Quelle: Computernetzwerke: Der Top-Down-Ansatz, 6. Ausgabe
Jim Kurose, Keith Ross; Pearson, März 2014

Technische Grundlagen der Informatik 2 – Teil 4: Layer 4 TCP

Philipp Rettberg / Sebastian Harnau

Block 7/18

Transportschicht (Layer 4)

TCP

TCP: Überblick

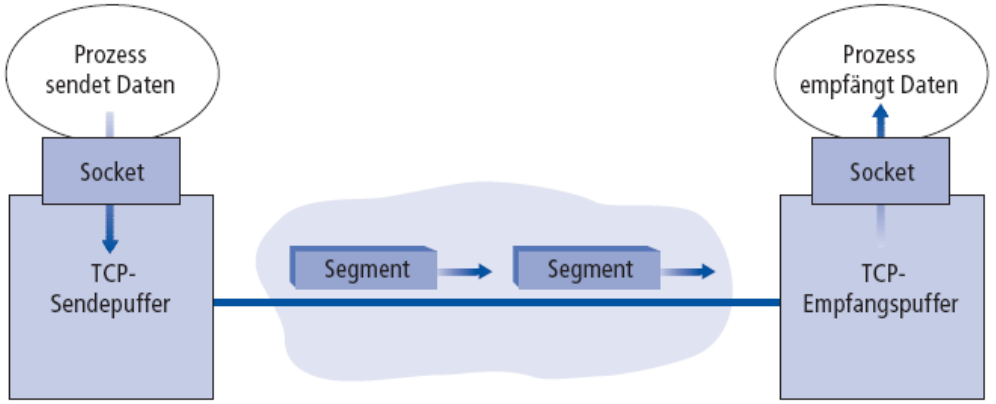
Vollduplex:
- Daten fließen in beide Richtungen
- MSS: Maximum Segment Size

Verbindungsorientiert:
- Handshaking (Austausch von Kontrollnachrichten) initialisiert den Zustand im Sender und Empfänger, bevor Daten ausgetauscht werden

Flusskontrolle:
- Sender überfordert den Empfänger nicht

Punkt-zu-Punkt:
- Ein Sender, ein Empfänger
- Zuverlässiger, reihenfolgeerhaltender Byte-Strom: Keine “Nachrichtengrenzen”

Pipelining:
- TCP-Überlast- und -Fluss-kontrolle verändern die Größe der Sender- & Empfängerfenster



TCP-Segmentaufbau

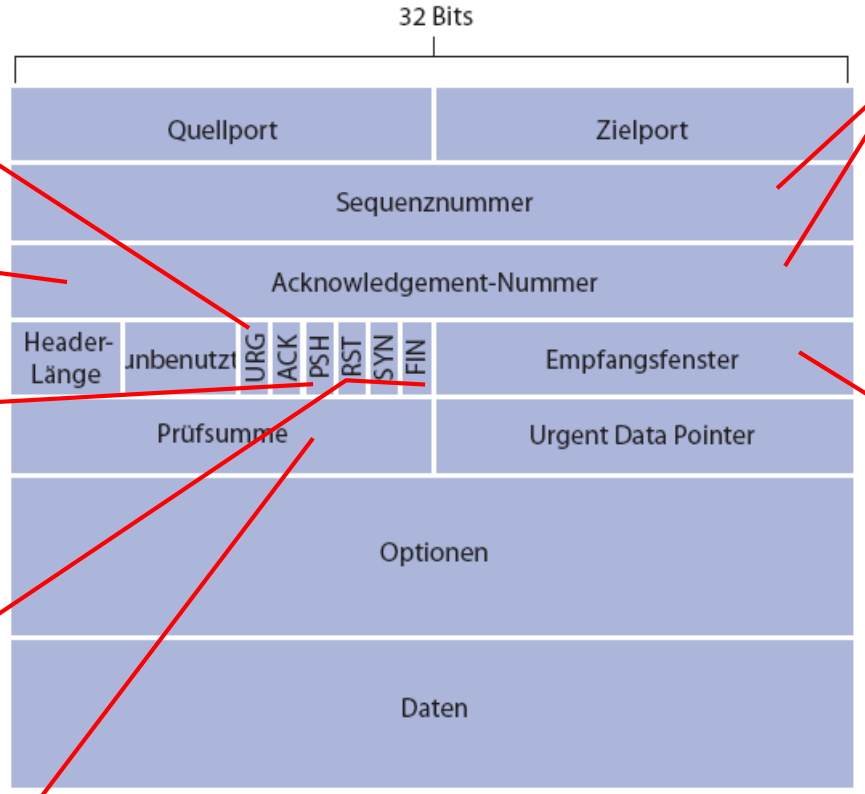
URG: urgent data (üblicherweise nicht benutzt)

ACK: ACK-Nummer gültig

PSH: Push – Daten direkt nach oben weitergeben (meist nicht benutzt)

RST, SYN, FIN: Befehle zum Verbindungs-auf- und -abbau

Prüfsumme (wie bei UDP)



Zählen Bytes und nicht Segmente!

Anzahl an Bytes, die der Emfpänger bereit ist, entgegenzunehmen

TCP-Verbindungsmanagement

TCP-Sender und TCP-Empfänger bauen eine Verbindung auf, bevor sie Daten austauschen.

Initialisieren der TCP-Variablen:

- Sequenznummern, Informationen für Flusskontrolle (z.B. RcvWindow)

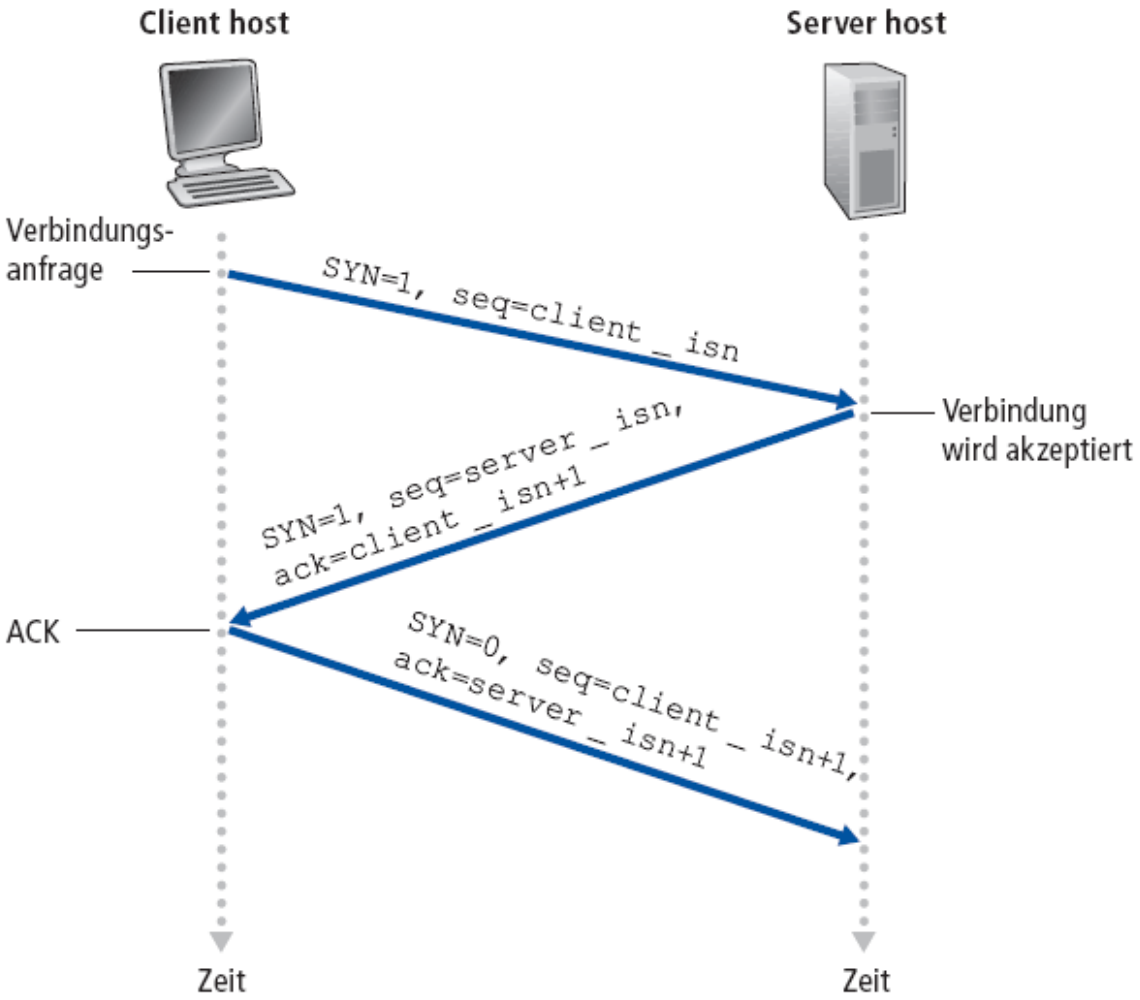
Client: Initiator

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

Server: vom Client kontaktiert

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

TCP: Drei-Wege-Handshake



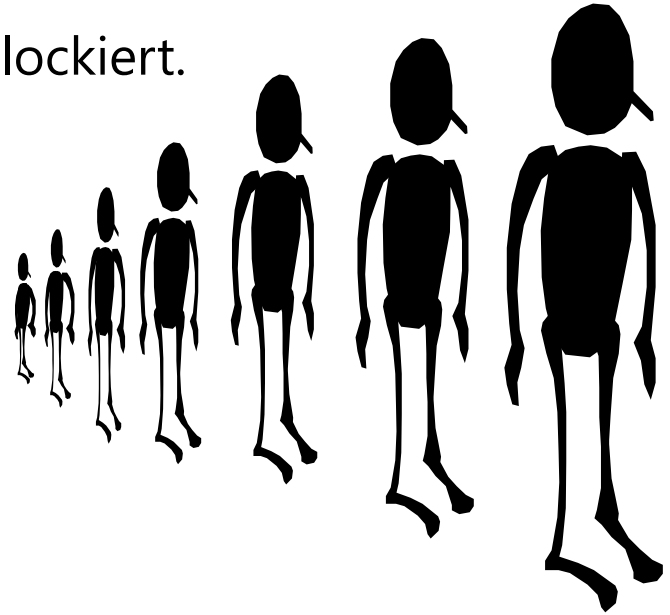
- Schritt 1: Client sendet TCP-SYN-Segment an den Server
  - Initiale Sequenznnummer (Client->Server)
  - keine Daten
- Schritt 2: Server empfängt SYN und antwortet mit SYNACK
  - Server legt Puffer an
  - Initiale Sequenznummer (Server->Client)
- Schritt 3: Client empfängt SYNACK und antwortet mit einem ACK – dieses Segment darf bereits Daten beinhalten

TCP: Der SYN-Flood-Angriff

- In Schritt 2 legt der Server Verbindungsvariablen und Puffer an.
- Was passiert, wenn Schritt 3 nicht erfolgt?
  - Die Löschung (nach Timeout) erfolgt erst nach einer Minute und mehr.
  - Bei entsprechender Paketzahl in kurzer Zeit wird der Server blockiert.

➢ **Denial-of-Service-Angriff**

  - Kein dauerhafter technischer Schaden oder Datenverlust
  - Server wegen Überlast für dritte nicht erreichbar.



TCP: SYN-Cookies

- Bei Anfrage ermittelt der Server eine initiale TCP-Sequenznummer (Cookie) aus:
  - Quell-IP-Adresse
  - Ziel-IP-Adresse
  - Quell-Portnummer
  - Geheimer Wert (nur dem Server bekannt)

- Rücksendung als SYN-ACK mit der speziellen Sequenznummer und **Vergessen der Kommunikationsanfrage**

- Antwortet der Client mit ACK, sendet er die errechnete Sequenznummer+1 zurück.
  - Neuberechnung der Sequenznummer
  - Wenn sie stimmt: Alloziierung einer offenen Verbindung mit Puffer etc.

- Antwortet der Client nicht, hat der Server keinen Schaden genommen.

TCP-Sequenznummern und -ACKs

Sequenznummern:

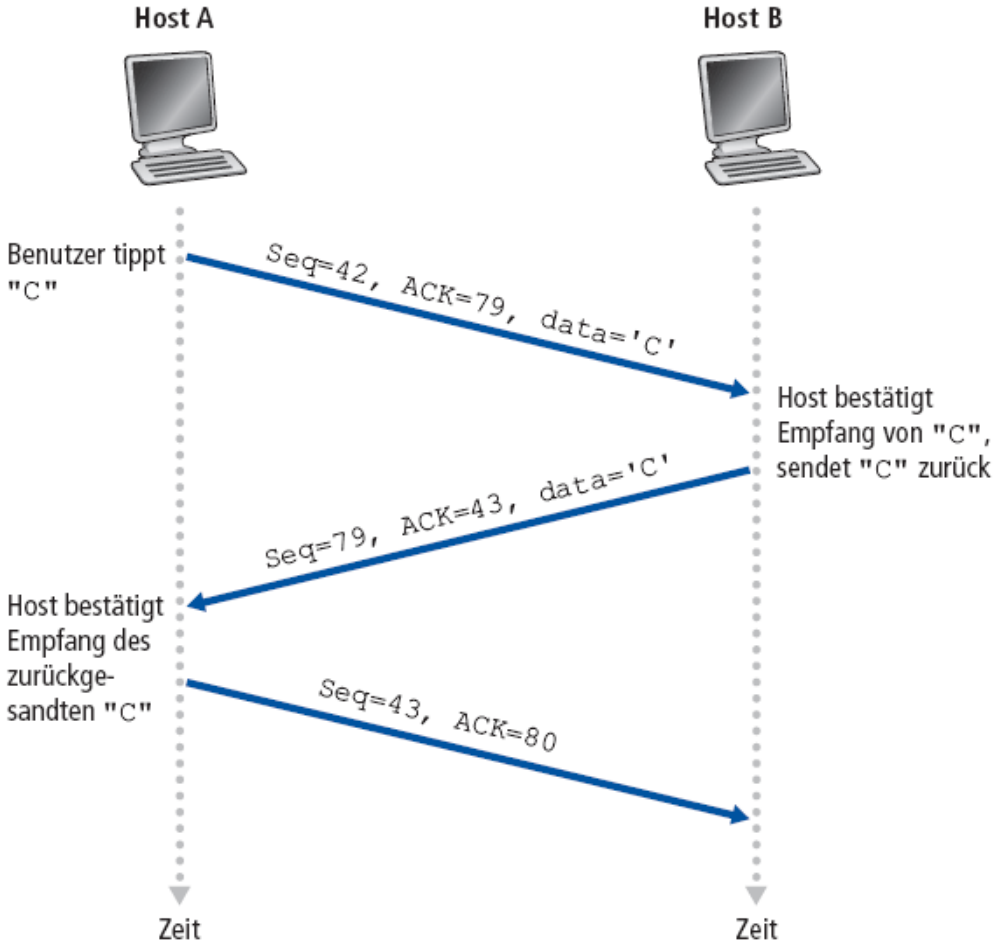
- Nummer des ersten Byte im Datenteil

ACKs:

- Sequenznummer des nächsten Byte, das von der Gegenseite erwartet wird
- Kumulative ACKs

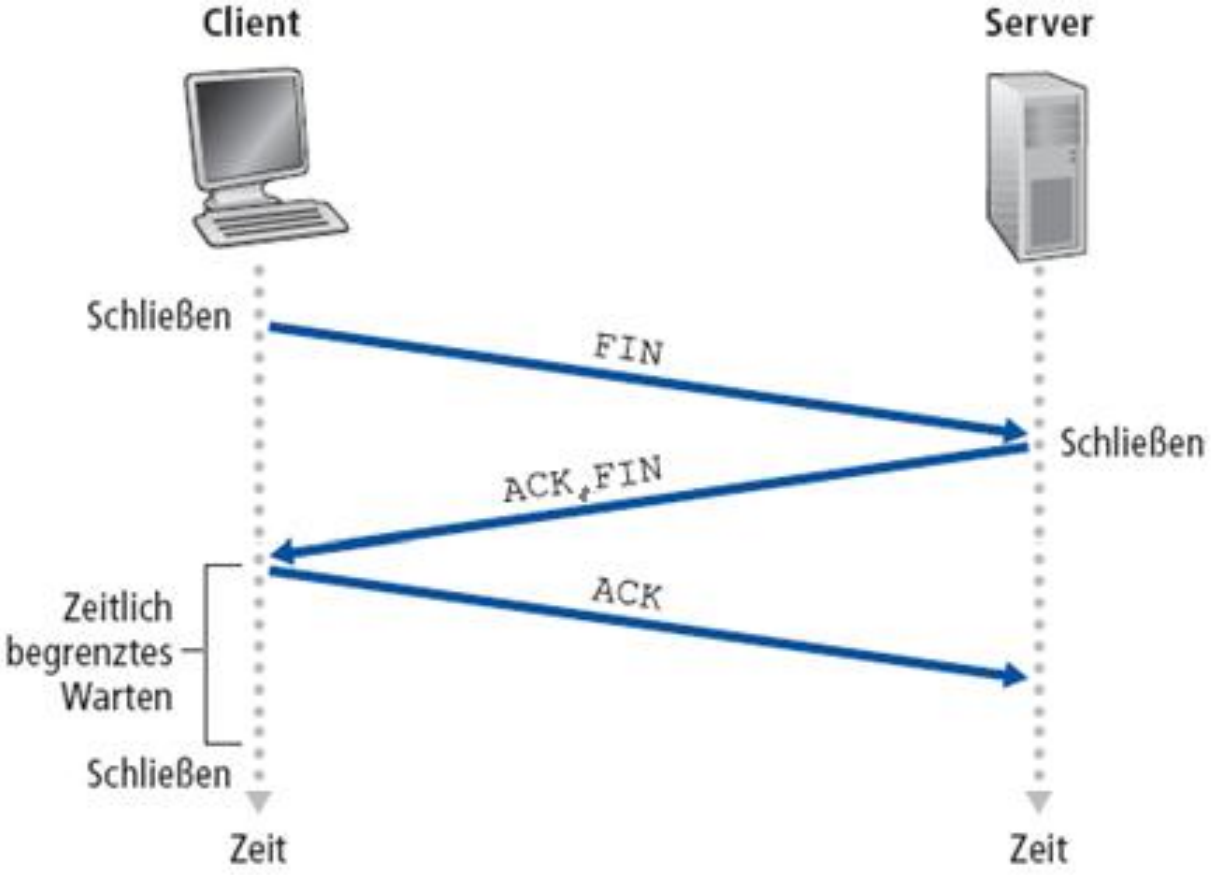
Frage: Wie werden Segmente behandelt, die außer der Reihe ankommen?

- Wird von der TCP-Spezifikation nicht vorgeschrieben!
- Bestimmt durch die Implementierung



TCP-Verbindungsmanagement - Schließen

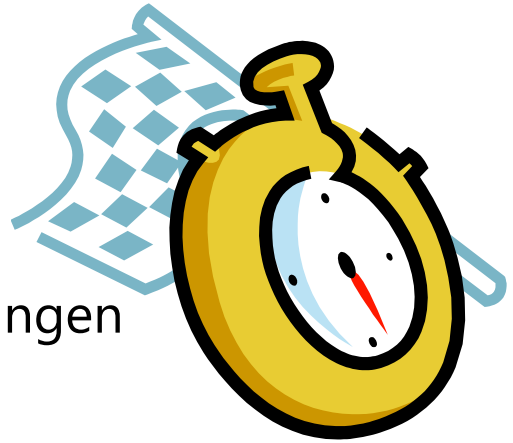
- `clientSocket.close();`
- Schritt 1: Client sendet ein TCP-FIN-Segment an den Server
- Schritt 2: Server empfängt FIN, antwortet mit ACK; dann sendet er ein FIN (kann im gleichen Segment erfolgen)
- Schritt 3: Client empfängt FIN und antwortet mit ACK . Beginnt einen “Timed- Wait”-Zustand – er antwortet auf Sendewiederholungen des Servers mit ACK
- Schritt 4: Server empfängt ACK und schließt Verbindung



TCP-Rundlaufzeit und -Timeout

Frage: Wie bestimmt TCP den Wert für den Timeout?

- Größer als die Rundlaufzeit (Round Trip Time, RTT)
- Aber: RTT ist nicht konstant



Zu kurz: unnötige Timeouts und dadurch unnötige Übertragungswiederholungen
Zu lang: langsame Reaktion auf den Verlust von Segmenten

Frage: Wie kann man die RTT schätzen?

SampleRTT:

- gemessene Zeit vom Absenden eines Segments bis zum Empfang des dazugehörenden ACKs
- Segmente mit Übertragungswiederholungen werden ignoriert
- SampleRTT ist nicht konstant, wir brauchen einen "glatteren" Wert -> Durchschnitt über mehrere Messungen

TCP-Rundlaufzeit und -Timeout

Um die geschätzte Rundenzeit **EstimatedRTT** zu ermitteln, werden die letzten Messungen gemittelt. Verfahren:

$$\mathbf{EstimatedRTT_n = (1- \alpha)*EstimatedRTT_{n-1} + \alpha*SampleRTT_n}$$

Dies nennt man das „**Exponential weighted moving average**“, weil der Einfluss vergangener Messungen sich exponentiell schnell verringert.

Üblicher Wert: $\alpha = 0.125$, d.h. Gewichtung alt zu neu mit 7:1

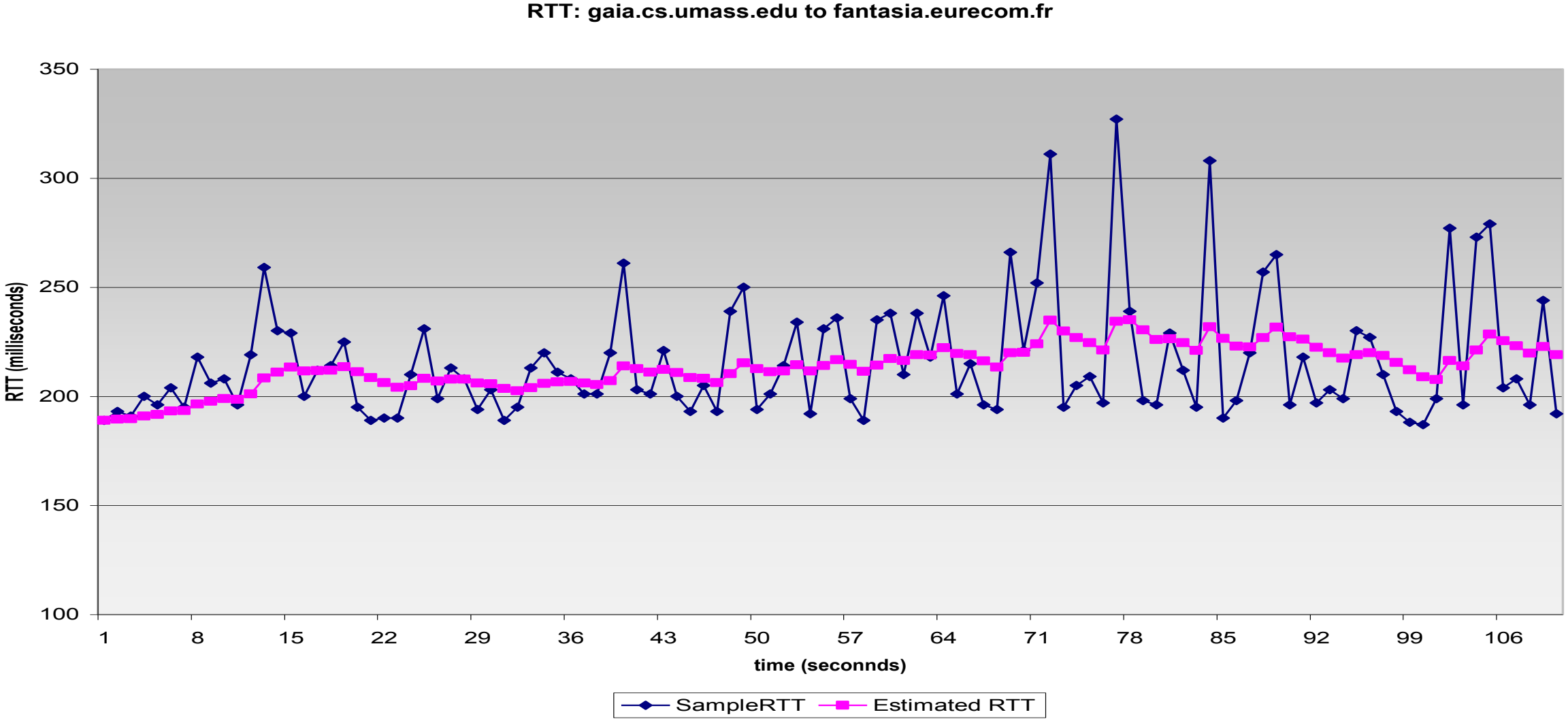
Der ermittelte Wert ist ein Anhaltspunkt, wie lange ein Paket in der Regel unter den gegebenen Umständen brauchen sollte.

TCP-Rundlaufzeit und –Timeout
Beispielberechnung

Alpha	Estimated RTT
0,125	15,00

Sample RTT	Estimated RTT Neu	Gewicht des Ausgangswerts
12	14,63	13,13
11	14,17	11,48
13	14,03	10,05
15	14,15	8,79
11	13,75	7,69
12	13,53	6,73
11	13,22	5,89
13	13,19	5,15
9	12,67	4,51
14	12,83	3,95
11	12,60	3,45

Beispiel für die RTT-Bestimmung



TCP-Rundlaufzeit und -Timeout

- Wie aus der Grafik ersichtlich würde ca. die Hälte der Pakete nach dem Timeout ankommen.
- Dementsprechend benötigt das EstimatedRTT einen “Sicherheitsabstand”
- Größere Schwankungen von EstimatedRTT -> größerer Sicherheitsabstand
- Bestimmung, wie sehr SampleRTT von EstimatedRTT abweicht (deviation=Dev):

$$DevRTT_n = (1-\beta)*DevRTT_{n-1} + \beta*|SampleRTT_n-EstimatedRTT_n|$$

(üblicherweise: $\beta = 0.25$)

- Zeitpunkt fürTimeout:

$$TimeoutInterval = EstimatedRTT_n + 4*DevRTT_n$$

TCP-Rundlaufzeit und –Timeout
Beispielberechnung und Übungsaufgabe

Alpha	Estimated RTT_0	Dev RTT_0	Beta
0,125	15,00	2,00	0,25

$$EstimatedRTT_n = (1 - \alpha) * EstimatedRTT_{n-1} + \alpha * SampleRTT_n$$

$$DevRTT_n = (1 - \beta) * DevRTT_{n-1} + \beta * |SampleRTT_n - EstimatedRTT_n|$$

$$TimeoutInterval_n = EstimatedRTT_n + 4*DevRTT_n$$

n	Sample RTT	Estimated RTT Neu	DevRTT Neu	Time-out
1	12	14,63	2,16	23,25
2	11	14,17	2,41	23,81
3	13	14,03	2,06	22,28
4	15	14,15	1,76	21,19
5	11	13,75	2,01	21,79
6	12	13,53	1,89	21,10
7	11	13,22	1,97	21,11
8	13			
9	9			
10	14			
11	11			

TCP: zuverlässiger Datentransfer

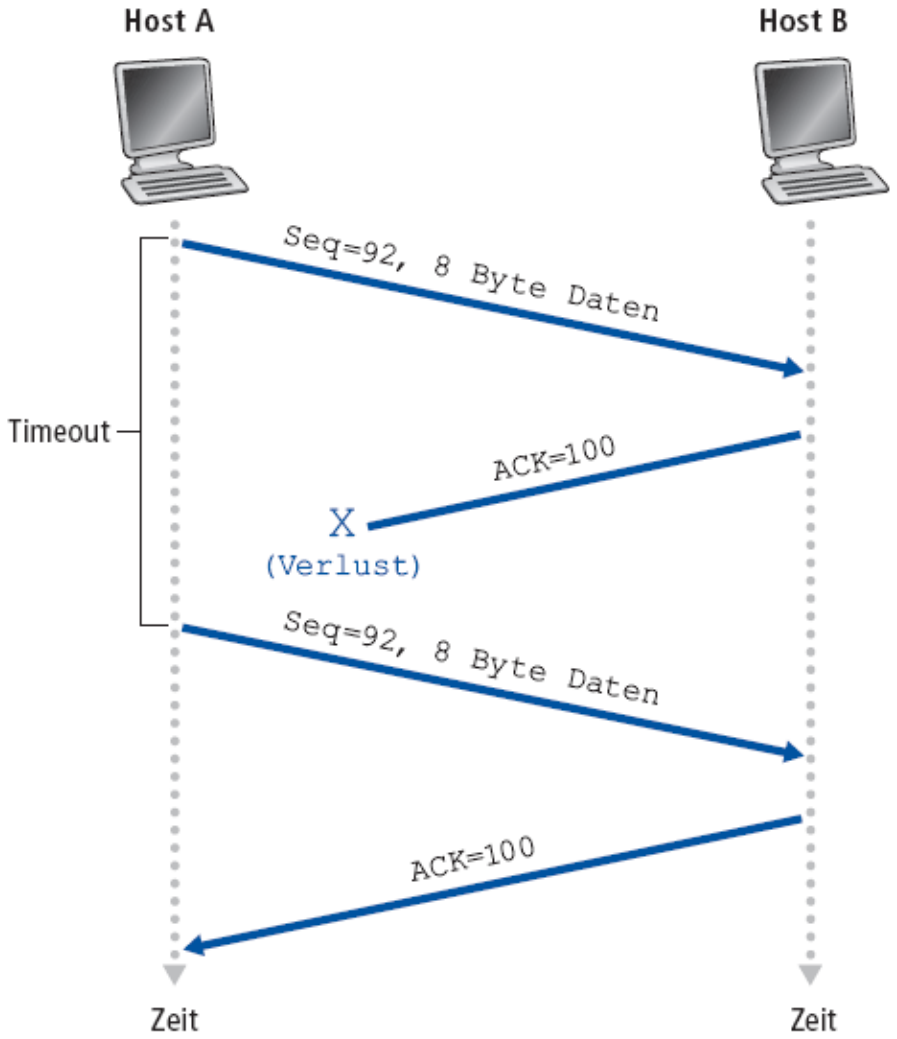
- TCP stellt einen **zuverlässigen Datentransfer** über den **unzuverlässigen Datentransfer** von IP zur Verfügung
- Pipelining von Segmenten
- Kumulative ACKs
- TCP verwendet **einen einzigen Timer** für Übertragungs-wiederholungen

- Übertragungswiederholungen werden ausgelöst durch:
  - Timeout
  - Doppelte ACKs
- Zu Beginn betrachten wir einen vereinfachten TCP-Sender:
  - Ignorieren von doppelten ACKs
  - Ignorieren von Fluss- und Überlastkontrolle

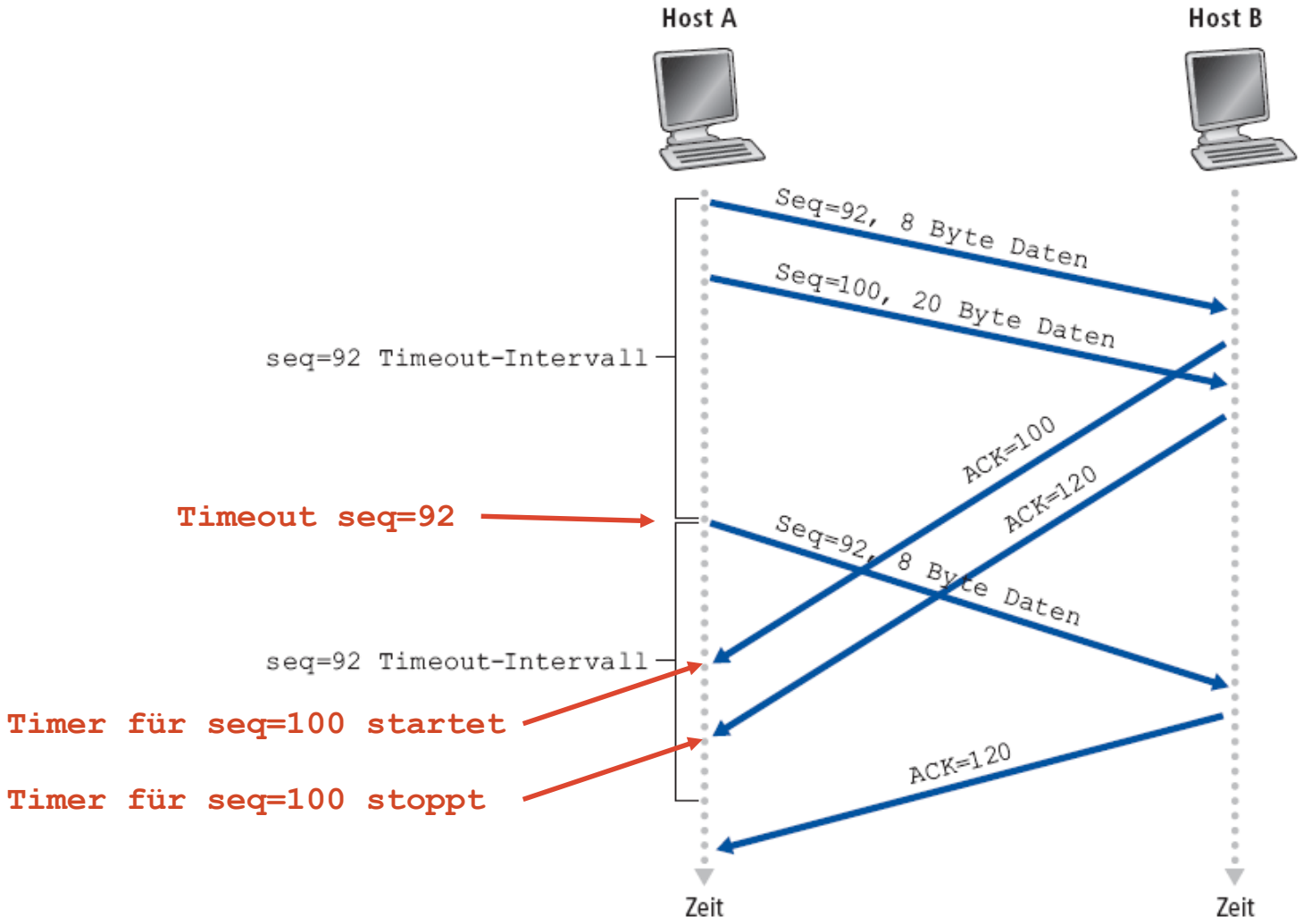
TCP-Ereignisse im Sender

- Daten von Anwendung erhalten:
  - Erzeuge Segment mit geeigneter Sequenznummer (Nummer des ersten Byte im Datenteil)
  - Timer für das älteste unbestätigte Segment starten, wenn er noch nicht läuft
  - Laufzeit des Timers: TimeOutInterval
- Timeout:
  - Erneute Übertragung des Segments, für das der Timeout aufgetreten ist
  - Starte Timer neu
  - ACK empfangen:
    - Wenn damit bisher unbestätigte Daten bestätigt werden:
      - Aktualisiere die Informationen über bestätigte Segmente
      - Starte Timer neu, wenn noch unbestätigte Segmente vorhanden sind

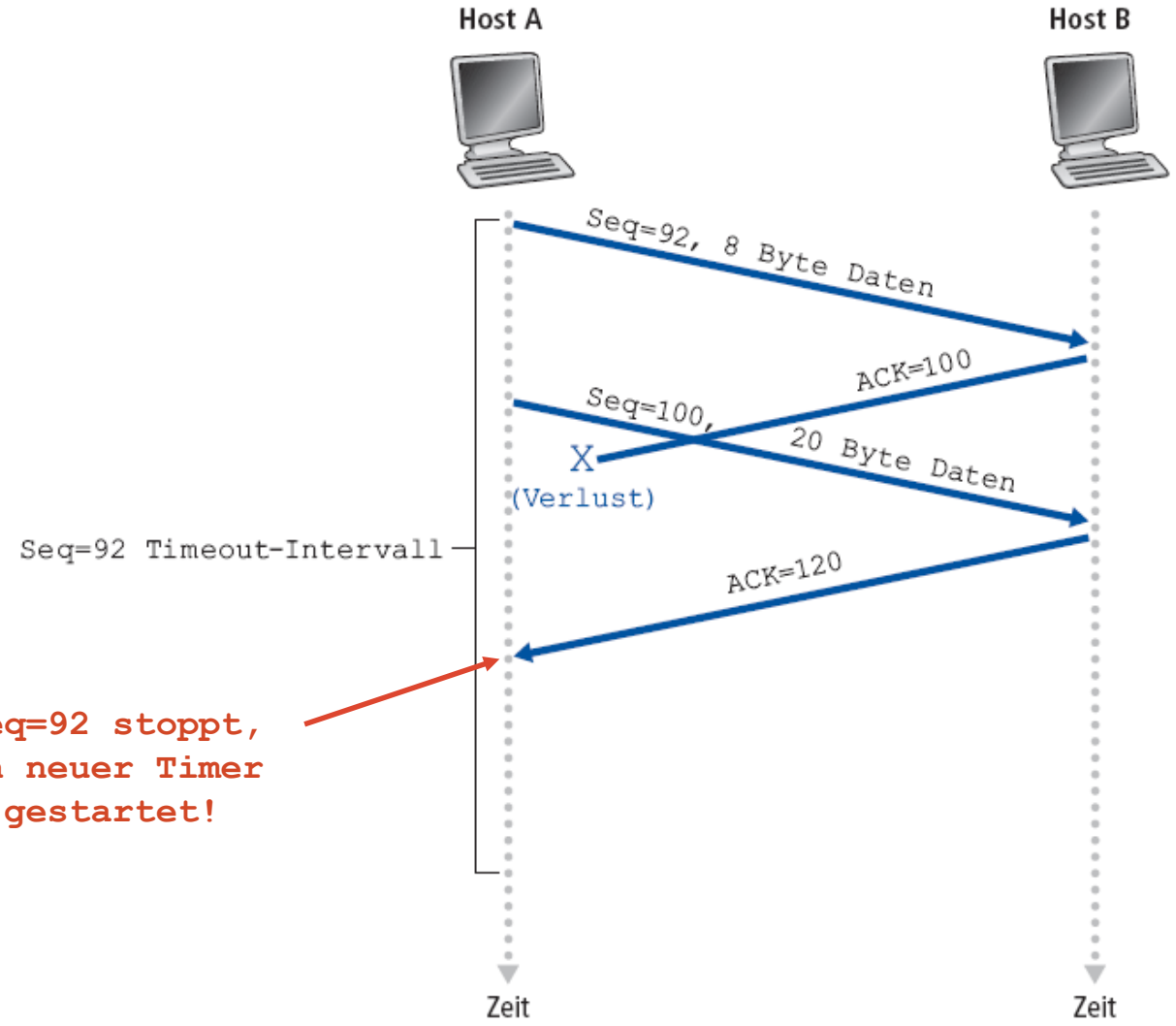
TCP: Beispiele für Übertragungswiederholungen – Paketverlust



TCP: Beispiele für Übertragungswiederholungen – verfrühter Timeout



TCP: Beispiele für Übertragungswiederholungen – kumulative ACKs



Timer für seq=92 stoppt, es wird kein neuer Timer für seq=100 gestartet!

Fast Retransmit

Zeit für Timeout ist häufig sehr lang:

- Große Verzögerung vor einer Neuübertragung

Erkennen von Paketverlusten durch doppelte ACKs:

- Sender schickt häufig viele Segmente direkt hintereinander
- Wenn ein Segment verloren geht, führt dies zu vielen doppelten ACKs

Wenn der Sender 3 Duplikate eines ACK erhält, dann nimmt er an, dass das Segment verloren gegangen ist:

- Fast Retransmit (schnelle Sendewiederholung):
  Segment erneut schicken, bevor der Timer ausläuft

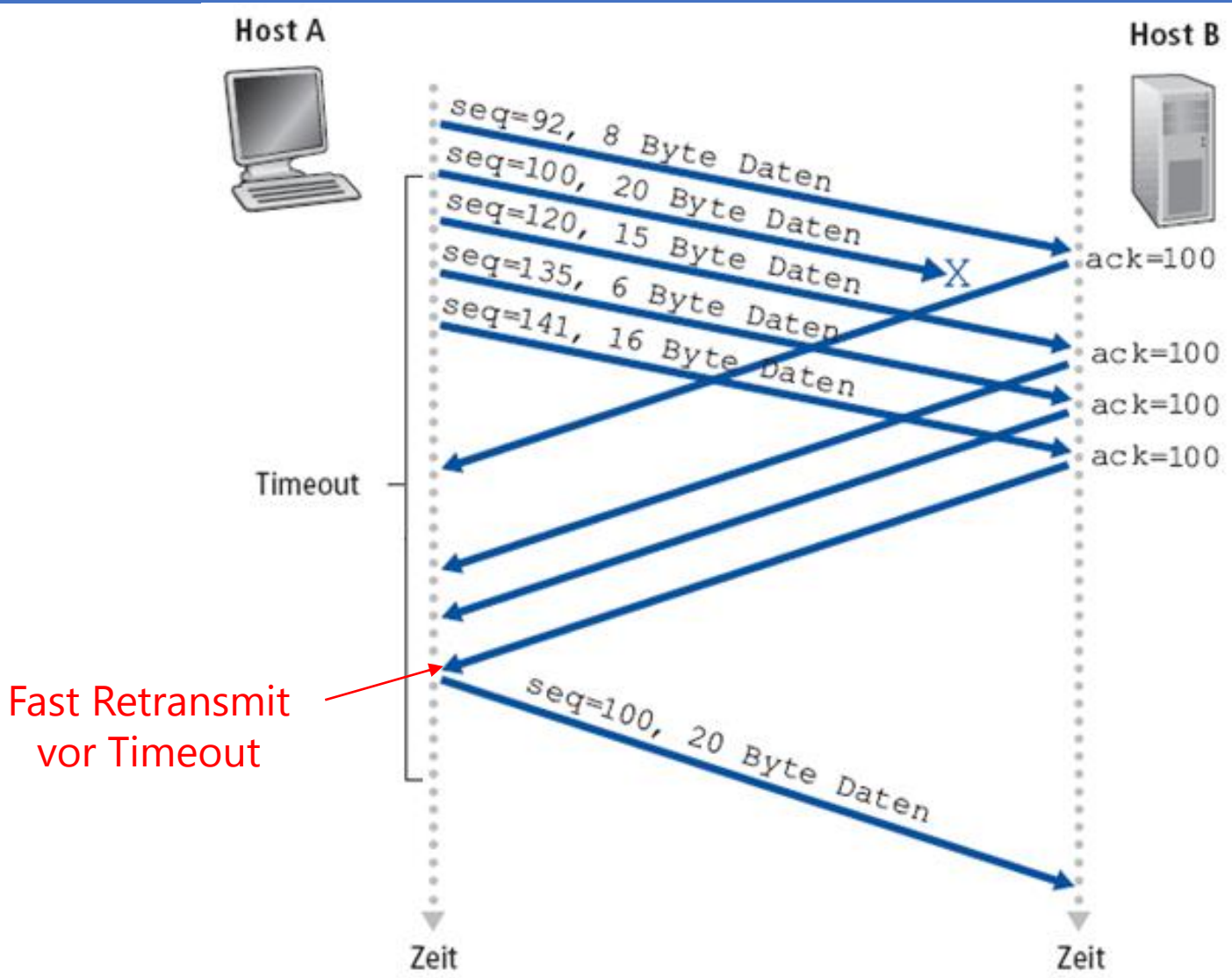
Fast Retransmit: Algorithmus

```
event: ACK empfangen, Acknowledgement-Nummer ist y
                if (y > SendBase) {
                        SendBase = y
                        if (wenn es noch unbestätigte Segmente gibt)
                                starte Timer
                    }
                else {
                          erhöhe den Zähler für doppelte ACKs für y um eins
                          if (Zähler für doppete ACKs für y = 3) {
                                  Neuübertragung des Segments mit Sequenznummer y
                           }
```

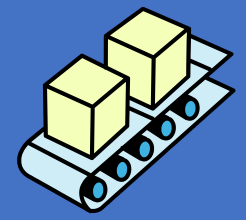
Ein doppeltes ACK für ein bereits bestätigtes Segment

Fast Retransmit

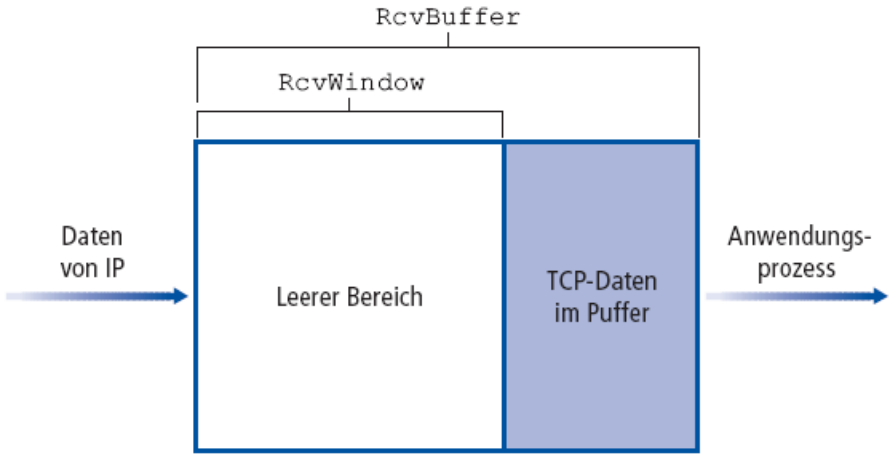
Fast Retransmit: Beispiel



TCP-Flusskontrolle



Empfängerseite von TCP hat einen Empfängerpuffer:



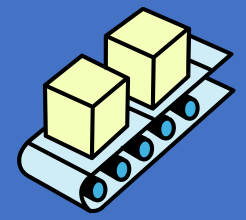
Die Anwendung kommt unter Umständen nicht mit dem Lesen hinterher

Flusskontrolle

Sender schickt nicht mehr Daten, als der Empfänger in seinem Puffer speichern kann

- Dienst zum Angleichen von Geschwindigkeiten: Senderate wird an die Verarbeitungsrate der Anwendung auf dem Empfänger angepasst

TCP-Flusskontrolle: Funktionsweise

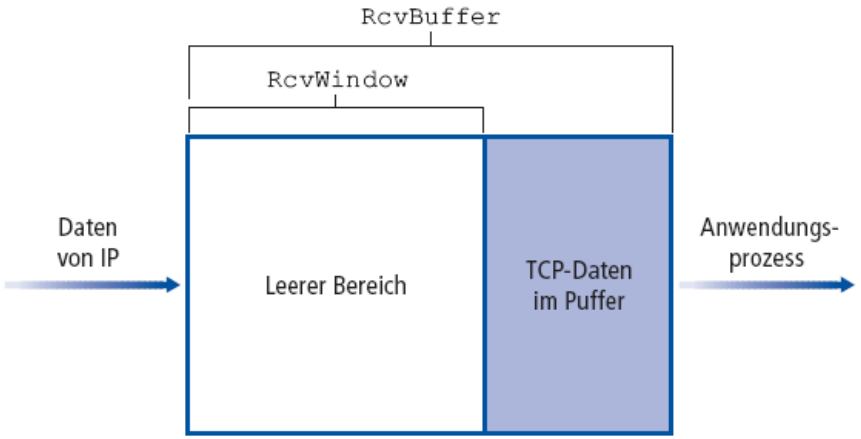


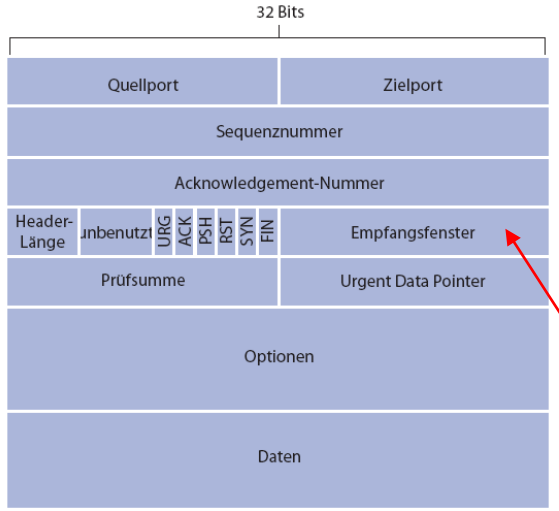
Annahme: Empfänger verwirft Segmente, die außer der Reihe ankommen

Platz im Puffer

= RcvWindow

= RcvBuffer-[LastByteRcvd - LastByteRead]





- **Empfänger** kündigt den Platz durch **RcvWindow** im TCP-Header an
- **Sender** begrenzt seine unbestätigt gesendeten Daten auf **RcvWindow**
  - Dann ist garantiert, dass der Puffer im Empfänger nicht überläuft

Rekapitulieren Sie: TCP-Mechanismen für die zuverlässige Datenübertragung

Mechanismus	Einsatzzweck, Kommentare