Technische Hochschule Ingolstadt

Fakultät Informatik

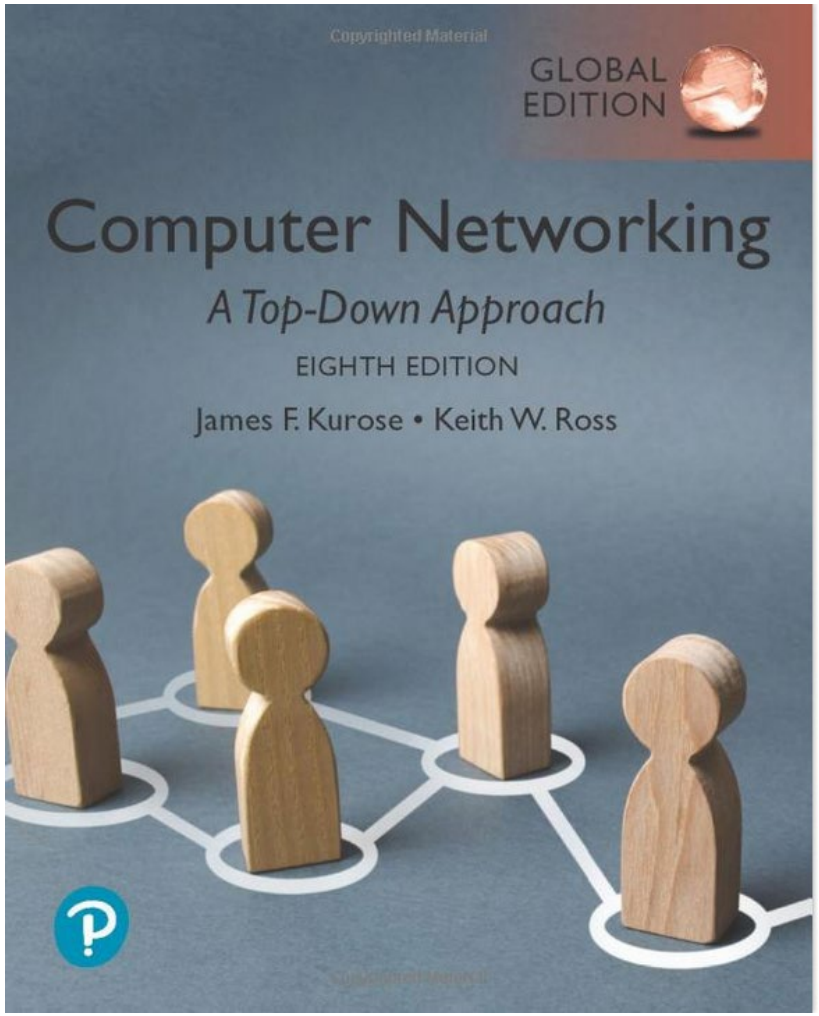
Kapitel 3: Transportschicht

FFI_NW WS 2024

Vorlesung „Netzwerke“

19.09.2024

Der Inhalt des Foliensatzes basiert auf bzw. ist adaptiert aus:



Computer Networking: A Top-Down Approach

8th edition [Global Edition]
Jim Kurose, Keith Ross
Pearson, 2021

ISBN-10 : 1292405465
ISBN-13 : 978-1292405469

Mehrere Ausgaben (auch deutsche Editionen) in der Bibliothek verfügbar

Unsere Ziele:

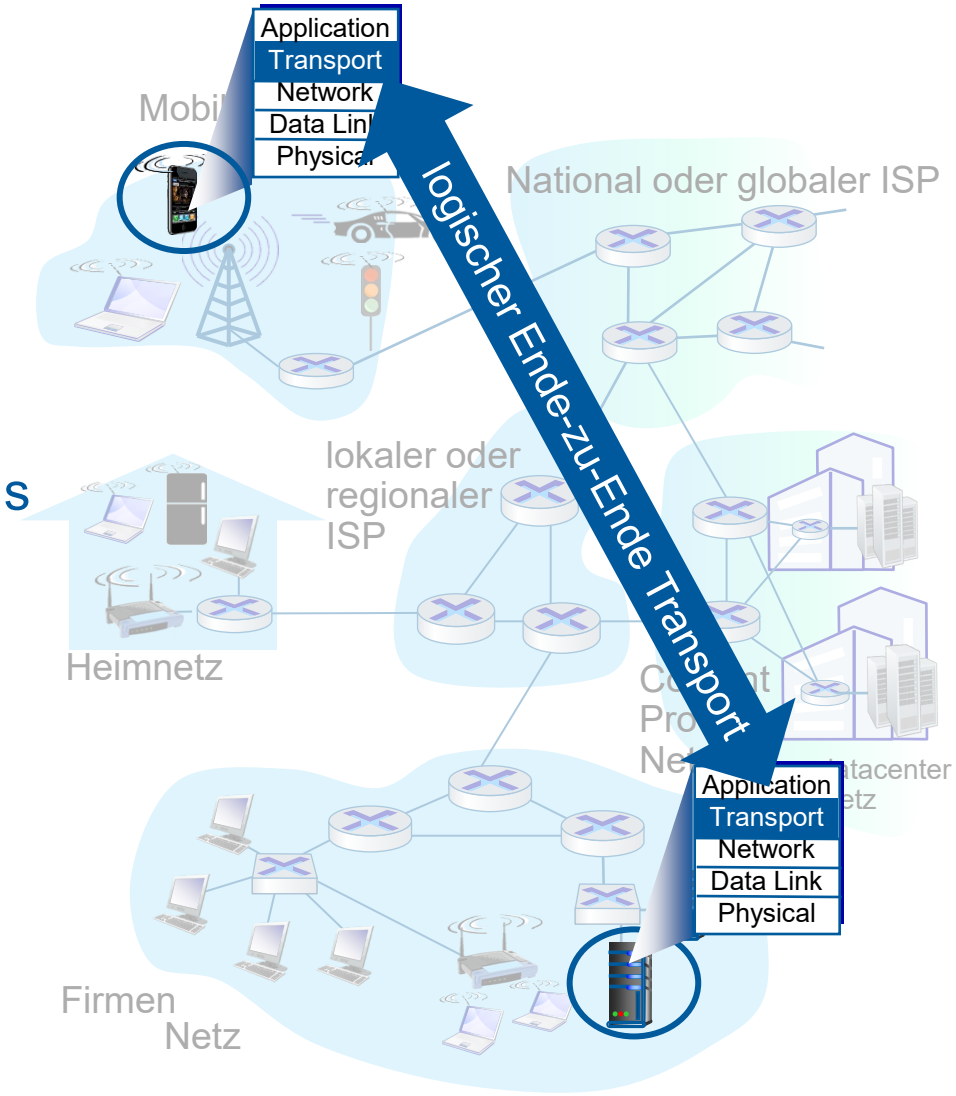
- **Verstehen der Prinzipien hinter den Diensten der Transportschicht:**
  - Multiplexen, Demultiplexen
  - Verlässliche Datenübertragung
  - Flusskontrolle
  - Überlastkontrolle
- **Die Protokolle der Transportschicht im Internet kennenlernen:**
  - UDP: verbindungsloser Transport
  - TCP: verbindungsorientierter, verlässlicher Transport
  - TCP Überlastkontrolle

Transportschicht: Übersicht

- **Dienste der Transportschicht**
- Multiplexen und Demultiplexen
- Verbindungsloser Transport: UDP
- Prinzipien verlässlicher Datenübertragung
- Verbindungsorientierter Transport: TCP
- Prinzipien der Überlastkontrolle
- TCP Überlastkontrolle
- Evolution der Transportschicht Funktionen

Transport Dienste und Protokolle

- stellt **logische Kommunikation** zwischen Applikationsprozessen auf verschiedenen Hosts zur Verfügung
- Aufgabe der Transportprotokolle in Endgeräten:
  - Sender: zerlegen von Applikationsnachrichten in **Segmente**, Weitergabe an Vermittlungsschicht
  - Empfänger: Zusammensetzen von Nachrichten aus Segementen, Weitergabe an Applikationsschicht
- Zwei Transportprotokolle stehen Internetapplikationen zur Verfügung
  - TCP, UDP



Haushaltsanalogie:

12 Kinder in Alices Haus schicken Briefe an 12 Kinder in Bobs Haus:

- Hosts = Häuser
- Prozesse = Kinder
- App Nachrichten = Briefe in Umschlägen

Transport vs Vermittlungsschicht: Dienste und Protokolle

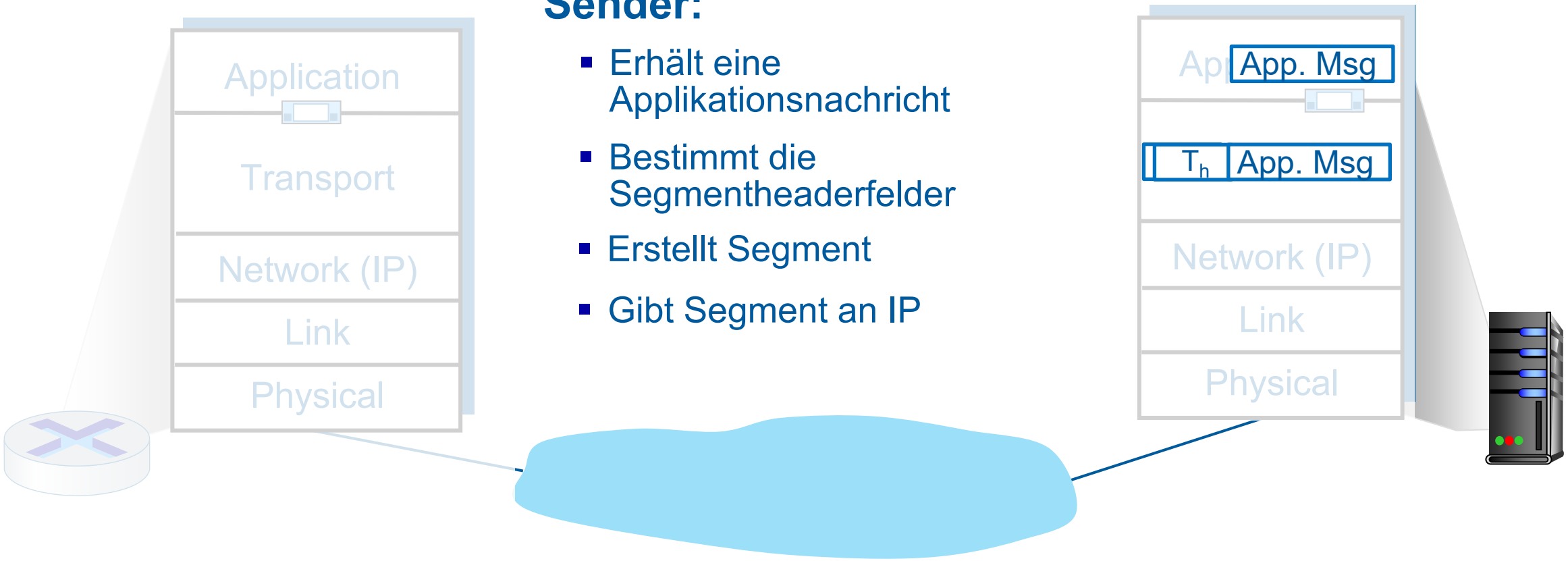
- **Vermittlungsschicht:** logische Kommunikation zwischen **Hosts**
- **Transportschicht:** logische Kommunikation zwischen **Prozessen**
  - Verlässt sich auf, erweitert Dienste der Vermittlungsschicht

Haushaltsanalogie:

12 Kinder in Alices Haus schicken Briefe an 12 Kinder in Bobs Haus:

- Hosts = Häuser
- Prozesse = Kinder
- App Nachrichten = Briefe in Umschlägen

Aufgaben der Transportschicht



Sender:

- Erhält eine Applikationsnachricht
- Bestimmt die Segmentheaderfelder
- Erstellt Segment
- Gibt Segment an IP

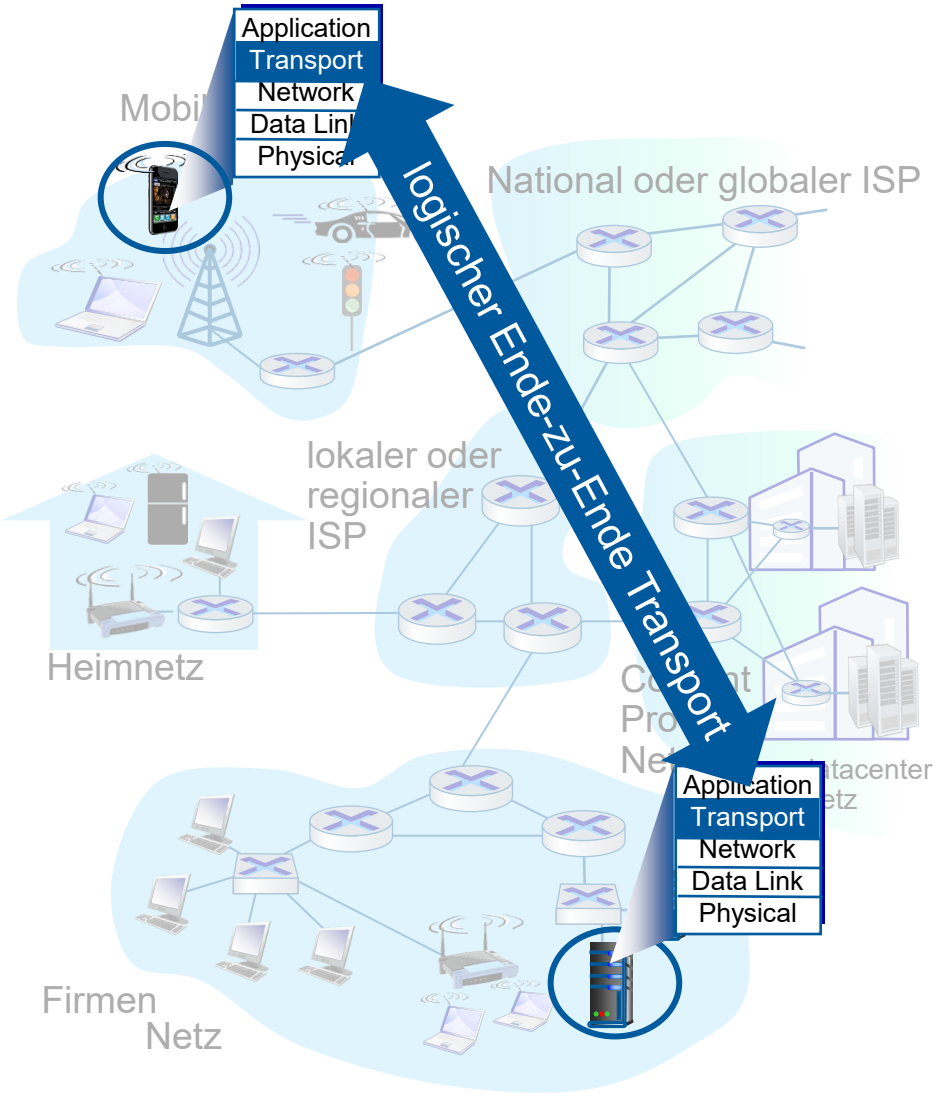
Aufgaben der Transportschicht

Empfänger:

- Empfängt Segment von IP
- Überprüft Headerwerte
- Extrahiert Applikationsnachricht
- Demultiplext Nachricht zur Applikation über Socket

Die zwei wichtigsten Internet Transportprotokolle

- **TCP:** Transmission Control Protocol
  - Zuverlässige Lieferung in Reihenfolge
  - Überlastkontrolle
  - Flusskontrolle
  - Verbindungsaufbau
- **UDP:** User Datagram Protocol
  - Unzuverlässige Lieferung ohne Reihenfolge
  - Einfache Erweiterung von “best-effort” IP
- Nicht verfügbare Dienste:
  - Latenzgarantien
  - Bandbreitengarantien



Transportschicht: Übersicht

- Dienste der Transportschicht
- **Multiplexen und Demultiplexen**
- Verbindungsloser Transport: UDP
- Prinzipien verlässlicher Datenübertragung
- Verbindungsorientierter Transport: TCP
- Prinzipien der Überlastkontrolle
- TCP Überlastkontrolle
- Evolution der Transportschicht Funktionen

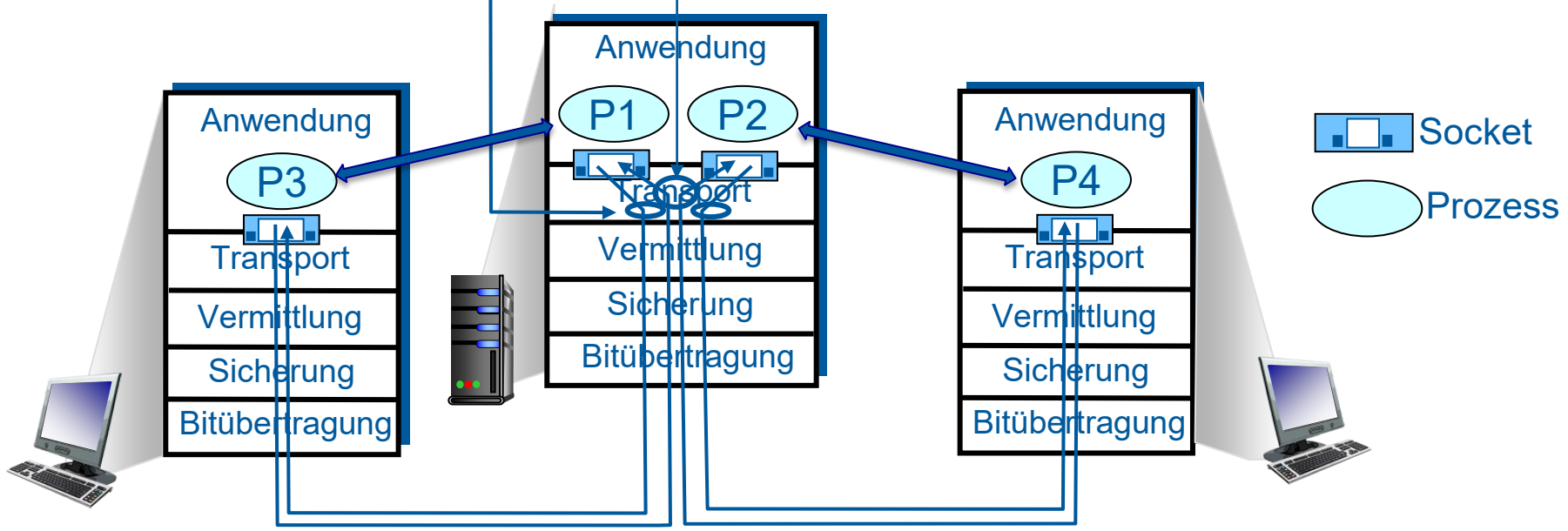
Multiplexen/Demultiplexen

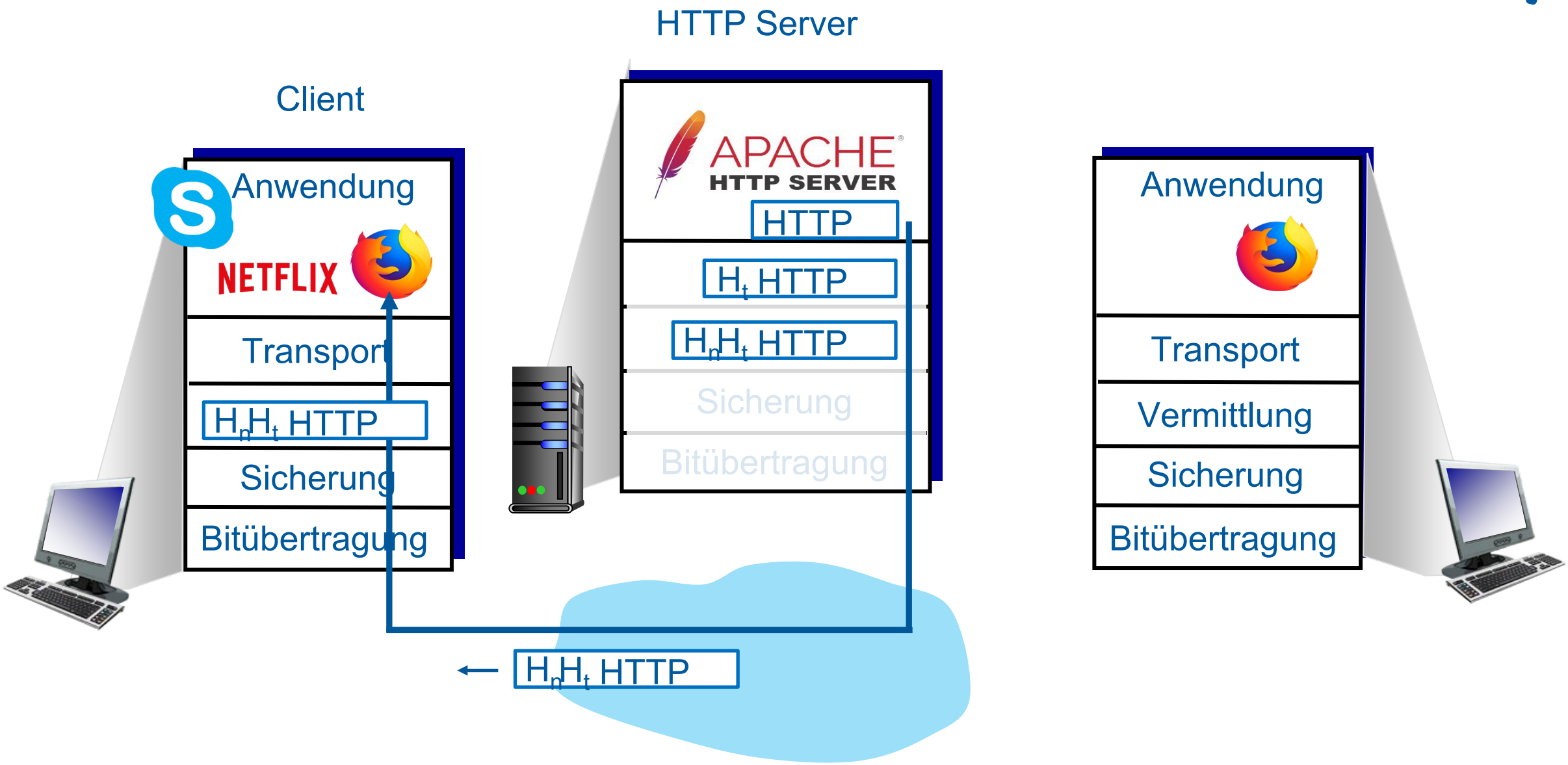
Multiplexen am Sender:

Behandeln von Daten von verschiedenen Sockets, Transportheader hinzufügen (für späteres Demultiplexen)

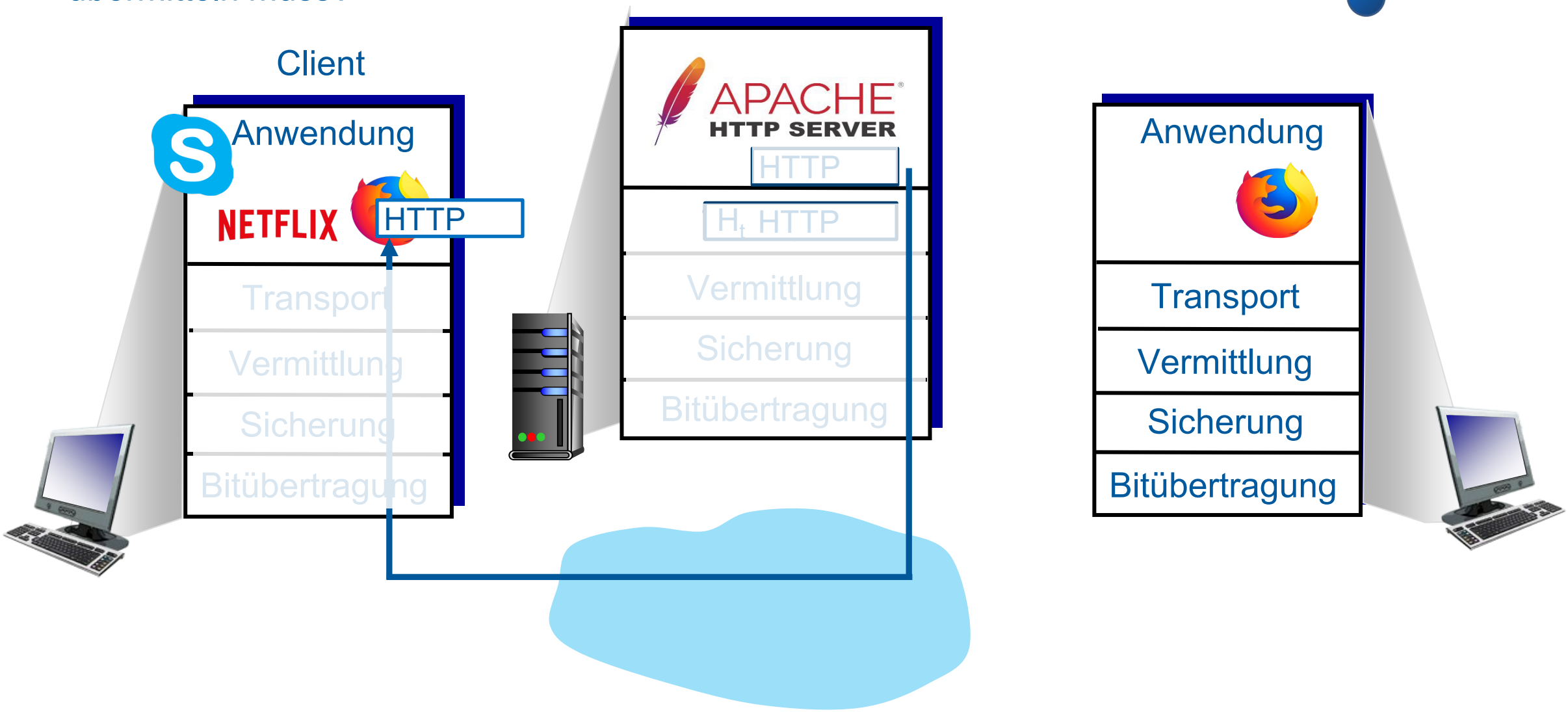
Demultiplexen am Empfänger:

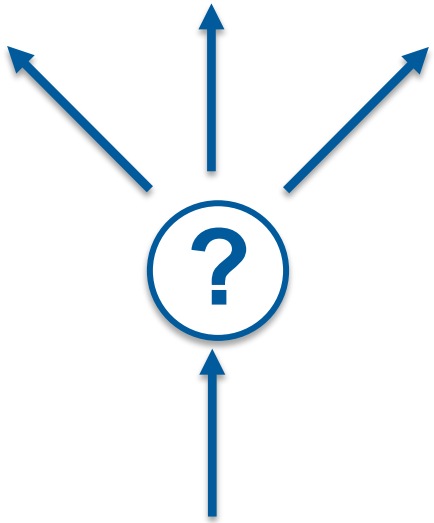
Nutzt Headerinformationen, um die empfangenen Segmente zu den richtigen Sockets zuzustellen



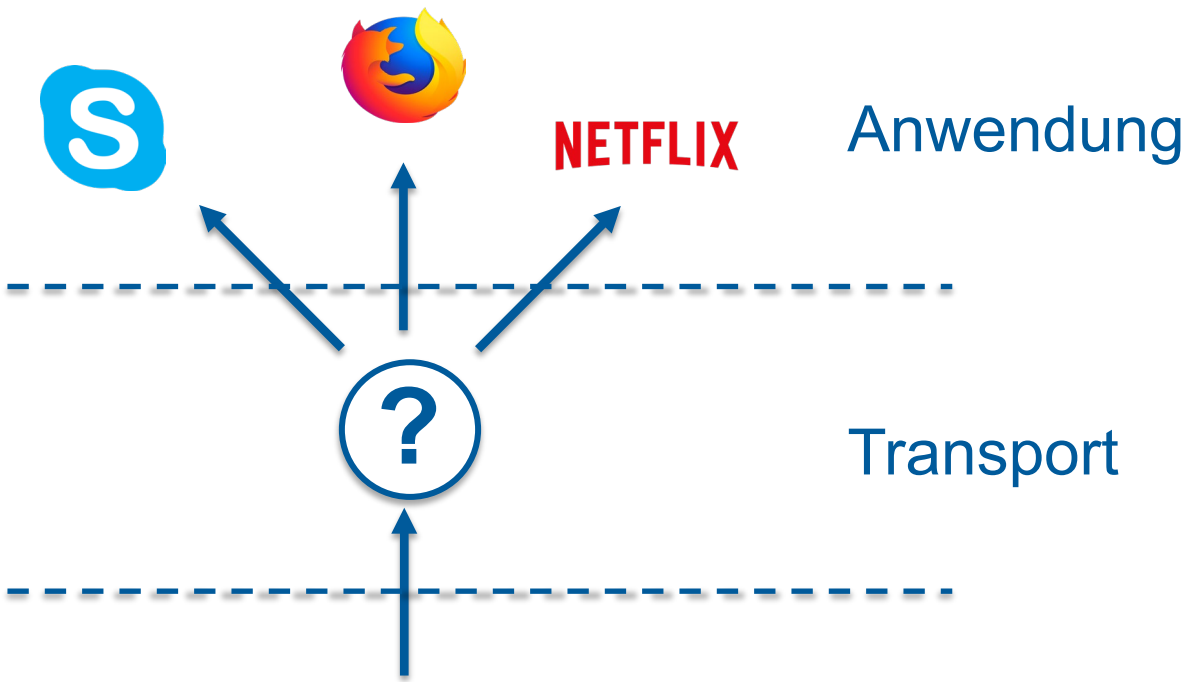

HTTP Server
Client
Anwendung
NETFLIX
Transport
H_nH_t HTTP
Sicherung
Bitübertragung
APACHE
HTTP SERVER
HTTP
H_t HTTP
H_nH_t HTTP
Sicherung
Bitübertragung
H_nH_t HTTP
Anwendung
Transport
Vermittlung
Sicherung
Bitübertragung

Frage: Woher wusste die Transportschicht, dass sie Nachrichten an den Firefox-Browserprozess und nicht an den Netflix-Prozess oder den Skype-Prozess übermitteln muss?



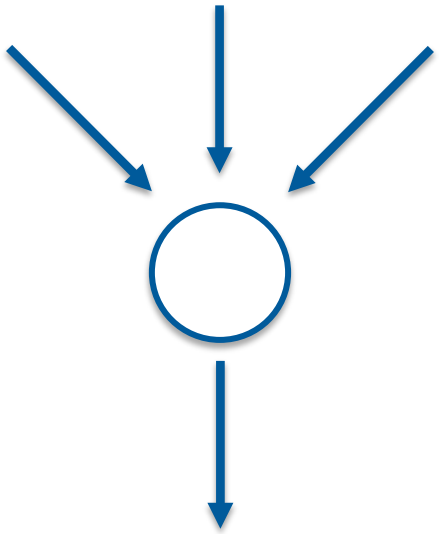


De-Multiplexen

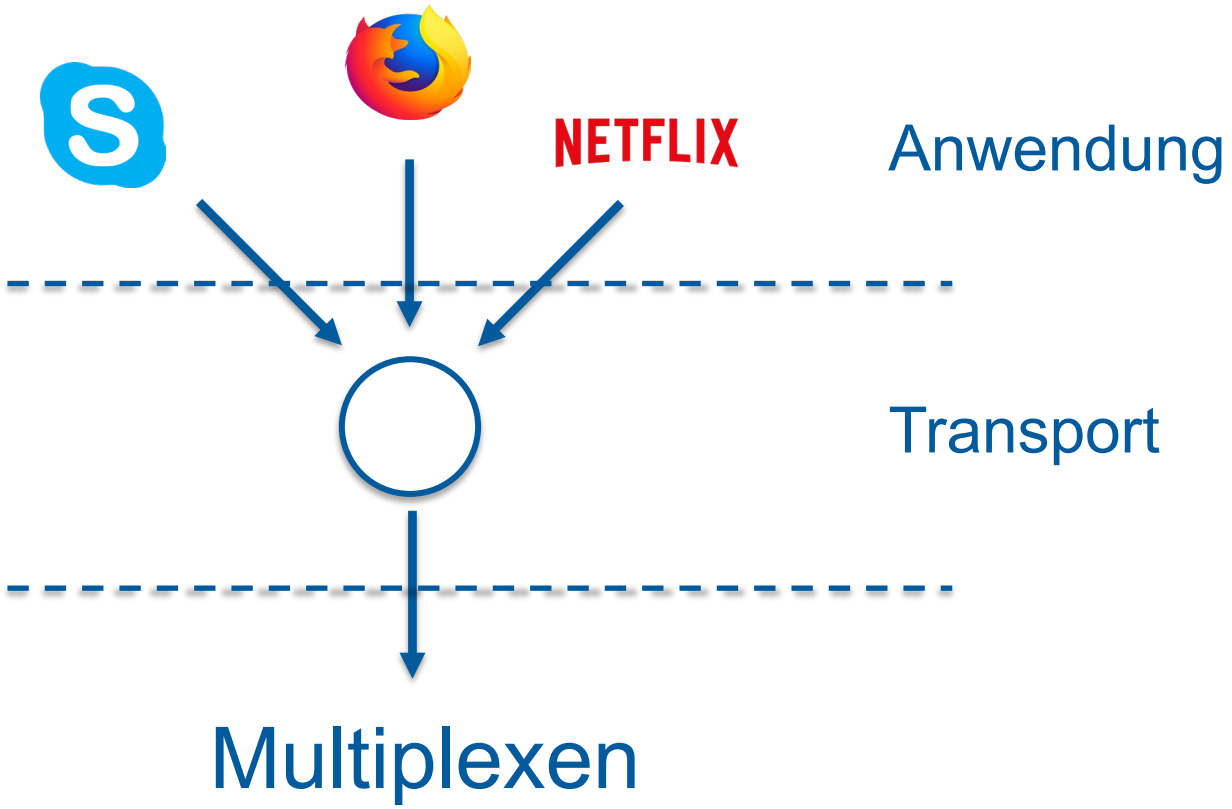


De-Multiplexen

Demultiplexen



Multiplexen

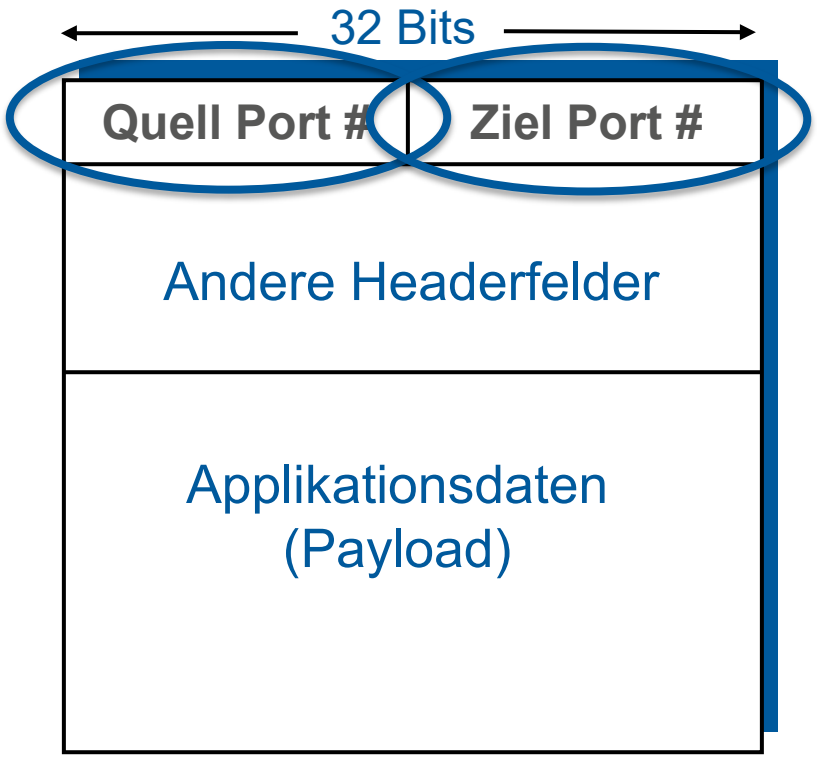

NETFLIX
Anwendung
Transport
Multiplexen

Multiplexen

https://tuev-thueringen.de

Wie Demultiplexen funktioniert

- Host empfängt IP-Datagramme
  - jedes Datagramm hat eine Quell- und Ziel IP Adresse
  - jedes Datagramm trägt ein Transportschichtsegment
  - jedes Segment hat eine Quell- und Zielportnummer
- Der Host nutzt **IP-Adressen & Port-Nummern** um ein Segment zum passenden Socket zu schicken



TCP/UDP Segment Format

Verbindungsloses Demultiplexen

- wenn ein Socket erstellt wird, muss eine **lokale** Port # angegeben werden:

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

- wenn ein Datagramm zum senden in einen UDP Socket erstellt wird, muss spezifiziert werden:
  - Ziel IP Adresse
  - Ziel Port #

wenn der empfangende Host ein UDP-Datagramm empfängt

- Überprüfen der Zielport # im Segment
- Leitet UDP-Segment an Socket mit dieser Port #

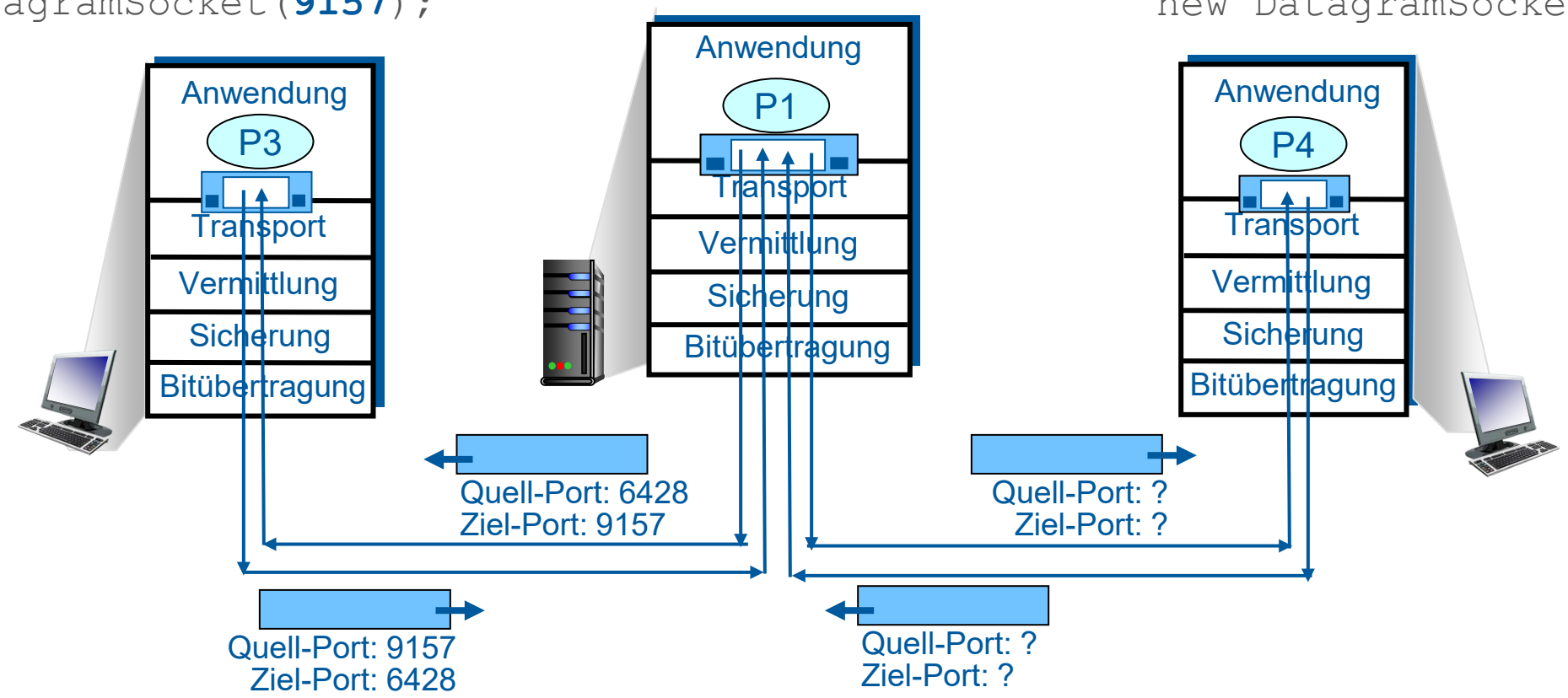
IP/UDP Datagramme mit **selber Zielport #**, aber verschiedenen Quell-IP Adressen und/oder Quellportnummern, werden an denselben **Socket** am Empfangshost geschickt

Verbindungsloses Demultiplexen: Beispiel

```
DatagramSocket
 serverSocket = new
 DatagramSocket(6428);
```

```
DatagramSocket mySocket2 =
 new DatagramSocket(9157);
```

```
DatagramSocket mySocket1 =
 new DatagramSocket(5775);
```

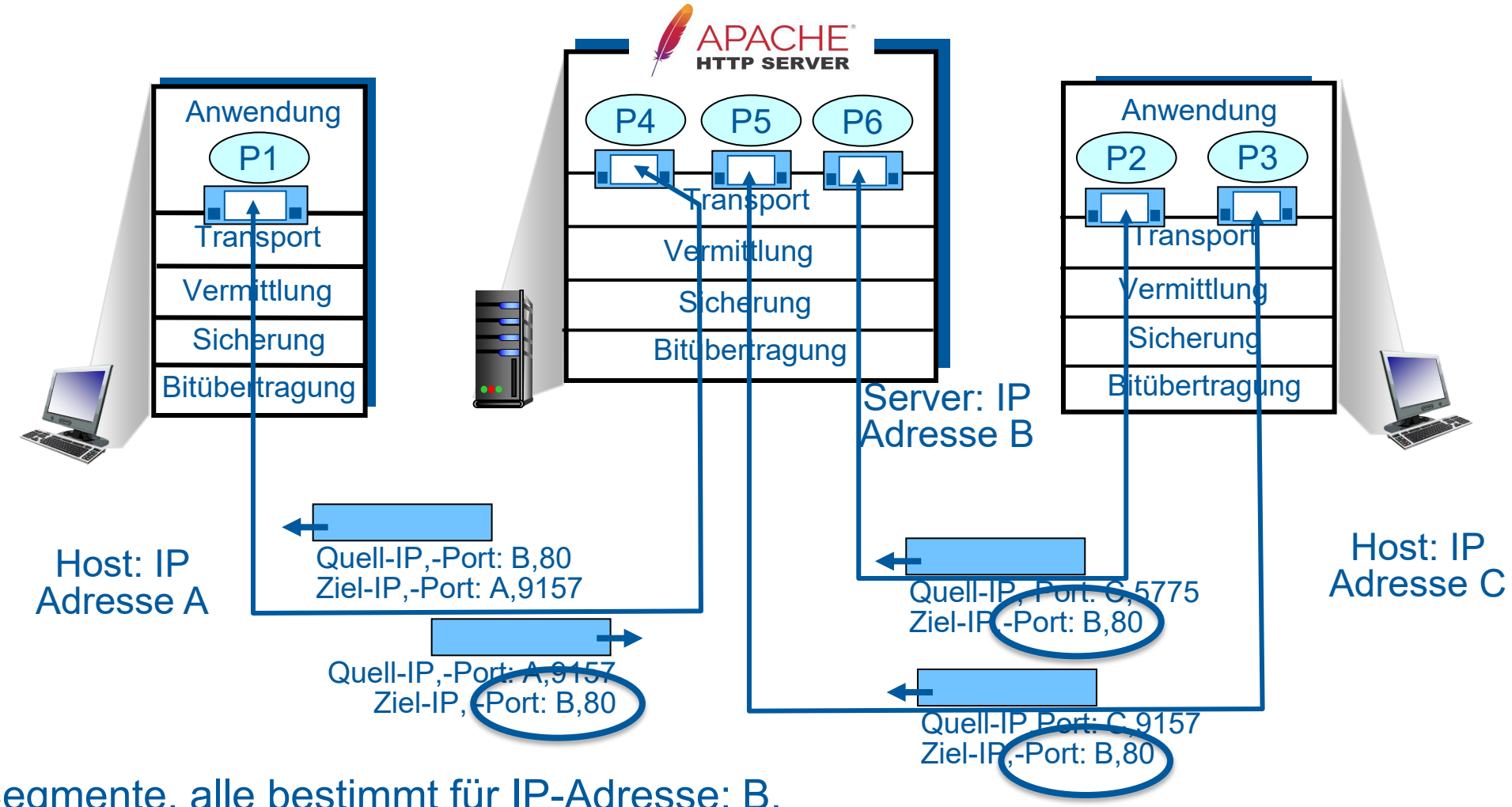


Verbindungs-orientiertes Demultiplexen

- TCP-Socket durch **4**-Tupel gekennzeichnet:
  - Quell-IP-Adresse
  - Quellportnummer
  - Ziel-IP-Adresse
  - Zielportnummer
- Demultiplexen: Empfänger nutzt **alle vier Werte (4-Tupel)** um das Segment an den passenden Sockel weiterzuleiten

- Ein Server kann viele TCP-Sockets gleichzeitig unterstützen:
  - jeder Socket ist durch sein eigenes 4-Tupel gekennzeichnet
  - jeder Socket ist mit einem anderen verbundenen Client assoziiert

Verbindungs-orientiertes Demultiplexen: Beispiel



Drei Segmente, alle bestimmt für IP-Adresse: B,
Ziel-Port: 80 werden an **verschiedene** Sockets demultiplext

Zusammenfassung

- Multiplexen, Demultiplexen: basierend auf Werten in Segment-, Datagramm-Headerfeldern
- **UDP:** Demultiplexen (nur) mit Zielportnummer
- **TCP:** Demultiplexen mit 4-Tupel: Quell- und Ziel-IP Adressen und Portnummern
- Multiplexen/Demultiplexen passiert auf **allen** Schichten

Transportschicht: Übersicht

- Dienste der Transportschicht
- Multiplexen und Demultiplexen
- **Verbindungsloser Transport: UDP**
- Prinzipien verlässlicher Datenübertragung
- Verbindungsorientierter Transport: TCP
- Prinzipien der Überlastkontrolle
- TCP Überlastkontrolle
- Evolution der Transportschicht Funktionen

- Simples Internet Transport Protokoll
- “best effort” Dienst, UDP-Segmente können:
  - verloren gehen
  - in falscher Reihenfolge geliefert werden

- **verbindungslos:**
  - kein Handshake zwischen UDP-Sender und Empfänger
  - jedes UDP-Segment wird unabhängig von anderen behandelt

Warum gibt es UDP?

- Kein Verbindungsaufbau (der Verzögerung bedeuten würde)
- einfach: kein Verbindungszustand beim Sender und Empfänger
- Kleine Header Größe
- Keine Überlastkontrolle
  - UDP kann so schnell senden, wie es möchte!
  - kann trotz Überlast im Netz funktionieren

UDP: User Datagram Protocol

- UDP wird verwendet für:
  - Streaming Multimedia Apps (tolerant gegenüber Verlust, empfindlich gegenüber der Rate)
  - DNS
  - SNMP
  - HTTP/3
- falls verlässlicher Datentransfer über UDP benötigt wird (z.B., HTTP/3):
  - Hinzufügen der benötigten Verlässlichkeit auf Applikationsschicht
  - Hinzufügen von Überlastkontrolle auf Applikationsschicht

UDP: User Datagram Protocol

INTERNET STANDARD

RFC 768

J. Postel
ISI
28 August 1980

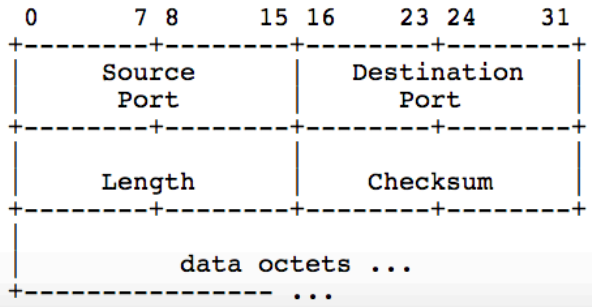
User Datagram Protocol

Introduction

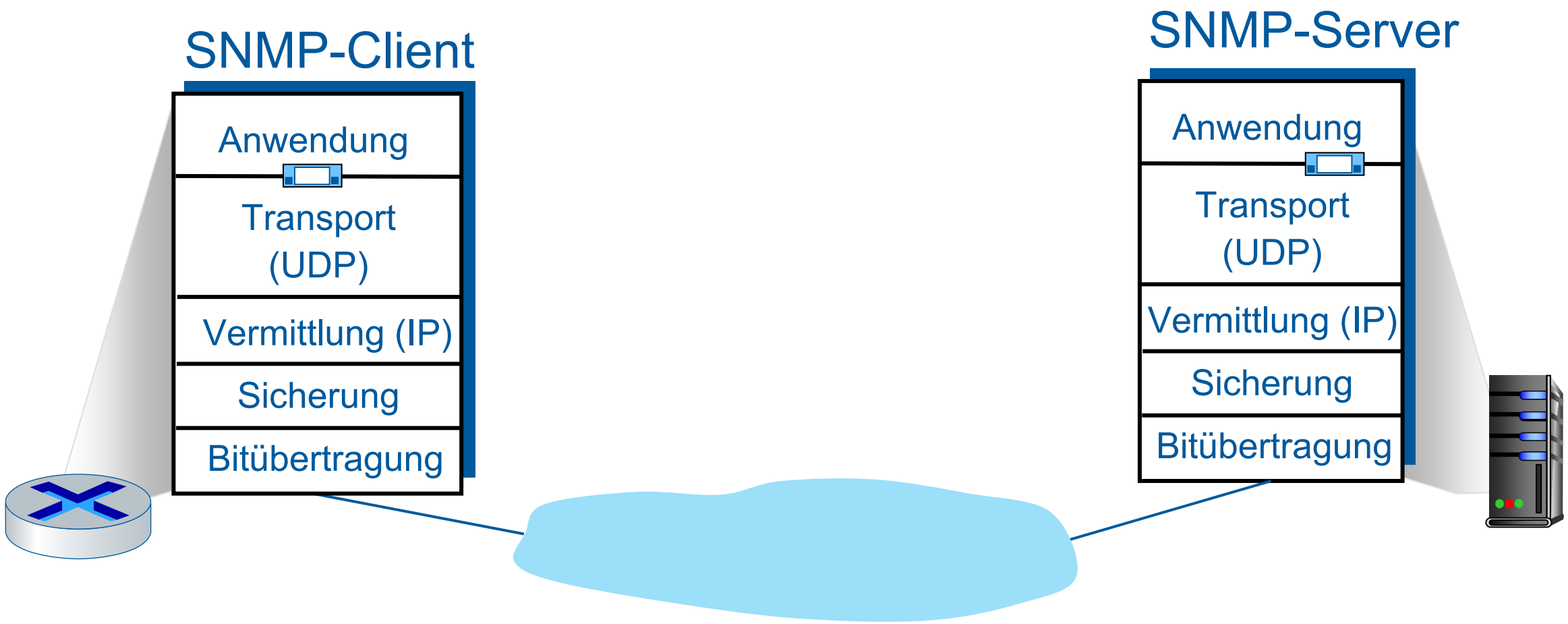
This User Datagram Protocol (UDP) is defined to make available a datagram mode of packet-switched computer communication in the environment of an interconnected set of computer networks. This protocol assumes that the Internet Protocol (IP) [1] is used as the underlying protocol.

This protocol provides a procedure for application programs to send messages to other programs with a minimum of protocol mechanism. The protocol is transaction oriented, and delivery and duplicate protection are not guaranteed. Applications requiring ordered reliable delivery of streams of data should use the Transmission Control Protocol (TCP) [2].

Format



UDP: Transportschicht Aufgaben

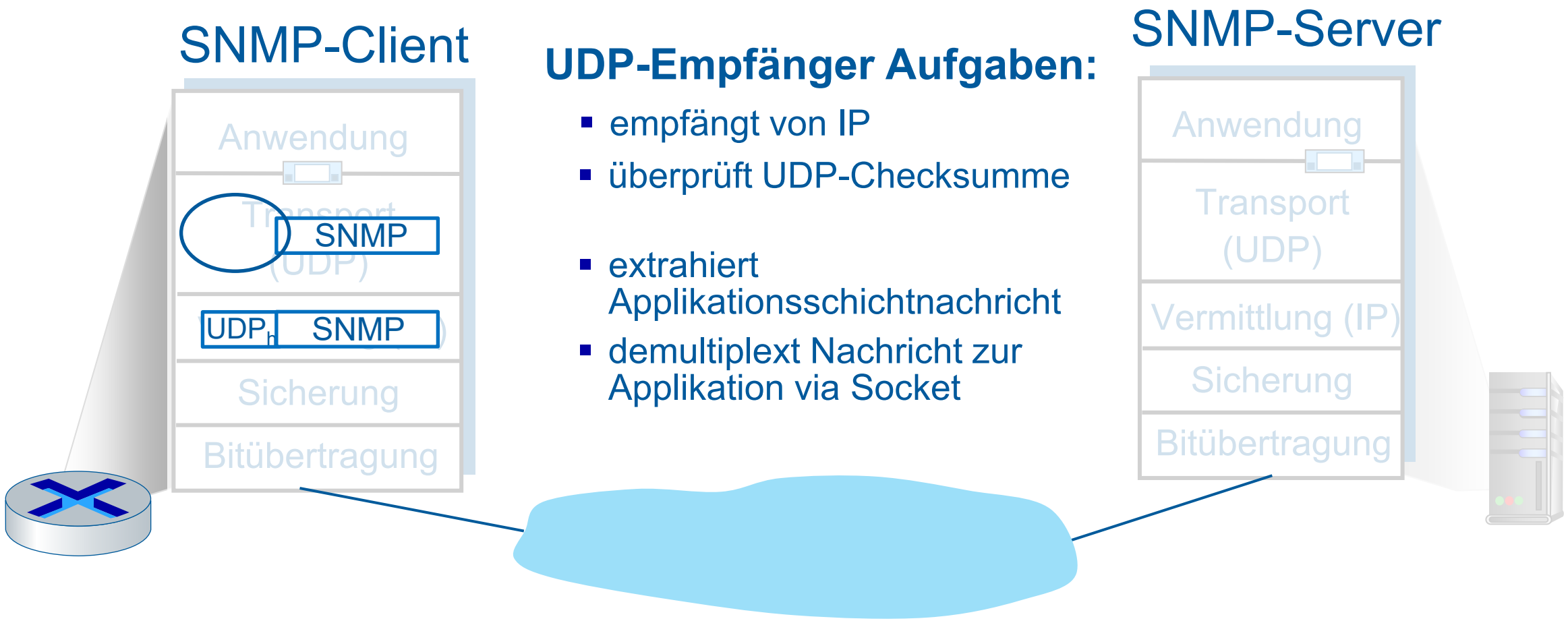


UDP: Transportschicht Aufgaben

UDP-Sender Aufgaben

- erhält eine Applikationsschicht Nachricht
- bestimmt Werte für UDP Segment Headerfelder
- erstellt UDP-Segment
- übergibt Segment an IP

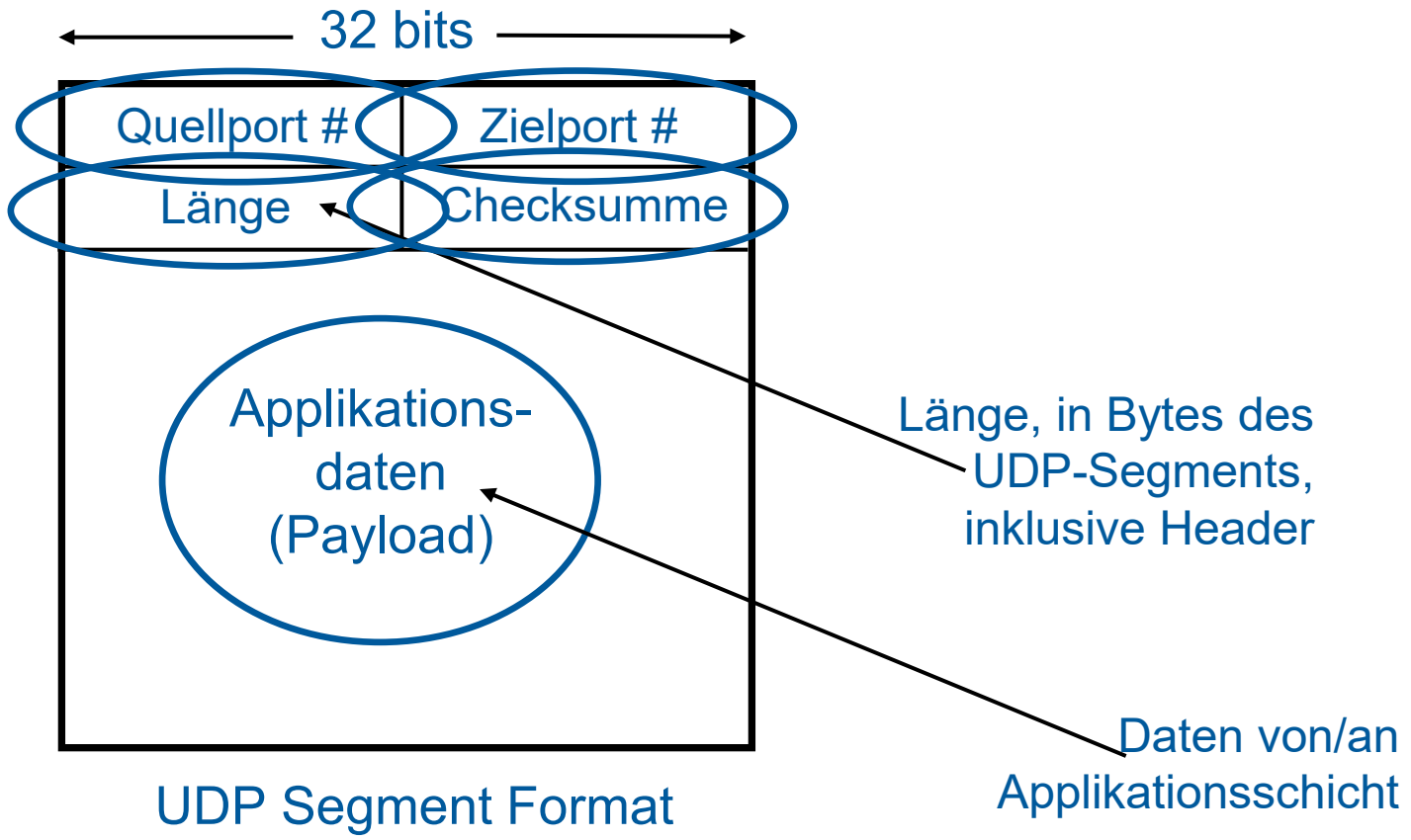
UDP: Transportschicht Aufgaben



UDP-Empfänger Aufgaben:

- empfängt von IP
- überprüft UDP-Checksumme
- extrahiert Applikationsschichtnachricht
- demultiplext Nachricht zur Applikation via Socket

UDP Segment Header



Ziel: Fehler (d.h., geflippte Bits) in übertragenem Segment erkennen

	1. Zahl	2. Zahl	Summe
Übertragen:	5	6	11
Empfangen:	4	6	11

vom Empfänger berechnete Checksumme ≠ vom Sender berechnete Checksumme (wie empfangen)

Ziel: Fehler (d.h., geflippte Bits) in übertragenem Segment erkennen

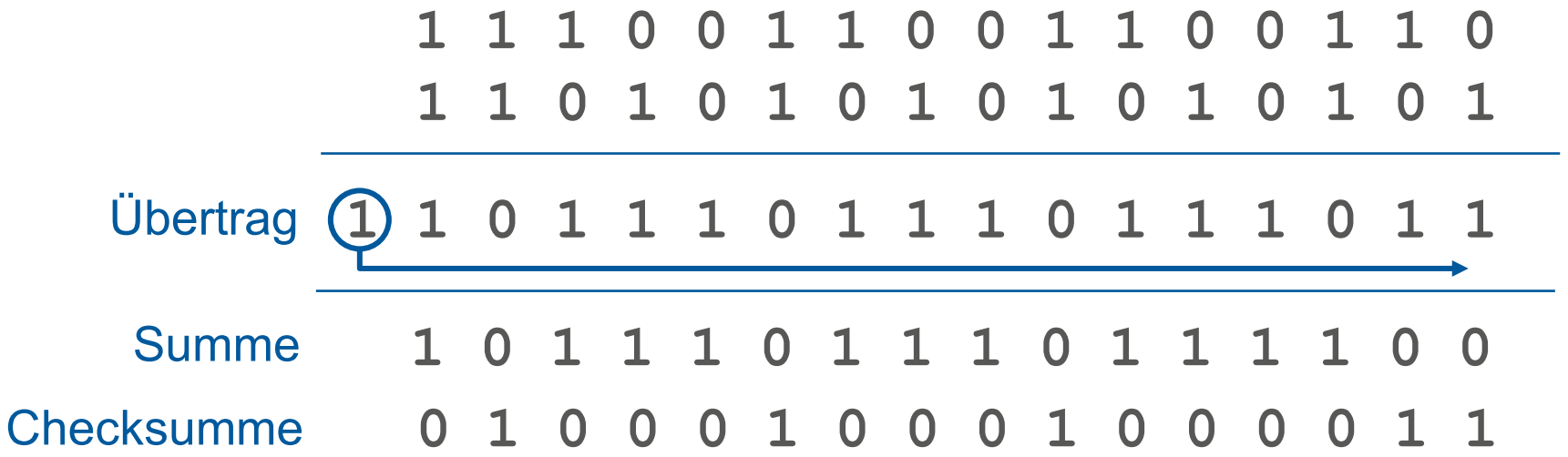
Sender:

- Behandeln des Inhaltes eines UDP-Segments (inklusive UDP-Headerfeldern und IP-Adressen) als Sequenz aus 16-Bit Ganzzahlen
- **Checksumme:** Addition (Einserkomplement) des Segmentinhalts
- Checksummenwert wird in das UDP-Checksummenfeld übertragen

Empfänger:

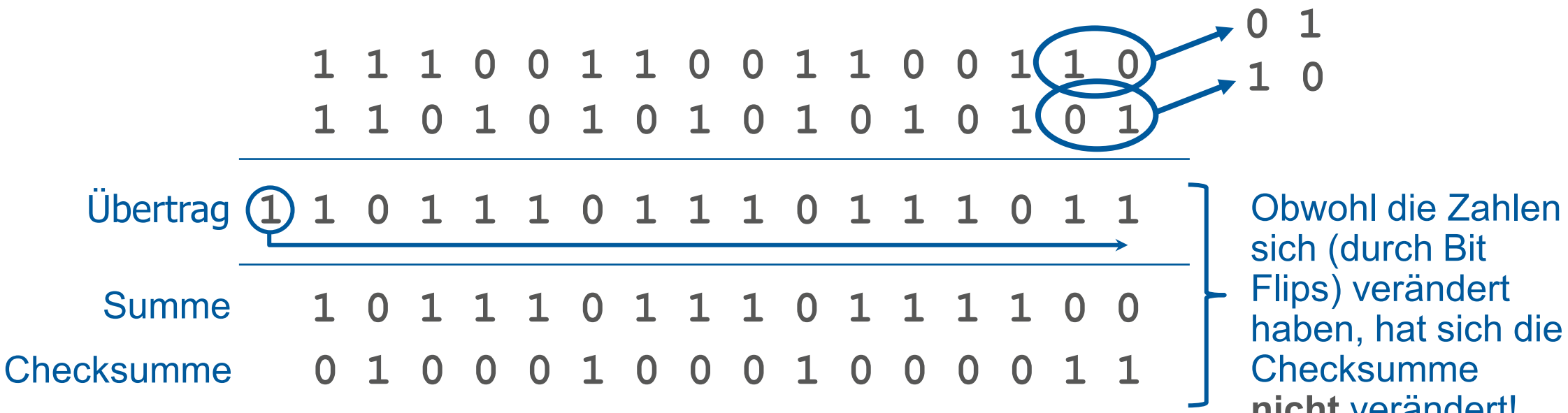
- Berechnen der Checksumme für das empfangene Segment
- Überprüfen, ob berechnete Checksumme dem Wert im Checksummenfeld entspricht:
  - Ungleich – Fehler gefunden
  - Gleich – Kein Fehler gefunden. Aber trotzdem können noch Fehler vorhanden sein….

Beispiel: addieren zweier 16-Bit Ganzzahlen



Hinweis: bei der Addition von Zahlen, muss ein Übertrag vom signifikantesten Bit zum Ergebnis addiert werden

Beispiel: addieren zweier 16-Bit Ganzzahlen



Zusammenfassung: UDP

- “einfaches” Protokoll:
  - Segmente können verloren gehen oder außer Reihenfolge ankommen
  - Best Effort Dienst: “Verschicken und auf das Beste hoffen”
- UDP hat Vorteile:
  - kein Setup/Handshake benötigt
  - kann noch funktionieren, wenn das Netz überlastet ist
  - Unterstützt Verlässlichkeit (Checksumme)
- Zusätzliche Funktionalität kann über UDP in der Applikationsschicht realisiert werden (z.B., HTTP/3)

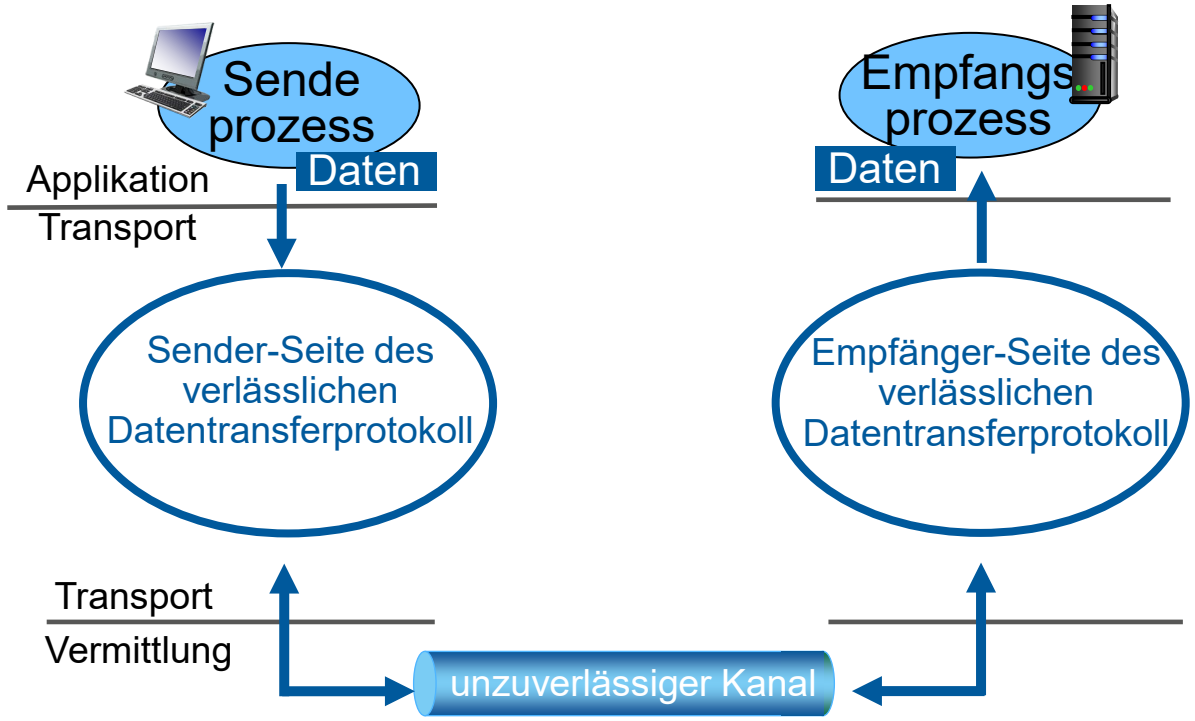
Transportschicht: Übersicht

- Dienste der Transportschicht
- Multiplexen und Demultiplexen
- Verbindungsloser Transport: UDP
- **Prinzipien verlässlicher Datenübertragung**
- Verbindungsorientierter Transport: TCP
- Prinzipien der Überlastkontrolle
- TCP Überlastkontrolle
- Evolution der Transportschicht Funktionen

Zuverlässiger Dienst-**Abstraktion**

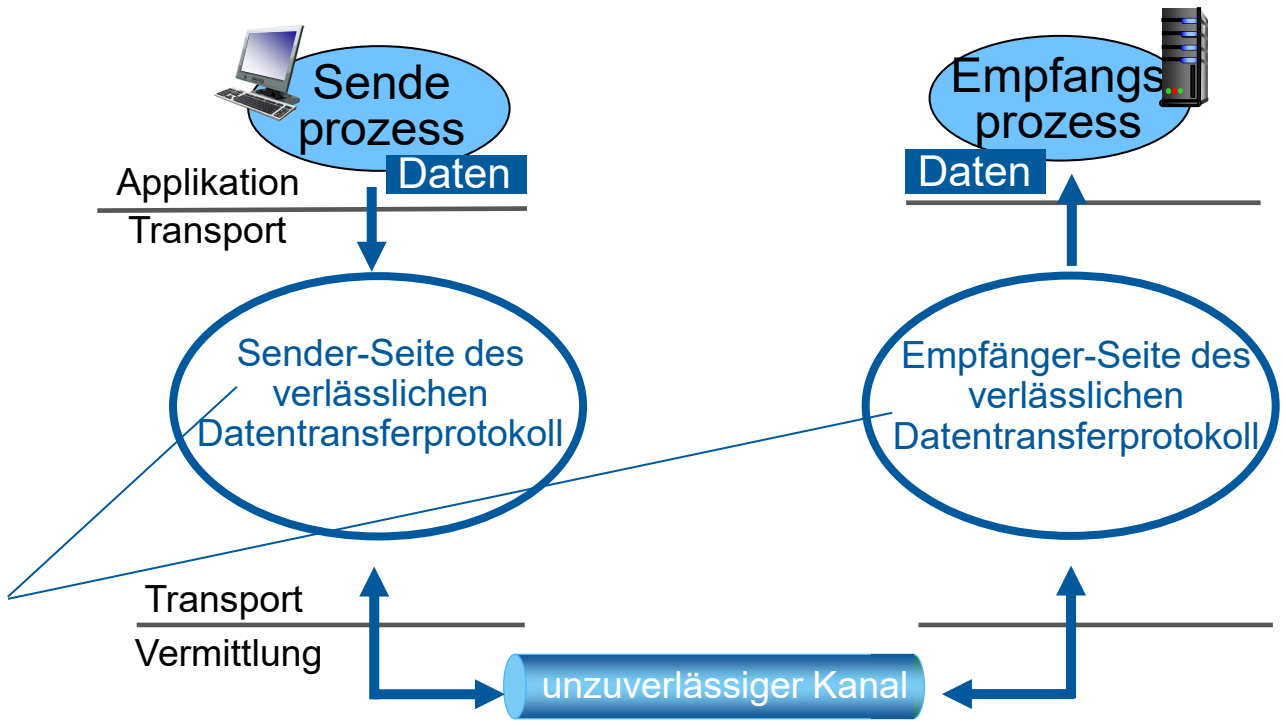
Prinzipien verlässlicher Datenübertragung

Zuverlässiger Dienst-**Abstraktion**



Zuverlässiger Dienst-**Implementierung**

Die Komplexität des verlässlichen Datentransferprotokoll hängt (stark) von den Charakteristiken des unzuverlässigen Kanals ab (verlorene, fehlerhafte, in der Reihenfolge geänderte Daten?)

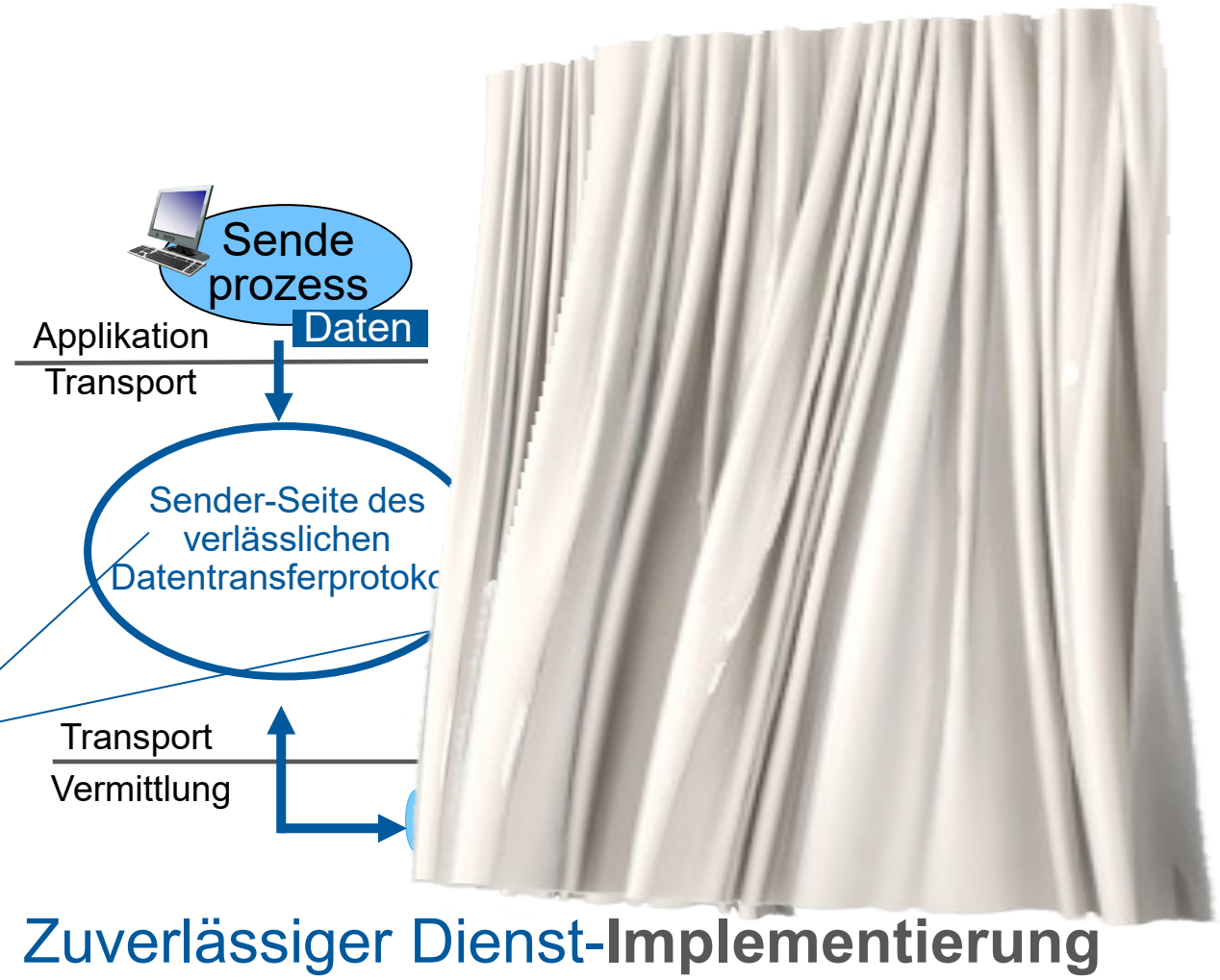


Zuverlässiger Dienst-**Implementierung**

Prinzipien verlässlicher Datenübertragung

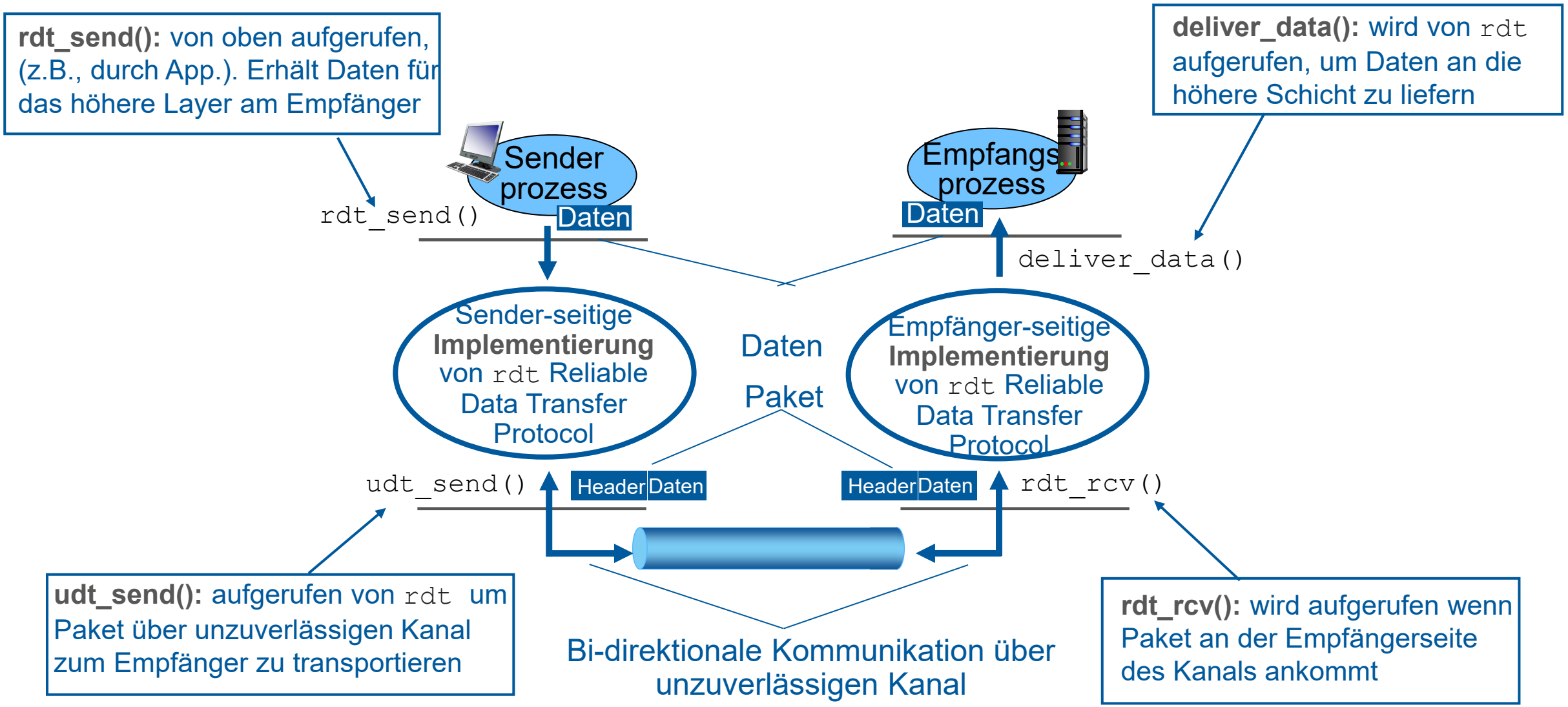
Sender, Empfänger kennen den “Zustand” des anderen *nicht*,
z.B., wurde eine Nachricht empfangen?

- sofern nicht durch Nachricht mitgeteilt



Zuverlässiger Dienst-**Implementierung**

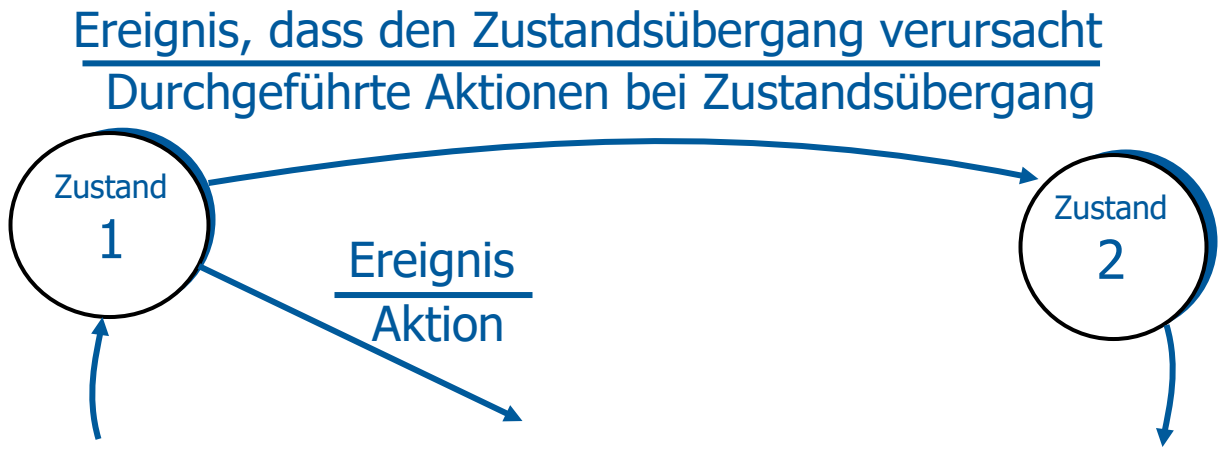
Reliable Data Transfer Protocol (rdt): Schnittstellen



Wir werden:

- inkrementell die Sender- und Empfangsseite des **R**eliable **D**ata **T**ransfer Protokoll (`rdt`) entwickeln
- nur uni-direktionalen Datentransfer betrachten
  - aber Kontrollinformation wird in beide Richtungen fließen!
- endliche Zustandsautomaten benutzen um den Sender, Empfänger zu spezifizieren

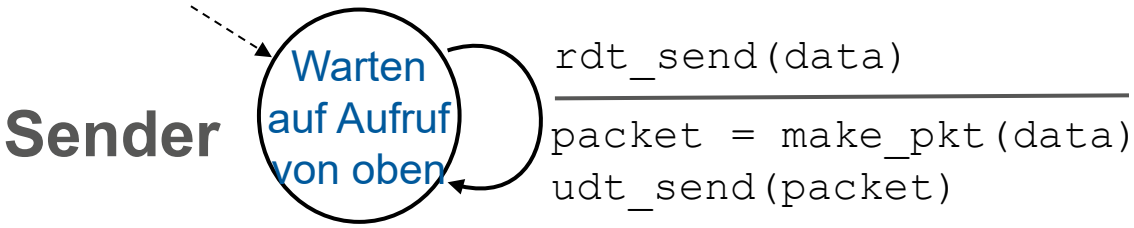
Zustand: wenn in diesem "Zustand" wird der nächste Zustand nur durch das nächste Ereignis bestimmt

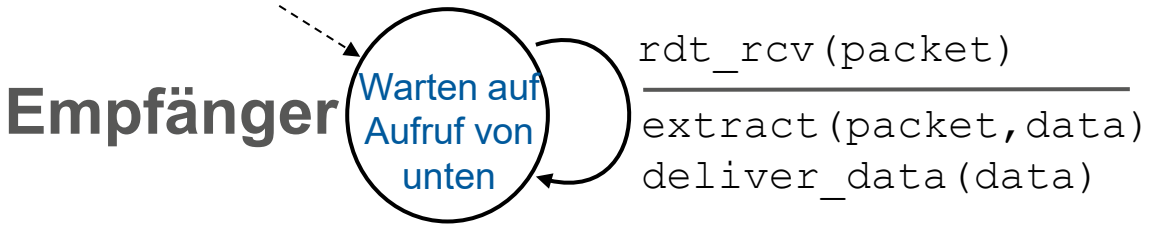


rdt1.0: Verlässliche Datenübertragung über einen zuverlässigen Kanal

- Kanal komplett zuverlässig
  - Keine Bitfehler
  - Kein Paketverlust

- **separate** Zustandsautomaten für Sender, Empfänger:
  - Sender sendet Daten auf Kanal
  - Empfänger liest Daten vom Kanal





rdt2.0: Kanal mit Bitfehlern

- Kanal kann Bits in Paket flippen
  - Checksumme (z.B., Internet Checksumme) um Bitfehler zu erkennen
- **die** Frage: wie Fehler beheben?

Frage: Wie beheben Menschen “Fehler” im Gespräch?

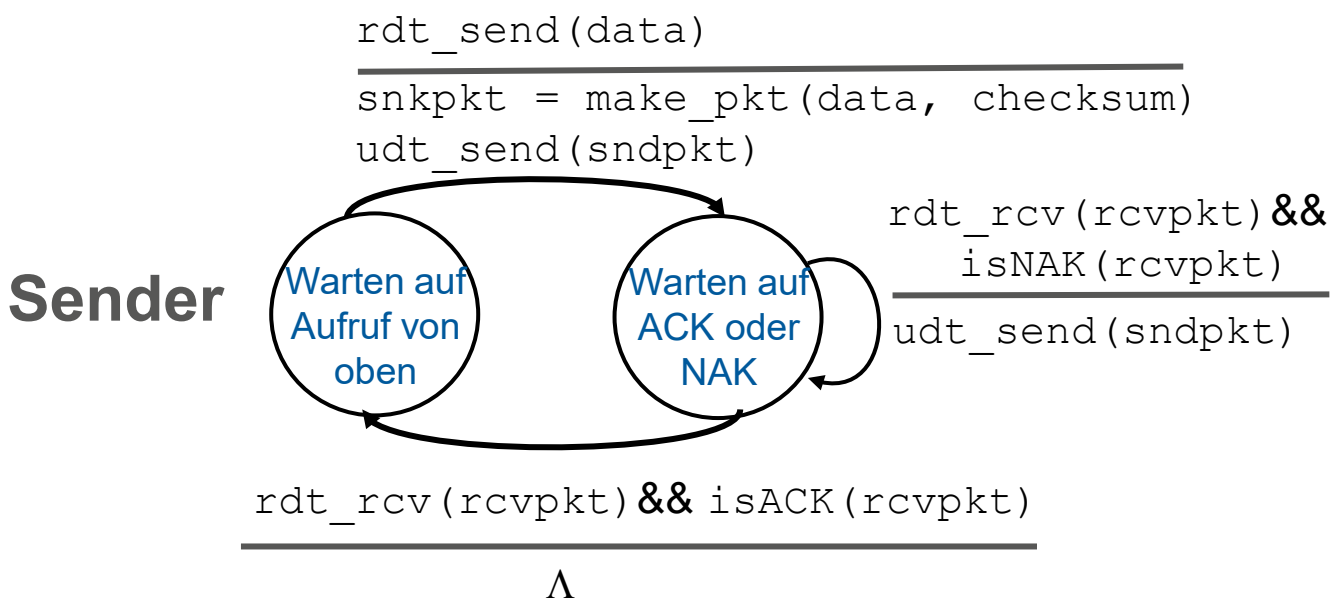
rdt2.0: Kanal mit Bitfehlern

- Kanal kann Bits in Paket flippen
  - Checksumme, um Bitfehler zu erkennen
- **die** Frage: wie Fehler beheben?
  - **Acknowledgements (ACKs):** Empfänger sagt dem Sender explizit, dass das Paket korrekt empfangen wurde
  - **Negative Acknowledgements (NAKs):** Empfänger sagt dem Sender explizit, dass das Paket fehlerhaft war
  - Sender **wiederholt Übertragung** des Pakets bei Empfang von NAK

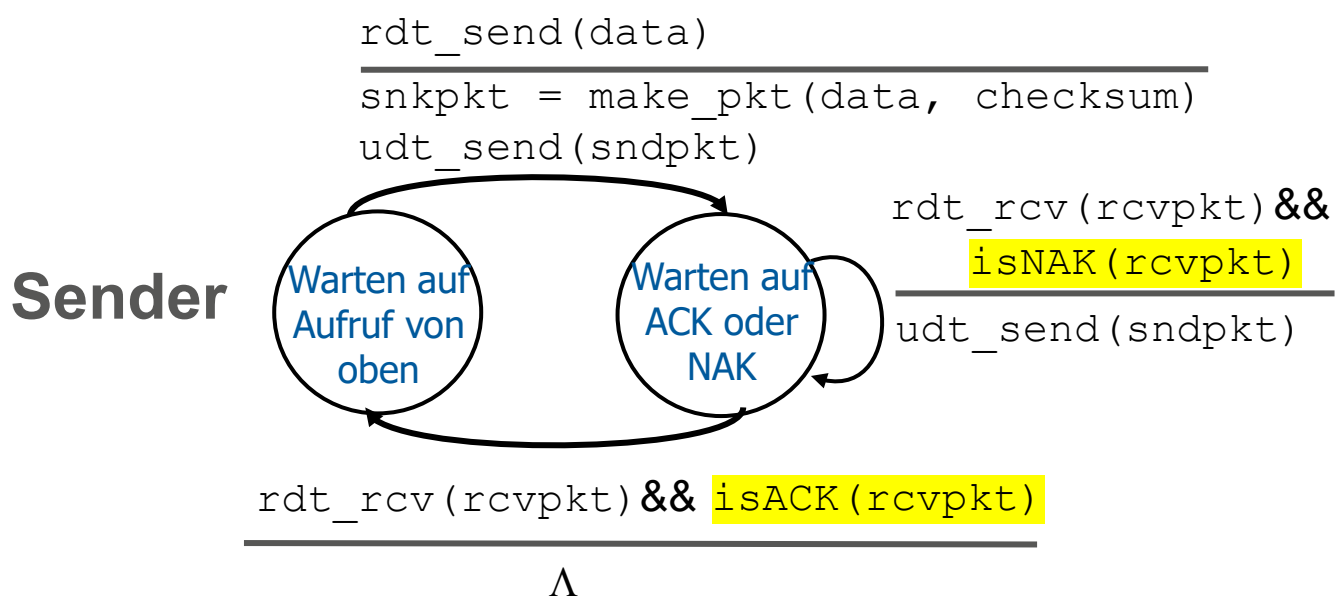
Stop and Wait

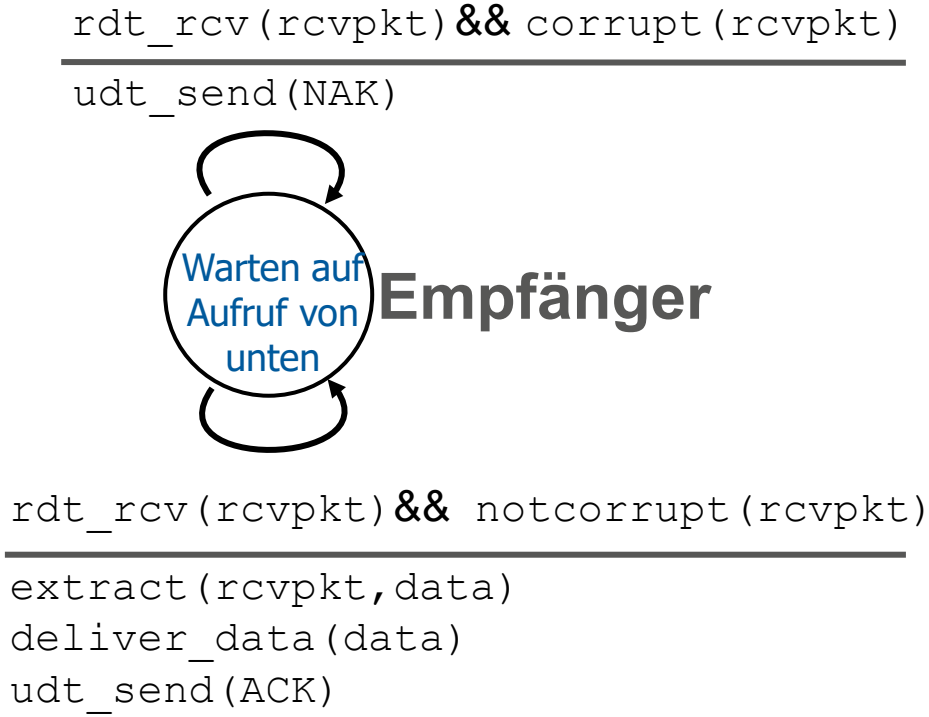
Sender sendet ein Paket und wartet dann auf die Antwort des Empfängers

rdt2.0: Spezifikation des Zustandsautomaten



rdt2.0: Spezifikation des Zustandsautomaten

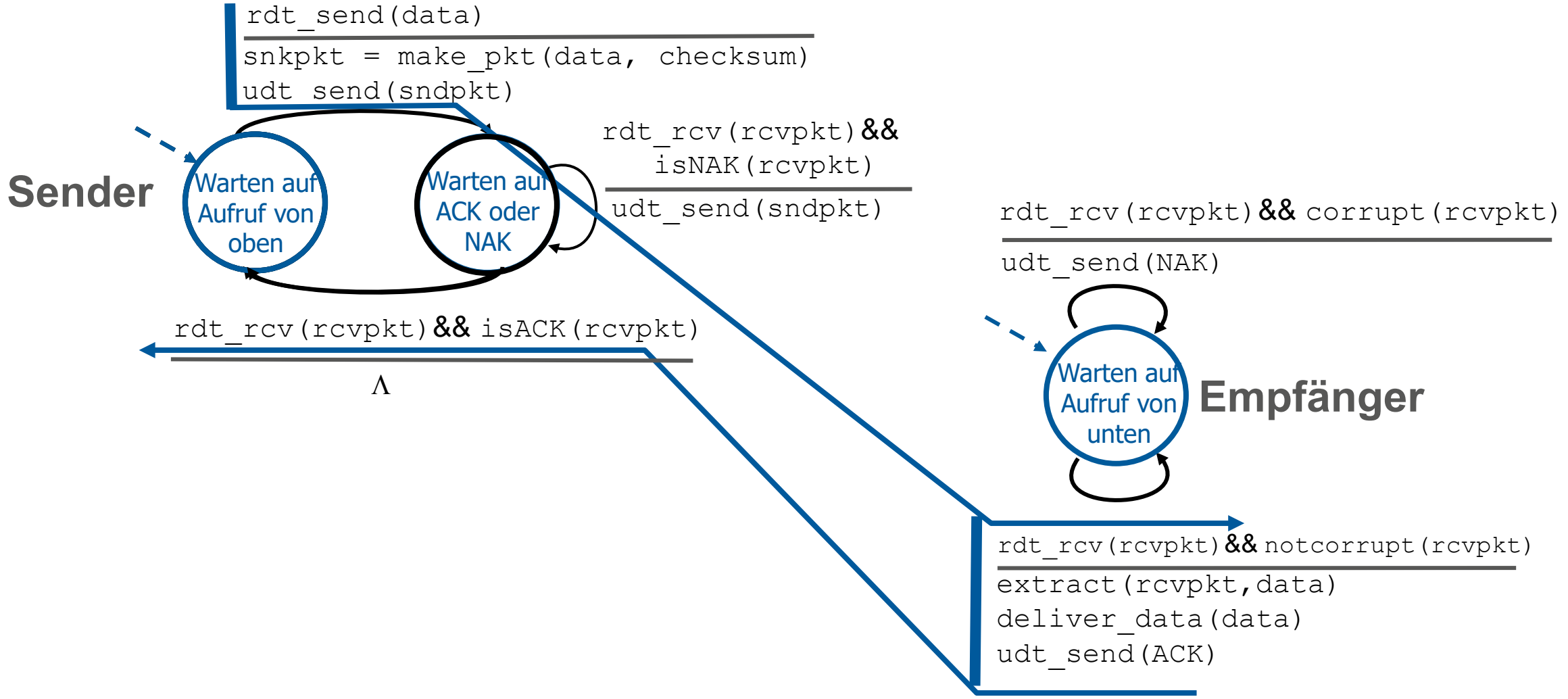




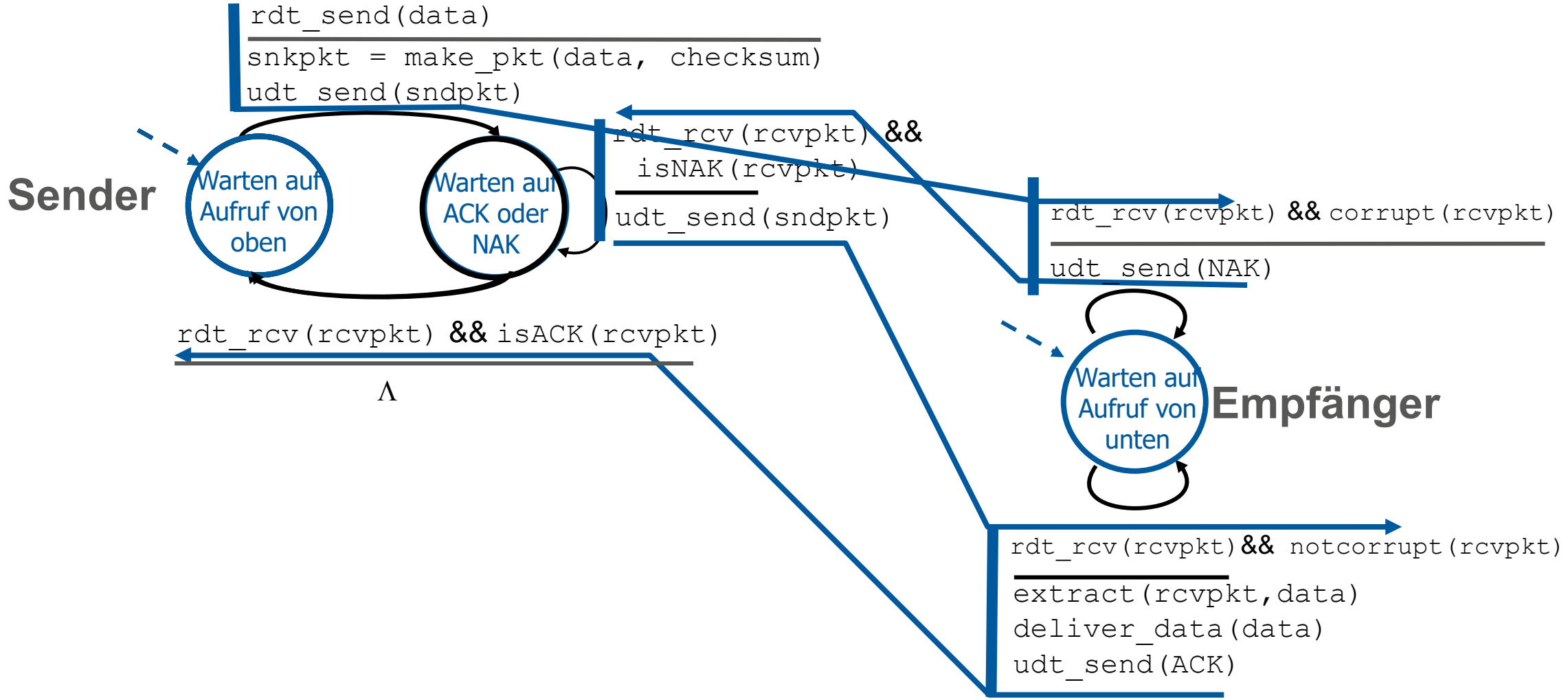
Hinweis: "Zustand" des Empfängers (hat der Empfänger meine Nachricht korrekt erhalten?) ist dem Sender nicht bekannt außer irgendwie vom Empfänger zum Sender kommuniziert

- Daher brauchen wir ein Protokoll!

rdt2.0: Betrieb ohne Fehler



rdt2.0: Fehlerhaftes Paket-Szenario



Was passiert wenn ACK/NAK fehlerhaft sind?

- Sender weiß nicht, was am Empfänger passiert ist
- Kann nicht einfach nochmal schicken: mögliche Dublette

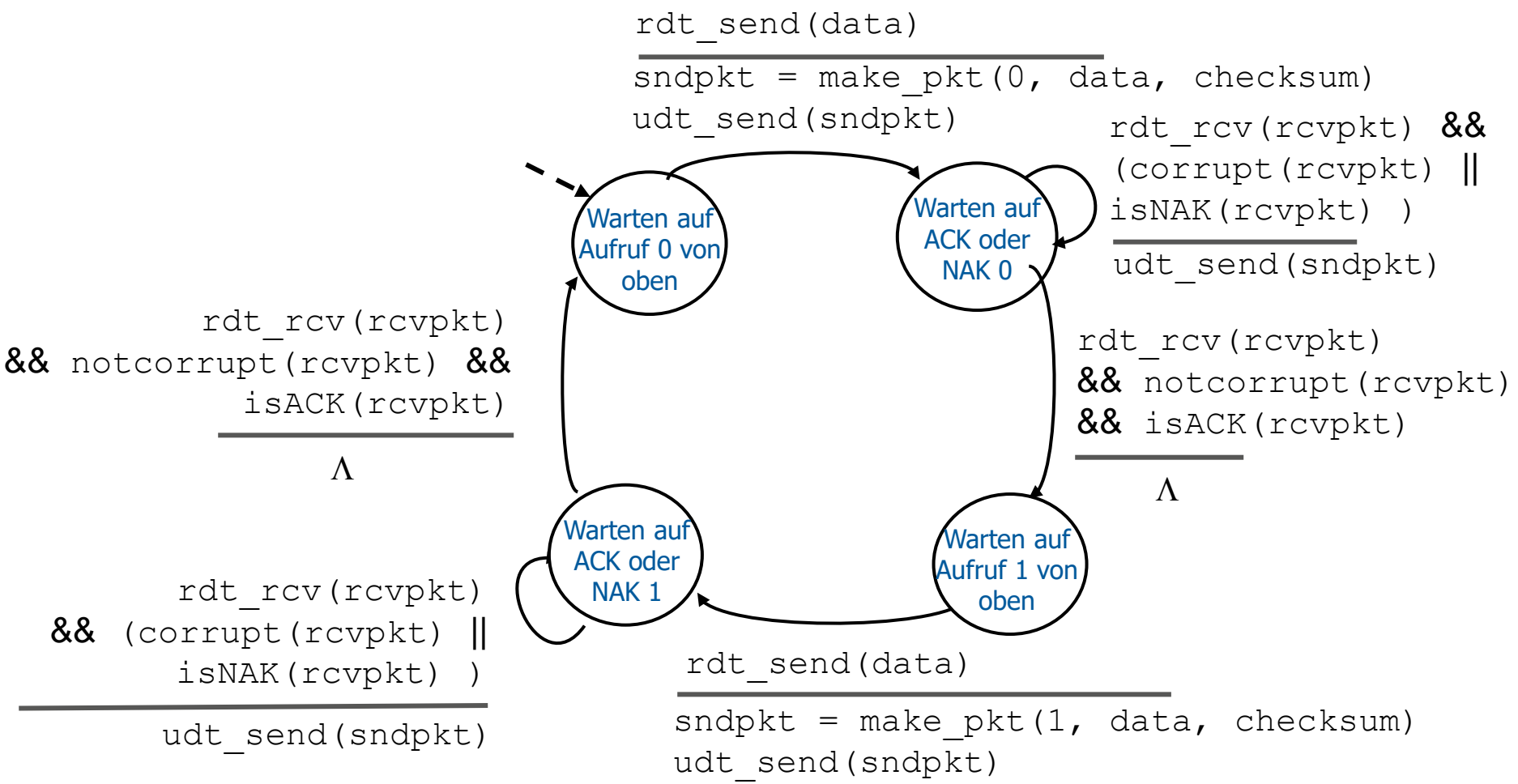
Behandeln von Dubletten:

- Sender sendet aktuelles Paket erneut, falls ACK/NAK fehlerhaft
- Sender fügt **Sequenznummer** zu jedem Paket hinzu
- Empfänger verwirft Dublette

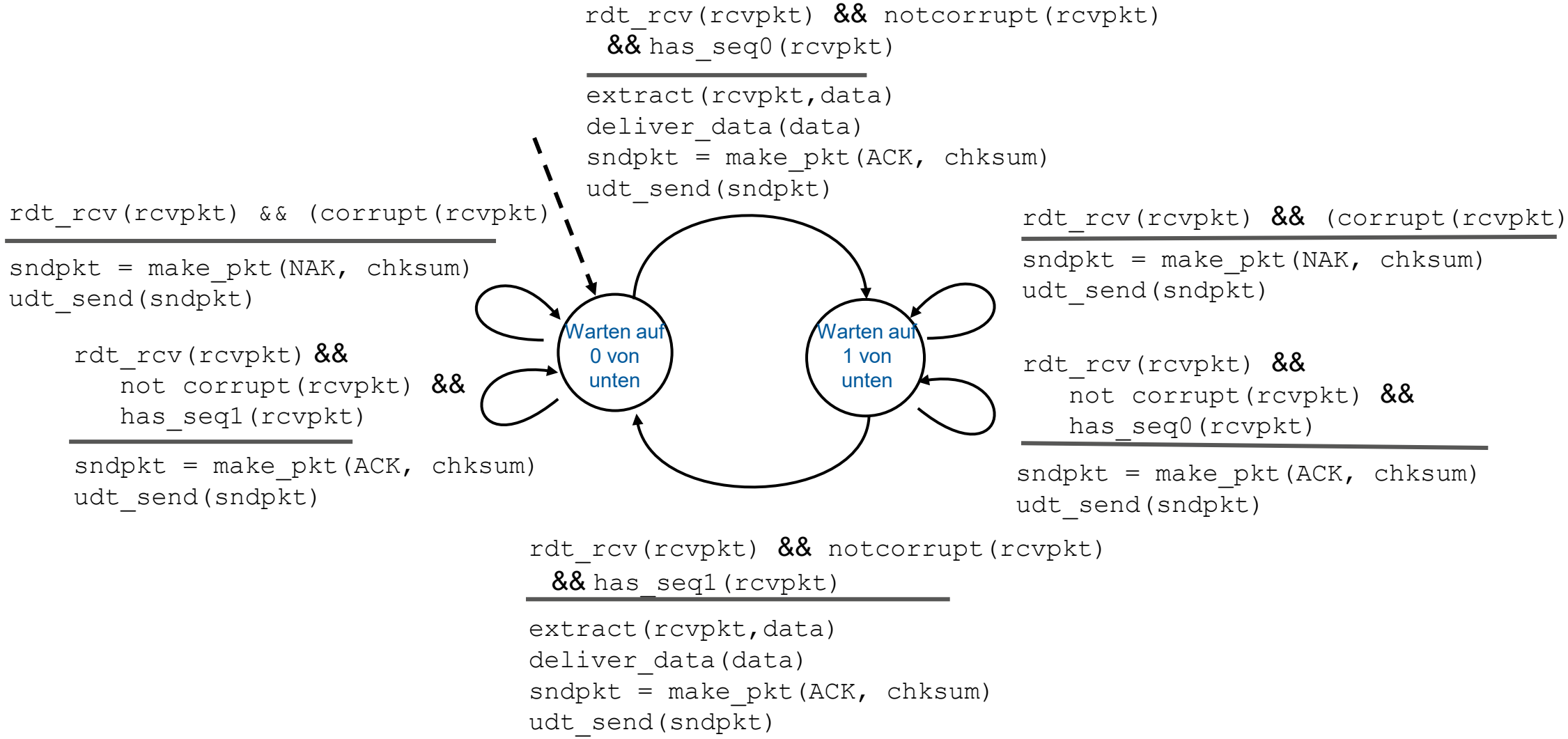
Stop and Wait

Sender sendet ein Paket und wartet dann auf die Antwort des Empfängers

rdt2.1: Sender, Behandeln fehlerhafter ACK/NAKs



rdt2.1: Empfänger, Behandeln fehlerhafter ACK/NAKs



rdt2.1: Diskussion

Sender:

- Sequenznr. zu Paket hinzugefügt
- Zwei Squenznr. (0,1) reichen aus. Warum?
- muss überprüfen, ob empfangenes ACK/NAK fehlerhaft
- Doppelt so viele Zustände
  - Zustand muss sich “merken” ob das “erwartete” Paket die Sequenznr. 0 oder 1 haben sollte

Empfänger:

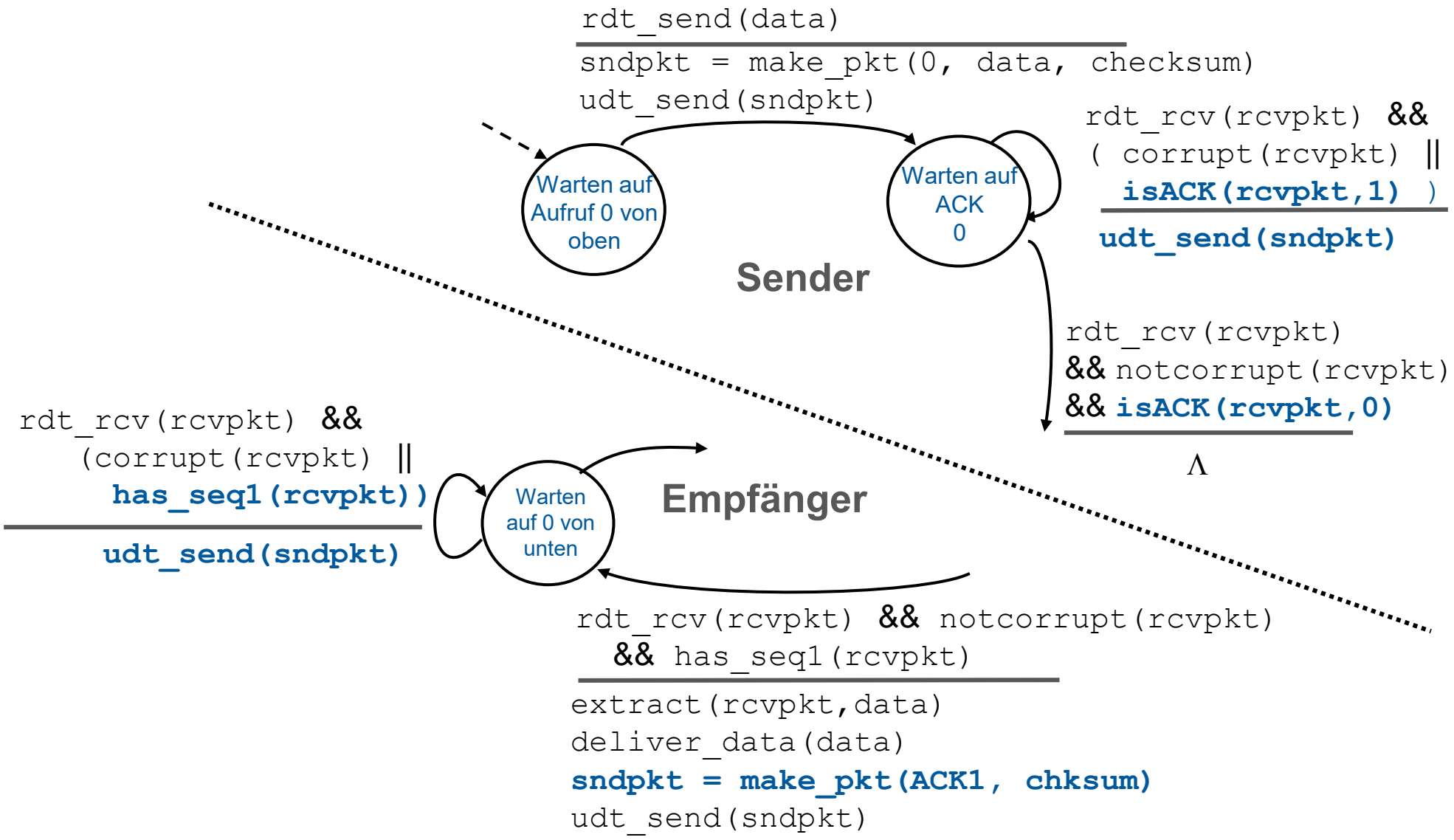
- muss überprüfen, ob empfangenes Paket eine Dublette ist
  - Zustand gibt an, ob 0 oder 1 als Paketsequenznr. erwartet wird
- Hinweis: Empfänger kann *nicht* wissen, ob sein letztes ACK/NAK am Sender korrekt empfangen wurde

rdt2.2: ein NAK-freies Protokoll

- Selbe Funktionalität wie rdt2.1, benutzt nur ACKs
- statt NAK, sendet der Empfänger ein ACK für das zuletzt korrekt empfangene Paket
  - Empfänger muss *explizit* Sequenznummer des bestätigten Pakets mitsenden
- Ein doppeltes ACK beim Sender resultiert in derselben Aktion wie ein NAK: *erneutes Senden des aktuellen Pakets*

TCP nutzt diesen NAK-freien Ansatz

rdt2.2: Sender, Empfänger Zustandsfragmente



Neue Kanalannahme: der Kanal kann auch Pakete *verlieren* (Daten, ACKs)

- Checksumme, Sequenznr., ACKs, Übertragungswiederholungen helfen…. reichen aber nicht ganz aus

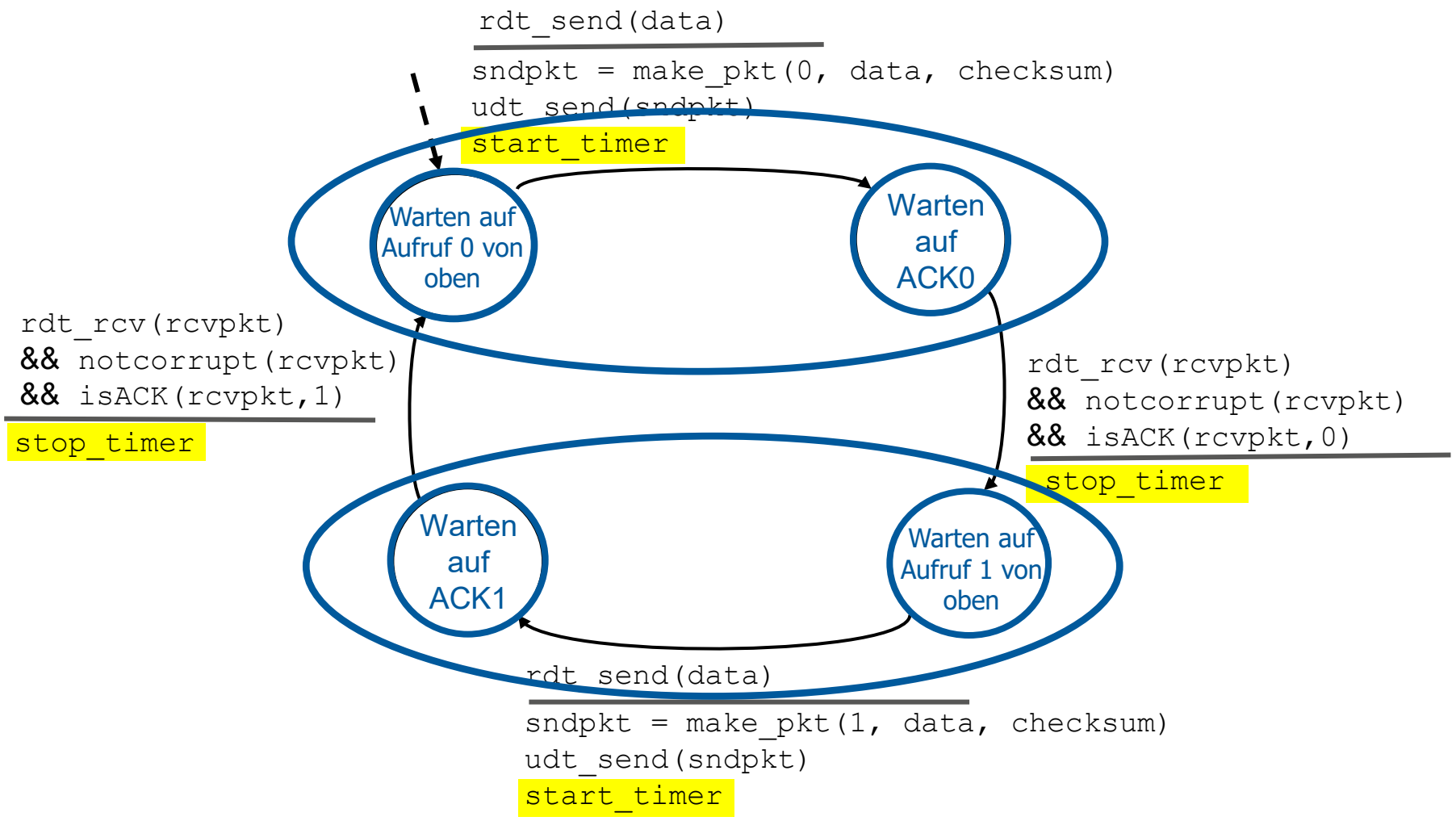
Frage: Wie behandeln *Menschen* verlorene Worte im Gespräch?

Ansatz: Sender wartet eine “vernünftige” Zeitspanne auf ein ACK

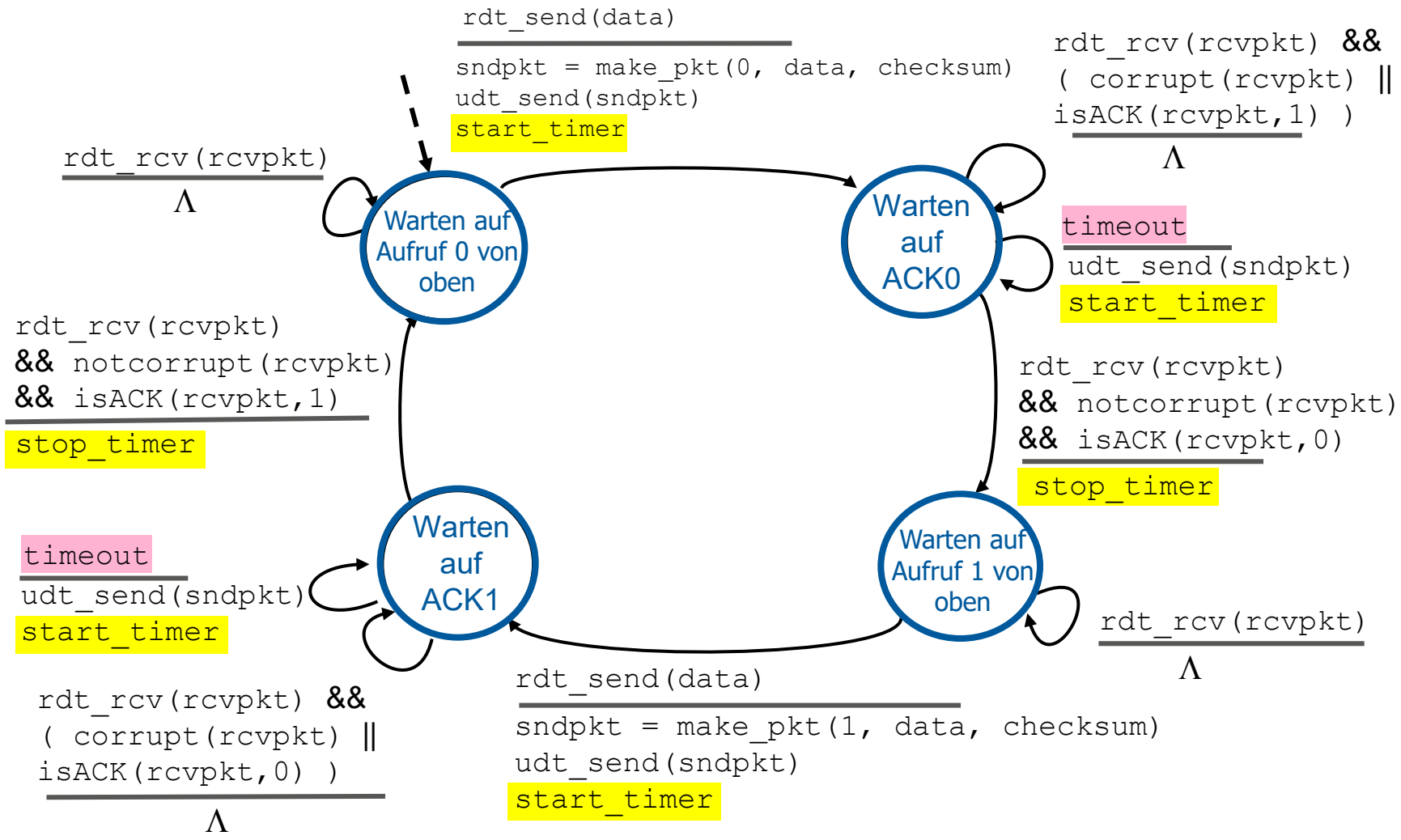
- Wiederholungsübertragung falls kein ACK in dieser Zeit empfangen wird
- falls Paket (oder ACK) nur verzögert (nicht verloren) ist:
  - Wiederholungsübertragung wird eine Dublette sein, aber die Sequenznummern beheben dies bereits!
  - Empfänger muss die Sequenznummer des bestätigten Pakets angeben
- Nutzen von Timer, um nach “vernünftiger” Zeitspanne zu unterbrechen

Timeout

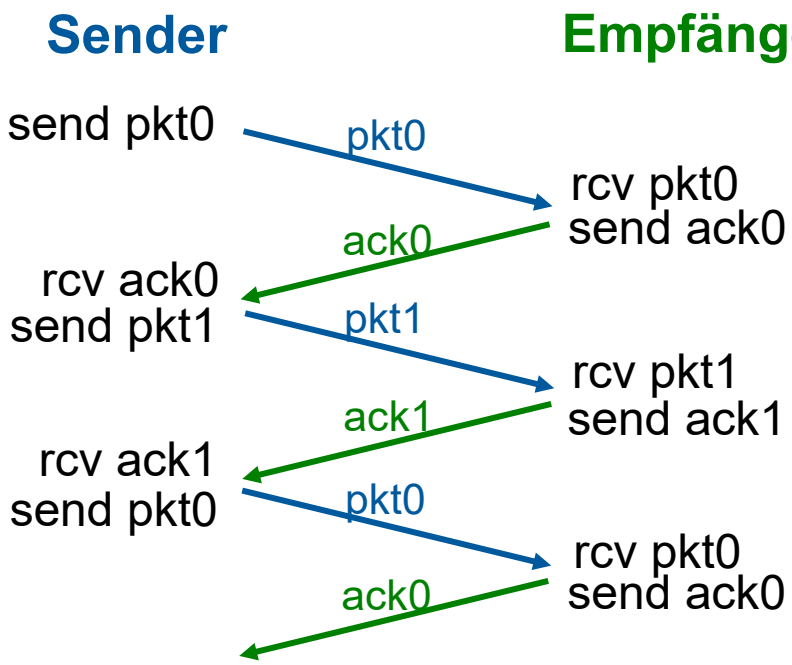
rdt3.0 Sender



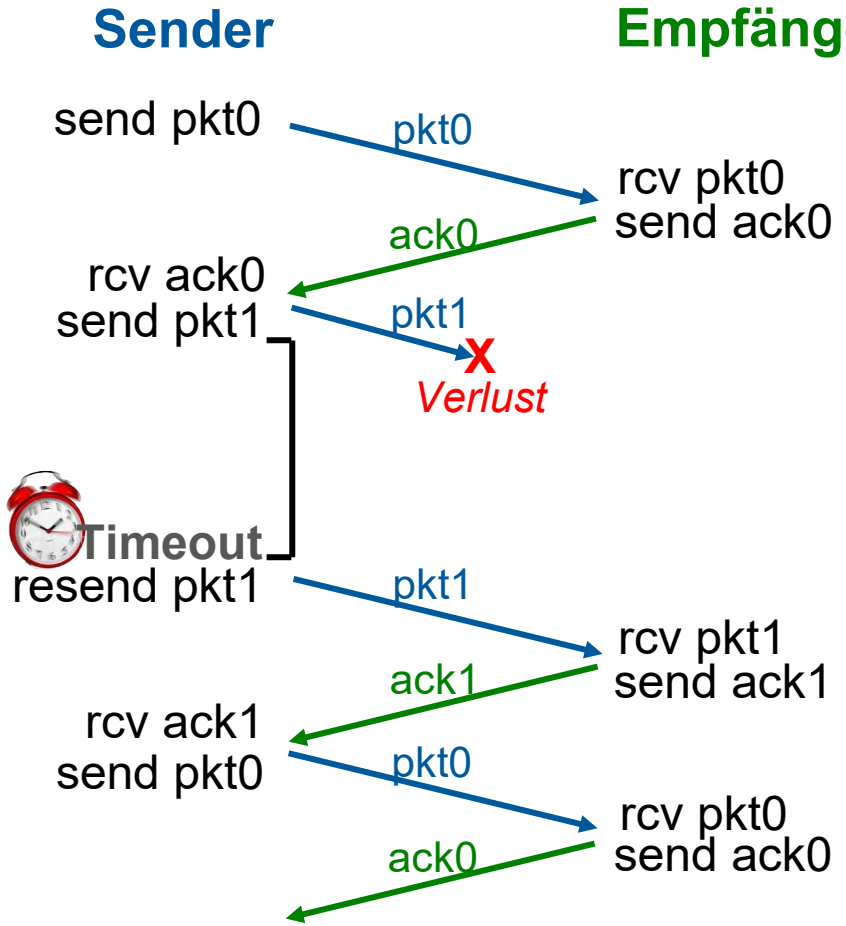
rdt3.0 Sender



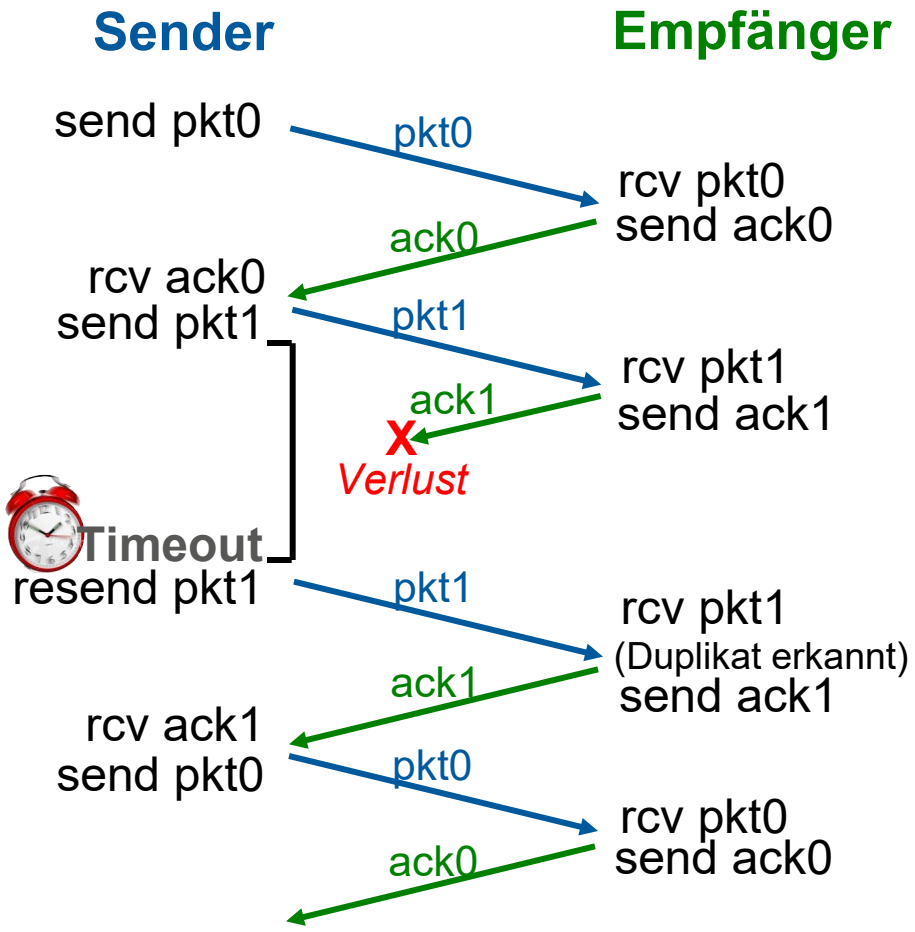
rdt3.0 in Aktion



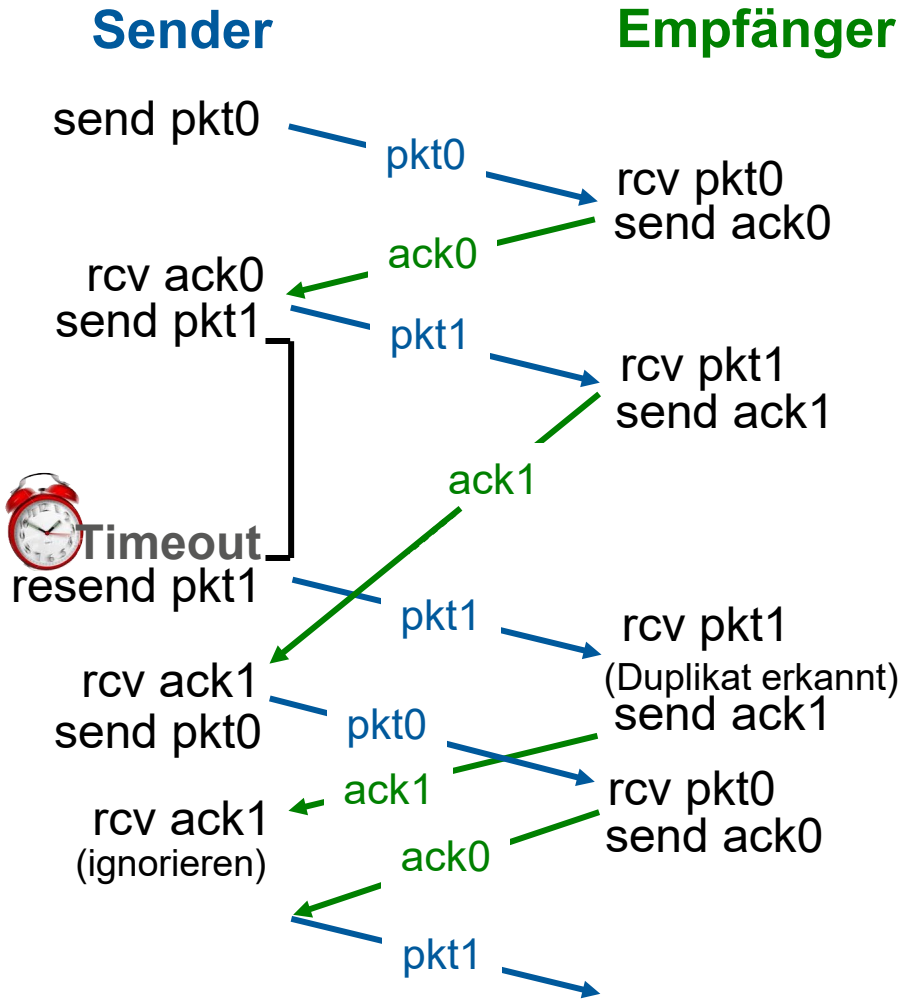
(a) Kein Verlust



(b) Paketverlust



(c) ACK-Verlust



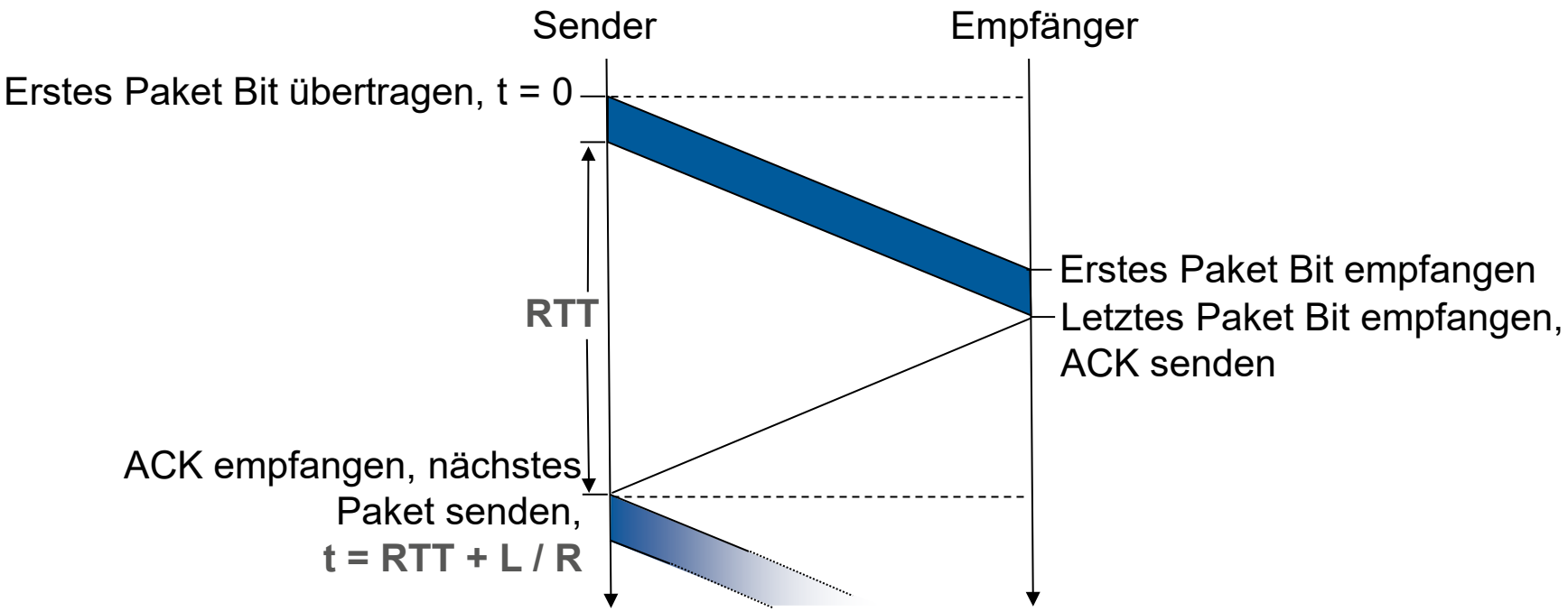
(d) Zu frühes Timeout/ verzögertes ACK

Leistung von rdt3.0 (Stop-and-Wait)

- U_{Sender}: **Auslastung** – Zeitanteil, den der Sender sendet
- Beispiel: 1 Gbit/s Link, 15 ms Verzögerung, 8000 Bit Paket
  - Zeit für das übertragen des Pakets auf den Kanal

$$D_{trans} = \frac{L}{R} = \frac{8000 \text{ Bits}}{10^9 \text{ Bit/s}} = 8 \text{ µsec}$$

rdt3.0: Stop-and-Wait Betrieb

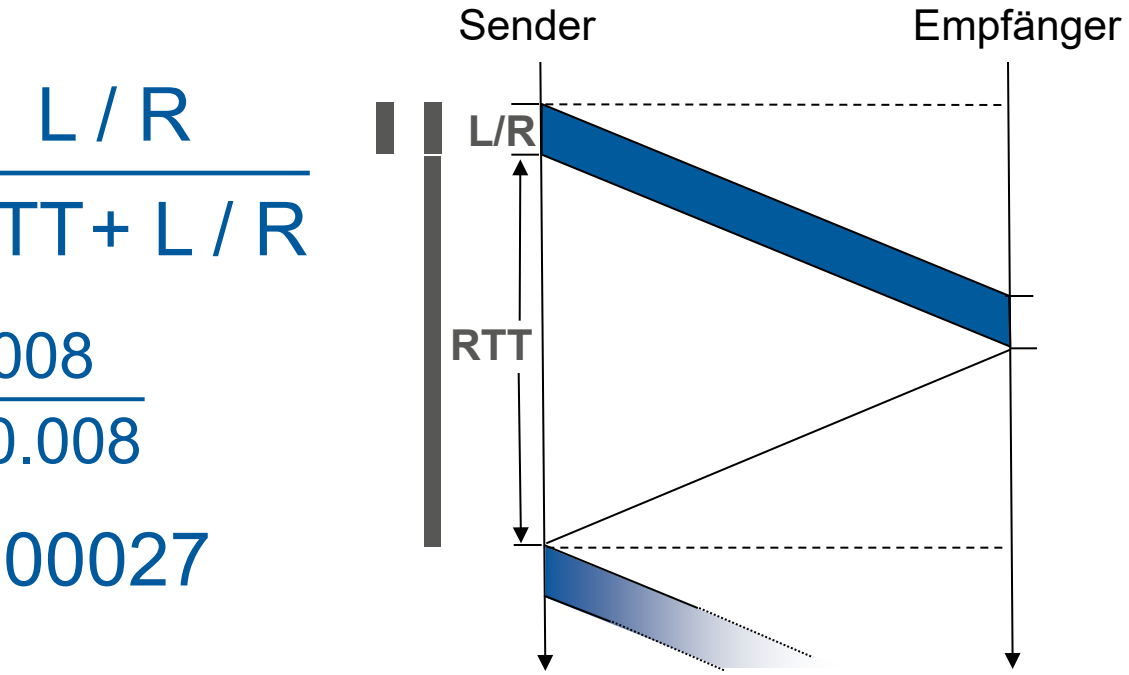


rdt3.0: Stop-and-Wait Betrieb

$$U_{Sender} = \frac{L / R}{RTT + L / R}$$

$$= \frac{.008}{30.008}$$

$$= 0.00027$$

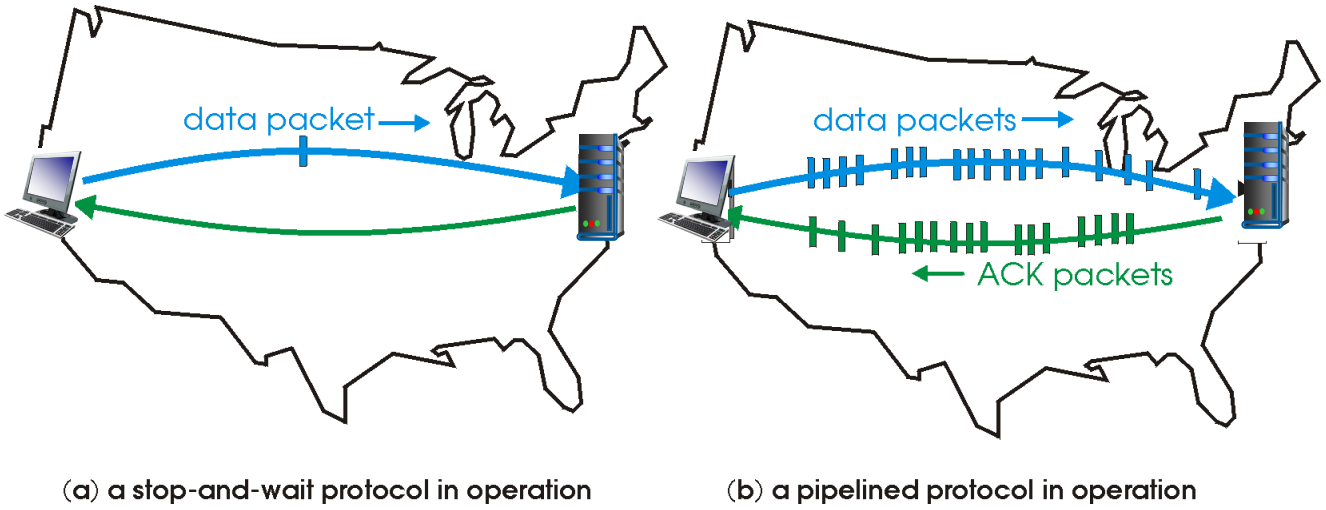


- Das rdt 3.0 Protokoll zeigt eine sehr schlechte Leistung!
- Das Protokoll beschränkt die Leistung der darunterliegenden Infrastruktur (Kanal)

rdt3.0: Pipeline Protokoll Betrieb

Pipelining: Sender erlaubt mehrere Pakete “im Flug”, die noch bestätigt werden müssen

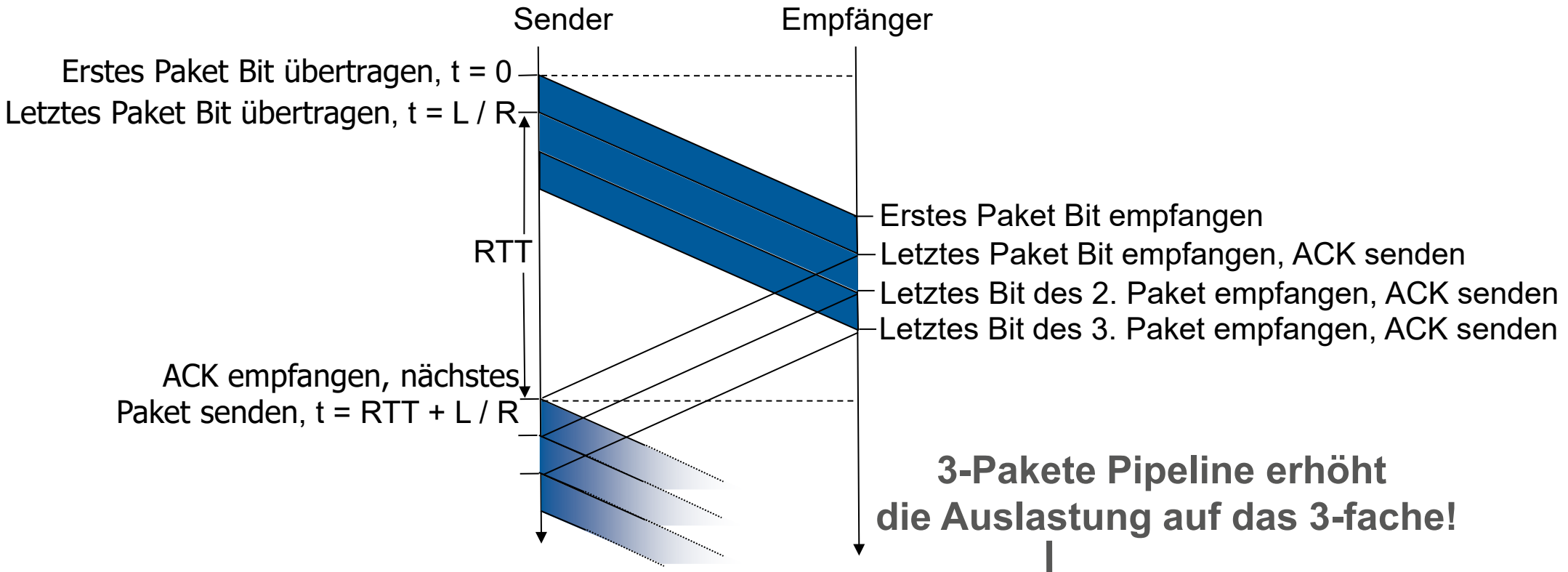
- Bereich der Sequenznummern muss vergrößert werden
- Puffern am Sender und/oder Empfänger



(a) a stop-and-wait protocol in operation

(b) a pipelined protocol in operation

Pipelining: Erhöhte Auslastung

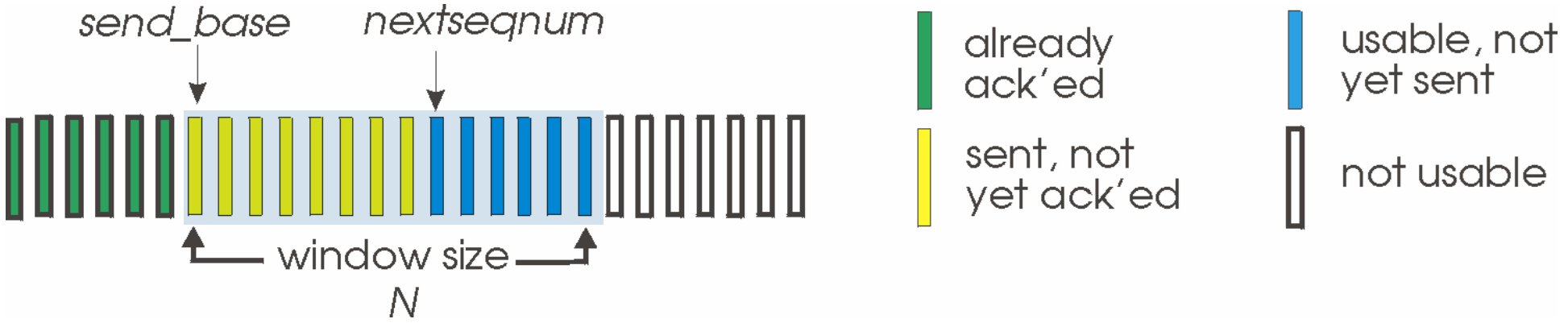


3-Pakete Pipeline erhöht die Auslastung auf das 3-fache!

$$U_{Sender} = \frac{3L / R}{RTT + L / R} = \frac{.0024}{30.008} = 0.00081$$

Go-Back-N: Sender

- Sender: “Fenster” von bis zu N, aufeinanderfolgend übertragener, aber ungeACKter Pakete
  - k-Bit Sequenznummer im Paketheader

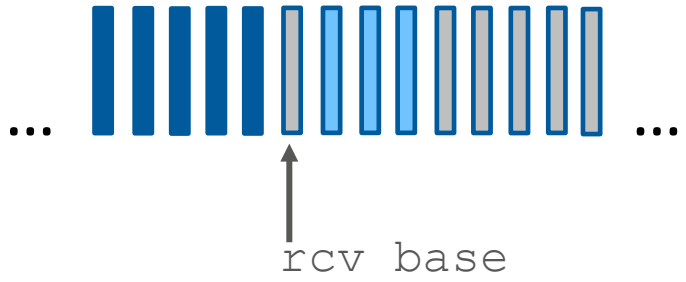


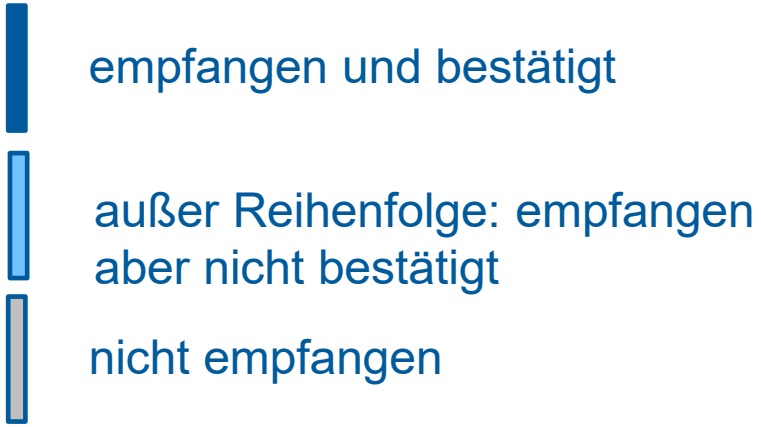
- **kumulatives ACK:** ACK(*n*): bestätigt alle Pakete bis einschließlich Sequenznr. n
  - bei Empfang von ACK(*n*): bewegen des Fensterbeginns zu *n+1*
- Timer für ältestes, unbestätigtes Paket
- **Timeout(n):** Übertragungswiederholung von Paket n und allen Paketen mit höherer Sequenznummer im Fenster

Go-Back-N: Empfänger

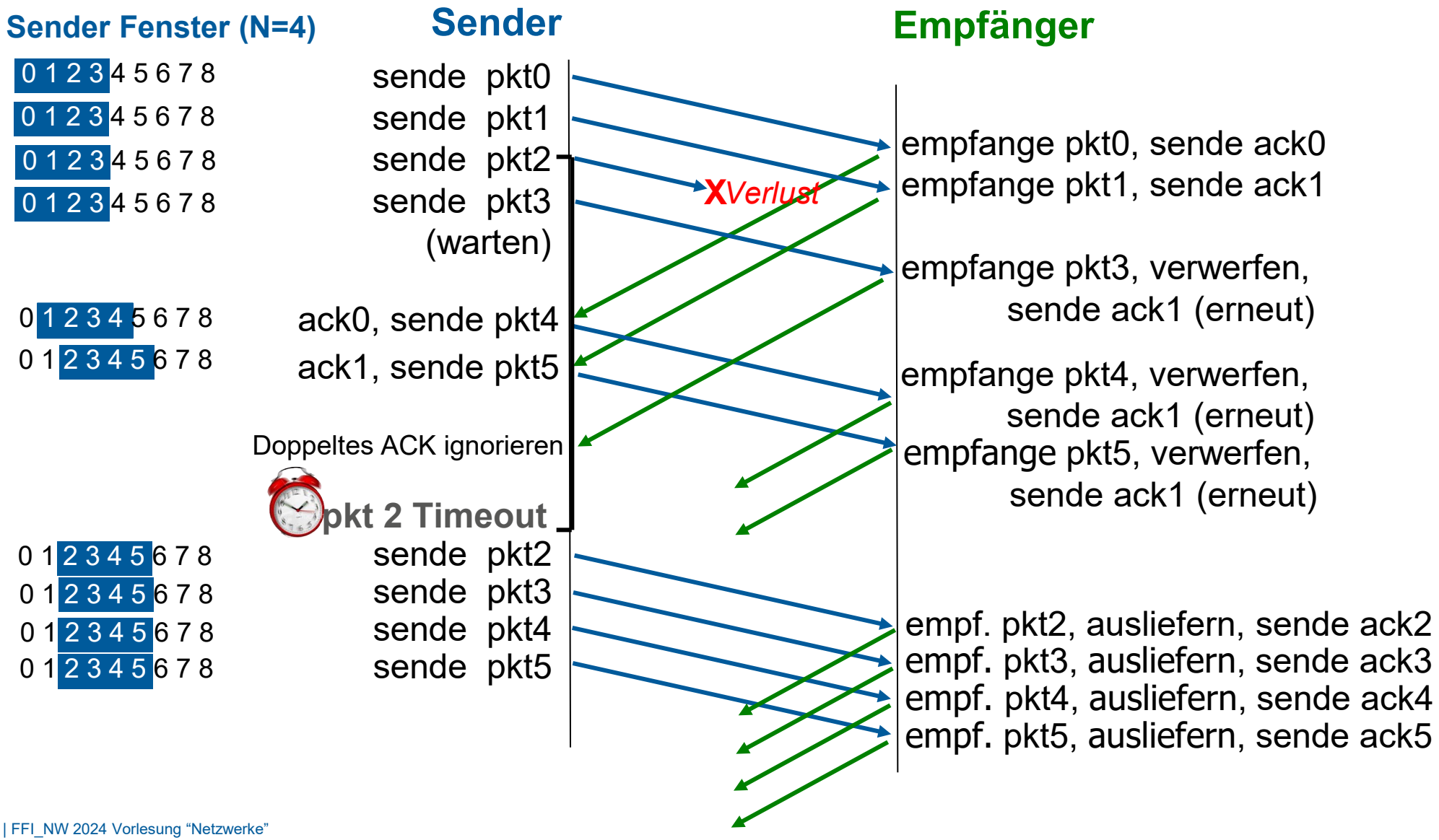
- nur ACKs: ACK wird immer für alle korrekt empfangenen Pakete mit Sequenznummer **in korrekter Reihenfolge** gesendet
  - kann ACK Dupletten erzeugen
  - muss sich nur **rcv_base** merken
- bei Empfang eines Paketes außer Reihenfolge:
  - verwerfen oder puffern: Implementierungsentscheidung
  - erneutes ACK für Paket mit höchster Sequenznummer in korrekter Reihenfolge

Empfängersicht des Sequenznummer-Raumes





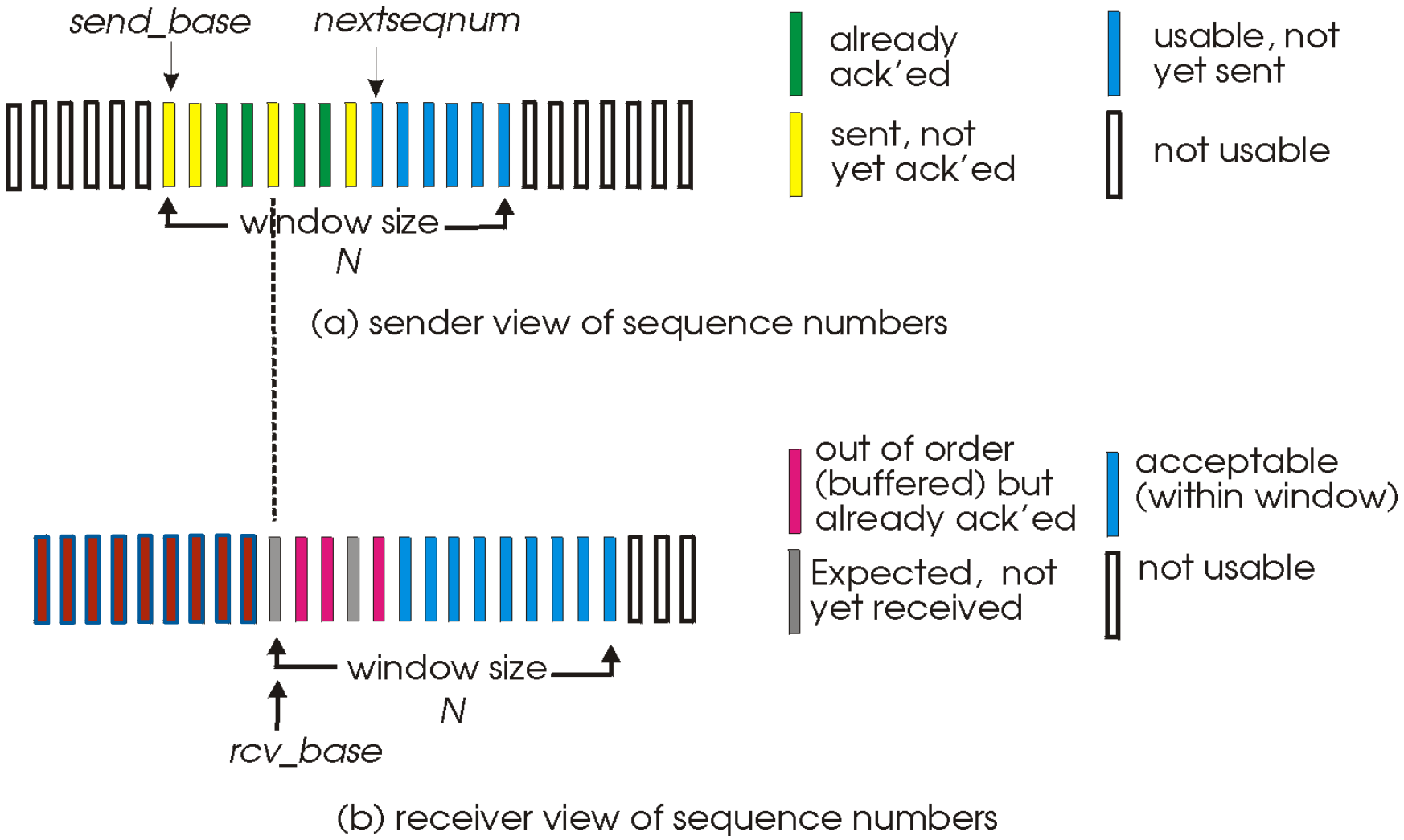
Go-Back-N in Aktion



Selektives Wiederholen

- Empfänger bestätigt **individuell** alle korrekt empfangenen Pakete
  - puffert Pakete, sofern benötigt, für schlussendliche Auslieferung an höhere Schicht
- Sender lässt ungeACKte Pakete individuell aus-timen/überträgt sie erneut
  - Sender hält Timer für jedes unbestätigte Paket
- Sendefenster
  - *N* aufeinanderfolgende Sequenznummern
  - beschränkt die Zahl an versendeten Sequenznummern, unbestätigten Paketen

Selektives Wiederholen: Sender-, Empfängerfenster



Sender

Daten von oben:

- Wenn nächste Sequenznummer verfügbar im Fenster, sende Paket

Timeout(*n*):

- Paket *n* erneut senden, Timer neustarten

ACK(*n*) in [sendbase,sendbase+N]:

- Paket *n* als empfangen markieren
- falls n kleinstes ungeACKtes Paket, Fenster auf nächste ungeACKte Sequenznummer setzen

Empfänger

Paket *n* in [rcvbase, rcvbase+N-1]

- Senden von ACK(*n*)
- Außer Reihenfolge: puffern
- In Reihenfolge: ausliefern (auch von gepufferten Paketen in Reihenfolge), Fenster auf nächstes noch nicht empfangene Paket setzen

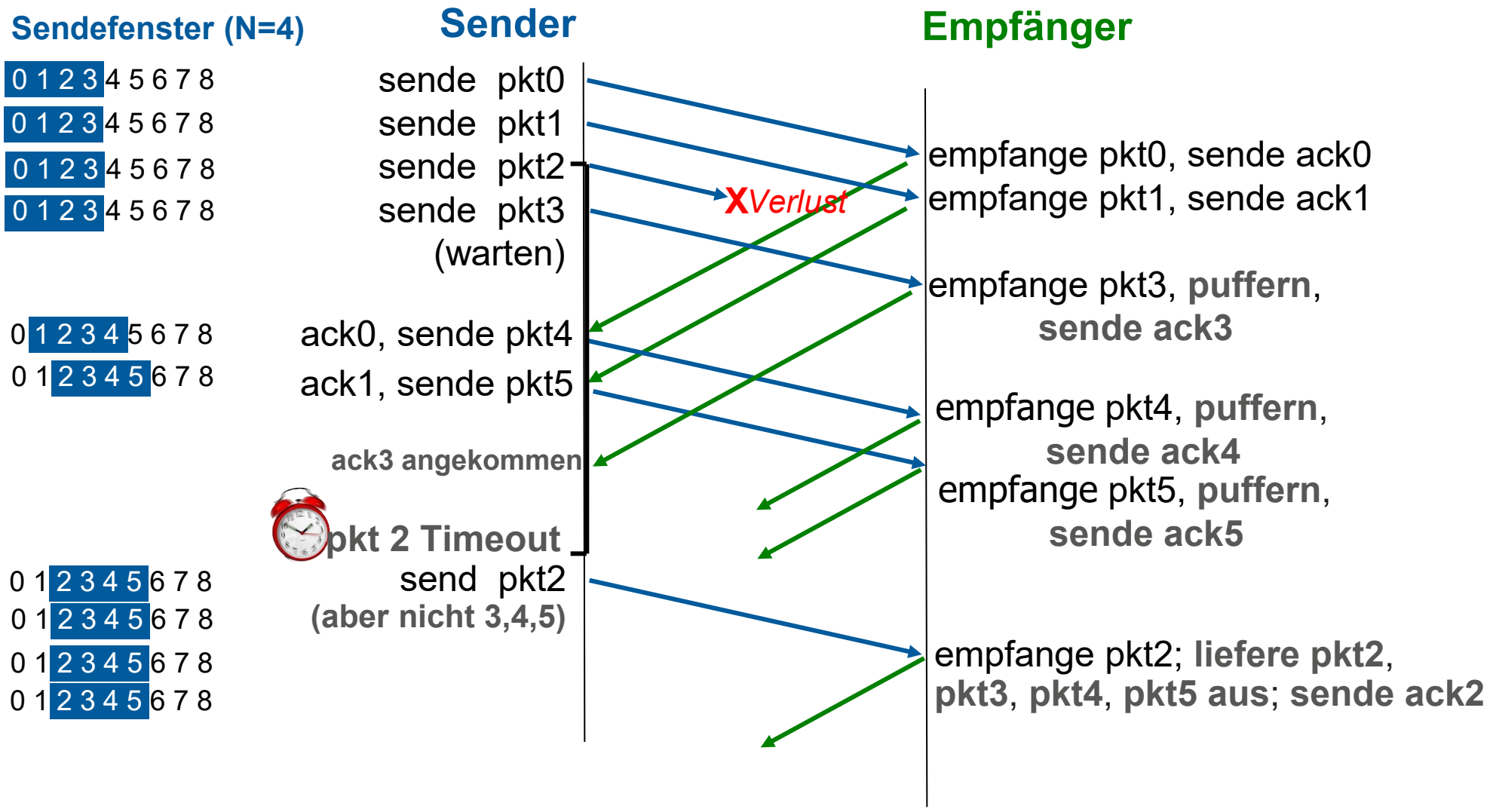
Paket *n* in [rcvbase-N,rcvbase-1]

- ACK(*n*)

ansonsten:

- ignorieren

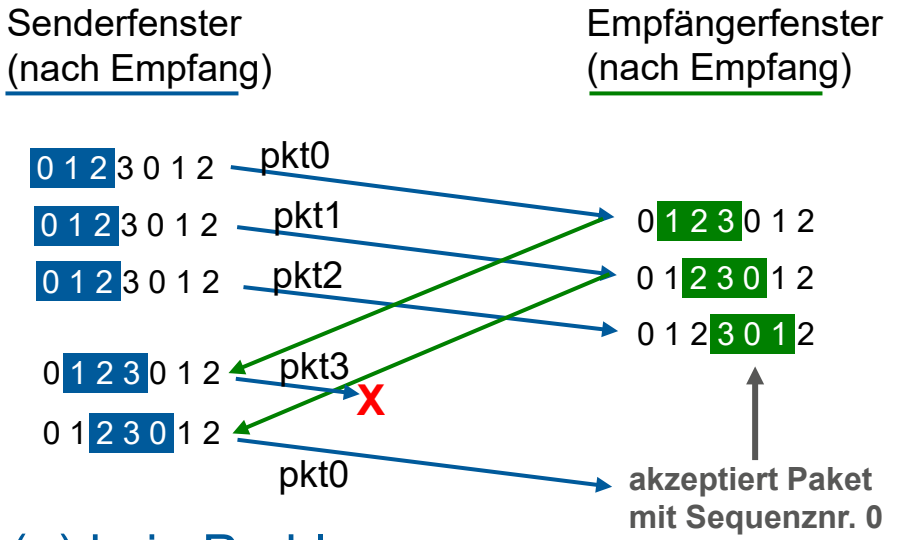
Selektives Wiederholen in Aktion



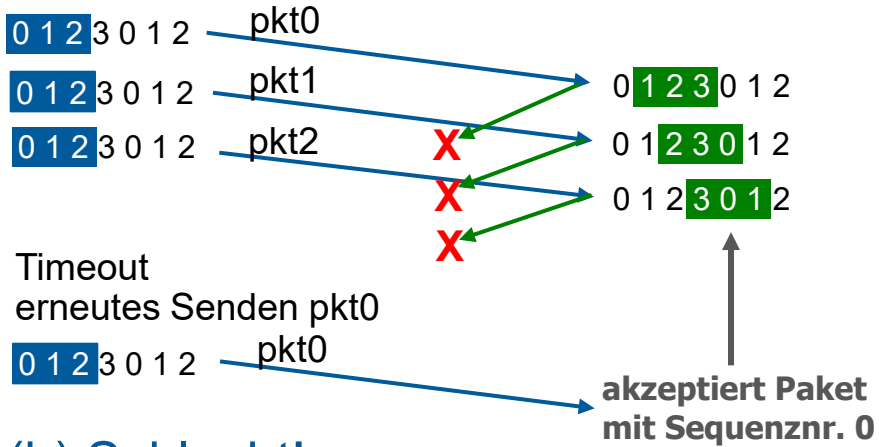
Selektives Wiederholen: ein Dilemma!

Beispiel:

- Sequenznr.: 0, 1, 2, 3 (Basis 4 Zählung)
- Fenstergröße=3



(a) kein Problem



(b) Schlecht!

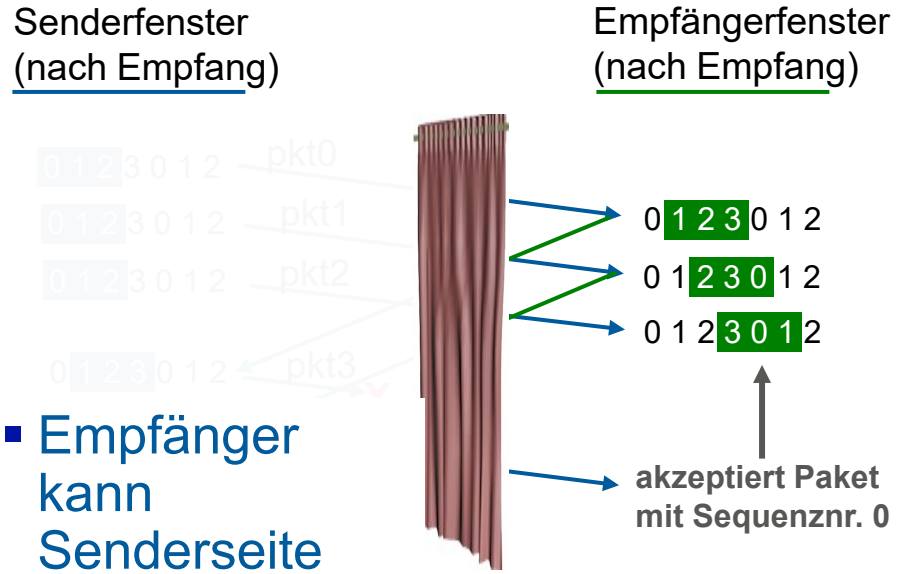
Selektives Wiederholen: ein Dilemma!

Beispiel:

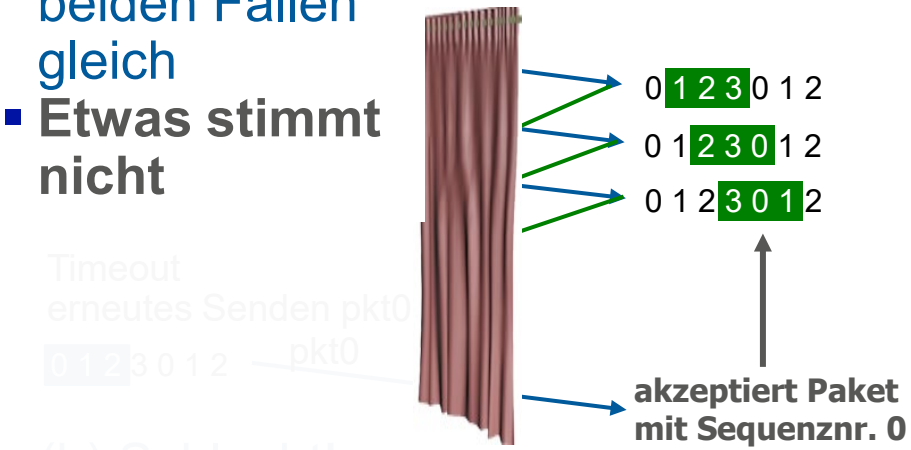
- Sequenznr.: 0, 1, 2, 3 (Basis 4 Zählung)
- Fenstergröße=3

Frage:

Welche Beziehung muss zwischen der Anzahl von Sequenznummer und der Fenstergröße bestehen, damit der Fall in Szenario b) vermieden werden kann?



- Empfänger kann Senderseite nicht sehen
- Empfänger verhalten in beiden Fällen gleich
- **Etwas stimmt nicht**



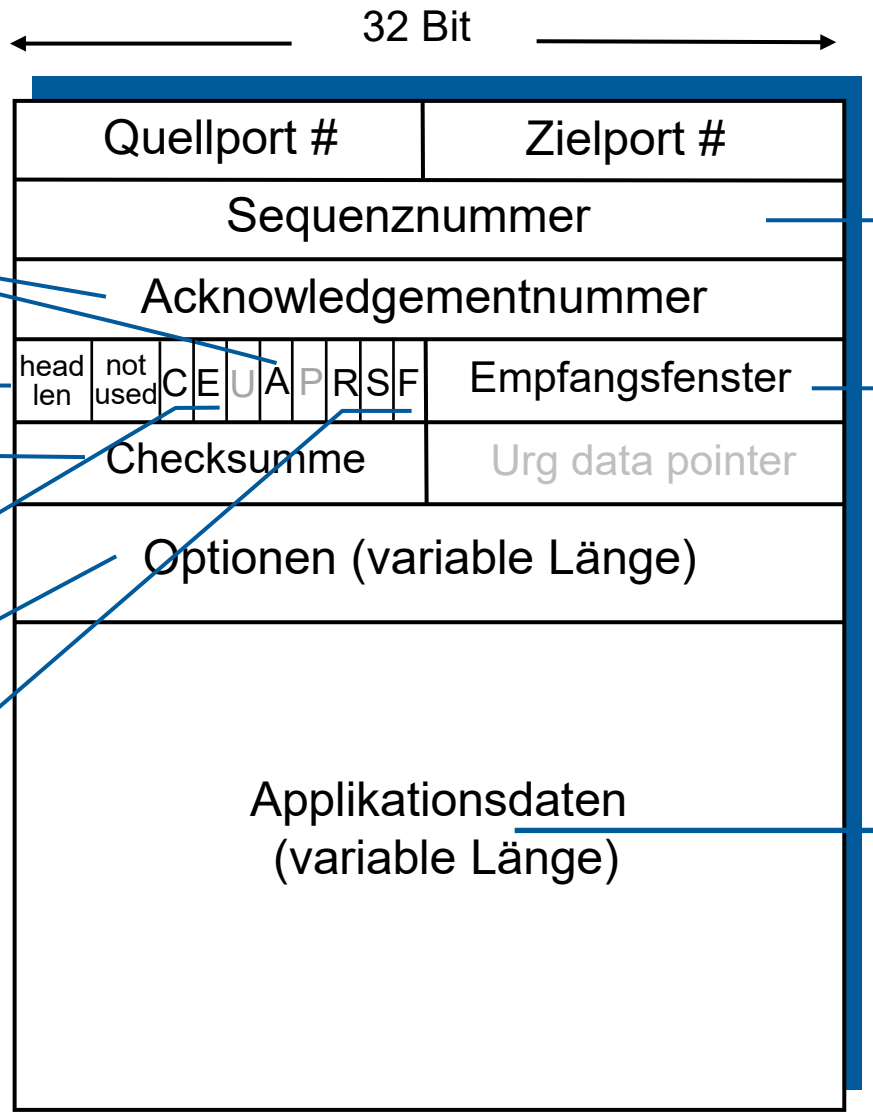
Transportschicht: Übersicht

- Dienste der Transportschicht
- Multiplexen und Demultiplexen
- Verbindungsloser Transport: UDP
- Prinzipien verlässlicher Datenübertragung
- **Verbindungsorientierter Transport: TCP**
  - Segmentstruktur
  - Zuverlässige Datenübertragung
  - Flusskontrolle
  - Verbindungsmanagement
- Prinzipien der Überlastkontrolle
- TCP Überlastkontrolle
- Evolution der Transportschicht Funktionen

TCP: Übersicht RFCs: 793,1122, 2018, 5681, 7323

- **Punkt-zu-Punkt:**
  - ein Sender, ein Empfänger
- **zuverlässiger, geordneter Byte Stream*:***
  - keine “Nachrichtengrenzen"
- **Full-duplex Daten:**
  - bi-direktionaler Datenfluss innerhalb derselben Verbindung
  - MSS: Maximum Segment Size
- **kumulative ACKs**
- **Pipelining:**
  - TCP Überlast- und Flusskontrolle bestimmen die Größe des Sendefensters
- **Verbindungs-orientiert:**
  - Verbindungsaufbau initialisiert Sender- und Empfängerzustand vor Datenübertragung
- **Fluss-Kontrolle:**
  - Der Sender überlastet den Empfänger nicht

TCP Segment Struktur



ACK: Sequenznr. Des nächsten erwarteten Byte; A-Bit: dies ist ein ACK

Länge (des TCP-Header)

Internet Checksumme

C,E: Überlastbenachrichtigung

TCP-Optionen

RST, SYN, FIN: Verbindungs-management

Segment Sequenznummer: zählt die Anzahl von Datenbytes im Bytestream (nicht Segmente!)

Flusskontrolle: Anzahl Bytes, die der Empfänger akzeptiert

Daten, die von der Applikation an den TCP-Socket gesendet wurden

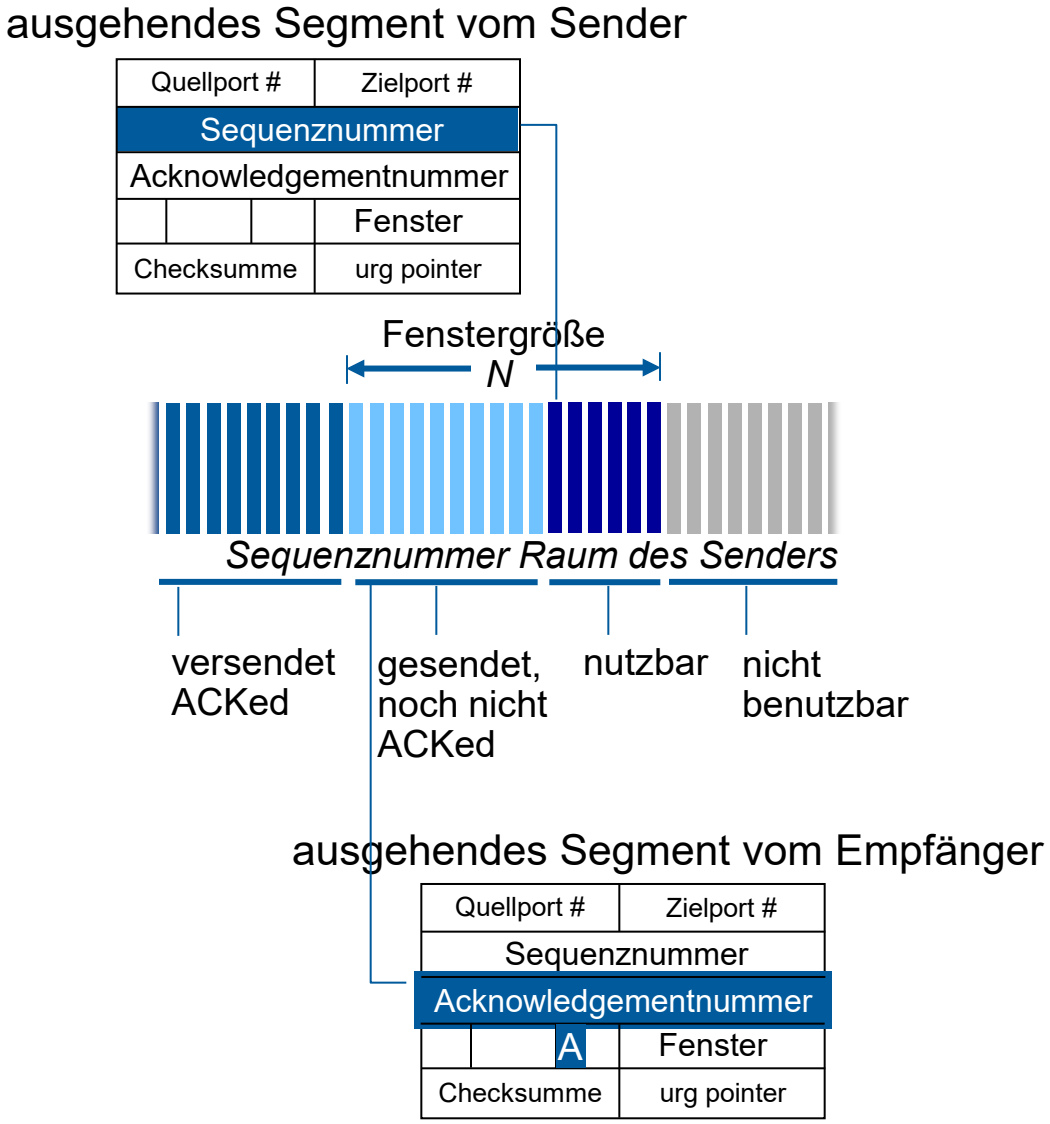
TCP-Sequenznummern, ACKs

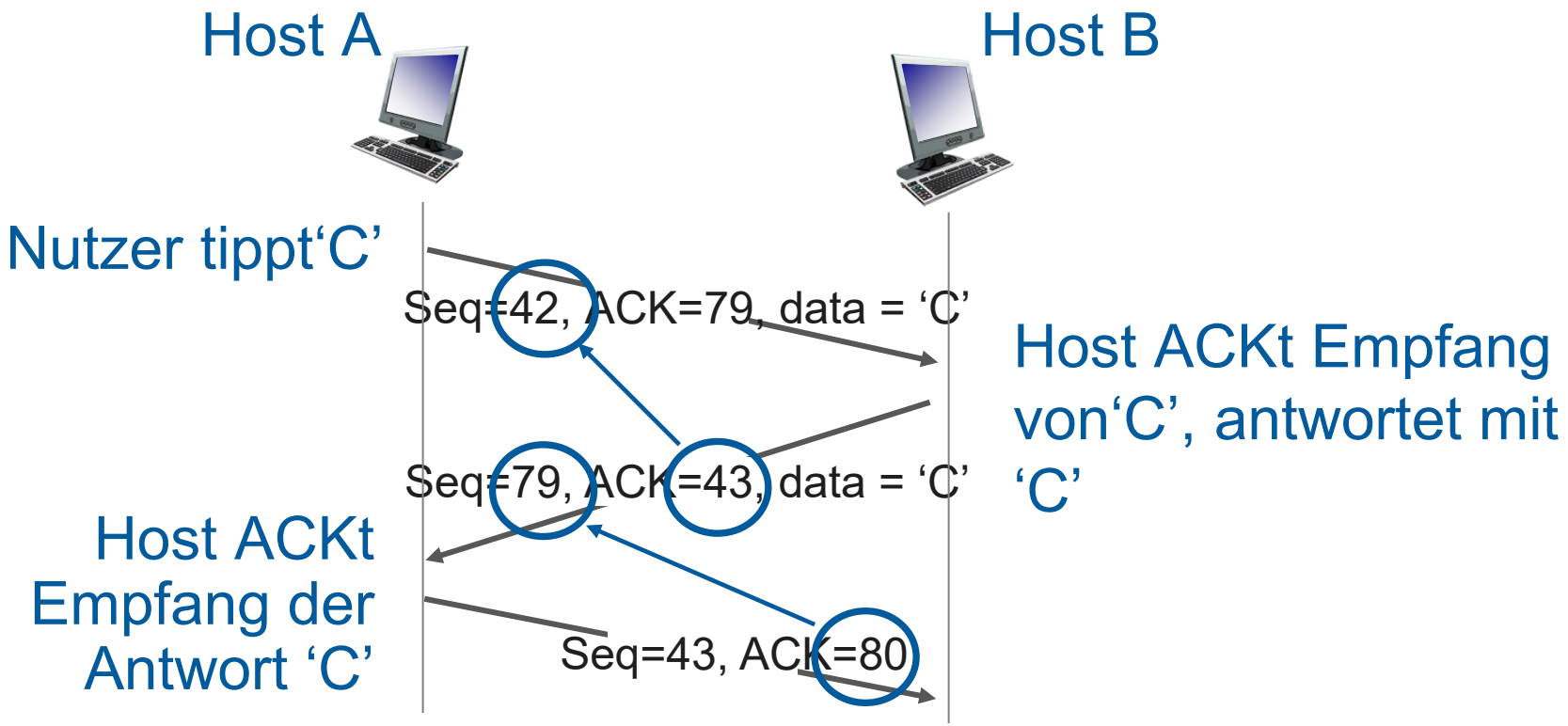
- **Sequenznummern:**
  - Byte Stream “Nummer” des ersten Byte der Daten im Segment

- **Acknowledgements:**
  - Sequenznummer des nächstes von der Gegenseite erwarteten Bytes
  - kumulatives ACK

Frage: wie behandelt der Empfänger Pakete außer Reihe

- **Antwort:** TCP spezifiziert das nicht, implementierungsabhängig





Einfaches Telnet Szenario

Frage: Wie wird die TCP Timeout Länge bestimmt?

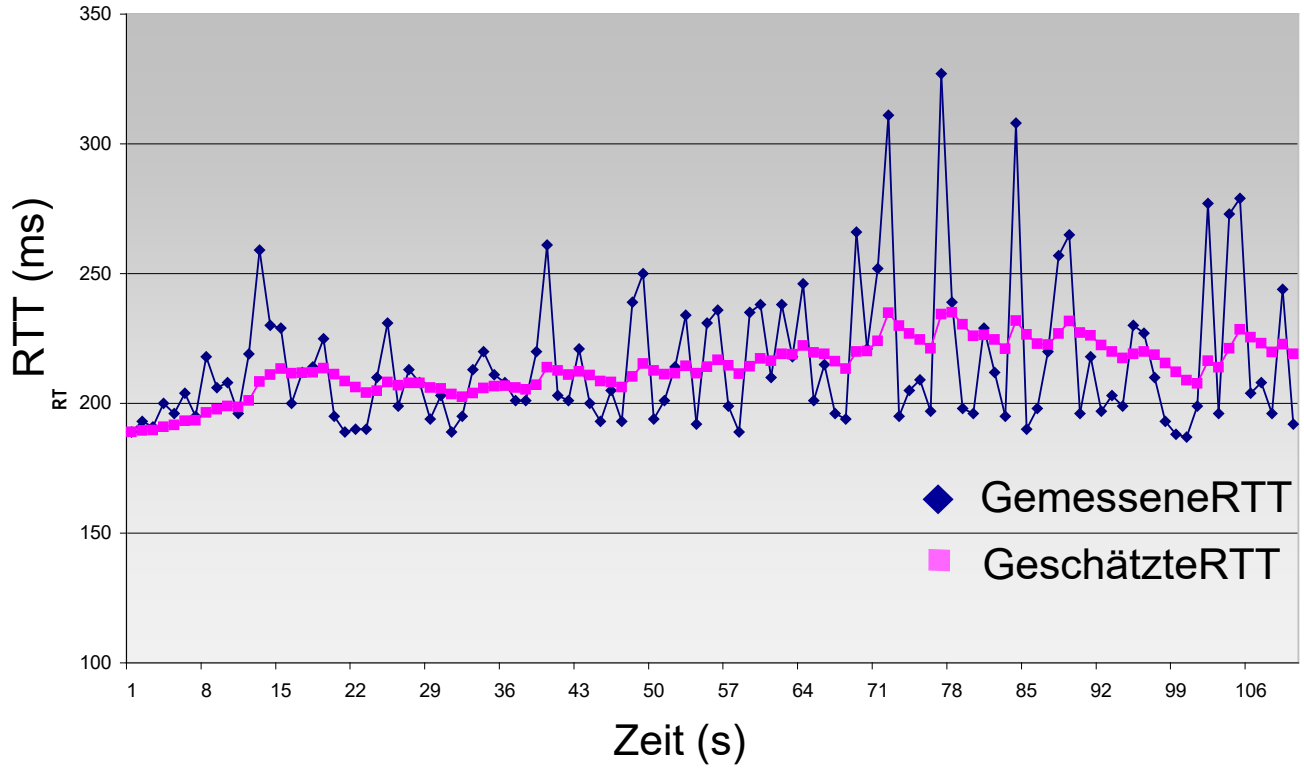
- länger als RTT, aber RTT variiert!
- **zu kurz:** verfrühter Timeout, unnötige Übertragungswiederholungen
- **zu lang:** langsame Reaktion auf Segmentverlust

Frage: Wie wird die RTT bestimmt?

- **GemesseneRTT:** Gemessene Zeit von Segmenttransmission bis ACK-Empfang
  - Wiederholungsübertragungen werden ignoriert
- **GemesseneRTT** variiert, “glattere” geschätzte RTT wünschenswert
  - Durchschnitt mehrerer **aktueller** Messungen, nicht nur der letzten **GemesseneRTT**

```
GeschätzteRTT = (1- α)*GeschätzteRTT + α*GemesseneRTT
```

- **E**xponential **W**eighted **M**oving **A**verage (**EWMA**)
- Einfluss vergangener Messungen schwindet exponentiell
- typischer Wert: α = 0.125



- Timeout Intervall: **`GeschätzteRTT`** plus “Sicherheitspuffer”
  - Bei hoher Variabilität der **`GeschätztenRTT:`** größerer Sicherheitspuffer benötigt

```
TimeoutIntervall = GeschätzteRTT + 4*DevRTT
```

Geschätzte RTT **“Sicherheitspuffer”**

- **`DevRTT:`** EWMA der Abweichung der **`GemessenenRTT`** von der **`GeschätztenRTT:`**

$$DevRTT = (1-\beta)*DevRTT + \beta*|GemesseneRTT-GeschätzteRTT|$$

(typisch, $\beta = 0.25$)

Ereignis: Daten von Applikation empfangen

- Erstellen von Segment mit Sequenznr.
- Sequenznr. ist die Byte-Stream Nummer des ersten Datenbytes im Segment
- Timer starten, sofern er nicht läuft
  - Timer läuft für das älteste ungeACKte Segment
  - Ablaufintervall: `TimeOutIntervall`

Ereignis: Timeout

- Segment, dass den Timeout verursacht hat erneut übertragen
- Timer neustarten

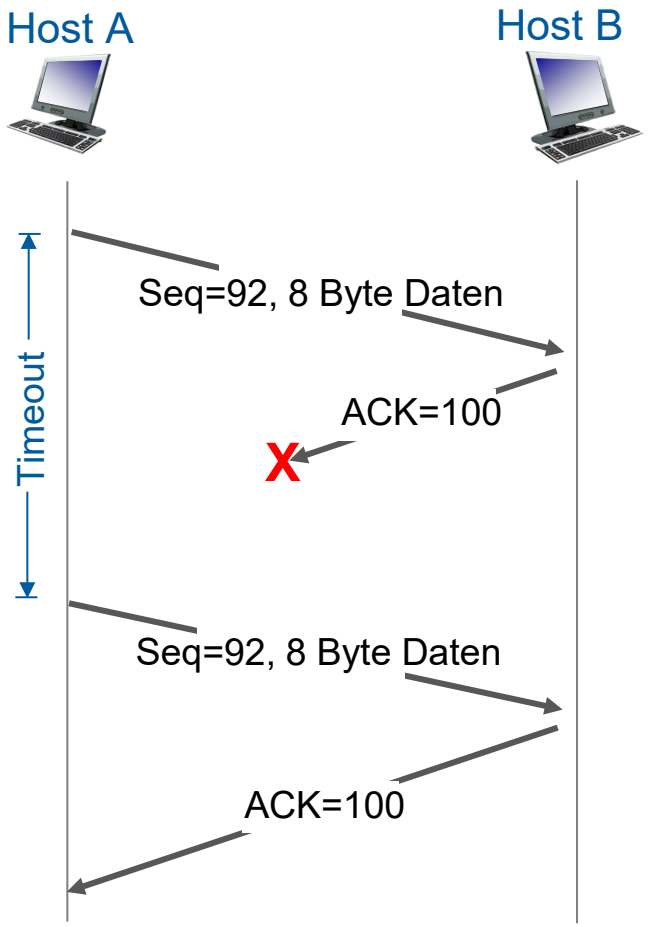
Ereignis: ACK empfangen

- falls ACK ein bisher ungeACKtes Segment bestätigt
  - Aktualisieren was bereits bestätigt ist
  - Starten des Timers, falls es noch ungeACKte Segmente gibt

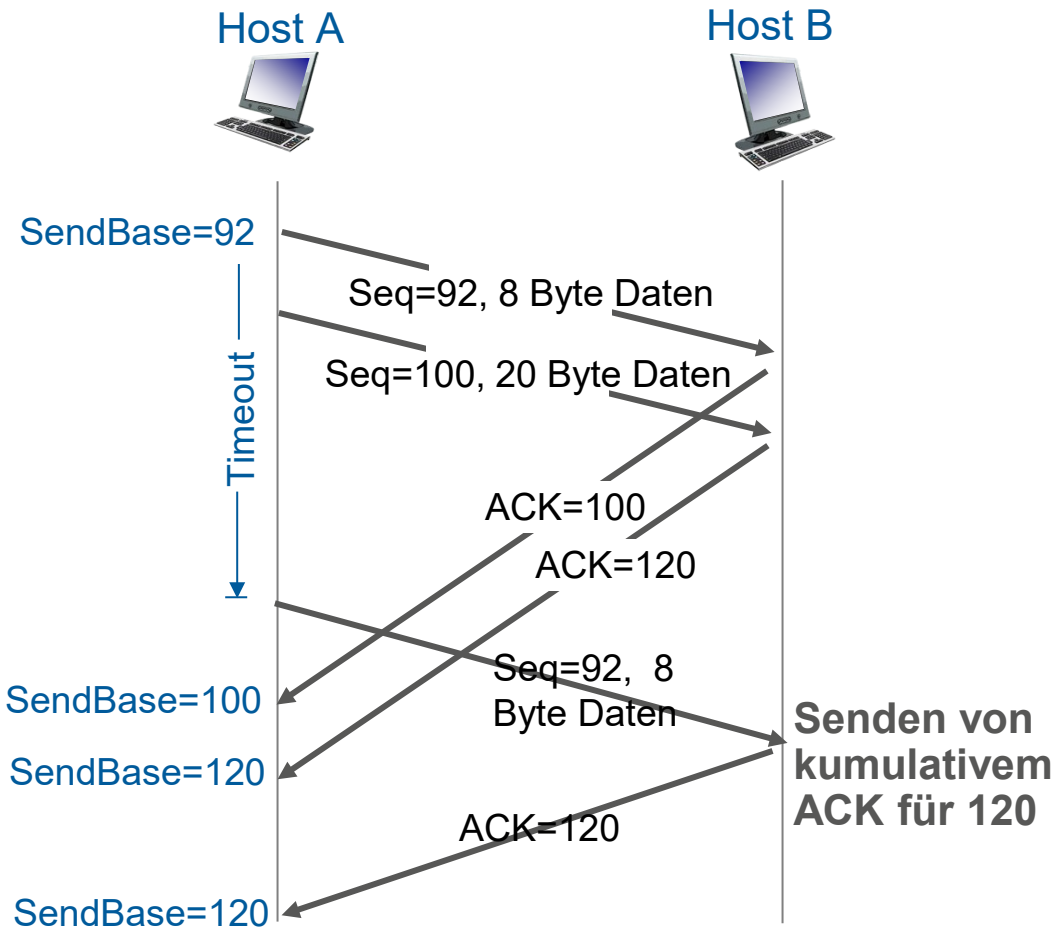
TCP-Empfänger: ACK-Erzeugung [RFC 5681]

Ereignis am Empfänger	TCP Empfänger Aktion
Empfang eines Segments in Reihenfolge mit erwarteter Sequenznr.; Alle Daten bis zur erwarteten Sequenznr. bereits geACKt	verzögertes ACK. Warte bis zu 500ms auf nächstes Segment. Falls kein nächstes Segment ankommt, ACK senden
Empfang eines Segments in Reihenfolge mit erwarteter Sequenznr.; Ein weiteres Segment noch unbestätigt	Sofortiges Senden eines einzelnen kumulativen ACKs, dass beide bestätigt
Empfang eines Segments in außer Reihenfolge mit höherer als erwarteter Sequenznr.; Lücke detektiert	Sofortiges Senden eines **Dubletten** *ACK*, dass die Sequenznr. Des nächsten erwarteten Segment angibt
Ankunft eines Segments, das die Lücke partiell oder vollständig füllt	Sofortiges Senden eines ACK, wenn das Segment am unteren Ende der Lücke beginnt

TCP: Szenarien für Wiederholungsübertragung

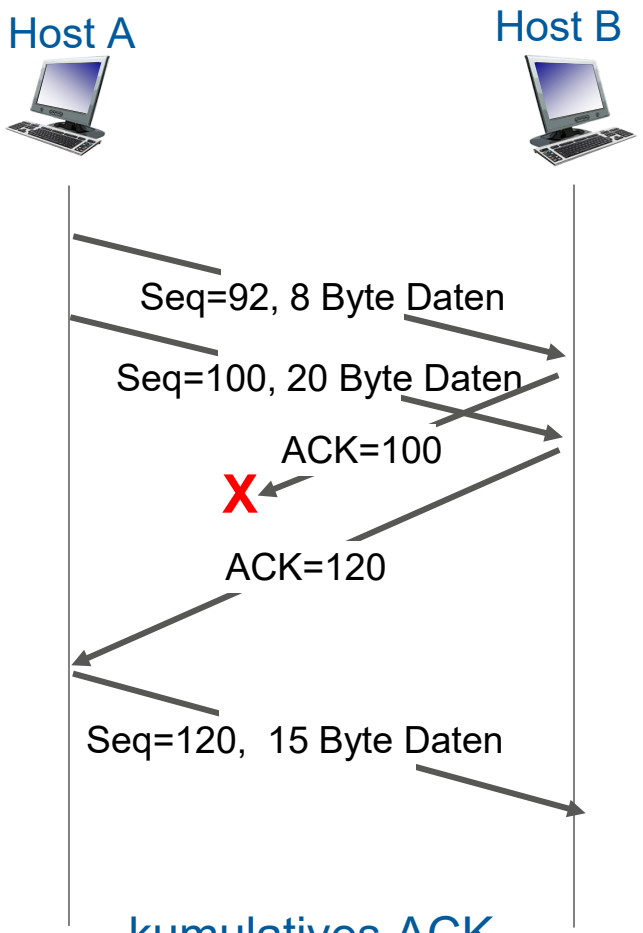


Verlorenes ACK



Verfrühter Timeout

TCP: Szenarien für Wiederholungsübertragung



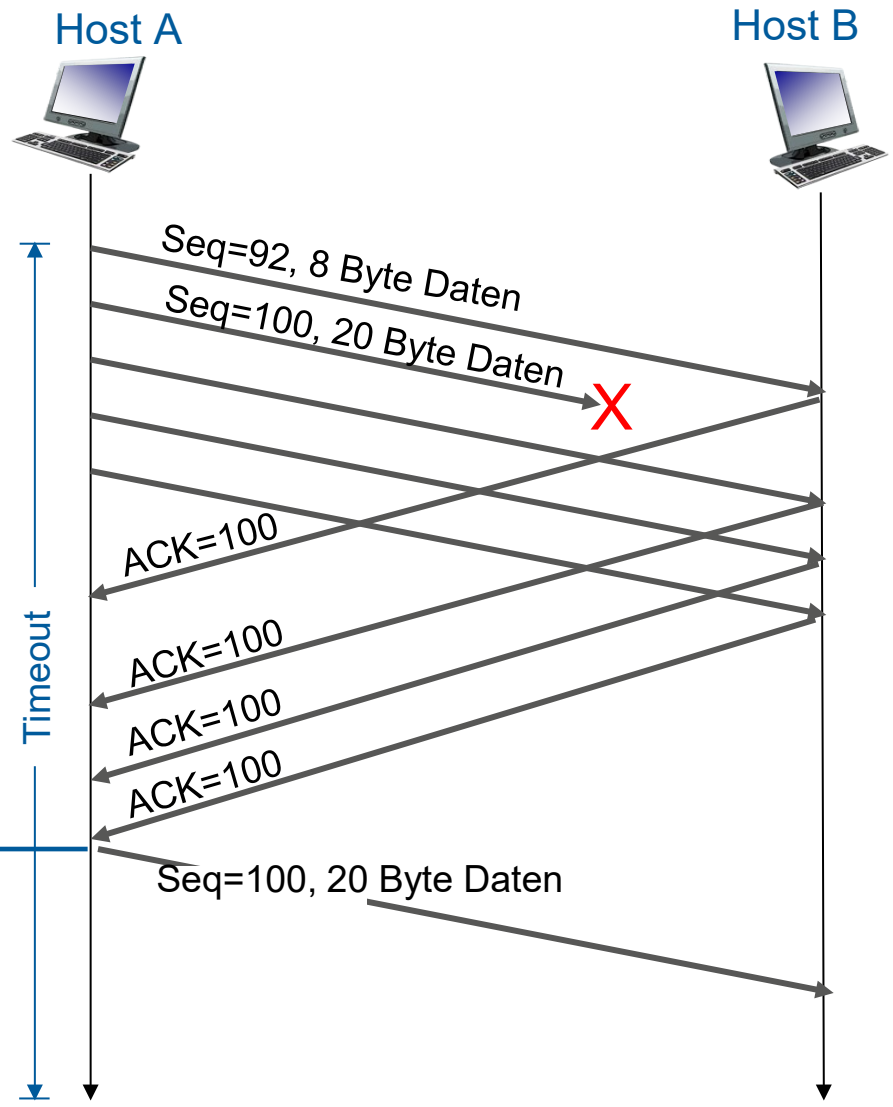
kumulatives ACK erledigt Bestätigung für zuvor verloren gegangenes ACK

TCP Fast Retransmit

falls der Sender 3 zusätzliche ACKs für dieselben Daten empfängt (“triple duplicate ACKs”), erneutes Senden des letzten ungeACKten Segment mit der kleinsten Sequenznr.

- Es ist wahrscheinlich, dass das ungeACKte Segment verloren ist, daher nicht auf Timeout warten

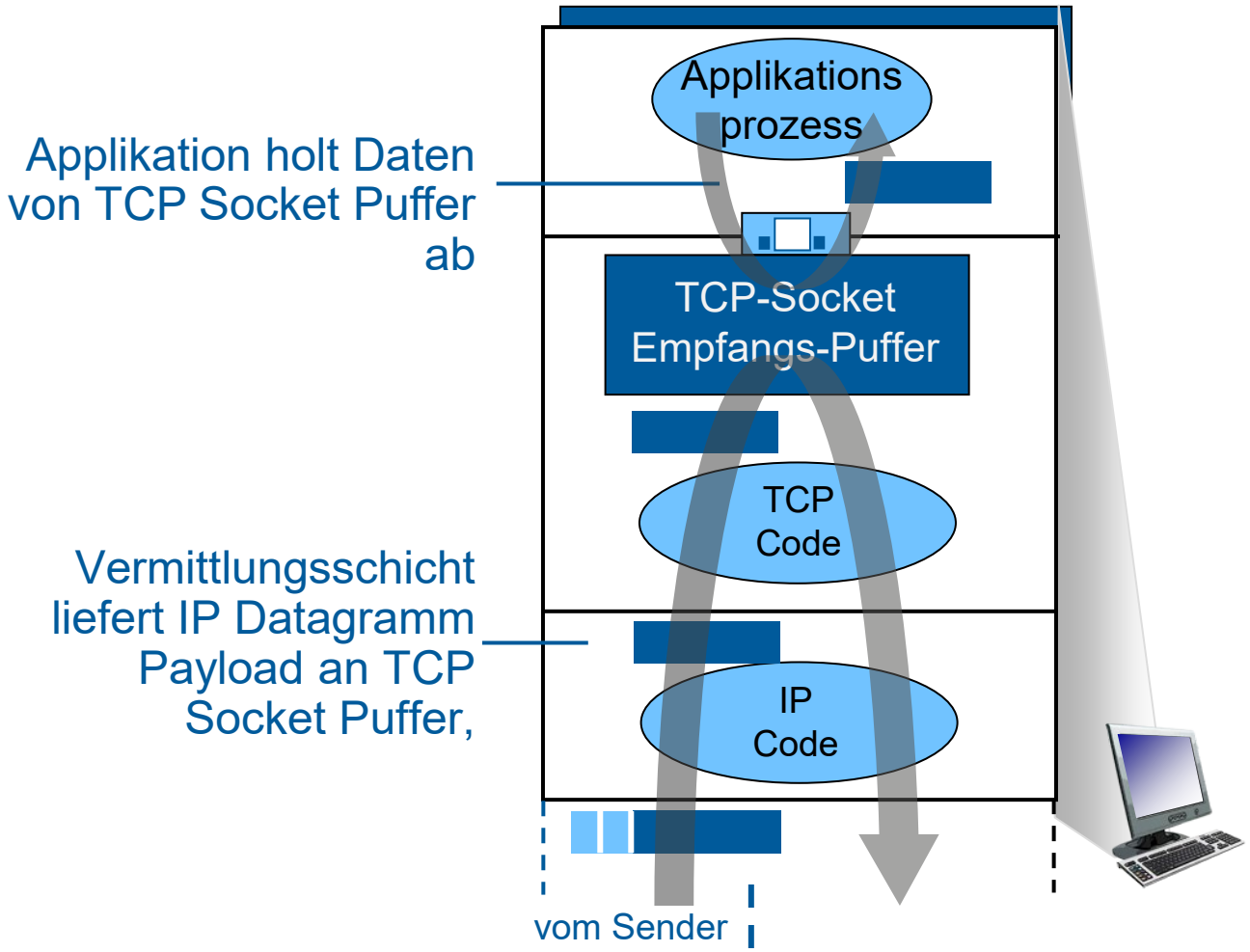
Der Empfang von drei Dupletten ACKs zeigt and, dass 3 Segmente nach einem fehlenden Segment empfangen wurden – verlorenes Segment wahrscheinlich. Daher erneut senden!



Transportschicht: Übersicht

- Dienste der Transportschicht
- Multiplexen und Demultiplexen
- Verbindungsloser Transport: UDP
- Prinzipien verlässlicher Datenübertragung
- **Verbindungsorientierter Transport: TCP**
  - Segmentstruktur
  - Zuverlässige Datenübertragung
  - Flusskontrolle
  - Verbindungsmanagement
- Prinzipien der Überlastkontrolle
- TCP Überlastkontrolle
- Evolution der Transportschicht Funktionen

Frage: Was passiert, wenn die Vermittlungsschicht schneller Daten liefert als die Applikationsschicht sie aus den Socket Puffern abholt?

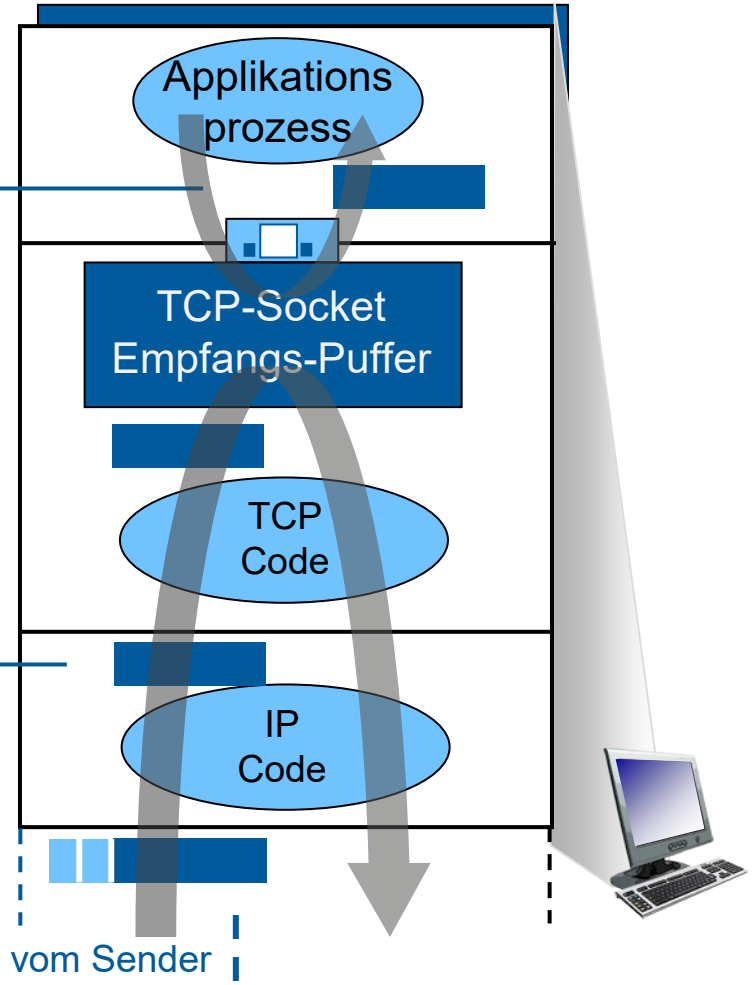


Empfänger Protokoll Stack

Frage: Was passiert, wenn die Vermittlungsschicht schneller Daten liefert als die Applikationsschicht sie aus den Socket Puffern abholt?

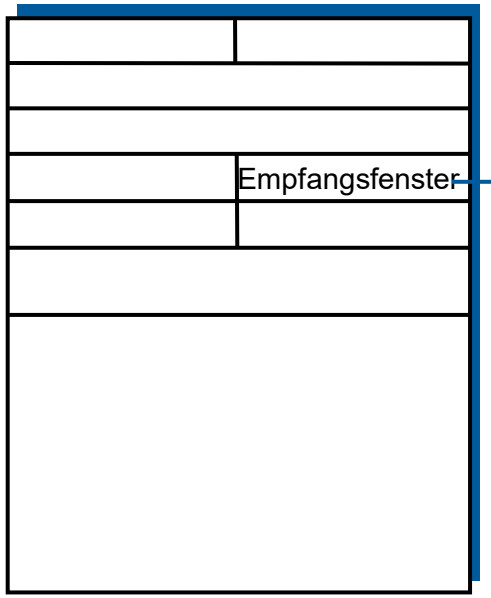
Applikation holt Daten von TCP Socket Puffer ab

Vermittlungsschicht liefert IP Datagramm Payload an TCP Socket Puffer,



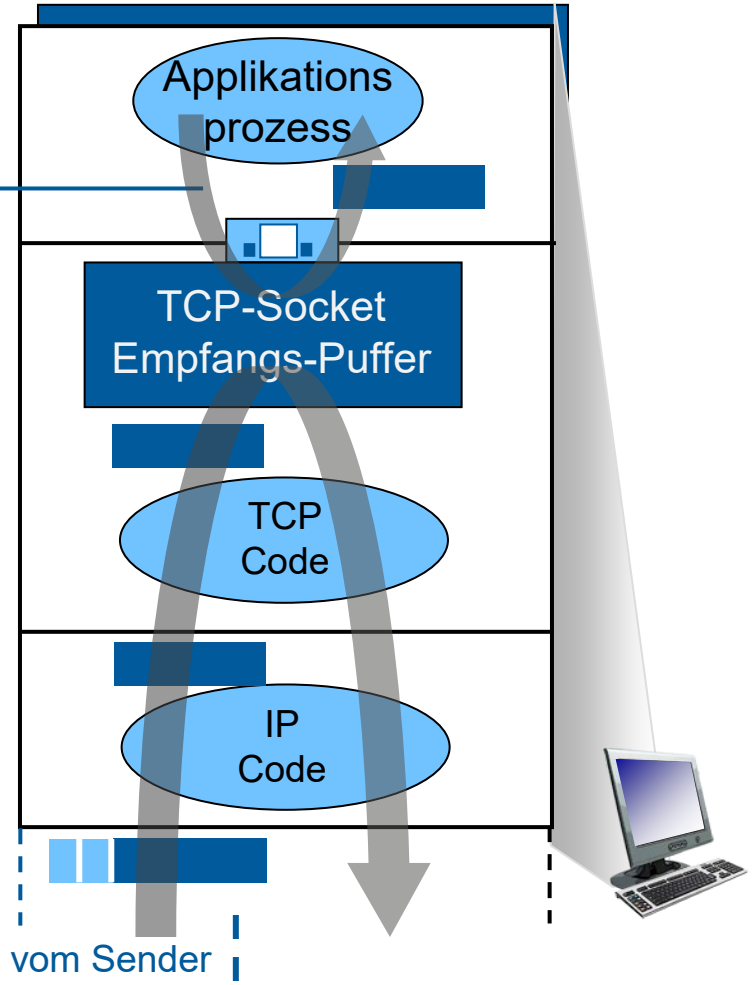
Empfänger Protokoll Stack

Frage: Was passiert, wenn die Vermittlungsschicht schneller Daten liefert als die Applikationsschicht sie aus den Socket Puffern abholt?



Flusskontrolle: Anzahl Byte, die der Empfänger akzeptiert

Applikation holt Daten von TCP Socket Puffer ab

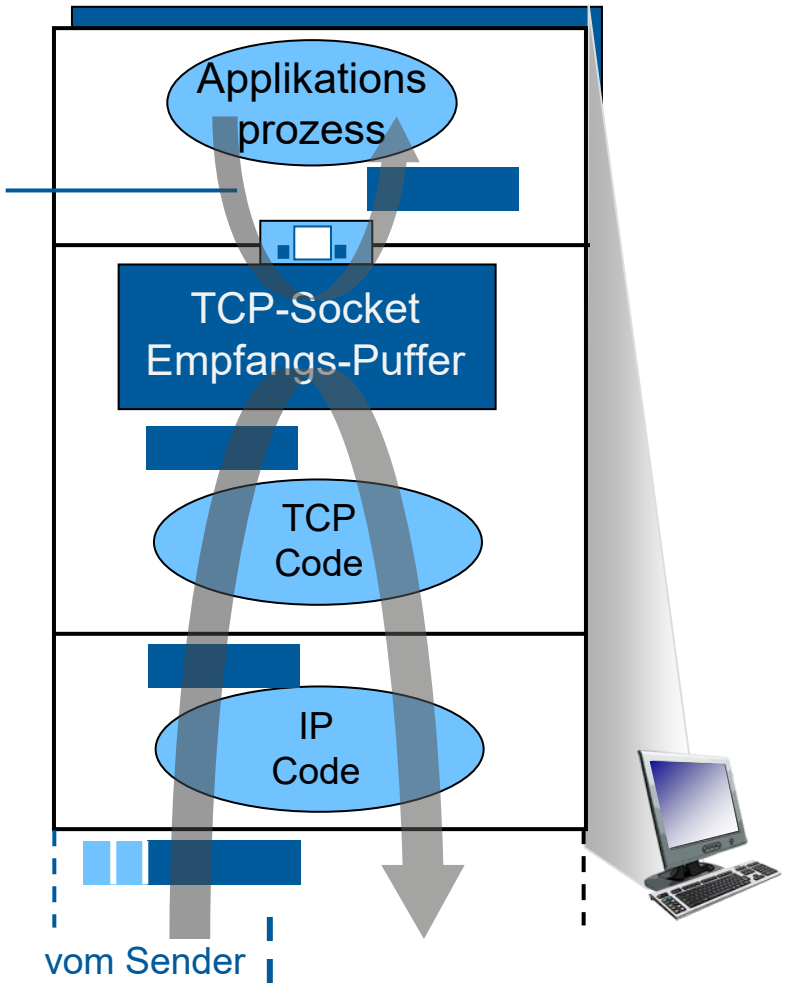


Empfänger Protokoll Stack

Frage: Was passiert, wenn die Vermittlungsschicht schneller Daten liefert als die Applikationsschicht sie aus den Socket Puffern abholt?

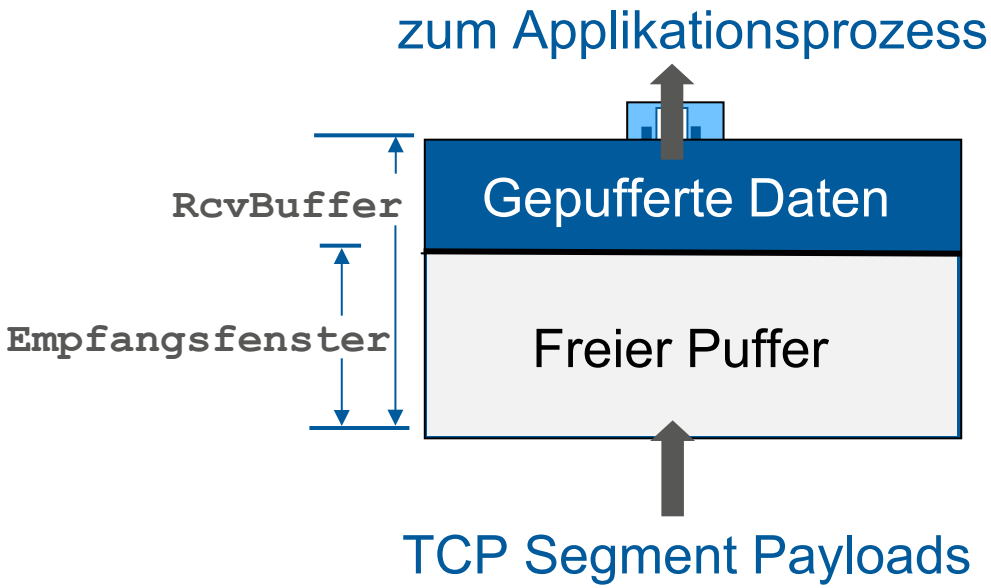
Flusskontrolle

Empfänger steuert Sender, so dass dieser den Empfangspuffer nicht durch zu schnelles Senden überlaufen lässt



Empfänger Protokoll Stack

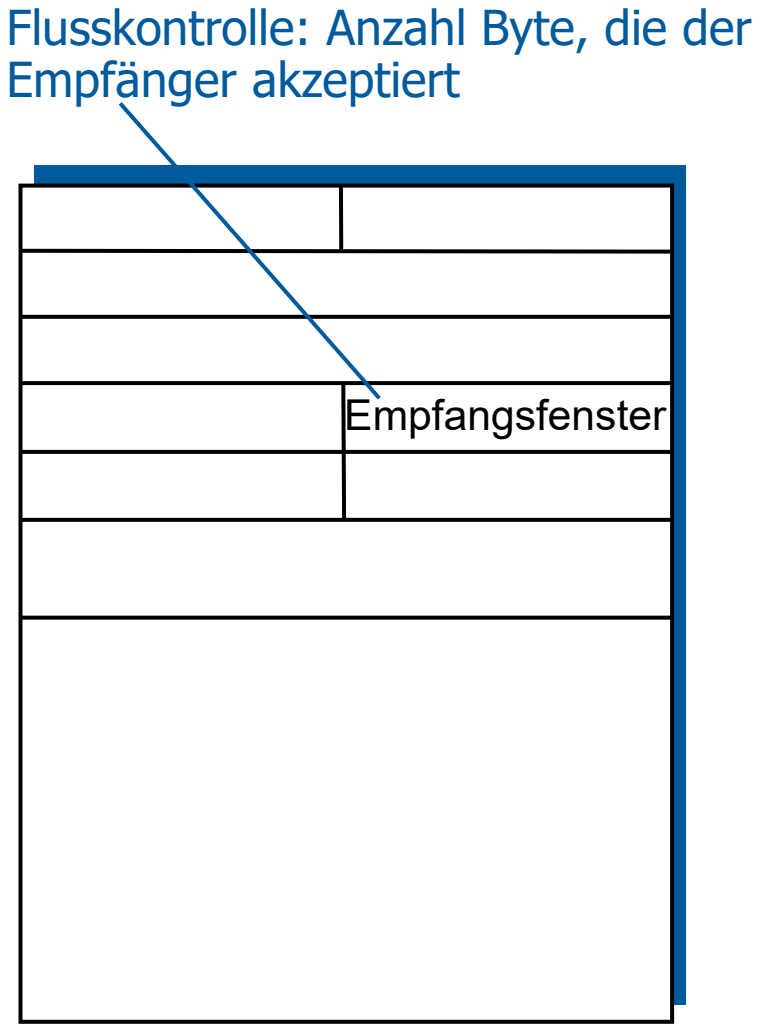
- TCP-Empfänger “bewirbt” freien Pufferplatz im `Empfangsfenster` Feld des TCP-Header
  - `RcvBuffer` Größe wird via Socket Optionen gesetzt (typisch 4096 Byte)
  - Viele Betriebssysteme passen den `RcvBuffer` automatisch an
- Der Sender begrenzt die Menge von ungeACKten (“im-Flug”) befindlichen Daten auf das `Empfangsfenster`
- garantiert, dass der Puffer nicht überläuft



TCP-Empfänger-seitiges Puffern

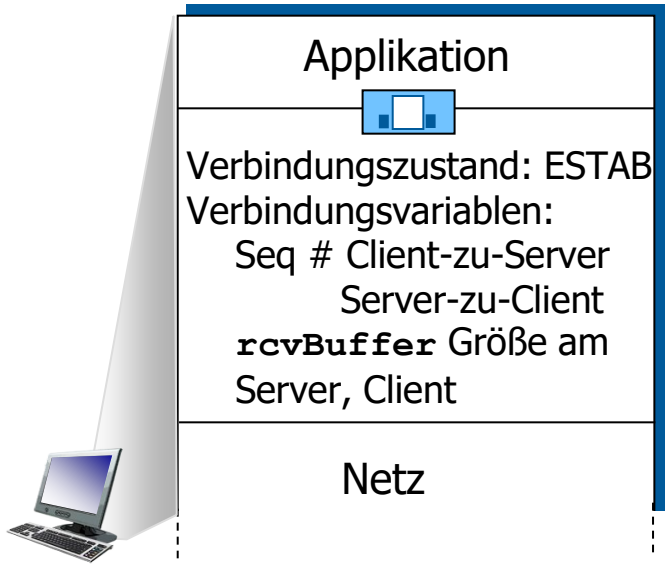
TCP-Flusskontrolle

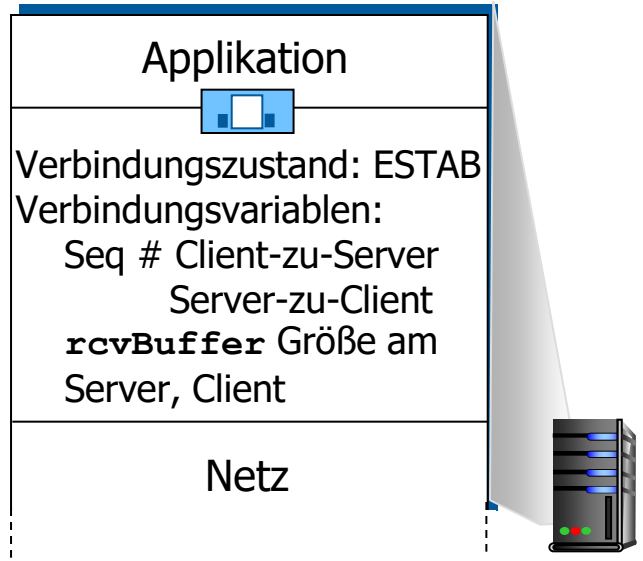
- TCP-Empfänger “bewirbt” freien Pufferplatz im `Empfangsfenster` Feld des TCP-Header
  - `RcvBuffer` Größe wird via Socket Optionen gesetzt (typisch 4096 Byte)
  - Viele Betriebssysteme passen den `RcvBuffer` automatisch an
- Der Sender begrenzt die Menge von ungeACKten (“im-Flug”) befindlichen Daten auf das `Empfangsfenster`
- garantiert, dass der Puffer nicht überläuft



Vor der Datenübertragung geben sich Sender und Empfänger “die Hand”:

- Vereinbarung zum Verbindungsaufbau (beide Seiten wollen sich verbinden)
- Vereinbarung zu Verbindungsparametern (z.B., Startsequenznummern)

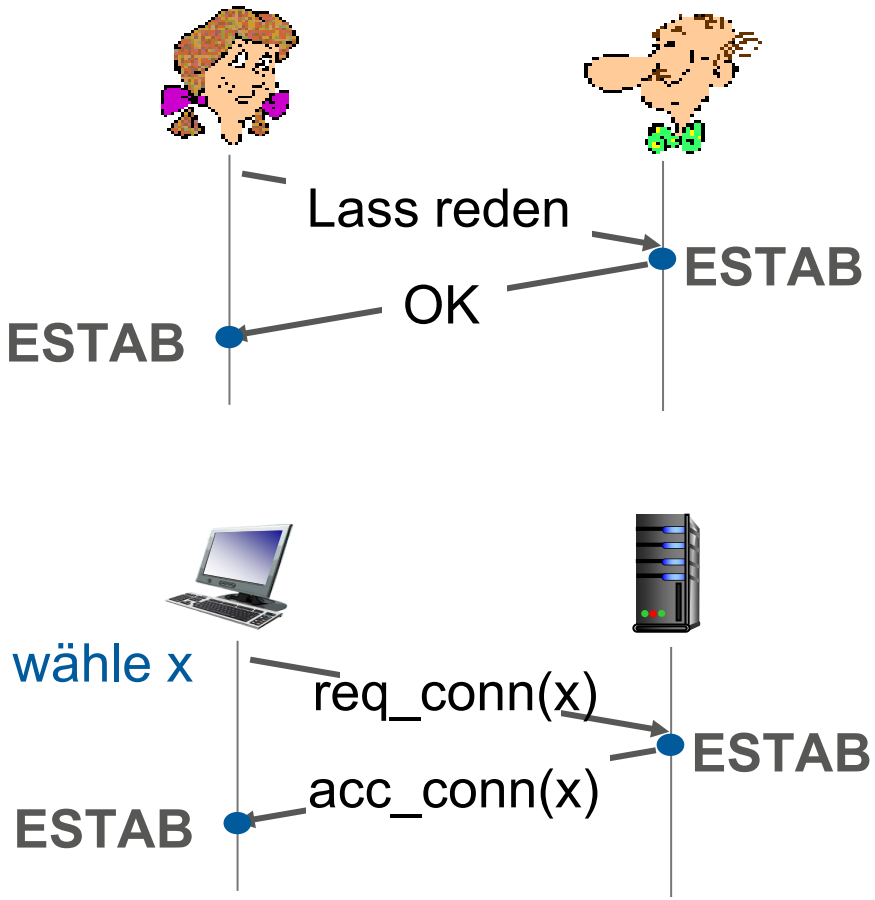





```
Socket clientSocket =
  newSocket("hostname","port number");
```

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

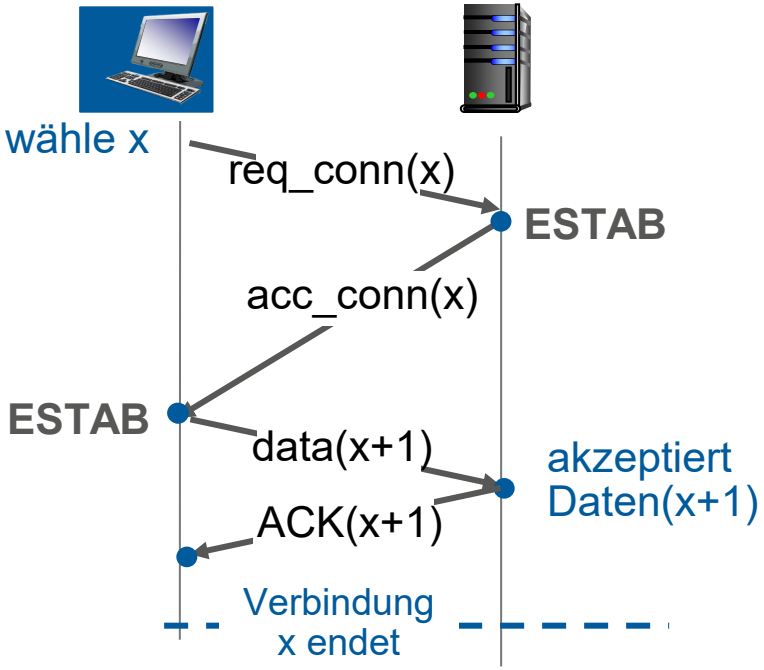
2-Wege Handshake:



Frage: Funktioniert ein 2-way Handshake immer im Netz?

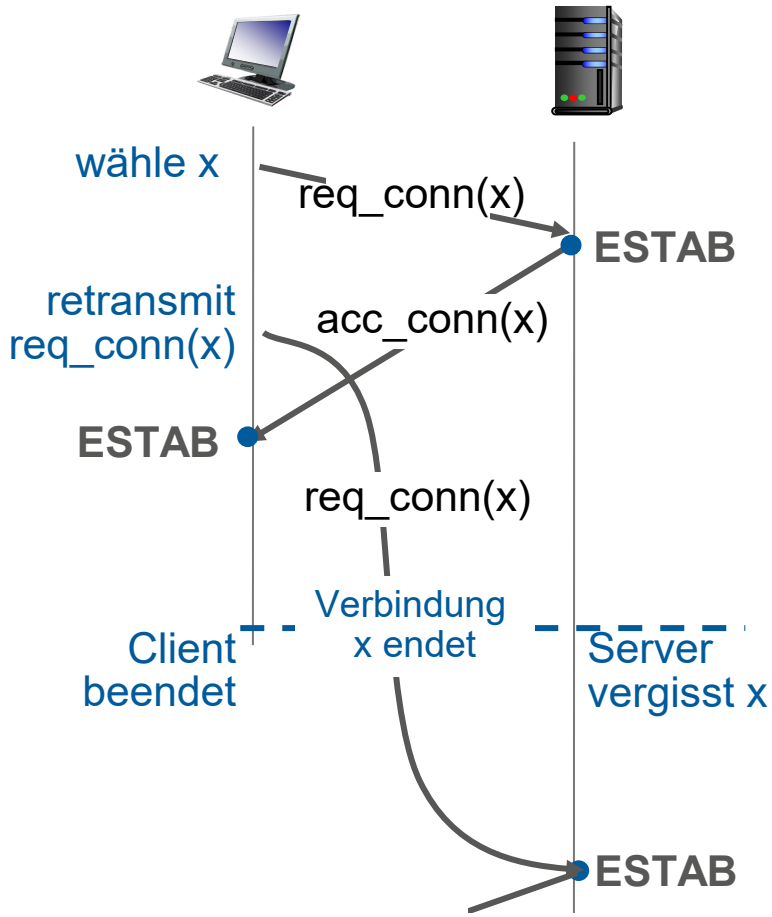
- Variable Verzögerung
- Übertragungswiederholung (z.B. req_conn(x)) wegen Paketverlust
- Änderung der Nachrichtenreihenfolge
- Gegenüber kann nicht “gesehen” werden

2-Wege Handshake Szenarien



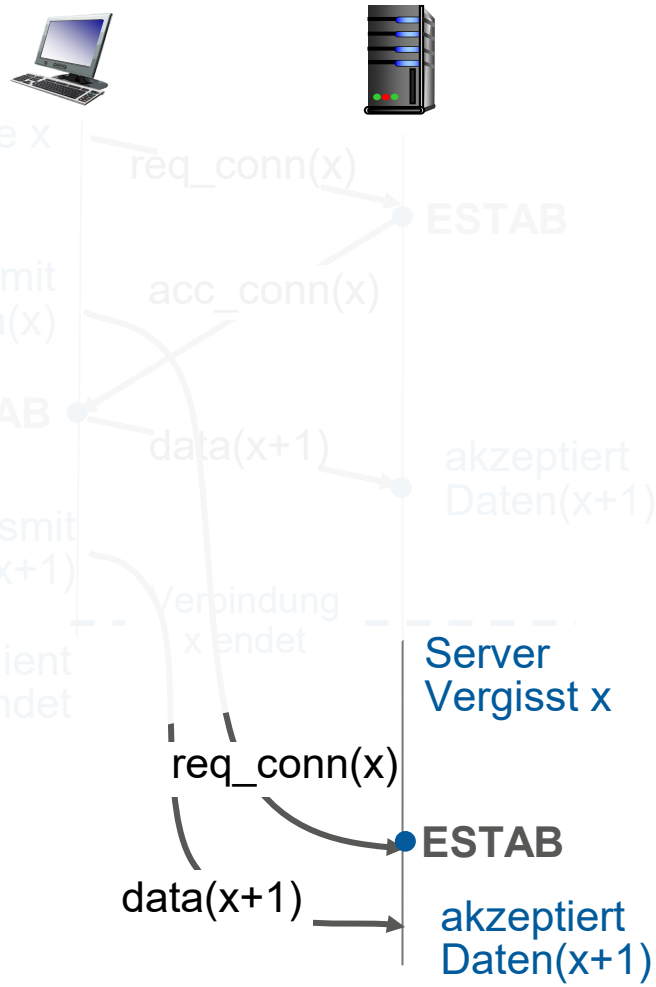
Kein Problem!

2-Wege Handshake Szenarien



Problem: Halb-offene Verbindung (kein Client)!

2-Wege Handshake Szenarien



Problem: Dublette akzeptiert!

Client Zustand

```
clientSocket = socket(AF_INET, SOCK_STREAM)
```

LISTEN

```
clientSocket.connect((serverName,serverPort))
```

wähle init Seq-num, x
sende TCP SYN

SYNSENT

SYNbit=1, Seq=x

wähle init Seq-num, y
sende TCP SYNACK,
bestätige SYN

SYNbit=1, Seq=y
ACKbit=1; ACKnum=x+1

empfängt SYNACK(x)
→Server ist live;
sende ACK für SYNACK;
Dieses Segment kann schon
Daten enthalten

ESTAB

ACKbit=1, ACKnum=y+1

empfängt ACK(y)
→ Client ist live

Server Zustand

```
serverSocket = socket(AF_INET,SOCK_STREAM)
serverSocket.bind(('',serverPort))
serverSocket.listen(1)
connectionSocket, addr = serverSocket.accept()
```

LISTEN

SYN RCVD

ESTAB

Ein menschliches 3-Wege Handshake Protokoll

Schließen einer TCP-Verbindung

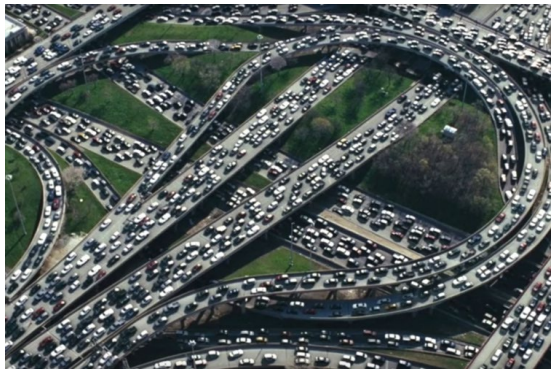
- Client & Server schließen jeweils ihre Seite der Verbindung
  - senden TCP-Segment mit FIN-Bit = 1
- beantworten ein empfangenes FIN mit einem ACK
  - beim Empfang eines FIN, kann das ACK mit eigenem FIN kombiniert werden
- gleichzeitiger FIN-Austausch ist möglich

Transportschicht: Übersicht

- Dienste der Transportschicht
- Multiplexen und Demultiplexen
- Verbindungsloser Transport: UDP
- Prinzipien verlässlicher Datenübertragung
- Verbindungsorientierter Transport: TCP
- **Prinzipien der Überlastkontrolle**
- TCP Überlastkontrolle
- Evolution der Transportschicht Funktionen

Überlast:

- informell: “Zu viele Quellen, senden zu viele Daten zu schnell für das **Netz**.”
- Folgeerscheinungen:
  - Lange Verzögerungen (Warten in Router Puffern)
  - Paketverlust (Pufferüberlauf in Routern)
- Verschieden von der Flusskontrolle!



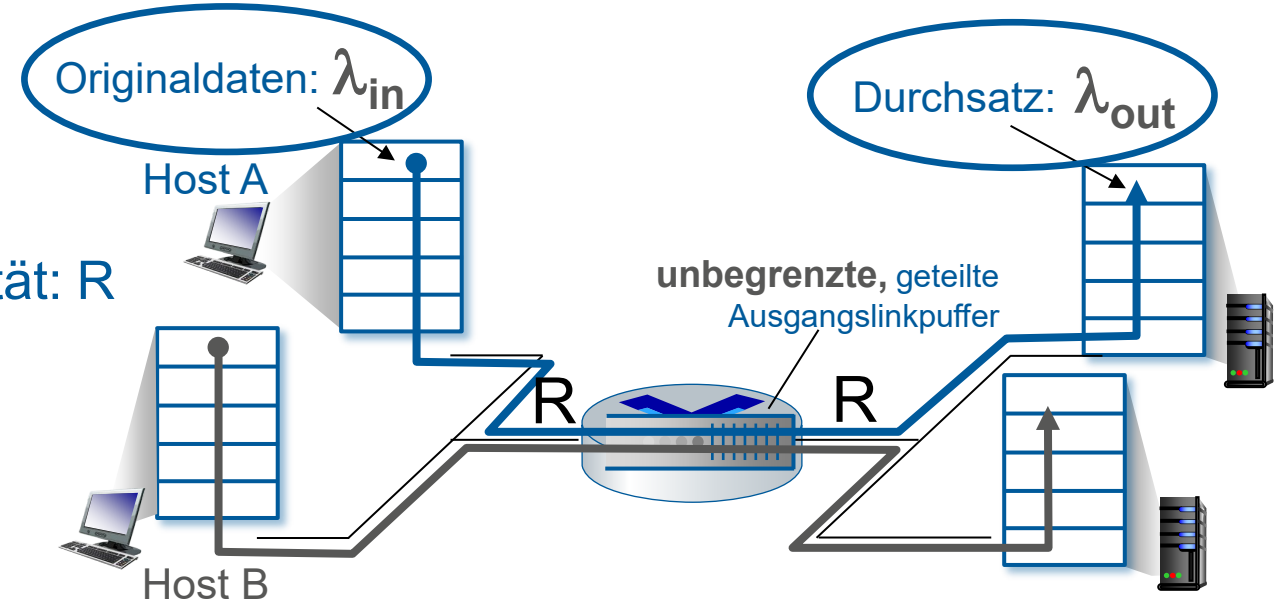
Überlastkontrolle: zu viele Sender senden zu schnell

Flusskontrolle: ein Sender sendet zu schnell für einen Empfänger

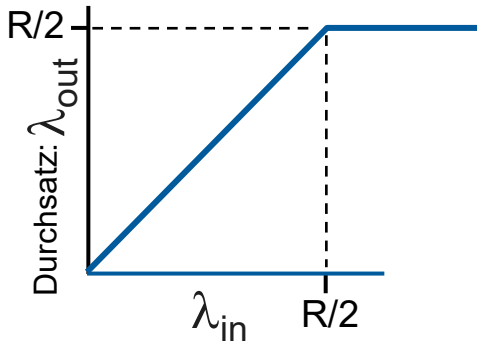
Ursachen/Kosten von Überlast: Szenario 1

Einfachstes Szenario:

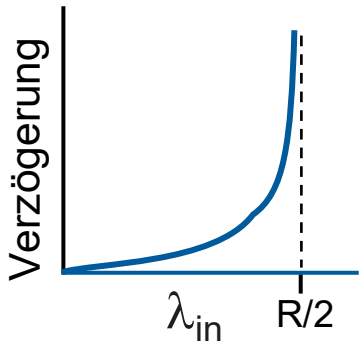
- Ein Router, unbegrenzte Puffer
- Eingangs-, Ausangslink Kapazität: R
- Zwei Flows
- Keine Wiederholungs-übertragungen notwendig



Frage: Was passiert, wenn sich die Ankunftsrate λ_{in} R/2 annähert?



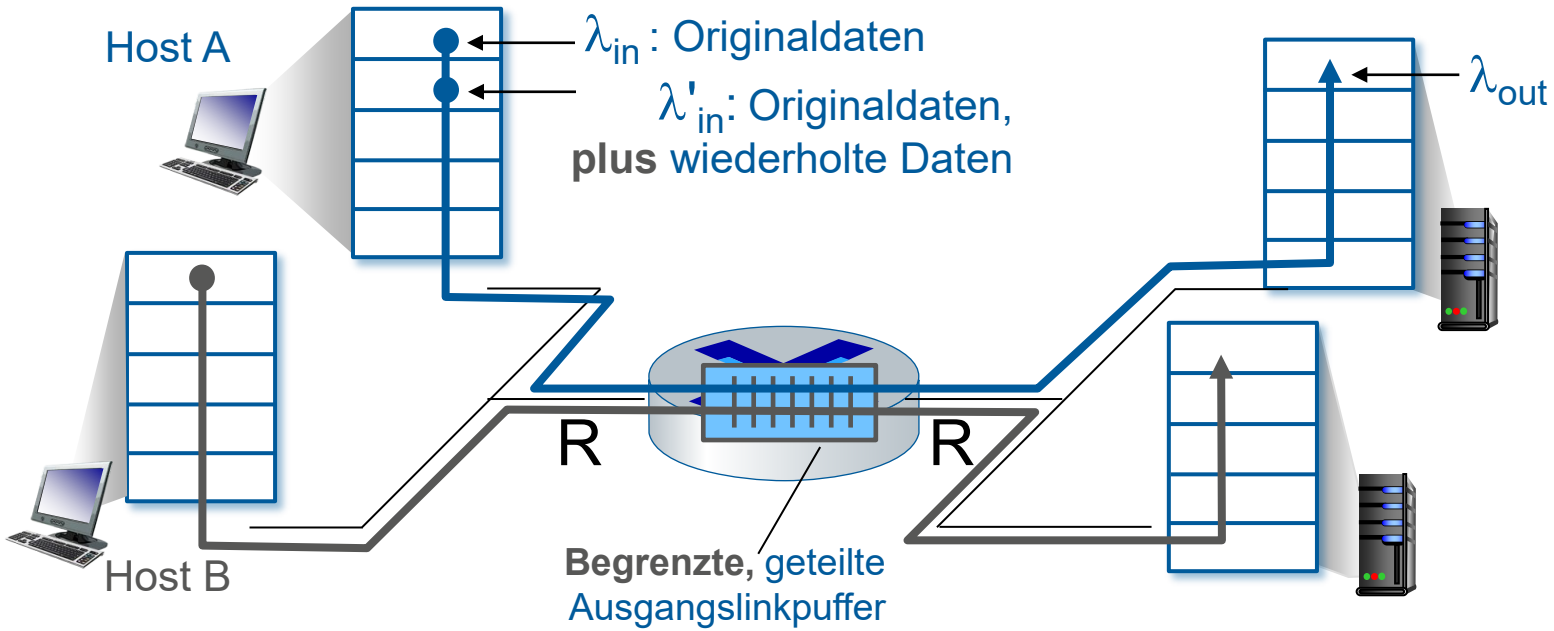
Maximaler Durchsatz pro Verbindung: R/2



Lange Verzögerungen, wenn sich die Ankunftsrate λ_{in} sich der Kapazitätsgrenze annähert

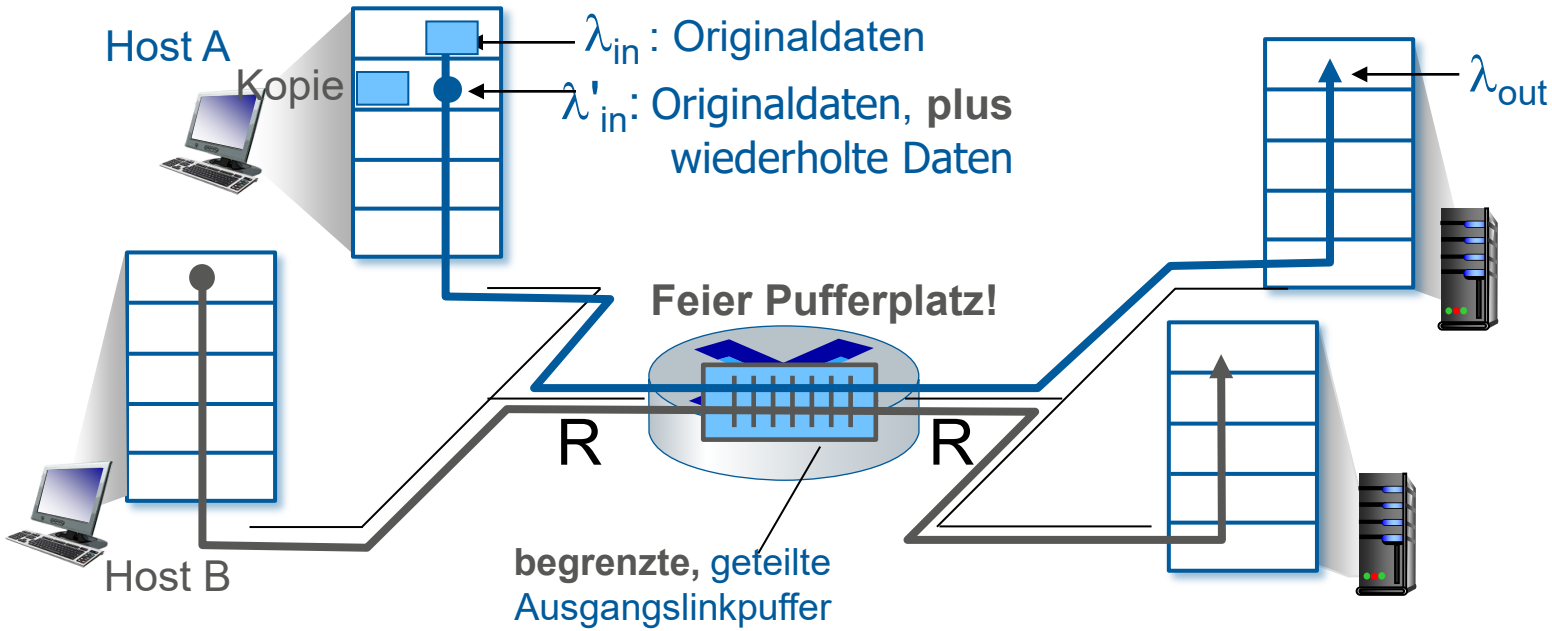
Ursachen/Kosten von Überlast: Szenario 2

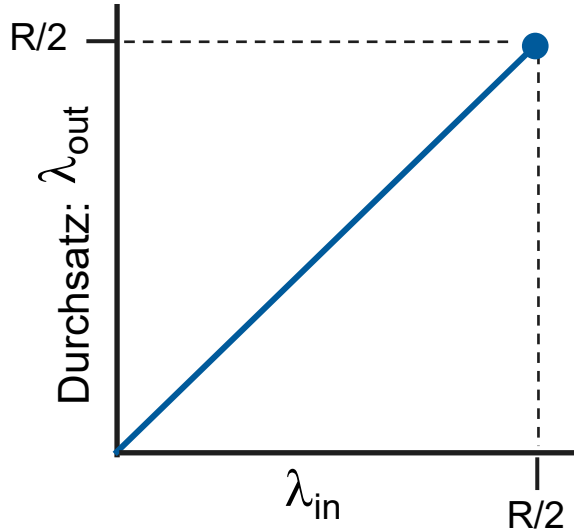
- Ein Router, **begrenzte** Puffer
- Wiederholungsübertragungen für verlorene, ausgetimte Pakete
  - Input von Applikationsschicht = Output zu Applikationsschicht: $\lambda_{in} = \lambda_{out}$
  - Transport-Schicht Input beinhaltet **Wiederholungsübertragungen**: $\lambda'_{in} \geq \lambda_{in}$



Idealfall: **Vollständiges Wissen**

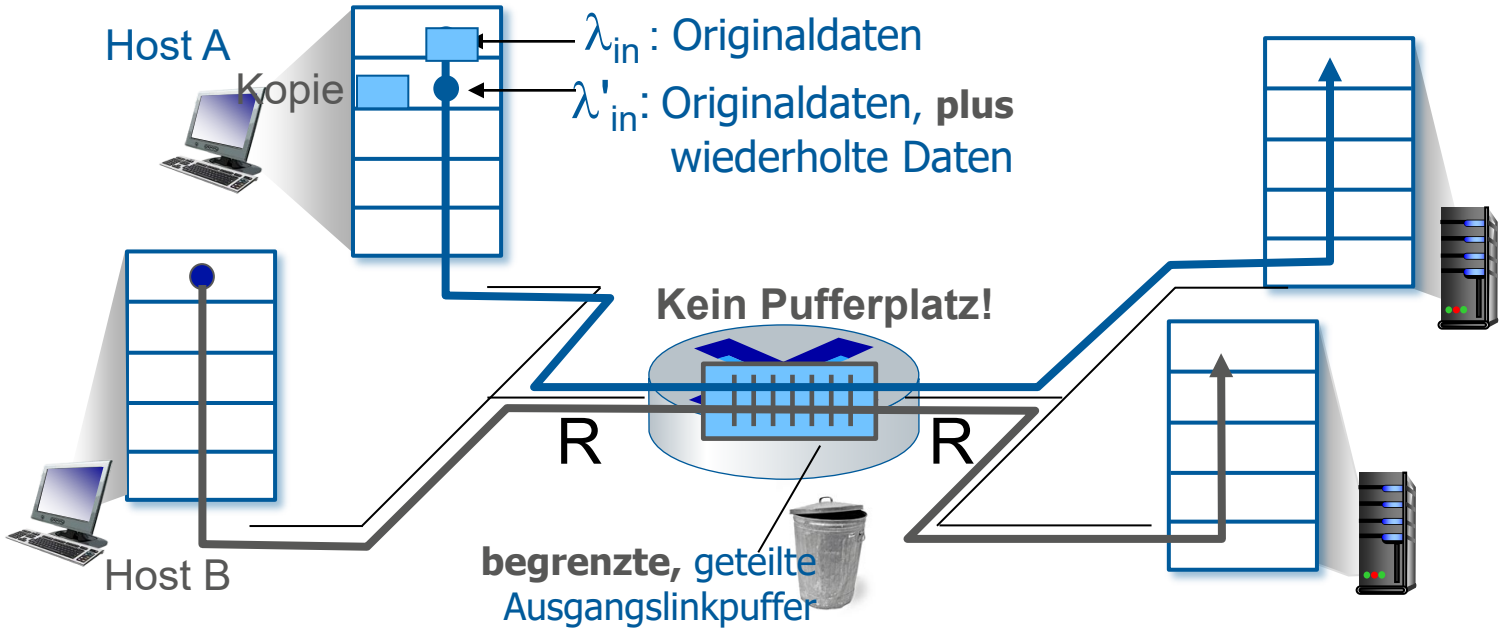
- Sender sendet nur, wenn Router-Pufferplatz verfügbar ist





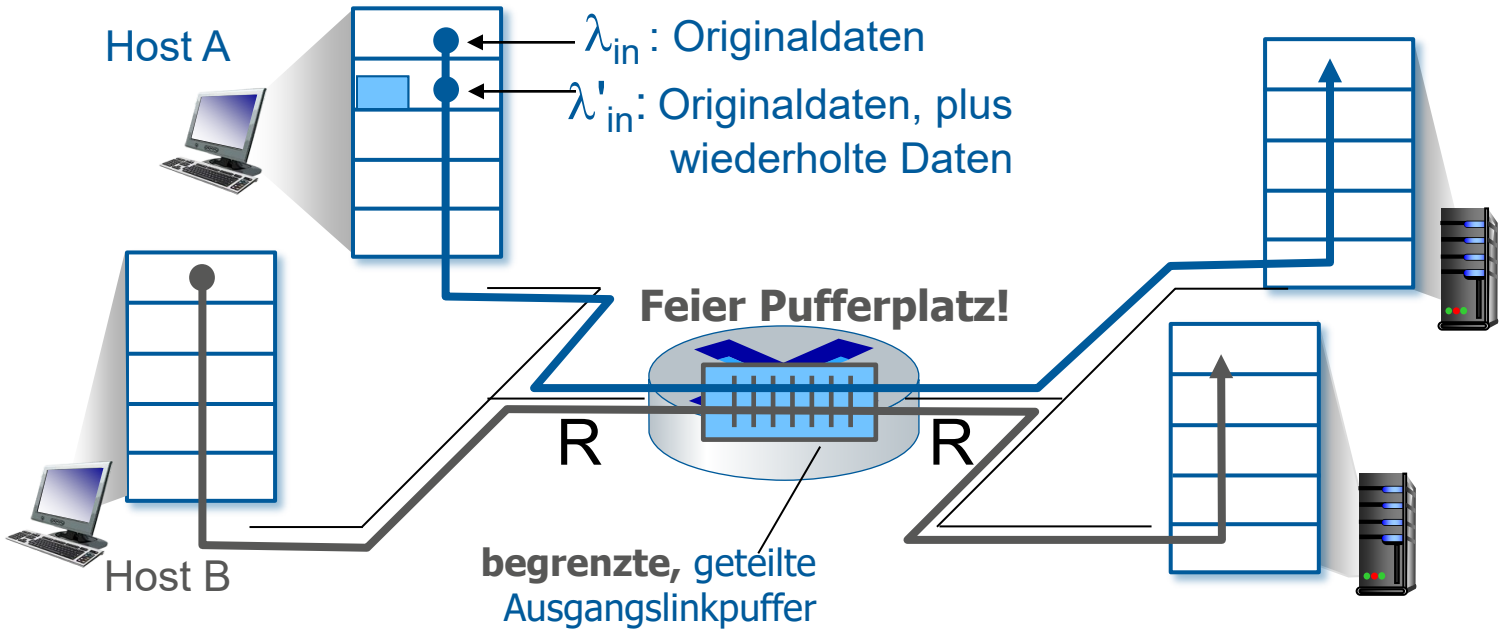
Idealisierung: **etwas** Wissen

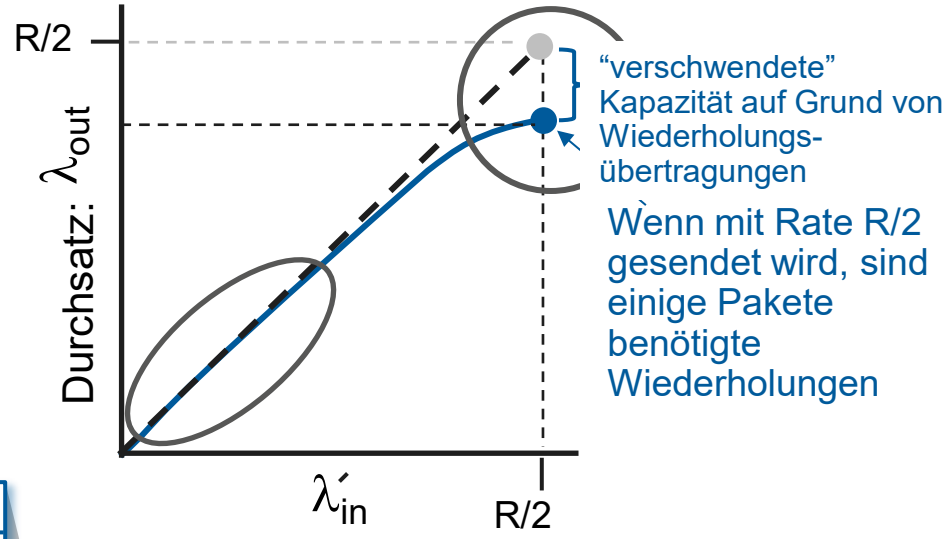
- Pakete können auf Grund voller Puffer verloren gehen (von Router verworfen)
- Der Sender weiß, wenn ein Paket verworfen wurde: wiederholt die Übertragung nur, wenn *bekannt* ist, dass das Paket verloren ist



Idealisierung: **etwas** Wissen

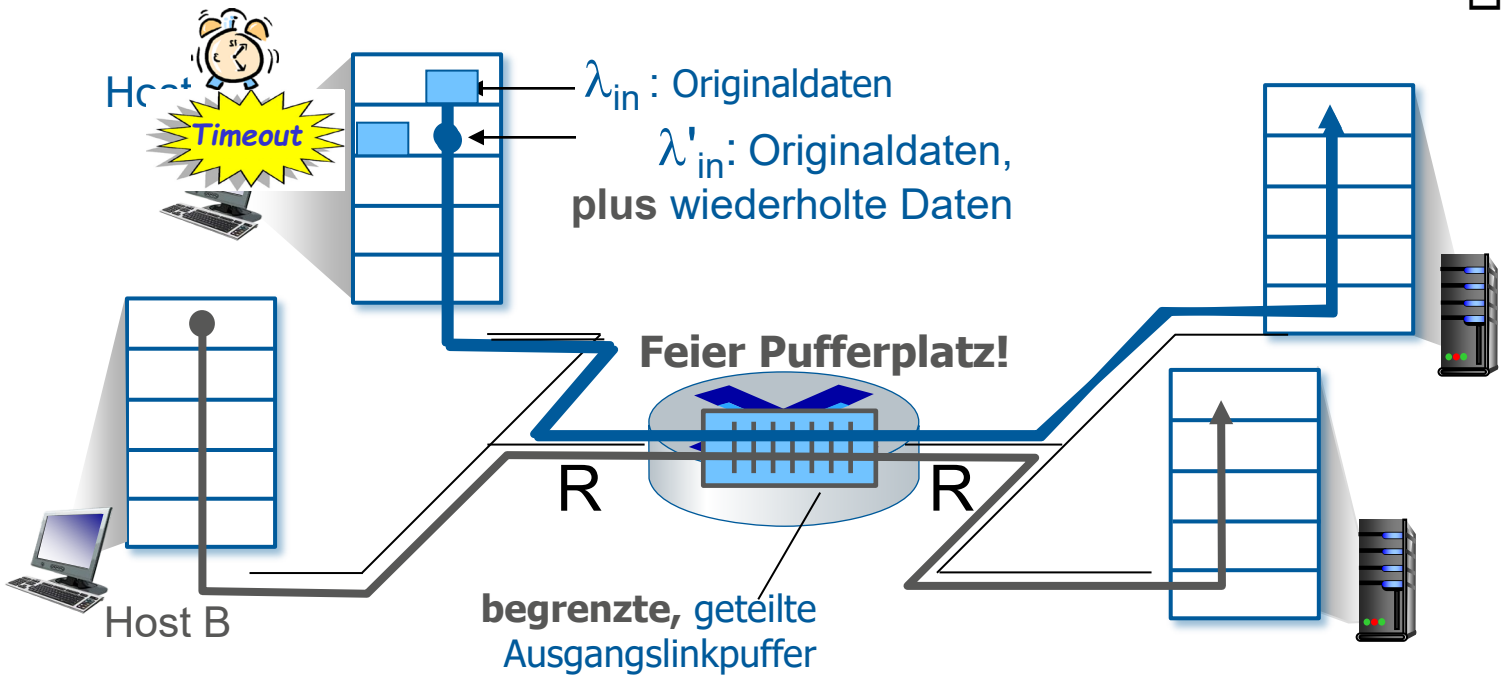
- Pakete können auf Grund voller Puffer verloren gehen (von Router verworfen)
- Der Sender weiß, wenn ein Paket verworfen wurde: wiederholt die Übertragung nur, wenn *bekannt* ist, dass das Paket verloren ist

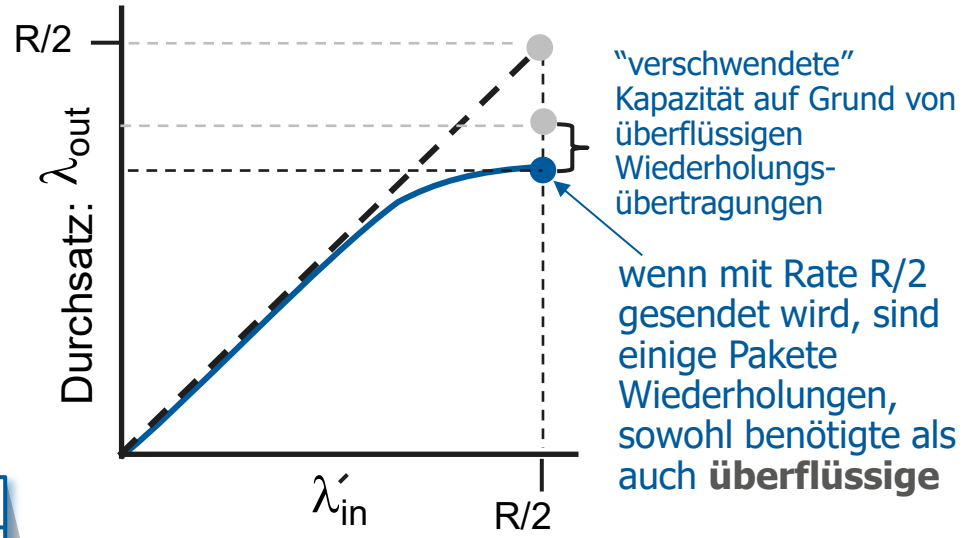




Realistisches Szenario: **überflüssige Dubletten**

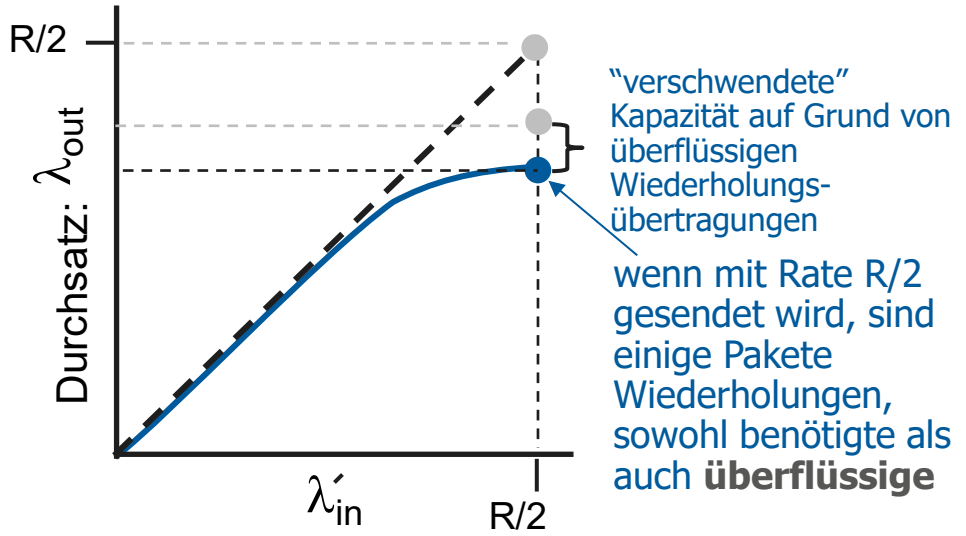
- Pakete können verloren gehen, am Router verworfen – benötigen Wiederholungsübertragungen
- Aber Sender kann frühzeitig aus-timen und sendet dann **zwei** Kopien, die **beide** ankommen





Realistisches Szenario: **überflüssige Dubletten**

- Pakete können verloren gehen, am Router verworfen – benötigen Wiederholungsübertragungen
- Aber Sender kann frühzeitig aus-timen und sendet dann **zwei** Kopien, die **beide** ankommen



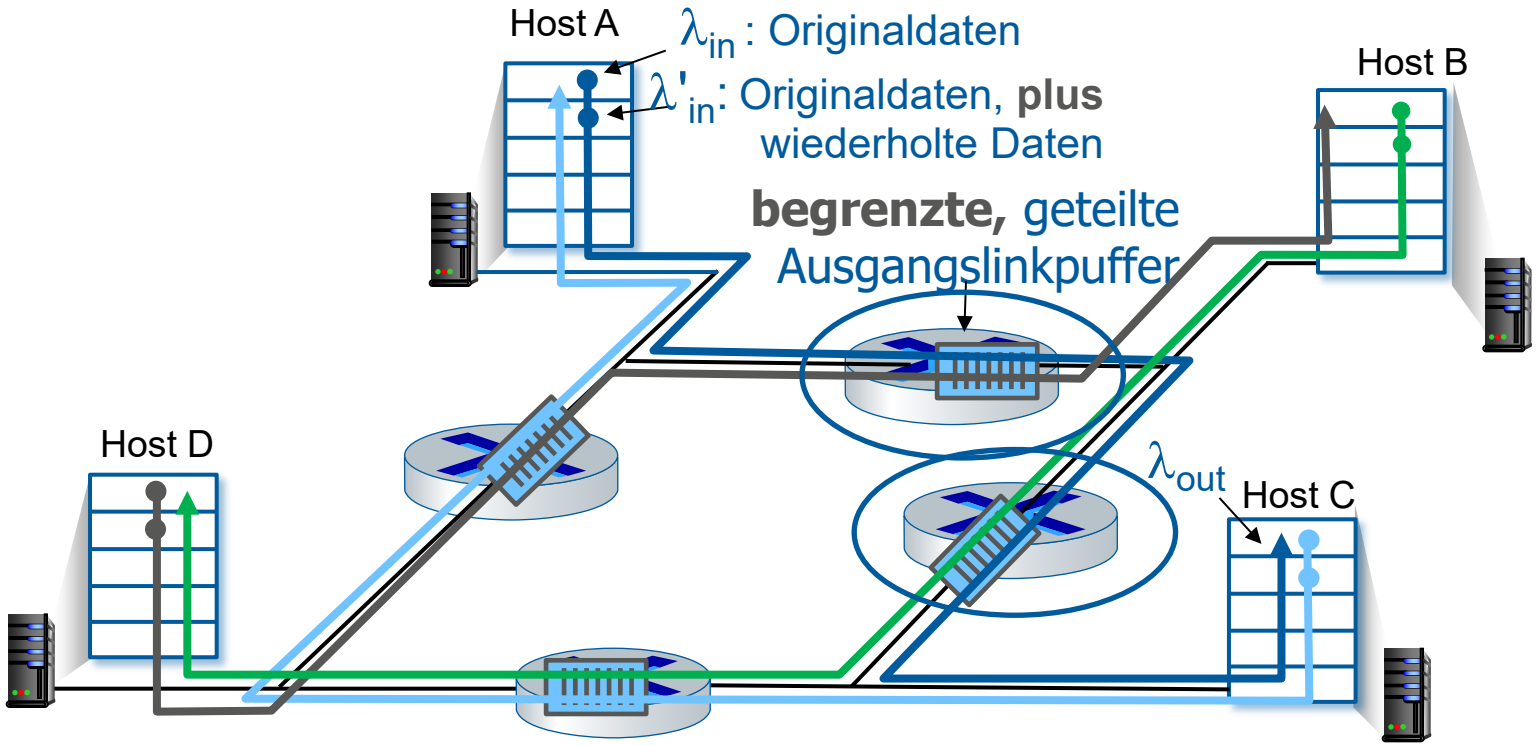
"Kosten" von Überlast:

- Mehr Arbeit (Wiederholungsübertragungen) für gegebenen Empfängerdurchsatz
- Überflüssige Wiederholungsübertragungen: Link überträgt mehrere Kopien eines Pakets, was den maximal erreichbaren Durchsatz reduziert

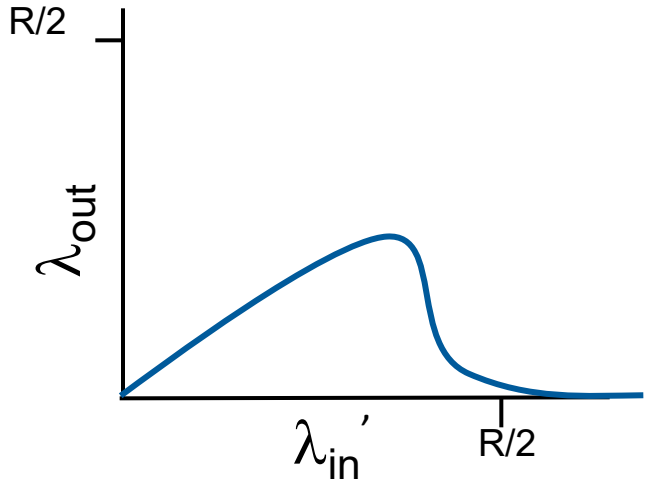
- ***Vier*** Sender
- **multi-hop** Pfade
- Timeout/Wiederholungen

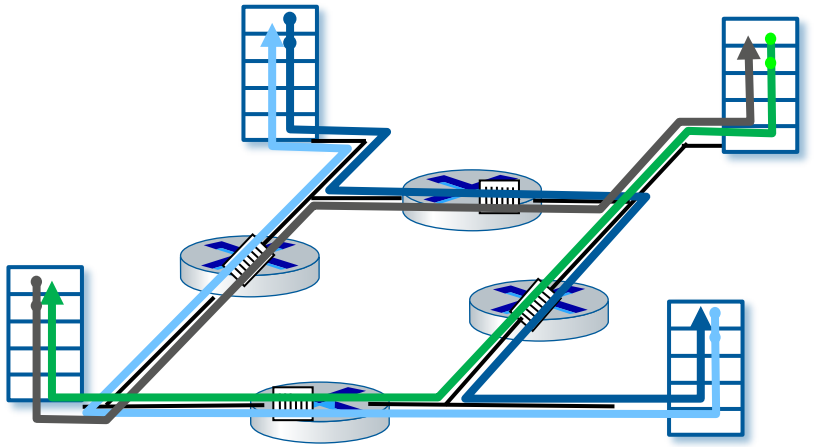
Frage: Was passiert, wenn sich λ_{in} und λ_{in}' erhöhen?

Anwort: wenn sich die dunkelblaue Rate λ_{in}' erhöht, werden alle ankommenden grauen Pakete an der oberen Warteschlange verworfen, grauer Durchsatz → 0



Ursachen/Kosten von Überlast: Szenario 3



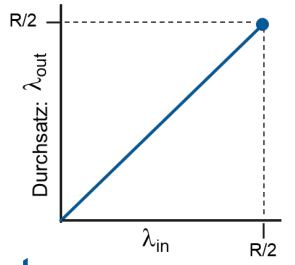


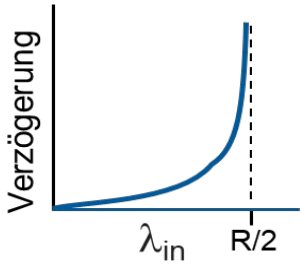
weitere “Kosten” von Überlast:

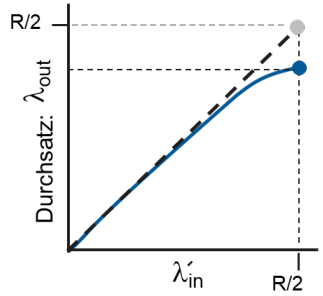
- wenn ein Paket verworfen wird, war jegliche Upstream Übertragungs- und Pufferkapazität verschwendet!

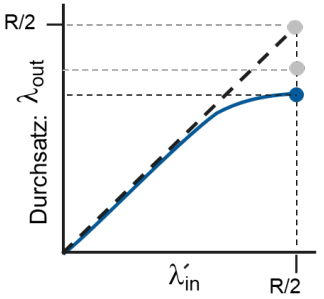
Ursachen/Kosten von Überlast: Erkenntnisse

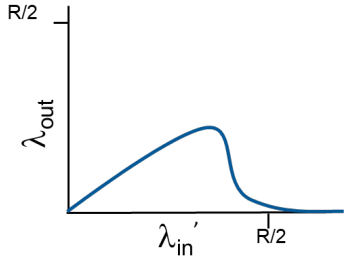
- Durchsatz kann nie die Kapazität übertreffen
- Verzögerung erhöht sich, wenn die Kapazitätsgrenze naht
- Verlust/Wiederholungen reduzieren den effektiven Durchsatz
- überflüssige Dubletten reduzieren den effektiven Durchsatz weiter
- Upstream Übertragungskapazität/Puffer verschwendet, wenn Paket im Anschluss verloren gehen





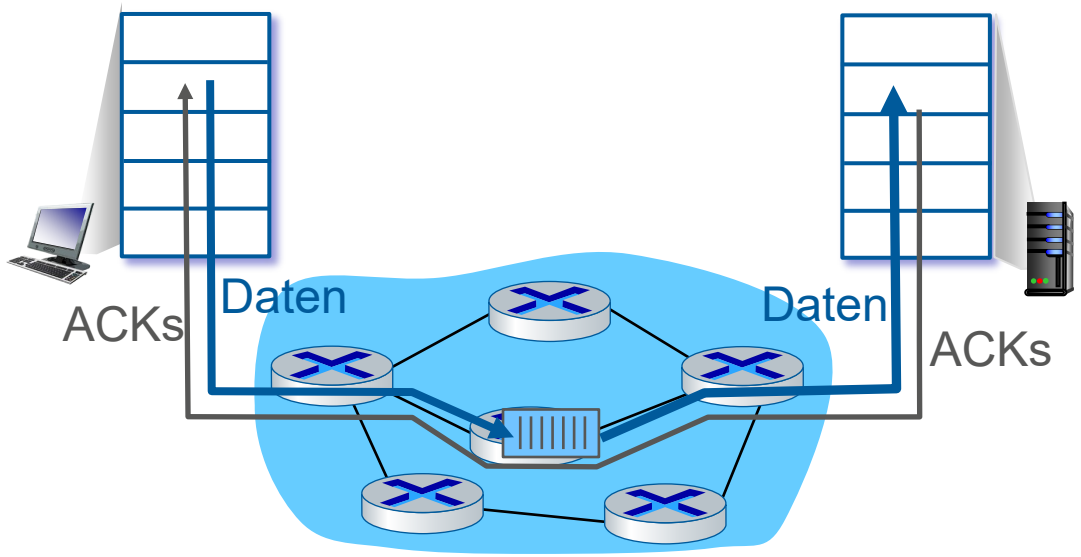






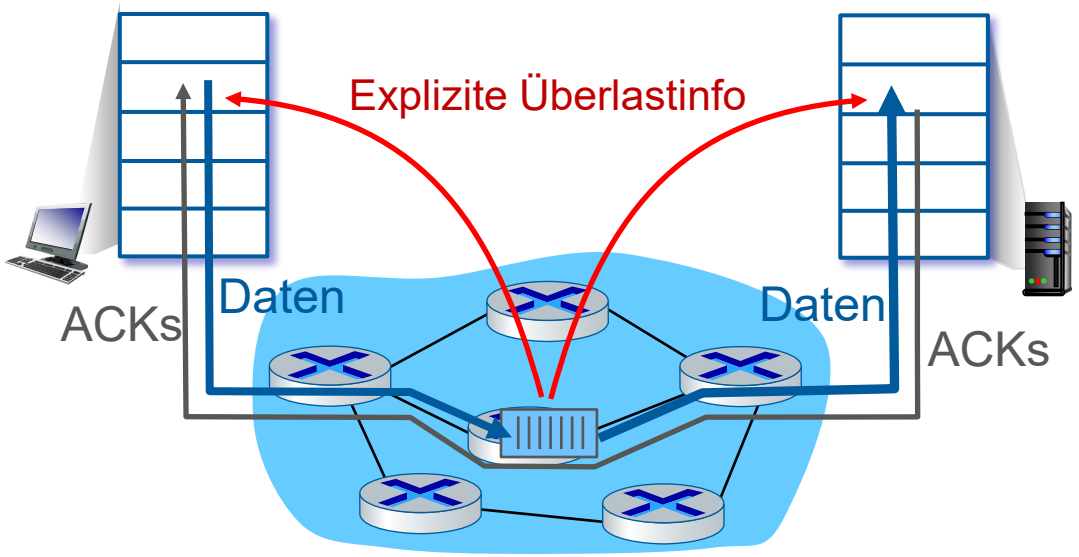
Ende-zu-Ende Überlastkontrolle:

- kein explizites Feedback vom Netz
- Überlast wird durch beobachteten Paketverlust, -verzögerung **abgeleitet**
- Ansatz von TCP



Netz-unterstütze Überlastkontrolle

- Router geben **direktes** Feedback an die sendenden/empfangenden Hosts mittels der Flows, die den überlasteten Router durchqueren
- Kann eine Überlastniveau anzeigen, oder eine explizite Senderate setzen
- TCP ECN, ATM, DECbit Protokolle



Transportschicht: Übersicht

- Dienste der Transportschicht
- Multiplexen und Demultiplexen
- Verbindungsloser Transport: UDP
- Prinzipien verlässlicher Datenübertragung
- Verbindungsorientierter Transport: TCP
- Prinzipien der Überlastkontrolle
- **TCP Überlastkontrolle**
- Evolution der Transportschicht Funktionen

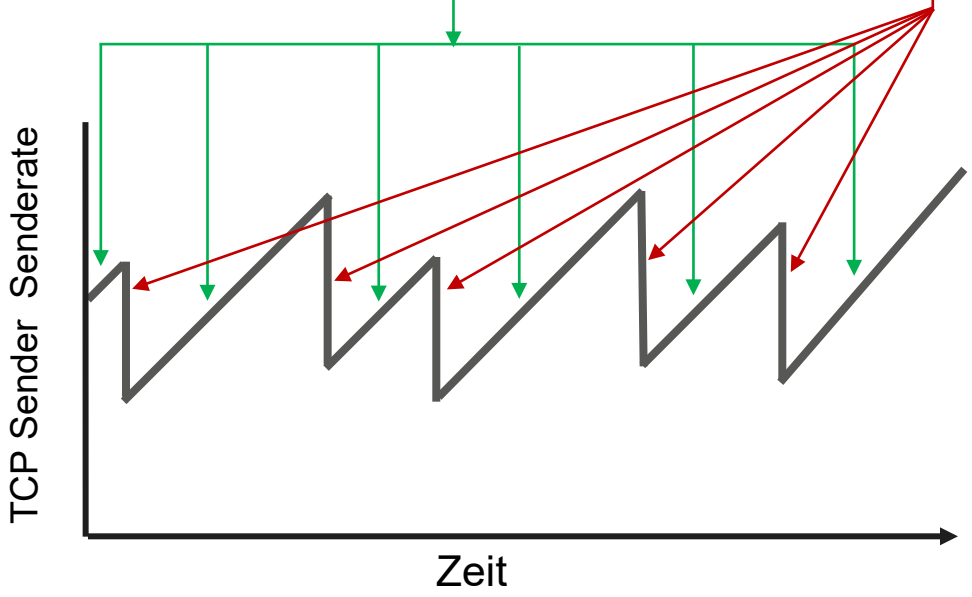
- **Ansatz:** Sender kann seine Senderate erhöhen bis ein Paketverlust (Überlast) auftritt, dann reduzieren der Senderate

Additive Increase

Erhöhen der Senderate um 1 Maximum Segment Size jede RTT bis ein Verlust erkannt wird

Multiplicative Decrease

Senderaten bei jedem Paketverlust halbieren



AIMD-Sägezahn-Verhalten: **Testen** der Bandbreite

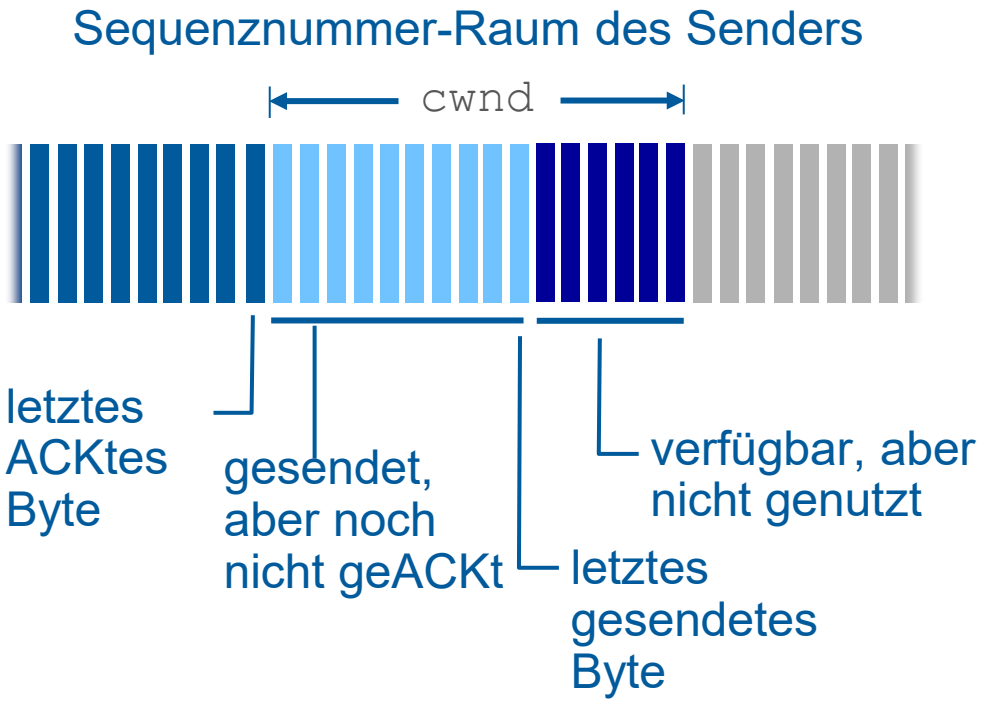
Multiplicative Decrease Details: Senderate wird

- halbiert im Falle eines Verlustes, erkannt durch ein Triple Duplicate ACK (TCP Reno)
- reduziert auf 1 MSS (Maximum Segment Size) wenn Verlust durch einen Timeout erkannt wird (TCP Tahoe)

Warum **AIMD?**

- AIMD – ein verteilter, asynchroner Algorithmus – von dem gezeigt wurde:
  - er optimiert die überlasteten Flussraten netzweit!
  - er hat wünschenswerte Stabilitätseigenschaften

TCP Überlastkontrolle: Details



TCP-Sendeverhalten:

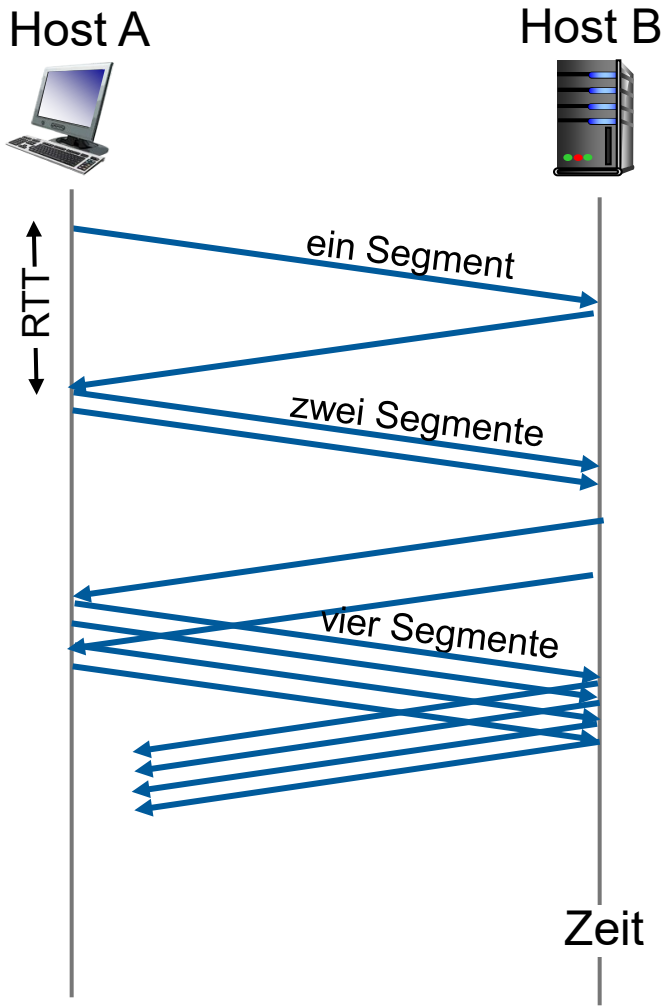
- **grob:** sende `cwnd` Byte, warte RTT auf ACKs, sende dann mehr Byte

$$\text{TCP-Rate} \approx \frac{\texttt{cwnd}}{\text{RTT}} \text{ Byte/s}$$

- TCP-Sender beschränkt die Übertragung: `LetztesByteGesendet - LetztesBytegeACKt ≤ cwnd`
- `cwnd` wird entsprechend der beobachteten Überlast im Netz dynamisch angepasst (implementiert TCP Überlastkontrolle)

TCP Slow Start

- zu Verbindungsbeginn, erhöhe Rate exponentiell bis zum ersten Verlust:
  - initial `cwnd` = 1 MSS
  - verdopple `cwnd` jede RTT
  - geschieht durch erhöhen des `cwnd` für jedes empfangene ACK
- **Zusammenfassung:** initiale Rate ist langsam, aber erhöht sich mit exponentieller Geschwindigkeit



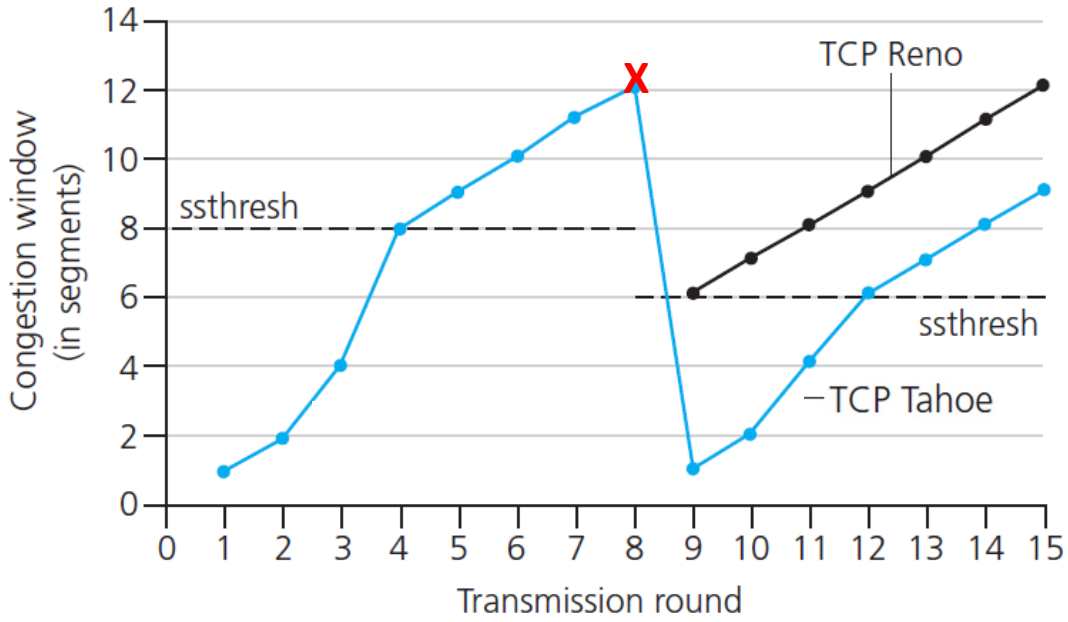
TCP: Vom Slow Start zur Überlastvermeidung

Frage: Wann sollte der exponentielle Anstieg linear werden?

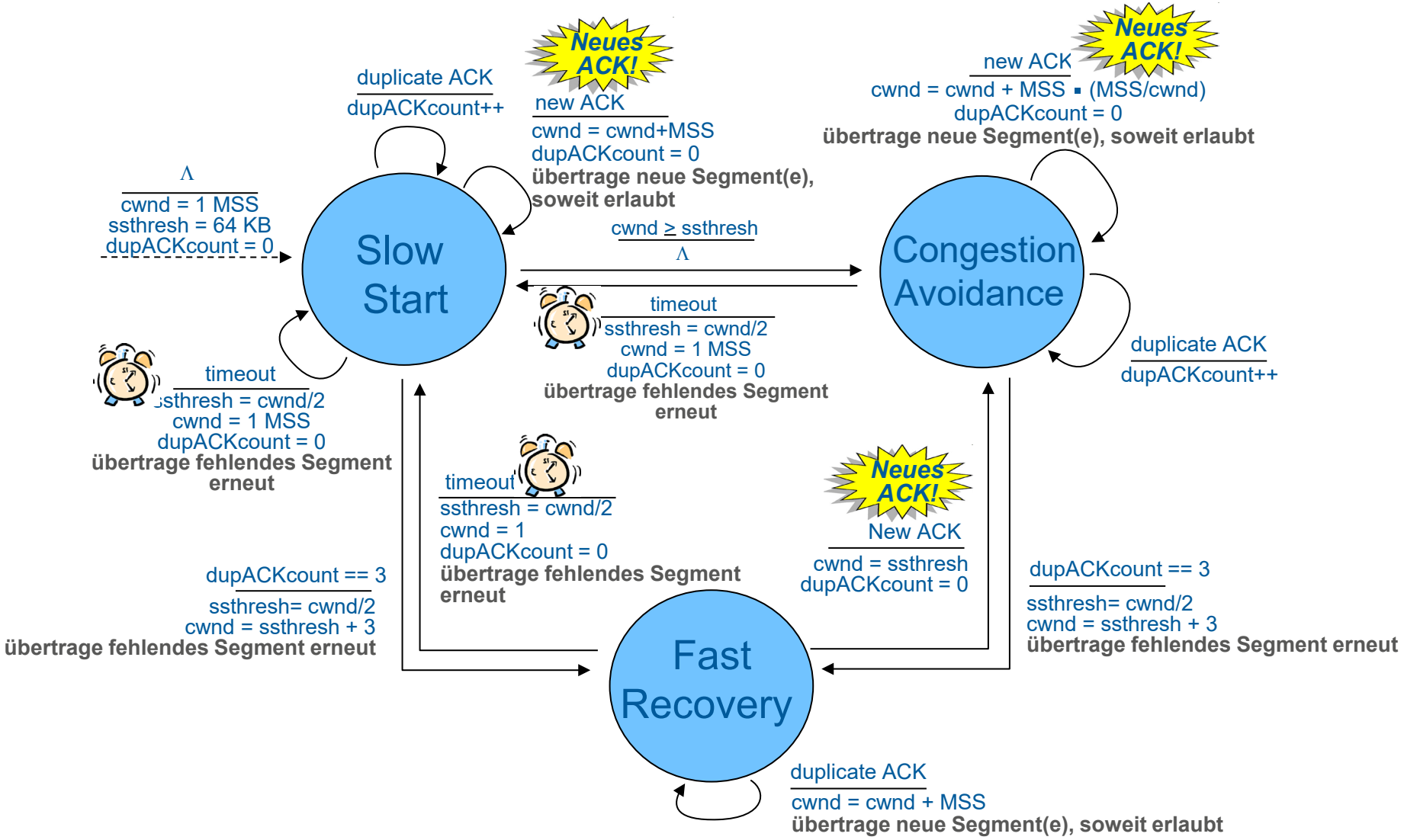
Antwort: wenn `cwnd` ½ seines Wertes vor dem Timeout erreicht.

Implementierung:

- Variable `ssthresh`
- bei Verlust wird `ssthresh` auf 1/2 `cwnd` kurz vor dem Verlust gesetzt

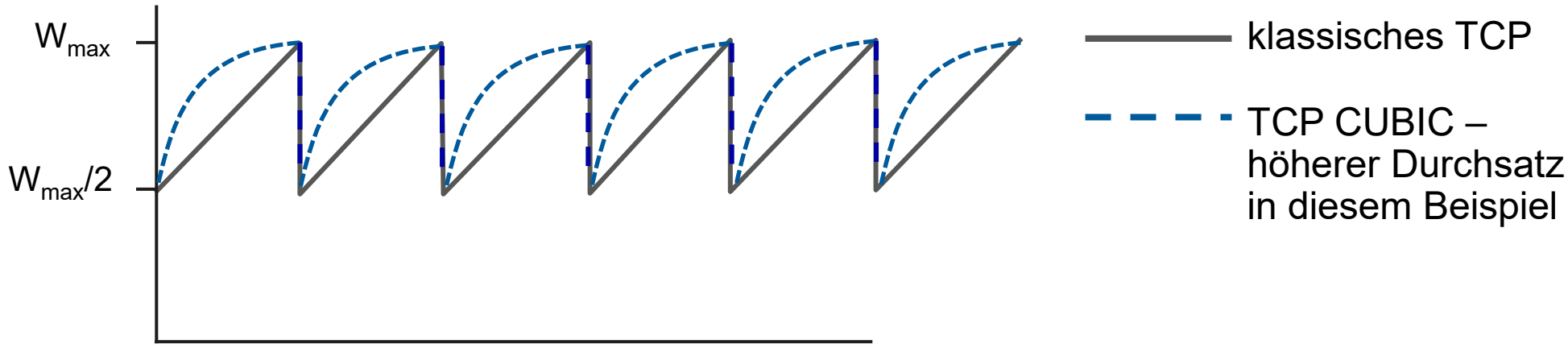


Zusammenfassung: TCP-Überlastkontrolle

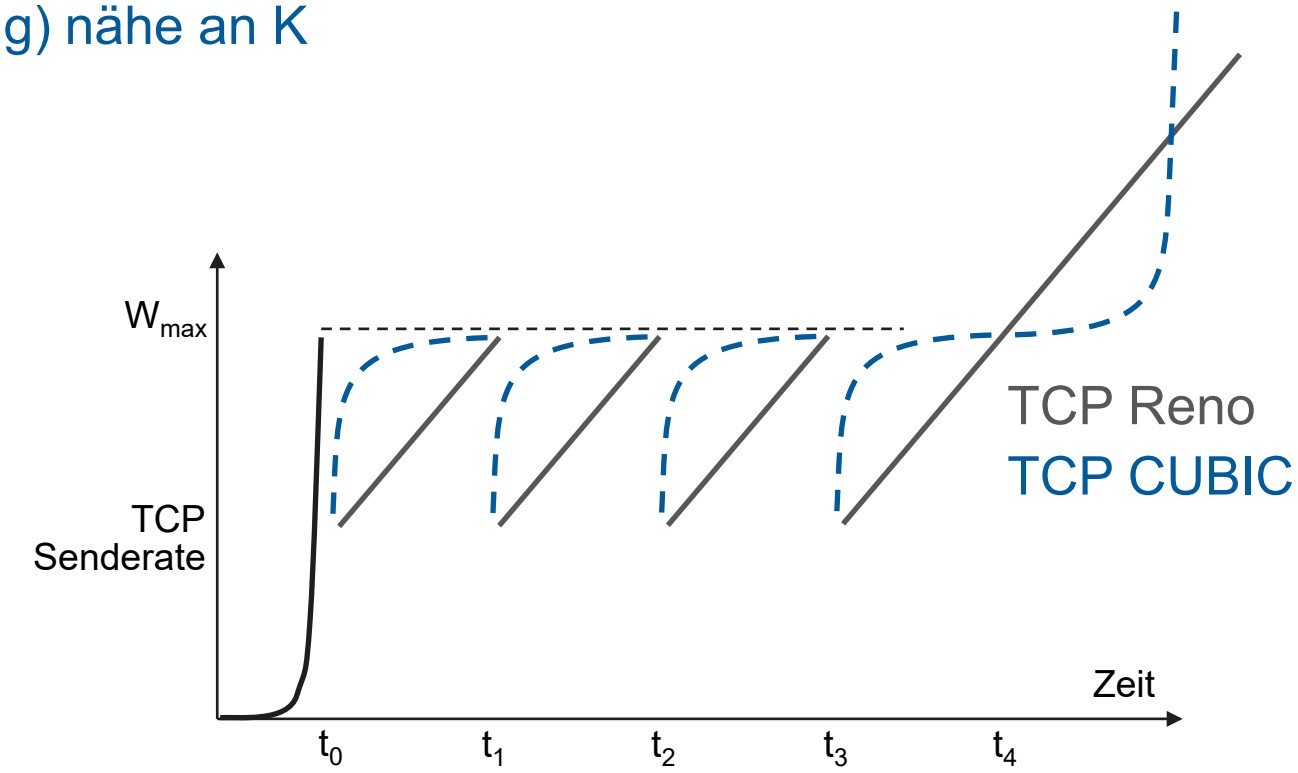


TCP CUBIC

- Gibt es einen besseren Weg als AIMD um auf nutzbare Bandbreite zu “testen”?
- Veranschaulichung:
  - W_{max}: Senderate bei der ein Verlust erkannt wurde
  - Überlastzustand des Flaschenhalslink hat sich wahrscheinlich (?) nicht sehr geändert
  - Nach der Reduktion der Senderate bei Verlust auf die Hälfte, initial *schnelleres* erhöhen der Rate bis zu W_{max}, aber dann *langsamere* Annäherung an W_{max}

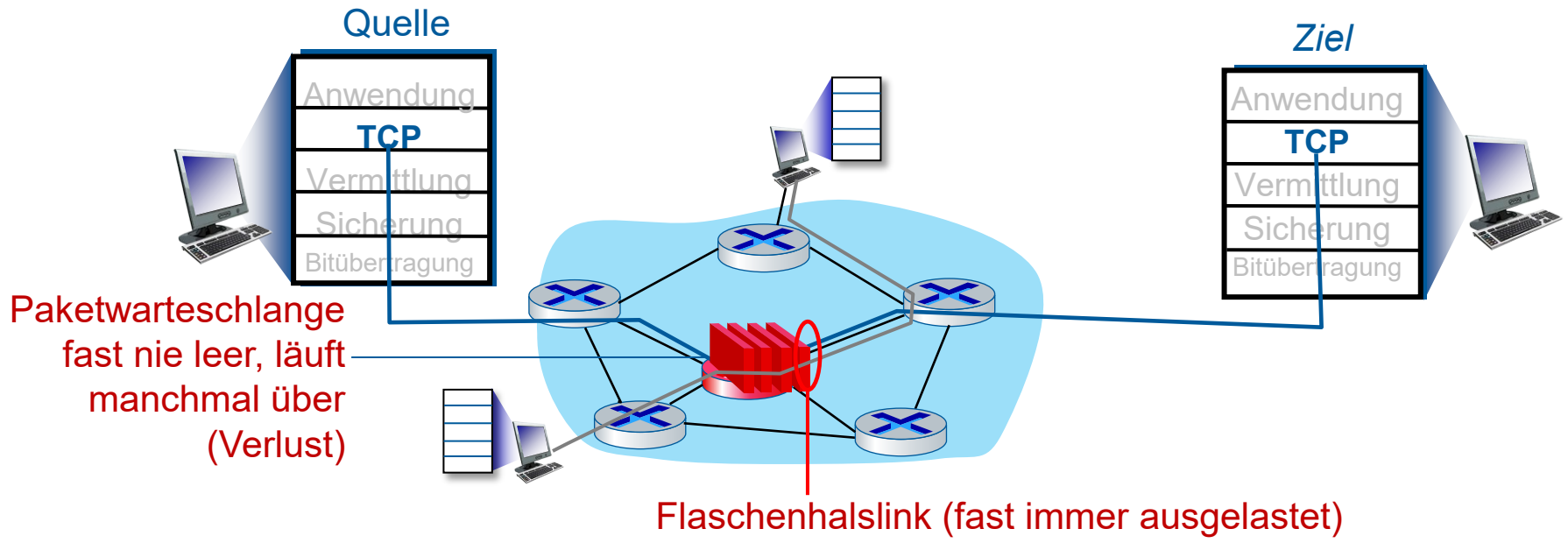


- K: Zeitpunkt zu dem das TCP Sendefenster W_{max} erreicht
  - K ist selbst einstellbar
- erhöhen von W nach einer **kubischen** Funktion des Abstands zwischen der aktuellen Zeit und K
  - größere Erhöhungen weiter weg von K
  - kleinere Erhöhung (vorsichtig) nähe an K
- TCP CUBIC Standard in Linux, populärste TCP-Variante bekannter Webserver



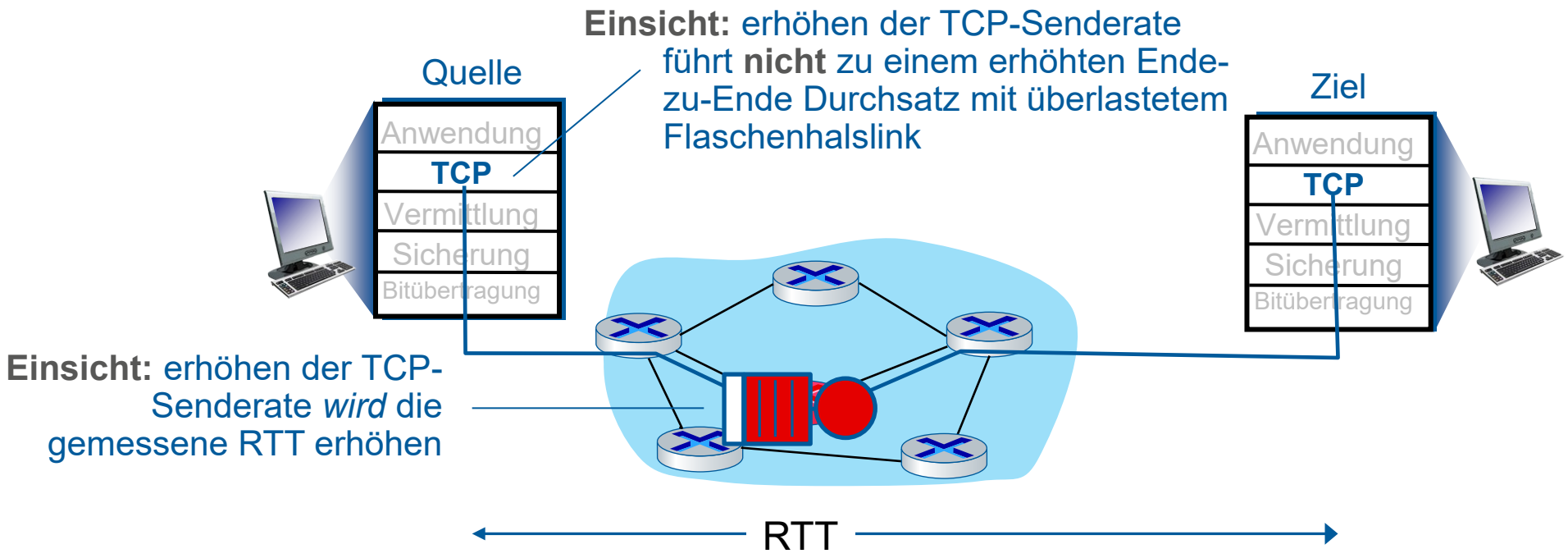
TCP und der überlastete "Flaschenhalslink"

- TCP (klassisch, CUBIC) erhöhen die TCP-Senderate bis Paketverlust an einem Router Ausgangslink auftritt: dem **Flaschenhalslink**



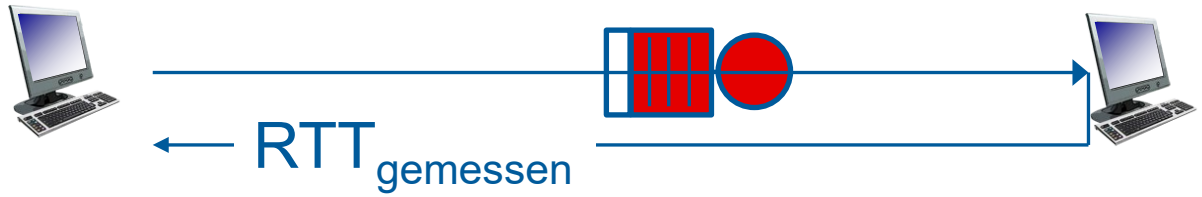
TCP und der überlastete "Flaschenhalslink"

- TCP (klassisch, CUBIC) erhöhen die TCP-Senderate bis Paketverlust an einem Router Ausgangslink auftritt: dem **Flaschenhalslink**
- Verstehen von Überlast: nützlich sich auf den überlasteten Flaschenhalslink zu konzentrieren



Ziel: "Die Ende-zu-Ende Leitung voll halten, aber nicht voller"

Die Sender-zu-Empfänger Leitung “voll halten, aber nicht voller”: den Flaschenhalslink ausgelastet halten, aber vermeiden von hohen Verzögerungen und Puffern



$$\text{gemessener Durchsatz} = \frac{\text{\# gesendeter Byte im letzten RTT Intervall}}{\text{RTT}_{\text{gemessen}}}$$

Verzögerungsbasierter Ansatz:

- RTT_{min} - minimal beobachtete RTT (Pfad nicht überlastet)
- nicht überlasteter Durchsatz mit Sendefenster `cwnd` ist cwnd/RTT_{min}

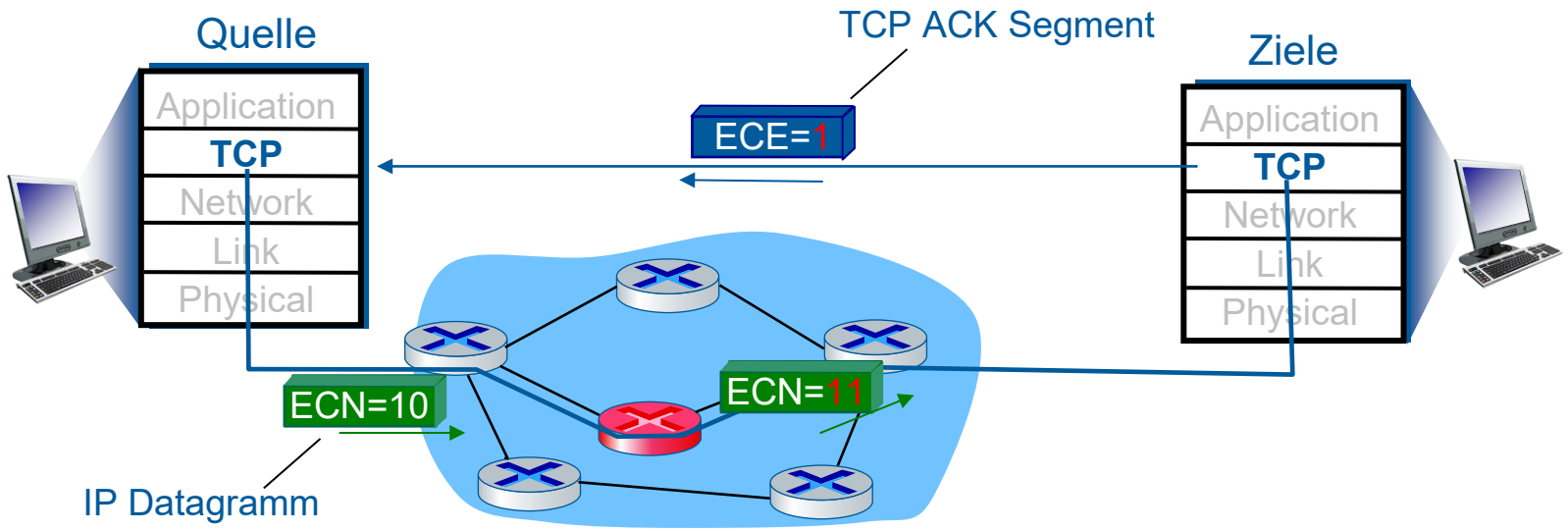
falls der gemessene Durchsatz “sehr nahe” am nicht überlasteten Durchsatz liegt,
    erhöhe `cwnd` linear /* da Pfad nicht überlastet */
ansonsten, falls der gemessene Durchsatz “weit unter” dem nicht überlasteten Durchsatz liegt,
    reduziere `cwnd` linear /* da Pfad überlastet */

Verzögerungsbasierte TCP-Überlastkontrolle

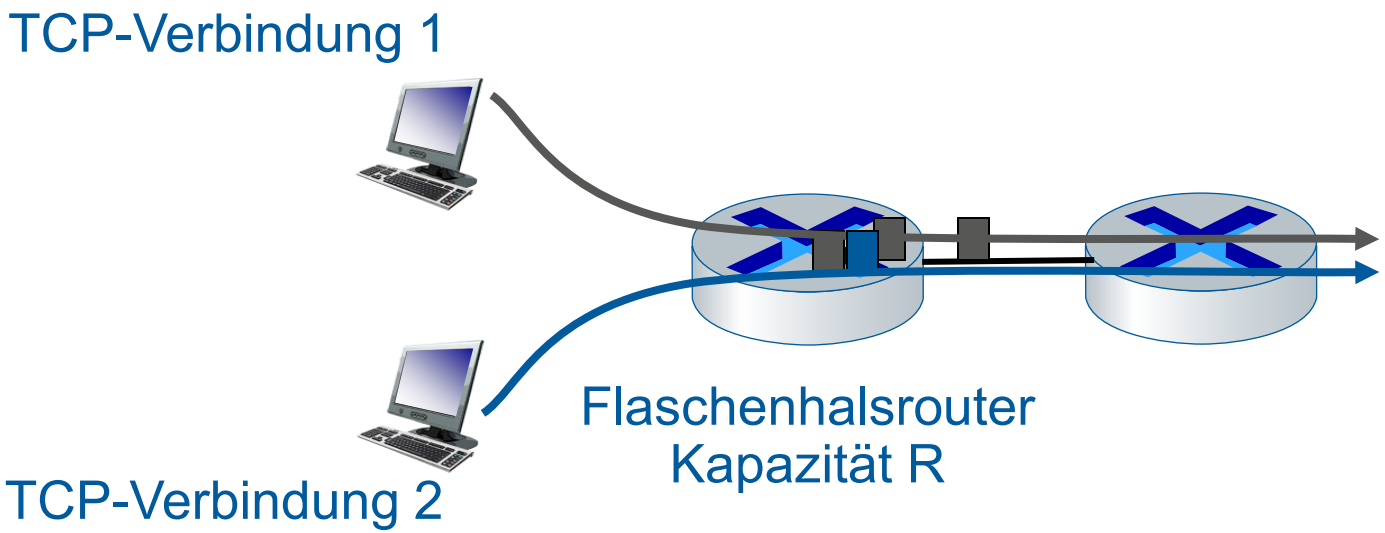
- Überlastkontrolle ohne erzwingen von Paketverlust
- Maximieren des Durchsatzes (“die Leitung gerade voll halten… ”), minimieren der Verzögerung (“…aber nicht voller”)
- einige eingesetzte TCP-Varianten wählen den verzögerungs-basierten Ansatz
  - BBR wird in Google’s (internem) Backbone Netz eingesetzt

Netz-unterstützte TCP-Überlastkontrolle ebenfalls möglich:

- zwei Bit im IP-Header (ToS Feld) werden *vom* **Router** markiert, um Überlast anzuzeigen
  - **Policy** bestimmt die vom Betreiber gewählte Markierung
- Überlasthinweis wird bis zum Zeil übertragen
- Ziel setzt ECE-Bit im ACK-Segment, um den Sender auf die Überlast hinzuweisen
- involviert sowohl IP (IP Header ECN Bit Markierung) und TCP (TCP Header C,E Bit Markierung)



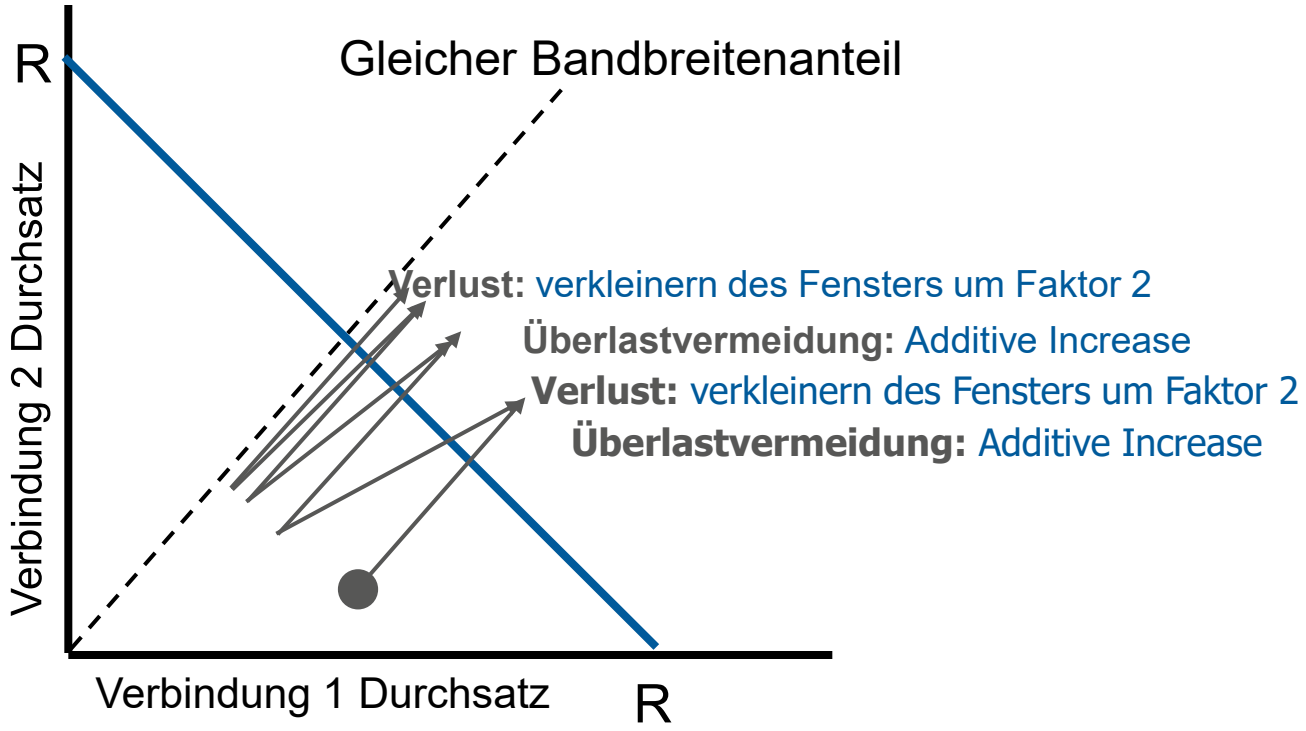
Fairness Ziel: falls *K* TCP Sitzungen sich denselben Flaschenhalslink mit Bandbreite *R* teilen, dann sollte jede Sitzung eine Durchschnittsrate von *R/K* haben



Frage: Ist TCP fair?

Beispiel: zwei konkurrierende TCP-Sitzungen:

- Additive Increase zeigt Steigung von 1, während sich der Durchsatz erhöht
- Multiplicative Decrease veringert den Durchsatz proportional



Ist TCP fair?

Antwort: Ja, unter idealisierten Annahmen:

- selbe RTT
- Nur feste Zahl von Sitzungen während der Überlastvermeidung

Fairness und UDP

- Multimedia Anwendungen verwendet TCP oft nicht
  - sie wollen ihre Übertragungsrate nicht von der Überlastkontrolle ausgebremst bekommen
- verwenden UDP stattdessen:
  - senden von Audio/Video mit einer konstanten Rate, tolerieren Verlust
- es gibt keine “Internet Polizei”, die das nutzen von Überlastkontrolle überwacht

Fairness, parallele TCP-Verbindungen

- Applikationen können *mehrere* parallele Verbindungen zwischen zwei Hosts aufbauen
- Web Browser machen das, z.B., Link mit Rate R mit 9 bestehenden Verbindungen:
  - neue App baut 1 TCP-Verbindung auf, erhält Rate R/10
  - neue App baut 11 TCP-Verbindungen auf, erhält Rate R/2

Transportschicht: Übersicht

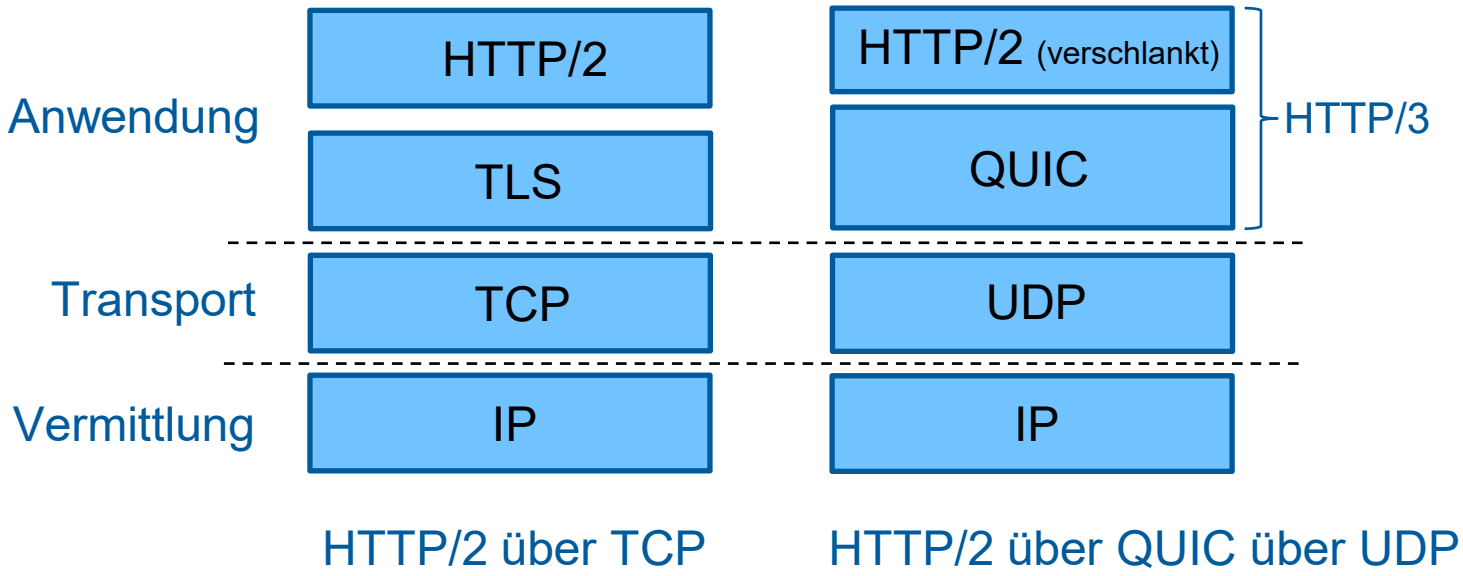
- Dienste der Transportschicht
- Multiplexen und Demultiplexen
- Verbindungsloser Transport: UDP
- Prinzipien verlässlicher Datenübertragung
- Verbindungsorientierter Transport: TCP
- Prinzipien der Überlastkontrolle
- TCP Überlastkontrolle
- **Evolution der Transportschicht Funktionen**

- TCP, UDP: Wichtigste Transportprotokolle seit 40 Jahren
- verschiedene Varianten von TCP wurden für spezielle Szenarien entwickelt:

Szenario	Herausforderungen
Lange, dicke Leitungen (Große Datentransfers)	Viele Pakete “im Flug” unterwegs; Verlust stoppt den Fluss
Drahtlose Netze	Verlust durch fehleranfällige Funkstrecken, Mobilität; TCP behandelt das als Überlast
Links mit hoher Verzögerung	Sehr lange RTTs
Datenzentrumsnetze	Verzögerungsempfindlich
Verkehrsflüsse im Hintergrund	Niederpriore, “Hintergrund” TCP-Ströme

- Verlagern von Transportschicht Funktionen in die Applikationsschicht, aufbauend auf UDP
  - HTTP/3: QUIC

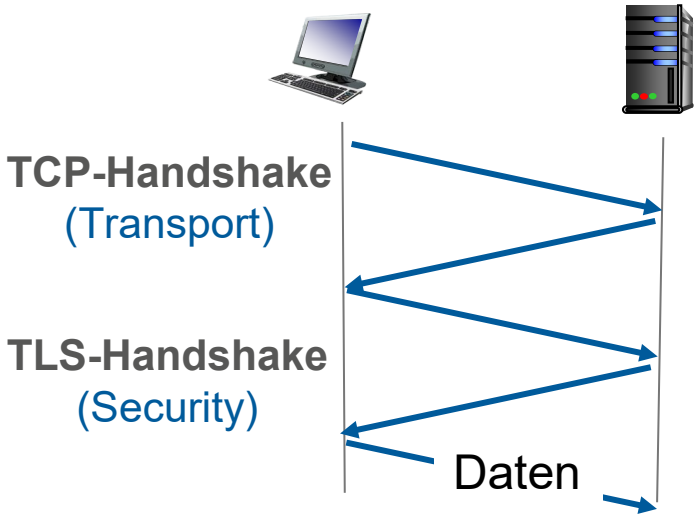
- Protokoll der Anwendungsschicht über UDP
  - Erhöhen der Leistung von HTTP
  - Wird von vielen Google Servern, Applikationen genutzt (Chrome, YouTube App)



Übernimmt die Ansätze aus diesem Kapitel für Verbindungsaufbau Fehlerkontrolle, Überlastkontrolle

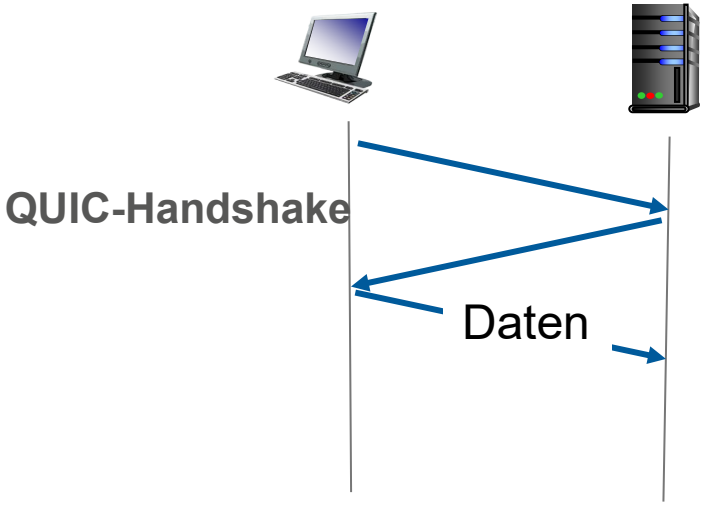
- **Fehler und Überlastkontrolle:** “Readers familiar with TCP’s loss detection and congestion control will find algorithms here that parallel well-known TCP ones.” [aus der QUIC-Spezifikation]
- **Verbindungsaufbau:** Zuverlässigkeit, Überlastkontrolle, Authentifizierung, Verschlüsselung, Zustand werden in nur einer RTT hergestellt
- mehrere Applikations“ströme” werden über eine einzelne QUIC-Verbindung multiplext
  - unabhängige Zuverlässigkeit, Sicherheit
  - Gemeinsame Überlastkontrolle

QUIC: Verbindungsaufbau



TCP (Zuverlässigkeit, Überlastkontrolle) + **TLS** (Authentifizierung, Verschlüsselung)

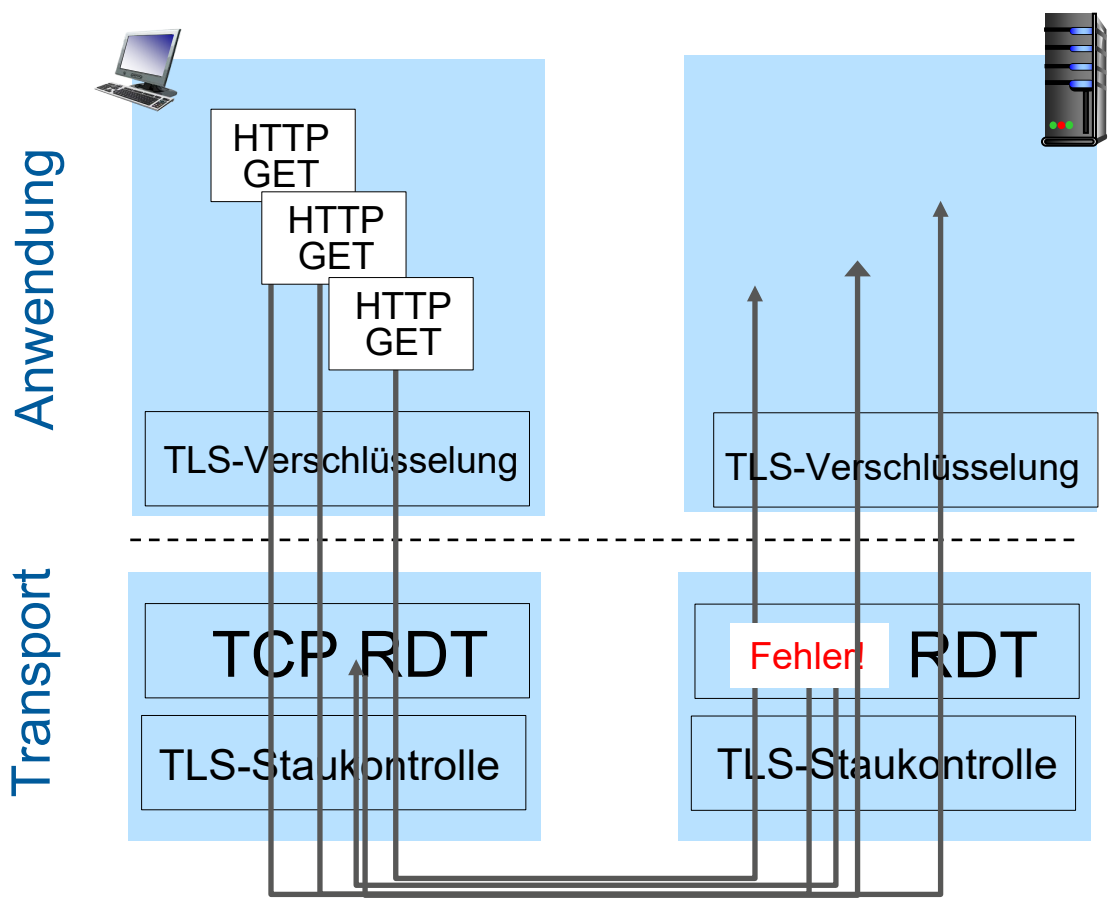
- 2 aufeinanderfolgende Handshakes



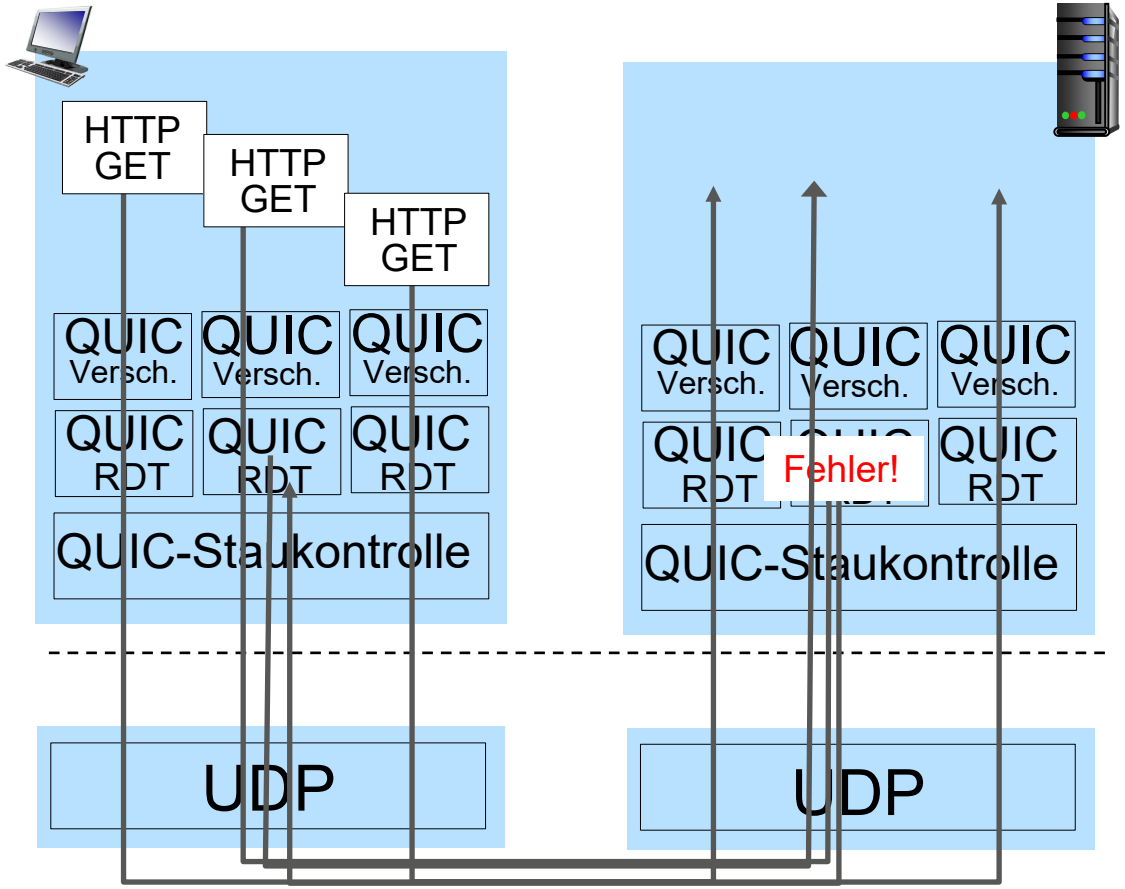
QUIC: Zuverlässigkeit, Überlastkontrolle, Authentifizierung, Verschlüsselung

- 1 Handshake

QUIC: Streams: Parallelität, keine HOL-Blockierung



(a) HTTP 1.1



(b) HTTP/2 mit QUIC: no HOL blocking