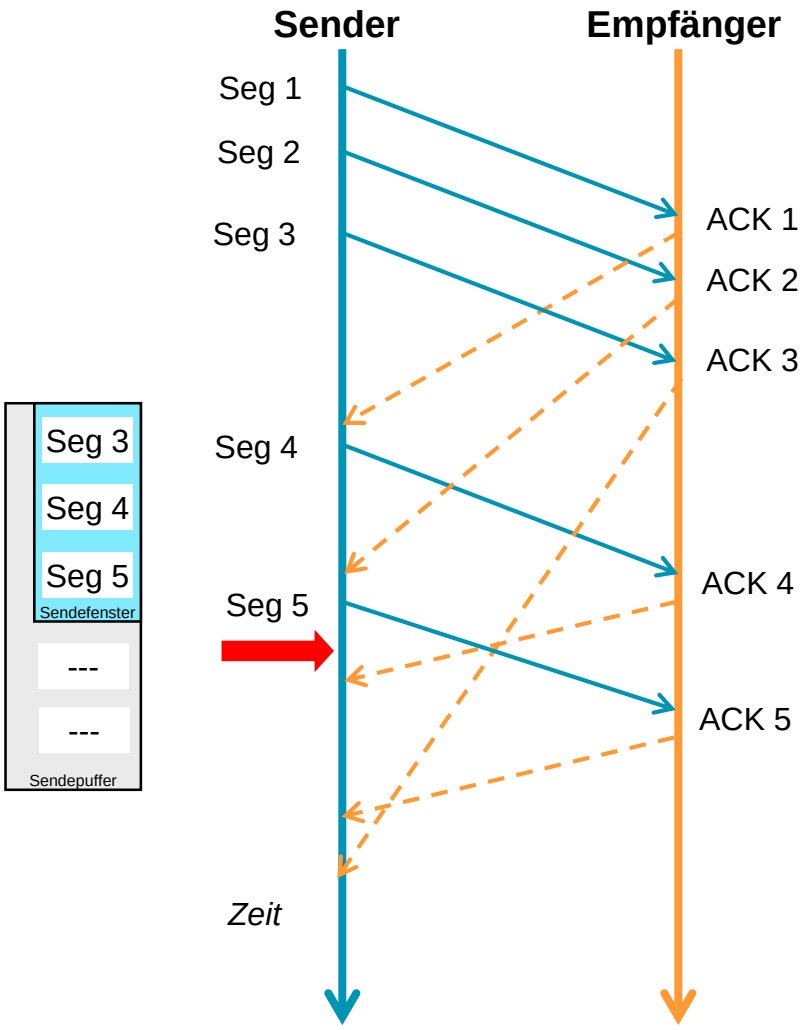
- **Schieben des Fensters**
- **Versenden von Segment 4**

Sliding-Window - detaillierter



- **Nun kommt ACK 2**
- **Was passiert?**

Sliding-Window - detaillierter



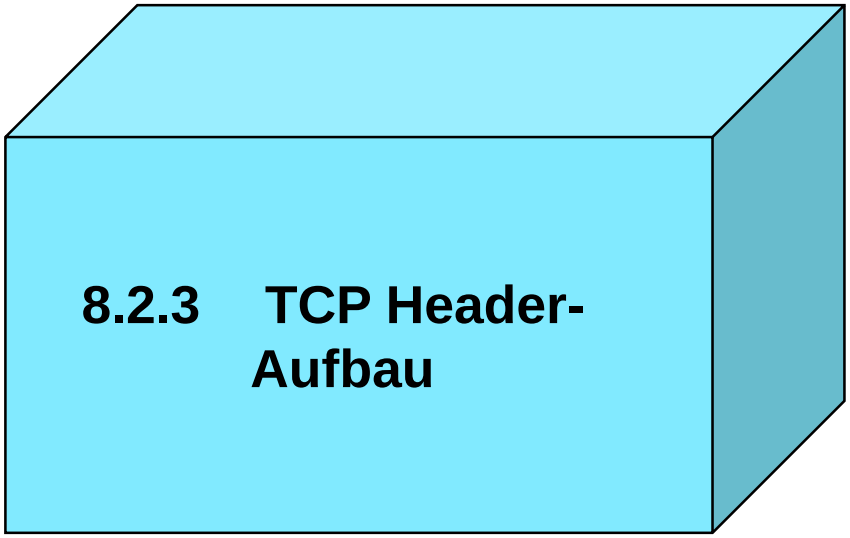
- **Schieben des Fensters**
- **Versenden von Seg 5**

Für Fortgeschrittene:

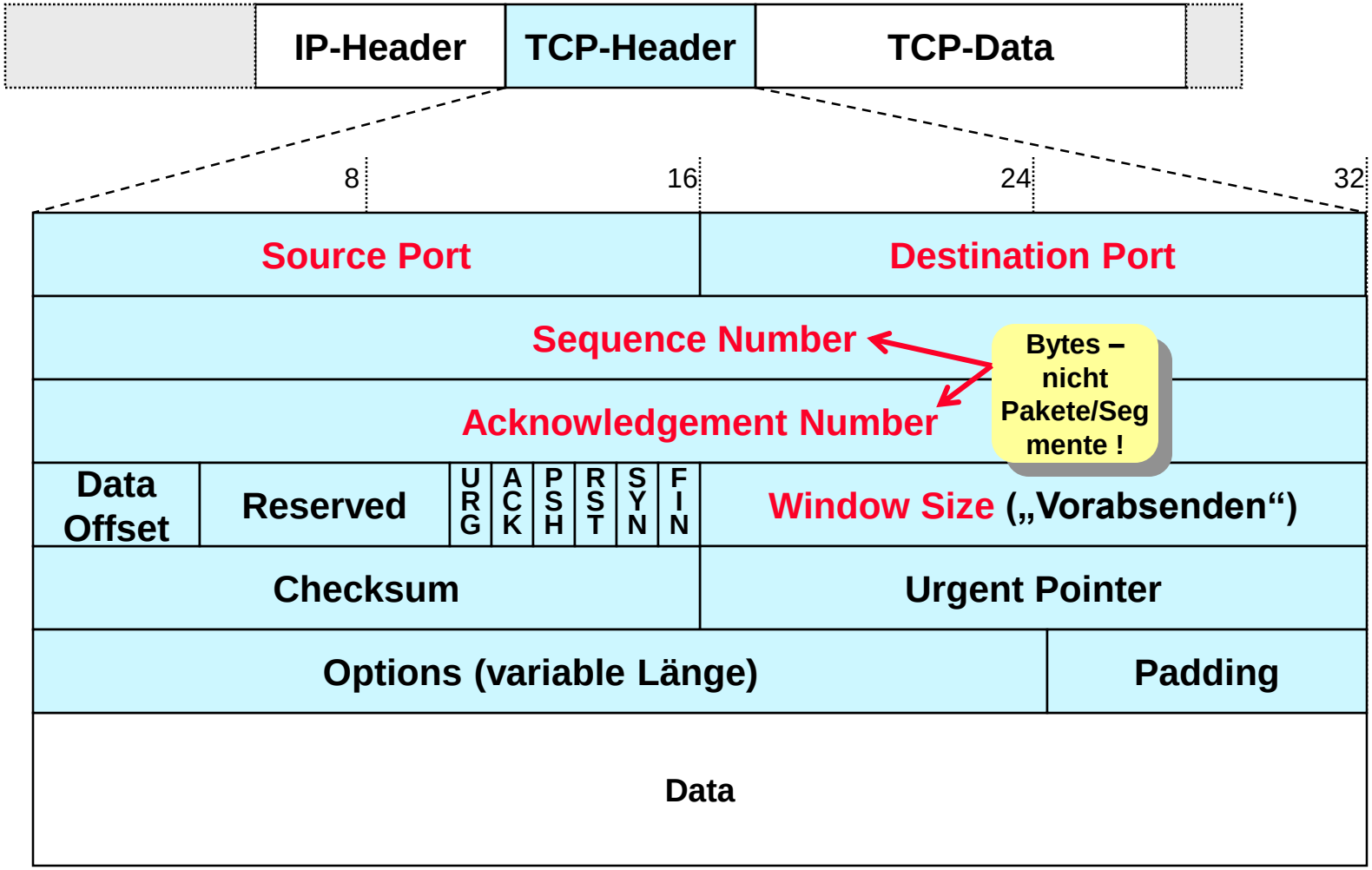
- **Was passiert, wenn ACK 4 und ACK 5 vor ACK 3 kommen?**

Frage an alle:

- **Wie groß soll das Sendefenster sein?**
- **Wer legt das Sendefenster fest?**



TCP-Header

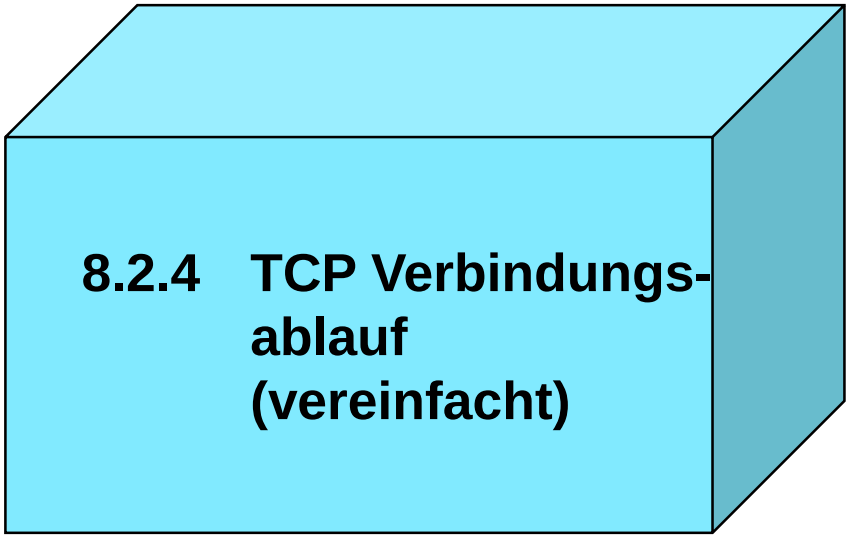

IP-Header
TCP-Header
TCP-Data
8
16
24
32
Source Port
Destination Port
Sequence Number
Acknowledgement Number
Bytes – nicht Pakete/Segmente !
Data Offset
Reserved
URG
ACK
PSH
RST
SYN
FIN
Window Size („Vorabsenden“)
Checksum
Urgent Pointer
Options (variable Länge)
Padding
Data

Aufbau des TCP-Headers

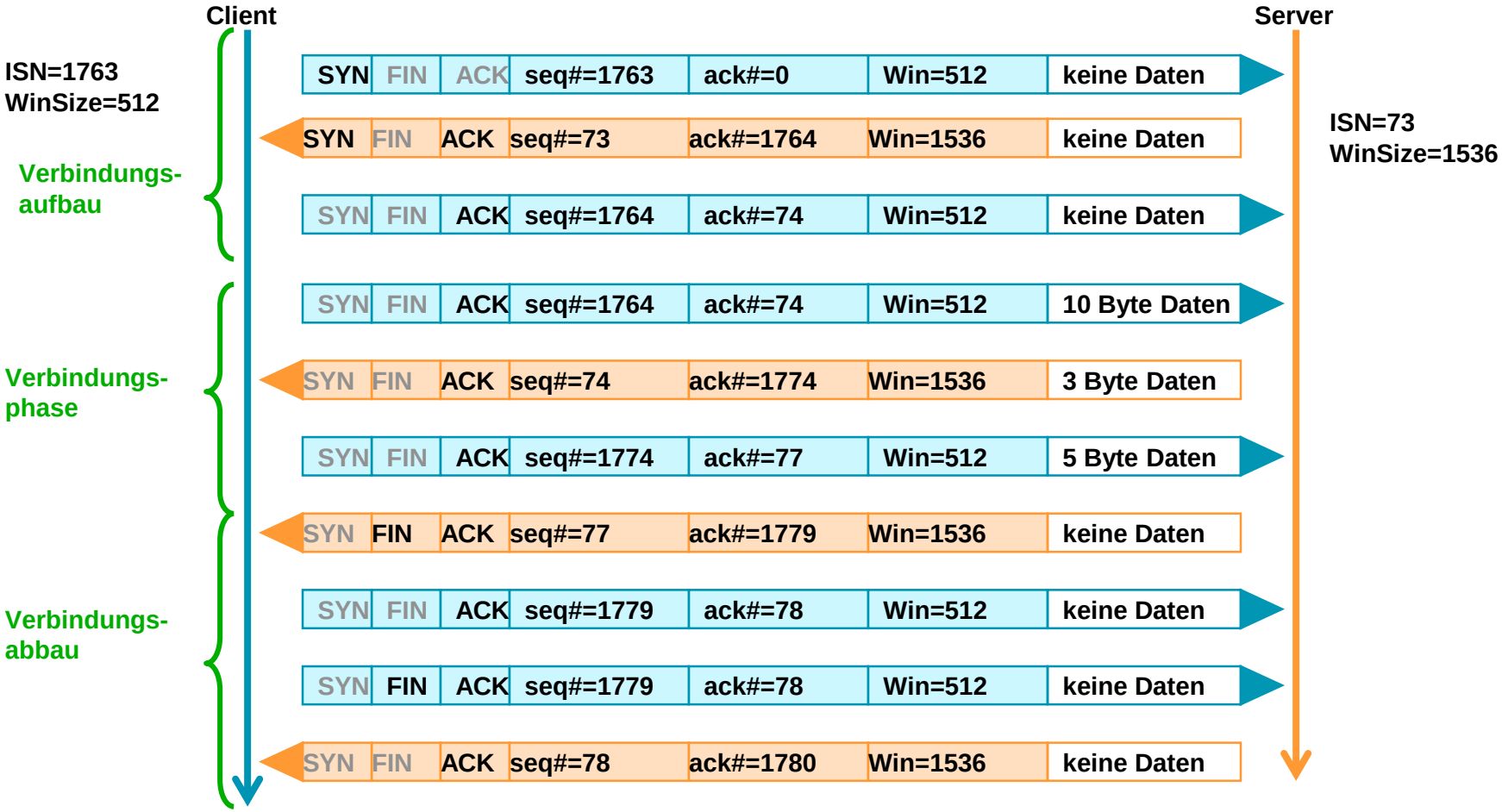
- **Source Port:** 16-Bit-Feld für lokalen TCP-User (Applikation)
- **Destination Port:** 16-Bit-Feld für remote TCP-User (Applikation)
- **Sequence number:** Position des aktuellen Blocks im laufenden Strom. Zeigt auf das erste neue Byte.
- **Acknowledgment Number:** Nächste Sequenznummer, die erwartet wird. Vorhergehende Nummern wurden korrekt empfangen.
- **Data Offset:** Länge des TCP-Headers in 32 Bit Blöcken
- **Flags:**
  - **URG** Urgent-Pointer-Flag (für Vorrang-Daten)
  - **ACK** Acknowledgement-Flag (Bestätigung)
  - **PSH** Push-Flag (direkte Weiterleitung an Anwendung)
  - **RST** Reset-Flag (Beenden der Verbindung aufgrund eines Fehlers)
  - **SYN** Synchronization-Flag (Synchronisationsvorgang)
  - **FIN** Final-Flag (ordentlicher Verbindungsabbau)

Aufbau des TCP-Haeders (Fortsetzung)

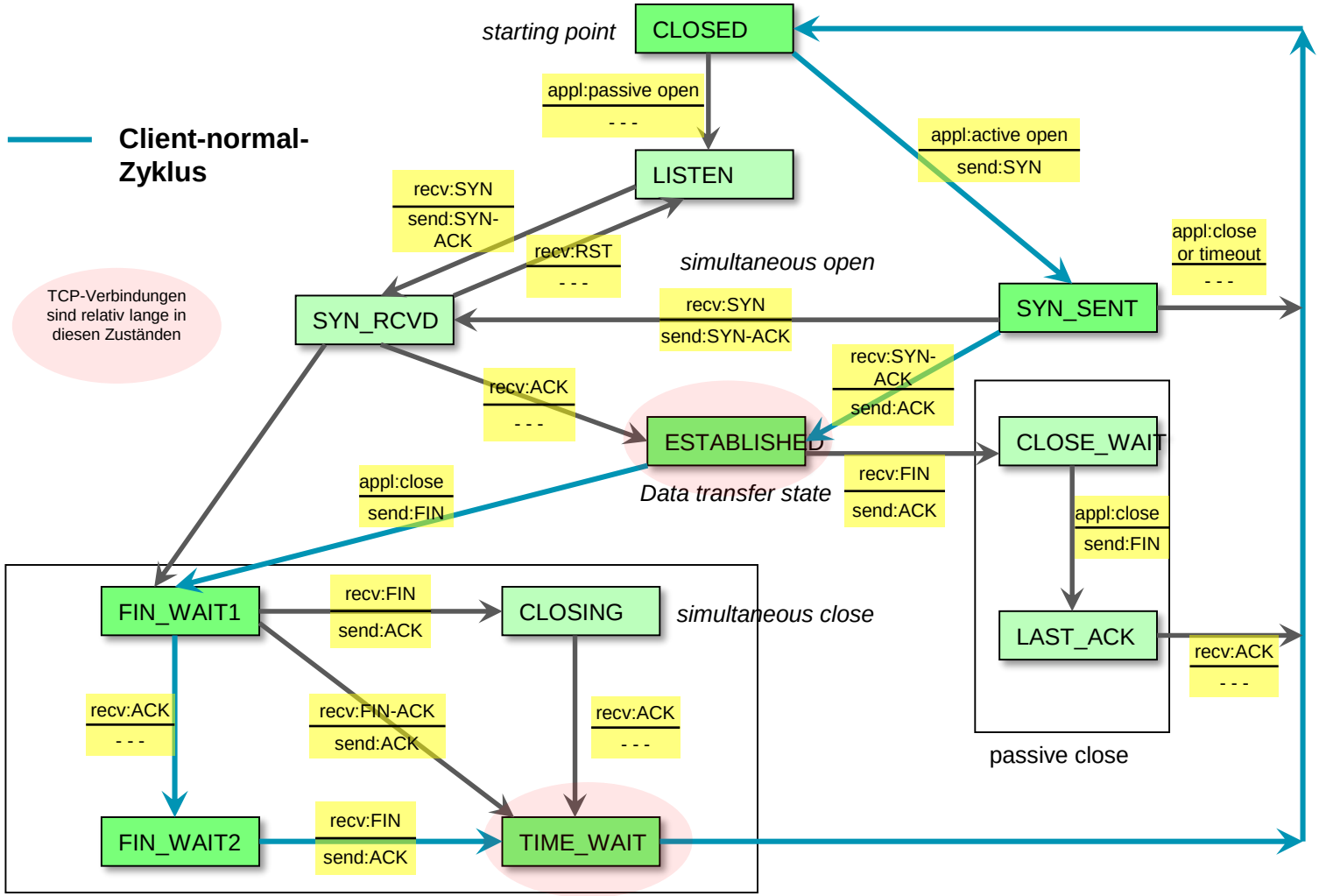
- **Window Size:** die noch verfügbare Puffergröße auf Empfangsseite, wichtige Info für den Sender, dient der Flusssteuerung
- **Urgent Pointer:** Zeiger auf Ende von dringlichen Daten
- **Ckecksum:** Prüfsumme über Header, Daten und einen Pseudoheader zum Schutz gegen fehlgeleitete Segmente
- **Options:** MSS (Maximum Segment Size) bei Verbindungsaufbau
- **Padding:** Fülldaten für 32 Bit-Grenze

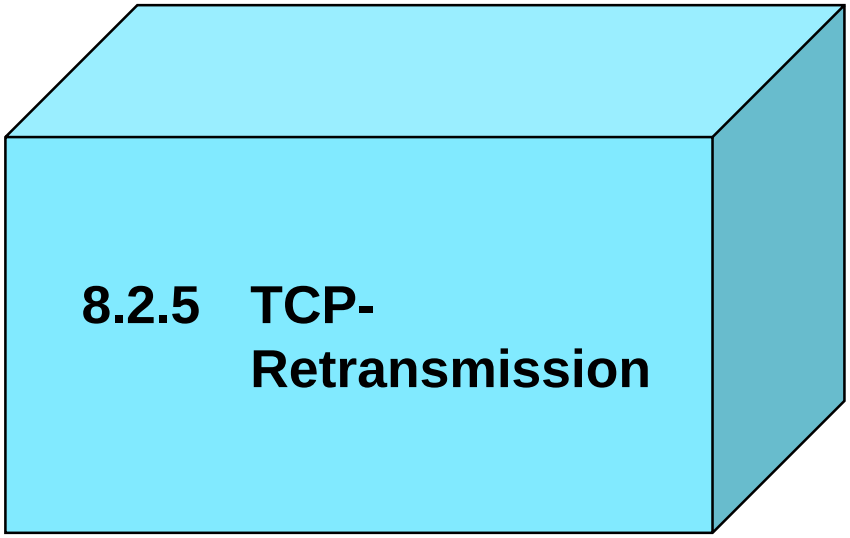


TCP-Verbindungsablauf (Normalfall, stark vereinfacht)

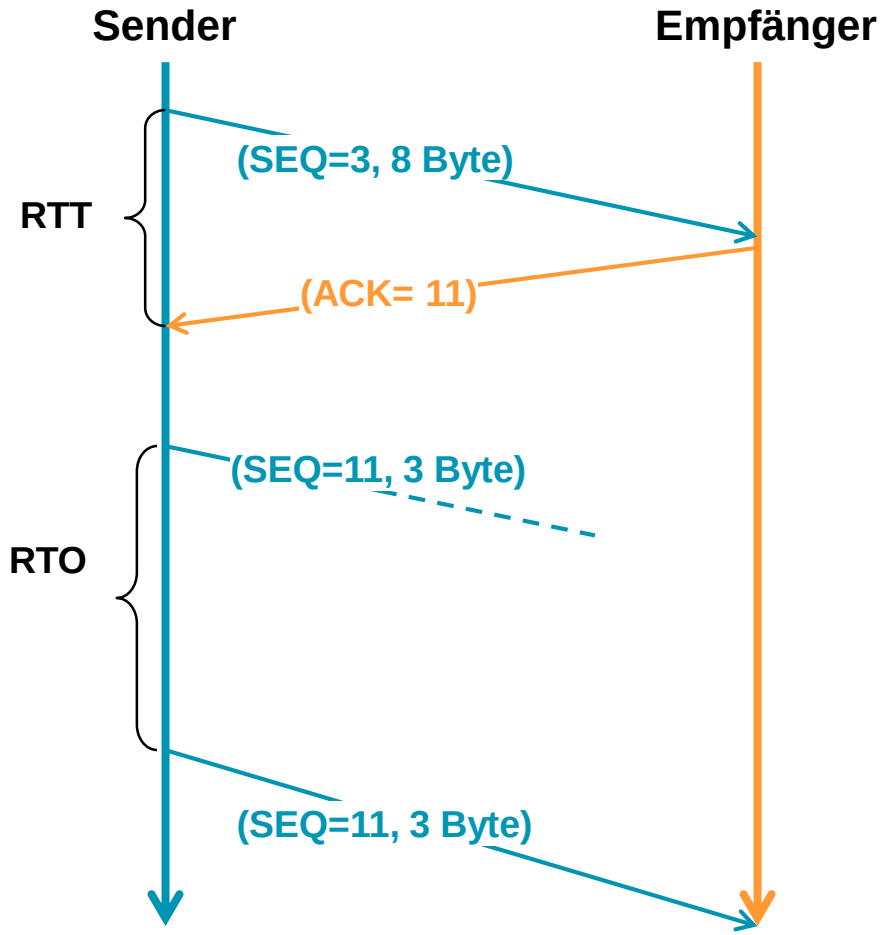


TCP Verbindungsmanagement (mit blauem Client-Life-Cycle)





TCP Retransmission - Aufgabenstellung



- **Faustregel:**
  RTO etwas größer als erwartete RTT
- **RTO zu groß:**
  schlechter Durchsatz
- **RTO zu klein:**
  unnötige Übertragungswiederholung

Also: Für effizientes TCP gute Schätzung der RTT notwendig

Fragen:

- **Anfangswert für RTO ?**
- **Wie soll zeitliche Änderung der RTT berücksichtigt werden ?**
- **Wie soll Timer nach Ablauf des RTOs gesetzt werden ?**

RTO-Timer nach RFC 793 (1981)

Schritt 1: **Kontinuierliche Mittelwertbildung der RTT (EWMA = Exponentially Weighted Moving Average):**

$$SRTT_{new} = \alpha * SRTT_{old} + (1-\alpha) * RTT_{akt}$$

SRTT	Smoothed Round Trip Time
α	smoothing Faktor (typisch 0,8 – 0,9 z.B. 7/8)

Schritt 2: **Varianzfaktor (+ Limits)**

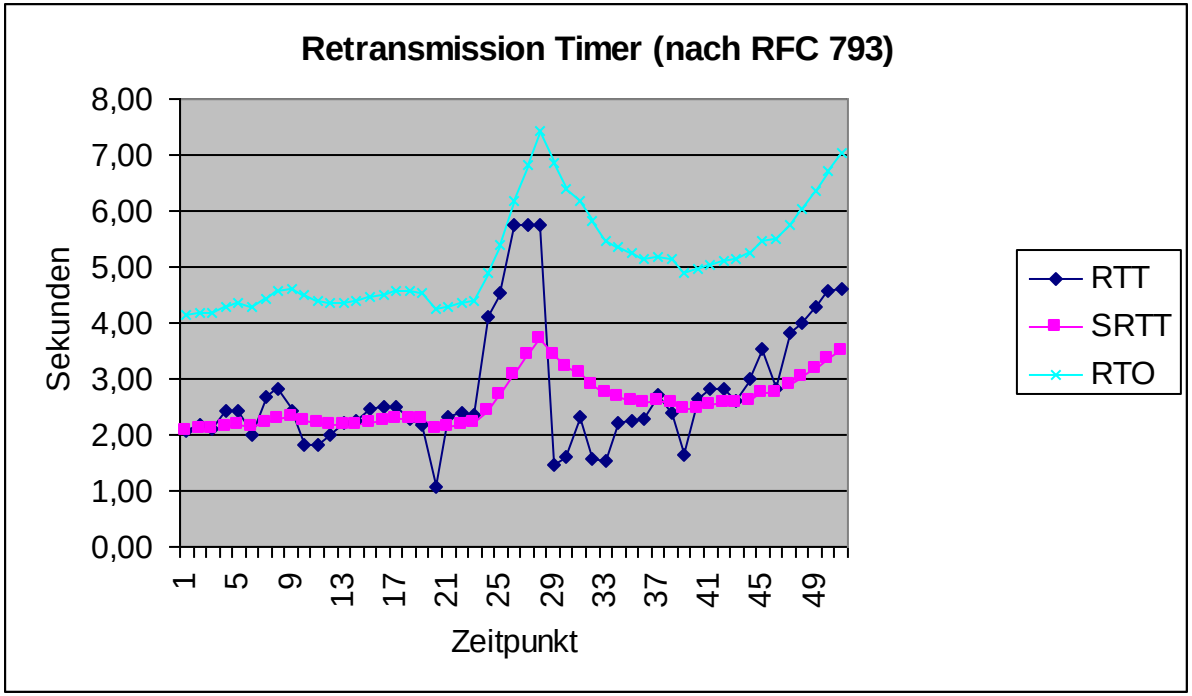
$$RTO = \min(\text{o-grenz}, \max(\text{u-grenz}, \beta * SRTT))$$

β	Varianzfaktor (typisch 2)
u-grenz	minimales RTO (z.B. 1 msec)
o-grenz	maximales RTO (z.B. 10 sec)

Bemerkungen:

- mathematisch einfache, fortlaufende Berechnung (Integer mögl.),
- bei geringen RTT-Schwankungen liegt RTO stets zu hoch,
- Problem der Initialisierung/Reinitialisieren des RTO-Timers

Beispielberechnung

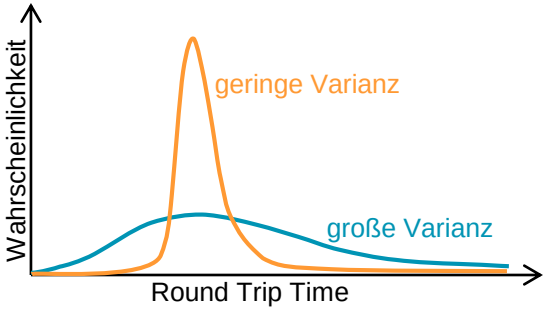


Interpretation und kritische Diskussion der Kurve !!

RTO-Timer nach Jacobson (1988)

Schlüsselbeobachtung:

- **Bei großer Last (starke Schwankung der RTT) ist größere Sicherheitszugabe für RTO notwendig**
- **Bei hoher RTT und geringer Varianz der RTT ist eine geringe Sicherheitszugabe notwendig**



Lösungsidee:

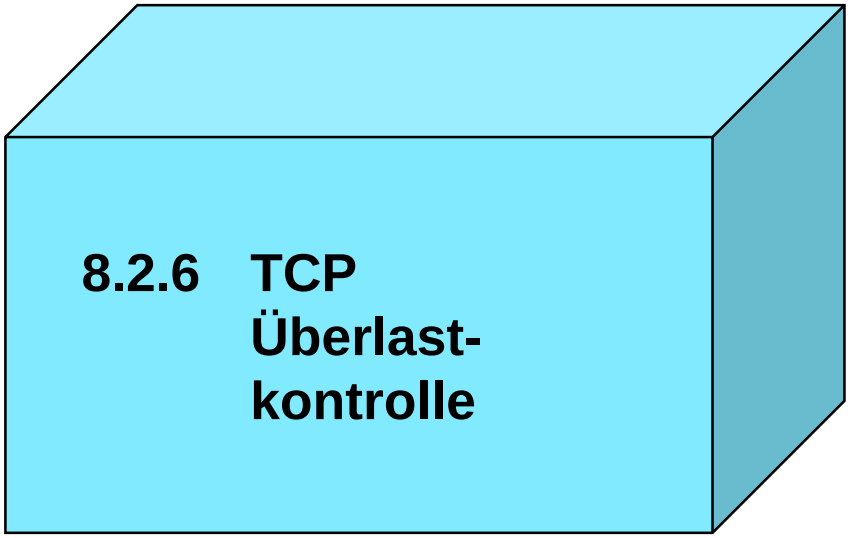
- **Mache den Wert des RTOs abhängig von der Standard-Abweichung**

$$\sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i-\mu)^2}$$

zu kompliziert !

Problem:

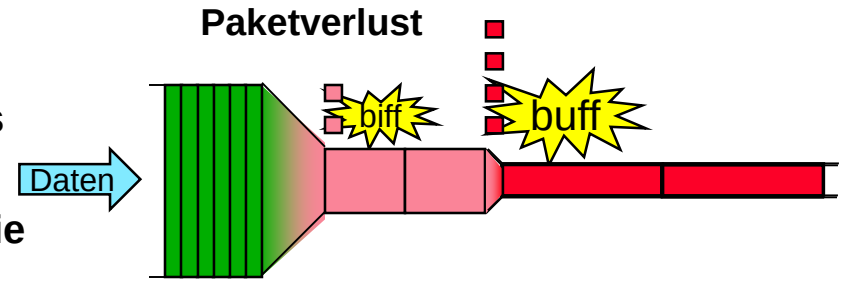
- **Berechnung der Standardabweichung ist recht komplex**
  –> nehme „mittlere“ Abweichung



Phasen der Überlastkontrolle

Start der TCP-Übertragung

- **Slow Start (Aufbau eines Gleichgewichts (= Equilibrium))**
- **→ genauer Erklärung: siehe nächste Folie**



Erhalt des Gleichgewichts

- **Optimale Einstellung des RTOs (Jacobson/Karel Algorithmus)**
- **Maximierung der Auslastung (–> Additive Increase, Überlastfenster (= spezielle Fenstergröße für Überlastkontrolle) wächst linear)**

Reaktion auf Überlastungsanzeichen

- **Stabilisierung der Situation (–> Multiplicative Decrease, Überlastfenster wird halbiert)**
- **Sicherstellung einer kontinuierlichen, unterbrechungsfreien Datenflusses**

Fairness der Resourcenverteilung

- **automatisch durch AIMD (= Additive Increase / Multiplicative Decrease)**

Slow Start

Ziel von Slow Start:

- **Rasches Heranführen der TCP-Verbindung an das Gleichgewicht**

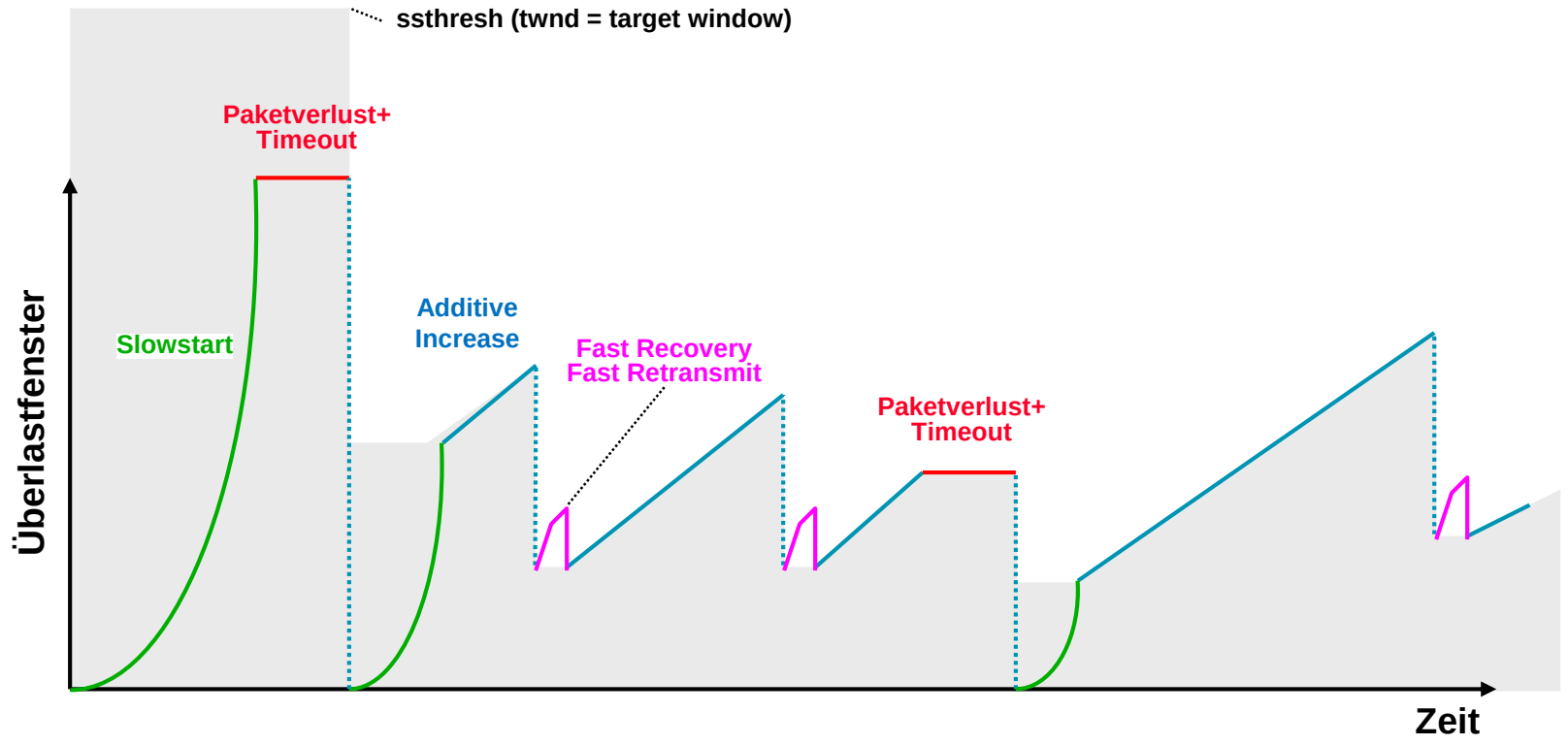
Strategie:

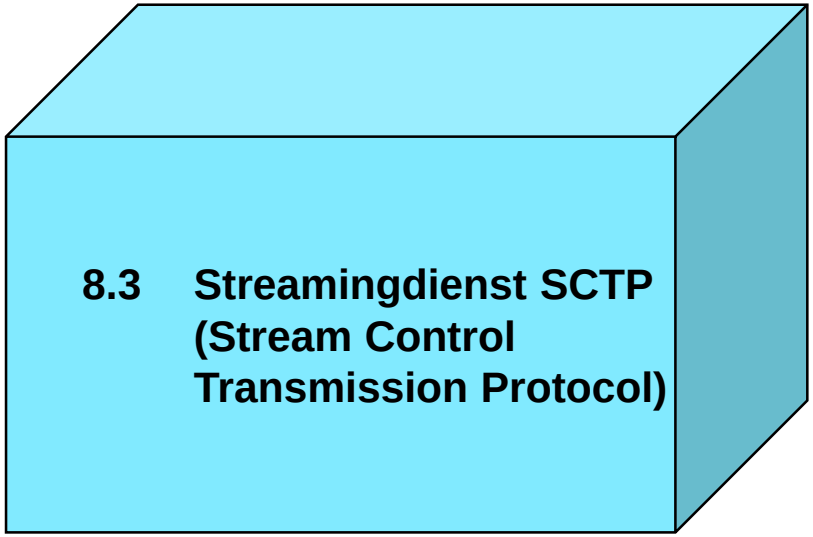
- **Progressives Testen der Belastbarkeit der Strecke Sender-Empfänger (bis in die Überlast hinein)**
- **Paketverluste (–> Retransmission Time Out) werden als Kennzeichen für eine Überlastsituation gewertet.**
  (Dies ist kritisch für mobile TCP –> niedrige Übertragungsrate)
- **Beruhigung des Netzes nach dem Eintreten einer Überlastsituation (Leeren der Router-Queues am Engpass)**

Einsatz:

- **Beim Start einer Verbindung**
- **nach Ablauf des Retransmission-Timers**
- **nach langer passiver TCP-Phase**

TCP Sägezahnverhalten





Lernziele:

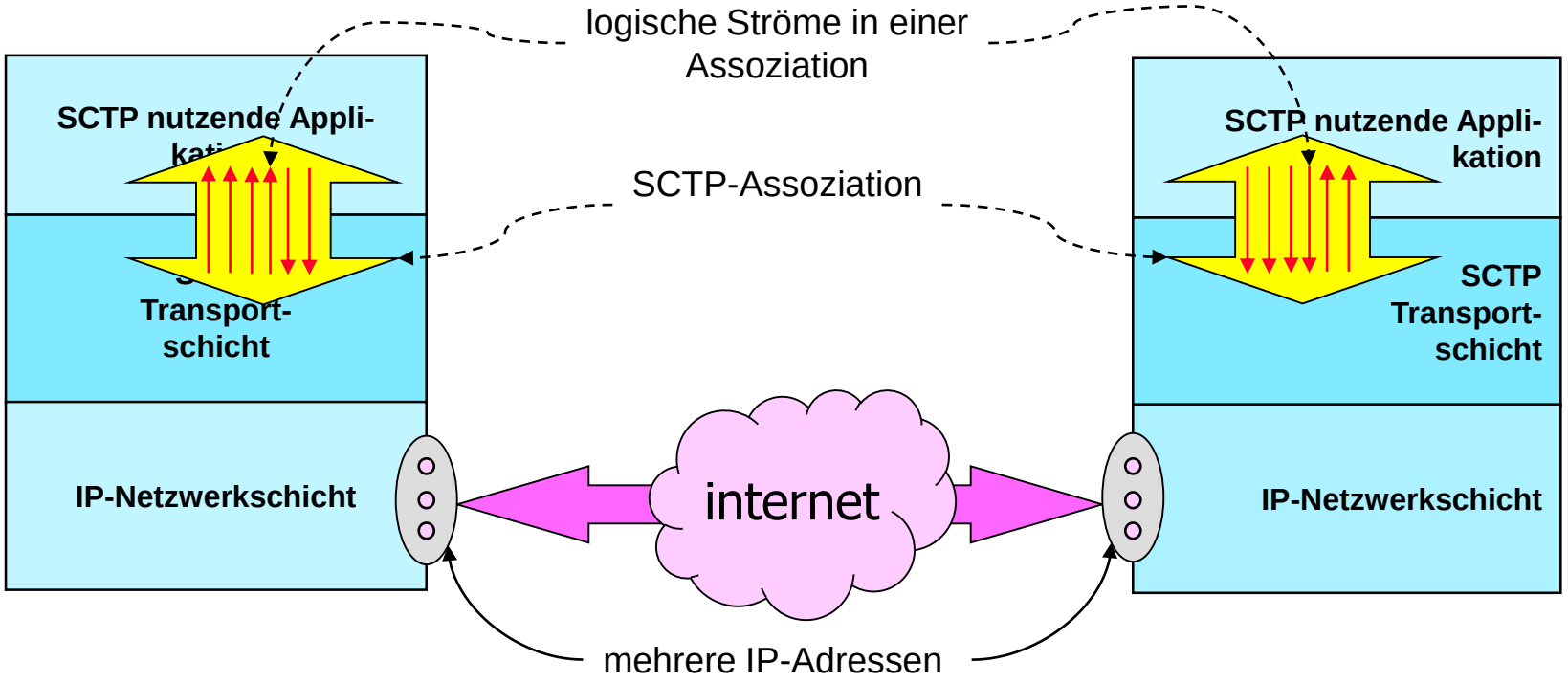
Nach Durcharbeiten dieses Teilkapitels sollen Sie Motivation und ausgewählte Grundkonzepte von STCP darstellen können.

Geschichte, Motivation und Überblick

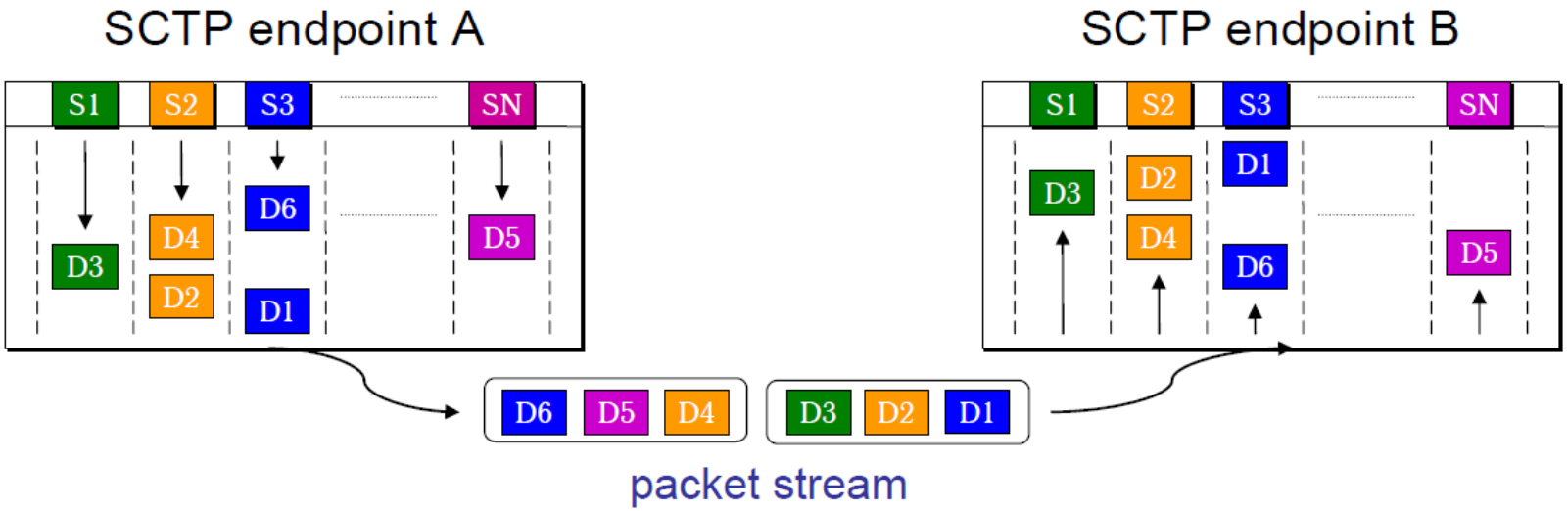
- TCP und UDP klassische Transportprotokolle (späte 60er Jahre)
  - UDP: unzuverlässiger Paketdienst; geeignet für Multicast und Broadcast
  - TCP: (absolut) zuverlässiger, byteorientierter bidirektionaler Streamingdienst
- Spezielle Anforderungen beim Transport von Signalisierungsdaten im Sprachverkehr führte zur Entwicklung des SCTP-Protokolls
  - Unterstützung eines Nachrichtenkonzepts
  - Unterstützung von mehreren unabhängigen Nachrichtenströmen (TCP-Problem „Head of Line Blocking“) → Multi-Streaming
  - Unterstützung alternativer Netzwerkpfade → Multi-Homing
  - Überwachung der Netzwerkpfade und Endpunkte
  - Selektive Bestätigung von übertragenen Daten
  - Optionale Reihenfolgesicherung
  - Sicherer Verbindungsaufbau (DoS-Angriffe)
- Oktober 2000: Die Signaling Transport (SIGTRAN) Gruppe der Internet Engineering Task Force (IETF) definiert im RFC 2960 das Stream Control Transmission Protocol

SCTP-Transportarchitektur

- **Konzept der Assoziation, die mehrere logische Ströme umfasst**
- **Unterstützung von Multihoming und Multistreaming.**



Multi-Streaming

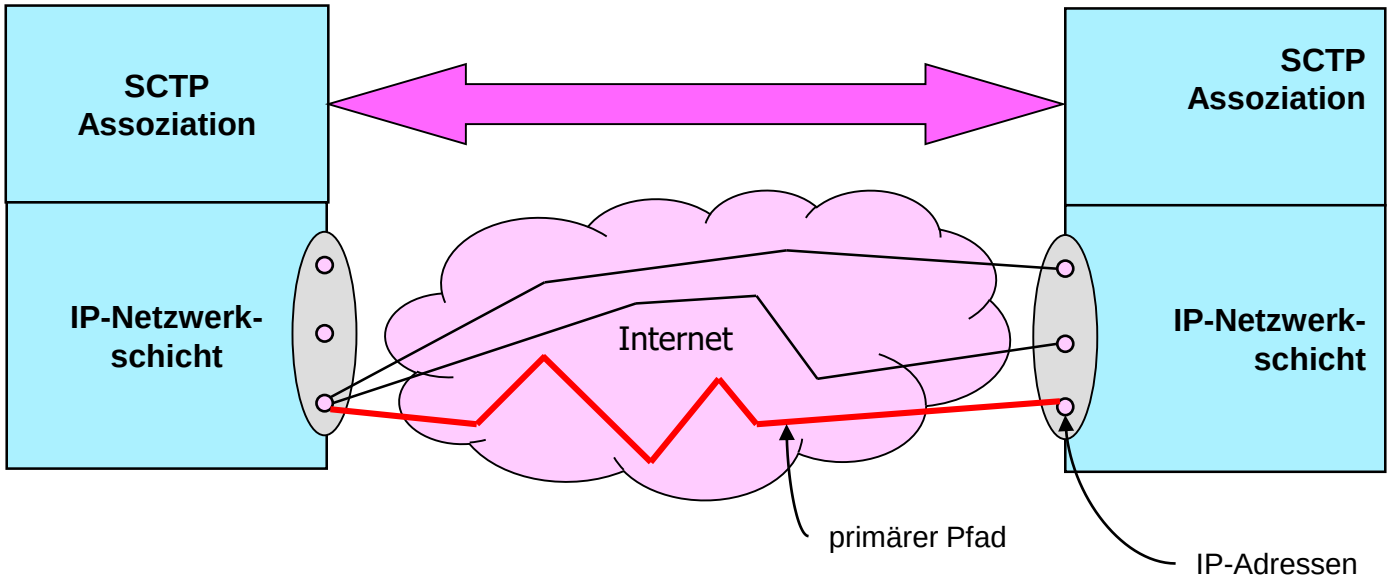


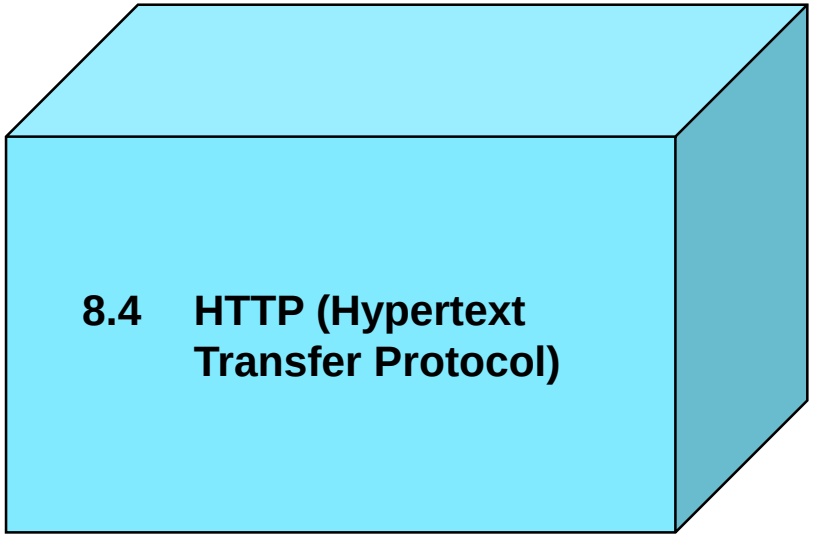
aus: Unurkhaan, Esbold: Secure End-to-End Transport over SCTP - A new security extension for SCTP, Dissertation; 2005

- **Begrenzung von „Head of Line Blocking“ (HoLB):**
  Übertragung der Nutzdaten erfolgt über unabhängige Kanäle bzw. Streams.
  Der Verlust eines Chunks (Pakets) beeinflusst nicht alle anderen Streams.

Multi-Homing

- **Zuordnung von mehrere IP-Adressen zum Endpunkt einer Assoziation (Port bleibt gleich).**
- **Dadurch sind mehrere Pfade möglich; für jeden Pfad wird ein kompletter Parametersatz für Fluss- und Überlastkontrolle gehalten.**
- **Ein ausgesuchter Pfad (primärer Pfad) überträgt Daten, die restlichen Pfade sind redundant.** (Zukünftiges Konzept: Concurrent Multipath Transfer, CMT)





8.4 HTTP (Hypertext Transfer Protocol)

Lernziele:

Nach Durcharbeiten dieses Unterkapitels sollen Sie das Anwendungsprotokoll HTTP einordnen und dessen Grundkonzepte darstellen können.

.

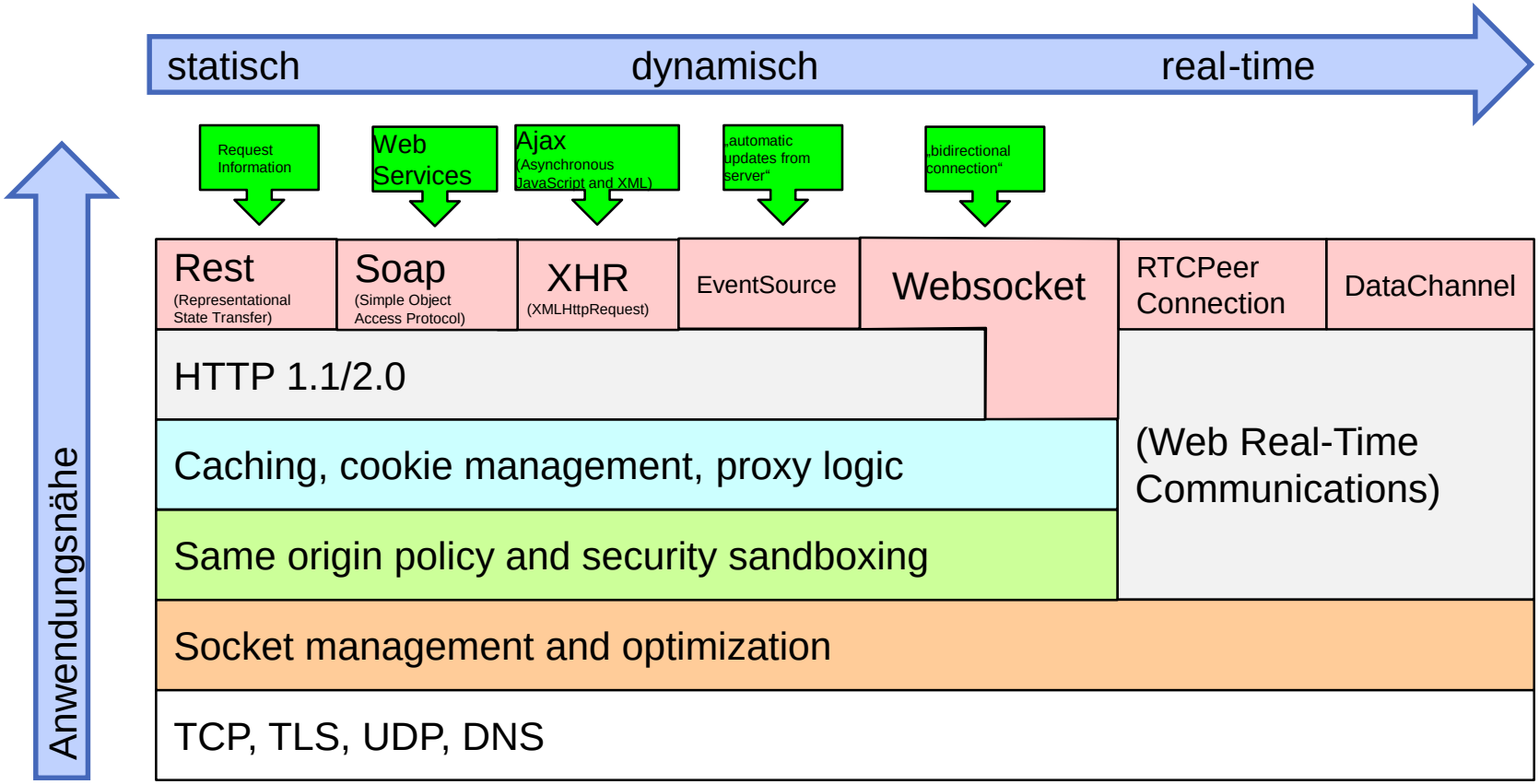
HTTP-Grundkonzept

- HTTP setzt auf das Protokoll TCP auf.
- Der Web-Server benutzt standardmäßig den TCP-Port 80.
- HTTP ist ein zustandsloses Request/Response-Protokoll.
- HTTP ist textorientiert.
- Typische Operationen sind
  - Head: Abruf von Metainformationen über das in der URL bezeichnete Dokument
  - Get: Abruf des in der URL bezeichneten Dokuments
  - Post: Ausgabe von Information an den Server
  - …
- Designziele von HTTP sind:
  - Einfachheit: Das Protokoll soll nur wenige Ressourcen beanspruchen.
    (Davon entfernen wir uns gegenwärtig wieder.)
  - Geschwindigkeit: Das Protokoll soll so schnell wie möglich sein.

Beispiel: Get auf die Seite : http://www.msv-turnen.de/preise.htm

```
Hypertext Transfer Protocol
  GET /preise.htm HTTP/1.1\r\n
    [Expert Info (Chat/Sequence): GET /preise.htm HTTP/1.1\r\n]
    Request Method: GET
    Request URI: /preise.htm
    Request Version: HTTP/1.1
  Host: www.msv-turnen.de\r\n
  Connection: keep-alive\r\n
  Upgrade-Insecure-Requests: 1\r\n
  User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/5
  Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image
  Accept-Encoding: gzip, deflate, sdch\r\n
  Accept-Language: de-DE,de;q=0.8,en-US;q=0.6,en;q=0.4\r\n
  \r\n
```

Web-APIs - Einordnung



compare https://hpbn.co/primer-on-browser-networking/ (Primer on Browser Networking)

Beispiel einer REST-API: Karnevalservice

Der Karneval Service liefert für ein gegebenes Jahr das Datum der Karnevalsfesttage:

- **Weiberfastnacht, Rosenmontag, Aschermittwoch sowie das Datum von Ostersonntag.**

Die Aufrufparameter sind `jahr` und `tag`. Der Parameter `jahr` ist eine natürliche Zahl zwischen 1900 und 2100. Mit dem Parameter `tag` wird der gewünschte Karnevalsfesttag gesteuert. Es gilt:

- **`W` = Weiberfastnacht,**
- **`R` = Rosenmontag,**
- **`A` = Aschermittwoch sowie**
- **`O` = Ostern.**

Der Rückgabewert ist ein JSON String, der die Bezeichnung des Karnevalsfesttags sowie das zugehörige Datum enthält.

REST-Prinzip: Funktionsaufruf und alle Parameter werden in die URL kodiert. Ergebnis wird im JSON-Format geliefert

Beispiel einer REST-API: Karnevalservice

Das Eingabeformat wird sofort aus nachfolgendem Beispiel klar.

Gesucht ist Weiberfastnacht im Jahr 2020. Eingabe-URL:

`https://rest.docklab.de/api/karneval?jahr=2020&tag=W`

Ausgabe:

- **Chrome:** `["Weiberfastnacht",[20,2,2020]]`
- **Firefox:**

```
0:    "Weiberfastnacht"
1:
    0:    20
    1:    2
    2:    2020
```

Geschichte von HTTP

1991: HTTP 0.9, Tim Berners-Lee (CERN) entwickelte HTTP.
Ziel: Einfachheit, einfach eine ASCII-Seite vom Server abrufen.

1992: HTTP 1.0, W3C.
Ziel: Übertragung beliebiger Dateiformate zwischen Client und Server.

1997: HTTP 1.1, RFC 2068.
Ziel: Performance-Optimierung, Caching, Pipelining.

2000: Roy Fielding: Doktorarbeit „Geburtsstunde der REST-API
(REST = REpresentational State Transfer)

2012: Erster HTTP 2.0 basierend auf SPDY (Google)

2015: HTTP 2.0. RFC 7540.
Konzept: Performance-Optimierung durch Multiplexing (parallele Anfragen), Stream-Priorisierung, Server-Push, „must“-TLS.

seit 2016: QUIC (Quick UDP Internet Connection)
Konzept : HTTP über TLS 1.3 und UDP.
Wird unterstützt von Google Chrome

2018: ca.50% der B2B-Kommunikation und –Collaboration laufen über REST.