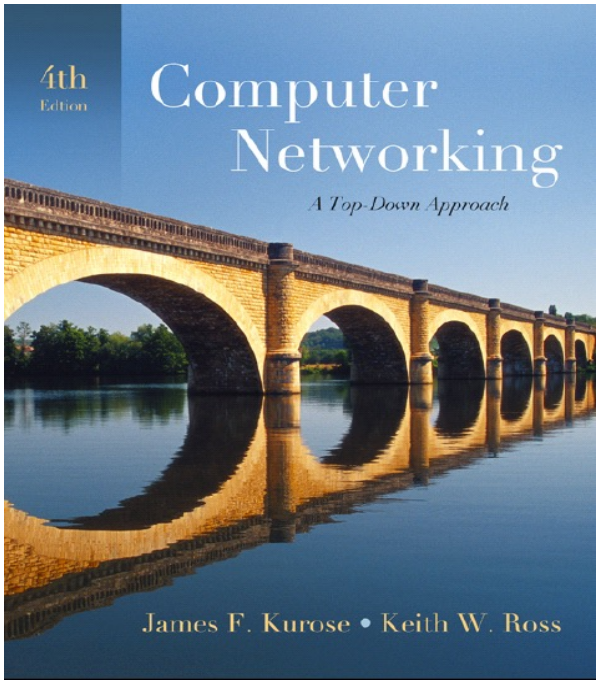
Kapitel 3
Transportschicht



Computernetzwerke: Der Top-Down-Ansatz ,
4. Ausgabe.
Jim Kurose, Keith Ross
Pearson, Juli 2008.

Ein Hinweis an die Benutzer dieses Foliensatzes:

Wir stellen diese Folien allen Interessierten (Dozenten, Studenten, Lesern) frei zur Verfügung. Da sie im PowerPoint-Format vorliegen, können Sie sie beliebig an Ihre Bedürfnisse anpassen. Wir haben sehr viel Arbeit in diesen Foliensatz investiert. Als Gegenleistung für dessen Verwendung bitten wir Sie um Folgendes:

- Wenn Sie diese Folien (z.B. in einer Vorlesung) verwenden, dann nennen Sie bitte die Quelle (wir wollen ja, dass möglichst viele Menschen unser Buch lesen!).

- Wenn Sie diese Folien auf einer Webseite zum Herunterladen anbieten, dann geben Sie bitte die Quelle und unser Copyright an diesem Foliensatz an.

Danke und viel Spaß beim Lehren und Lernen mit diesem Foliensatz! JFK/KWR

Kapitel 3: Transportschicht

Ziele:

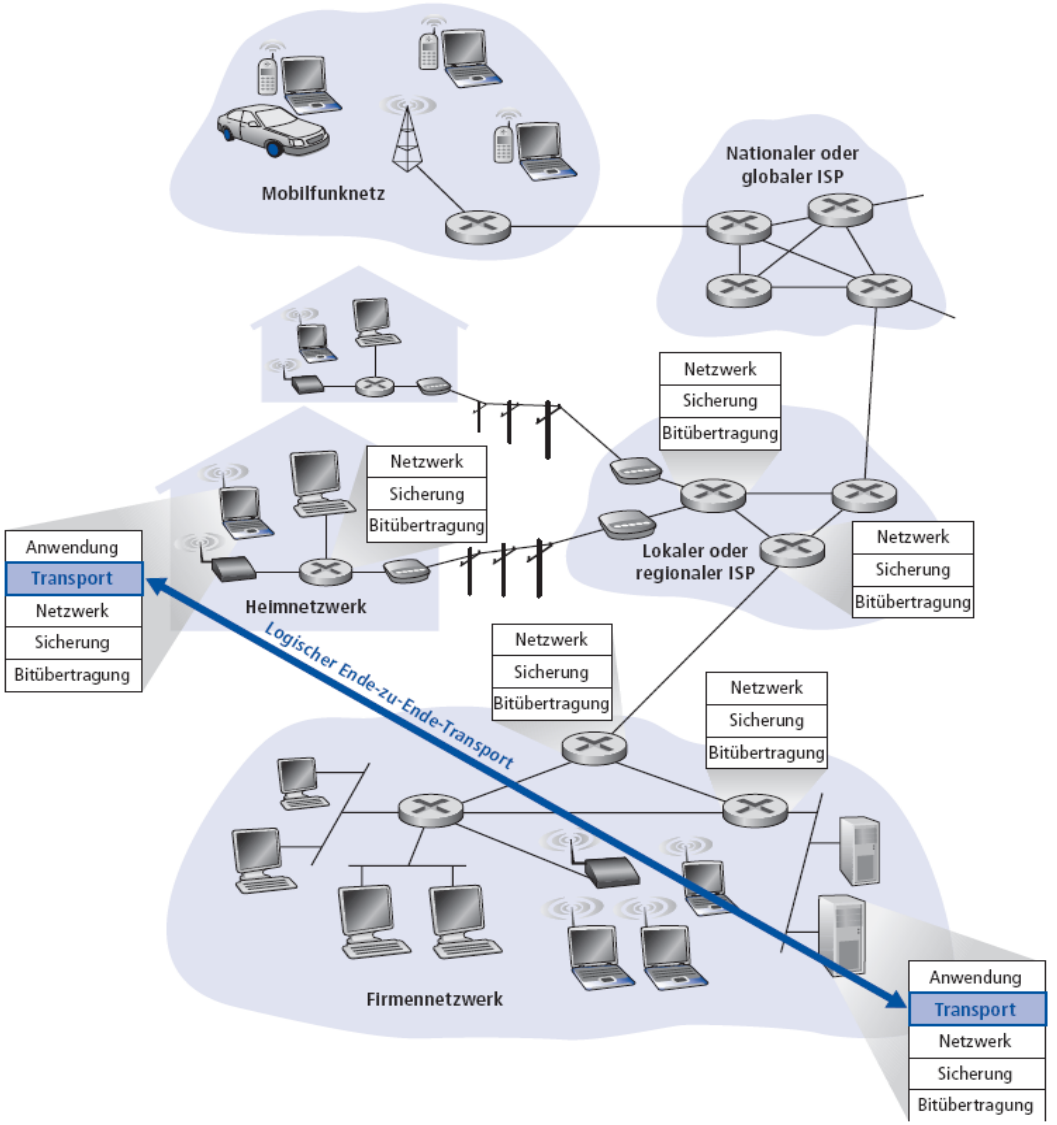
- Verständnis prinzipieller Eigenschaften von Diensten der Transportschicht:
  - Multiplexing/ Demultiplexing
  - Zuverlässiger Transport von Daten
  - Flusskontrolle
  - Überlastkontrolle
- Etwas über die Protokolle der Transportschicht im Internet lernen:
  - UDP: verbindungslos
  - TCP: verbindungsorientiert
  - TCP-Überlastkontrolle

Kapitel 3: Inhalt

- 3.1 Dienste der Tranportschicht
- 3.2 Multiplexing und Demultiplexing
- 3.3 Verbindungsloser Transport: UDP
- 3.4 Grundlagen der zuverlässigen Datenübertragung
- 3.5 Verbindungsorientierter Transport: TCP
  - Segmentstruktur
  - Zuverlässigkeit
  - Flusskontrolle
  - Verbindungsmanagement
- 3.6 Grundlagen der Überlastkontrolle
- 3.7 TCP-Überlastkontrolle
- 3.8 Socket-Programmierung

Transportdienste und -protokolle

- Stellen *logische Kommunikation* zwischen Anwendungsprozessen auf verschiedenen Hosts zur Verfügung
- Transportprotokolle laufen auf Endsystemen
  - Sender: teilt Anwendungsnachrichten in Segmente auf, gibt diese an die Netzwerkschicht weiter
  - Empfänger: fügt Segmente wieder zu Anwendungs-nachrichten zusammen, gibt diese an die Anwendungs-schicht weiter
- Es exisitieren verschiedene Transportschichtprotokolle
  - Internet: TCP und UDP



Transport- und Netzwerkschicht

- *Netzwerkschicht:* logische Kommunikation zwischen Hosts
- *Transportschicht:* logische Kommunikation zwischen Prozessen
  - verwendet und erweitert die Dienste der Netzwerkschicht

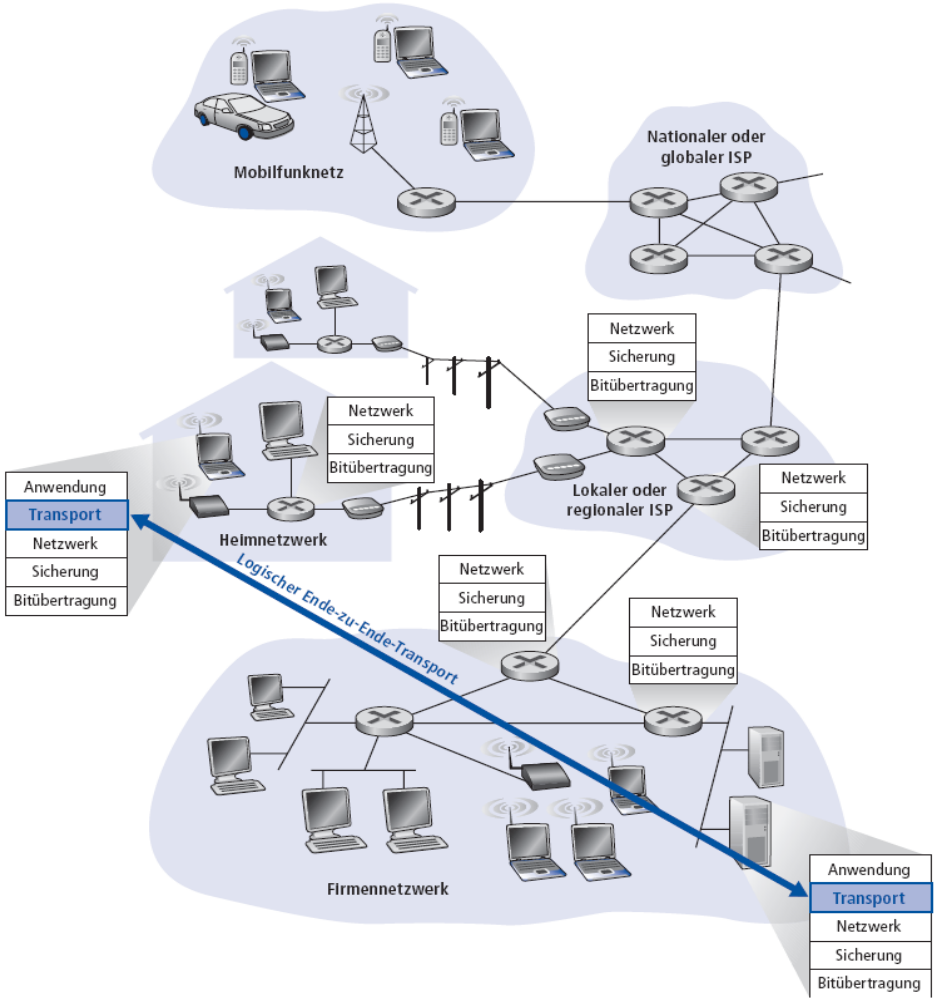
Eine Analogie:

12 Kinder senden Briefe an 12 andere Kinder

- Prozess = Kind
- Anwendungsnachricht = Brief in einem Umschlag
- Host = Haus
- Transportprotokolle = Ann und Bill
- Netzwerkprotokoll = gewöhnlicher Postdienst

Transportprotokolle im Internet

- Zuverlässige, reihenfolgeerhaltende Auslieferung (TCP)
  - Überlastkontrolle
  - Flusskontrolle
  - Verbindungsmanagement
- Unzuverlässige Datenübertragung ohne Reihenfolgeerhaltung: UDP
  - Minimale Erweiterung der “Best-Effort”-Funktionalität von IP
- Dienste, die nicht zur Verfügung stehen:
  - Garantien bezüglich Bandbreite oder Verzögerung



Kapitel 3: Inhalt

- 3.1 Dienste der Tranportschicht
- 3.2 Multiplexing und Demultiplexing
- 3.3 Verbindungsloser Transport: UDP
- 3.4 Grundlagen der zuverlässigen Datenübertragung
- 3.5 Verbindungsorientierter Transport: TCP
  - Segmentstruktur
  - Zuverlässigkeit
  - Flusskontrolle
  - Verbindungsmanagement
- 3.6 Grundlagen der Überlastkontrolle
- 3.7 TCP-Überlast-kontrolle
- 3.8 Socket-Programmierung

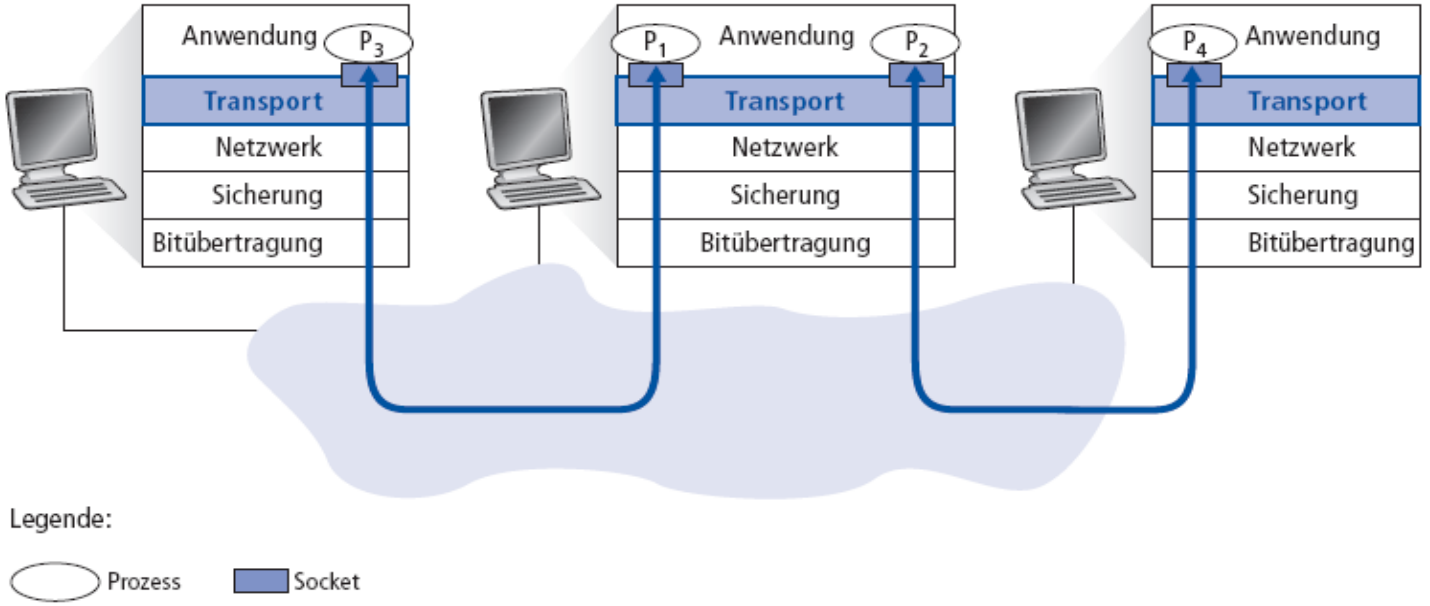
Multiplexing/Demultiplexing

Multiplexing beim Sender:

Daten von mehreren Sockets einsammeln, Daten mit einem Header versehen (der später für das Demultiplexing verwendet wird)

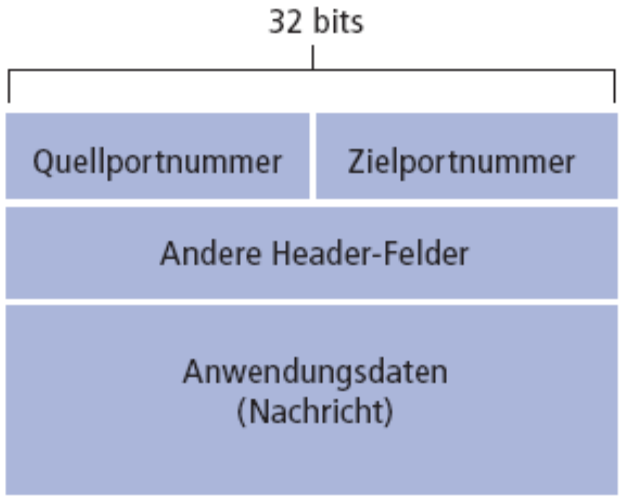
Demultiplexing beim Empfänger:

Empfangene Segmente am richtigen Socket abliefern



Wie funktioniert Demultiplexing?

- Host empfängt IP-Datagramme
  - Jedes Datagramm hat eine Absender-IP-Adresse und eine Empfänger-IP-Adresse
  - Jedes Datagramm beinhaltet ein Transport-schichtsegment
  - Jedes Segment hat eine Absender- und eine Empfänger-Portnummer
- Hosts nutzen IP-Adressen und Portnummern, um Segmente an den richtigen Socket weiterzuleiten



TCP/UDP-Segmentformat

Verbindungsloses Demultiplexing

- Sockets mit Portnummer anlegen:

```
DatagramSocket mySocket1 = new
   DatagramSocket(12534);
DatagramSocket mySocket2 = new
   DatagramSocket(12535);
```

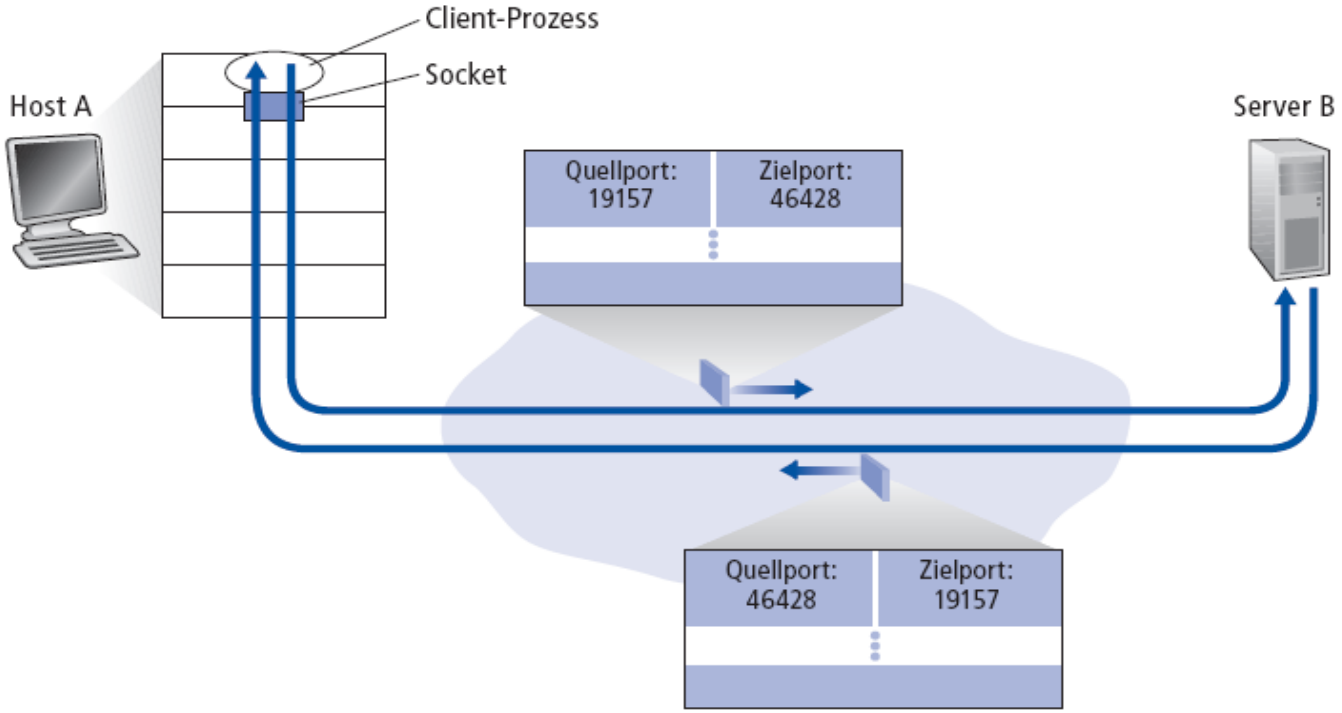
- Beim UDP, ein 2-Tupel identifiziert den Socket (Socket Nummer):

(Empfänger-IP-Adresse, Empfänger-Portnummer)

- Wenn ein Host ein UDP-Segment empfängt:
  - Lese Empfänger-Portnummer
  - Das UDP-Segment wird an den UDP-Socket mit dieser Portnummer weitergeleitet
- IP-Datagramme mit anderer Absender-IP-Adresse oder anderer Absender-Portnummer werden an denselben Socket ausgeliefert

Verbindungsloses Demultiplexing

```
DatagramSocket serverSocket = new DatagramSocket(46428);
```

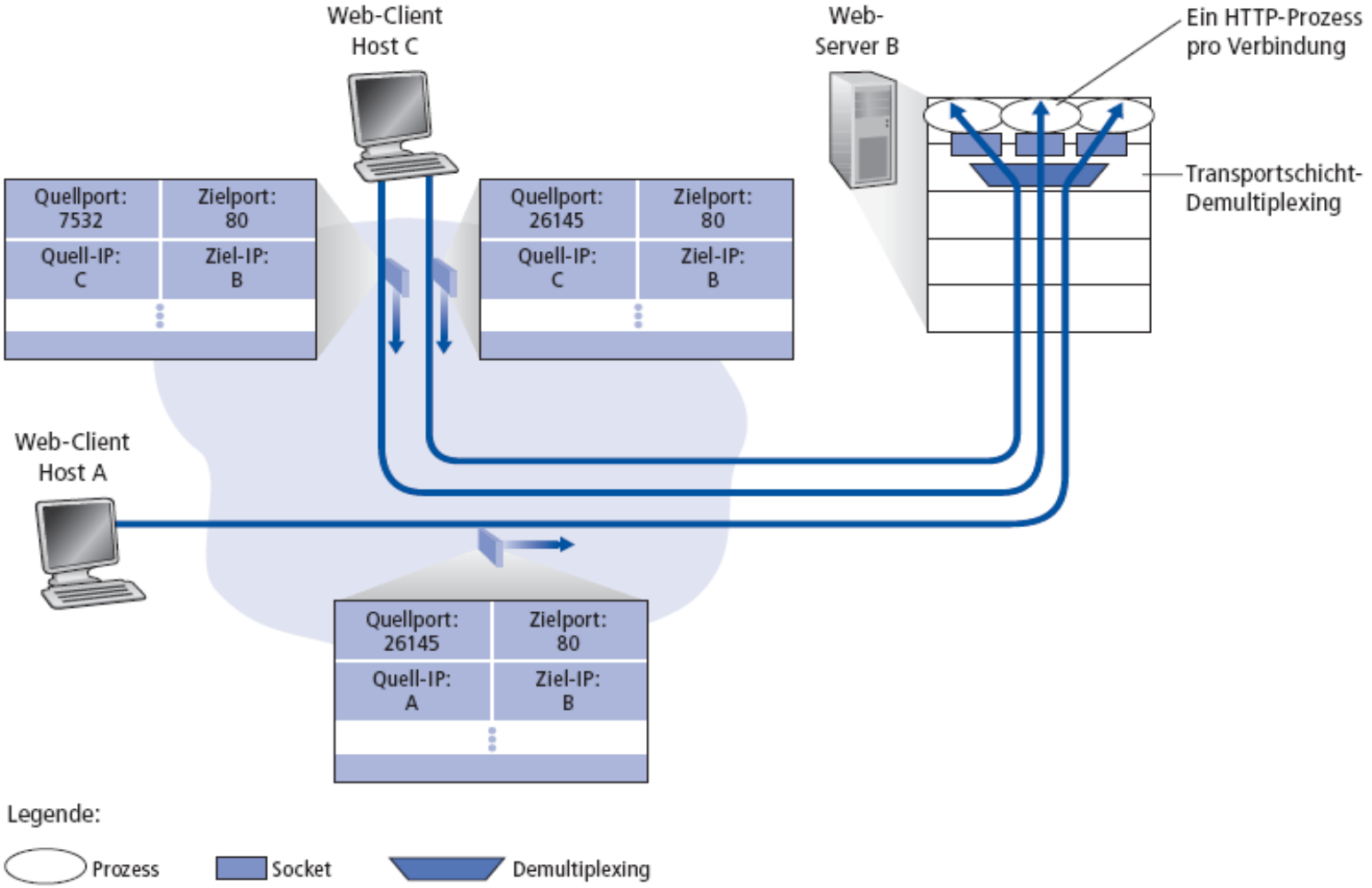


Quellport ist der Port, an den geantwortet werden soll

Verbindungsorientiertes Demultiplexing

- Beim TCP, ein 2-Tupel identifiziert den Socket (Socket Nummer):
  - Absender-IP-Addresse
  - Absender-Portnummer
  - Empfänger-IP-Adresse
  - Empfänger-Portnummer
- Empfänger nutzt alle vier Werte, um den richtigen TCP-Socket zu identifizieren
- Server kann viele TCP-Sockets gleichzeitig offen haben:
  - Jeder Socket wird durch sein eigenes 4-Tupel identifiziert
- Webserver haben verschiedene Sockets für jeden einzelnen Client
  - Bei nichtpersistentem HTTP wird jede Anfrage über einen eigenen Socket beantwortet (dieser wird nach jeder Anfrage wieder geschlossen)

Verbindungsorientiertes Demultiplexing



Kapitel 3: Inhalt

- 3.1 Dienste der Tranportschicht
- 3.2 Multiplexing und Demultiplexing
- 3.3 Verbindungsloser Transport: UDP
- 3.4 Grundlagen der zuverlässigen Datenübertragung
- 3.5 Verbindungsorien-tierter Transport: TCP
  - Segmentstruktur
  - Zuverlässigkeit
  - Flusskontrolle
  - Verbindungsmanagement
- 3.6 Grundlagen der Überlastkontrolle
- 3.7 TCP-Überlast-kontrolle
- 3.8 Socket-Programmierung

UDP: User Datagram Protocol [RFC 768]

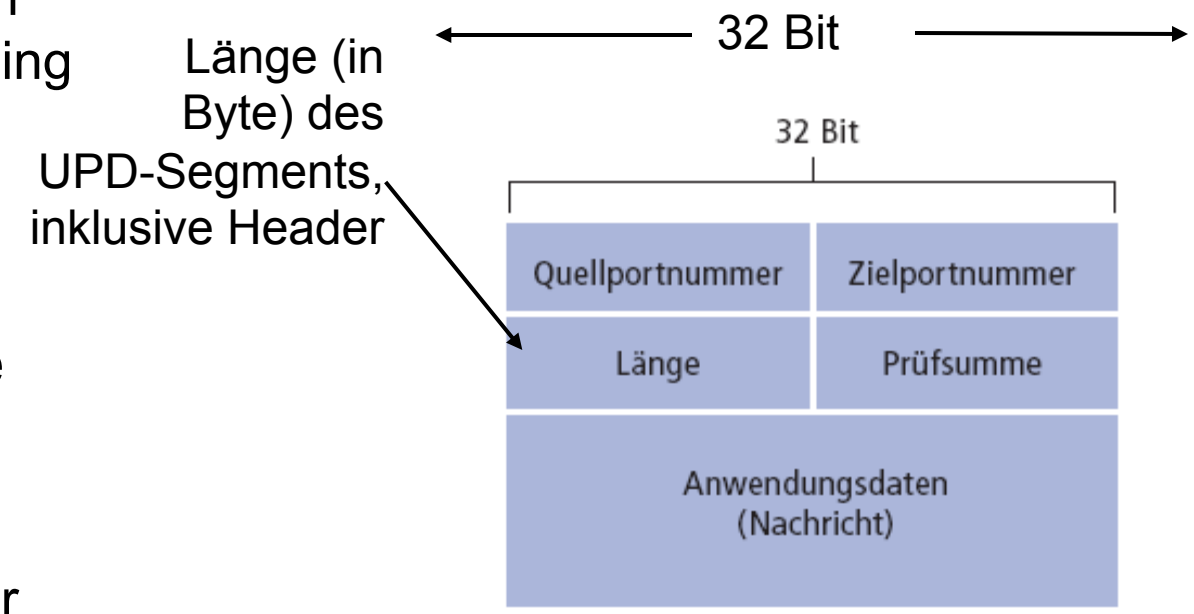
- Minimales Internet-Transportprotokoll
- “Best-Effort”-Dienst, UDP-Segmente können:
  - verloren gehen
  - in der falschen Reihen-folge an die Anwendung ausgeliefert werden
- *Verbindungslos:*
  - kein Handshake zum Verbindungsaufbau
  - jedes UDP-Segment wird unabhängig von allen anderen behandelt

Warum gibt es UDP?

- Kein Verbindungsaufbau (der zu Verzögerungen führen kann)
- Einfach: kein Verbindungszustand im Sender oder Empfänger
- Kleiner Header
- Keine Überlastkontrolle: UDP kann so schnell wie von der Anwendung gewünscht senden
- Prüfsumme

UDP: Fortsetzung

- Häufig für Anwendungen im Bereich Multimedia-Streaming eingesetzt
  - verlusttolerant
  - Mindestrate
- Andere Einsatzgebiete
  - DNS
  - SNMP
- Zuverlässiger Datentransfer über UDP: Zuverlässigkeit auf der Anwendungsschicht implementieren
  - Anwendungsspezifische Fehlerkorrektur!



UDP-Segment-Format

UDP-Prüfsumme

Ziel: Fehler im übertragenen Segment erkennen (z.B. verfälschte Bits)

Sender:

- Betrachte Segment als Folge von 16-Bit-Integer-Werten
- Prüfsumme: Addition (1er-Komplement-Summe) dieser Werte
- Sender legt das invertierte Resultat im UDP-Prüfsummenfeld ab

Empfänger:

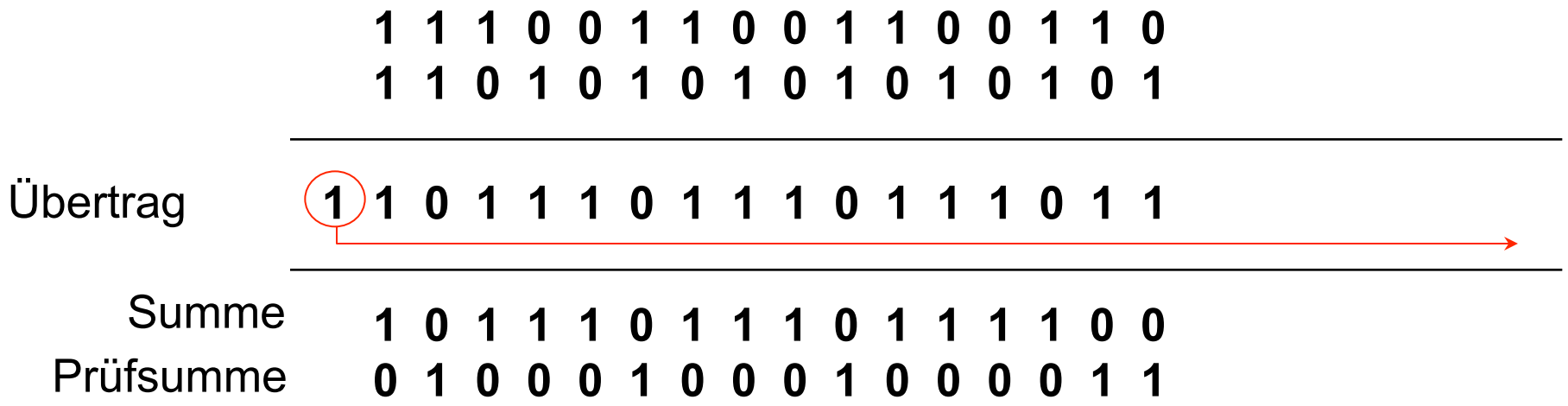
- Berechne die Prüfsumme des empfangenen Segments inkl. des Prüfsummenfeldes
- Sind im Resultat alle Bits 1?
  - NEIN – Fehler erkannt
  - JA – kein Fehler erkannt
    Vielleicht liegt trotzdem ein Fehler vor? Mehr dazu später
    …

UDP-Prüfsumme

bits	0 – 7	8 – 15	16 – 23	24 – 31
0	Source address			
32	Destination address			
64	Zeros	Protocol	UDP length	
96	Source Port		Destination Port	
128	Length		Checksum	
160	Data			

Prüfsummenbeispiel

- Anmerkung
  - Wenn Zahlen addiert werden, dann wird ein Übertrag aus der höchsten Stelle zum Resultat an der niedrigsten Stelle addiert
- Beispiel: Addiere zwei 16-Bit-Integer-Werte

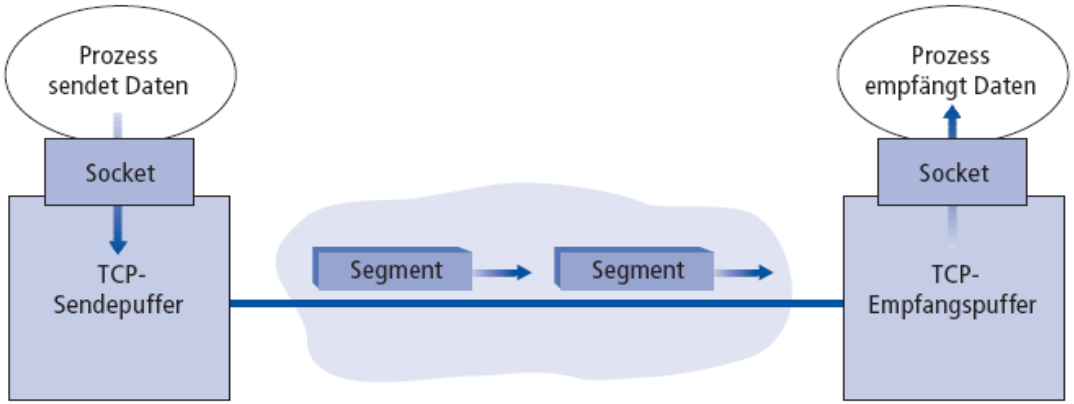


Kapitel 3: Inhalt

- 3.1 Dienste der Tranportschicht
- 3.2 Multiplexing und Demultiplexing
- 3.3 Verbindungsloser Transport: UDP
- 3.4 Grundlagen der zuverlässigen Datenübertragung
- 3.5 Verbindungsorientierter Transport: TCP
  - Segmentstruktur
  - Zuverlässigkeit
  - Flusskontrolle
  - Verbindungsmanagement
- 3.6 Grundlagen der Überlastkontrolle
- 3.7 TCP-Überlast-kontrolle
- 3.8 Socket-Programmierung

TCP: Überblick RFCs: 793, 1122, 1323, 2018, 2581

- Punkt-zu-Punkt:
  - Ein Sender, ein Empfänger
- Zuverlässiger, reihenfolgeerhaltender Byte-Strom:
  - Keine “Nachrichtengrenzen”
- Pipelining:
  - TCP-Überlast- und –Fluss-kontrolle verändern die Größe des Fensters
- *Sender- & Empfängerfenster*



- 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

TCP-Segmentaufbau

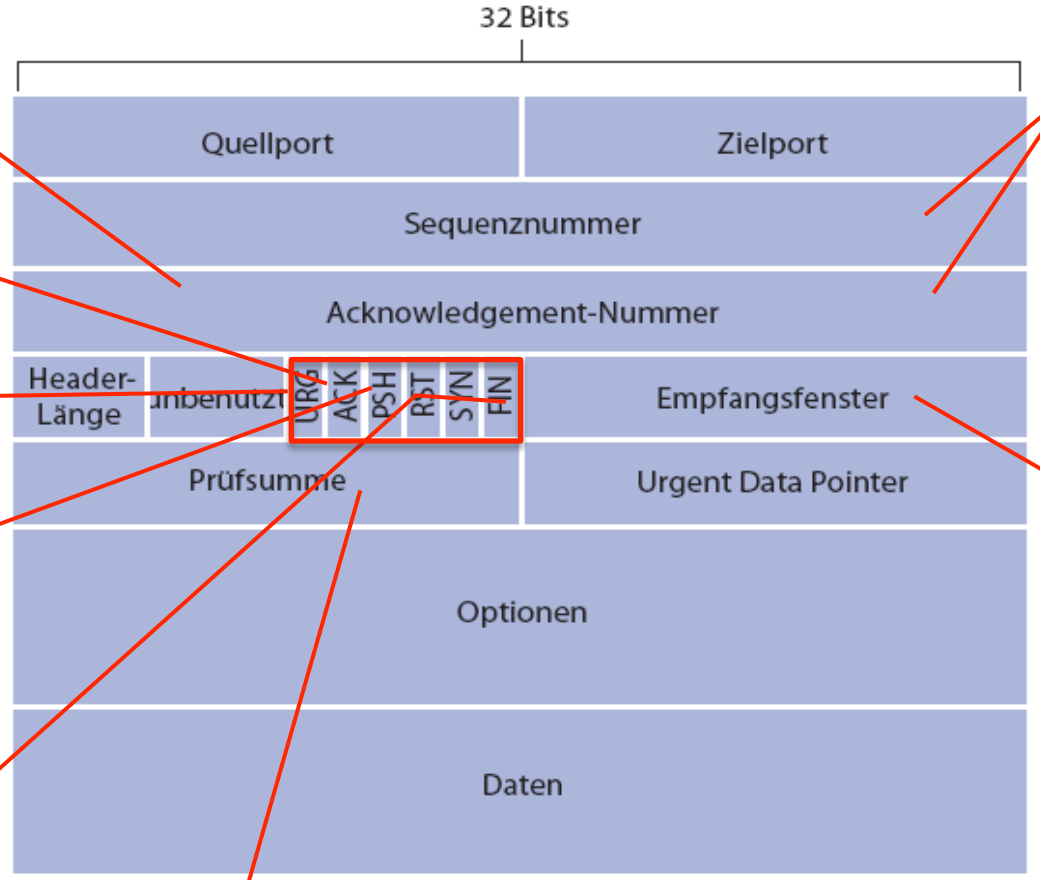
ACK: ACK-Nummer gültig

ACK: Bit um zeigen dass ACK gültig ist

URG: urgent data (üblicherweise nicht benutzt)

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

RST, SYN, FIN: Befehle zum Verbindungsauf- und -abbau



Prüfsumme (wie bei UDP)

Zählen Bytes und nicht Segmente!

Anzahl an Bytes, die der Emfpänger bereit ist, entgegenzu-nehmen

TCP-Sequenznummern und -ACKs

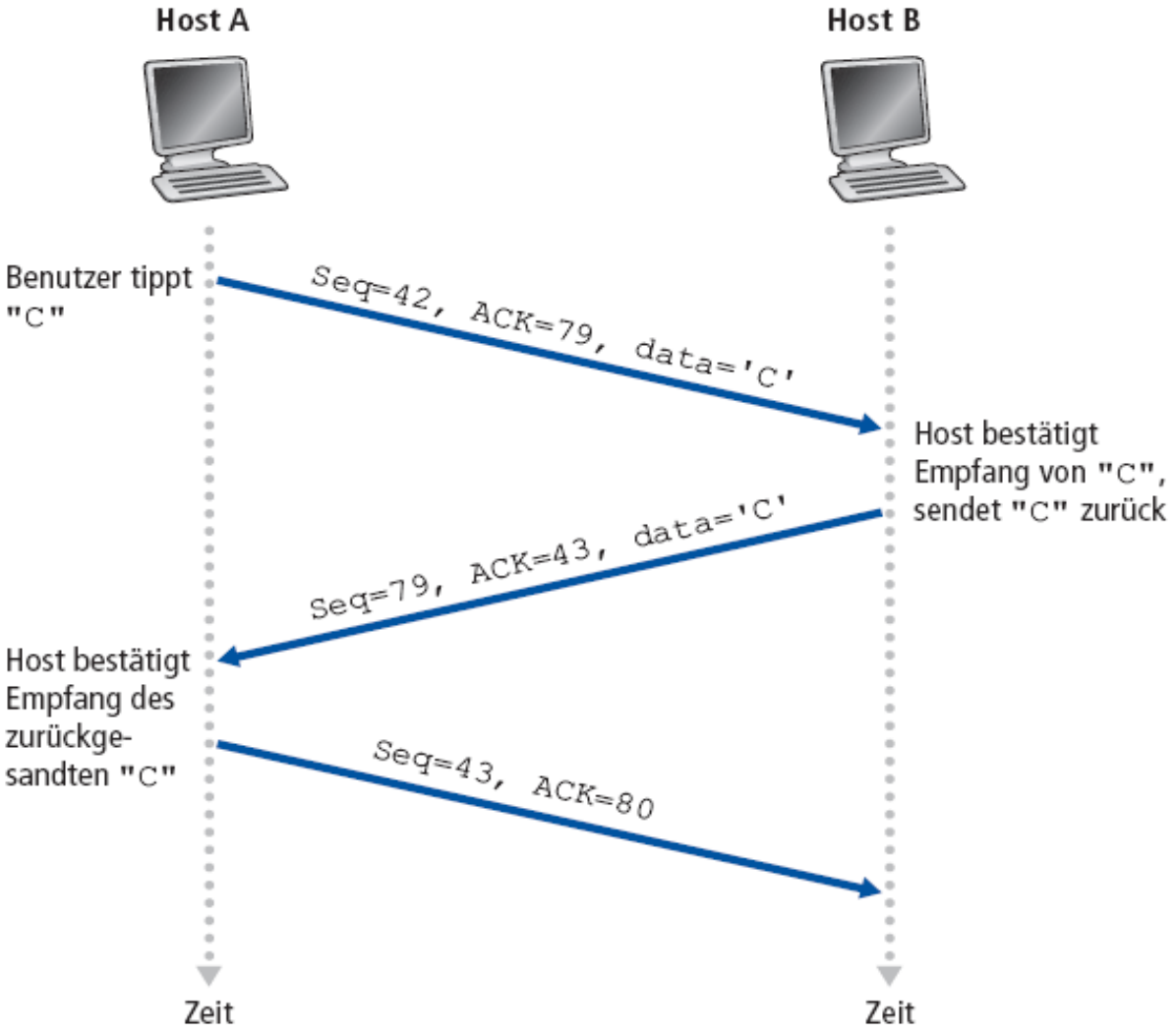
Sequenznummern:

- 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-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
  - Unnötige Übertragungs-wiederholungen
- Zu lang: langsame Reaktion auf den Verlust von Segmenten

Frage: Wie kann man die RTT schätzen?

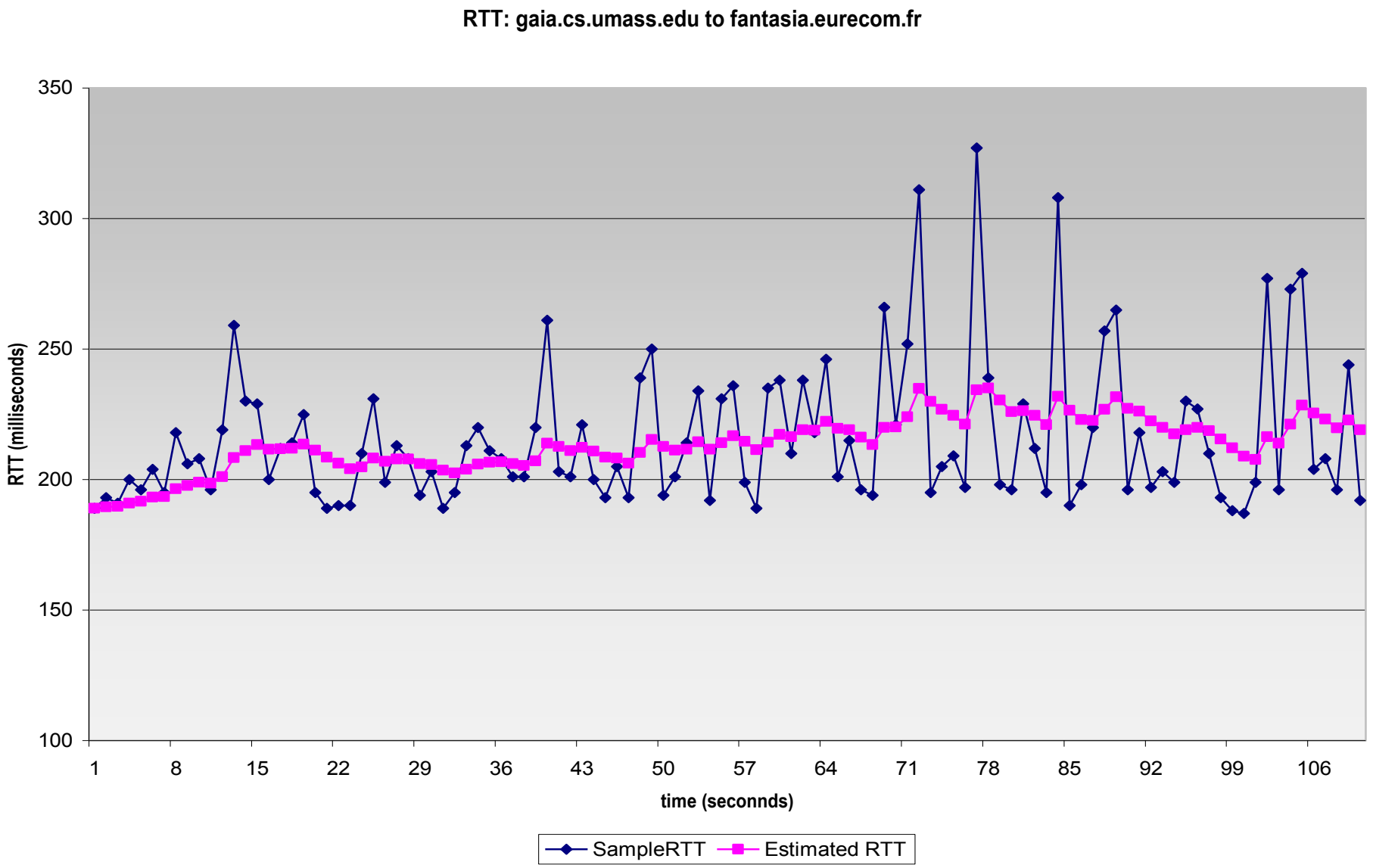
- `SampleRTT`: gemessene Zeit vom Absenden eines Segments bis zum Empfang des dazugehörenden ACKs
  - Segmente mit Übertragungswiederholungen werden ignoriert
- **`SampleRTT`** ist nicht konstant, wir brauchen einen "glatteren" Wert
  - Durchschnitt über mehrere Messungen

TCP-Rundlaufzeit und -Timeout

$$\mathbf{EstimatedRTT = (1- \alpha)*EstimatedRTT + \alpha*SampleRTT}$$

- Exponential weighted moving average
- Einfluss vergangener Messungen verringert sich exponentiell schnell
- Üblicher Wert: $\alpha = 0.125$

Beispiel für die RTT-Bestimmung



TCP-Rundlaufzeit und -Timeout

Bestimmen des Timeout

- **`EstimatedRTT`** plus “Sicherheitsabstand”
  - Größere Schwankungen von **`EstimatedRTT`** -> größerer Sicherheitsabstand
- Bestimme, wie sehr SampleRTT von EstimatedRTT abweicht:

`DevRTT = (1-β)*DevRTT +`
`β*|SampleRTT-EstimatedRTT|`

`(üblicherweise: β = 0.25)`

Dann bestimme den Timeout wie folgt:

`TimeoutInterval = EstimatedRTT + 4*DevRTT`

TCP: zuverlässiger Datentransfer

- TCP = zuverlässigen Datentransfer über den unzuverlässigen IP
- Pipelining von Segmenten
- Kumulative ACKs
- TCP verwendet einen einzigen Timer für Übertragungswieder-holungen

- Übertragungswieder-holungen 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 Sequenz-nummer
  - Nummer des ersten Byte im Datenteil
- Timer starten, wenn er noch nicht läuft
  - Timer für das älteste unbestätigte Segment
- Laufzeit des Timers: `TimeOutInterval`

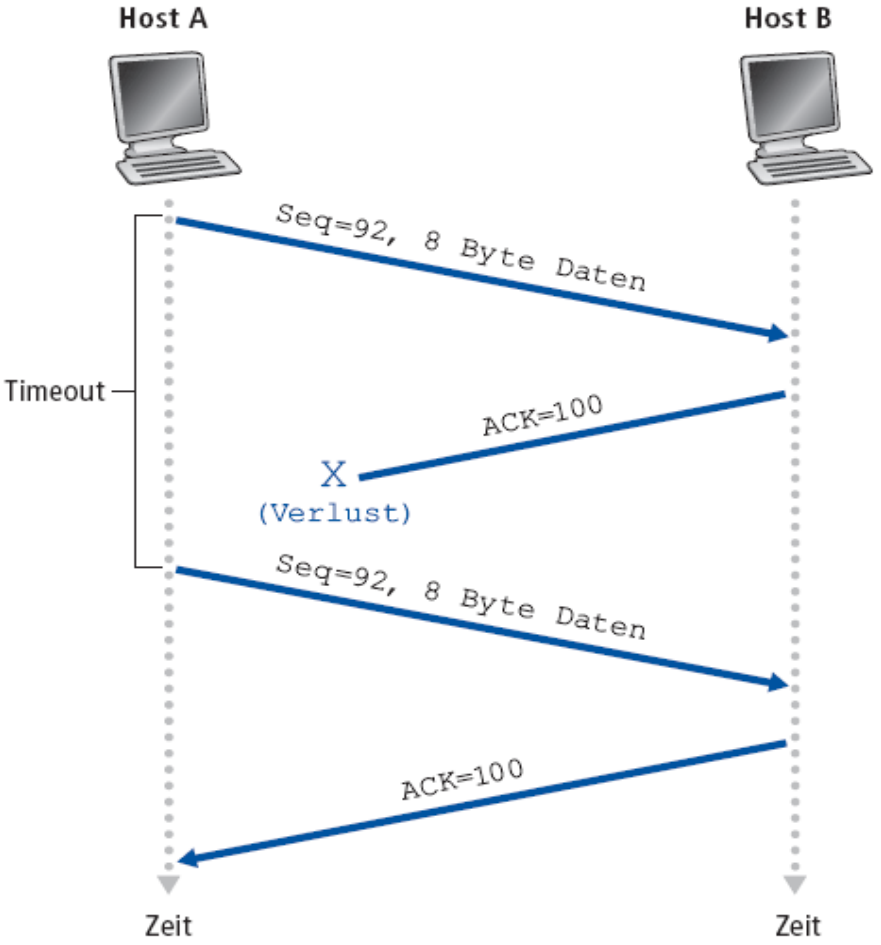
Timeout:

- Erneute Übertragung des Segments, für das der Timeout aufgetreten ist
- Starte Timer neu

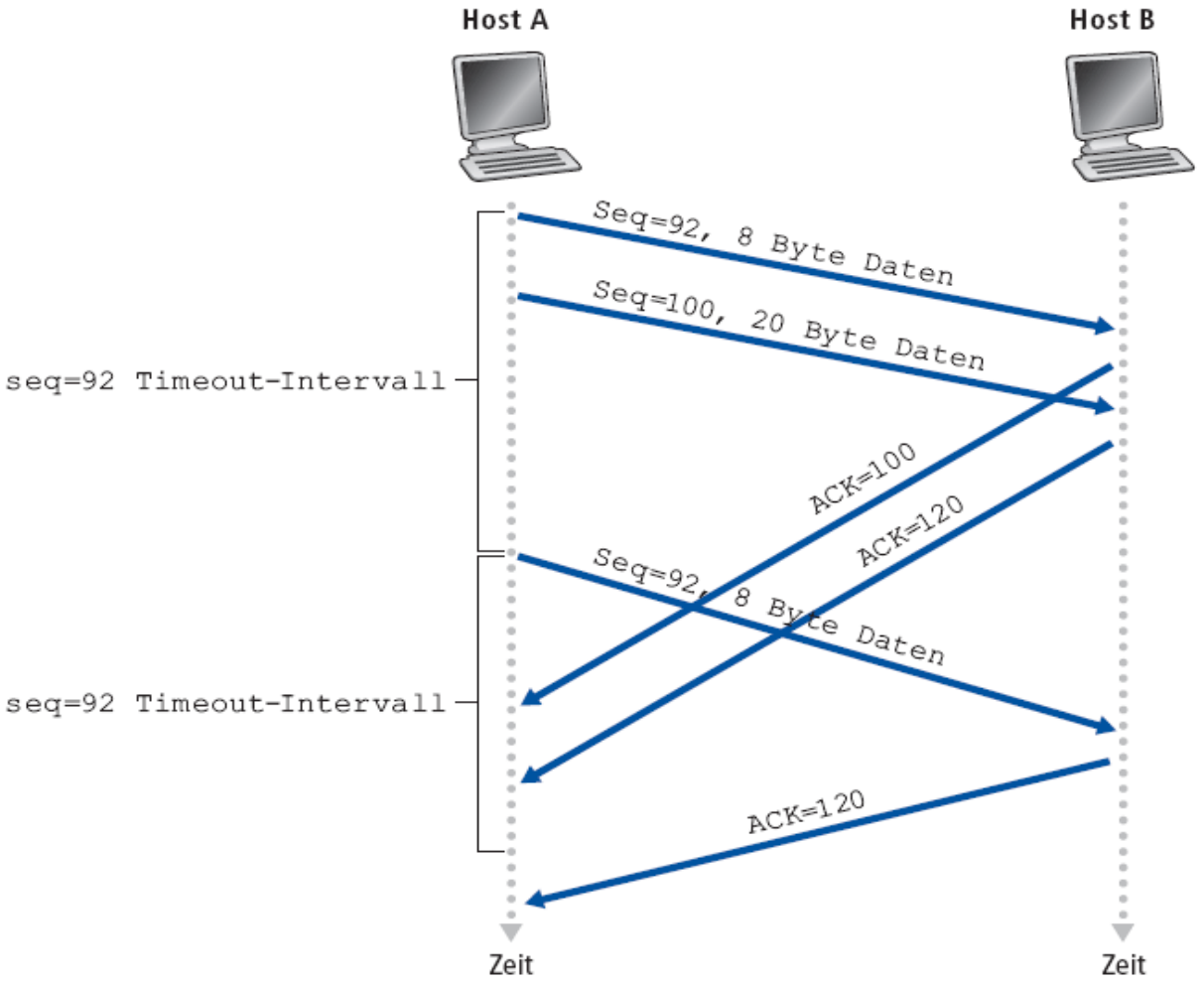
ACK empfangen:

- Wenn damit bisher unbestätigte Daten bestätigt werden:
  - Aktualisiere die Informationen über bestätigte Segmente
  - Starte Timer neu, wenn noch unbestätigte Segmente vorhanden sind

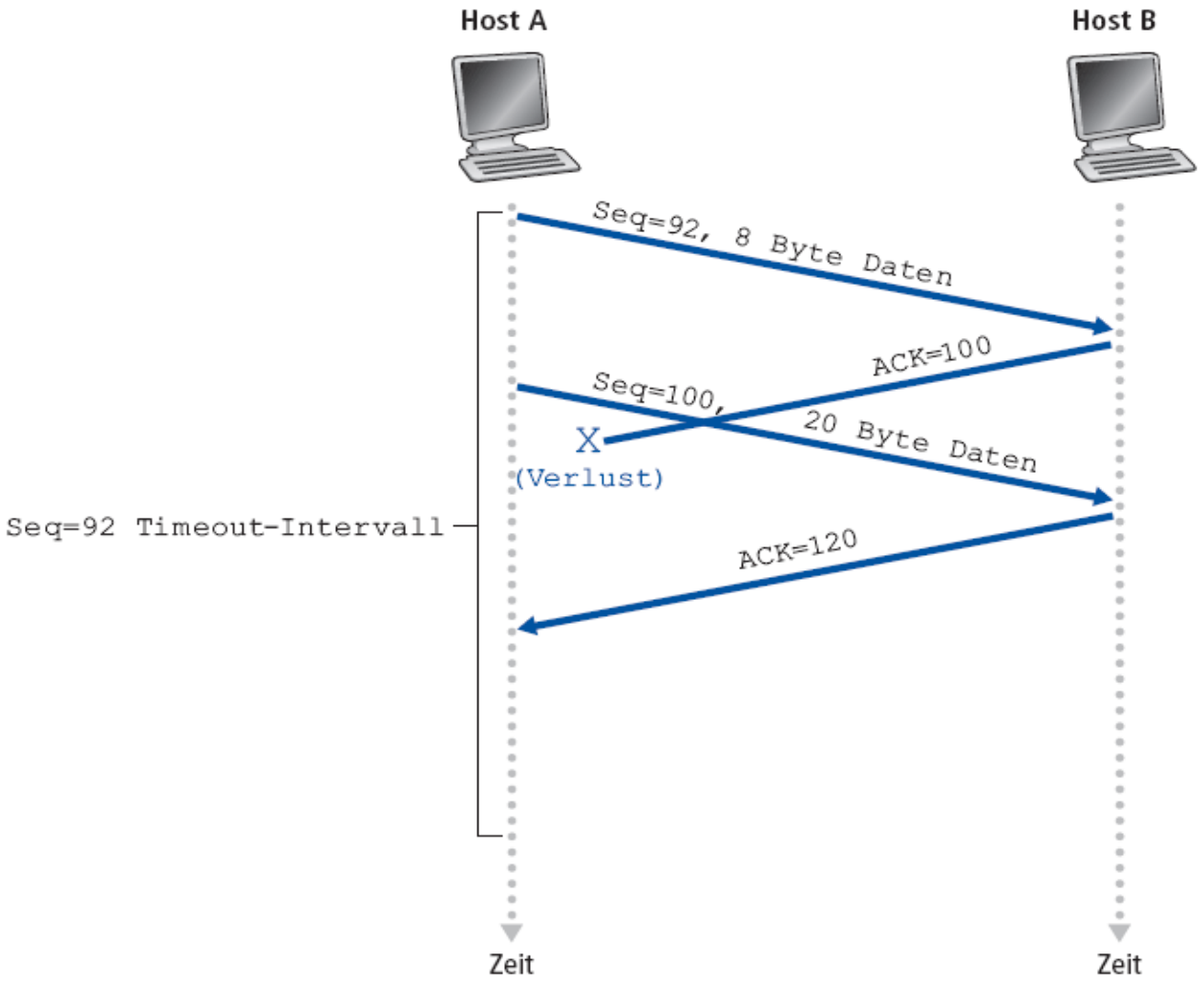
TCP: Beispiele für Übertragungs-wiederholungen – Paketverlust



TCP: Beispiele für Übertragungs-wiederholungen – verfrühter Timeout



TCP: Beispiele für Übertragungs-wiederholungen – kumulative ACKs



TCP-ACK-Erzeugung [RFC 1122, RFC 2581]

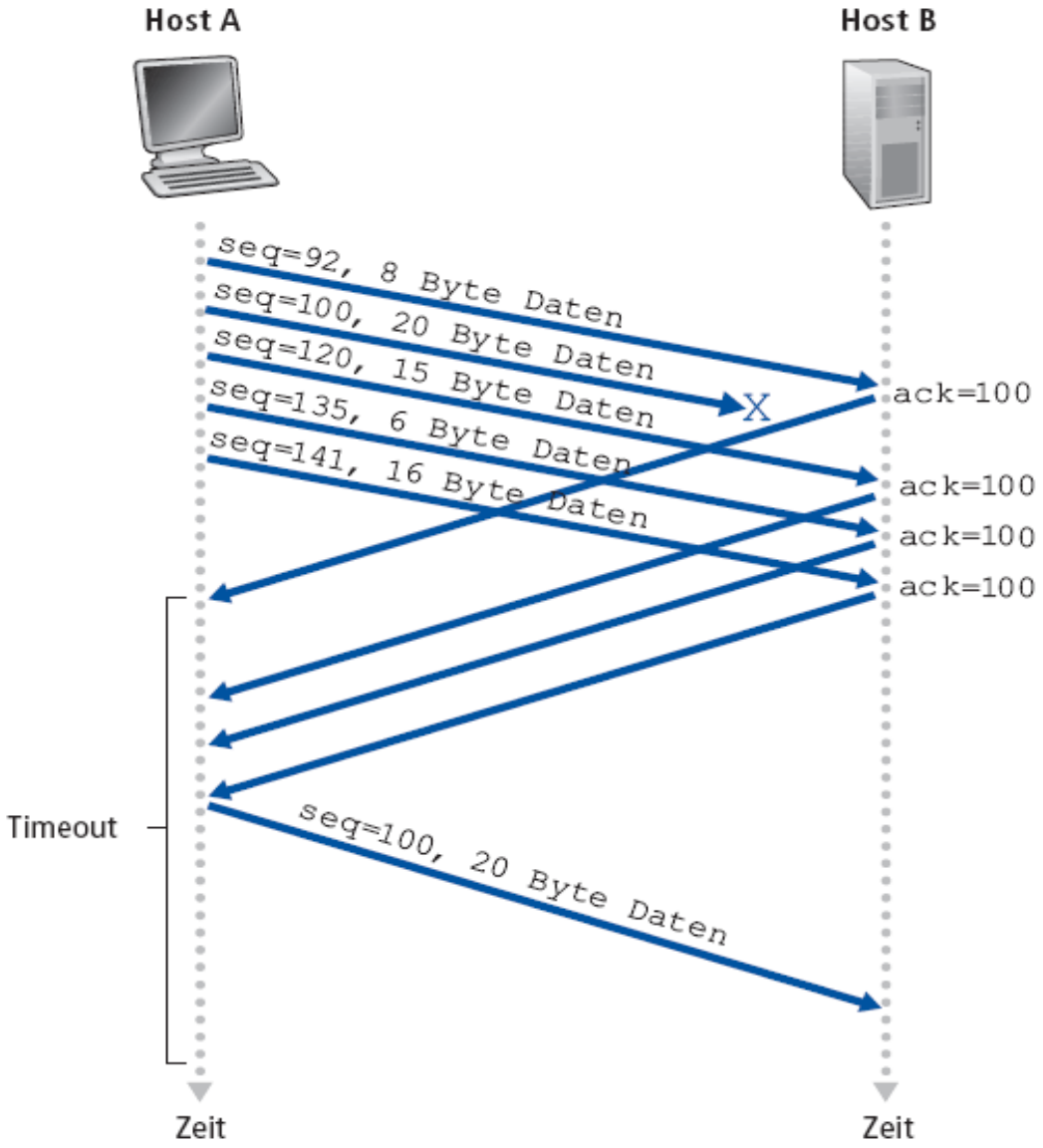
Ereignis	Aktion des TCP-Empfängers
Ankunft des Segmentes in der richtigen Reihenfolge mit der erwarteten Sequenznummer. Alle Daten bis zur erwarteten Sequenznummer sind bereits bestätigt.	Verzögertes ACK. Wartet bis zu 500 ms auf die Ankunft eines anderen Segmentes in richtiger Reihenfolge. Wenn das nächste Segment nicht in diesem Zeitintervall eintrifft, wird ein ACK gesendet.
Ankunft eines Segmentes in der richtigen Reihenfolge mit erwarteter Sequenznummer. Ein anderes Segment in der korrekten Reihenfolge wartet auf die ACK-Übertragung.	Sendet sofort ein einzelnes kumulatives ACK, bestätigt beide in richtiger Reihenfolge eingetroffene Segmente.
Ankunft eines Segmentes außerhalb der Reihenfolge mit einer Sequenznummer, die größer ist als erwartet. Lücke im Bytestrom aufgetreten.	Sendet sofort ein doppeltes ACK, in dem er die Sequenznummer des nächsten erwarteten Bytes angibt.
Ankunft eines Segmentes, das die Lücke in den erhaltenen Daten ganz oder teilweise ausfüllt.	Sendet sofort ein ACK, vorausgesetzt, das Segment beginnt mit der Sequenznummer des nächsten erwarteten Bytes. Bestätigt alle nun lückenlos vorliegenden Bytes.

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

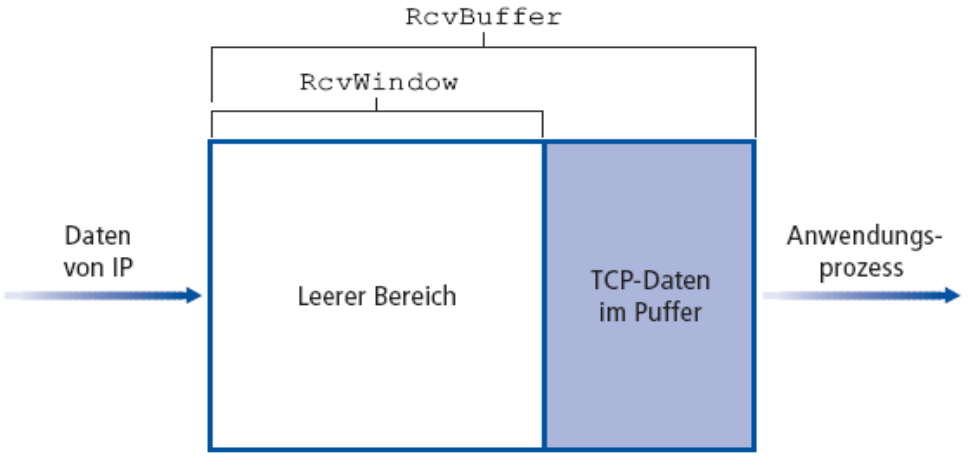
- Wenn der Sender 3 Duplikate eines ACK erhält, dann nimmt er an, dass das Segment verloren gegangen ist:
  - Fast Retransmit (schnelle Sende-wiederholung): Segment erneut schicken, bevor der Timer ausläuft

Fast Retransmit: Beispiel



TCP-Flusskontrolle

- Empfängerseite von TCP hat einen Empfängerpuffer:



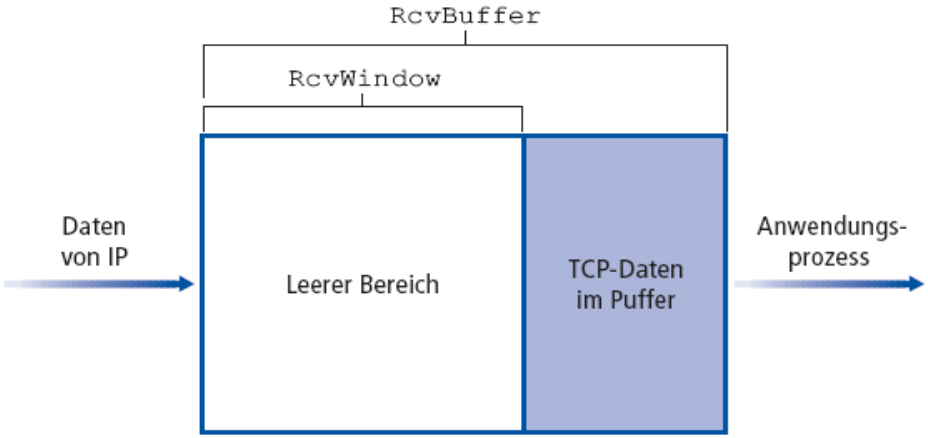
- Die Anwendung kommt unter Umständen nicht mit dem Lesen hinterher

Flusskontrolle

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

- Dienst zum Angleich 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

TCP-Verbindungsmanagement

Erinnerung: 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();
```

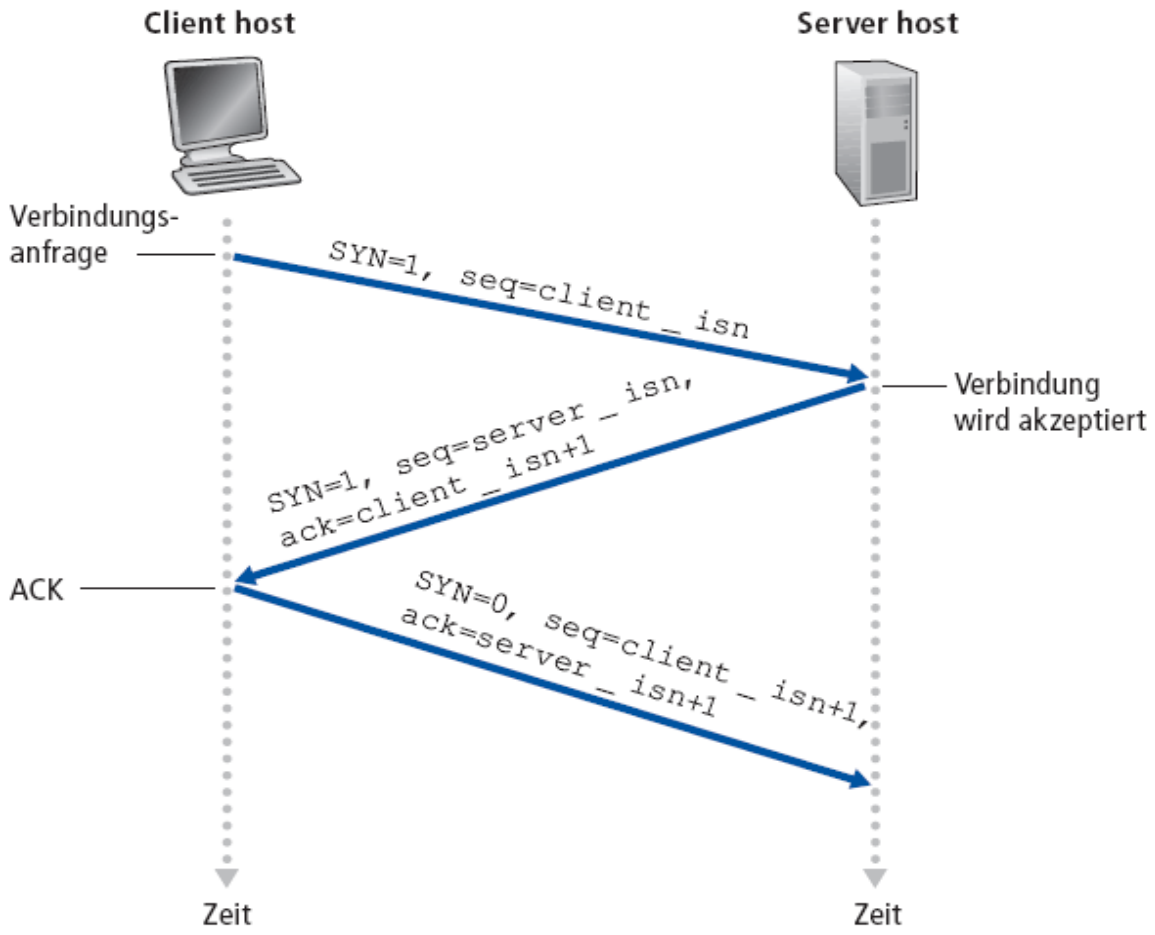
Drei-Wege-Handshake:

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
  - Initale Sequenznummer (Server->Client)

Schritt 3: Client empfängt SYNACK und antwortet mit einem ACK – dieses Segment darf bereits Daten beinhalten

TCP: Drei-Wege-Handshake



TCP-Verbindungsmanagement

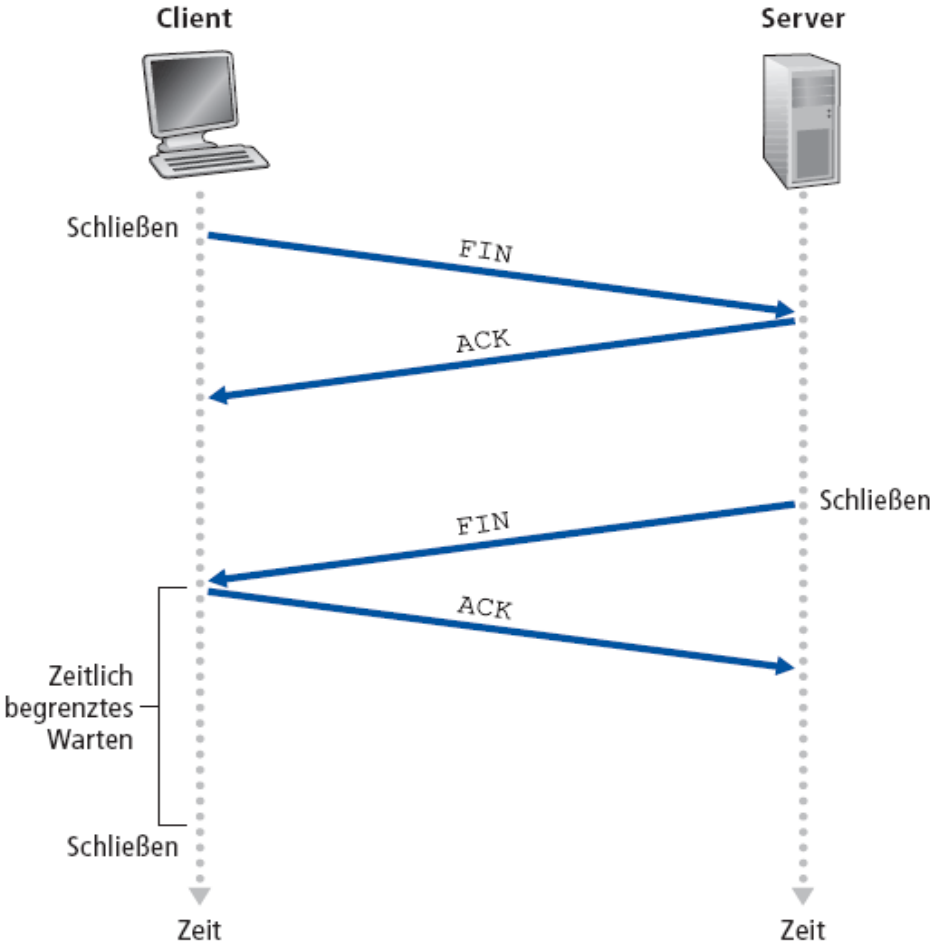
Schließen einer Verbindung

Client schließt socket:

```
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)



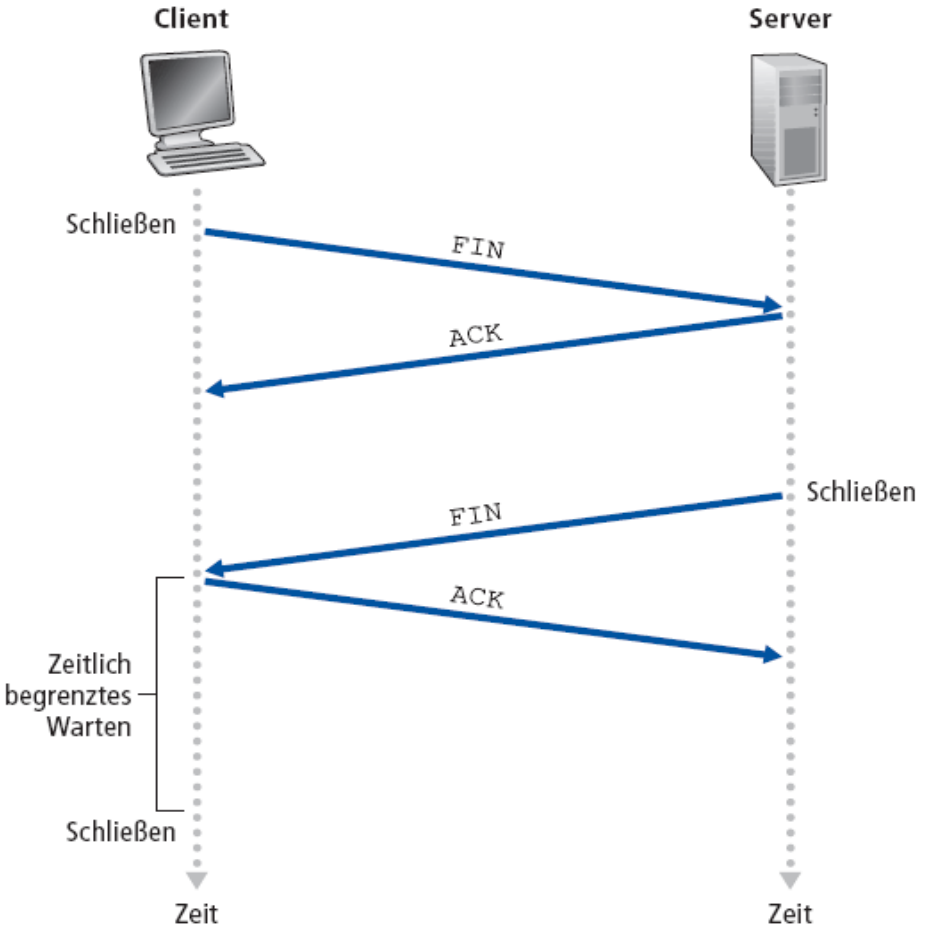
TCP-Verbindungsmanagement

Schritt 3: Client empfängt FIN und antwortet mit ACK

- Beginnt einen “Timed- Wait”-Zustand – er antwortet auf Sende-wiederholungen des Servers mit ACK

Schritt 4: Server, empfängt ACK und schließt Verbindung

Anmerkung: Mit kleinen Änderungen können so auch gleichzeitig abgeschickte FINs behandelt werden



Kapitel 3: Inhalt

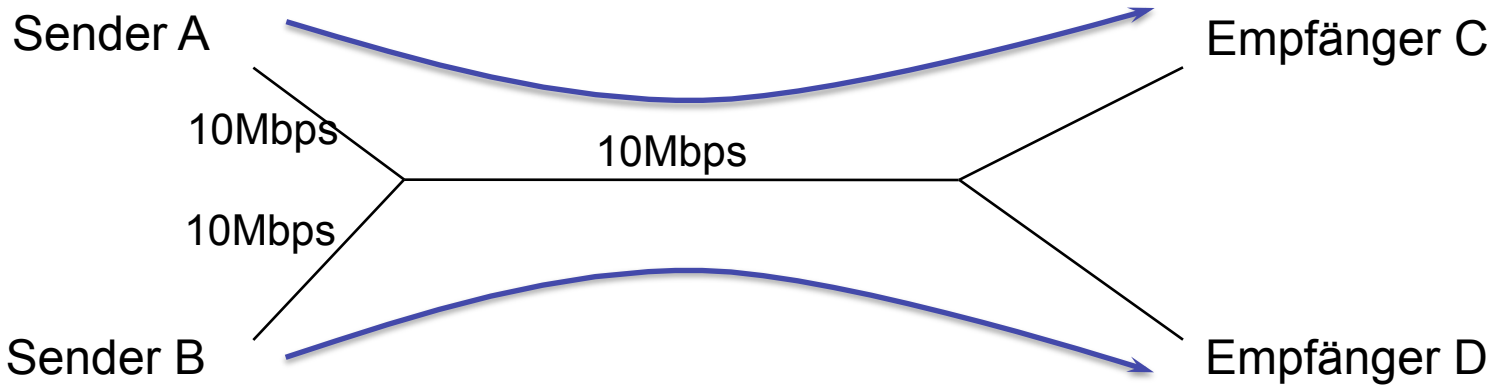
- 3.1 Dienste der Tranportschicht
- 3.2 Multiplexing und Demultiplexing
- 3.3 Verbindungsloser Transport: UDP
- 3.4 Grundlagen der zuverlässigen Datenübertragung
- 3.5 Verbindungsorientierter Transport: TCP
  - Segmentstruktur
  - Zuverlässigkeit
  - Flusskontrolle
  - Verbindungsmanagement
- 3.6 Grundlagen der Überlastkontrolle
- 3.7 TCP-Überlast-kontrolle
- 3.8 Socket-Programmierung

Überlastsproblem

- Problem: Alle Sender senden mit maximalen Durchsatz
- Zustand: bei den Empfängern (Router) die Puffer voll

↓

Netzwerküberlast (congestion)



- Ursache: Eingangskapazitäten sind größer als Ausgangskapazitäten

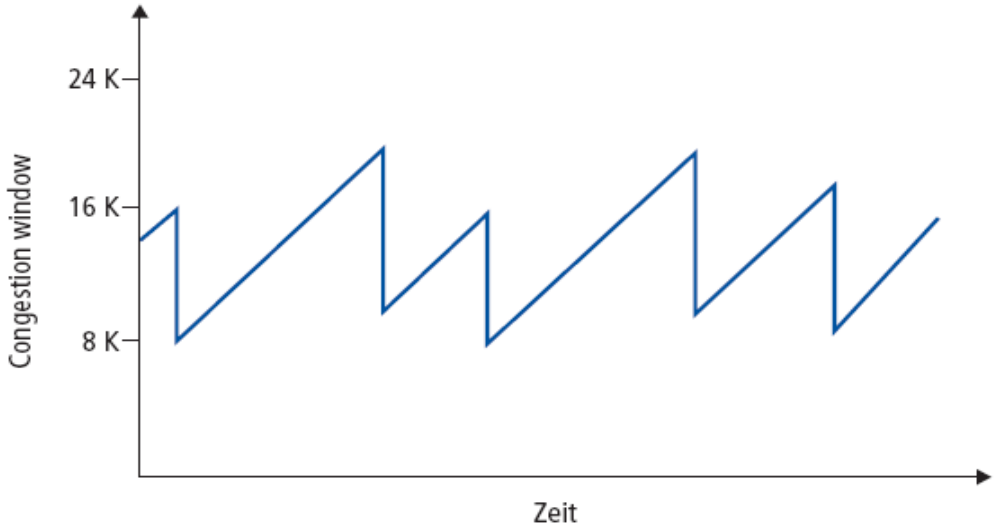
Grundlagen der Überlastkontrolle

- Überlast:
  - Informell: „Zu viele Systeme senden zu viele Daten, das Netzwerk kann nicht mehr alles transportieren“
  - Verschieden von Flusskontrolle!
  - Erkennungsmerkmale:
    - Paketverluste (Pufferüberlauf in den Routern)
    - Lange Verzögerungen (Warten in den Routern)
- Eines der zentralen Probleme in Computernetzwerken!

TCP-Überlastkontrolle: Additive Increase, Multiplicative Decrease

- *Ansatz:* Erhöhe die Übertragungsrate (Fenstergröße), um nach überschüssiger Bandbreite zu suchen, bis ein Verlust eintritt
  - *Additive Increase:* Erhöhe **CongWin** um eine MSS pro RTT, bis ein Verlust erkannt wird
  - *Multiplicative Decrease*: Halbiere **CongWin,** wenn ein Verlust erkannt wird

Sägezahnverlauf: nach überschüssiger Bandbreite suchen



TCP-Überlastkontrolle: Details

- Sender begrenzt die Übertragung:

  `LastByteSent-LastByteAcked`

  `≤ CongWin`

- Allgemein gilt:

$$\text{Rate} = \frac{\text{CongWin}}{\text{RTT}} \quad \text{Byte/s}$$

- **`CongWin`** ist dynamisch, hängt von der wahrgenommenen Netzwerklast ab

Wie erkennt der Sender Überlast?

- Verlustereignis = Timeout *oder* drei doppelte ACKs
- TCP: Sender verringert seine Rate (**`CongWin`**) nach einem Verlustereignis
- Drei Mechanismen:
  - AIMD
  - Slow Start
  - Vorsichtiges Verhalten nach einem Timeout

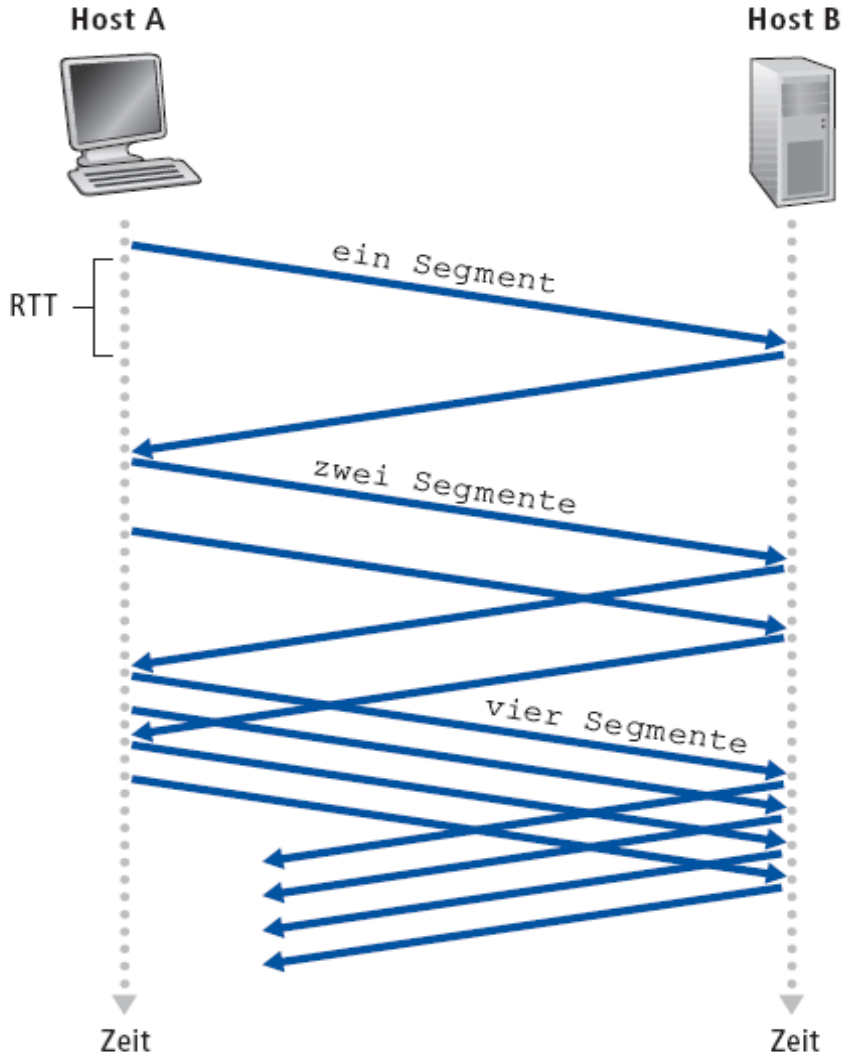
TCP: Slow Start

- Bei Verbindungsbeginn: **`CongWin`** = 1 MSS
  - Beispiel: MSS = 500 Byte & RTT = 200 ms
  - Initiale Rate = 20 kBit/s
- Verfügbare Bandbreite kann viel größer als MSS/RTT sein
  - Die Rate sollte sich schnell der verfügbaren Rate anpassen

- Bei Verbindungsbeginn: Erhöhe die Rate exponentiell schnell, bis es zum ersten Verlust-ereignis kommt

TCP: Slow Start

- Bei Verbindungsbeginn: Erhöhe die Rate exponentiell schnell bis zum ersten Verlust-ereignis
  - Verdoppeln von **`CongWin`** in jeder RTT
  - Realisiert: **`CongWin`** um 1 für jedes erhaltene ACK erhöhen
- Ergebnis: Initiale Rate ist gering, wächst aber exponentiell schnell

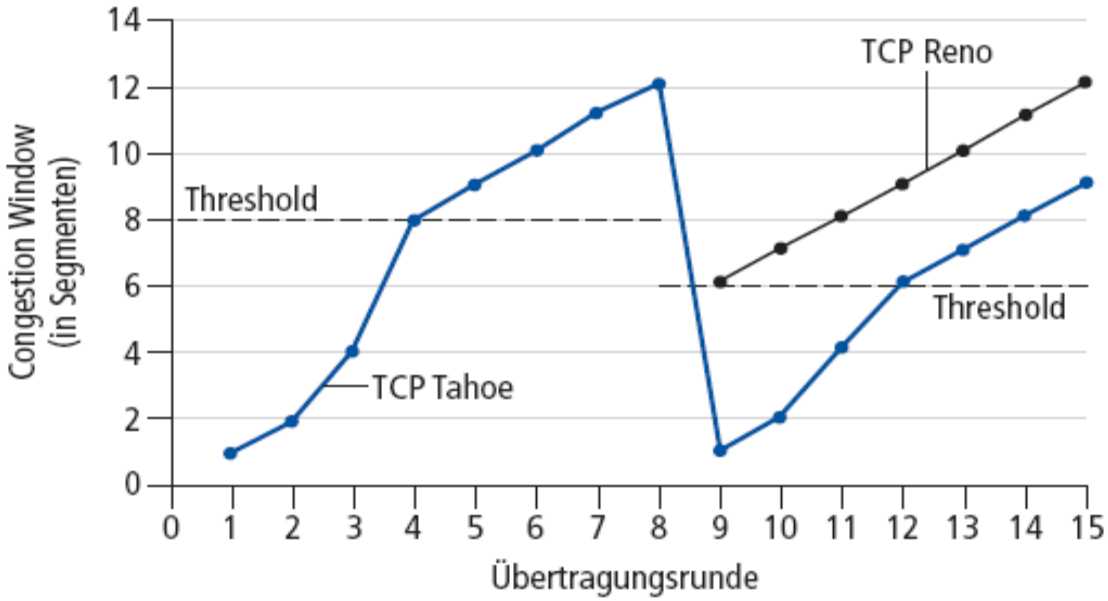


Details

Frage: Wann soll vom exponentiellen Wachstum zum linearen Wachstum übergegangen werden?

Antwort: Wenn **`CongWin`** die Hälfte des Wertes vor dem letzten Verlustereignis erreicht hat

.



Implementierung:

- Variabler `Threshold`
- Bei einem Verlustereignis wird `Threshold` auf die Hälfte von CongWin vor dem Verlustereignis gesetzt

Details: Verluste erkennen

- Fast recovery (nur Reno) – Nach 3 ACK Duplikaten
  - **`CongWin`** halbieren, `Threshold` auf diesen Wert setzen
  - Fenster wächst dann linear
- Nach Timeout (und 3 Ack Duplikaten in Tahoe)
  - **`CongWin`** auf eine MSS setzen, `Threshold` auf die Hälfte des alten CongWin setzen
  - Fenster wächst dann exponentiell, bis `Threshold` erreicht ist
  - … danach linear

Philosophie:

- Drei doppelte ACKs zeigen an, dass das Netzwerk in der Lage ist, Pakete auszuliefern
- Timeout ist “schlimmer”, weil gar keine Rückmeldung mehr erfolgt

Zusammenfassung: TCP-Überlastkontrolle

- Wenn **`CongWin`** kleiner als `Threshold` ist, befindet sich der Sender in der Slow-Start-Phase, das Fenster wächst exponentiell.
- Wenn **`CongWin`** größer als `Threshold` ist, befindet sich der Sender in der Congestion-Avoidance-Phase, das Fenster wächst linear.
- Wenn ein Triple Duplicate ACK auftritt (drei ~~doppelte~~ ACKs Duplikaten für dasselbe Segment), wird `Threshold` auf **`CongWin/2`** gesetzt und dann **`CongWin`** auf ssthresh.
- Wenn ein Timeout auftritt, werden `Threshold` auf **`CongWin/2`** und **`CongWin`** auf eine 1 MSS gesetzt.

TCP-Überlastkontrolle im Sender (TCP Reno)

Zustand	Ereignis	Reaktion der TCP-Überlastkontrolle	Kommentar
Slow Start (SS)	ACK für zuvor unbestätigte Daten empfangen	*CongWin* = *CongWin* + *MSS*, Wenn (*CongWin* > Threshold), setze Zustand auf „Congestion Avoidance"	Führt zu einer Verdopplung von *CongWin* in jeder *RTT*.
Congestion Avoidance (CA)	ACK für zuvor unbestätigte Daten empfangen	*CongWin* = *CongWin* + *MSS* · (*MSS*/*CongWin*)	Additive Increase, resultiert in einer Zunahme von *CongWin* um 1 *MSS* jede *RTT*.
SS oder CA	Verlustereignis entdeckt durch drei doppelte ACKs	Threshold = *CongWin*/2, *CongWin* = Threshold, setze Zustand auf „Congestion Avoidance"	Fast Recovery, implementiert Multiplicative Decrease. *CongWin* kann nicht unter 1 *MSS* fallen.
SS oder CA	Timeout	Threshold = *CongWin*/2, *CongWin* = 1 *MSS*, setze Zustand auf „Slow Start"	Erneute Slow-Start-Phase
SS oder CA	Doppeltes ACK empfangen	Erhöhe den Zähler für doppelte ACKs für das bestätigte Segment	*CongWin* und Threshold werden nicht verändert.

TCP-Durchsatz

- Was ist der durchschnittliche TCP-Durchsatz bei gegebener Fenstergröße und RTT?
  - Vernachlässigen von Slow Start
- Sei W die Fenstergröße, wenn das Verlustereignis eintritt
- Wenn das Fenster W groß ist, dann ist der Durchsatz W/RTT
- Direkt nach dem Verlust verringert sich das Fenster auf W/2 und damit der Durchsatz auf W/2RTT
- Durchschnittlicher Durchsatz: 0.75 W/RTT

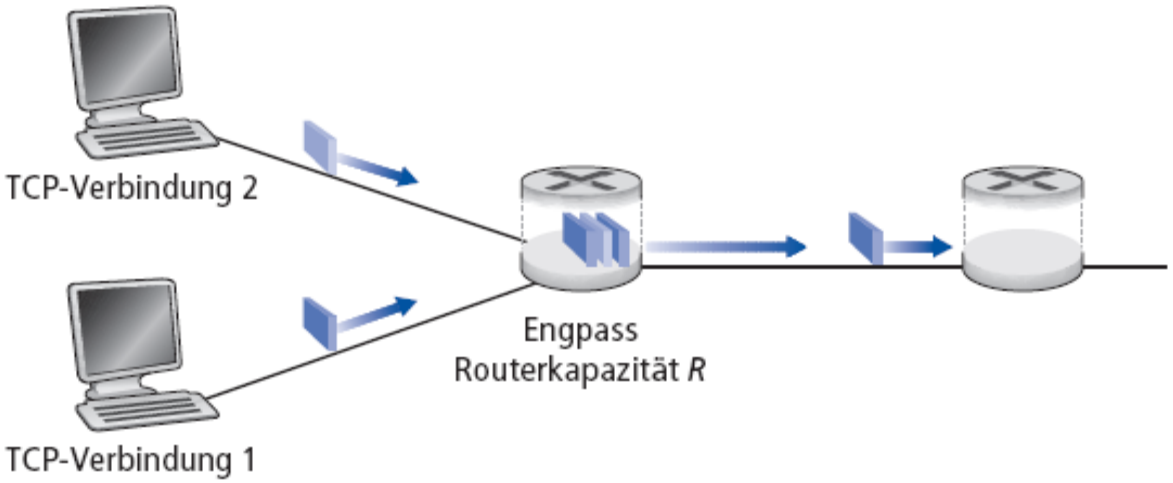
TCP Varianten

- TCP Tahoe: einfach, fangt immer mit 2*MSS an beim Paketen verlust
- TCP Reno: halbiert die **CongWin** beim Paketenverlust
- TCP Vegas: RTT gemessen fur jedes Paket im Puffer (nicht nur letztes)
- TCP Hybla: Satellitcommunication (höhes RTT)
- TCP BIC und CUBIC: verbesserte Versionen für Netzwerke mit höher Kapazität und Verzögerung (long fat networks)

QUIC

TCP-Fairness

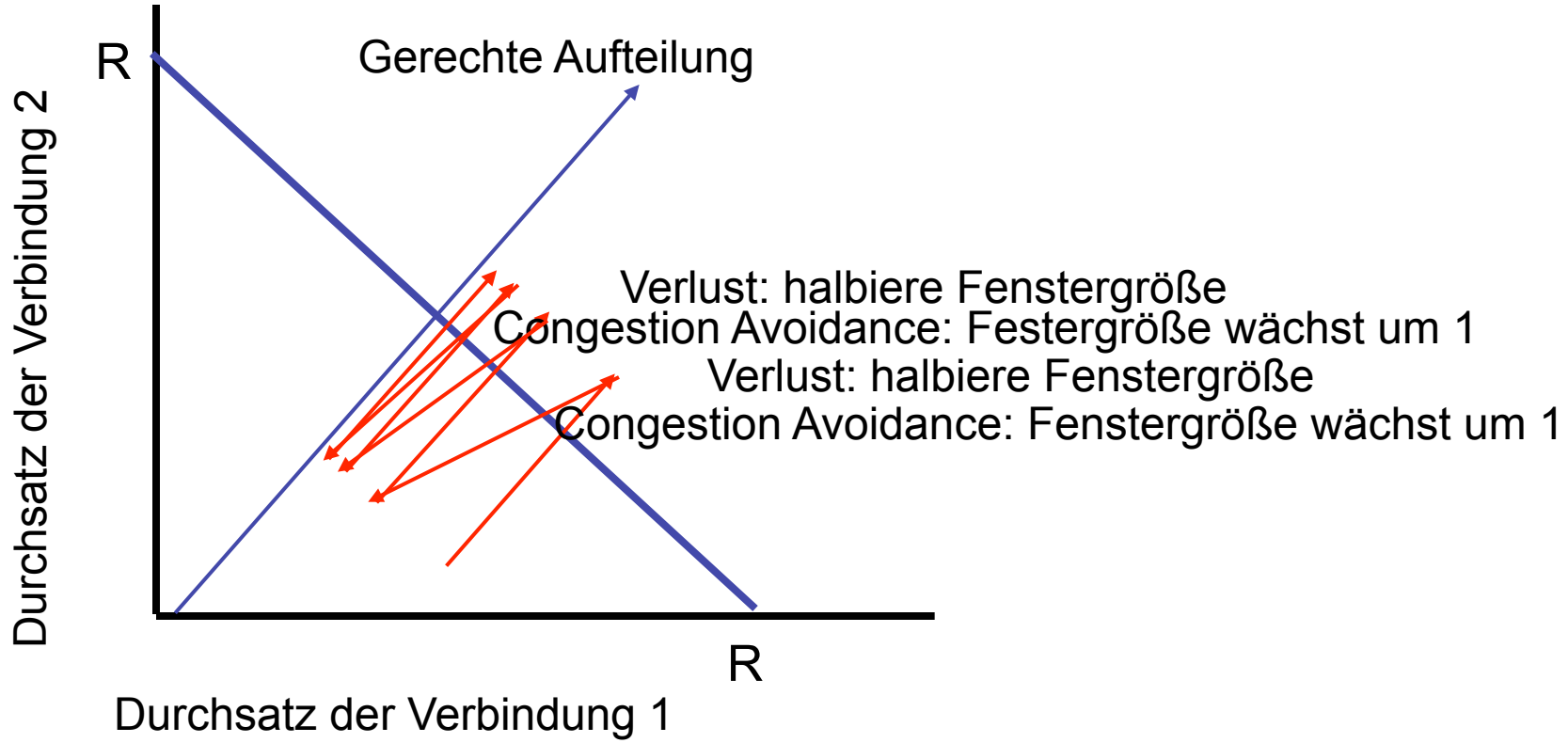
Ziel: Wenn K TCP-Sitzungen sich denselben Engpass mit Bandbreite R teilen, dann sollte jede eine durchschnittliche Rate von R/K erhalten



Warum ist TCP fair?

Zwei Verbindungen im Wettbewerb:

- Additive Increase führt zu einer Steigung von 1, wenn der Durchsatz wächst
- Multiplicative Decrease reduziert den Durchsatz proportional



Mehr zum Thema Fairness

Fairness und UDP

- Viele Multimedia-anwendungen verwenden kein TCP
  - Wollen nicht, dass die Rate durch Überlast-kontrolle reduziert wird
- Stattdessen: Einsatz von UDP
  - Audio-/Videodaten mit konstanter Rate ins Netz leiten, Verlust hinnehmen
- Forschungsgebiet: TCP-Friendliness

Fairness und drahtlose Netze

- Problem mit fernen Terminale

Fairness und parallele TCP-Verbindungen:

- Eine Anwendung kann zwei oder mehr parallele TCP-Verbindun-gen öffnen
- Webbrowser machen dies häufig
- Beispiel: Engpass hat eine Rate von R, bisher existieren neun Verbindungen
  - Neue Anwendung legt eine neue TCP-Verbindung an und erhält die Rate R/10
  - Neue Anwendung legt elf neue TCP-Verbindungen an und erhält mehr als R/2!

Akella, A. and Seshan, S. and Karp, R. and Shenker, S. and Papadimitriou, C. "*Selfish behavior and stability of the Internet: a game-theoretic analysis of TCP*", SIGCOMM 2002

Kapitel 3: Inhalt

- 3.1 Dienste der Tranportschicht
- 3.2 Multiplexing und Demultiplexing
- 3.3 Verbindungsloser Transport: UDP
- 3.4 Grundlagen der zuverlässigen Datenübertragung
- 3.5 Verbindungsorientierter Transport: TCP
  - Segmentstruktur
  - Zuverlässigkeit
  - Flusskontrolle
  - Verbindungsmanagement
- 3.6 Grundlagen der Überlastkontrolle
- 3.7 TCP-Überlast-kontrolle
- 3.8 Socket-Programmierung

Socket-Programmierung

Ziel: lernen, wie man eine Client/Server-Anwendung programmiert, die über Sockets kommuniziert

Socket-API

- Eingeführt in BSD4.1 UNIX, 1981
- Sockets werden von Anwendungen erzeugt, verwendet und geschlossen
- Client/Server-Paradigma
- Zwei Transportdienste werden über die Socket-API angesprochen:
  - Unzuverlässige Paketübertragung
  - Zuverlässige Übertragung von Datenströmen

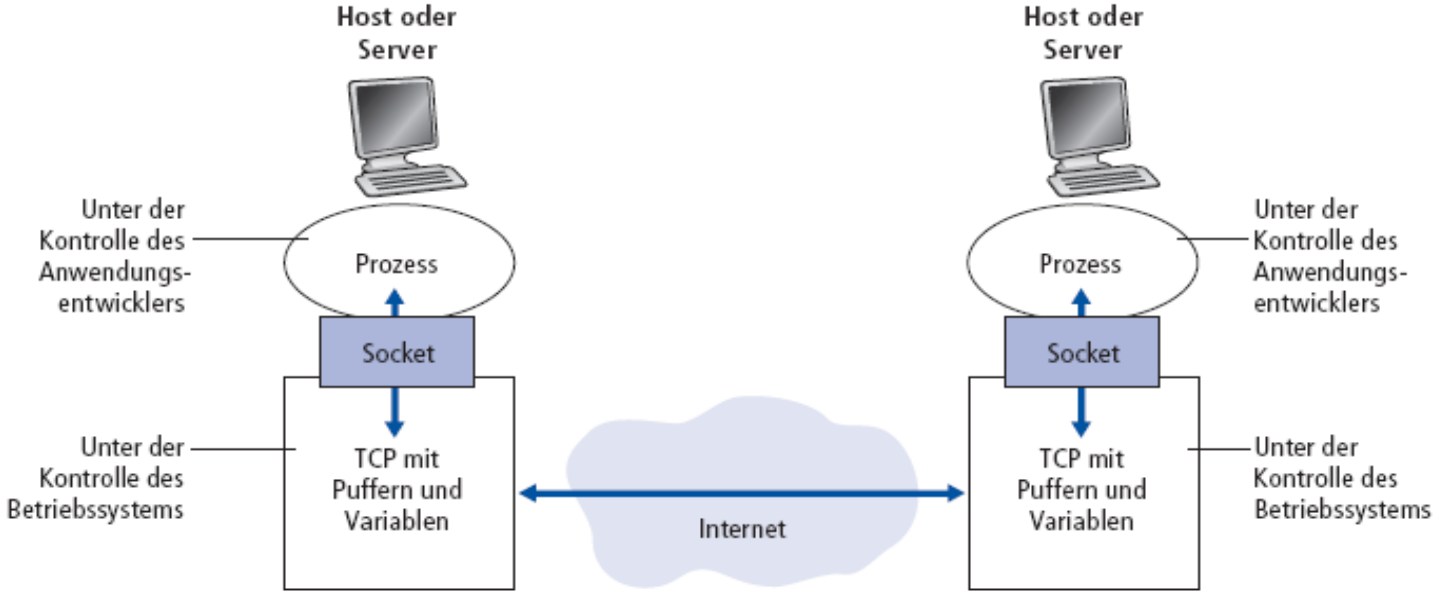
Socket

Eine Schnittstelle auf einem Host, kontrolliert durch das Betriebssystem, über das ein Anwendungsprozess sowohl Daten an einen anderen Prozess senden als auch von einem anderen Prozess empfangen kann.
Ein Socket ist eine Art “Tür” zum Computernetzwerk.

Socket-Programmierung mit TCP

Socket: eine „Tür“ zwischen dem Anwendungsprozess und dem Transportprotokoll (UDP oder TCP)

Wichtigster TCP-Dienst (aus Sicht der Anwendung): Zuverlässiger Transport eines Byte-Stroms vom sendenden zum empfangenden Prozess



Socket-Programmierung mit TCP

Client kontaktiert Server

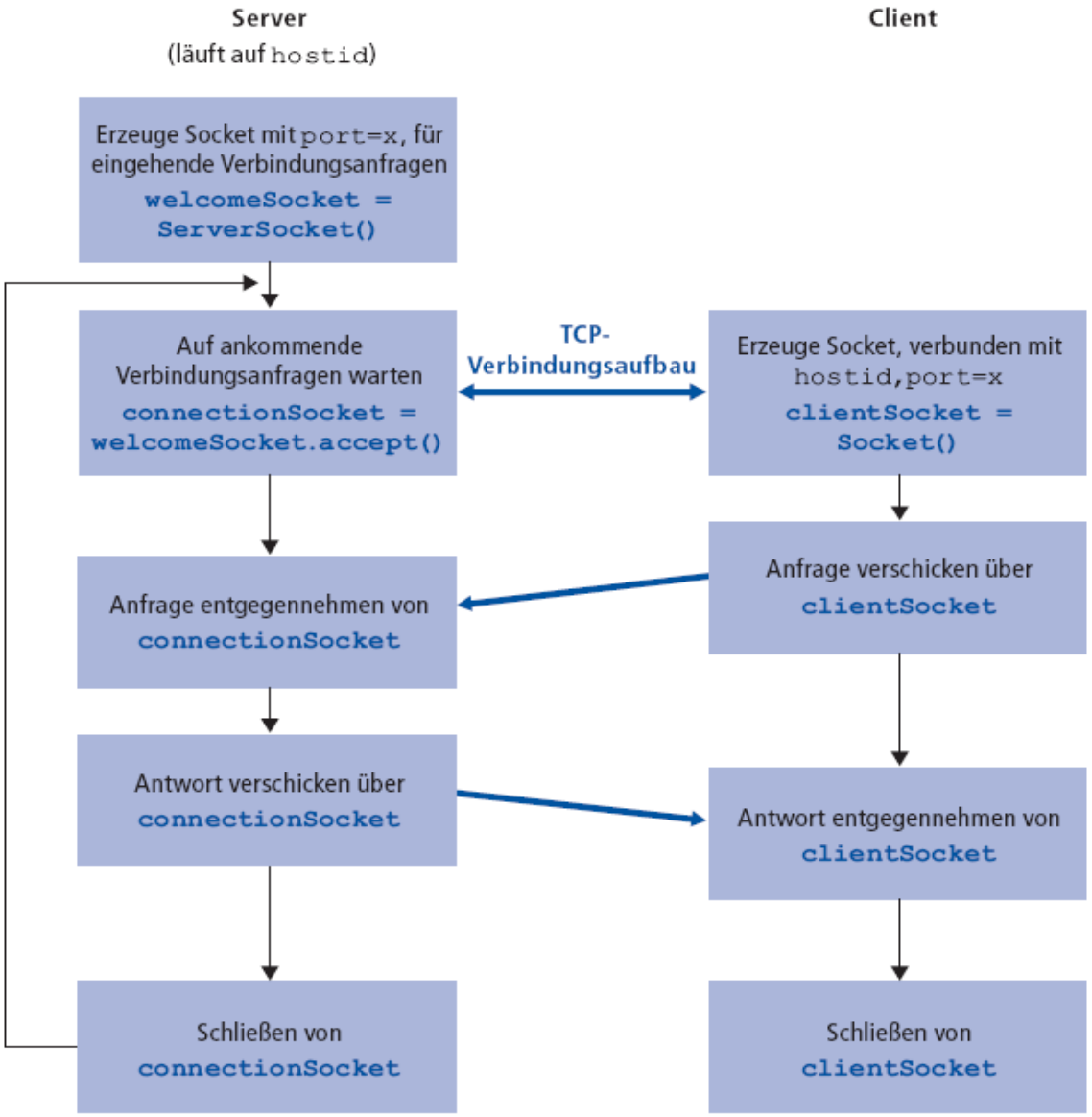
- Server-Prozess muss laufen
- Server muss einen Socket (eine Tür) angelegt haben, der Client-Anfragen entgegennimmt

Vorgehen im Client:

- Anlegen eines Client-TCP-Sockets
- Angeben von IP-Adresse und Portnummer des Server-prozesses
- Durch das Anlegen eines Client-TCP-Sockets wird eine TCP-Verbindung zum Server-prozess hergestellt

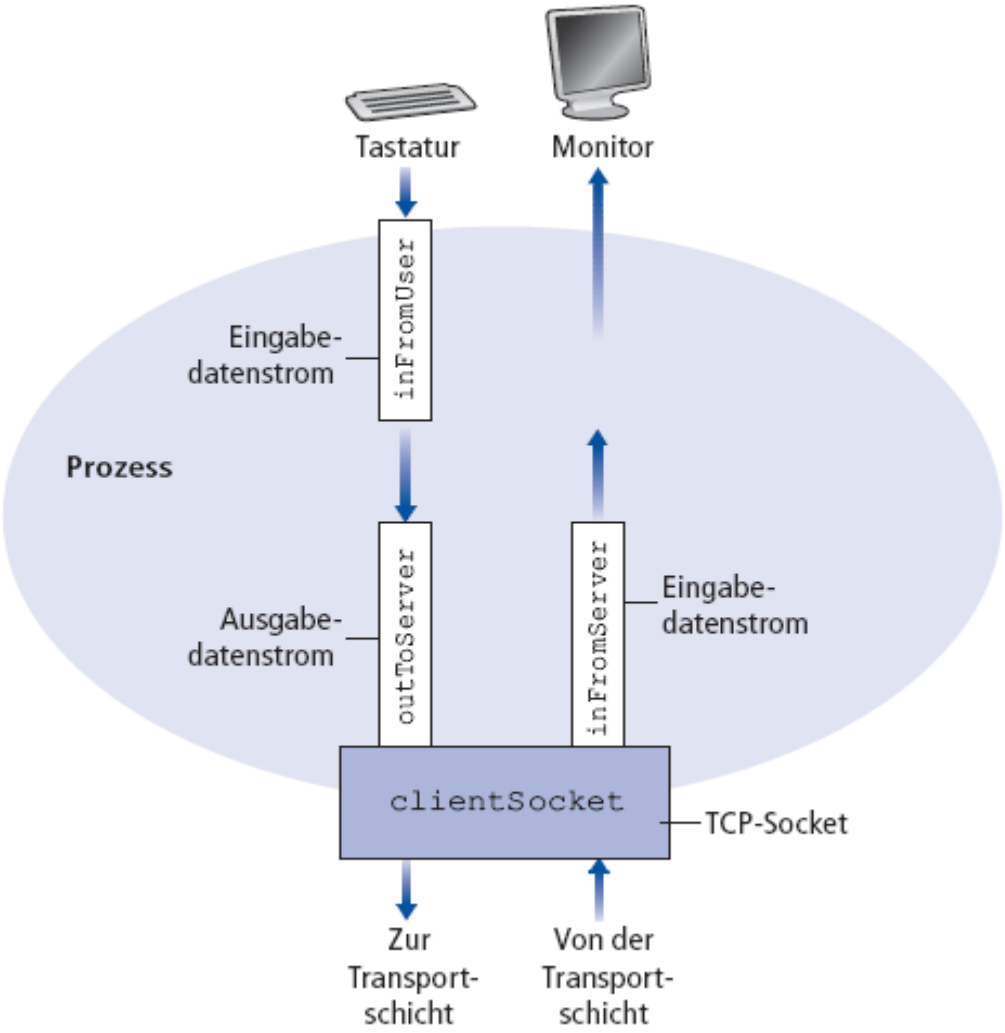
- Wenn der Serverprozess von einem Client kontaktiert wird, dann erzeugt er einen neuen Socket, um mit diesem Client zu kommunizieren
  - So kann der Server mit mehreren Clients kommunizieren
  - Portnummern der Clients werden verwendet, um die Verbindungen zu unterscheiden

Client/Server-Socket-Programmierung: TCP



(Daten-)Ströme

- Ein Strom ist eine Folge von Bytes, die in einen Prozess hinein- oder aus ihm hinausfließen
- Ein Eingabestrom ist mit einer Quelle verbunden, z.B. Tastatur oder Socket
- Ein Ausgabestrom ist mit einer Senke verbunden, z.B. dem Monitor oder einem Socket



Socket-Programmierung mit TCP

Beispiel für eine Client/Server-Anwendung:

1) Client liest Zeilen von der Standardeingabe (**inFromUser** Strom) und sendet diese über einen Socket (**outToServer** Strom) zum Server
2) Server liest die Zeile vom Socket
3) Server konvertiert die Zeile in Großbuchstaben und sendet sie zum Client zurück
4) Client liest die konvertierte Zeile vom Socket (**inFromServer** Strom) und gibt sie aus

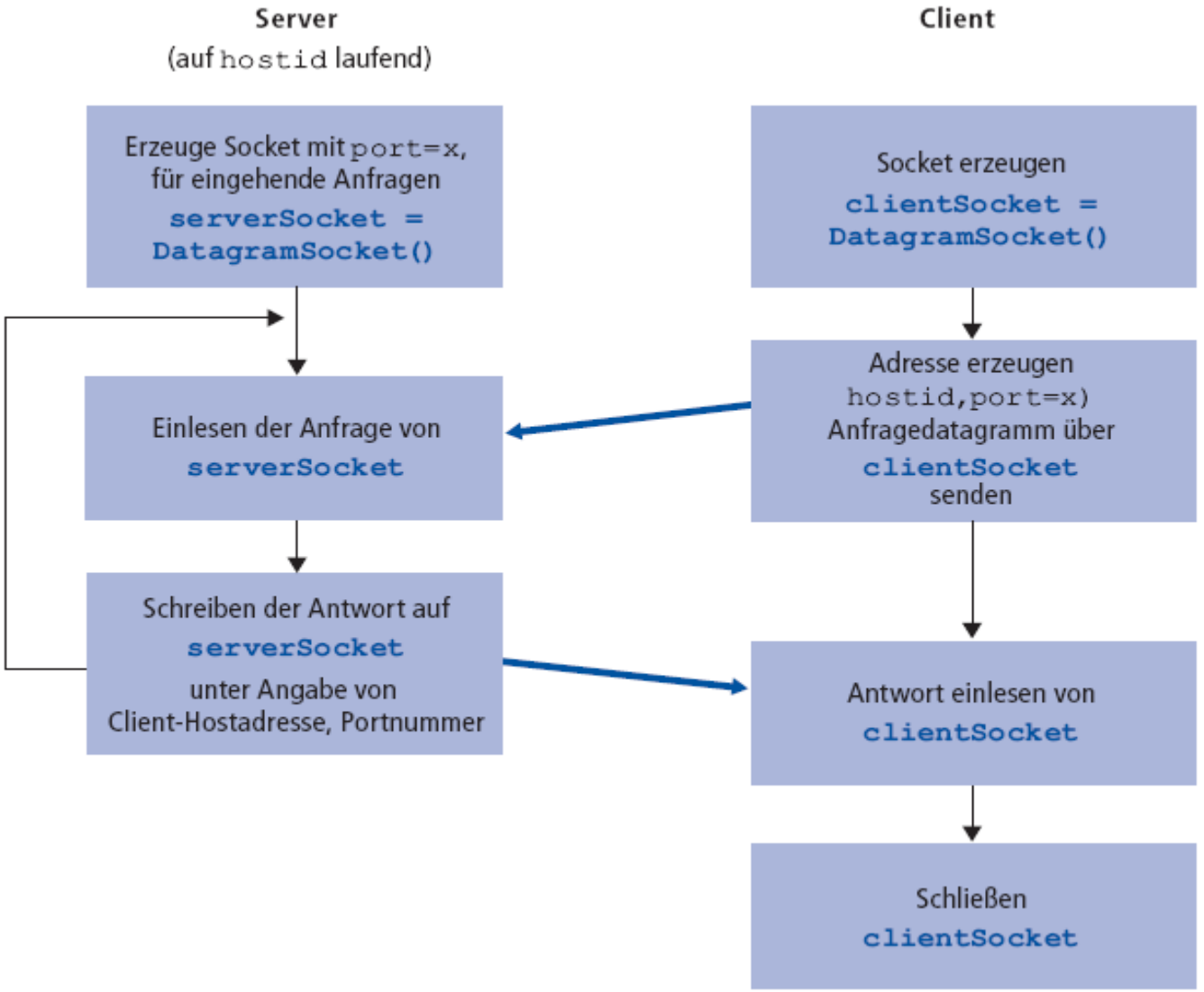
Socket-Programmierung mit *UDP*

UDP: keine „Verbindung“ zwischen Client und Server

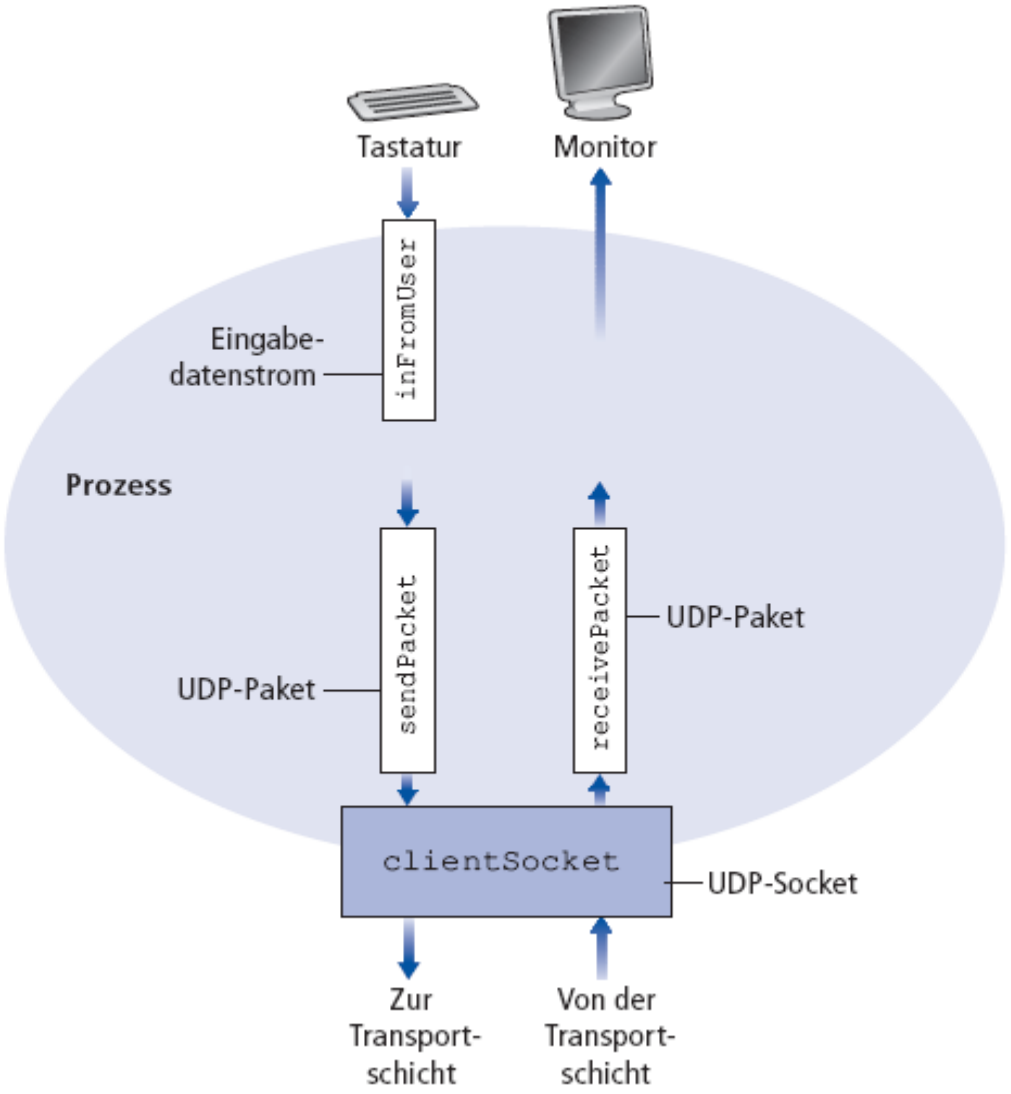
- Kein Verbindungsaufbau
- Sender hängt explizit die IP-Adresse und Port-nummer des empfangenden Prozesses an jedes Paket an
- Server liest die IP-Adresse und die Port-nummer des sendenden Prozesses explizit aus dem empfangenen Paket aus

UDP: Pakete können in falscher Reihenfolge empfangen werden oder ganz verloren gehen

Client/Server-Socket-Programmierung: UDP



Beispiel: Java-Client (UDP)



Kapitel 3: Zusammenfassung

- Grundlagen der Dienste in der Transportschicht:
  - Multiplexing, Demultiplexing
  - Zuverlässiger Datentransfer
  - Flusskontrolle
  - Überlastkontrolle
- Im Internet realisiert durch:
  - UDP
  - TCP

Als Nächstes:

- Wir verlassen den Rand des Netzwerkes (Anwendungs- und Transportschicht) …
- … und arbeiten uns ins Innere des Netzwerkes vor