

# Transportschicht

## ■ Einführung

## ■ UDP

## ■ Fehlerkontrolle

## ■ TCP

- Segmentformat
- Fehlerkontrolle
- Verbindungsauf- und -abbau
- Schätzung der RTT
- Fluss- und Überlastkontrolle schätzung von Round-trip
- Leistungsanalyse
- Multipath TCP über handy und ethernet z.b.

## ■ TLS

## ■ QUIC

# TCP

## ■ Transmission Control Protocol

- das verbreitete zuverlässige Transportprotokoll im Internet
- u.a. RFCs 793, 1323, 2018, 5681
- Punkt-zu-Punkt: ein Sender, ein Empfänger
- reihenfolgebewahrender Bytestrom
- fensterbasierte Fehlerkontrolle
- vollduplex: 2 entgegengesetzte Datenströme
- verbindungsorientiert: Auf- und Abbau einer Verbindung
- Flusskontrolle: Mechanismus, um Überschreitung der Kapazität des Empfängers zu verhindern
- Überlastkontrolle: Mechanismus, um Überlastung des Netzes zu verhindern

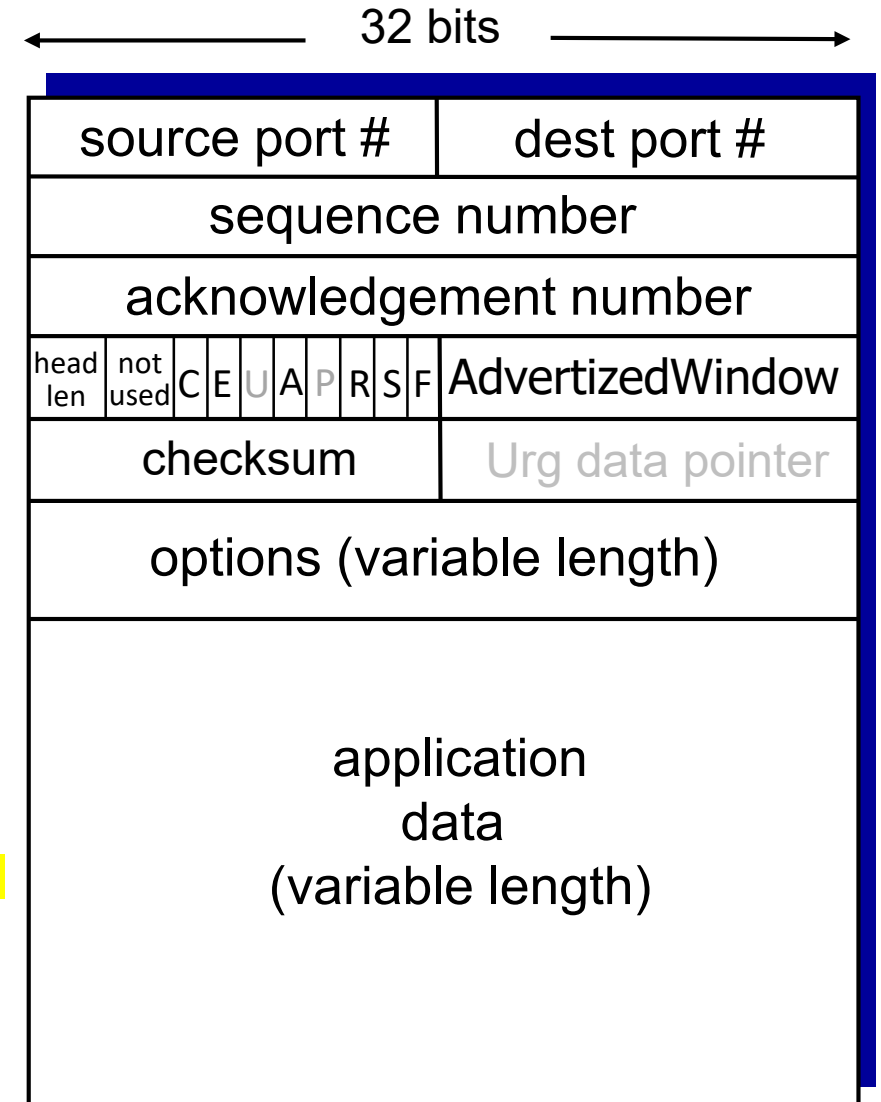
# Transportschicht

- Einführung
- UDP
- Fehlerkontrolle
- TCP
  - Segmentformat
  - Fehlerkontrolle
  - Verbindungsauf- und -abbau
  - Schätzung der RTT
  - Fluss- und Überlastkontrolle
  - Leistungsanalyse
  - Multipath TCP
- TLS
- QUIC

# TCP: Segmentformat

- **sequence number**: Nummer des ersten Bytes des Segments im Bytestrom
- **ack. number**: Nummer des nächsten erwarteten Bytes im Bytestrom  
ACK 42 heißt alles bis einschließlich 41 angekommen
- **Flags mit Steuerinformation**:
  - CWR (Congestion Window Reduced) überlastkontrolle
  - ECE (ECN-Echo)
  - URG (urgent pointer gültig)
  - ACK (ACK gültig) bestätigt ein segment
  - PSH (Push Segment) sofort an anwendung ausliefern
  - RST (Verbindung zurücksetzen)
  - SYN (synchronisiere Verbindung) am anfang des verbindungsbaus
  - FIN (beende Verbindung)
- **AdvertizedWindow**: Fenstergröße für Flusssteuerung
- **checksum**: Prüfsumme (wie UDP)

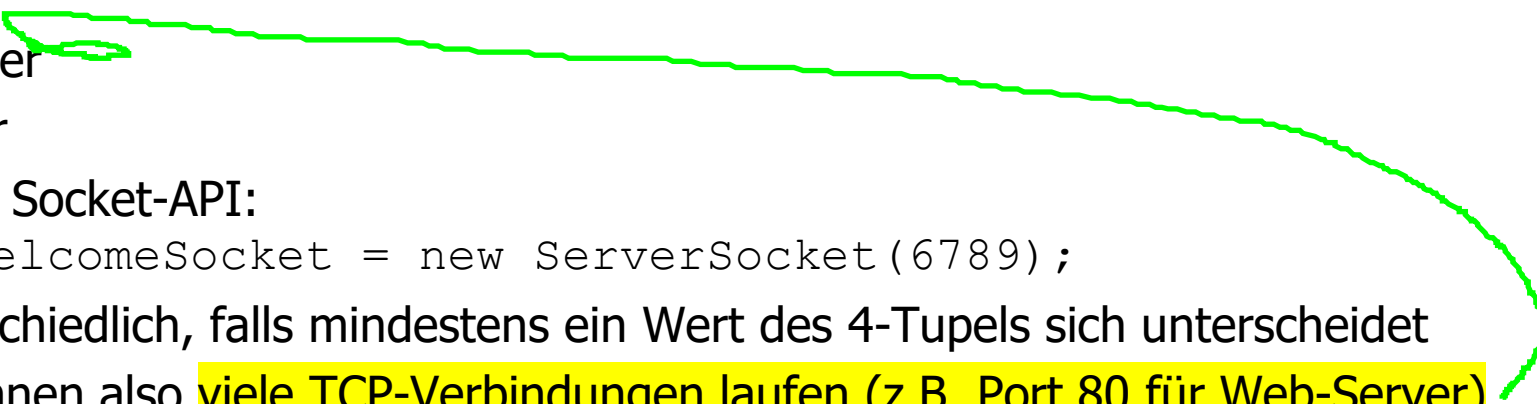
Quelle: Kurose, Ross.  
*Computer Networking:  
A Top-Down Approach*,  
7th Ed., Pearson  
Education, 2020.



# TCP

## ■ Multiplexen und Demultiplexen

- TCP-Verbindung eindeutig gekennzeichnet durch 4-Tupel
    - Quell-IP-Adresse
    - Ziel-IP-Adresse
    - Quellportnummer
    - Zielportnummer
  - realisiert z.B. durch Socket-API:  

```
ServerSocket welcomeSocket = new ServerSocket(6789);
```
  - Sockets sind unterschiedlich, falls mindestens ein Wert des 4-Tupels sich unterscheidet
  - über einen Port können also viele TCP-Verbindungen laufen (z.B. Port 80 für Web-Server)
- 

man kann damit die aufrufe von mehreren clients unterscheiden

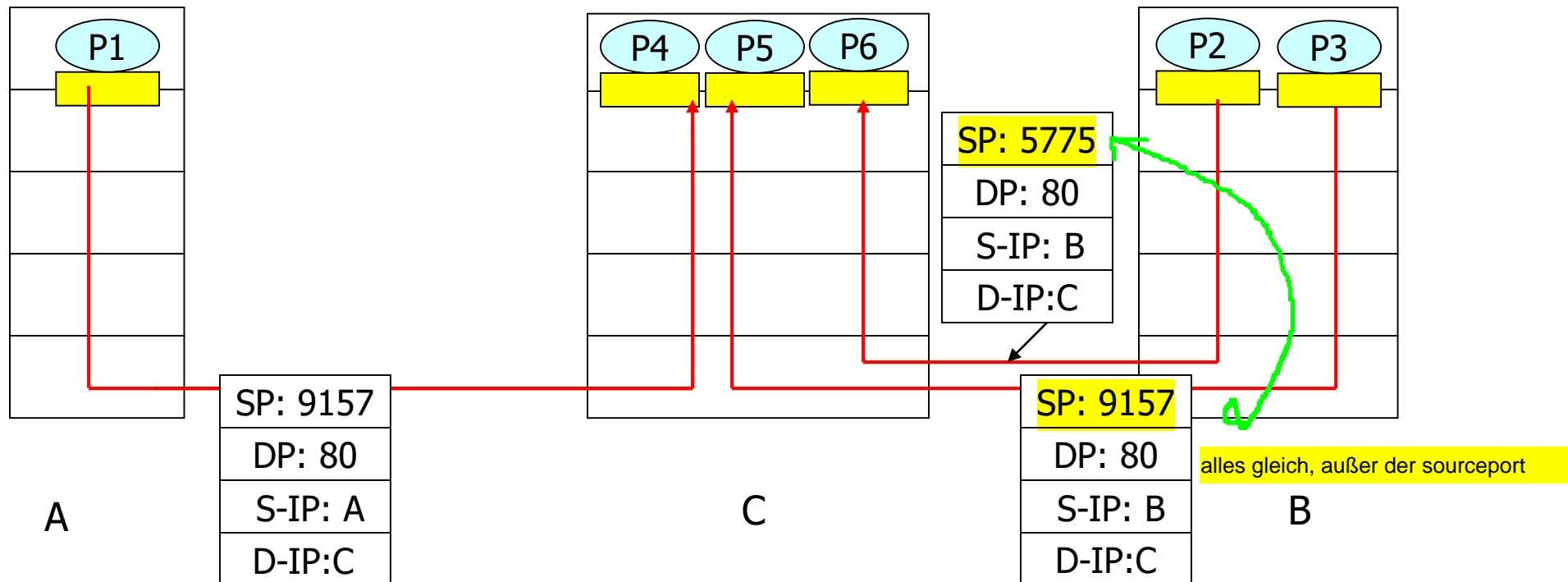
## ■ Pseudo-Header

- wie in UDP, einschließlich Prüfsummenberechnung

# TCP

## ■ Multiplexen und Demultiplexen, Beispiel:

Quelle: Kurose, Ross.  
*Computer Networking: A Top-Down Approach*, 7th Ed.,  
Pearson Education, 2017.



# Transportschicht

- Einführung
- UDP
- Fehlerkontrolle
- TCP
  - Segmentformat
  - Fehlerkontrolle
  - Verbindungsauf- und -abbau
  - Schätzung der RTT
  - Fluss- und Überlastkontrolle
  - Leistungsanalyse
  - Multipath TCP
- TLS
- QUIC

# TCP: Fehlerkontrolle

## ■ Fehlerkontrolle in TCP

- Mischform von Go-Back-N und Selective Repeat und weiterer Elemente
  - Puffer auf Sender- und Empfängerseite selektive
  - ein Timer nicht N für jedes Paket, Go Back n
  - kumulative ACKs go back n
  - Sequenz- und ACK-Nummern beziehen sich nicht auf Pakete also wie ein seek-head in einer datei. Beide teilen mit, was sie als
    - Sequenznummer = Position des ersten Bytes des Segments im Bytestrom
    - ACK-Nummer = Position des nächsten erwarteten Bytes im Bytestrom
- diverse Implementierungsoptionen, im Folgenden wird eine vereinfachte „Standardform“ beschrieben
- außerdem wird im Folgenden wegen der Übersichtlichkeit die Behandlung von Bitfehlern nicht beschrieben, sie können genauso wie bei Go-Back-N oder Selective Repeat behandelt werden



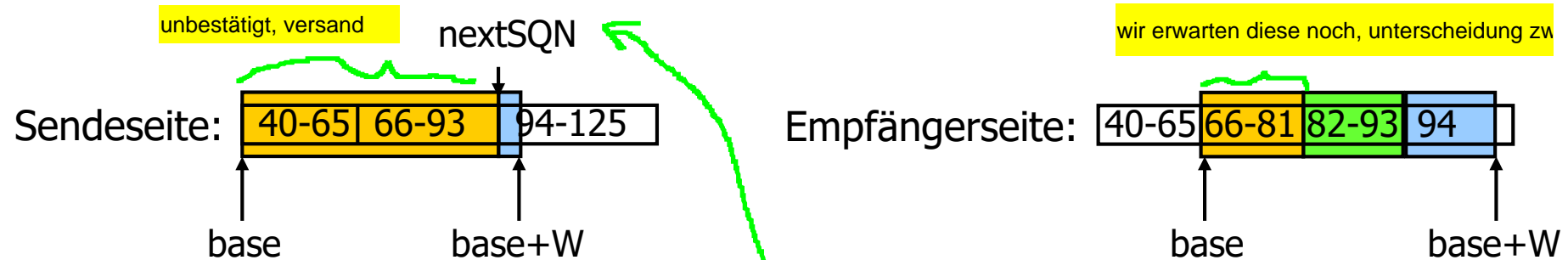
# TCP: Fehlerkontrolle

## ■ Überblick über die Fehlerkontrolle bei TCP

- der Sender darf mehrere Segmente vor Erhalt eines ACKs senden (bis zu einer von verschiedenen Mechanismen abhängigen maximalen Gesamtzahl von Bytes) Schiebefenster, flex fenstergröße für last/Flusskontrolle
- er startet beim Senden des ersten Segments eines Fensters einen Timer
- er puffert die unbestätigten Segmente
- wenn der Timer abläuft, wird das erste unbestätigte Segment des Fensters erneut gesendet
- der Empfänger schickt kumulative ACKs mit der Position des ersten noch nicht empfangenen Bytes
- das Fenster wird auf Sender- und Empfängerseite immer bis zur nächsten Lücke geschoben

# TCP: Fehlerkontrolle

## ■ Sende- und Empfängerfenster



- base: erstes Byte des Fensters
- base+W: erstes Byte außerhalb des Fensters
- nextSQN: erstes Byte des nächsten noch nicht gesendeten Segments
- das Fenster auf Sendeseite enthält versendete unbestätigte und ungesendete Pakete
- das Fenster auf Empfängerseite enthält empfangene Pakete und Lücken und Platz für unempfangene Pakete

# TCP: Fehlerkontrolle

## ■ informelle Beschreibung des Protokolls

### ● Verhalten des Senders

1. wenn Daten zum Senden und Platz im Fenster: erstelle Segment mit nextSQN und sende es mit IP, erhöhe nextSQN um Länge der Daten; wenn es das erste Paket im Fenster ist, starte Timer
2. wenn ein ACK mit ACK-Nr. im Fenster zurückkommt, schiebe das Fenster bis zu dieser ACK-Nr.; wenn das Fenster leer ist, stoppe den Timer, sonst starte den Timer neu
3. wenn der Timeout abläuft, sende das erste unbestätigte Paket des Fensters erneut, starte den Timer erneut

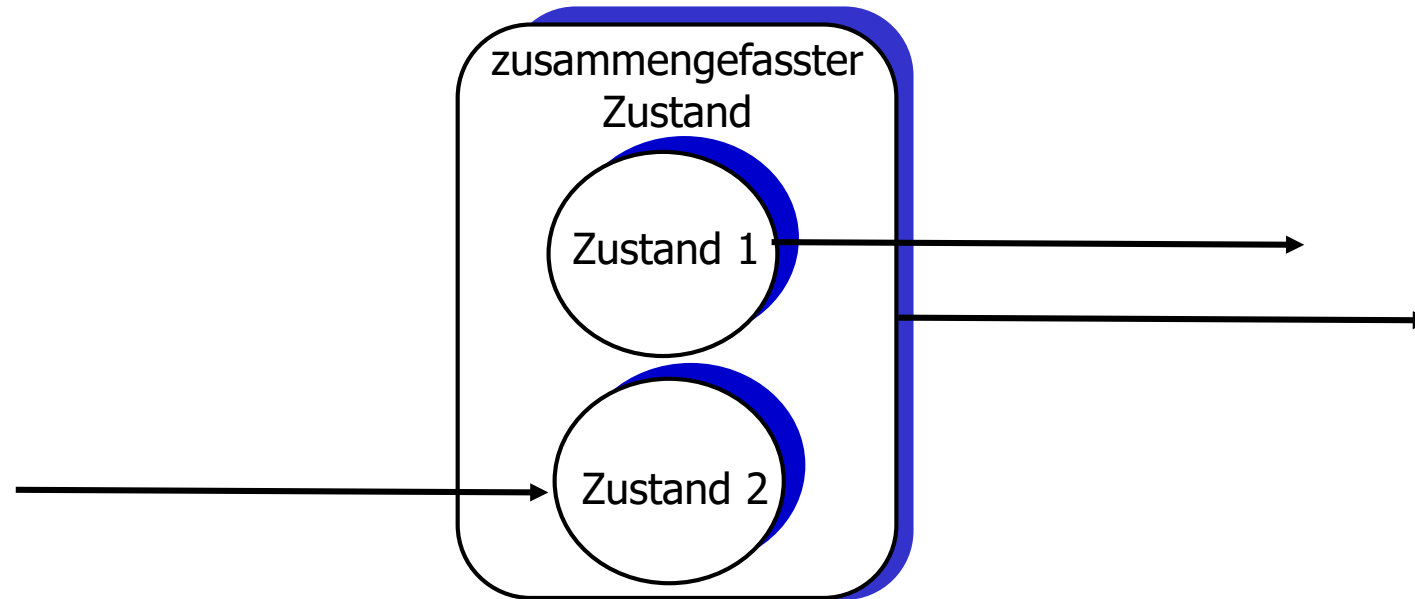
# TCP: Fehlerkontrolle

- Verhalten des Empfängers
  - wenn ein Segment ankommt und
    - **SEQN = Fensteranfang** ist und alle vorherigen Segmente bereits bestätigt sind: schiebe Fensteranfang bis zum nächsten erwarteten Byte und warte ein Timeout (500 ms), wenn bis dahin **kein neues Segment** ankommt, **schicke ein ACK mit dem Fensteranfang** (**delayed ACK**)  
das ACK ist verloren gegangen
    - **SEQN = Fensteranfang** ist und ein vorheriges Segment noch nicht bestätigt wurde, schiebe Fensteranfang bis zum nächsten erwarteten Byte und schicke **sofort ein kumulatives ACK mit dem Fensteranfang**  
da wo das nächste bit erwartet wird
    - **SEQN > Fensteranfang** ist, puffere die Daten und schicke sofort ein kumulatives ACK mit dem Fensteranfang  
noch unvollständiges Fenster, also erwarte ich immernoch das gleiche Fenster
    - **es eine Lücke teilweise oder ganz füllt**, puffere die Daten, schiebe **Fensteranfang bis zum nächsten erwarteten Byte** und **schicke sofort ein kumulatives ACK mit dem Fensteranfang**  
Lücke gefüllt, also kann das fenster weiter gehen. ACK dafür senden

# TCP: Fehlerkontrolle

## ■ Beschreibung durch Statecharts

- neues Element: zusammengefasste Zustände



- grafische Vereinfachung: Zustandsübergänge, die an einem zusammengefassten Zustand beginnen, gelten für jeden inneren Zustand
- Zustandsübergänge können auch direkt an inneren Zuständen beginnen

# TCP: Sender

$[nextSQN \geq base + W]$  /  
signal refusal to application

$[nextSQN < base + W]$  /  
segment[nextSQN] =  
TCPsegment(nextSQN, data, checksum);  
IP\_send(segment[nextSQN]);  
if nextSQN = base start\_timer;  
nextSQN += length(data)

am anfang des Fensters

/base=1;  
nextSQN=1

otherwise/

TCP\_send(data)

timeout /  
IP\_send(segment[base]);  
start\_timer

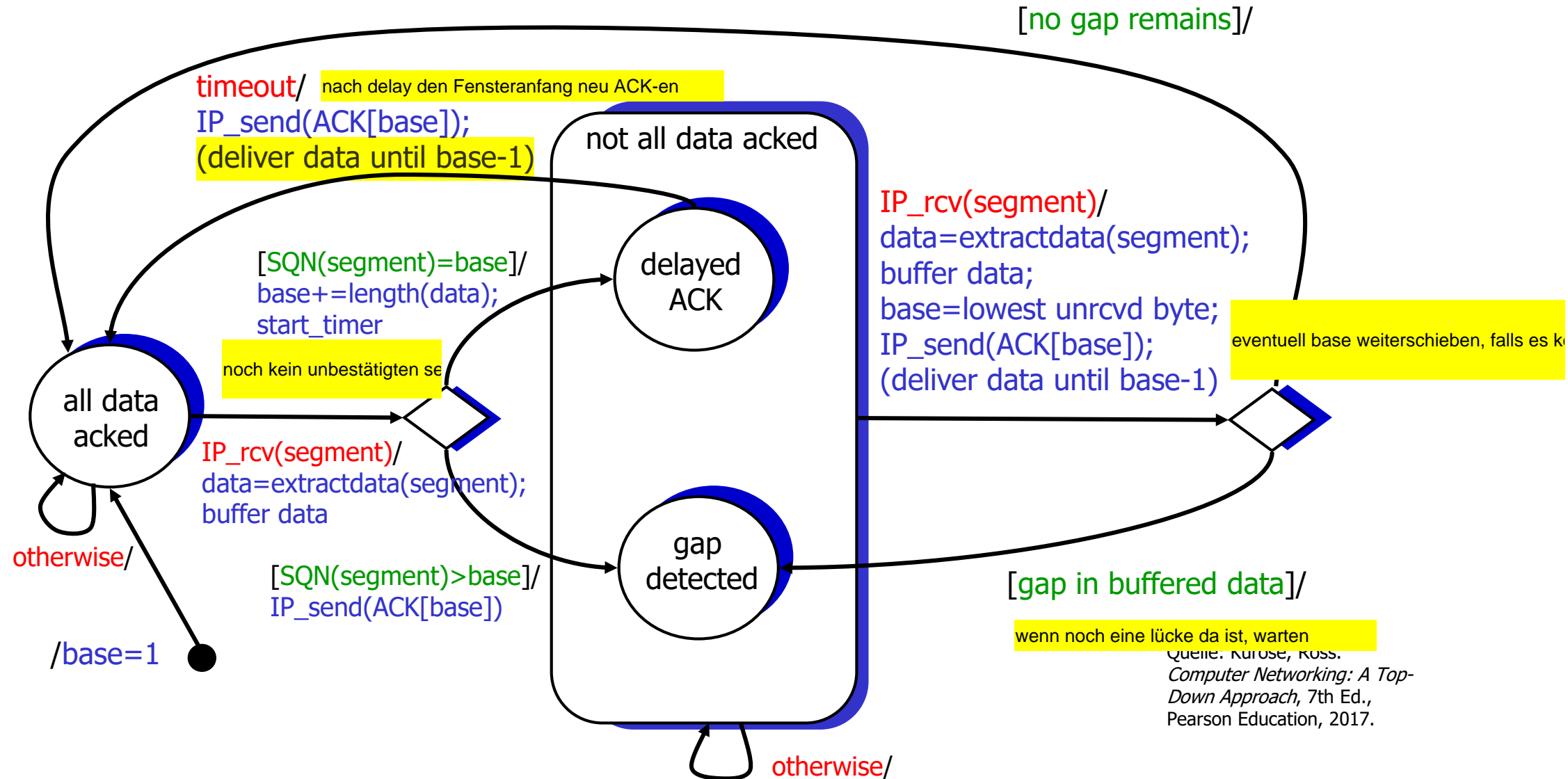
nicht rechtzeitig ack bekommen, vom anfang des Fe

IP\_rcv(ACK)  $[base < acknum(ACK) \leq nextSQN]$  /  
base = acknum(ACK);  
if nextSQN = base stop\_timer else start\_timer

Fenster voll gesendet, weiterschieben

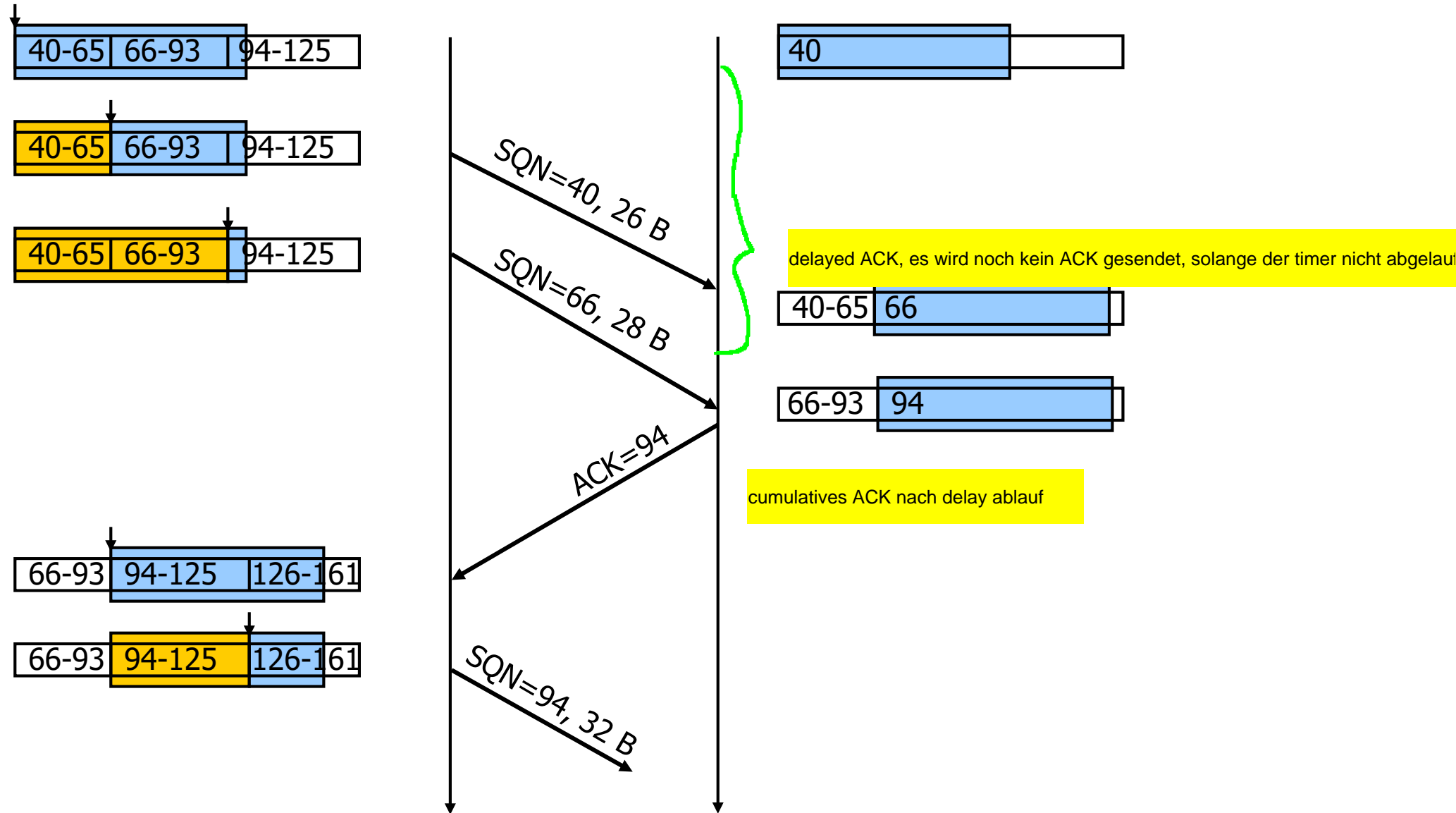
Quelle: Kurose, Ross.  
*Computer Networking: A Top-Down Approach*, 7th Ed.,  
Pearson Education, 2017.

# TCP: Empfänger



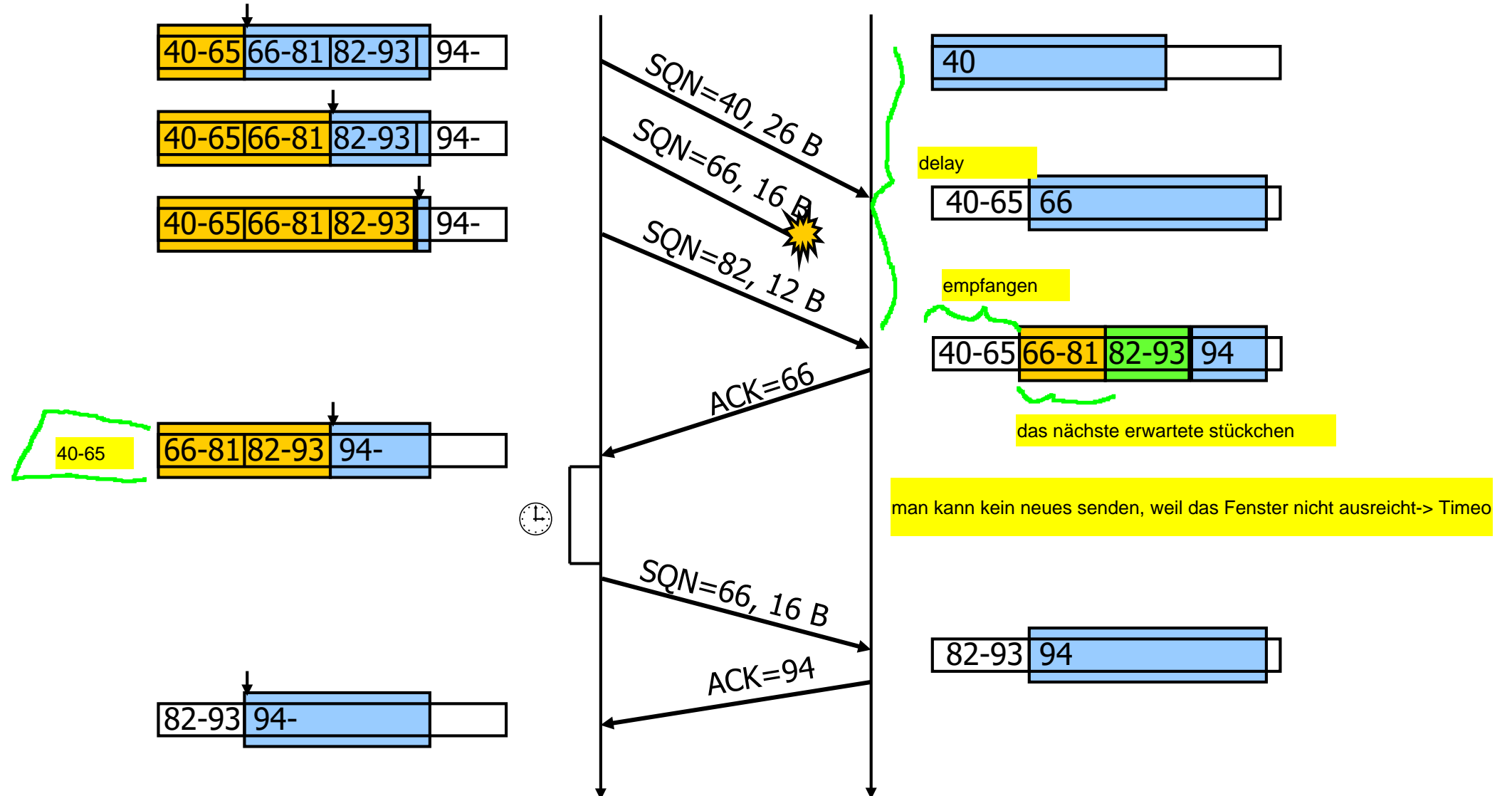
Quelle: Kurose, Ross.  
Computer Networking: A Top-Down Approach, 7th Ed.,  
Pearson Education, 2017.

## TCP: Fehlerkontrolle, normaler Ablauf





# TCP: Fehlerkontrolle, Paketverlust

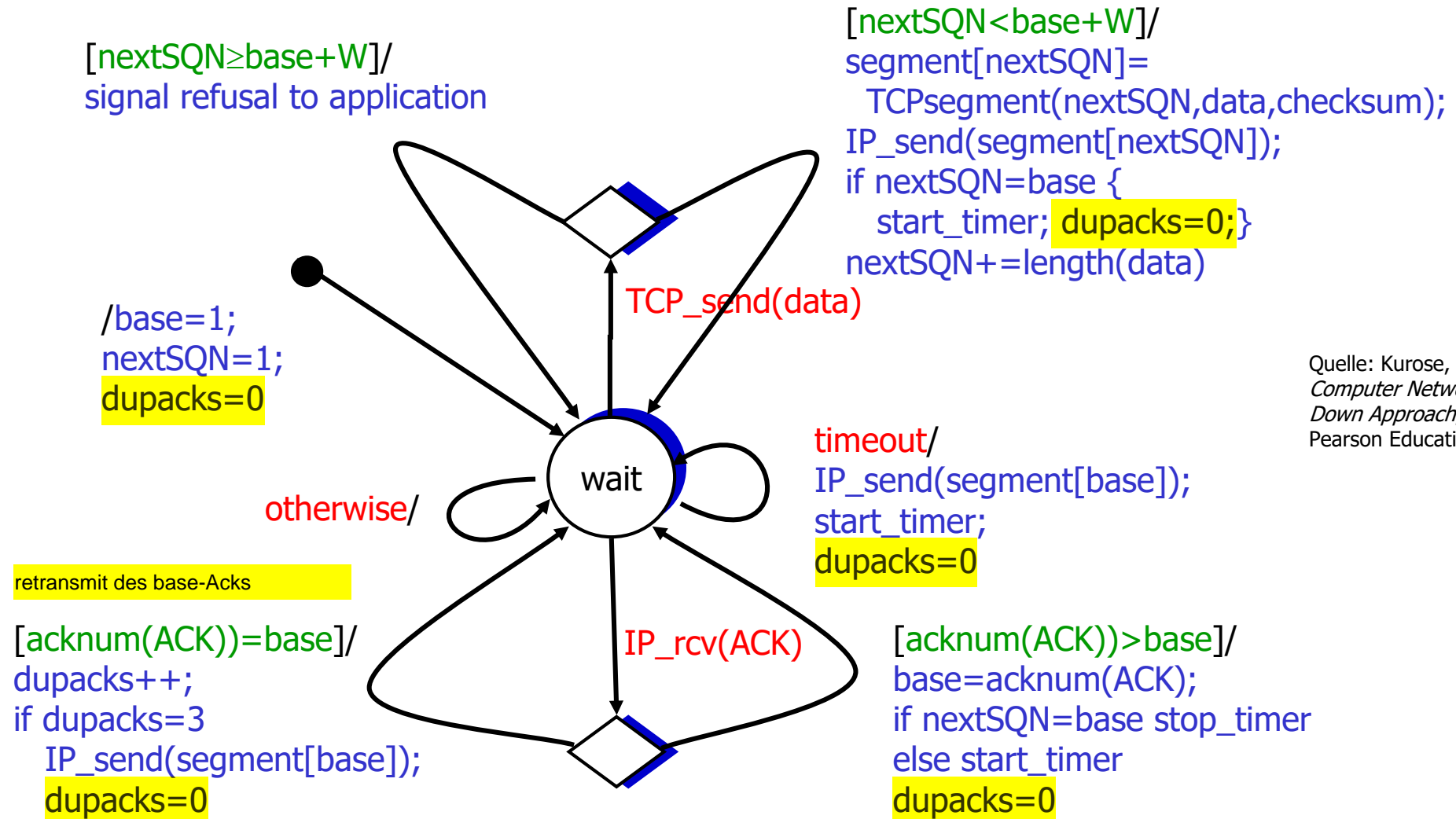


# TCP: Fehlerkontrolle

## ■ Fast Retransmit

- es dauert relativ lange, bis ein Paketverlust bemerkt wird und noch länger bei mehreren Paketverlusten
- ACKs mit der gleichen ACK-Nr. heißen **doppelte ACKs** schnellerer Hinweis als timeout, aber aufpassen bei "verirrten ACKs"
- sie sind ein schnellerer Hinweis auf ein fehlendes Segment
- bei **Fast Retransmit** wird bei **3 doppelten ACKs** (also 4 ACKs mit der gleichen ACK-Nr.) eine Sendewiederholung des Segments mit der SQN ausgelöst
- Anpassung des Statecharts (ein ACK-Zähler **dupacks** wird benötigt):
  - Beim erwarteten ACK wird der dupacks-Zähler auf 0 gesetzt
  - Sollten durch Fehler doppelte ACKs eintreffen, wird dupacks inkrementiert und bei 3 der Fast Retransmit ausgelöst

# TCP Sender mit Fast Retransmit



Quelle: Kurose, Ross.  
 Computer Networking: A Top-Down Approach, 7th Ed.,  
 Pearson Education, 2017.

# TCP: Fehlerkontrolle

## ■ Bemerkungen

- TCP ist **voll duplex**: es werden zwei logische Verbindungen realisiert, eine in jede Richtung
- ACKs reisen Huckepack (**Piggybacking**): Segmente mit Daten in die eine Richtung werden als ACKs in die andere Richtung benutzt data und ACK kombinieren
- das **delayed ACK** soll die Anzahl von ACKs reduzieren

## ■ Offene Probleme

- **mehrfache Paketverluste in einem Fenster haben katastrophalen Effekt** auf den Durchsatz, da der Sender (durch kumulatives ACK) für jedes fehlende Paket mindestens eine Round-Trip-Time warten muss
- Lösung: es gibt eine TCP-Erweiterung **Selective Acknowledgments** (**SACK**), bei der zusätzlich im Optionsfeld selektive ACKs gesendet werden

# TCP: SACK

## ■ Die Grundidee

- Selective Acknowledgments (SACK) informieren den Sender über einzelne Pakete, welche im Fenster liegen und nicht durch kumulative ACKs bestätigt werden können
- der Sender muss diese (nach Timeout) nicht erneut übertragen
- die Informationen werden über TCP Optionsfelder übertragen

also wenn ein paket mitten im Fenster da ist, die vorher

## ■ Allgemeine Regeln

- normale ACKs werden unverändert verschickt (Kompatibilität)
- SACKs werden für das erste Paket außer der Reihe verschickt
- Empfänger: verschickt so viele SACKs wie möglich (Platz im Header)
- Sender: entsprechende Neuübertragungen werden initiiert

## TCP: SACK Headeroptionen

### ■ TCP Sack-Permitted Option

- nur erlaubt wenn SYN Flag gesetzt

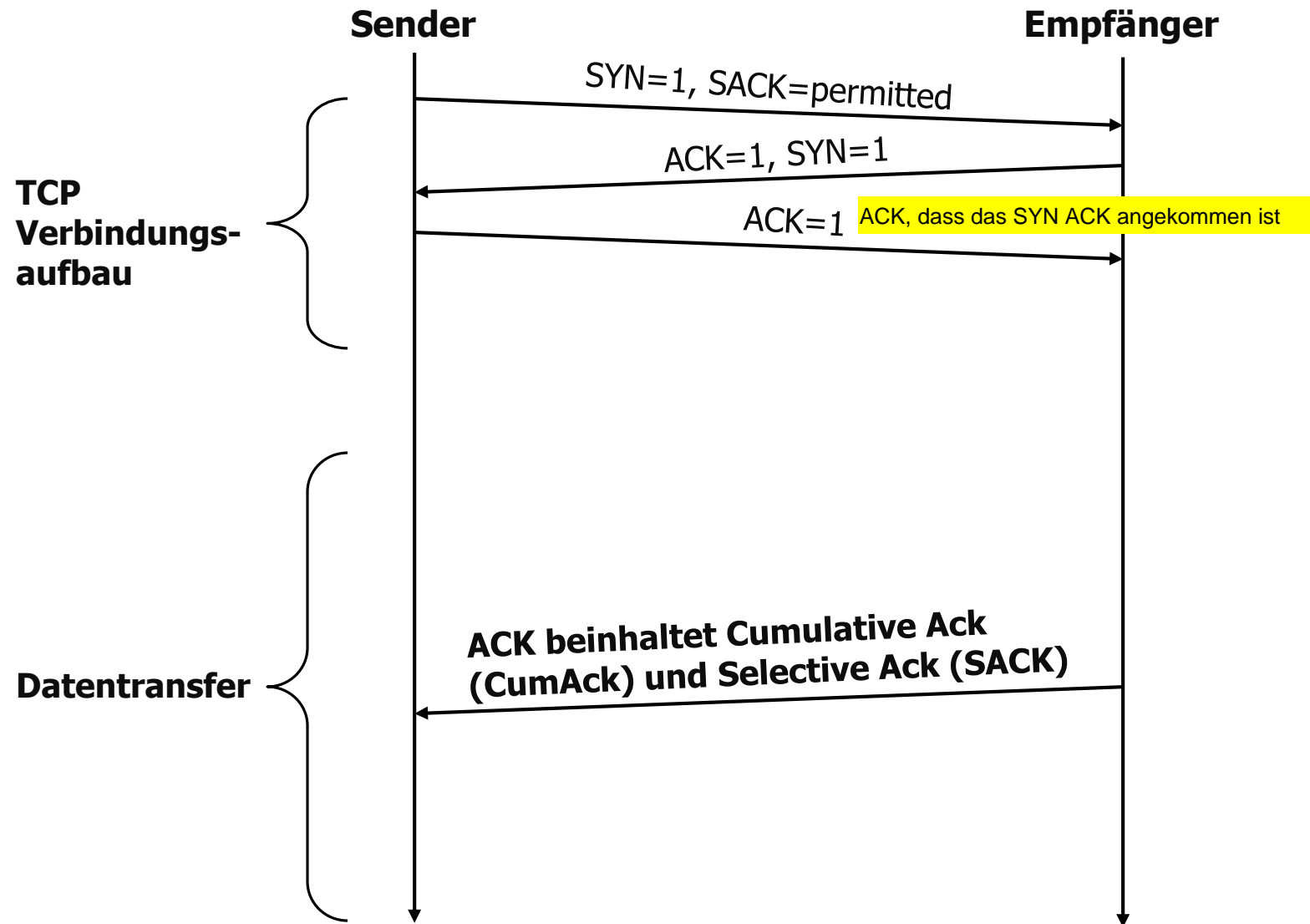
		Art=4	Länge=2
--	--	-------	---------

### ■ Sack Option Format

- jeder empfangene Block geht vom Byte des linken Rand bis vor das Byte des rechten Rands

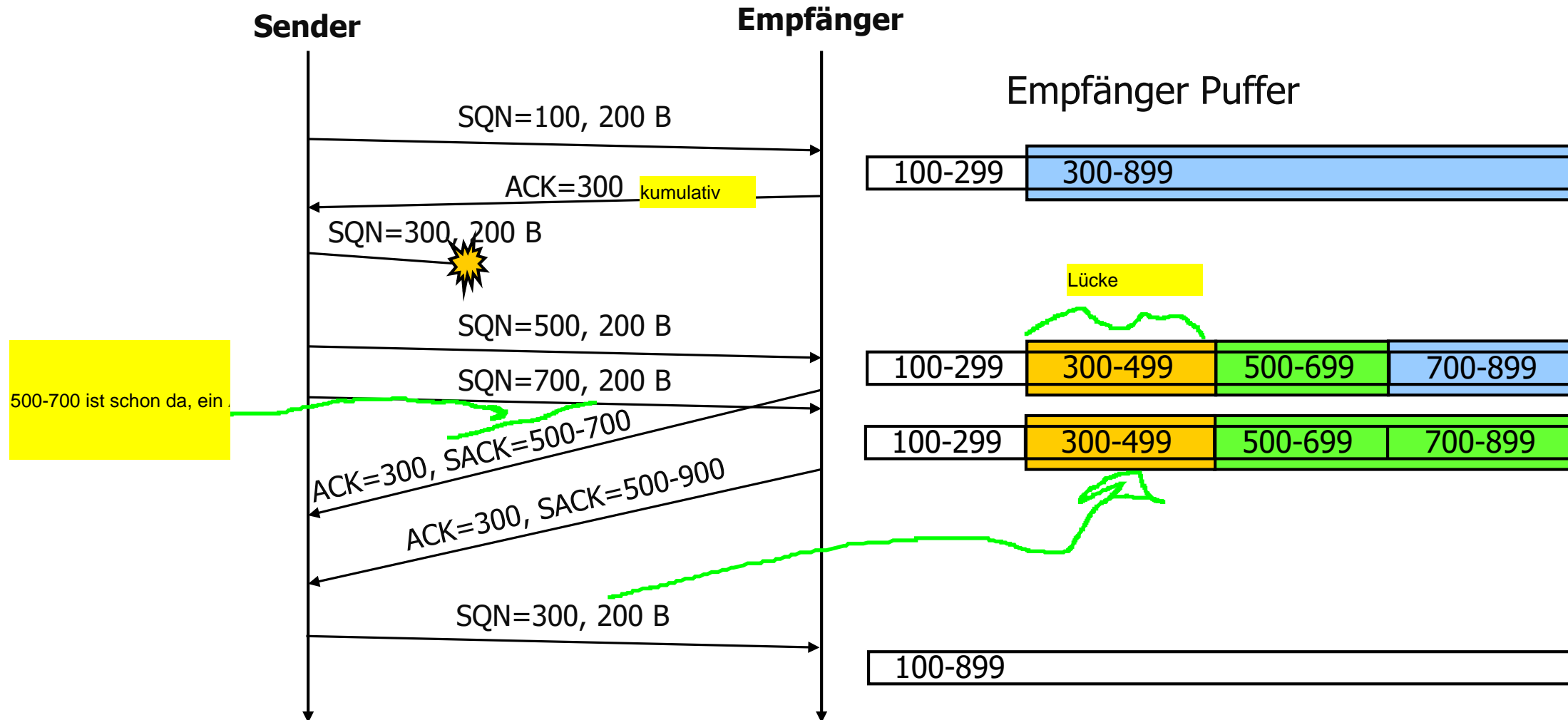
		Art=5	Länge=8n+2
Linker Rand 1. Block	bis zum byte vor dem Rechten Rand(also		
Rechter Rand 1. Block			
...			
Linker Rand n. Block			
Rechter Rand n. Block			

## TCP: SACK – Permitted and SACK



# TCP: SACK Beispiel

- SACKs informieren über (isoliert) empfangene Datenblöcke





# TCP: Fehlerkontrolle

## ■ Größe des Sequenznummerraums

- das Sequenznummerfeld ist 32 Bits groß, es gibt also  $2^{32}$  Sequenznummern
- die Bedingung für Schiebefensterprotokolle ist erfüllt:  $2^{32} > 2^{16}$  Schiebefenstergröße
  - Sequenznummer vs. max. Fenstergröße
- Zeiten für den Überlauf der Sequenznummern
  - bei 10 Mbps: 57 Minuten
  - bei 1 Gbps: 34 Sekunden
- für hohe Bitraten also etwas kurz
- TCP-Erweiterung verwendet Zeitstempel im Options-Feld für weitere Unterscheidung, um Verwechslungen von Segmenten zu vermeiden

# Transportschicht

- Einführung
- UDP
- Fehlerkontrolle
- TCP
  - Segmentformat
  - Fehlerkontrolle
  - Verbindungsauf- und -abbau
  - Schätzung der RTT
  - Fluss- und Überlastkontrolle
  - Leistungsanalyse
  - Multipath TCP
- TLS
- QUIC

# TCP: Verbindungsauf- und -abbau

## ■ Verbindungsaufbau: Anwendungs-API

- aktiver Client:

```
Socket clientSocket = new Socket("hostname", "port");
```

nicer Port-string

- passiver Server:

```
Socket connectionSocket = welcomeSocket.accept();
```

## ■ veranlasst 3-Wege-Handshake

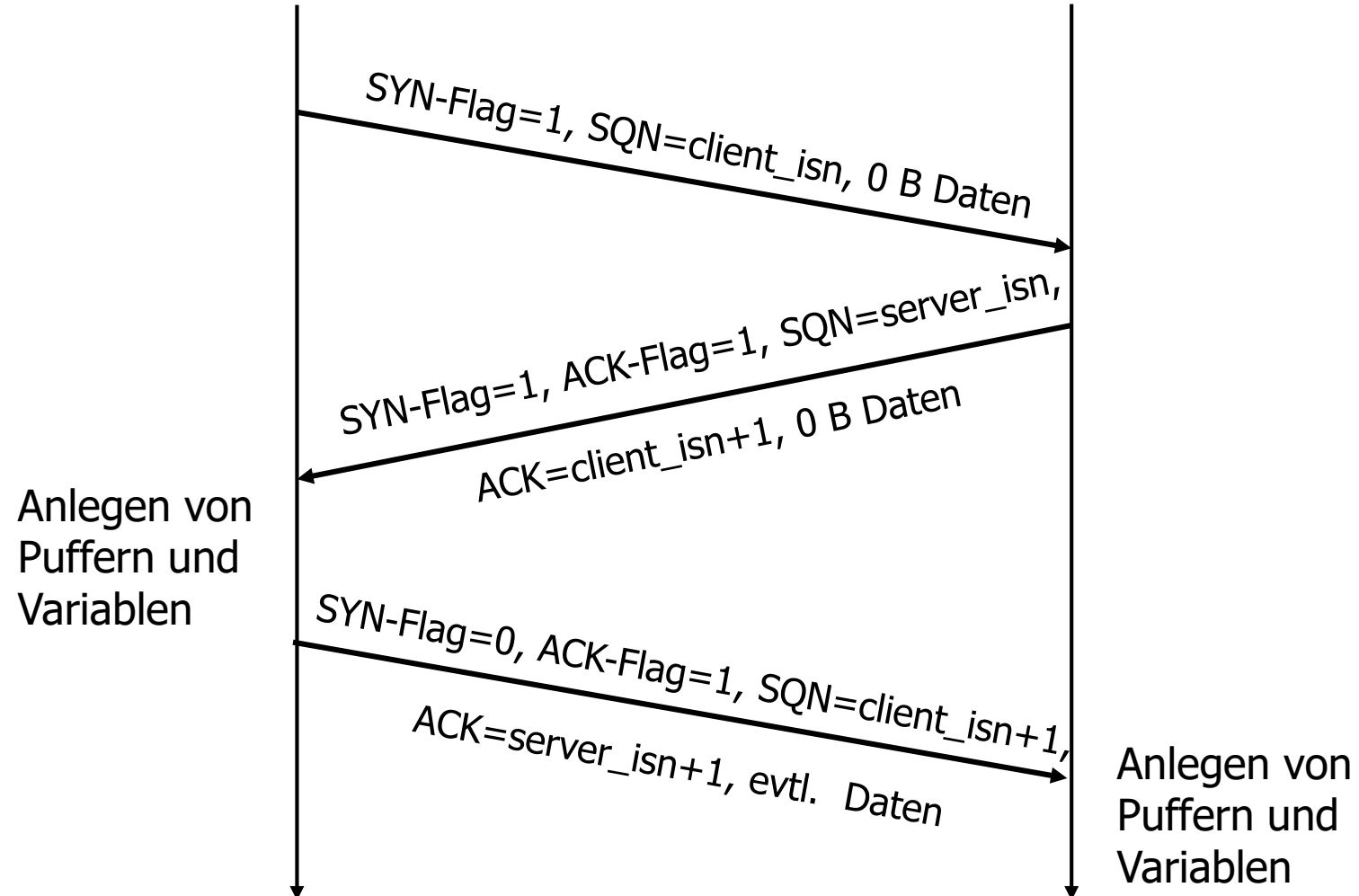
- **SYN-Segment:** Client sendet Segment mit SYN-Flag=1, zufälliger initialer Client-SQN (client\_isn), ohne Daten
- **SYNACK-Segment:** Server sendet Segment mit SYN-Flag=ACK-Flag=1, zufälliger initialer Server-SQN (server\_isn), ACK=client\_isn+1, ohne Daten; er legt Puffer und Variablen an
- **ACK-Segment:** Client sendet Segment mit ACK-Flag=1; SQN=client\_isn+1, ACK=server\_isn+1 und ggfs. Daten; er legt Puffer und Variablen an

Call and response von ACKs. Client beginnt mit einer SQN

In realität sendet noch keiner Daten bei diesem ACK mit

# TCP: Verbindungsauf- und -abbau

## ■ 3-Wege-Handshake:



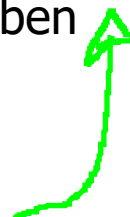
# TCP: Verbindungsauf- und -abbau

## ■ Sequenznummern bei SYN- und FIN-Segmenten

- Segmente mit SYN-Flag=1 oder FIN-Flag=1 dürfen keine Daten enthalten, die nächste SQN muss aber um Eins inkrementiert werden, damit diese Segmente explizit bestätigt werden können

## ■ Verbindungsabbau

- jede Seite kann Verbindungsabbau durch Segment mit FIN-Flag=1 veranlassen
- die andere Seite bestätigt mit ACK-Flag=1
- beide Seiten müssen ihre Hälfte der Verbindung schließen also muss auch ein FIN segment schicken
- hat eine Seite geschlossen, sendet sie keine Daten mehr, nimmt aber noch welche an
- **Time Wait:** die Seite, die den Verbindungsabbau veranlasst, wartet zum Schluss noch 2 Segmentlebensdauern, um noch mögliche alte Segmente zu empfangen (und eine neue TCP-Verbindung davor zu schützen), übliche Werte: 30 s - 2 min., solange werden Variablen der Verbindung gehalten und Socket wird nicht neu vergeben



Man hält das Socket und die Buffer also, damit neue Anwendungen keine alten daten bekommen LÜCKE?

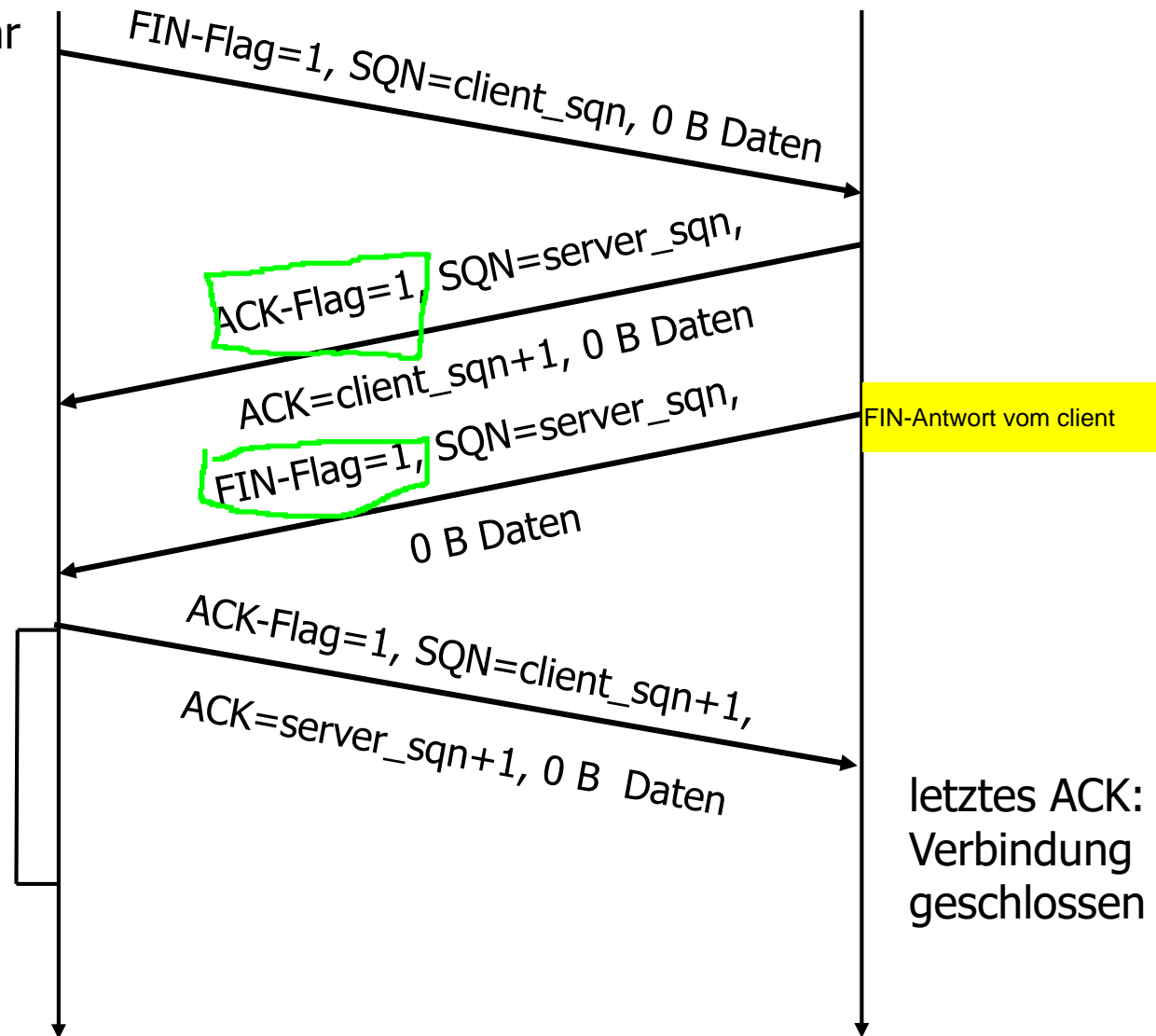
## TCP: Verbindungsauf- und -abbau, Beispiel

linke Seite hat nichts mehr zu senden und bricht ab:

**Time Wait:**

2 Segmentlebensdauern  
auf mögliche alte  
Segmente warten

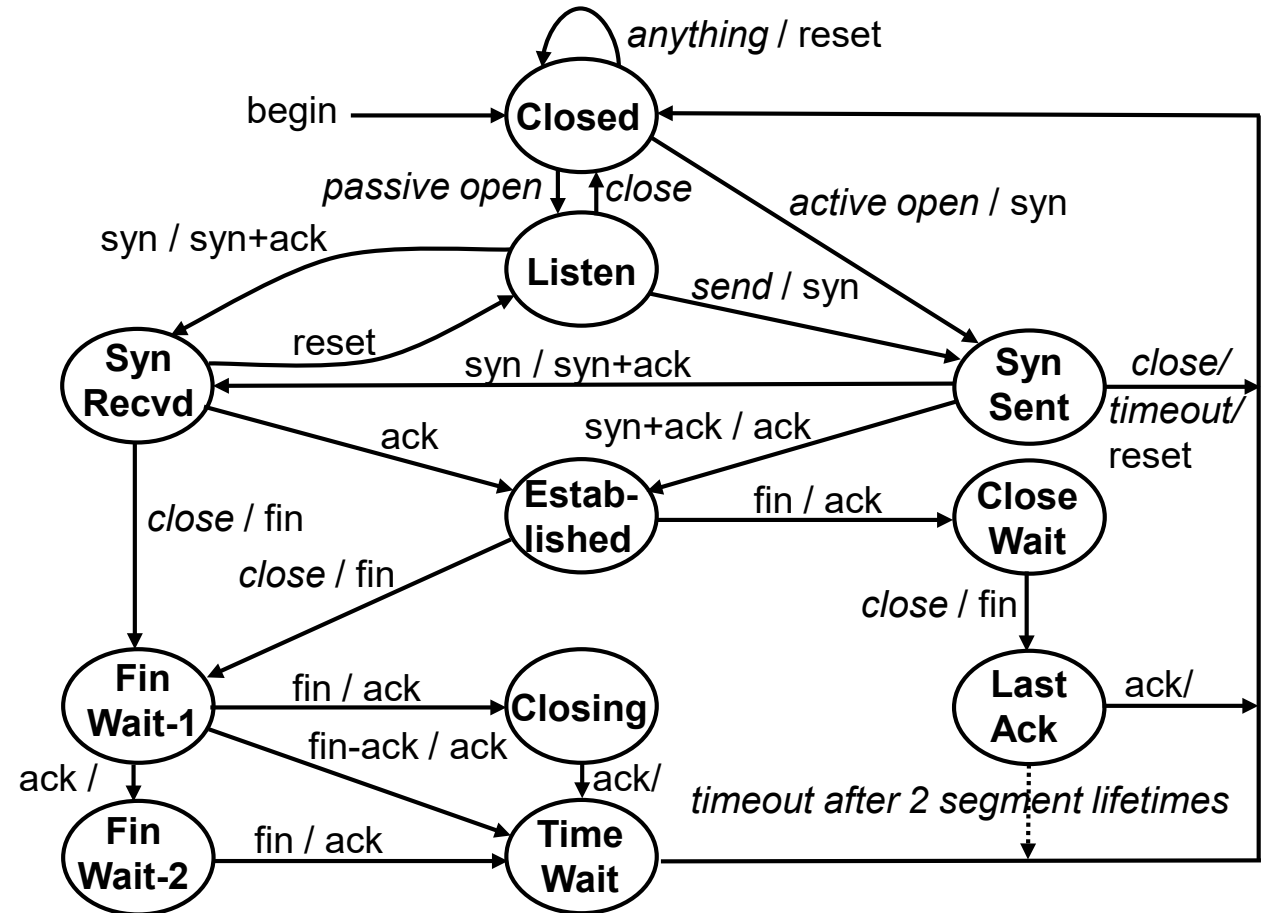
Verbindung geschlossen



# TCP: Verbindungsauf- und -abbau

## ■ Zustandsmaschine

- in RFC 793 ist Zustandsmaschine für Verbindungsauf- und -abbau
- verbesserte Version, enthält alle Möglichkeiten:



# Transportschicht

- Einführung
- UDP
- Fehlerkontrolle
- TCP
  - Segmentformat
  - Fehlerkontrolle
  - Verbindungsauf- und -abbau
  - **Schätzung der RTT**
  - Fluss- und Überlastkontrolle
  - Leistungsanalyse
  - Multipath TCP
- TLS
- QUIC



# TCP: Schätzung der RTT

## ■ Timeout für Sendewiederholungen

- der Sender muss einen Timeout wählen
- ein ACK kann **frühestens nach RTT** zurückkommen **Round-Trip-Time**
- ist der Timeout zu **klein**, gibt es unnötige Sendewiederholungen
- ist der Timeout zu **groß**, kann erst spät auf Fehler reagiert werden
- der passende Timeout hängt von der Konfiguration ab und **ändert sich dynamisch**
- Vorgehen von TCP
  - **Zeitstempel** für Segment und ACK,  
**Differenz = Messung der aktuellen RTT**
  - mean und std mit Toleranz** **Durchschnitt** und **Abweichung** aus mehreren Messungen bestimmen, daraus Timeout ableiten
  - Messungen **bei Sendewiederholungen nicht verwenden**

man weiß nicht, wann genau diese losgeschickt wurden. Nur die neuesten ACKs

# TCP: Schätzung der RTT

## ■ Bestimmung der RTT

- jede Messung ergibt ein SampleRTT
- gleitender Durchschnitt (Exponentially Weighted Moving Average):  
$$\text{EstimatedRTT} = (1-\alpha) \times \text{EstimatedRTT} + \alpha \times \text{SampleRTT}$$
genau F\*CKING umgekehrt wie es jeder andere macht
- bei großem  $\alpha$  reagiert Durchschnitt stark auf aktuelle Schwankungen, bei kleinem  $\alpha$  gibt es größere Stabilität, aber langsamere Reaktion auf Änderungen, typischer Wert:  $\alpha = 0,125$

## ■ mittlere Abweichung

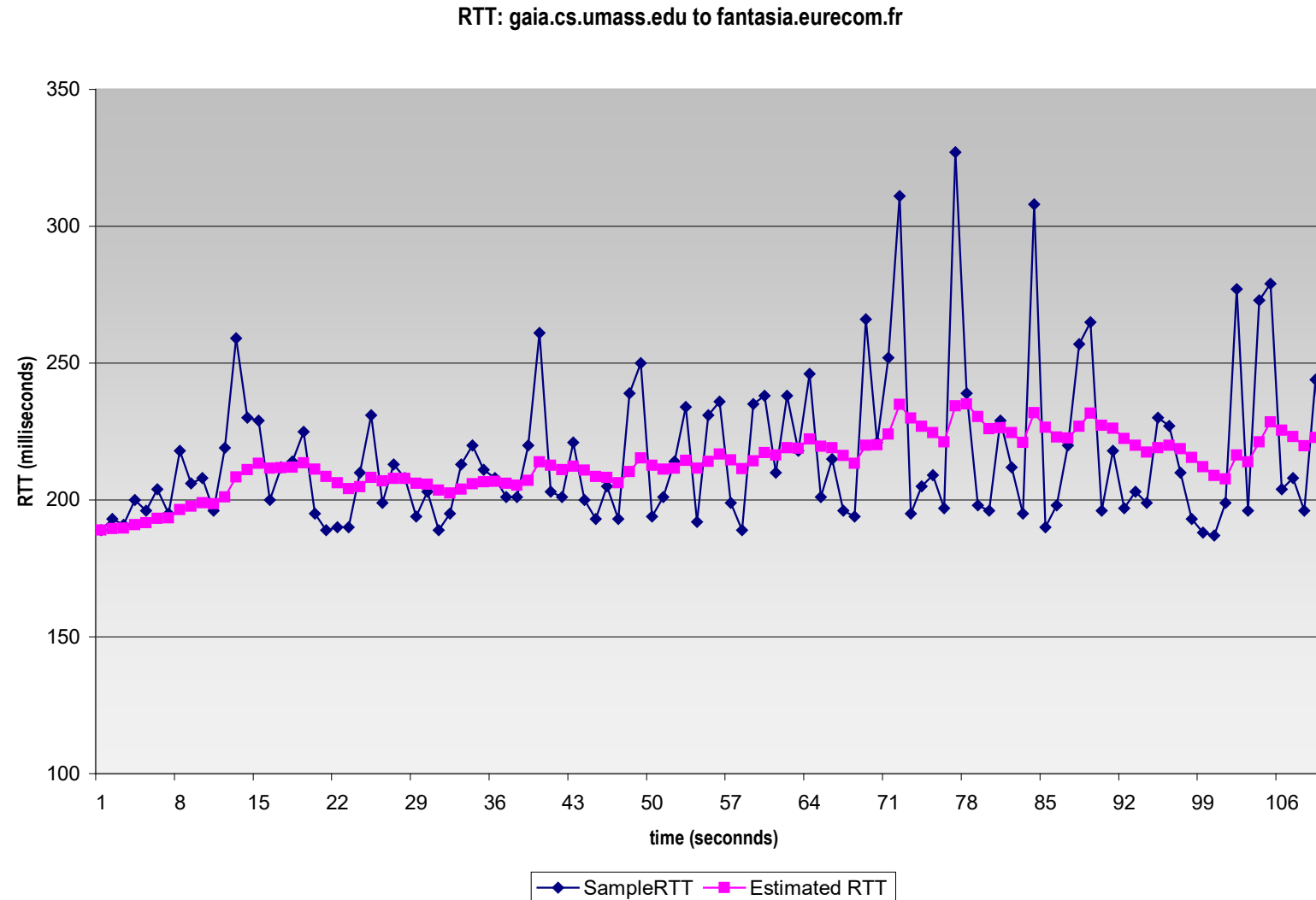
- wieder als gleitender Durchschnitt, ähnlich zu Standardabweichung
- $\text{DevRTT} = (1-\beta) \times \text{DevRTT} + \beta \times |\text{SampleRTT} - \text{EstimatedRTT}|$ moving average von std
- typisch:  $\beta = 0,25$

## ■ Timeout

- geschätzte RTT + aus Abweichung abgeleitete Sicherheit:
- $\text{TimeoutInterval} = \text{EstimatedRTT} + 4 \times \text{DevRTT}$
- **Timeout Backoff:** wenn der Timeout ausgelöst wird, wird er jeweils verdoppelt und wird benutzt, bis neues SampleRTT da ist  
also als sicherheit, falls plötzlich irgendwas mit der Leitung passiert ist

# TCP: Schätzung der RTT

## ■ Beispiel für RTT-Schätzung:



# Transportschicht

- Einführung
- UDP
- Fehlerkontrolle
- TCP
  - Segmentformat
  - Fehlerkontrolle
  - Verbindungsauf- und -abbau
  - Schätzung der RTT
  - Fluss- und Überlastkontrolle
  - Leistungsanalyse
  - Multipath TCP
- TLS
- QUIC

# TCP: Fluss- und Überlastkontrolle

## ■ Flusskontrolle (Flow Control)

- Mechanismus, mit dem der Empfänger den Sender steuern kann, damit er ihn nicht überlastet aber auch so schnell sendet wie möglich
- üblicherweise durch Benachrichtigung über Fenstergröße

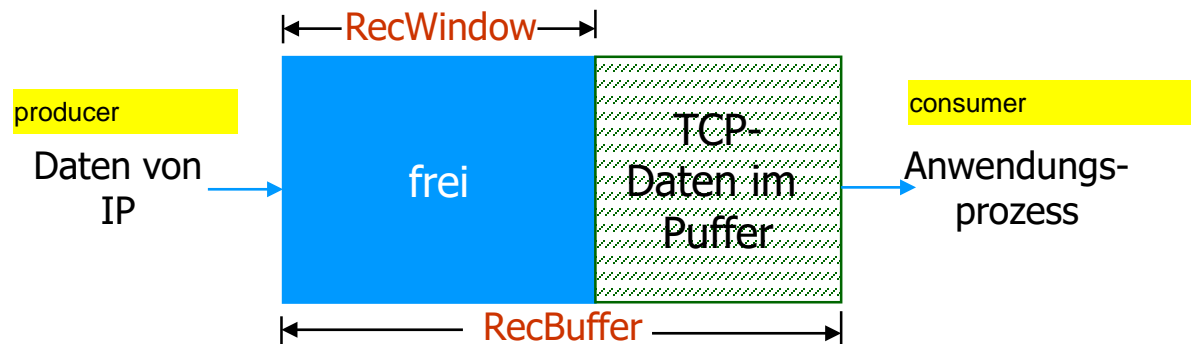
## ■ Überlastkontrolle (Congestion Control)

- Mechanismus, mit dem der Sender davon abgehalten wird, das Netz (Bitraten und Puffer) zu überlasten  
spez. Puffer in Router
- kann durch explizite Signale des Netzes an den Sender erfolgen
  - z.B. mit TCP/IP zur expliziten Überlastbenachrichtigung: Explicit Congestion Notification (ECN)
- häufig wird jedoch kein solcher expliziter Mechanismus verwendet, stattdessen leitet sich der Sender aus den zurückkommenden ACKs Informationen über den Netzzustand ab und reagiert entsprechend
  - z.B. doppelte ACKs (duplicate ACKs), erhöhte RTTs

# TCP: Flusskontrolle

## ■ Prinzip der Flusskontrolle

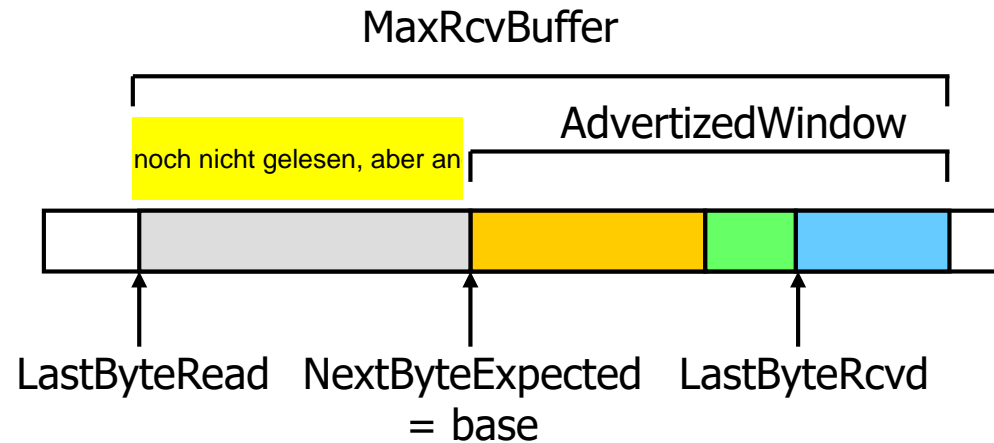
- die Empfängerseite besitzt einen Puffer, IP fügt neue empfangene Daten ein, die Anwendung liest Daten aus



- der jeweils freie Pufferplatz wird der Senderseite mitgeteilt
  - die Senderseite besitzt einen Puffer, in den die Anwendung neue Daten schreibt und mit IP soviel Daten entfernt werden, wie es der Puffer der Empfangsseite zulässt
  - die Anwendung auf Sendeseite blockiert, wenn der Puffer voll ist
  - dadurch reguliert die Empfängeranwendung die Senderanwendung
- also wenn der Empfänger n bit platz hat, dann sendet der Sender

# TCP: Flusskontrolle

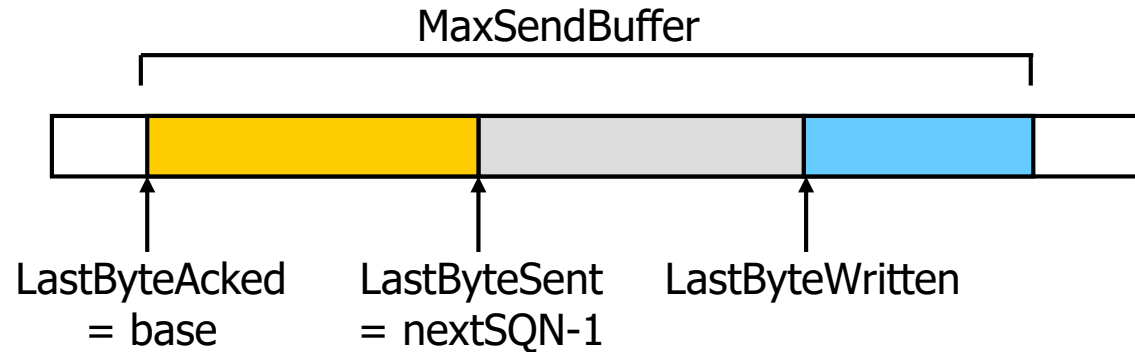
## ■ Puffer auf Empfängerseite



- LastByteRead: das letzte an die Anwendung ausgelieferte Byte
- NextByteExpected: das nächste erwartete Byte
- LastByteRcvd: das letzte empfangene Byte
- MaxRcvBuffer: insgesamt zur Verfügung stehender Pufferplatz
- **AdvertizedWindow** =  $\text{MaxRcvBuffer} - ((\text{NextByteExpected} - 1) - \text{LastByteRead})$ : freier Pufferplatz, wird dem Sender mitgeteilt  
also Max-grau

# TCP: Flusskontrolle

## ■ Puffer auf Senderseite



- LastByteAcked: das letzte bestätigte Byte
- LastByteSent: das letzte gesendete Byte
- LastByteWritten: das letzte von der Anwendung geschriebene Byte
- MaxSendBuffer: insgesamt zur Verfügung stehender Pufferplatz
- **EffectiveWindow** = AdvertizedWindow - (LastByteSent - LastByteAcked)
- Sender sendet nur, falls **EffectiveWindow > 0**
- Anwendung schreibt nur, falls **LastByteWritten - LastByteAcked ≤ MaxSendBuffer**

das effektive ist also was der Client hat, minus das orange

also noch blau im buffer ist



# TCP: Flusskontrolle

## ■ Bemerkungen zum Ablauf der Flusskontrolle

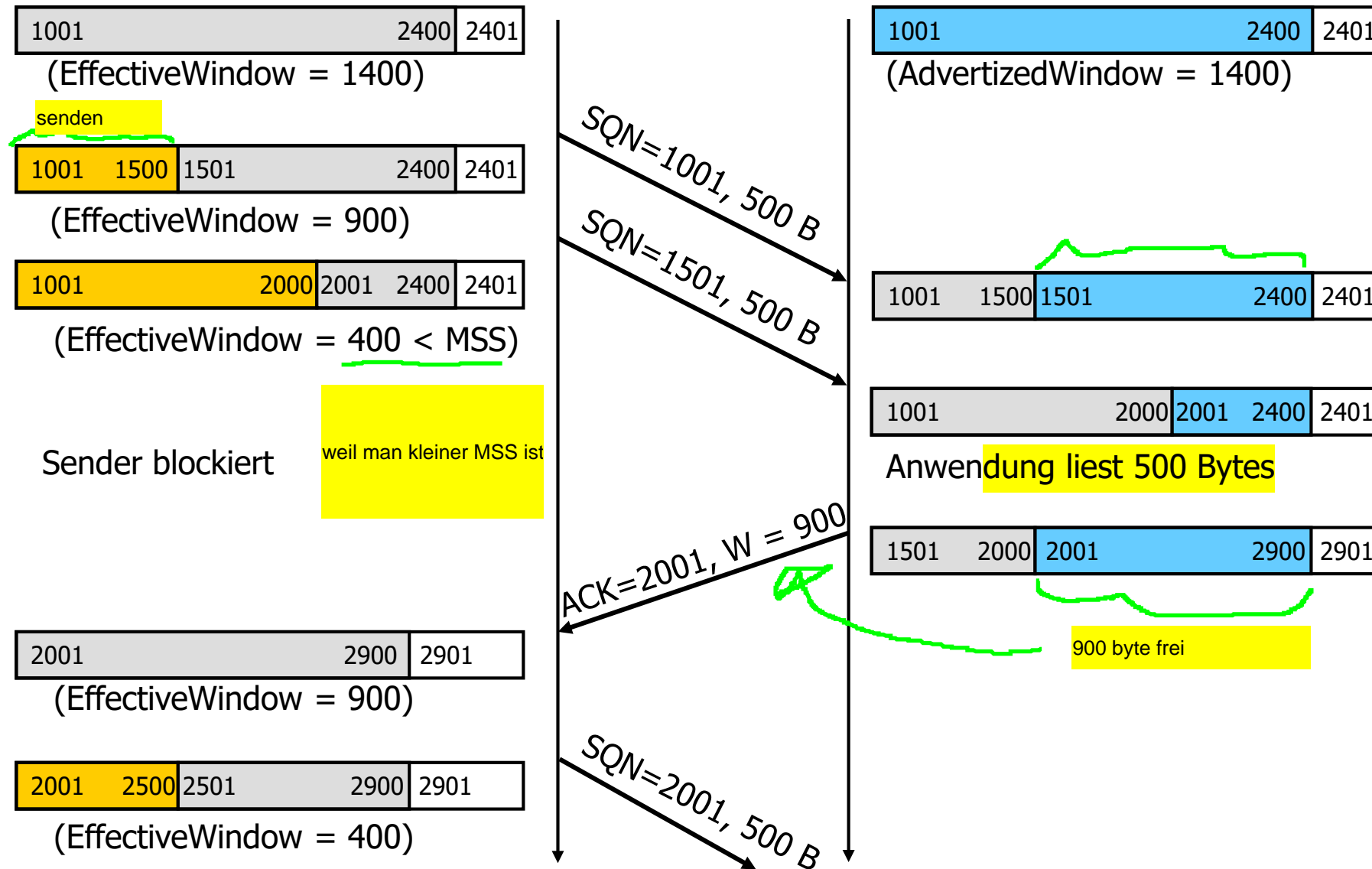
- initial wird AdvertizedWindow möglichst groß eingestellt
- nach AdvertizedWindow = 0 werden periodisch Sondensegmente mit 1 Byte gesendet, sonst kommen evtl. nie ACKs mit wieder größerem AdvertizedWindow zurück
- Vermeidung des Silly Window Syndroms
  - Segmente mit wenig Daten sind ineffizient
  - wenn die Puffer voll sind und kleine Segmente gesendet werden, so werden ACKs für diese gesendet und der Sender wird erneut ein kleines Segment senden usw. also man sendet nur noch winzige pakete, und di
  - Abhilfe: MSS (Maximum Segment Size), Default sind 536 Bytes
  - auf Empfängerseite: wenn der Empfänger ein AdvertizedWindow = 0 bekannt gibt, wartet er danach bis er ein AdvertizedWindow  $\geq$  MSS bekannt geben kann
  - auf Senderseite: Sender sendet kleineres Segment als MSS nur, wenn keine weiteren unbestätigten Segmente unterwegs sind

also man geht nur unter die MSS mindestgröße, wenn man sonst schon alles verschickt hat

# TCP: Flusskontrolle, Beispiel

MaxSendBuffer=MaxRcvBuffer=1400 Bytes

MSS = 500 Bytes



# TCP: Flusskontrolle

## ■ Ist das Fenster groß genug?

- kann es Verwechslungen von Segmenten geben?
  - das AdvertizedWindow-Feld ist 16 Bits groß, also kann das Fenster  $2^{16}$  Bytes groß sein
  - die Bedingung für Schiebefensterprotokolle ist erfüllt:  $2^{32} \gg 2 \cdot 2^{16}$
- kann der Sender den Kanal gefüllt halten?
  - auf der nächsten Folie ist zu sehen, dass für manche Konstellationen das Bitraten-Verzögerungs-Produkt so groß ist, dass das maximale Fenster dafür nicht ausreicht
  - als Abhilfe kann im Options-Feld des ersten Segments ein Window-Scale-Faktor  $F \leq 14$  gesetzt werden, die Fenstergröße ergibt sich dann immer aus  $\text{AdvertizedWindow} \cdot 2^F$
  - es gilt immer noch  $2^{32} > 2 \cdot 2^{30}$

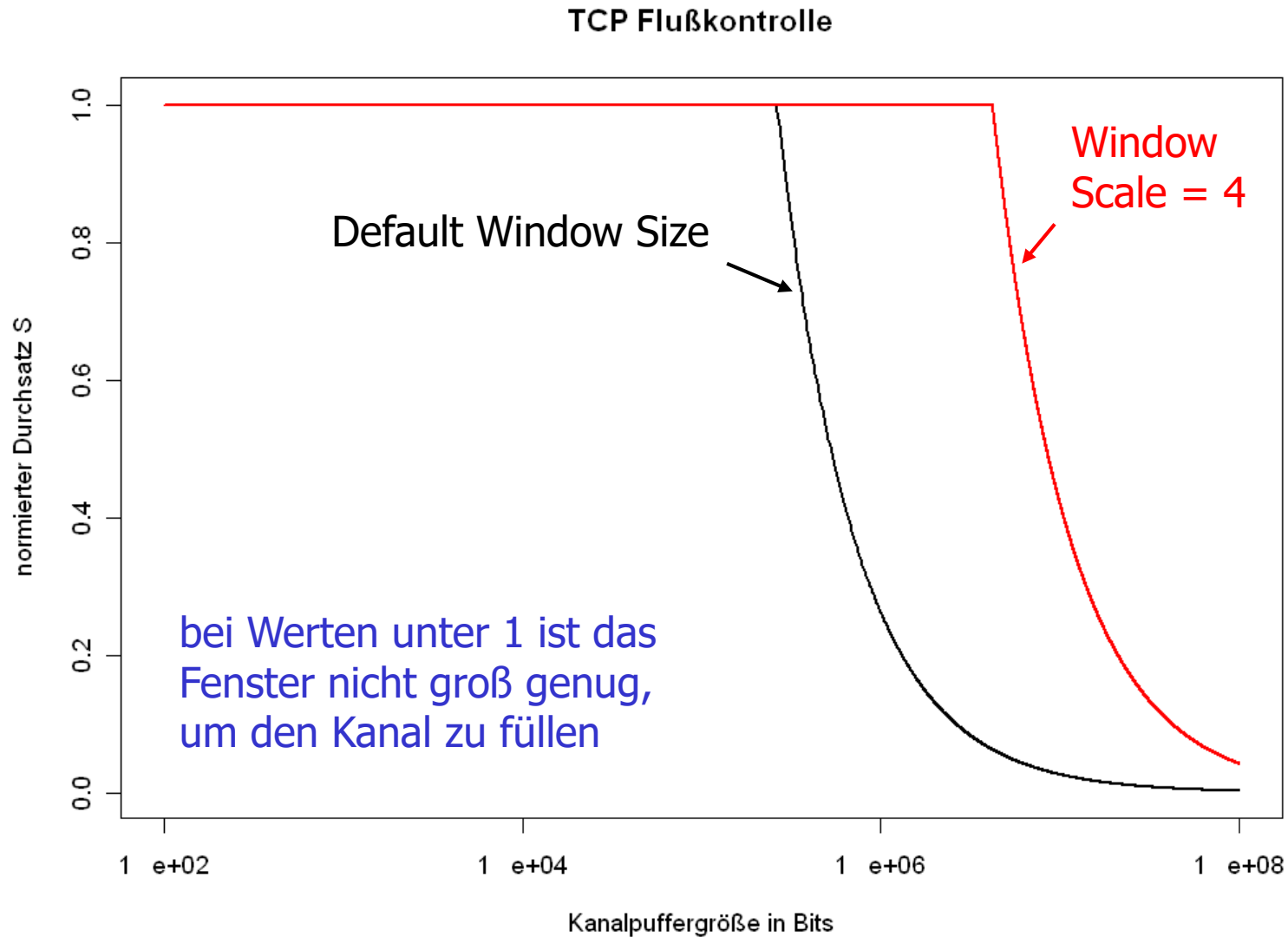
sprich wenn man ein  $W > 2^{16}$  brä

maximal also  $2^{16} \cdot 2^{14} = 2^{30}$



Nachteil mit Window-Scale-Faktor: Man kann nicht mehr so feingl

# TCP: Flusskontrolle



# TCP: Überlastkontrolle

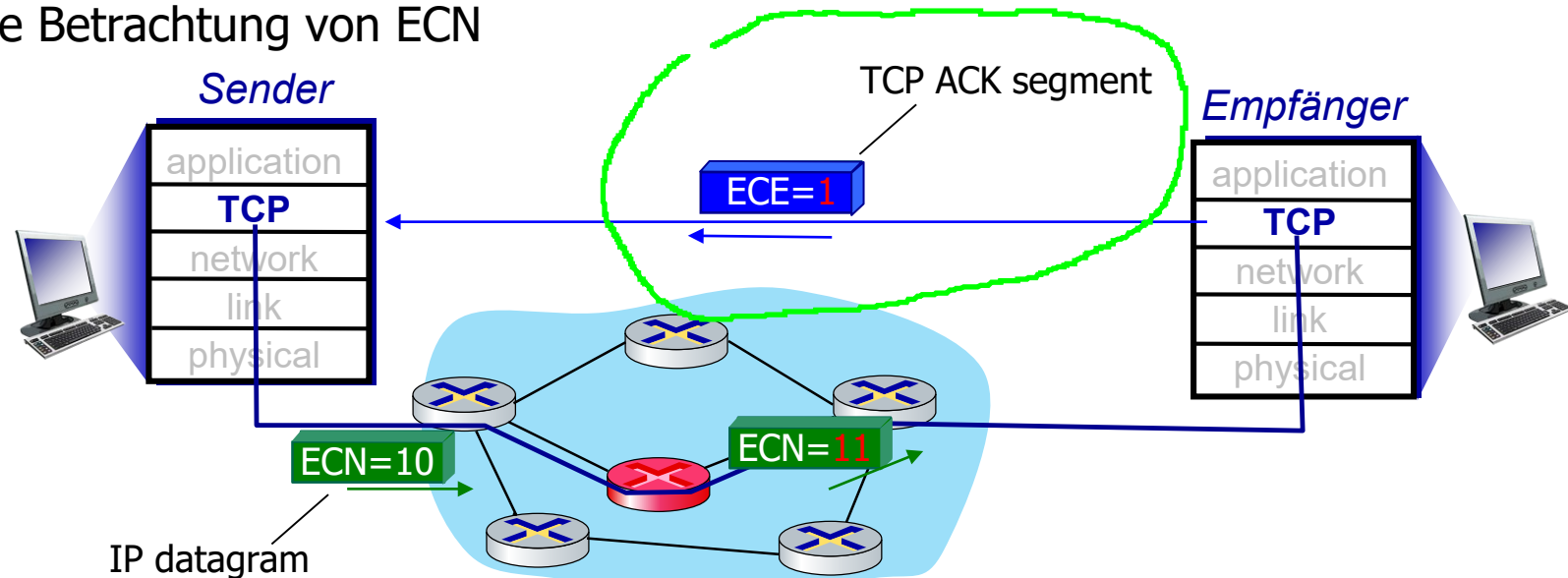
## ■ Überblick über die Überlastkontrolle

- der TCP-Sender versucht aus den zurückkommenden ACKs Informationen über die mögliche Senderate (Durchsatz) zu erhalten
- abhängig davon wird **Sendefenster vergrößert bzw. verkleinert** also volle Puffer ==> paketverlust ==> doppelte ACKs. Problem, nur imp
  - klassisch: Bei **vollen Puffern müssen Pakete verworfen werden, Paketverluste führen zu doppelten ACKs**, diese werden als Überlast interpretiert (**loss-based congestion control**)
    - Problem: Paketverluste wegen **schlechter Linkeigenschaften werden fälschlicherweise als Überlast interpretiert**
    - Beispiel: **TCP Reno (im Folgenden betrachtet), CUBIC** LINUX
  - alternativ: Gefüllte Puffer führen zu **steigenden Verzögerungen bzw. RTTs** (**delay-based congestion control**) also approximation über RTT die füllung der Puffer
    - Beispiel: TCP Vegas, BBR

# TCP: Überlastkontrolle

## ■ Zusätzlich: Überlastkontrolle durch Explicit Congestion Notification (ECN)

- IP-Header können 2 Bits für ECN enthalten erstes bit "aktiv" zweites bit "congestion detected"
- Wenn ECN=10 im IP-Header gesetzt und Überlast in Router, wird ECN=11 in IP-Header gesetzt
- Dies wird TCP auf Empfangsseite signalisiert, sendet ECE=1 im TCP-Header zurück
- TCP auf Sendeseite kann darauf wie auf Verlust reagieren und Sendefenster reduzieren und signalisiert dies mit CWR=1 im TCP-Header
- Zusammenspiel von IP und TCP benötigt
- im folgenden keine Betrachtung von ECN



Quelle: Kurose, Ross.  
*Computer Networking:  
A Top-Down Approach*,  
7th Ed., Pearson  
Education, 2020.

# TCP: Überlastkontrolle

## ■ Überblick über verlustbasierte Überlastkontrolle

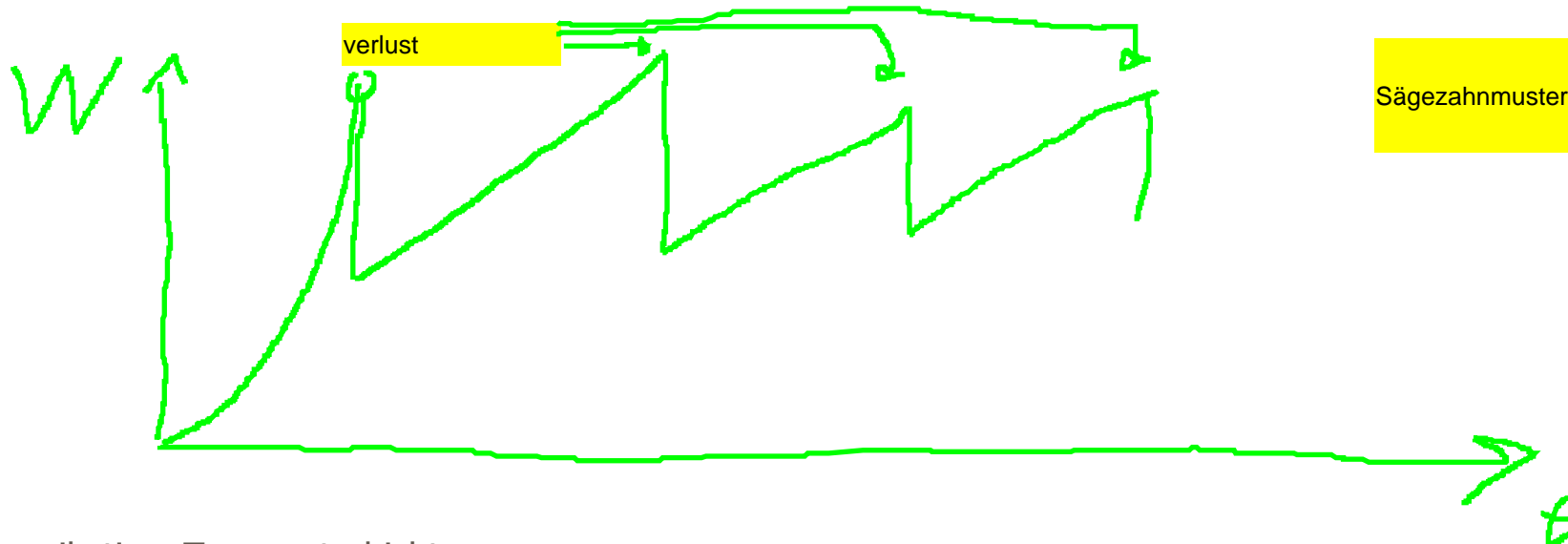
- hierzu gibt es das **CongestionWindow**, das zusammen mit der Flusskontrolle zur Ermittlung des tatsächlichen Sendefensters verwendet wird:
    - $\text{MaxWindow} = \text{Min}(\text{CongestionWindow}, \text{AdvertizedWindow})$
    - $\text{EffectiveWindow} = \text{MaxWindow} - (\text{LastByteSent} - \text{LastByteAcked})$

also statt dem Advertized window wird einfach das maxwin
  - der Durchsatz ergibt sich ungefähr aus  $\text{CongestionWindow} / \text{RTT}$
  - durch Vergrößerung des CongestionWindows vergrößert der Sender den Durchsatz und versucht sich an den möglichen Durchsatz anzunähern
  - bei einem Verlust (durch 3 doppelte ACKs oder einen Timeout zu erkennen) wird das CongestionWindow und damit der Durchsatz wieder verkleinert
- Fast retransmit oder normaler retransmit

# TCP: Überlastkontrolle

## ■ 3 Mechanismen

- **Slow Start:** am Anfang erhöht der Sender das CongestionWindow beginnend mit einer MSS exponentiell bis er durch 3 doppelte ACKs erfährt, dass ein Segment verlorengegangen ist
- danach erfolgt **AIMD (Additive Increase, Multiplicative Decrease)**: das CongestionWindow wird halbiert und dann linear bis zum nächsten Erhalt von 3 doppelten ACKs erhöht
- danach wieder AIMD ...
- **konservative Reaktion nach Timeout:** dann wird Slow Start bis zur Hälfte des aktuellen CongestionWindows und danach AIMD durchgeführt



Sägezahnmuster Charakteristisch für TCP



# TCP: Überlastkontrolle

## ■ Mehr Details:

### ● Slow Start

- setze  $\text{CongestionWindow} = \text{MSS}$
- nach Erhalt eines ACKs:  $\text{CongestionWindow} += \text{MSS}$  (hierdurch wird ein exponentielles Ansteigen realisiert)
- bis  $\text{Threshold}$  erreicht ist, dann Additive Increase (am Anfang ist  $\text{Threshold}$  unendlich)

Ich glaube das soll  $\text{CongestionWindow} += \text{CongestionWindow}$  sein...

### ● nach 3 doppelten ACKs:

- Multiplicative Decrease:  $\text{Threshold} = \text{CongestionWindow}/2$ ;  $\text{CongestionWindow} /= 2$   
also das maximum halbiert
- Additive Increase: bei Erhalt eines ACKs  
 $\text{CongestionWindow} += \text{MSS} \times (\text{MSS}/\text{CongestionWindow})$ 
  - hierdurch wird ein Wachstum um ca. ein MSS pro RTT realisiert
  - z.B.:  $\text{MSS} = 1.460$  Bytes,  $\text{CongestionWindow} = 14.600$  Bytes, jedes ACK vergrößert um ca.  $1/10$  MSS, 10 ACKs um ca. 1 MSS

### ● nach Timeout

- $\text{Threshold} = \text{CongestionWindow}/2$ ;  $\text{CongestionWindow} = \text{MSS}$

wie 3 ACKs nur mir zurücksetzen des Congestion windows auf minimum und dann Additive increase

# TCP: Überlastkontrolle

## ■ Beispiel für Slow Start

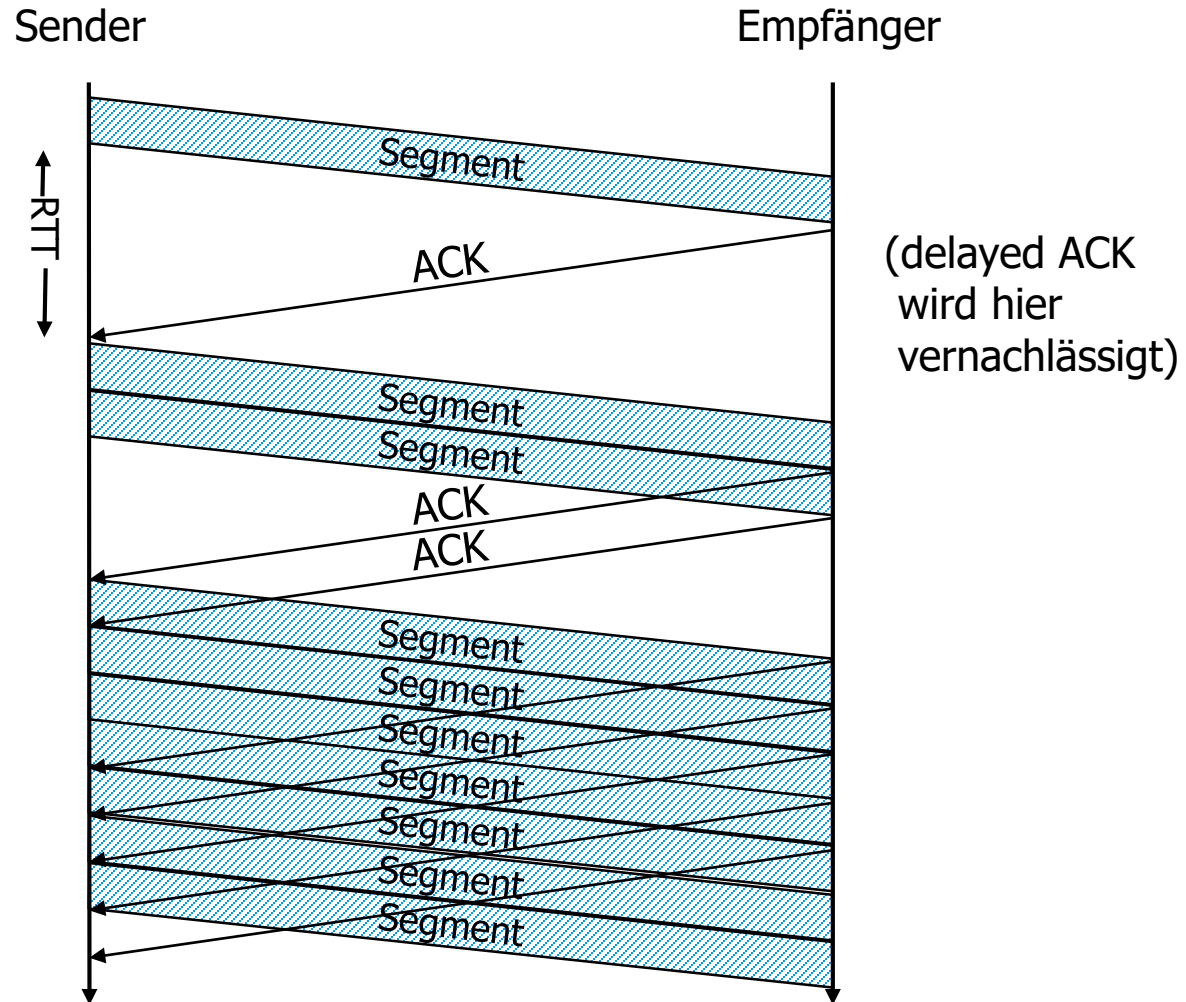
CongestionWindow = 1 MSS

CongestionWindow = 2 MSS

CongestionWindow = 4 MSS

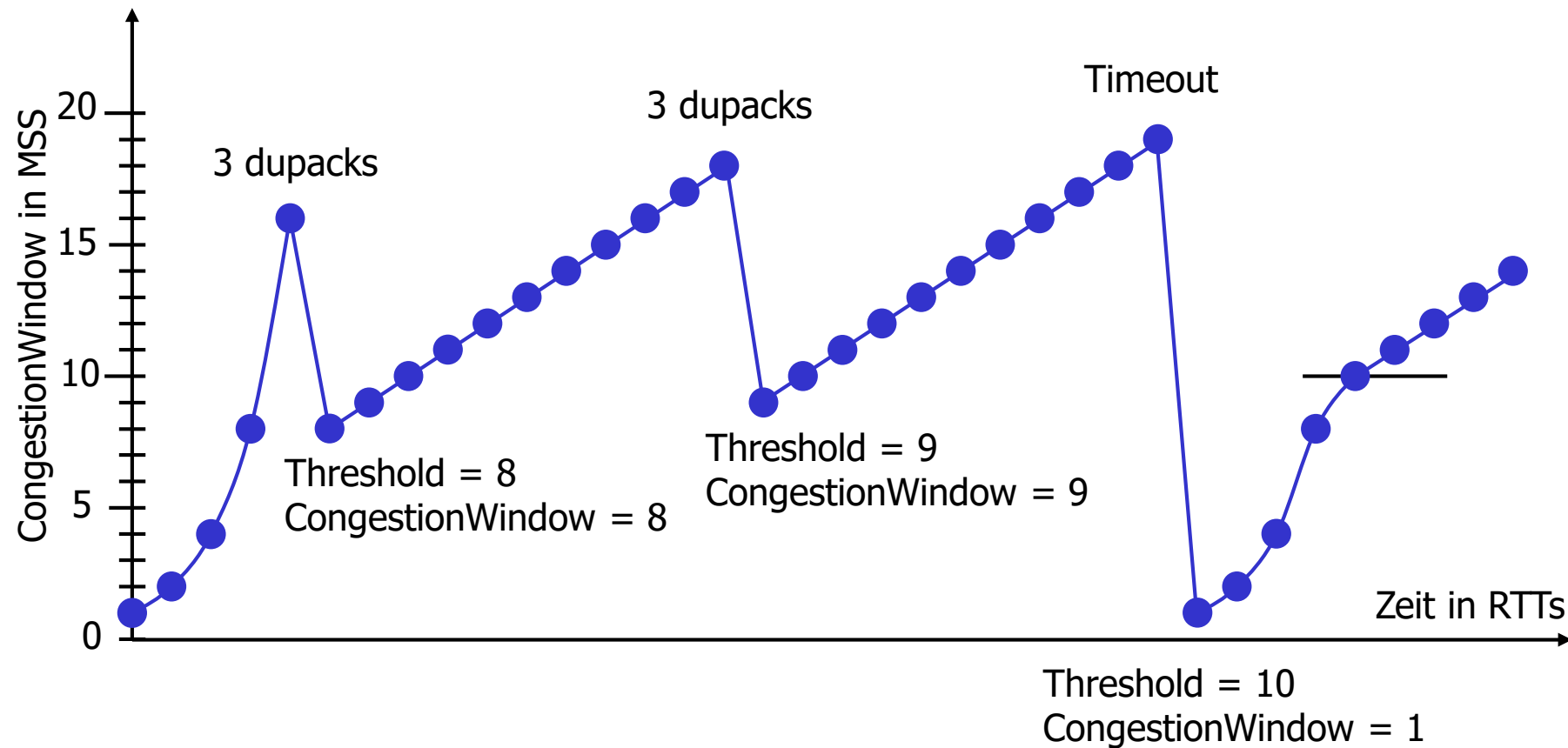
(ab hier kann ununterbrochen  
gesendet werden)

CongestionWindow = 8 MSS



# TCP: Überlastkontrolle

## ■ Beispiel für zeitlichen Ablauf:



# TCP: Überlastkontrolle

## ■ ungefähre Durchschnitt

- Annahme: nur AIMD, Vernachlässigung von Slow-Start am Anfang und nach Timeouts
- CongestionWindow pendelt ungefähr zwischen dem maximalen Wert  $W$  und der Hälfte  $W/2$
- die Bitrate also zwischen  $W/RTT$  und  $\frac{1}{2} W/RTT$
- im Mittel ergibt sich  $\frac{3}{4} W/RTT$

passiert zu selten

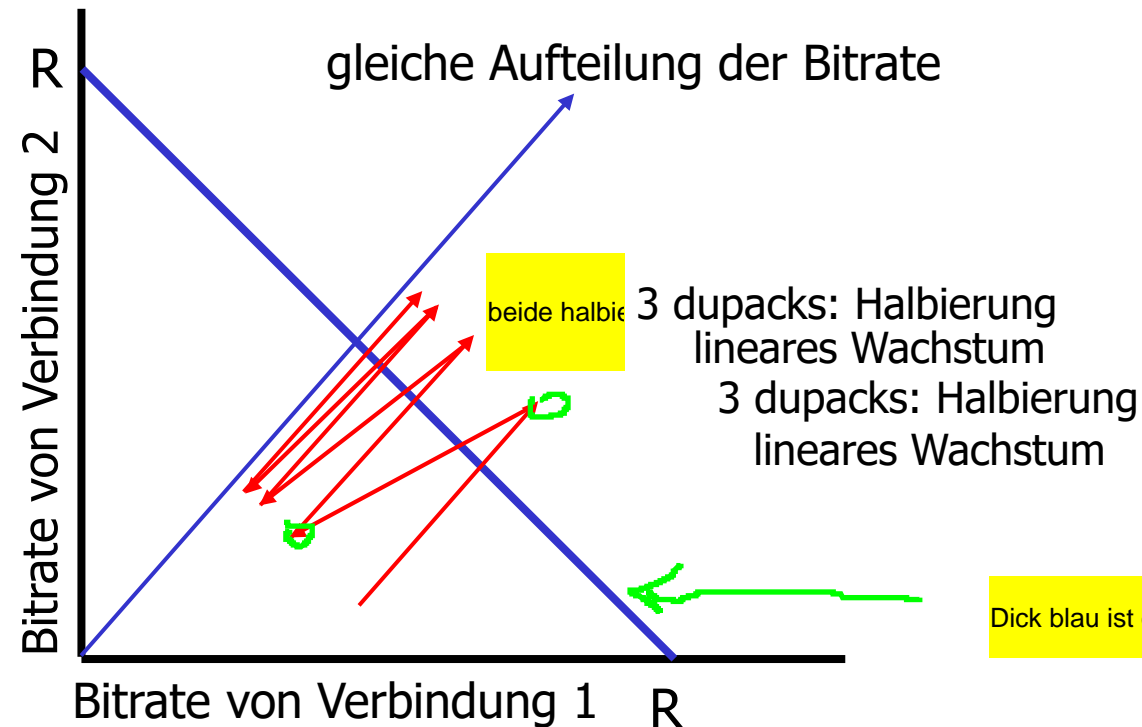


so funktioniert mathe nicht, aber okay...

# TCP: Fairness der Überlastkontrolle

## ■ Szenario

- 2 TCP-Verbindungen teilen sich die Bitrate  $R$  eines Kanals
- Fairness: jede Verbindung sollte  $R/2$  erhalten
- Vernachlässigung des Slow Starts



Dick blau ist das maximum  $R$  die einer von beiden alleine maxim.

# TCP: Beispiel für Zeit-Sequenz-Diagramm

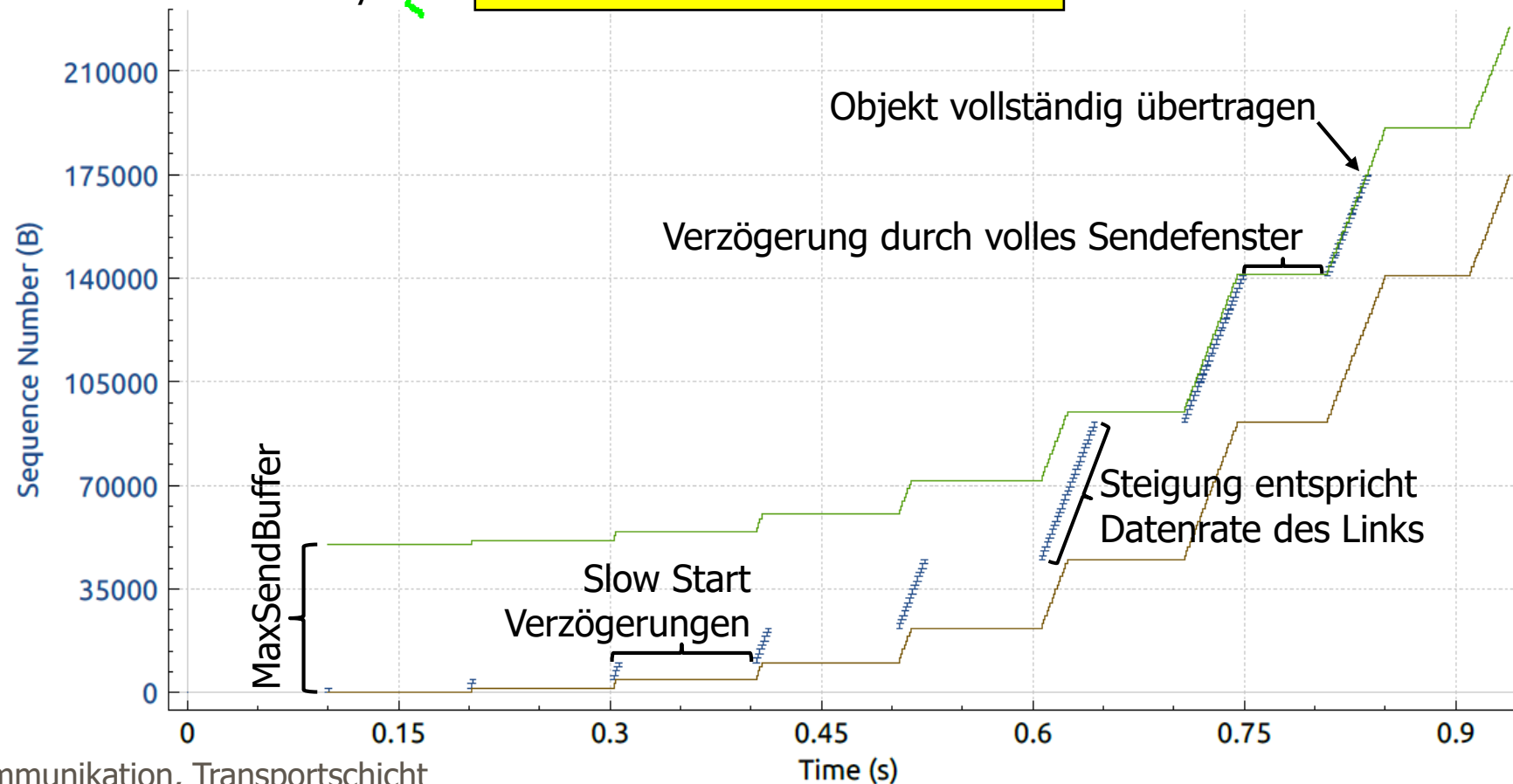
## ■ Netzwerksimulator ns-3, Grafik ([tcptrace](#)) erstellt mit Wireshark

Link mit 10 Mbit/s und 50 ms Verzögerung

Objektgröße 175 kByte

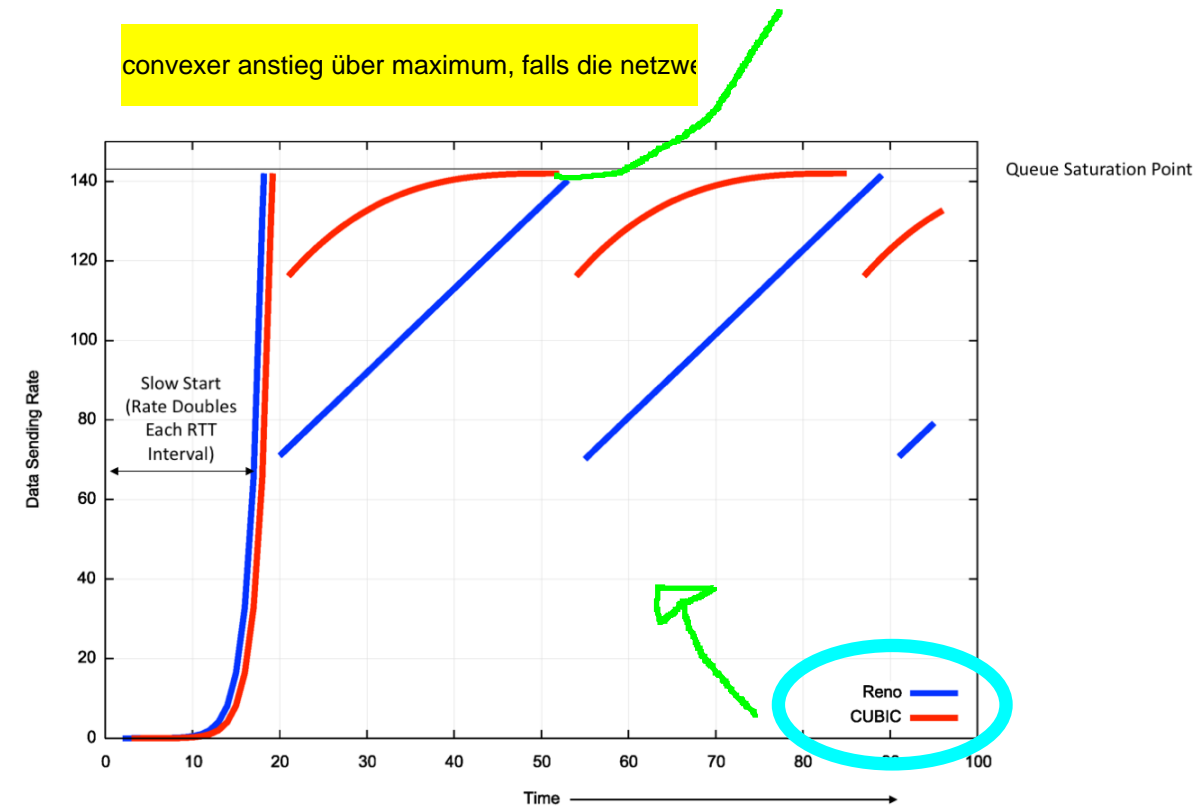
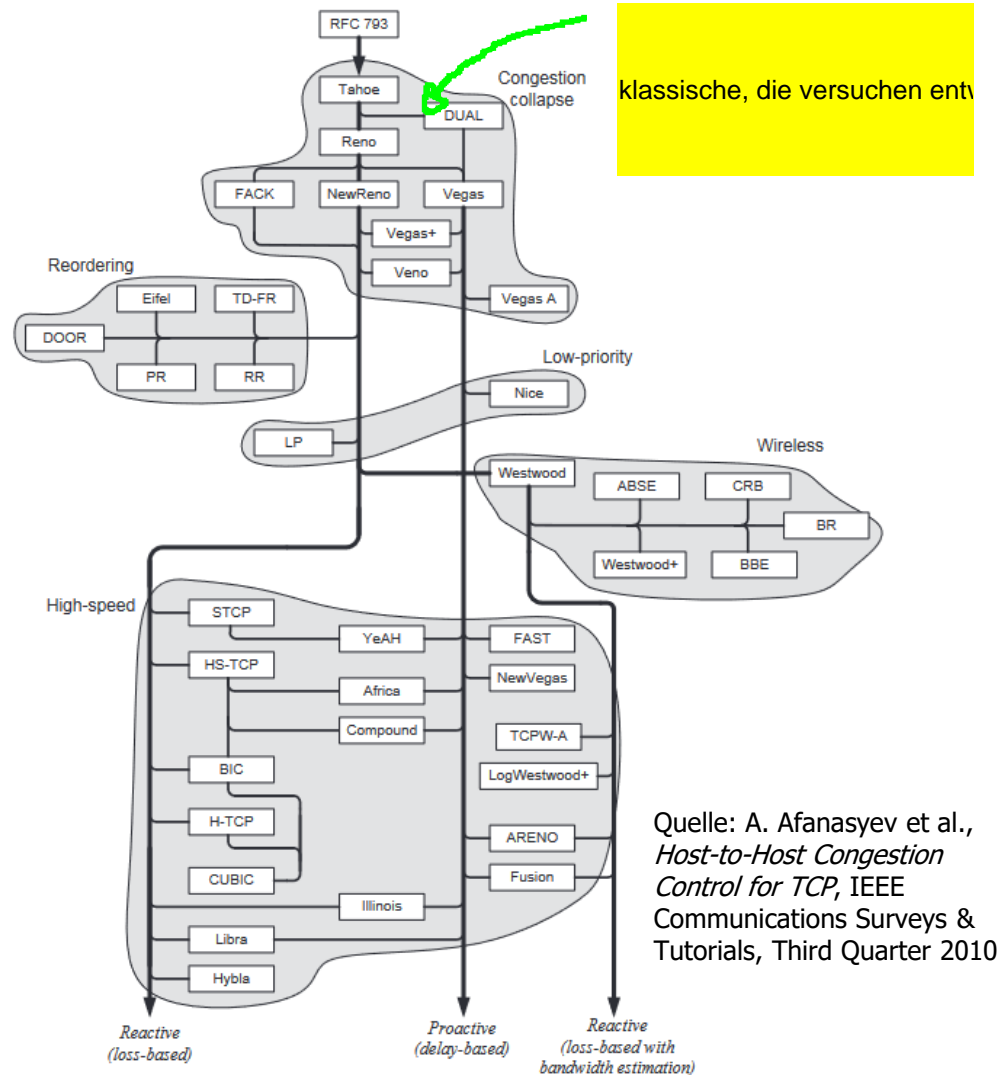
MTU 1500 Byte

Maximum Transfer Size = Maximum segment size+head



# Ausblick: Andere Überlastalgorithmen

## ■ Vielzahl alternativer Überlastalgorithmen, heute weit verbreitet: TCP CUBIC



CUBIC benutzt exponentiellen slow-start und nähert sich dem Maximum an. *new kid on the TCP block*, 05/2017, <https://blog.apnic.net/2017/05/09/bbr-new-kid-tcp-block/>

# Transportschicht

- Einführung
- UDP
- Fehlerkontrolle
- TCP
  - Segmentformat
  - Fehlerkontrolle
  - Verbindungsauf- und -abbau
  - Schätzung der RTT
  - Fluss- und Überlastkontrolle
  - Leistungsanalyse
  - Multipath TCP
- TLS
- QUIC



# TCP: Leistungsanalyse

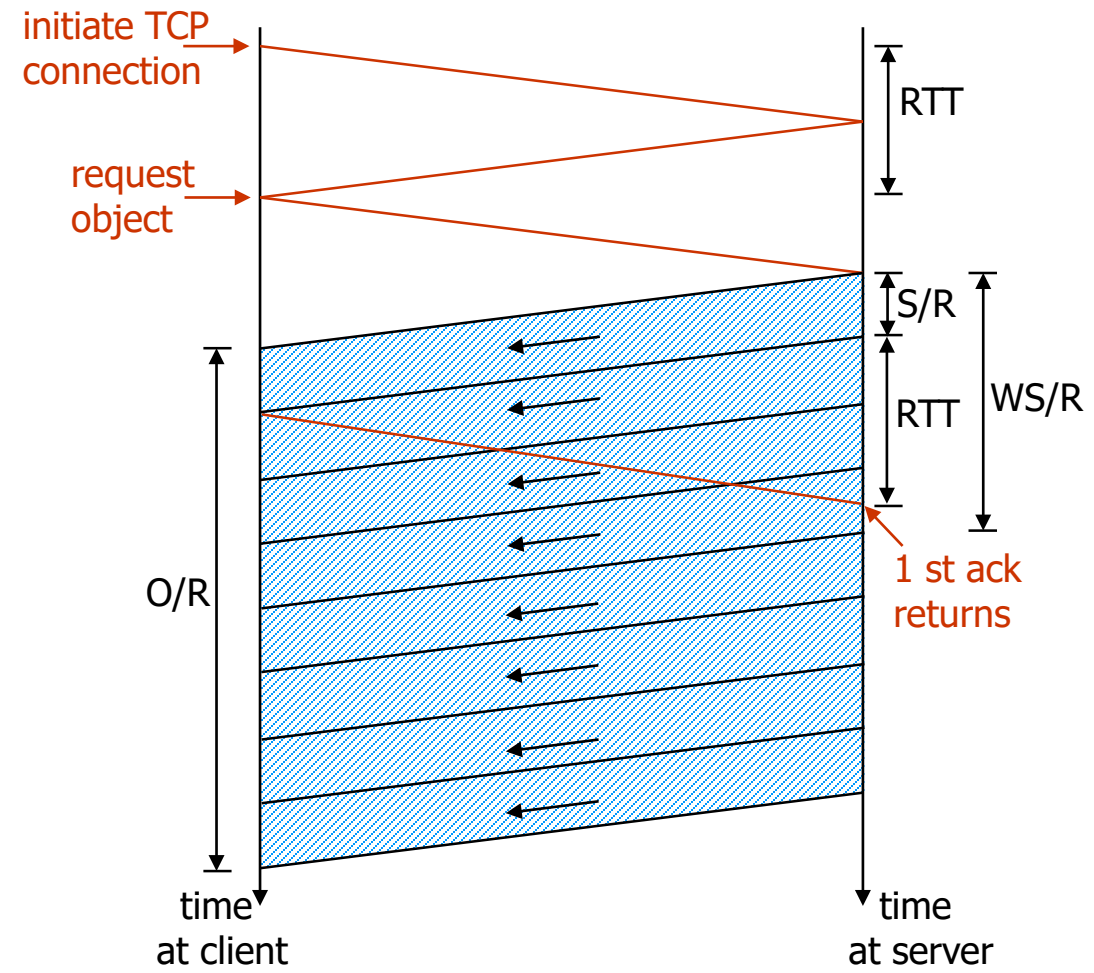
## ■ Zeit zum Kopieren eines Objektes mit TCP

- hängt ab von Objektgröße, Bitrate, Ausbreitungsverzögerung und Verzögerungen durch Protokollmechanismen
- insbesondere Slow-Start kann sich spürbar auswirken
- Annahmen
  - keine Bitfehler und Verluste, keine schwankenden Bitraten und Verzögerungen, ACKs benötigen keine Sendezeit, keine Bearbeitungszeiten, Fenster der Flusskontrolle immer groß genug
- Notation
  - S: MSS in Bits
  - O: Objektgröße in Bits
  - R: Bitrate
  - RTT: round trip time
  - W: Fenstergröße in MSSs
- Analyse zunächst für feste Fenstergröße, dann für wachsendes Fenster wie bei Slow Start

# TCP: Leistungsanalyse, festes Fenster

## ■ Festes Fenster, 1. Fall

- Fenster füllt den Kanal:  
 $WS/R > RTT + S/R$
- Verzögerung =  $2RTT + O/R$



# TCP: Leistungsanalyse, festes Fenster

## ■ Festes Fenster, 2. Fall

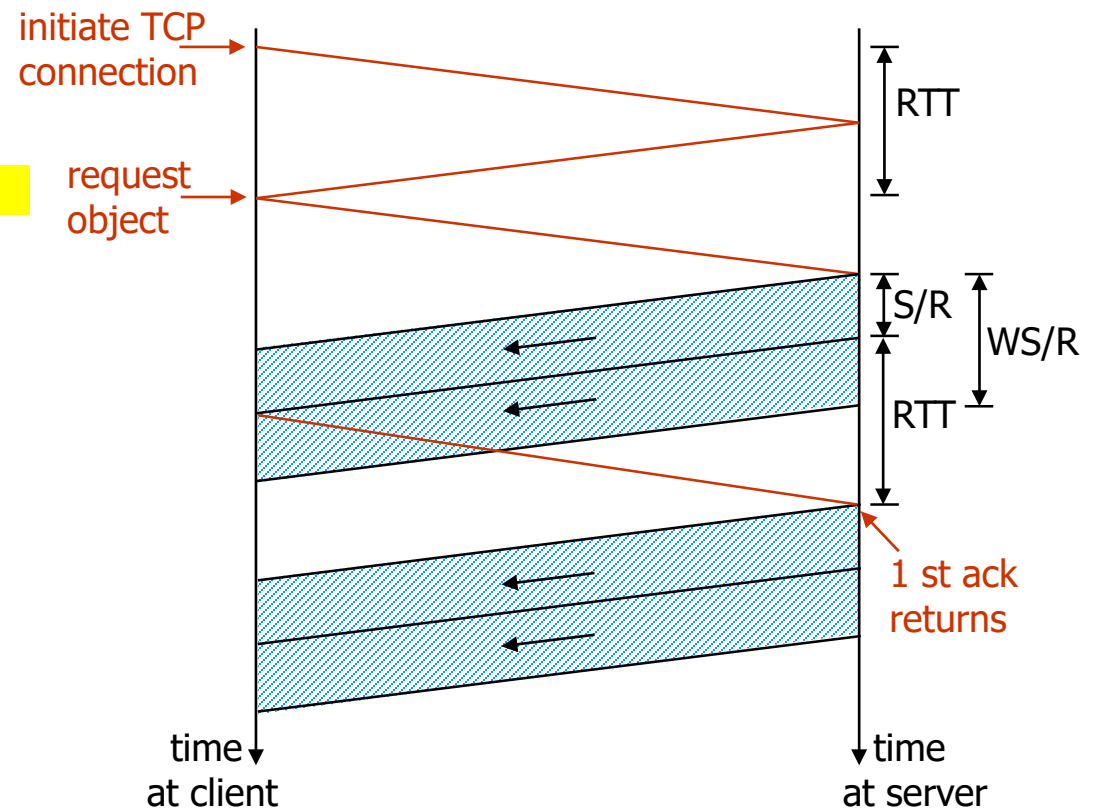
- Fenster ist nicht groß genug um den Kanal zu füllen:

$$WS/R < RTT + S/R$$

- Verzögerung =  $2RTT + O/R + (K-1)[S/R + RTT - WS/R]$   
sollte das fenster groß genug sein, fällt RTT weg

- K gibt die Anzahl von Fenstern an, die das Objekt benötigt:

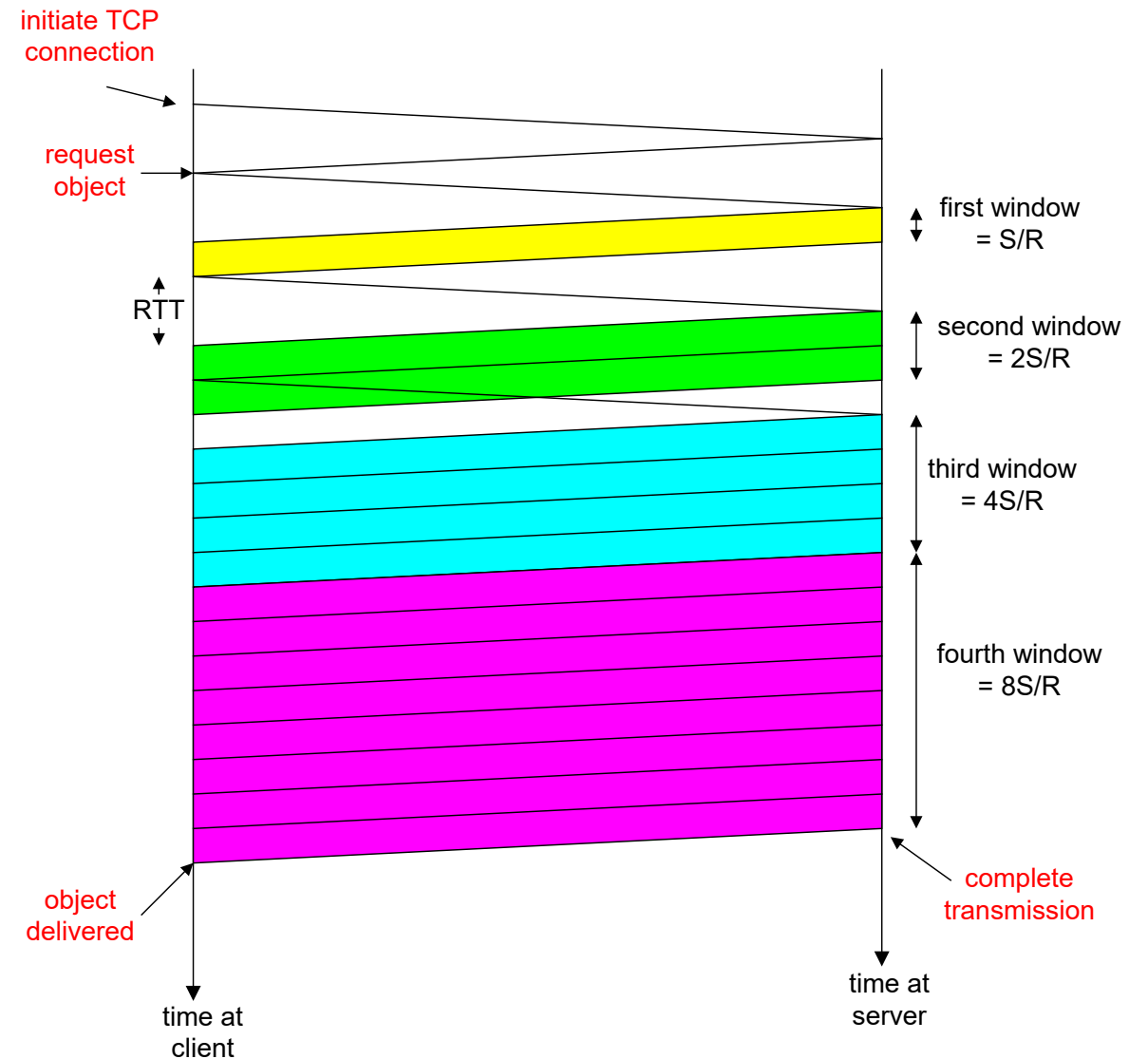
$$K = \left\lceil \frac{O}{WS} \right\rceil$$



# TCP: Leistungsanalyse, Fenster wächst wie bei Slow Start

## ■ wachsendes Fenster wie bei Slow Start:

- 2RTT für den Verbindungsaufbau
- O/R für das Senden
- O/S Segmente
- K Fenster
- **P Slow-Start-Wartezeiten**
- hier:
  - O/S = 15
  - K = 4
  - P = 2



# TCP: Leistungsanalyse, Fenster wächst wie bei Slow Start

## ■ Verzögerungszeit

- Verzögerung =  $2 \text{ RTT} + O/R + \text{Slow-Start-Wartezeiten}$
- Slow-Start-Wartezeit im k-ten Fenster
  - $2^{k-1} S/R = \text{Sendezeit im k-ten Fenster}$
  - $S/R + \text{RTT} = \text{Zeit vom Sendebeginn bis zum Erhalt des ACKs}$
  - $\max[S/R + \text{RTT} - 2^{k-1} S/R, 0] = \text{Wartezeit im k-ten Fenster}$
- mit  $P = \text{Anzahl von Slow-Start-Wartezeiten:}$

$$\begin{aligned}\text{Verzögerung} &= \frac{O}{R} + 2\text{RTT} + \sum_{k=1}^P \text{Wartezeit}_k \\ &= \frac{O}{R} + 2\text{RTT} + \sum_{k=1}^P \left[ \frac{S}{R} + \text{RTT} - 2^{k-1} \frac{S}{R} \right] \\ &= \frac{O}{R} + 2\text{RTT} + P \left[ \text{RTT} + \frac{S}{R} \right] - (2^P - 1) \frac{S}{R}\end{aligned}$$

also P mal slow-s

sum  $2^{k-1} = (2^P - 1)$

## TCP: Leistungsanalyse, Fenster wächst wie bei Slow Start

### ■ Berechnung der Anzahl von Slow-Start-Wartezeiten

- K = Anzahl der Fenster, die für das Objekt benötigt werden

$$= \min\{k : 2^0 S + 2^1 S + \dots + 2^{k-1} S \geq O\} = \min\{k : 2^0 + 2^1 + \dots + 2^{k-1} \geq O/S\}$$

$$= \min\{k : 2^k - 1 \geq \frac{O}{S}\} = \min\{k : k \geq \log_2(\frac{O}{S} + 1)\} = \left\lceil \log_2(\frac{O}{S} + 1) \right\rceil$$

worst-case wir haben nur slow-starts

- Q = Anzahl von Slow-Start-Wartezeiten bei unendlich großem Objekt

$$= \max_k \left( \frac{S}{R} + RTT - 2^{k-1} \frac{S}{R} \geq 0 \right) = \max_k \left( 2^{k-1} \leq 1 + \frac{RTT}{S/R} \right) = \left\lceil \log_2 \left( 1 + \frac{RTT}{S/R} \right) \right\rceil + 1$$


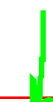

wir konvergieren zu Q slow starts im limes

- dann P = min(Q, K-1) Wartezeiten

## TCP: Leistungsanalyse, Fenster wächst wie bei Slow Start

### ■ Endergebnis:

anzahl der slow-start wartezeiten:  $P = m_i$


$$\text{Verzögerung} = 2RTT + \frac{O}{R} + P \left[ RTT + \frac{S}{R} \right] + (2^P - 1) \frac{S}{R}$$

- enthält Produkt von  $P$  und  $RTT$ , also: wenn  $RTT$  groß und/oder viele Slow-Start-Wartezeiten auftreten, kann die Verzögerung spürbar werden

## TCP: Leistungsanalyse, Beispiele

- Szenario I:  $S=536$  Bytes,  $RTT=100$  ms (intern.),  $O=100$  KB (lang)

$R$	$O/R$	<i>Verzögerung</i>
28 Kbps	28.6 s	28.9 s
100 Kbps	8 s	8.4 s
1 Mbps	800 ms	1.5 s
10 Mbps	80 ms	0.98 s

- Szenario II:  $S=536$  Bytes,  $RTT=100$  ms,  $O=5$  KB (kurz)

$R$	$O/R$	<i>Verzögerung</i>
28 Kbps	1.43 s	1.73 s
100 Kbps	0.48 s	0.757 s
1 Mbps	40 ms	0.52 s
10 Mbps	4 ms	0.50 s

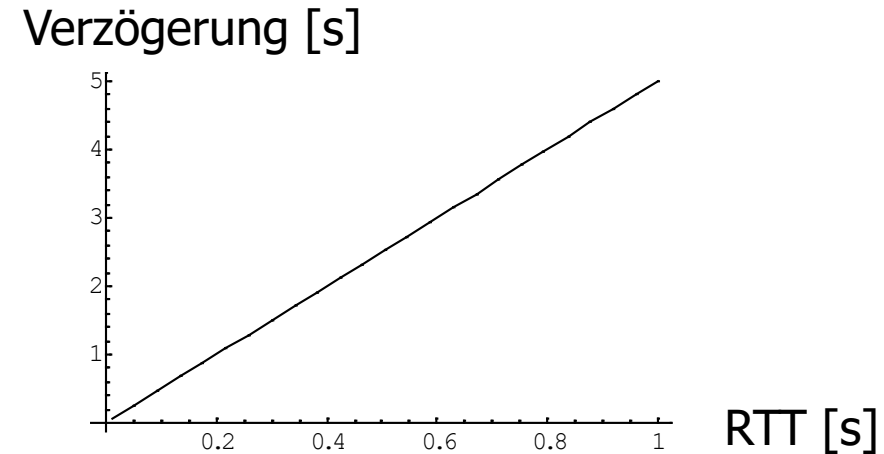
- Szenario III:  $S=536$  Bytes,  $RTT=1$  s (Überlast),  $O=5$  KB

$R$	$O/R$	<i>Verzögerung</i>
28 Kbps	1.43 s	5.8 s
100 Kbps	0.48 s	5.2 s
1 Mbps	40 ms	5.0 s
10 Mbps	4 ms	5.0 s

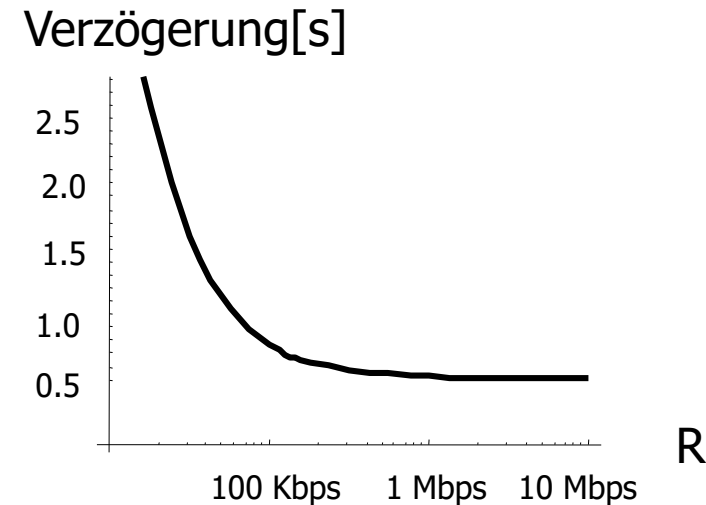


## TCP: Leistungsanalyse, Beispiele

- $S=536$  Bytes,  $O=5$  KB,  
 $R=1$  Mbps, RTT wird verändert:



- $S=536$  Bytes,  $O=5$  KB,  
RTT=100 ms,  $R$  wird verändert:  
von 10 Kbps bis 10 Mbps  
(logarithmische Skalierung):



# TCP: Leistungsanalyse, Antwortzeiten bei Web-Seiten

## ■ Web-Seite mit:

- 1 Basis-HTML-Seite ( $O$  Bits)
- $M$  Bilder (auch jeweils  $O$  Bits)

## ■ nicht-persistentes HTTP:

- $M+1$  sequentielle TCP-Verbindungen
- Antwortzeit =  $(M+1)O/R + (M+1)2RTT + \text{Slow-Start-Wartezeiten}$

## ■ persistentes HTTP:

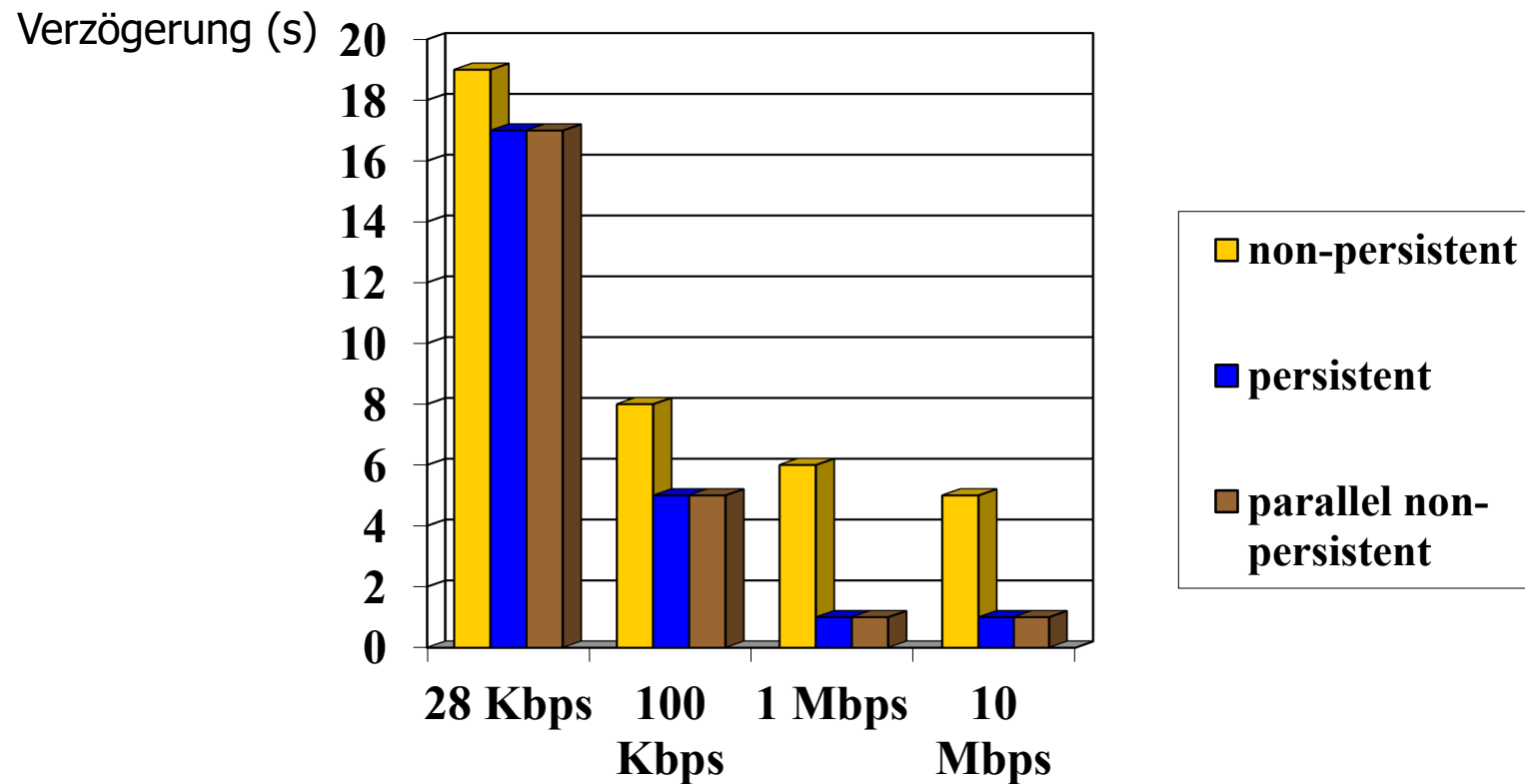
- 2 RTT für Basis-Seite
- 1 RTT für  $M$  Bilder
- Antwortzeit =  $(M+1)O/R + 3RTT + \text{Slow-Start-Wartezeiten}$

## ■ nicht-persistentes HTTP mit $X$ parallelen Verbindungen

- Annahme:  $M/X$  ist ganze Zahl
- 1 TCP-Verbindung für Basis-Seite
- $M/X$  Mengen von parallelen Verbindungen für Bilder
- Antwortzeit =  $(M+1)O/R + (M/X+1)2RTT + \text{Slow-Start-Wartezeiten}$

## TCP: Leistungsanalyse

RTT = 100 ms, O = 5 Kbytes, M=10, X=5

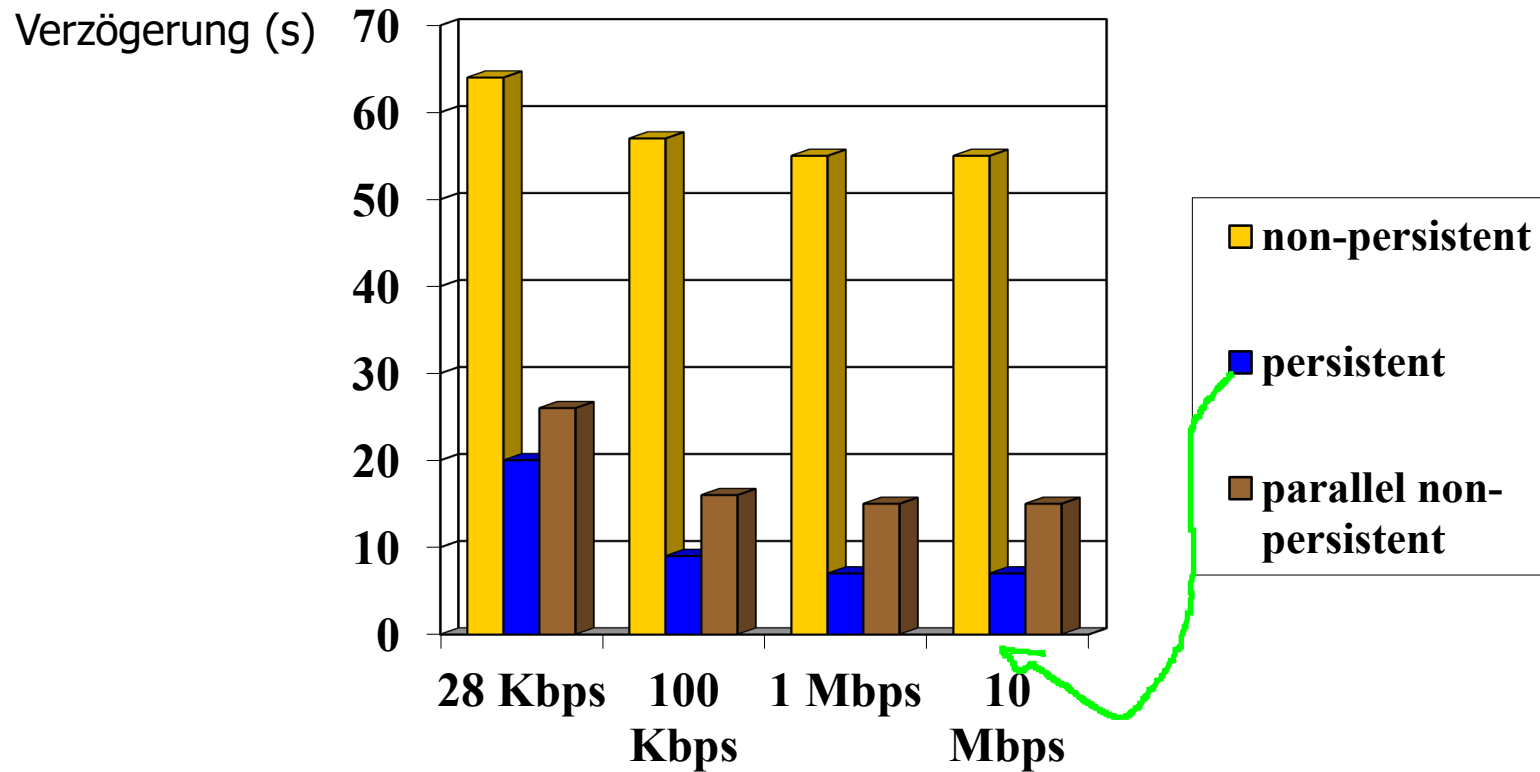


Für geringe Bitraten wird die Antwortzeit durch die Übertragungszeit dominiert; persistente Verbindungen ergeben nur geringen Vorteil gegenüber parallelen Verbindungen.

# TCP: Leistungsanalyse

anzahl paralleler verbindungen

RTT = 1 s, O = 5 Kbytes, M=10, X=5



Für große RTTs wird die Antwortzeit durch Slow-Start-Wartezeiten dominiert, persistente Verbindungen ergeben insbesondere für große Bitraten-Verzögerungs-Produkte Vorteile.

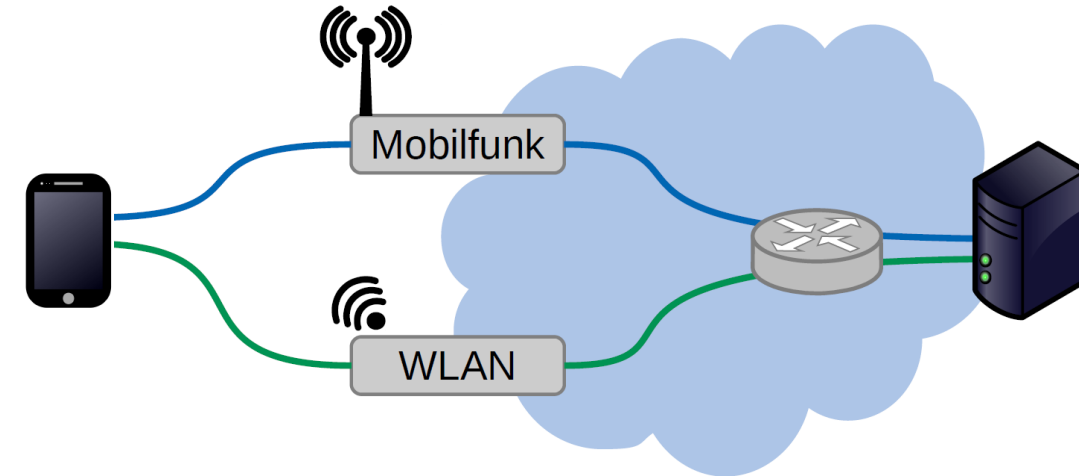
# Transportschicht

- Einführung
- UDP
- Fehlerkontrolle
- TCP
  - Segmentformat
  - Fehlerkontrolle
  - Verbindungsauf- und -abbau
  - Schätzung der RTT
  - Fluss- und Überlastkontrolle
  - Leistungsanalyse
  - Multipath TCP
- TLS
- QUIC

# Multipath TCP

## ■ Multipath TCP

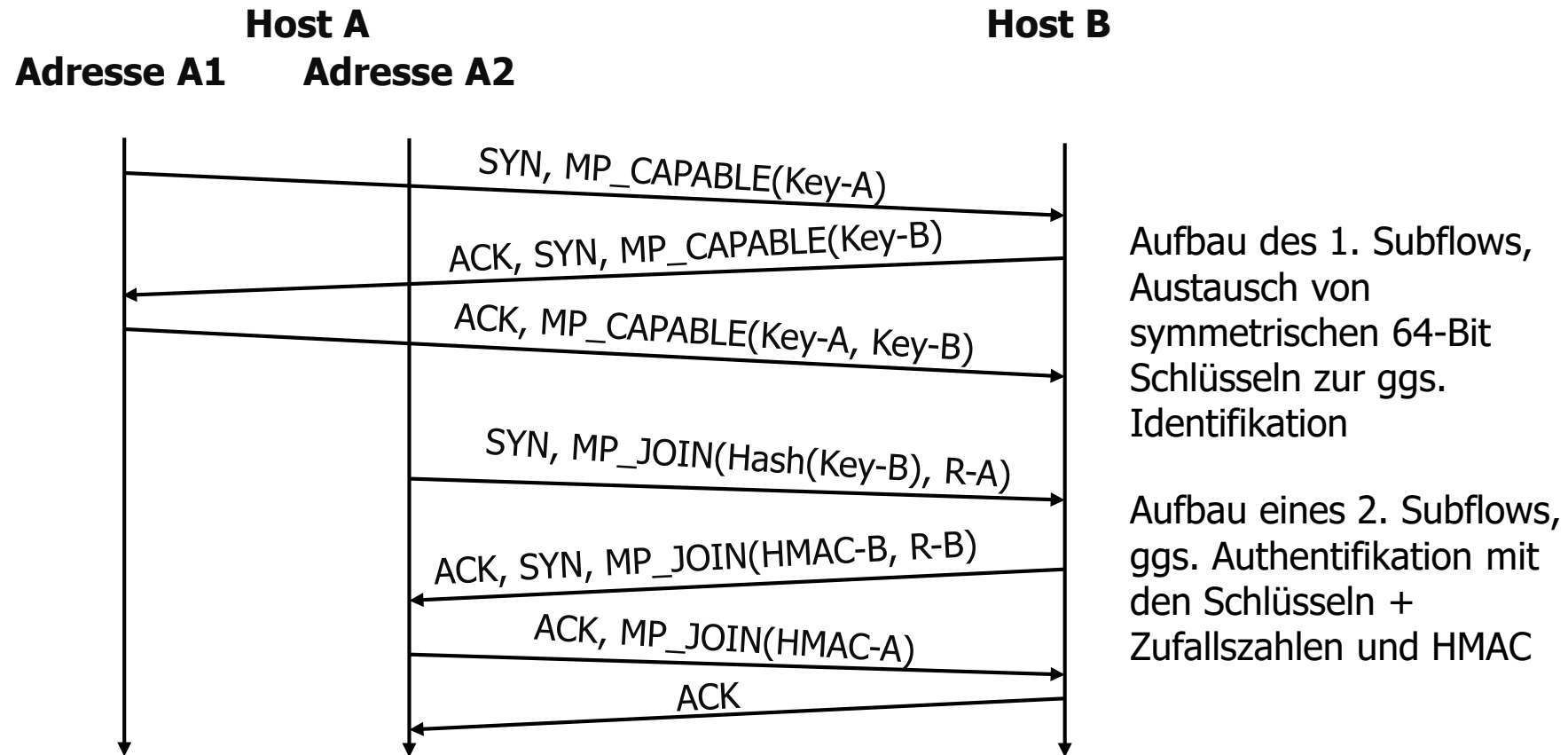
- Erweiterung von TCP, um gleichzeitig mehrere Pfade für eine Verbindung nutzen zu können
  - Beispiel: Kombination von Mobilfunk und WLAN
- experimenteller RFC 6824, RFC 6894 für API, u.a.
- Ziele: Erhöhung des Durchsatzes, bessere Ressourcenauslastung, bessere Resilienz (Widerstandsfähigkeit bei Fehlern), Abwärtskompatibilität
- Ansatz:
  - MPTCP als Zwischenschicht in Transportschicht
  - transparent, TCP Subflows erscheinen wie normales TCP
  - optional API zur Nutzung der Funktionen von MPTCP
  - Nutzung der Optionsfelder von TCP
    - Aufbau eines initialen Subflows: MP\_CAPABLE
    - Aufbau weiterer Subflows: MP\_JOIN
    - übergeordnete 64-Bit Data Sequence Number



Anwendung	
MPTCP	
Subflow (TCP)	Subflow (TCP)
IP	IP

# Multipath TCP

## ■ Beispielhafter Aufbau von 2 Subflows:



# Transportschicht

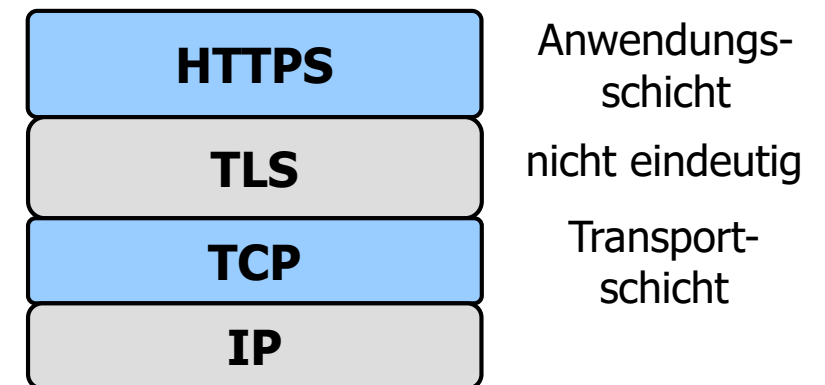
- Einführung
- UDP
- Fehlerkontrolle
- TCP
- TLS
- QUIC



# Transport Layer Security (TLS)

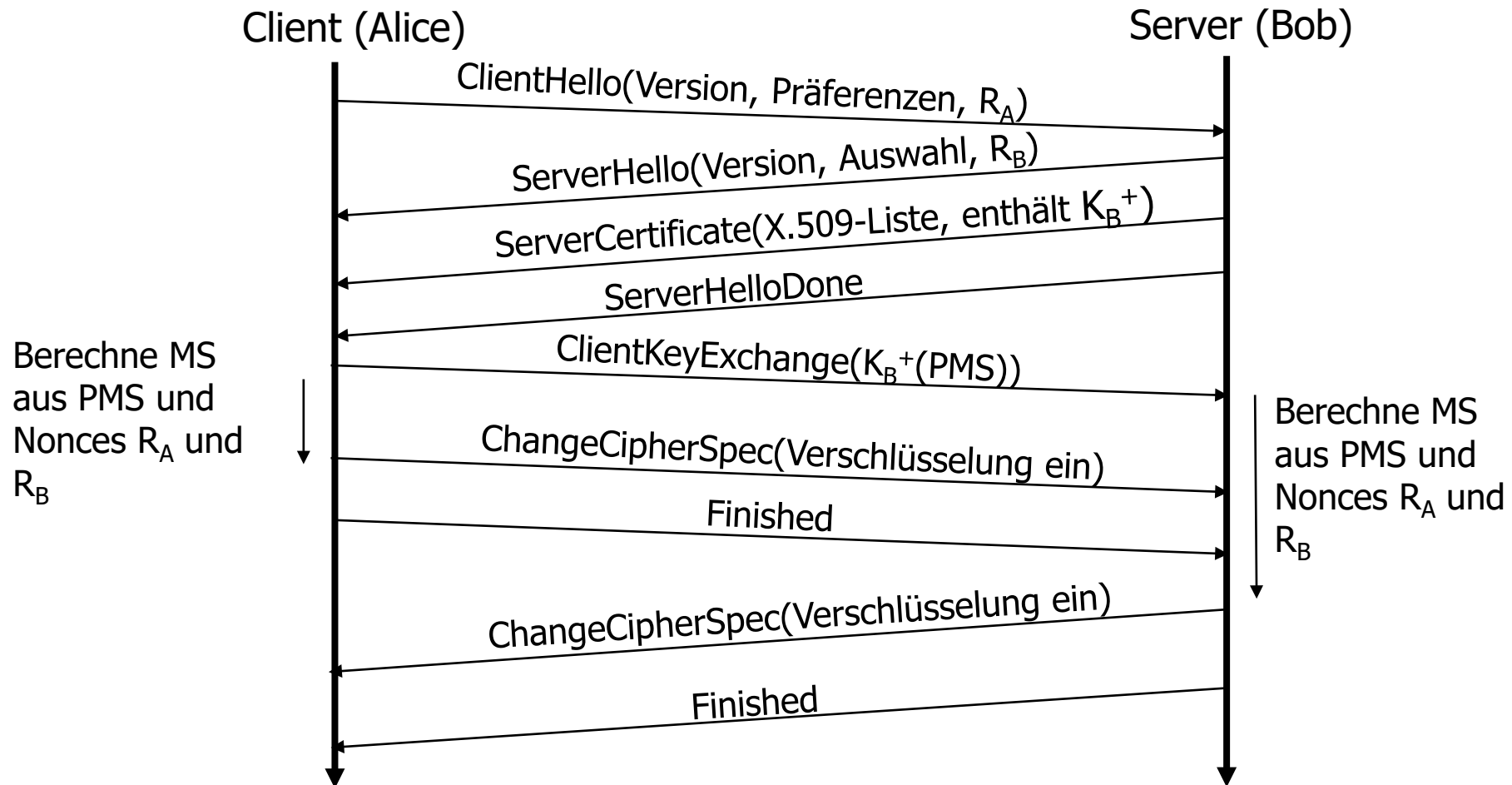
## ■ TLS 1.2 (RFC 5246)

- ursprünglich Secure Socket Layer (SSL), Netscape
- Sicherung der Transportschicht, allgemeine API für Anwendungen, Nutzung insbesondere durch HTTP (HTTPS), auch Version für UDP
- auf TCP-Verbindungs Aufbau folgt
  - **Handshake:**  
Aushandlung von Algorithmen (Verschlüsselung, MAC), Austausch von Nonces, Zertifikaten, Premaster Secret (PMS), Ableitung von Master Secret (MS) mit 2 symmetrischen Sitzungsschlüsseln, 2 MAC-Schlüsseln, 2 Initialisierungsvektoren für CBC
  - **Record-Protokoll (Datenaustausch):**  
Fragmentierung, Kompression, Hinzufügen von MAC, Verschlüsselung, Hinzufügen eines Headers
  - **Verbindung schließen**  
über spezielles Header-Feld (authentifiziert durch MAC)



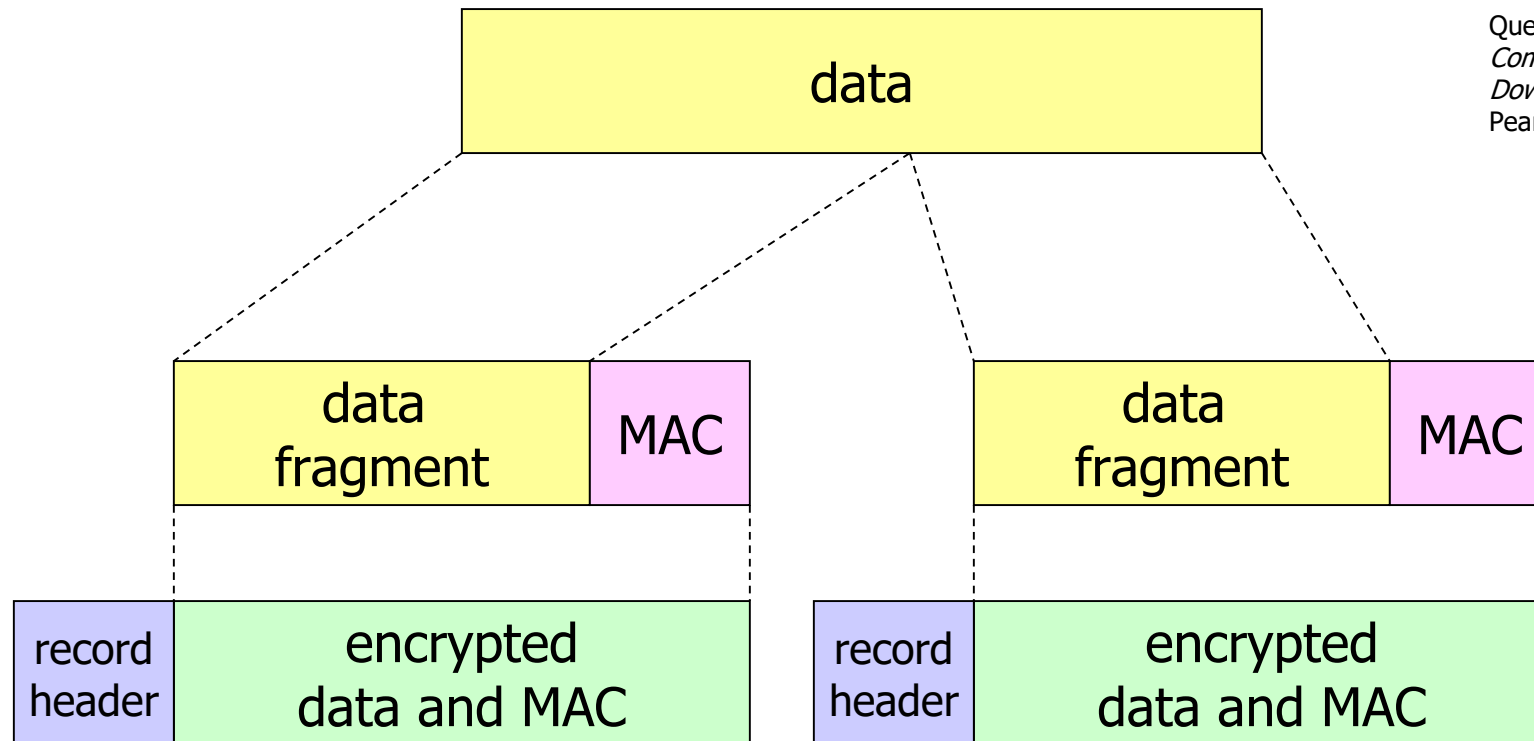
# Transport Layer Security (TLS)

## ■ Handshake (Bsp.):



# Transport Layer Security (TLS)

## ■ Record Protocol:



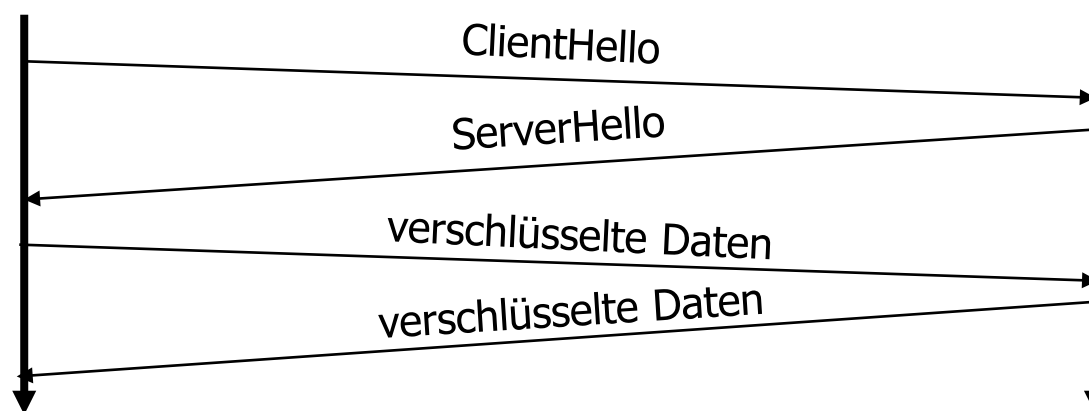
Quelle: Kurose, Ross.  
*Computer Networking: A Top-Down Approach*, 7th Ed.,  
Pearson Education, 2017.

- Record Header: Type, Version, Length
- MAC: Sequenznummer, Schlüssel

# Transport Layer Security (TLS)

## ■ TLS 1.3 (RFC 8446)

- Aktualisierung der kryptografischen Algorithmen
- kürzere Handshakes:
  - nur **Zwei-Wege Handshake** zum Austausch aller benötigten Informationen (Algorithmen, Parameter, Schlüssel, Zertifikate, Nonces, ...), danach bereits verschlüsseltes Senden von Daten
  - basiert auf Diffie-Hellman-Schlüsselaustausch und/oder vorher ausgetauschtem Schlüssel (**Pre-Shared Key, PSK**)
  - **0-RTT Mode**: ohne Handshake bei Wiederaufnahme (Resumption), Daten können sofort verschlüsselt gesendet werden



# Transportschicht

- Einführung
- UDP
- Fehlerkontrolle
- TCP
- TLS
- QUIC

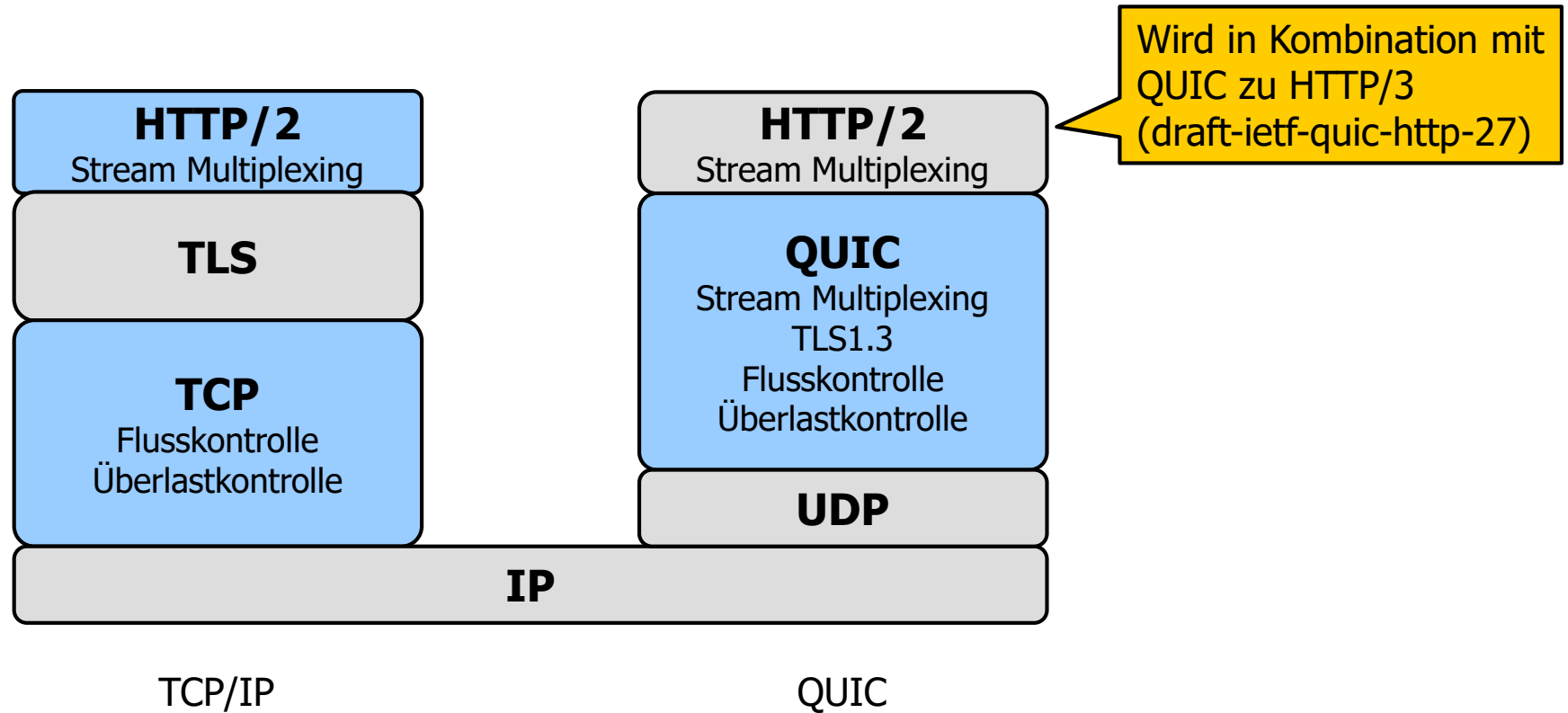
# QUIC

## ■ Quick UDP Internet Connections (QUIC)

- Zuverlässiges Transportprotokoll entwickelt von Google, bereits eingesetzt beim Zugriff vom Chrome Browser auf Google Webseiten
- Momentan in der Standardisierung bei der IETF
- Stream Multiplexing: verschränkte Übertragung mehrerer Objekte, vgl. Stream Control Transmission Protocol (RFC 4960) und HTTP/2
- Basiert auf UDP, um nicht von Firewalls oder anderen Middleboxes geblockt zu werden („Ossification of the Internet“)
- Header weitestgehend verschlüsselt  
(keine Manipulationsmöglichkeiten durch Middleboxes, vgl. Ende-zu-Ende Prinzip)
- Forward Error Correction (FEC)
- Protokollimplementierung Teil der Applikation, nicht des Betriebssystems
- Multipath QUIC in Vorbereitung

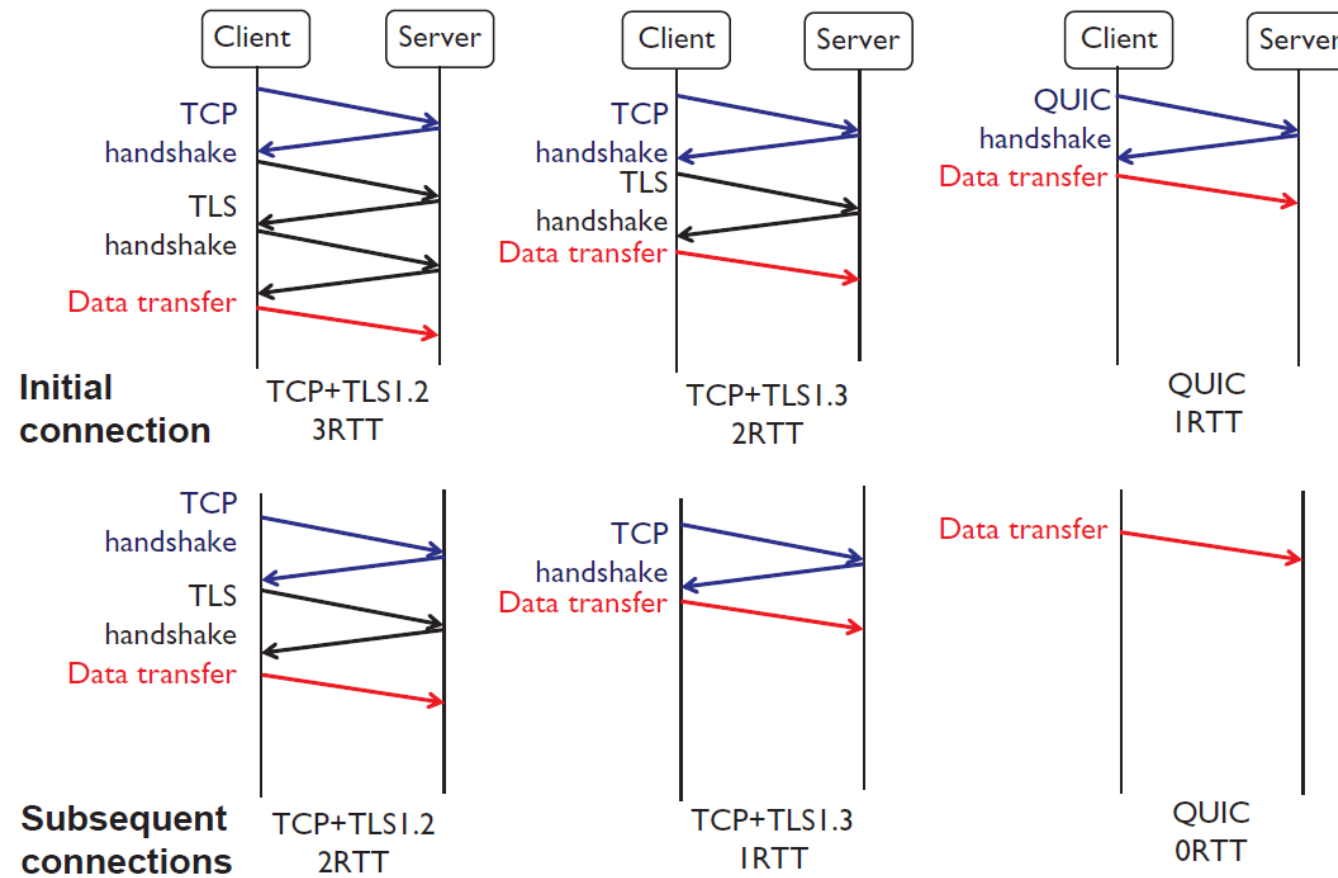
# QUIC

## ■ QUIC im Schichtenmodell



# QUIC

## ■ Schneller Verbindungsaufbau



Quelle: Yong Cui et al. "Innovating Transport with QUIC: Design Approaches and Research Challenges" IEEE Internet Computing



# QUIC Beispiel: http/2+quic/46 (Browser und Server von Google)

Google Chrome → Rechtsklick in Hauptfenster → Untersuchen → Network

The screenshot shows a Google Chrome browser window displaying the YouTube channel page for Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU). The page features a blue header with the text "KNOWLEDGE IN MOTION" and a video player showing a cityscape. The channel has 1.59K subscribers. The Network tab in the developer tools is open, showing a list of requests made by the browser. The requests are filtered by "All" and show a timeline of network activity. The table below lists the requests:

Name	Status	Protocol	Type	Size	Waterfall
favcon...	200	http/2+quic/46	png	16.0 KB	
videopl...	200	http/2+quic/46	xhr	128 KB	
videopl...	200	http/2+quic/46	xhr	320 KB	
videopl...	200	http/2+quic/46	xhr	236 KB	
M3l8V...	200	http/2+quic/46	script	5.4 KB	
ad_stat...	200	http/2+quic/46	script	385 B	
genera...	204	http/2+quic/46	text/pl...	36 B	
csi_204...	204	http/2+quic/46	text/ht...	63 B	
videopl...	200	http/2+quic/46	xhr	575 KB	
watchti...	204	http/2+quic/46	text/ht...	44 B	
atr?ns...	204	http/2+quic/46	xhr	101 B	
qoe?ev...	204	http/2+quic/46	text/ht...	46 B	
log_ev...	200	http/2+quic/46	xhr	103 B	

At the bottom of the Network tab, the summary shows: 96 requests, 4.5 MB transferred, 10.9 MB resources, and a finish time of 15.55 s.