Netzwerktechnologien
3 VO

Univ.-Prof. Dr. Helmut Hlavacs
helmut.hlavacs@univie.ac.at

Dr. Ivan Gojmerac
gojmerac@ftw.at

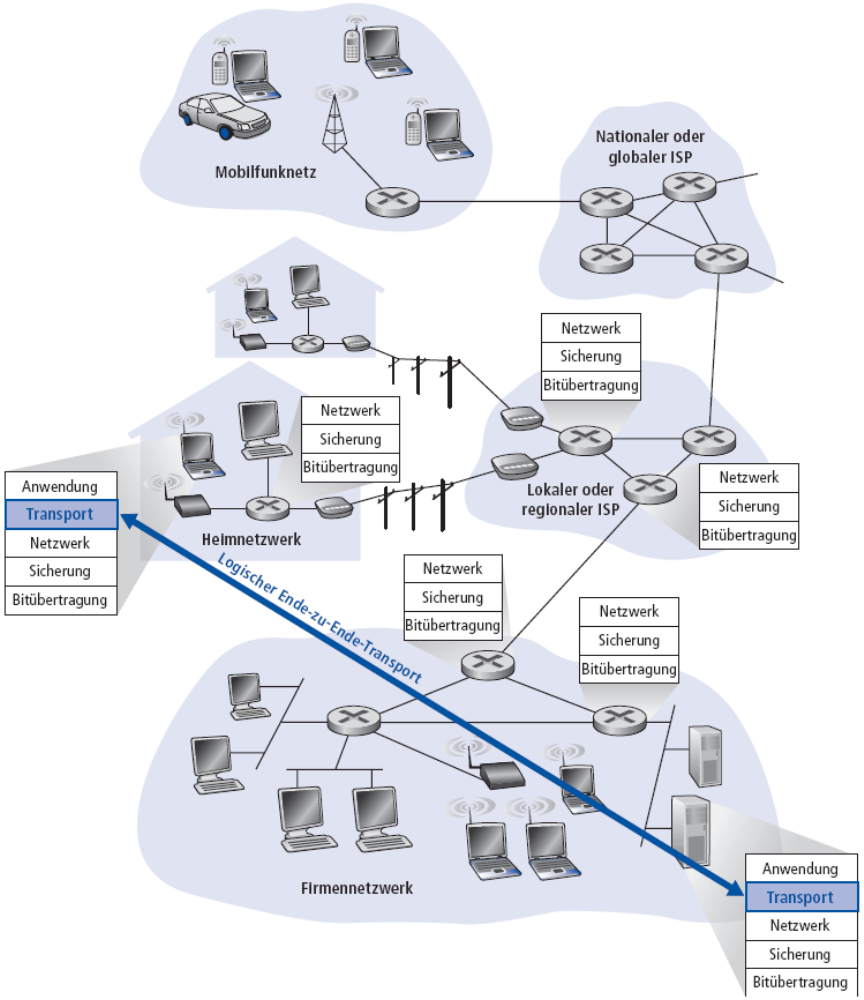
Bachelorstudium Medieninformatik
SS 2012

Kapitel 3 – Transportschicht

3.1 Dienste der Transportschicht
3.2 Multiplexing und Demultiplexing
3.3 Verbindungsloser Transport: UDP
3.4 Grundlagen der zuverlässigen Datenübertragung
3.5 Verbindungsorientierter Transport: TCP
3.6 Grundlagen der Überlastkontrolle
3.7 TCP-Überlastkontrolle

3.1 Dienste der Transportschicht

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



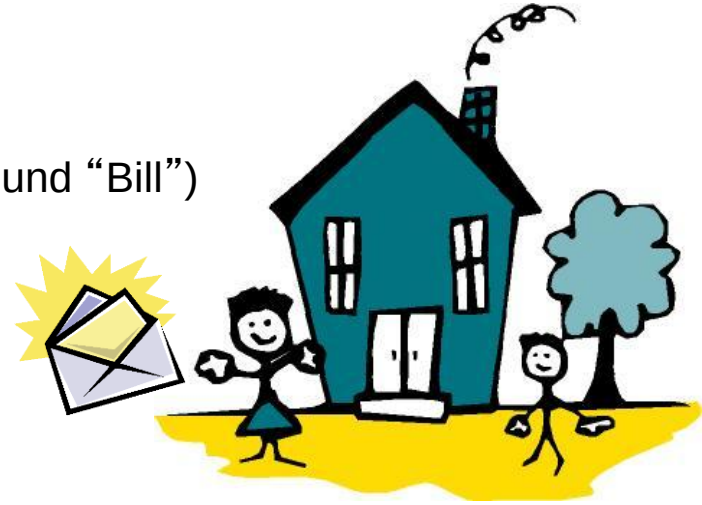
3.1 Unterschied: Transport- und Netzwerkschicht

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

<u>Analogie: Briefzustellung an mehrere Bewohner des selben Hauses</u>

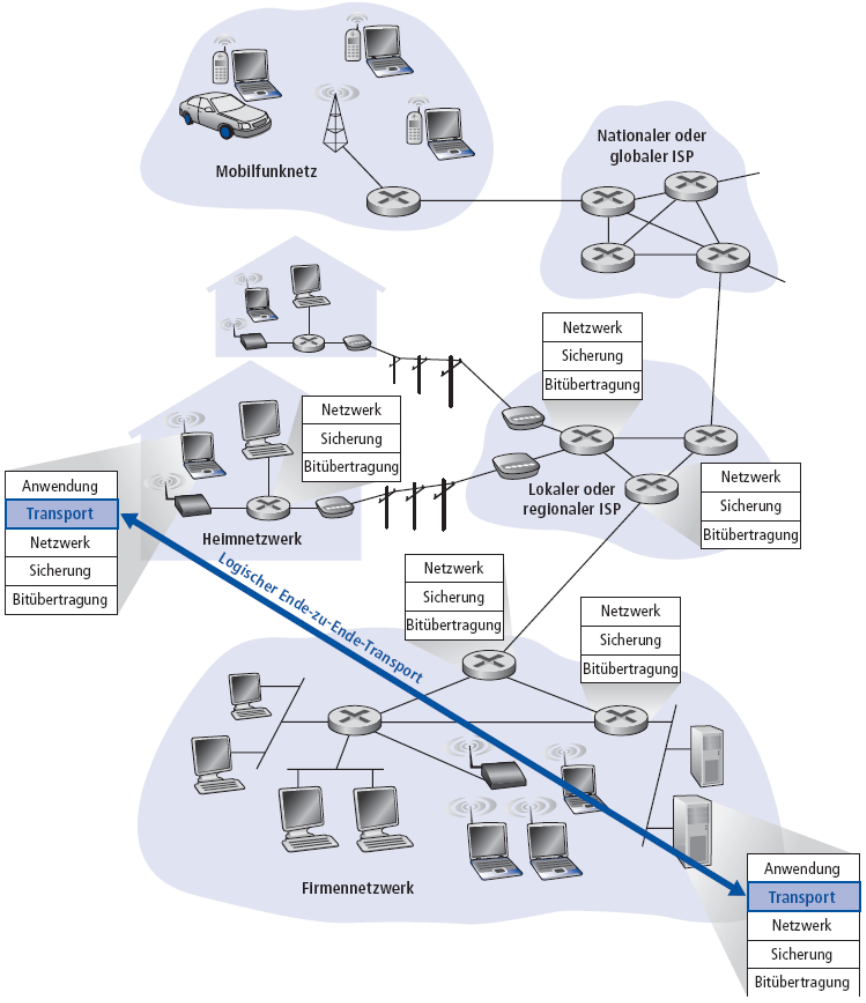
→ *12 Kinder senden Briefe an 12 andere Kinder.*

- Prozess = Kind
- Anwendungsnachricht = Brief in einem Umschlag
- Host = Haus
- Transportprotokolle = Namen der Kinder (z.B. “Anne” und “Bill”)
- Netzwerkprotokoll = Postdienst

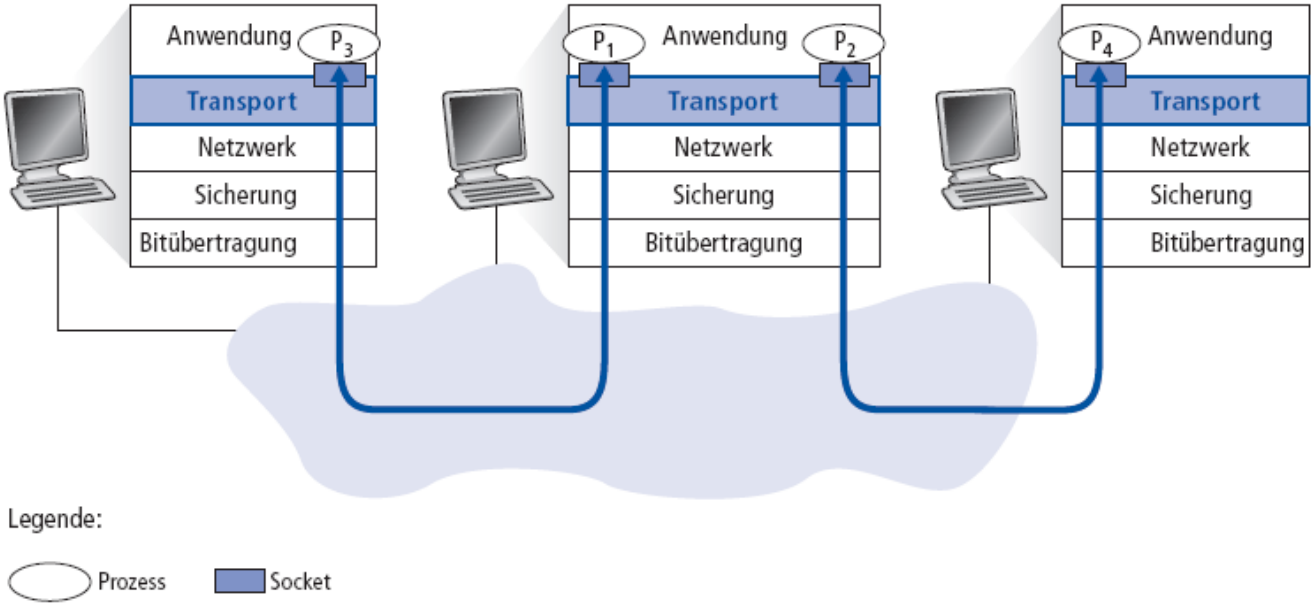


3.1 Transportprotokolle im Internet

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



3.2 Multiplexing und Demultiplexing



- **Multiplexing** beim Sender:

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

- **Demultiplexing** beim Empfänger:

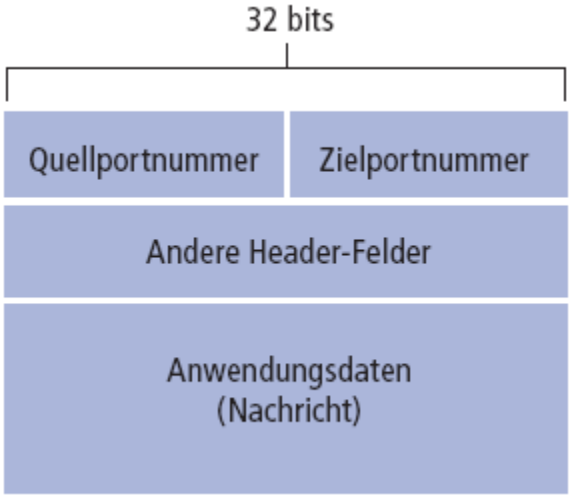
Empfangene Segmente am richtigen Socket abliefern.

3.2.1 Demultiplexing

Analogie: Briefzustellung an mehrere Bewohner des selben Hauses

Bill erhält einen Stapel Briefe vom Briefträger und gibt jeden davon dem Kind, dessen Name auf dem Brief steht.

- Host empfängt IP-Datagramme
  - Jedes Datagramm hat eine Absender-IP-Adresse und eine Empfänger-IP-Adresse
  - Jedes **Datagramm** beinhaltet ein **Transportschichtsegment**
  - 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

3.2.1 Verbindungsloses Demultiplexing (UDP)

- Sockets mit Portnummer anlegen:

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

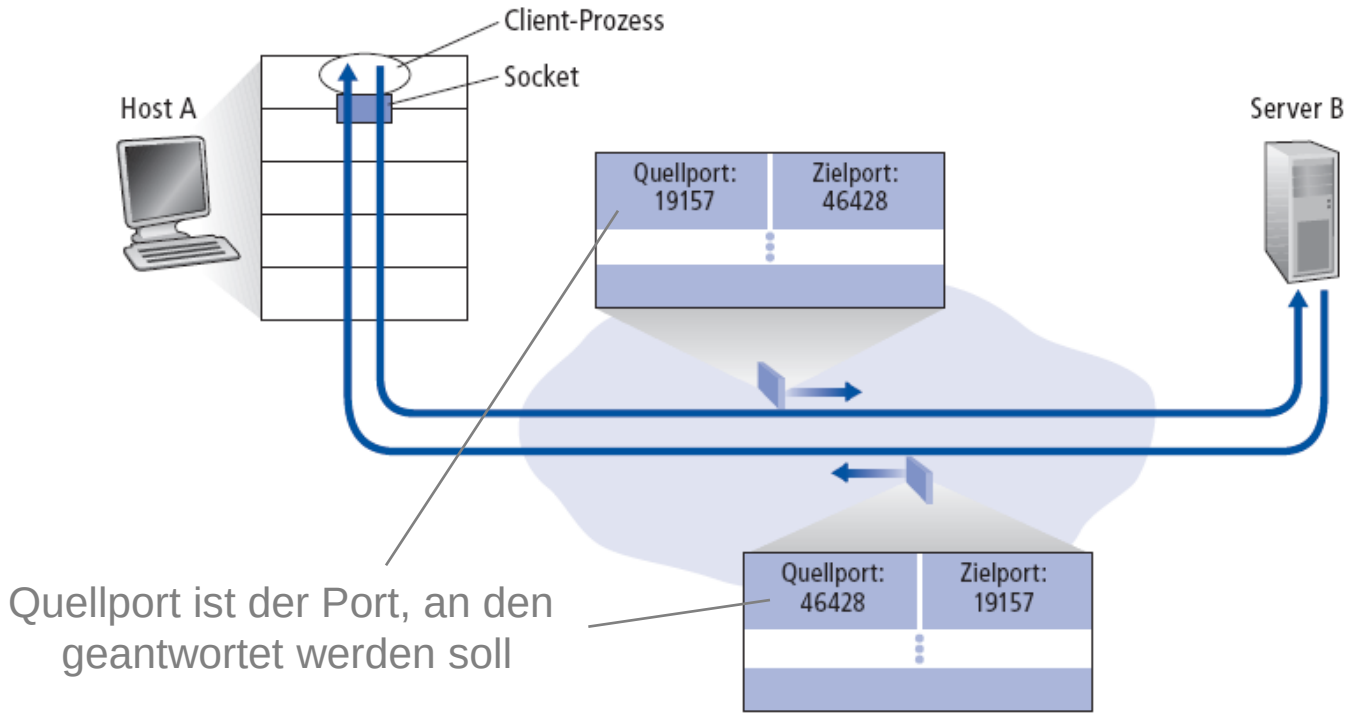
- **UDP**-Socket wird durch ein 2-Tupel identifiziert:

(Empfänger-IP-Adresse, Empfänger-Portnummer)

- Wenn ein Host ein UDP-Segment empfängt:
  1. Lese Empfänger-**Portnummer**
  2. 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

3.2.1 Verbindungsloses Demultiplexing

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



3.2.2 Verbindungsorientiertes Demultiplexing (TCP)

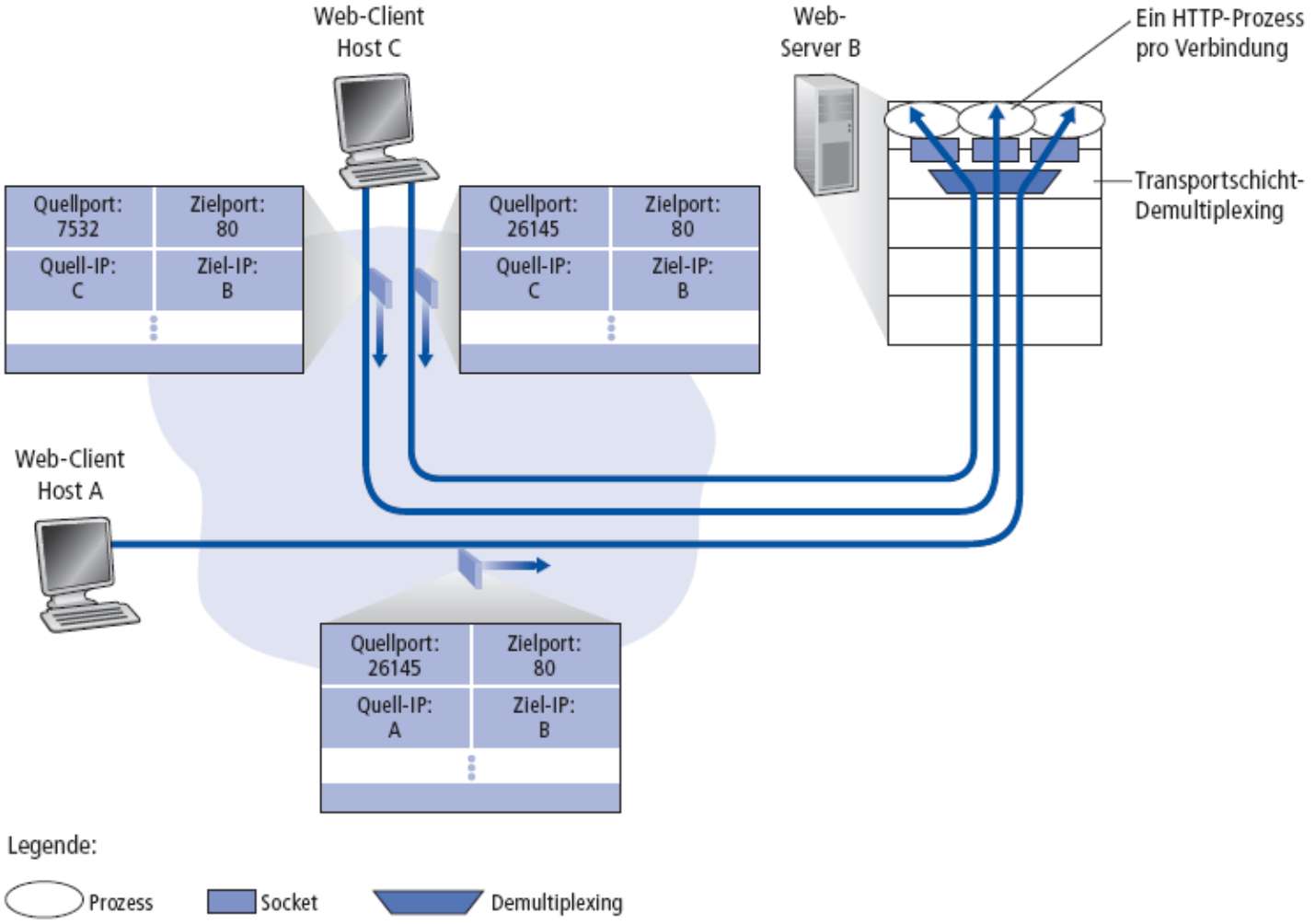
- **TCP**-Socket wird durch ein 4-Tupel identifiziert:
  - Absender-IP-Addresse
  - Absender-Portnummer
  - Empfänger-IP-Adresse
  - Empfänger-Portnummer

→ *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)

3.2.2 Verbindungsorientiertes Demultiplexing (TCP)



3.3 Verbindungsloser Transport: UDP

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

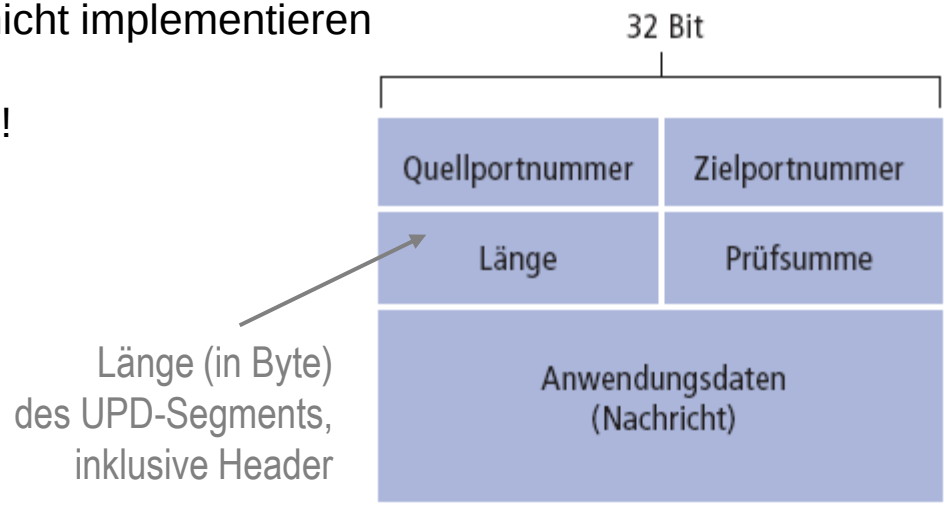
Warum gibt es UDP?

- 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

3.3 Verbindungsloser Transport: UDP

- 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 Segmentformat

3.3.1 Fehlerkorrekturmechanismus: 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

3.3.1 Fehlerkorrekturmechanismus: UDP Prüfsumme

Zahlen werden addiert und ein Übertrag aus der höchsten Stelle wird zum Resultat an der niedrigsten Stelle addiert.

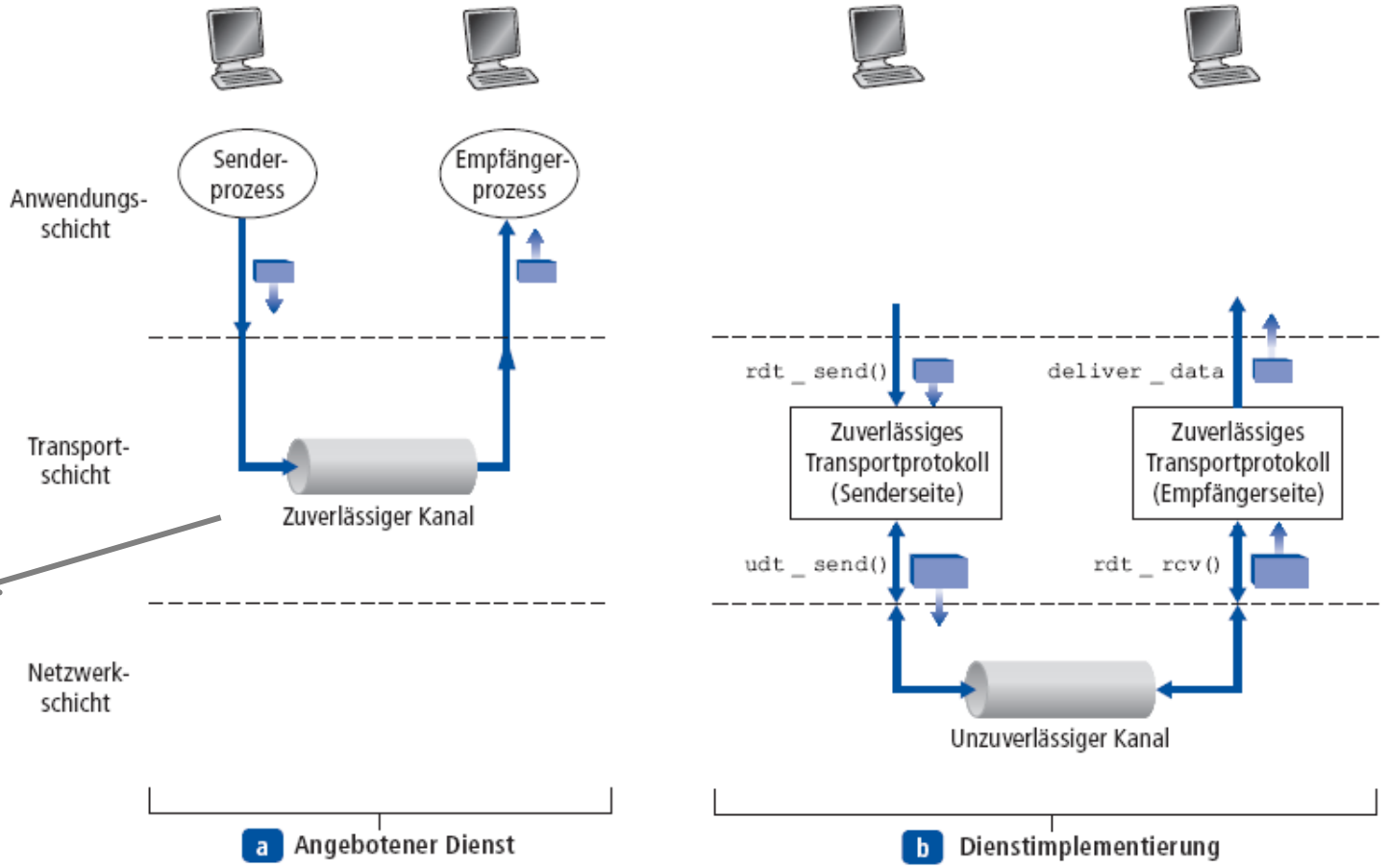
Beispiel:
Addiere zwei 16-Bit-Integer-Werte.

	1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
+	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
Übertrag	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
Summe (mit Übertrag)	1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
Prüfsumme	0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

3.4 Zuverlässigkeit der Datenübertragung

- Wichtig für Anwendungs-, Transport- und Sicherungsschicht
- Schicht unterhalb des zuverlässigen Datentransferprotokolls kann unzuverlässig sein

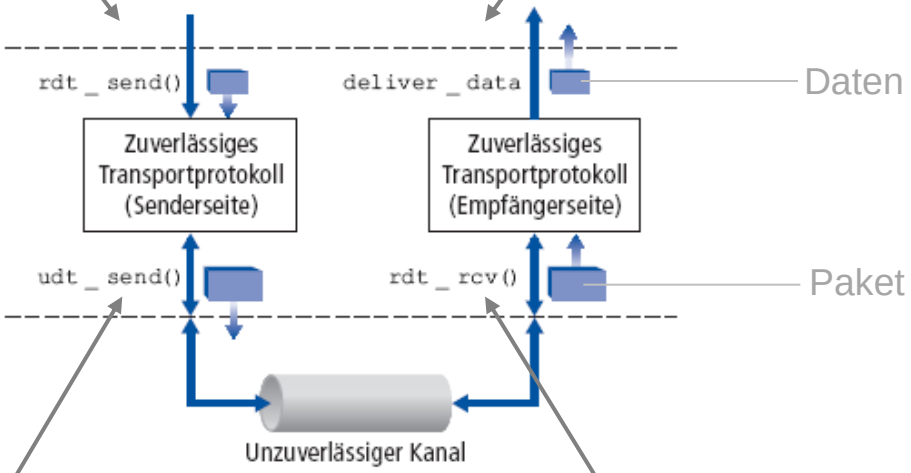
Eigenschaften des unzuverlässigen Kanals bestimmen die Komplexität des Protokolls zur zuverlässigen Datenübertragung (rdt)



3.4 Grundlagen der zuverlässigen Datenübertragung

`rdt_send()`: von oben (z.B. der Anwendung) aufgerufen, übergebene Daten sollen beim Empfänger nach oben ausgeliefert werden

`deliver_data()`: von rdt aufgerufen, um Daten nach oben weiterzugeben



`udt_send()`: von rdt aufgerufen, um Paket über unzuverlässigen Kanal zum Empfänger zu übertragen

`rdt_rcv()`: von unten aufgerufen, wenn ein Paket auf der Empfängerseite des unzuverlässigen Kanals ankommt

* rdt = reliable data transfer (zuverlässiger Datentransfer)
udt = unreliable data transfer (unzuverlässiger Datentransfer)

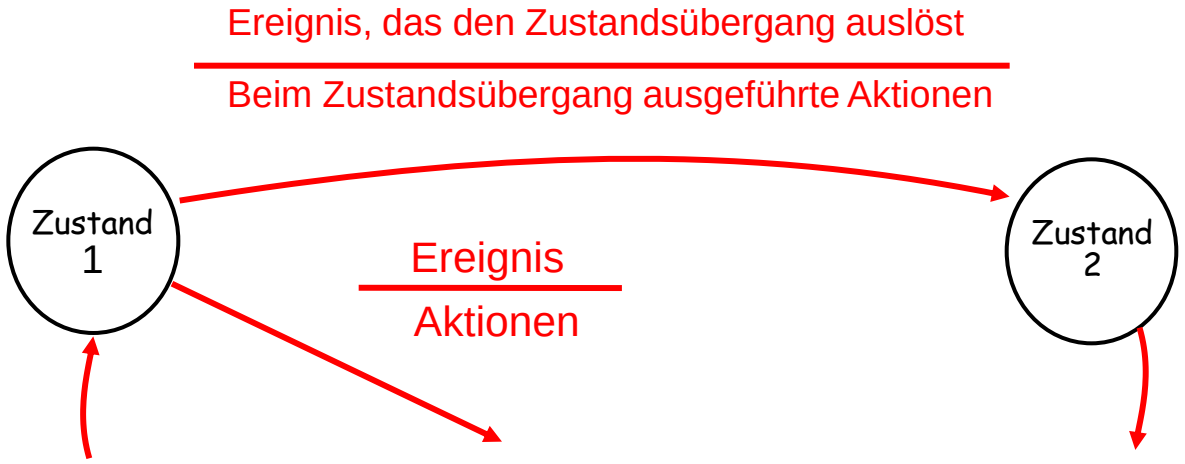
3.4.1 Aufbau eines zuverlässigen Datentransferprotokolls

- Immer komplexer werdender Protokolle
- Ziel: einwandfreies, zuverlässiges Datentransferprotokoll (*rdt Protokoll*)
  - Trotz unzuverlässigen Kanals.
- Zunächst nur unidirektionaler Datenverkehr
  - Kontrollinformationen fließen in **beide** Richtungen!
- Endliche Automaten

3.4.1 Endliche Automaten

- Finite state machine = FSM
- Um Sender und Empfänger zu spezifizieren

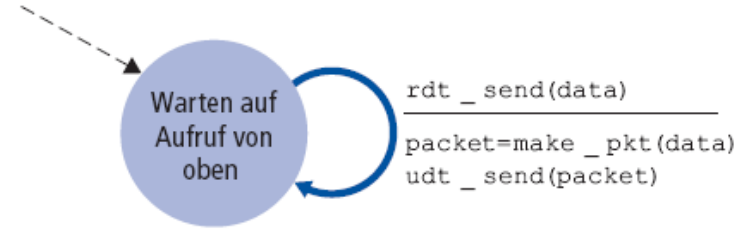
Zustand: Durch den Zustand und ein Ereignis ist der Nachfolgezustand bestimmt



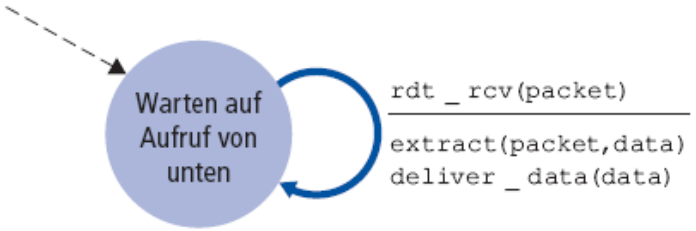
3.4.1 Aufbau eines zuverlässigen Datentransferprotokolls

Zuverlässiger Datentransfer über einen perfekt zuverlässigen Kanal: **rdt1.0**

- Der Übertragungskanal ist absolut zuverlässig:
  - Keine Verfälschung von Bits
  - Kein Verlust ganzer Rahmen/Pakete
- Je ein endlicher Automat für Sender und Empfänger:
  - Sender übergibt Daten an den zuverlässigen Kanal
  - Empfänger erhält die Daten vom zuverlässigen Kanal



a rdt1.0-Sender



b rdt1.0-Empfänger

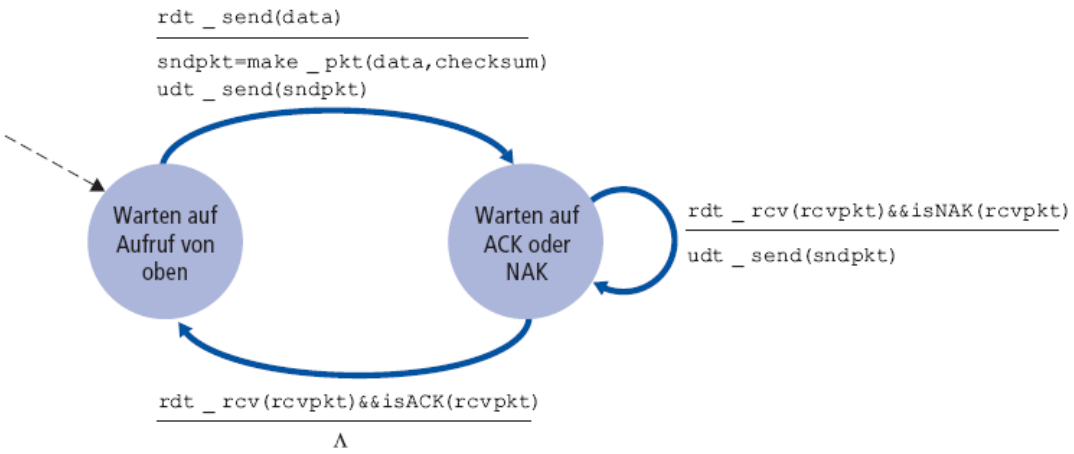
3.4.1 Aufbau eines zuverlässigen Datentransferprotokolls

Zuverlässiger Datentransfer über einen Kanal mit Bitfehlern: **rdt2.0**

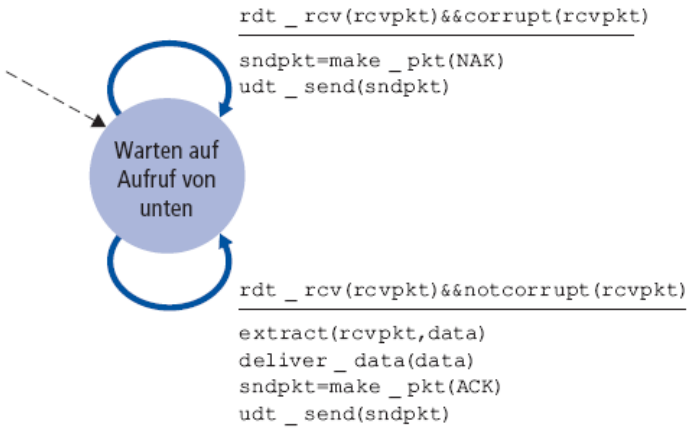
- Verfälschte Bits sind durch eine Prüfsumme erkennbar
- Mechanismen zur zuverlässigen Datenübertragung:
  - *Acknowledgements (ACKs):*

    Empfänger sagt dem Sender explizit, dass das Paket erfolgreich empfangen wurde.
  - *Negative Acknowledgements (NAKs):*

    Empfänger sagt dem Sender explizit, dass das Paket fehlerbehaftet war. Sender wiederholt Übertragung für diese Pakete.



a rdt2.0-Sender



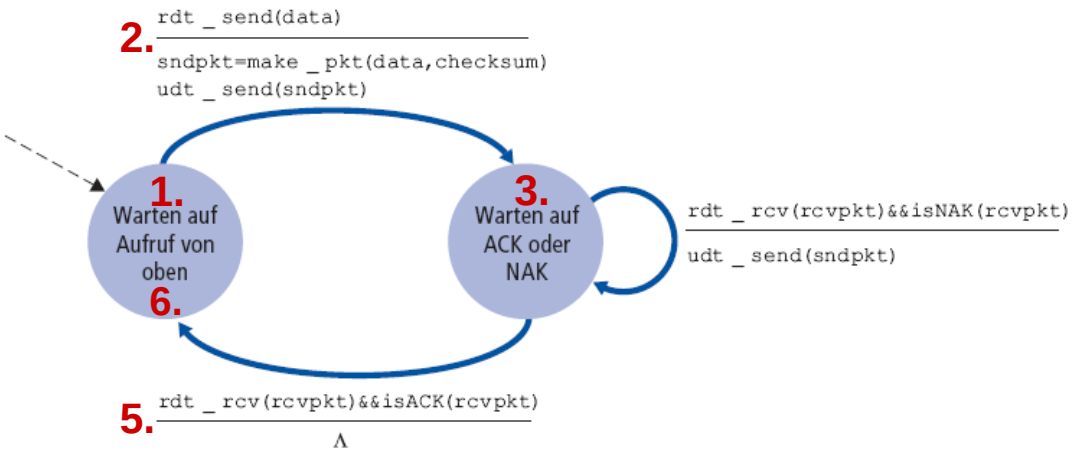
b rdt2.0-Empfänger

3.4.1 Aufbau eines zuverlässigen Datentransferprotokolls

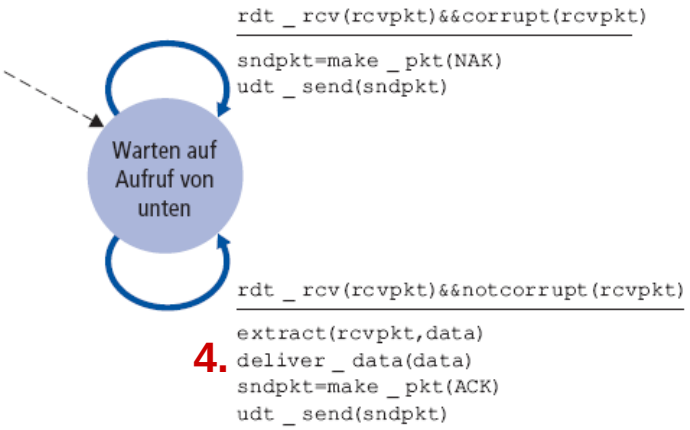
Zuverlässiger Datentransfer über einen Kanal mit Bitfehlern: **rdt2.0**

- Neue Mechanismen in rdt2.0:
  - Fehlererkennung
  - Kontrollnachrichten vom Empfänger an den Sender, → ARQ-Protokoll (ARQ = Automatic Repeat reQuest, automatische Wiederholungsanfrage)

Ablauf **ohne** Fehler:



a rdt2.0-Sender



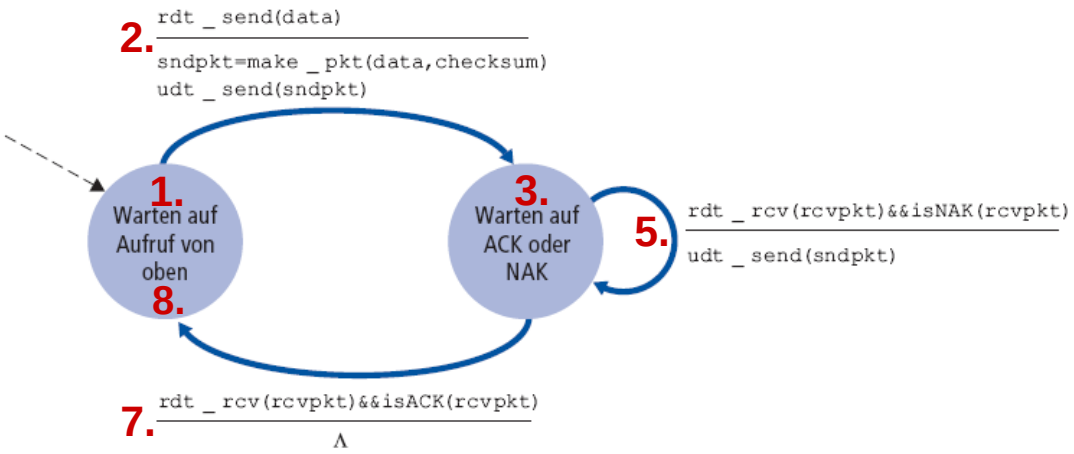
b rdt2.0-Empfänger

3.4.1 Aufbau eines zuverlässigen Datentransferprotokolls

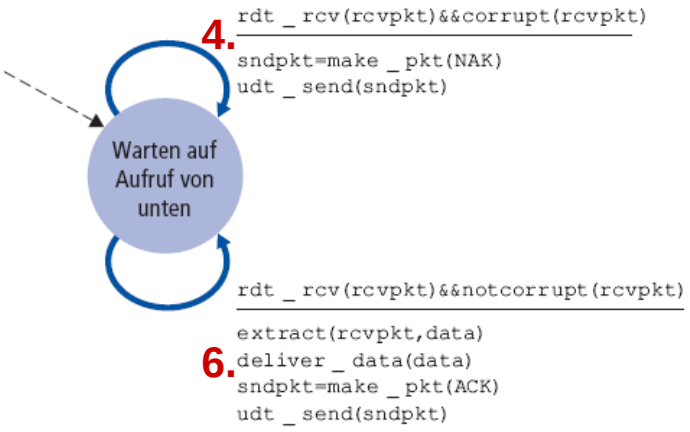
Zuverlässiger Datentransfer über einen Kanal mit Bitfehlern: **rdt2.0**

- Neue Mechanismen in rdt2.0:
  - Fehlererkennung
  - Kontrollnachrichten vom Empfänger an den Sender, → ARQ-Protokoll (ARQ = Automatic Repeat reQuest, automatische Wiederholungsanfrage)

Ablauf **mit** Fehler:



a rdt2.0-Sender



b rdt2.0-Empfänger

3.4.1 Aufbau eines zuverlässigen Datentransferprotokolls

Zuverlässiger Datentransfer über einen Kanal mit Bitfehlern: **rdt2.0**

→ **ABER: ACK/NAK**-Pakete können auch **fehlerhaft** sein!

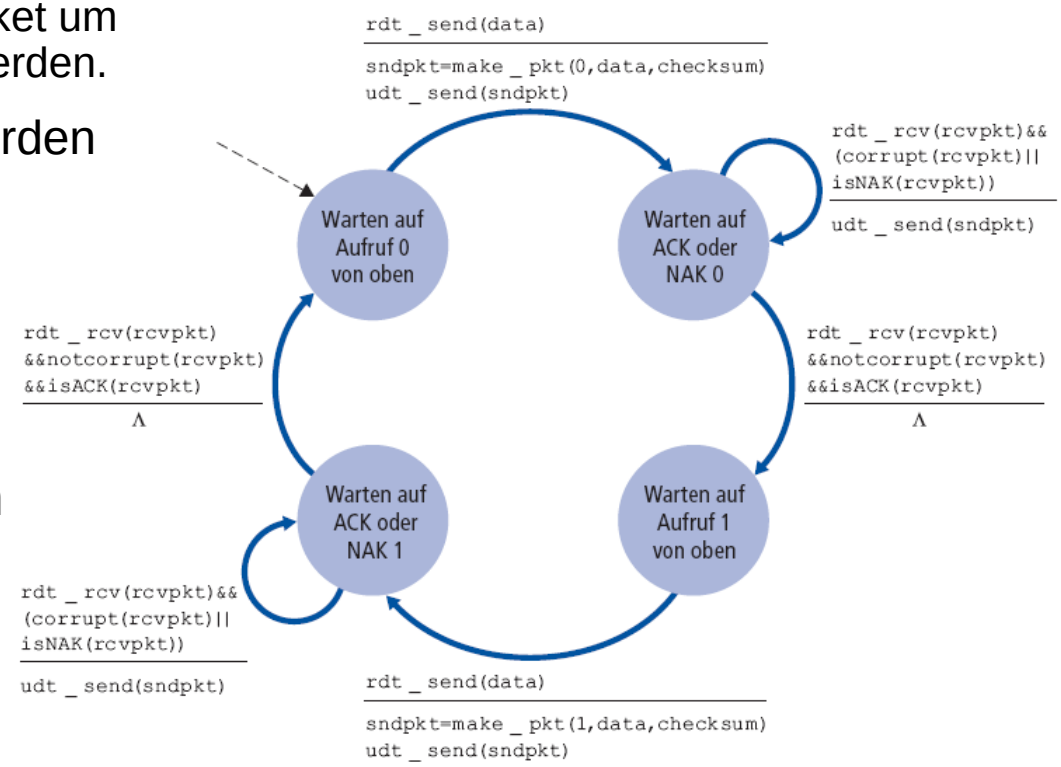
- Bei fehlerhaften ACK/NAK weiß Sender nicht, was beim Empfänger passiert ist
- Unzufriedenstellende Lösungsansätze:
  - **Sender sendet ACK/NAK für jedes ACK/NAK des Empfängers.**
    Doch was passiert, wenn dieses verfälscht wird?
  - **Genügend Prüfsummenbits, um dem Absender zu ermöglichen Bitfehler zu erkennen und zu korrigieren.**
    Lösung funktioniert nur für einen Kanal der Pakete zwar verändern aber nicht verlieren kann.
  - **Übertragungswiederholung.**
    Kann zur erneuten Übertragung eines bereits korrekt empfangenen Pakets führen.
  - Verbreiteter Lösungsansatz:
    Fortlaufende Sequenznummer in Datenpakete einfügen.

→ Bei Stop-and-Wait-Protokollen genügt eine 1-Bit-Zahl zum Vergleich der Sequenznummer mit der des zuletzt empfangenen Paketes.

3.4.1 Aufbau eines zuverlässigen Datentransferprotokolls

rdt2.1: Sender mit Behandlung von **verfälschten ACKs/NAKs**

- Sequenznummern hinzugefügt
- Zwei Sequenznummern reichen (0,1).
  Sind die Sequenznummern zweier aufeinanderfolgender Pakete gleich, so handelt es sich bei dem zweiten Paket um ein **Duplikat** und kann verworfen werden.
- Verfälschte ACKs und NAKs werden korrekt behandelt



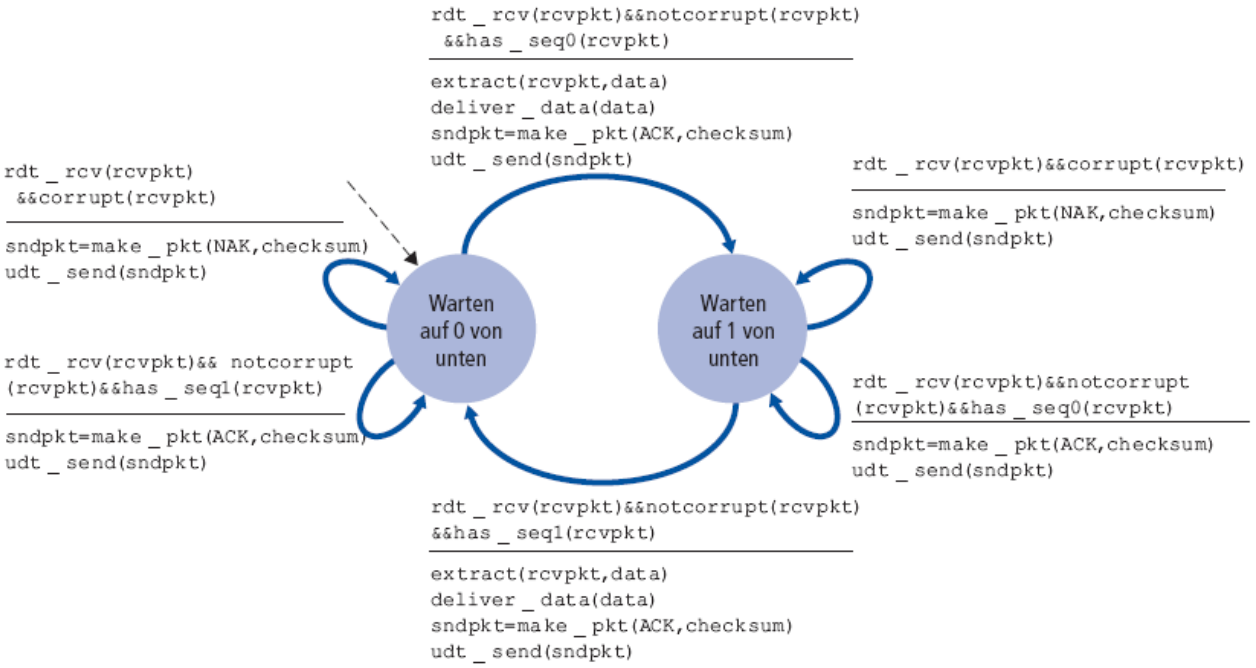
→ Doppelte Anzahl von Zuständen im Vergleich zu rdt2.0:
Zustände müssen sich „merken“, ob das aktuelle Paket die Sequenznummer 0 oder 1 hat

3.4.1 Aufbau eines zuverlässigen Datentransferprotokolls

rdt2.1: Empfänger mit Behandlung von verfälschten ACKs/NAKs

- Muss überprüfen, ob empfangene Pakete Duplikate sind
  Zustand bestimmt, ob die nächste erwartete Sequenznummer 0 oder 1 ist
- **Wichtig**: Der Empfänger weiß NICHT, ob der Sender das letzte ACK/NAK unverfälscht empfangen hat!

→ Lässt sich erst am nächsten empfangenen Datenpaket erkennen.



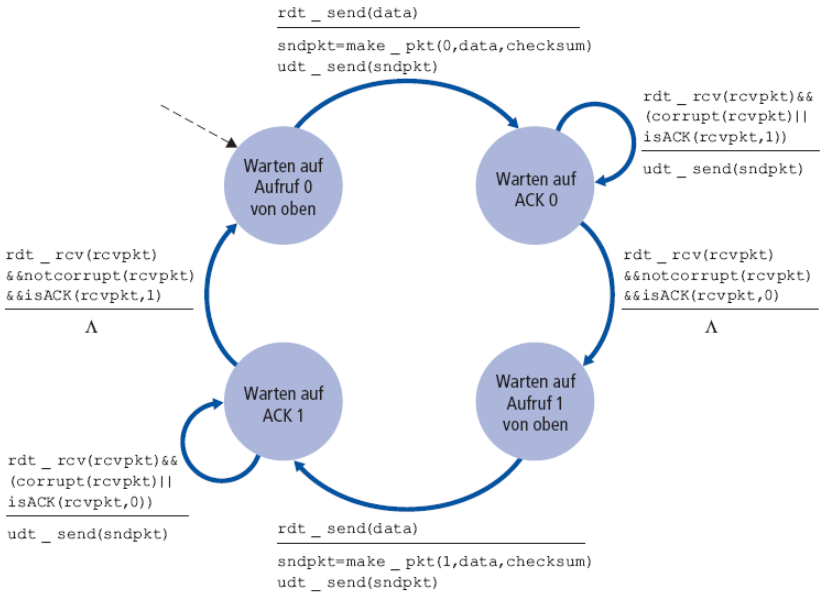
3.4.1 Aufbau eines zuverlässigen Datentransferprotokolls

rdt2.2: Elimination von NAKs

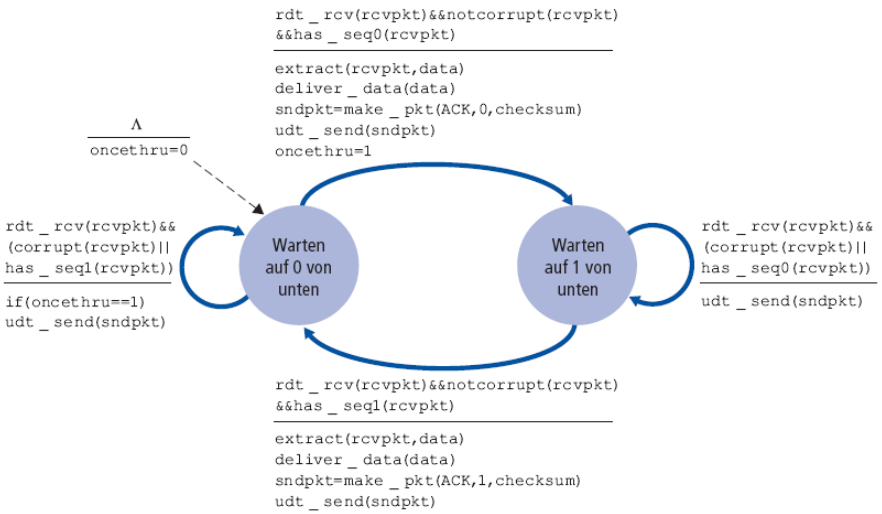
- Anstelle von NAKs: ACK für das letzte korrekt empfangene Paket

→ Empfänger muss die Sequenznummer des bestätigten Paketes im ACK mitschicken

- „Veraltete“ Sequenznummer im ACK wird vom Sender als NAK interpretiert



a rdt2.2-Sender



b rdt2.2-Empfänger

3.4.1 Aufbau eines zuverlässigen Datentransferprotokolls

Zuverlässiger Datentransfer über einen Kanal mit Bitfehlern **und Paketverlusten**: **rdt3.0**

- Der Kanal kann nun auch ganze Pakete verlieren.

  Fehlererkennung, Sequenznummern, ACKs und Übertragungswiederholungen helfen weiter, reichen aber nicht aus

Problem: Wie wird der Verlust von Paketen behandelt?

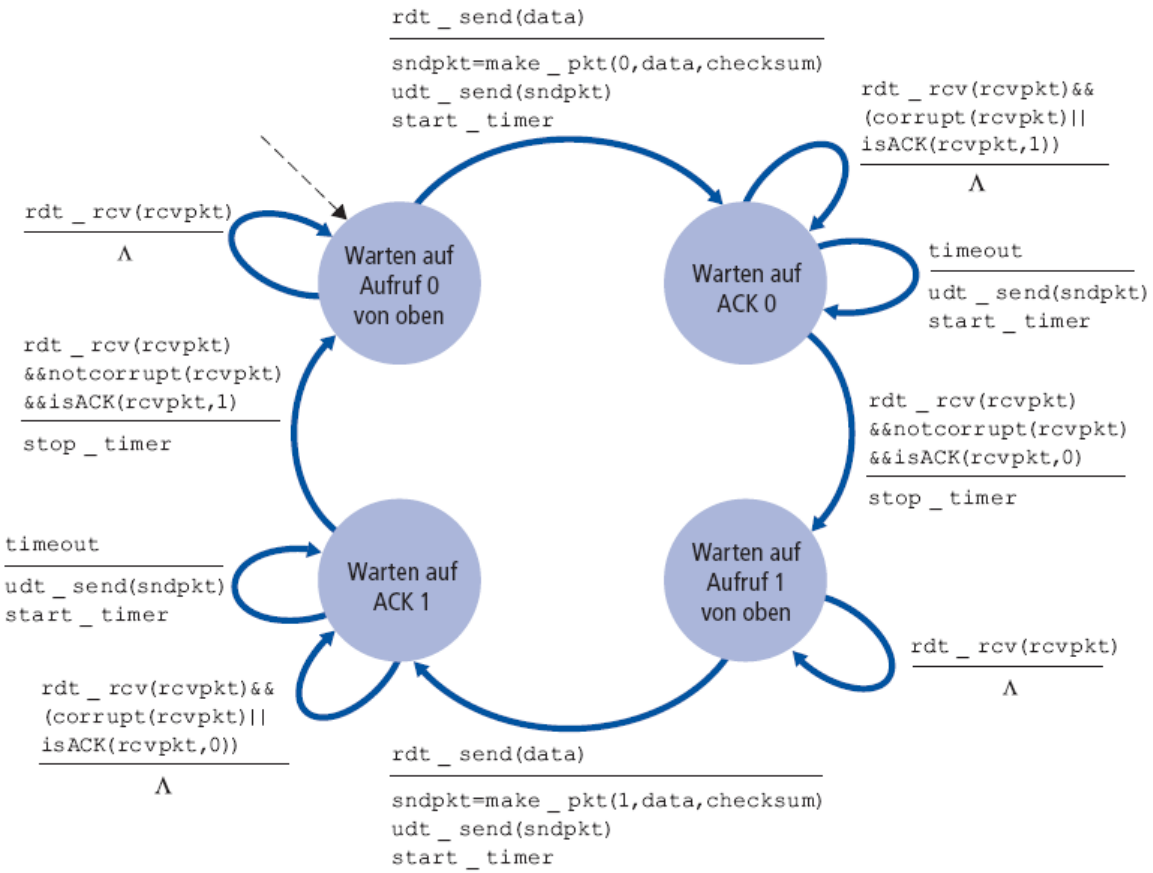
Lösung: Sender wartet eine „vernünftige Zeitspanne“ auf ein ACK

- Wenn dann kein ACK angekommen ist, wird die Übertragung wiederholt
- Wenn das Paket (oder das ACK) nur verzögert wurde und nicht verloren gegangen ist: Paket ist ein Duplikat, dies wird durch seine Sequenznummer erkannt und vom Empfänger verworfen

→ Erfordert **Timer zum Stoppen der Zeit**

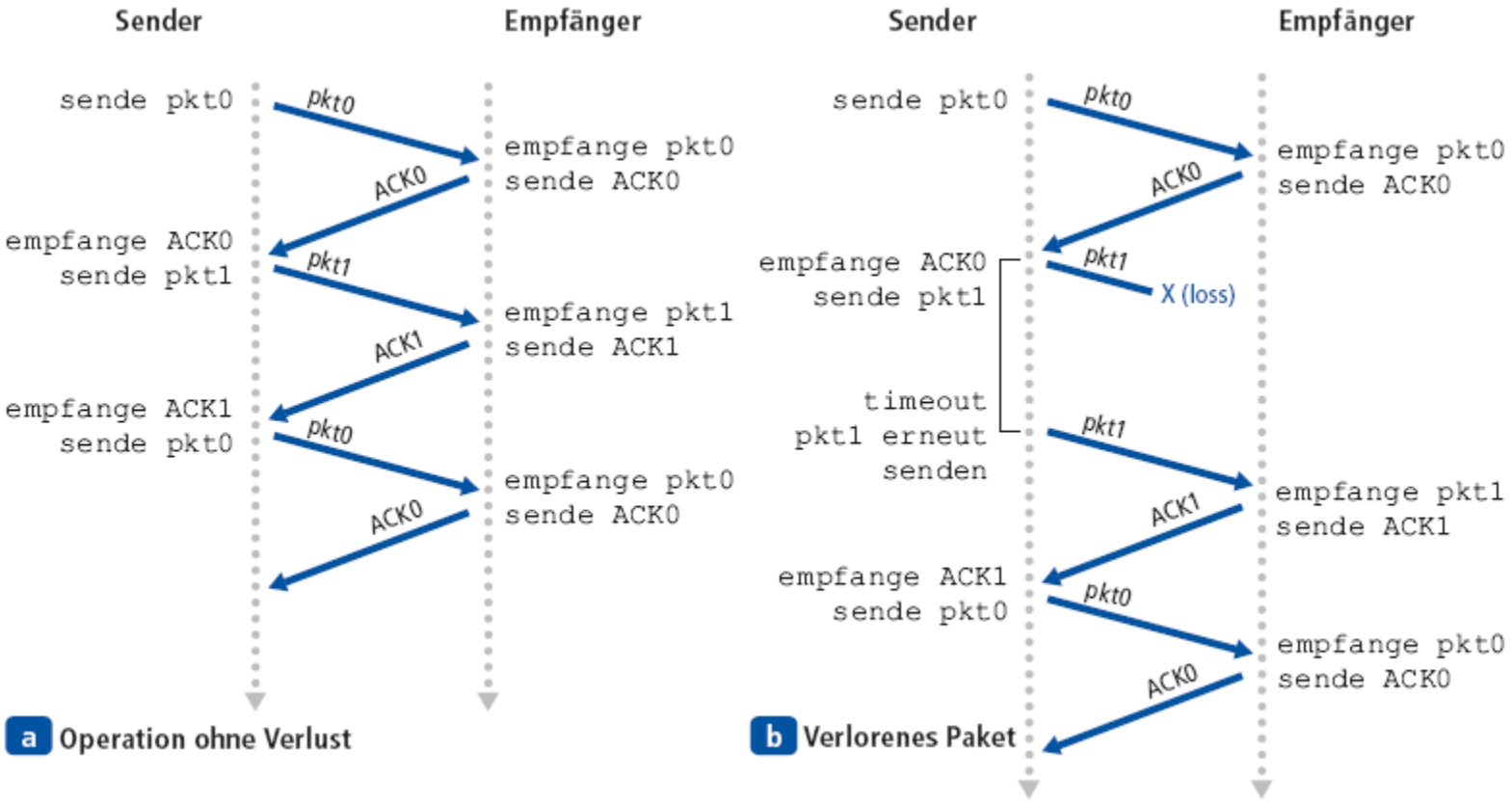
3.4.1 Aufbau eines zuverlässigen Datentransferprotokolls

rdt3.0: Sender mit Wartezeiten



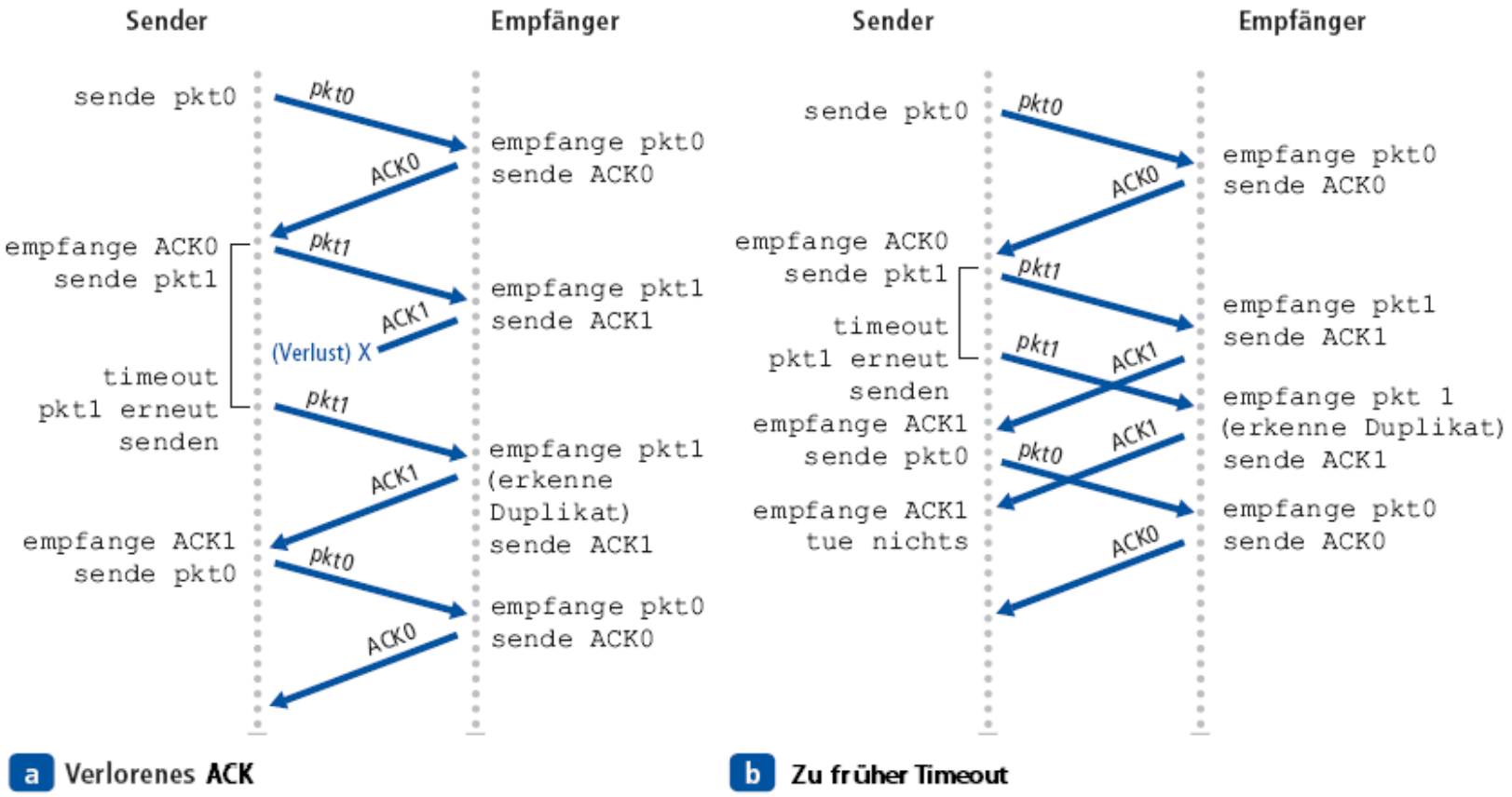
3.4.1 Aufbau eines zuverlässigen Datentransferprotokolls

rdt3.0: Arbeitsweise



3.4.1 Aufbau eines zuverlässigen Datentransferprotokolls

rdt3.0: Arbeitsweise



3.4.1 Aufbau eines zuverlässigen Datentransferprotokolls

rdt3.0: Performance

• rdt3.0 funktioniert, aber die **Performance ist schlecht**!

Beispiel: 1 GBit/s Link, 15 ms Ausbreitungsverzögerung, 8000 Bit Paketgröße

$$T_{transmit} = \frac{L}{R} = \frac{8000\ bit}{10^9\ Bit/s} = 0.008\ ms$$

$$U_{sender} = \frac{L/R}{RTT + L/R} = \frac{.008}{30.008} = 0.00027$$

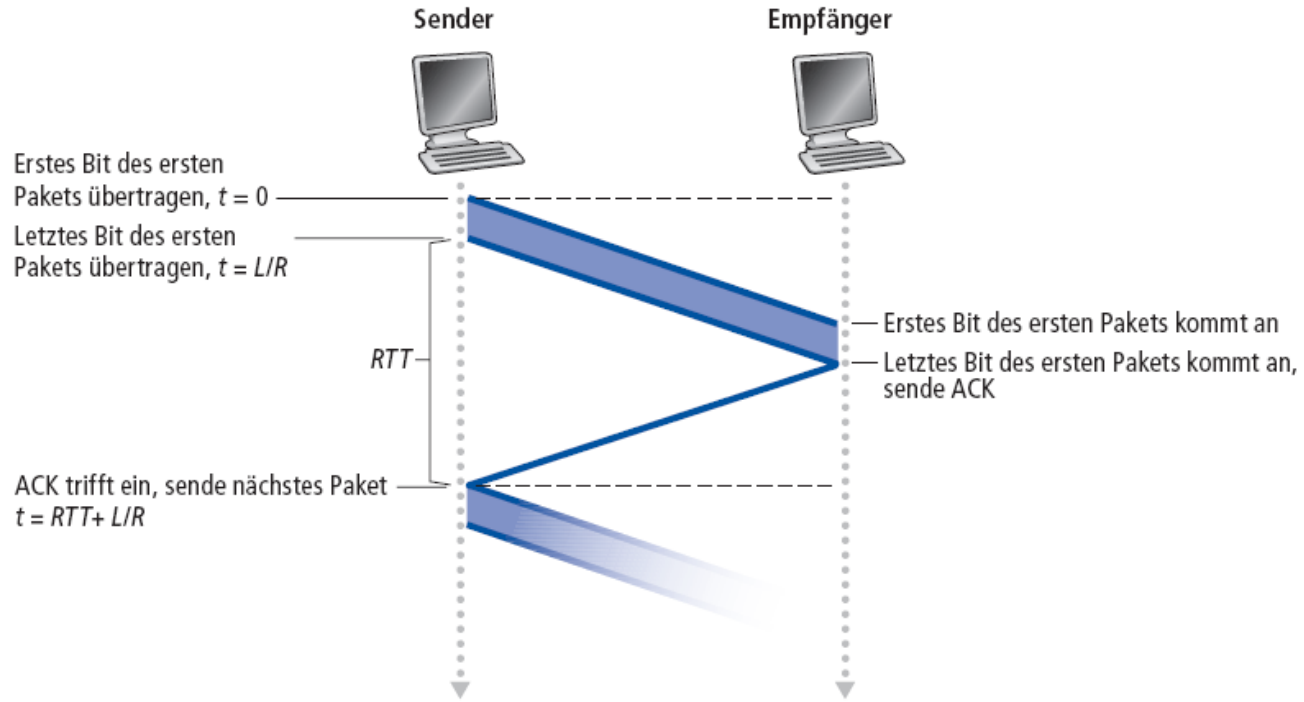
U_{sender}: Utilization (engl. für Auslastung) – Anteil der Zeit, in der tatsächlich gesendet wird

•Einmal 8000 Bit alle ~30 ms -> 33KBit/s Durchsatz über einen Link mit 1 GBit/s

→ Das Protokoll beschränkt die Ausnutzung physikalischer Ressourcen!

3.4.1 Aufbau eines zuverlässigen Datentransferprotokolls

rdt3.0: Stop-and-Wait-Ablauf

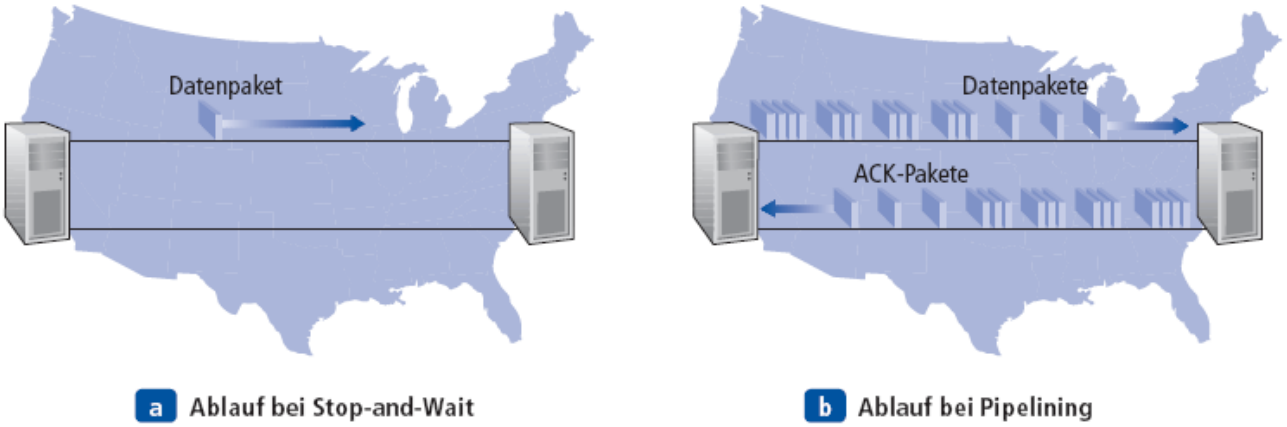


a Ablauf bei Stop-and-Wait

$$U_{sender} = \frac{L / R}{RTT + L / R} = \frac{.008}{30.008} = 0.00027$$

3.4.2 Zuverlässige Datentransferprotokolle mit Pipelining

Der Kern des Leistungsproblems mit rtd3.0 liegt darin, daß es ein Stop-and-Wait-Protokoll ist. Stop-and-Wait im Vergleich zu Pipelining:

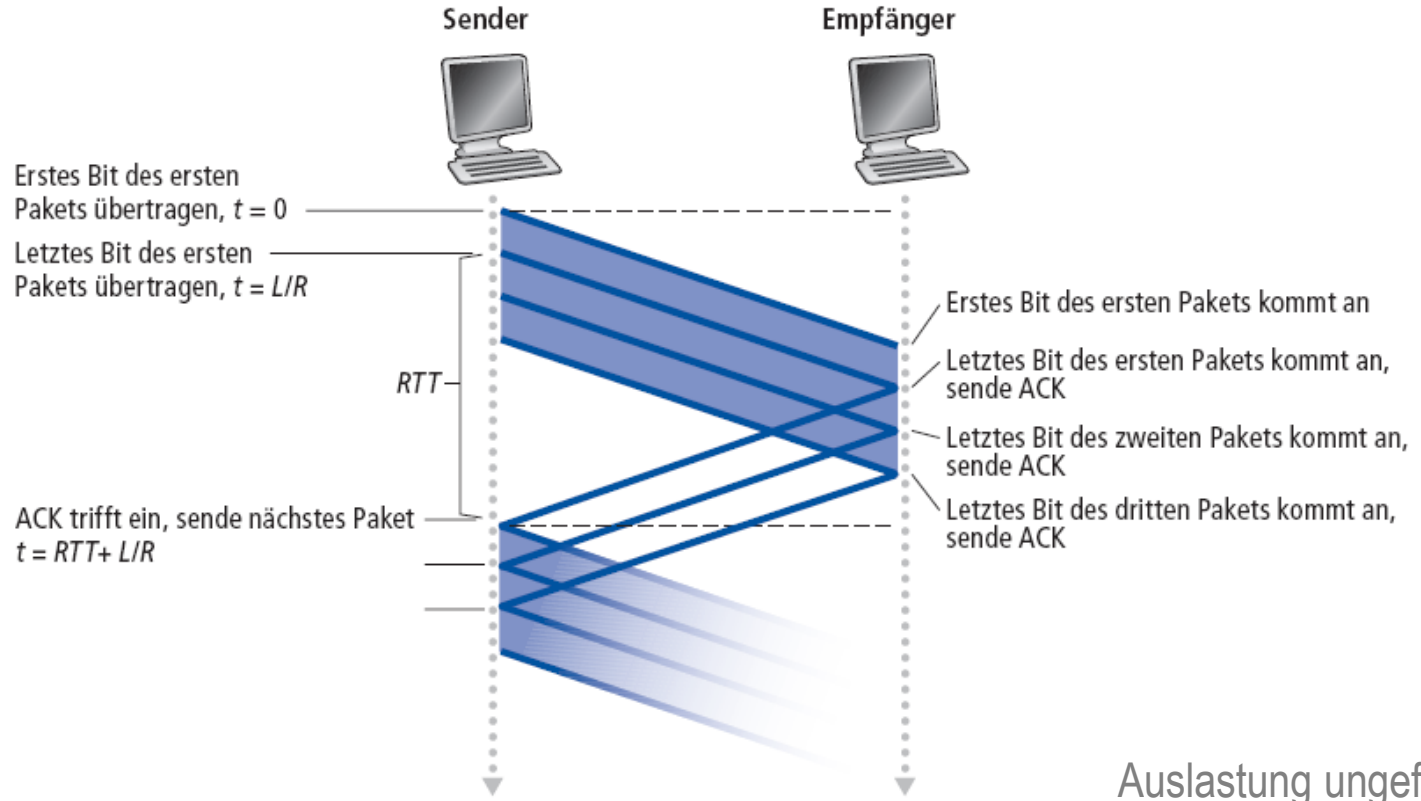


Pipelining: Sender lässt nicht nur eines, sondern mehrere unbestätigte Pakete zu

- ► Die Anzahl der Sequenznummern muss erhöht werden
- ► Pakete müssen beim Sender und/oder Empfänger gepuffert werden

2 prinzipielle Arten von Protokollen mit Pipelining: Go-Back-N und Selective Repeat

3.4.2 Zuverlässige Datentransferprotokolle mit Pipelining



b Ablauf bei Pipelining

Auslastung ungefähr um Faktor 3 erhöht!

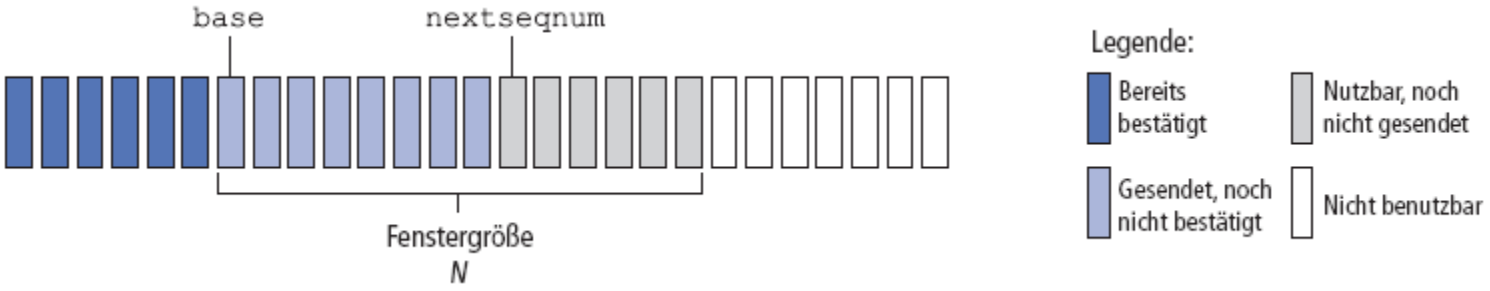
$$U_{sender} = \frac{3 * L / R}{RTT + L / R} = \frac{.024}{30.008} = 0.0008$$

3.4.2 Pipelining Techniken: GBN vs. SR

	Go-Back-N (GBN)	Selective Repeat (SR)
Empfänger sendet ACKs	Kumulativ	Für jedes Paket einzeln
Timer	Nur **einer** für das älteste unbestätigte Paket	Jedes Paket hat einen eigenen Timer
Verhalten Sender bei Timeout	Alle Pakete im Fenster erneut senden	Nur ein einzelnes Paket
Fenster notwendig	Nur bei Sender	Bei Sender und Empfänger
Puffer beim Empfänger	Nicht notwendig	Notwendig
Implementation	Einfach	Komplex
Statusvariablen	Weniger	Mehr
Effizienz	Schlechter	Besser

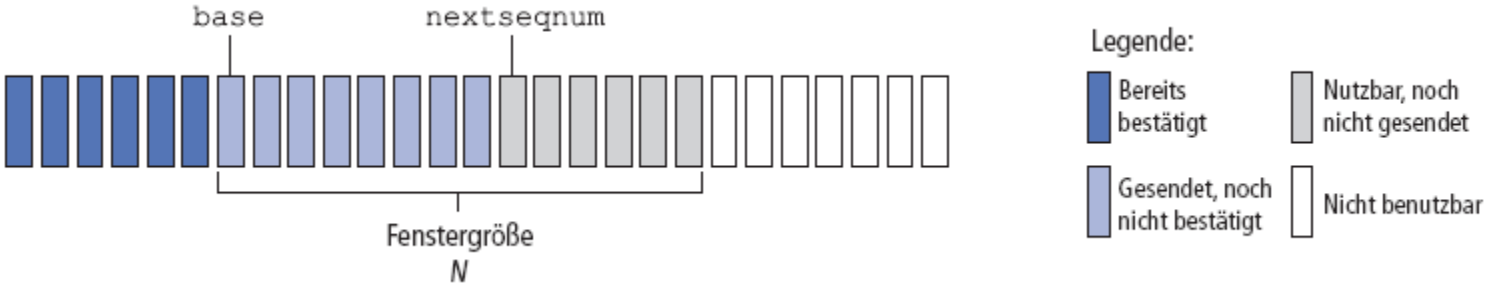
3.4.3 Go-Back-N (GBN)

Sender darf bis zu N Pakete senden ohne auf Bestätigung zu warten.



- **k-Bit-Sequenznummer** im Paketkopf
- Ein **„Fenster“** von bis zu N aufeinanderfolgenden unbestätigten Paketen wird zugelassen
- **ACK(n):** Bestätigt alle Pakete bis zu und einschließlich dem Paket mit Sequenz-nummer n (Doppelte ACKs sind möglich)
- Ein **Timer** für jedes unbestätigte Paket
- **Timeout(n):** alle Pakete mit der Sequenznummer n und höher neu übertragen

3.4.3 Go-Back-N (GBN)



Bereits bestätigt: **[0, `base`-1]** entsprechen Paketen, die schon gesendet und bestätigt worden sind.

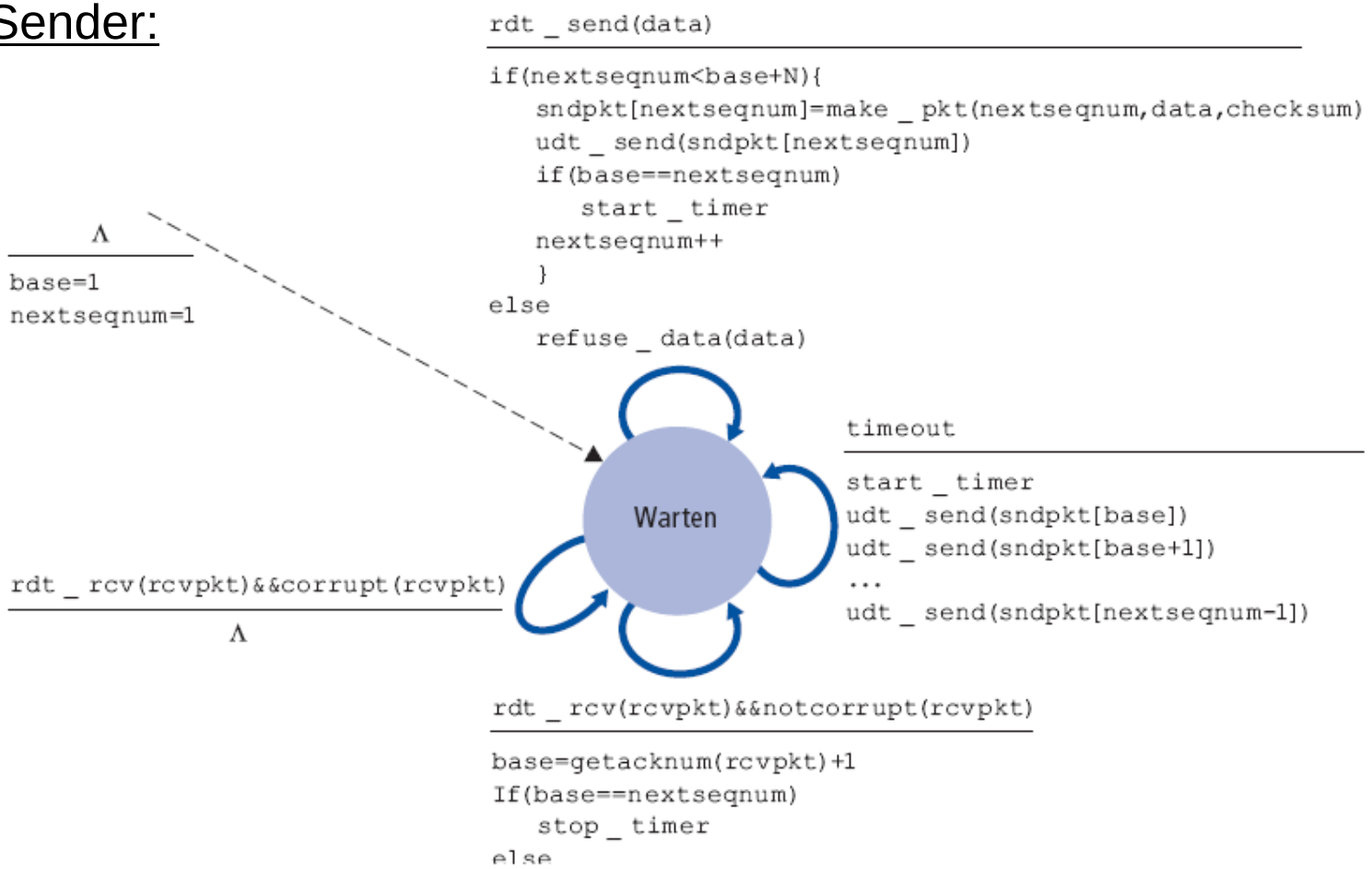
Gesendet, noch nicht bestätigt: **[`base, nextseqnum`-1]** umfasst Pakete, die zwar gesendet wurden, aber noch nicht bestätigt worden sind.

Nutzbar, noch nicht gesendet: **[`nextseqnum, base`+N-1]** enthält Sequenznummern, die für Pakete verwendet werden können, die sofort abgesandt werden dürfen sollten Daten von der darüberliegenden Schicht eintreffen.

Nicht benutzbar: **Sequenznummern >= `base`+N** können nicht benutzt werden bis das Paket mit der Sequenznummer `base` bestätigt wurde.

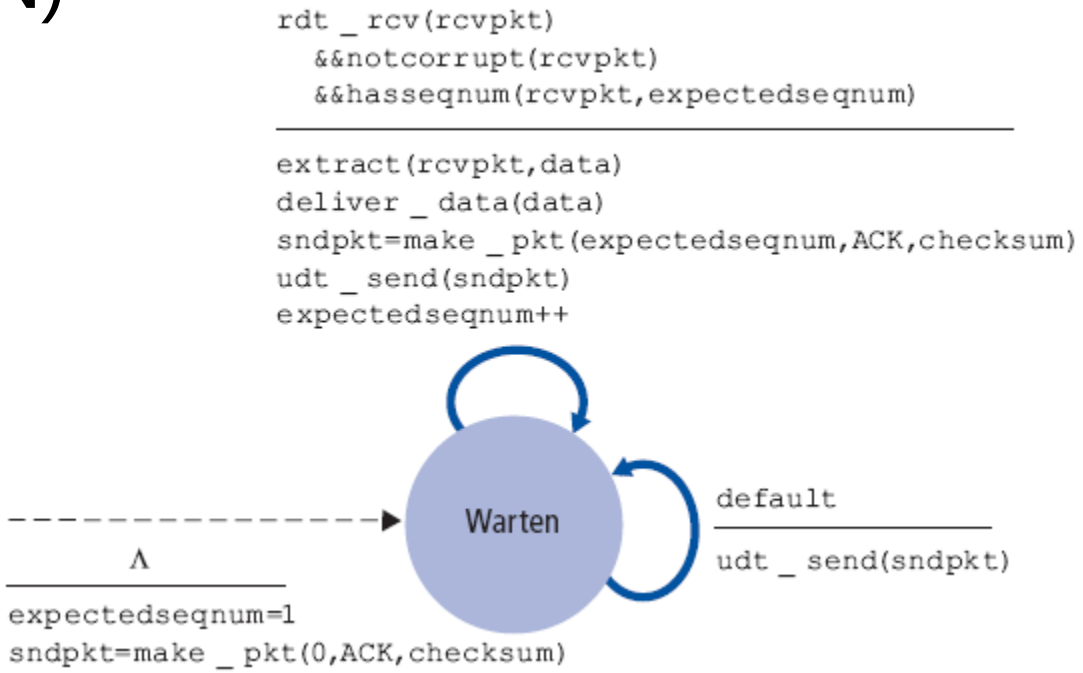
3.4.3 Go-Back-N (GBN)

GBN Sender:



3.4.3 Go-Back-N (GBN)

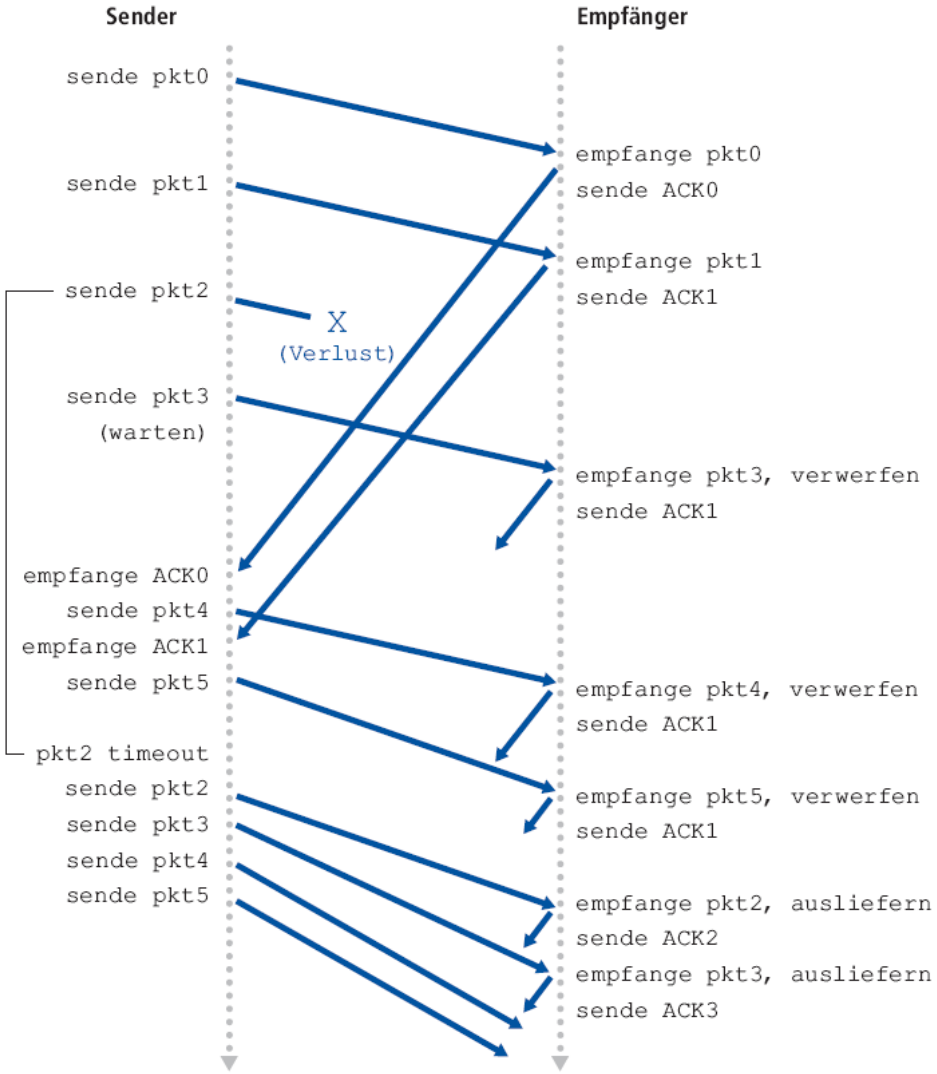
GBN Empfänger:



- Sende ACK für korrekt empfangene Pakete mit der aktuell erwarteten Sequenznummer
  - Kann doppelte ACKs erzeugen
  - Muss sich nur **`expectedseqnum merken`**
- Pakete außer der Reihe:
  - Verwerfen (nicht puffern)!
  - Paket mit der höchsten Sequenznummer in Reihe erneut bestätigen

3.4.3 Go-Back-N (GBN)

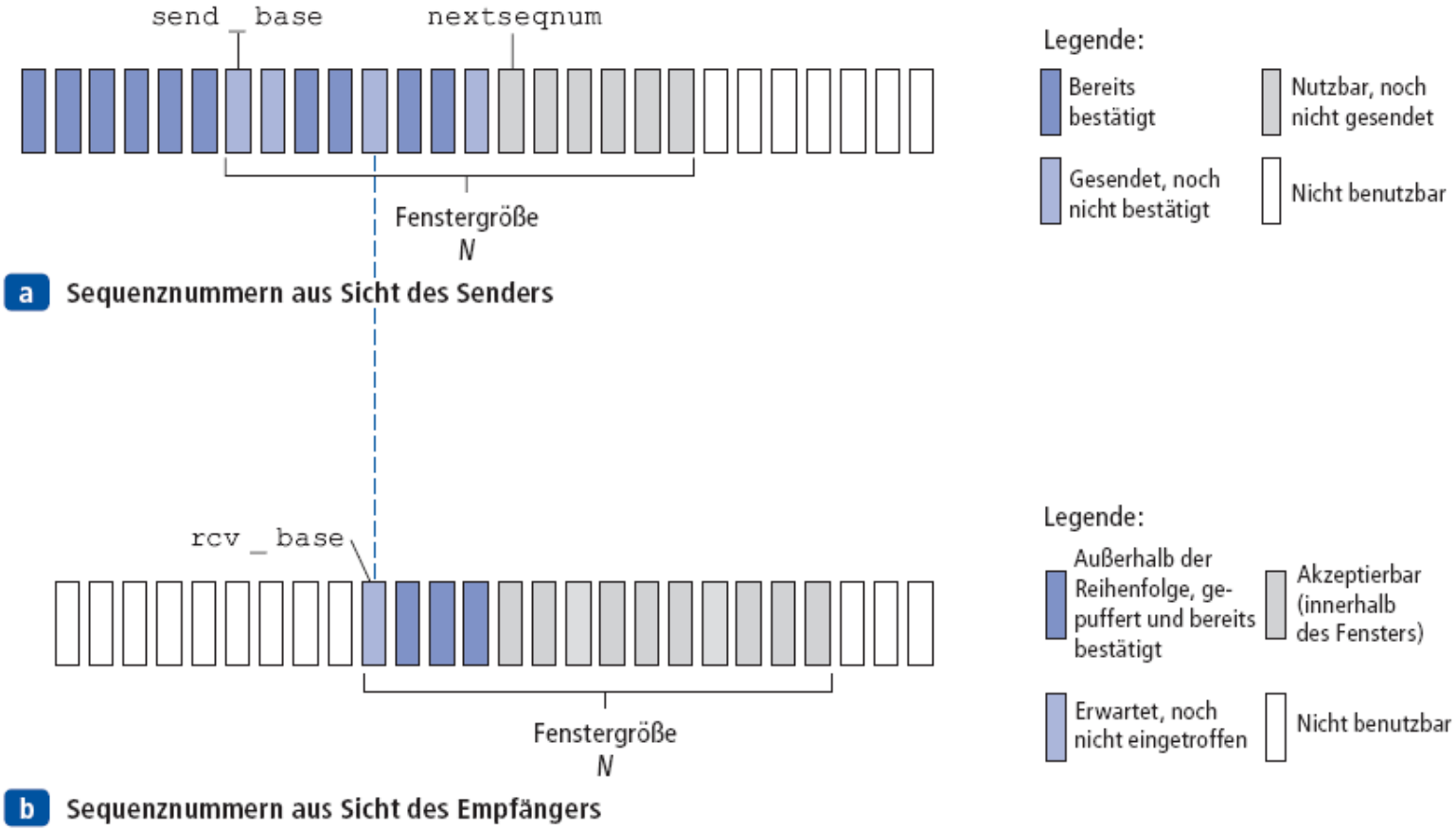
GBN Ablauf:



3.4.4 Selective Repeat (SR)

- Empfänger bestätigt jedes empfangene Paket einzeln
- Empfänger puffert korrekt empfangene Pakete, die außer der Reihe empfangen wurden, in einem Empfängerfenster
  - Ausliefern an die nächste Schicht, wenn dies in der richtigen Reihenfolge möglich ist
- Sender wiederholt eine Übertragung nur für individuelle Pakete
  - Ein Timer für jedes unbestätigte Paket
- Fenster des Senders
  - N aufeinanderfolgende Sequenznummern
  - Beschränkt wieder die unbestätigten, ausstehenden Pakete

3.4.4 Selective Repeat (SR)



3.4.4 Selective Repeat (SR)

Sender:

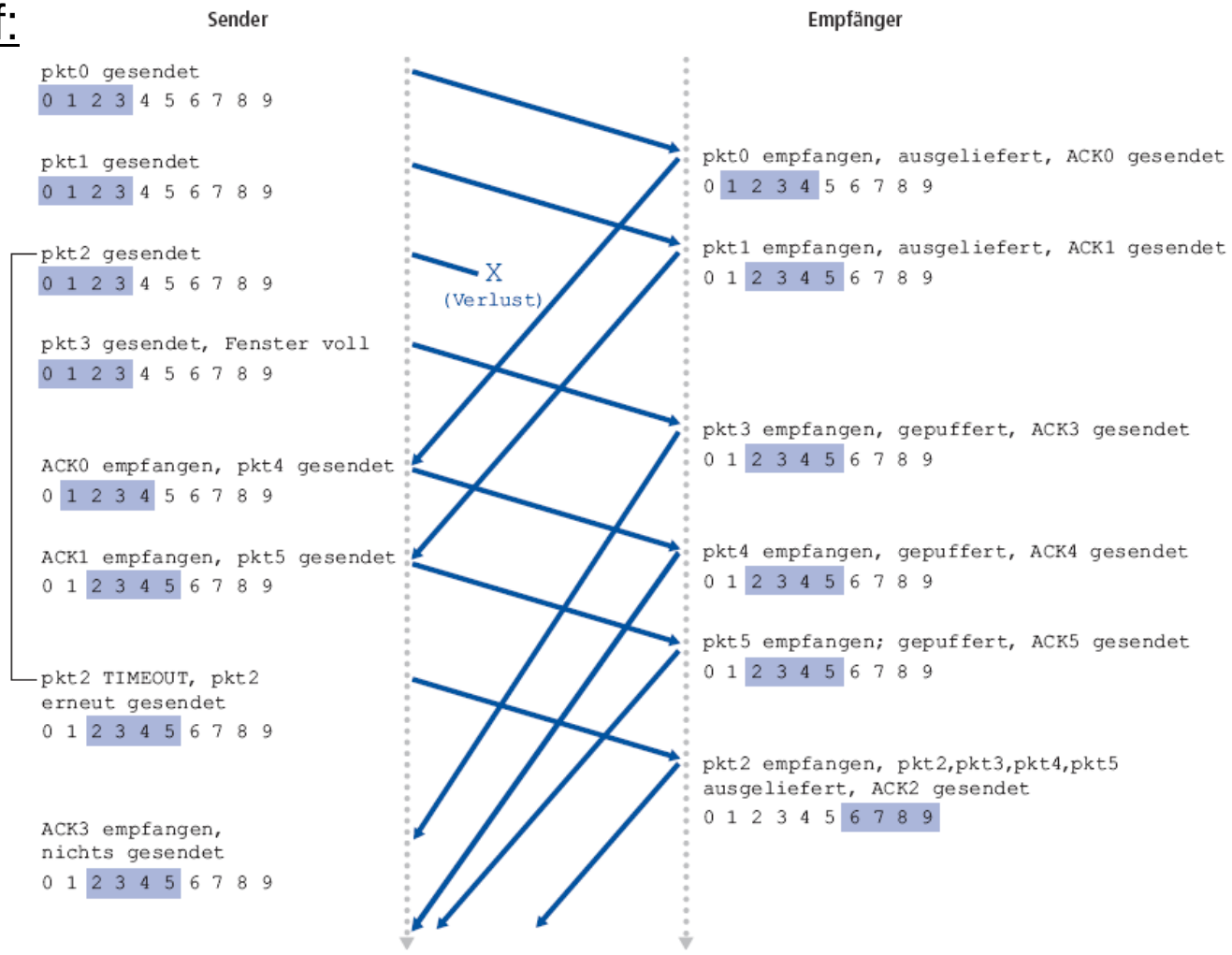
- **Daten von oben**:
  - Wenn nächste Sequenznummer im Fenster liegt: Paket senden, Timer(n) starten!
- **Timeout(n):**
  - Paket n erneut übertragen, Timer(n) starten
- **ACK(n) aus** [sendbase,sendbase+N]
  - Paket n als empfangen markieren
  - Wenn n die kleinste unbestätigte Sequenznummer ist, Fenster zur neuen kleinsten unbestätigten Sequenznummer verschieben

Empfänger:

- **Paket aus** [rcvbase, rcvbase+N-1]
  - Sende ACK(n)
  - Außer der Reihe: Puffern
  - In der Reihe: Ausliefern (auch alle gepufferten Pakete ausliefern, die jetzt in der Reihe sind), Fenster zum nächsten erwarteten Paket verschieben
- **Paket aus** [rcvbase-N,rcvbase-1]
  - ACK(n)
- **Sonst:**
  - Ignoriere das Paket

3.4.4 Selective Repeat (SR)

SR Ablauf:



3.4.4 Selective Repeat (SR)

SR Problemfall:

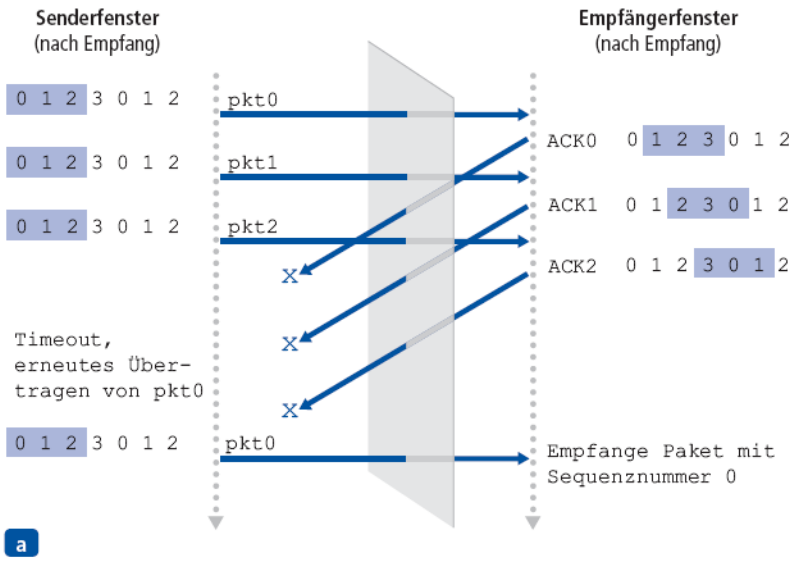
Beispiel:

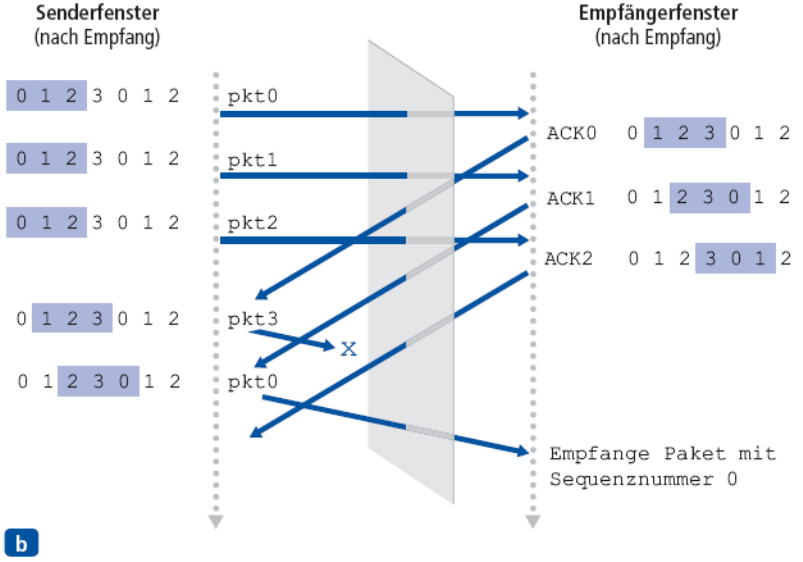
- Sequenznummern: 0-3
- Fenstergröße=3

Fall a: Die ACKs der ersten drei Pakete gehen verloren und der Absender überträgt diese Pakete erneut.
Der **Empfänger erhält** am Ende ein Paket mit Sequenznummer 0
→ **eine Kopie** des ersten übertragenen Paketes.

Fall b: Die ACKs der ersten drei Pakete werden richtig abgeliefert. Der Absender bewegt daher sein Fenster vorwärts und sendet Pakete mit den Sequenznummern 3 und 0. Das Paket mit Sequenznummer 3 geht verloren, aber das Paket mit Sequenznummer 0 kommt an
→ ein Paket, das **neue Daten** enthält.

→ **Empfänger kann beide Fälle nicht unterscheiden**! Gibt in Fall a die Kopie des alten Pakets als neue Daten nach oben!

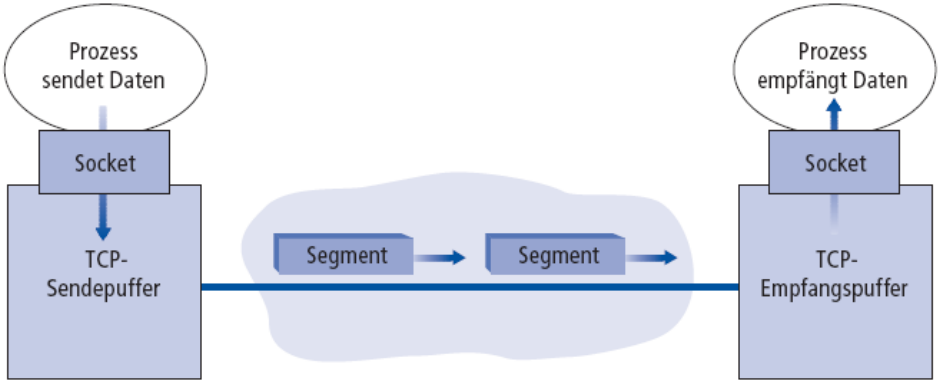




3.5 Verbindungsorientierter Transport: TCP

- Definiert in RFC 793, RFC 1122, RFC 1323, RFC 2018 und RFC 2581.
- **Verbindungsorientiert**, weil vor einer Datenübertragung erst ein “Handshake” zwischen den beiden involvierten Prozesse durchgeführt werden muss um die Parameter des folgenden Datentransfers (in Form mehrerer TCP-Zustandsvariablen) auszuhandeln.
- Läuft ausschließlich auf den Endsystemen und nicht in den dazwischen liegenden Netzwerkelementen (Router und Switches der Sicherungsschicht). Zwischengeschaltete Router sind sich der TCP-Verbindungen gar nicht bewusst.
- Bietet einen **Vollduplexdienst**: Verfahren, das die gleichzeitige Datenübertragung in beide Richtungen ermöglicht.
- Ist immer eine **Punkt-zu-Punkt-Verbindung**, besteht also immer zwischen einem einzelnen Sender und einem einzelnen Empfänger. Multicasting ist mit TCP nicht möglich.

3.5 Verbindungsorientierter Transport: TCP



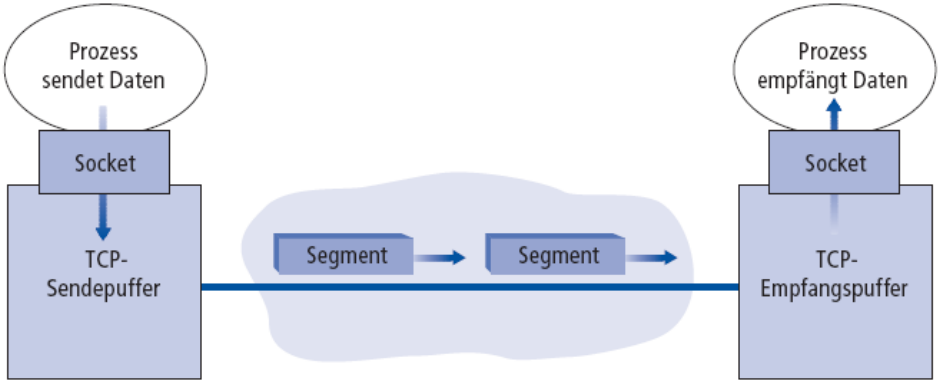
1. "**Drei-Wege-Handshake**" (three-way handshake) zum Verbindungsaufbau:

2. Client-Prozess überträgt einen Datenstrom durch den Socket

3. TCP packt diese Daten in den Sendepuffer der Verbindung

4. Diese Daten werden in TCP-Segmenten zur Netzwerkschicht heruntergereicht
   - **MSS** (maximum segment size = maximale Segmentgröße)
   - **MTU** (maximum transmission unit = maximale Übertragungseinhheit):

3.5 Verbindungsorientierter Transport: TCP



5. Die Netzwerkschicht verkapselt die Segmente in IP-Datagramme und sendet sie ins Netz
6. Sobald TCP auf dem Server ein Segment erhält, werden die Daten in den Eingangspuffer der zugehörigen TCP-Verbindung eingefügt
7. Die Anwendung liest den Datenstrom aus diesem Puffer

→ Eine TCP-Verbindung besteht aus zwei Hälften:
Puffer, Variablen und eine Socket-Verbindung zu einem Prozess in einem Host sowie einem weiteren Satz dieser Elemente im anderen Host.

3.5.2 TCP-Segmentstruktur

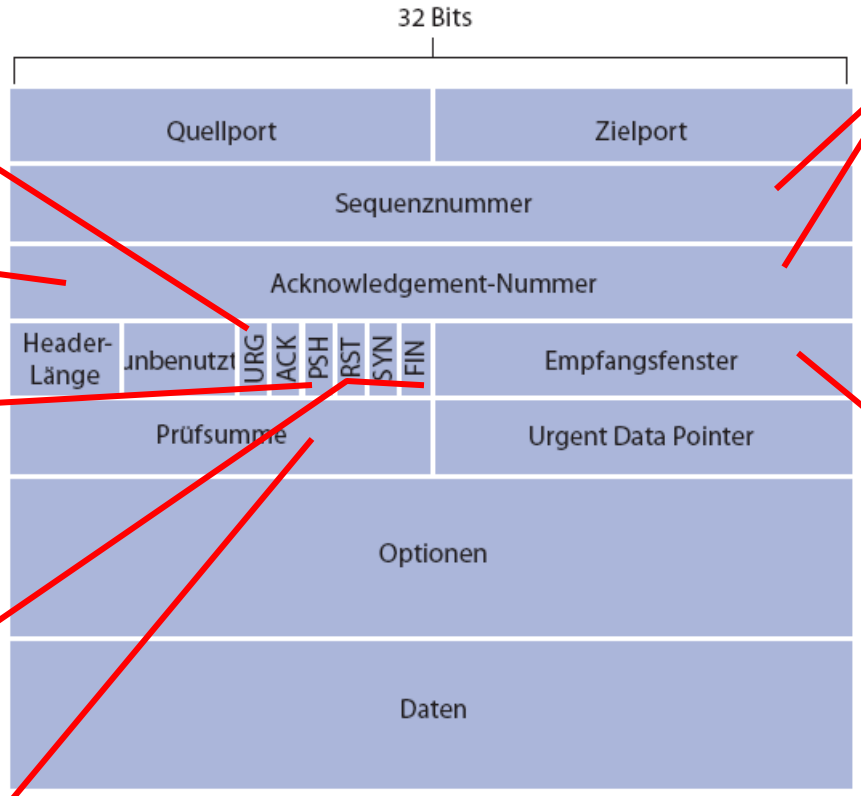
URG: urgent data (üblicherweise nicht benutzt)

ACK: ACK-Nummer gültig

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

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

Prüfsumme (wie bei UDP)



Zählen Bytes und nicht Segmente!

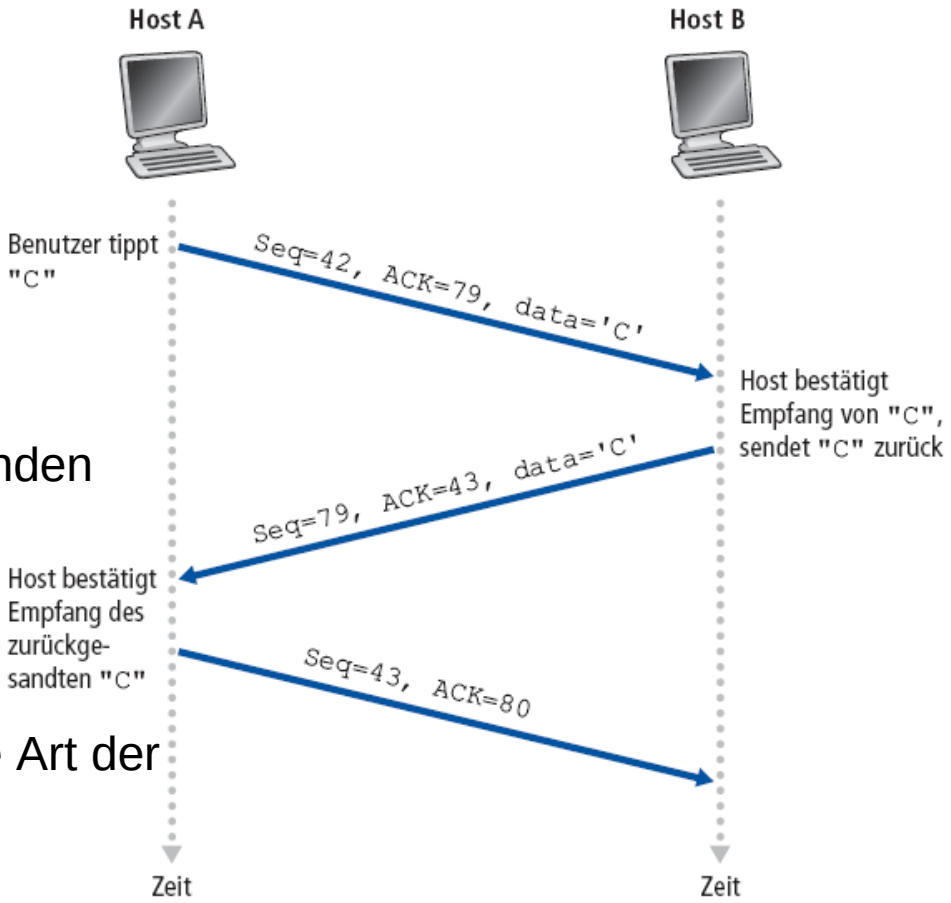
Anzahl an Bytes, die der Emfpänger bereit ist, entgegenzunehmen

3.5.2 TCP-Sequenznummern und -ACKs

Zwei der wichtigsten Felder im TCP-Segment-Header:

Sequenznummern:
Nummer des ersten Byte im Datenteil.
Die Anfangssequenznummern werden von beiden Seiten zufällig gewählt.

Acknowledgementnummern (ACKs):
Sequenznummer des nächsten Byte, das von der Gegenseite erwartet wird.
Da TCP nur Bytes bis zum ersten fehlenden Byte im Strom bestätigt, heißt es, daß TCP **kumulative ACKs** verwendet.
Da jedes ACK als Teil eines Segments transportiert wird, das tatsächliche Daten enthalten **kann**, nennt man diese Art der Bestätigung *piggybacked* (huckepack).



3.5.2 TCP-Sequenznummern und -ACKs

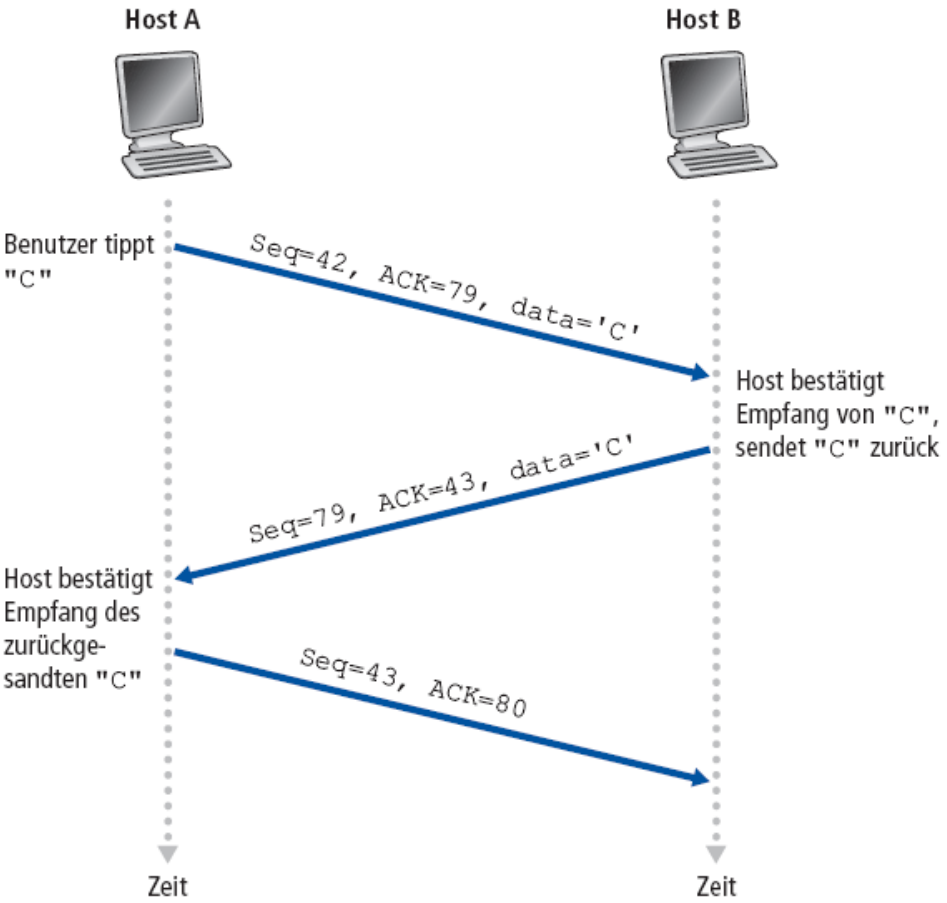
→ Was passiert wenn Segmente außer der Reihe ankommen?

In den RFCs für TCP nicht vorgegeben, wird der Implementation überlassen.

Zwei Möglichkeiten:

1. Der Empfänger verwirft sofort Segmente die nicht in der richtigen Reihenfolge eintreffen. (Kann das Empfängerdesign vereinfachen.)

2. Der Empfänger speichert die erhaltenen Bytes zwischen und wartet auf die fehlenden Bytes, um die Lücke zu schließen. (In der Praxis verwendeter Ansatz, da hinsichtlich der Nutzung der Netzwerkbandbreite effektiver.)



3.5.3 TCP Rundlaufzeit und Timeout

Wie bestimmt TCP den Wert für den Timeout?

- Muss größer als die Rundlaufzeit (Round Trip Time, RTT) sein
  Aber RTT ist nicht konstant
- Wird der Wert zu klein gewählt (zu kurz): → unnötige Timeouts
  Führt zu unnötigen Übertragungswiederholungen
- Wird der Wert zu groß gewählt (zu lang): → langsame Reaktion auf den Verlust von Segmenten

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, daher wird der Durchschnitt über mehrere Messungen verwendet

3.5.3 TCP Rundlaufzeit und Timeout

Der Durchschnitt der **`SampleRTT`** Messungen wird **`EstimatedRTT`** genannt.

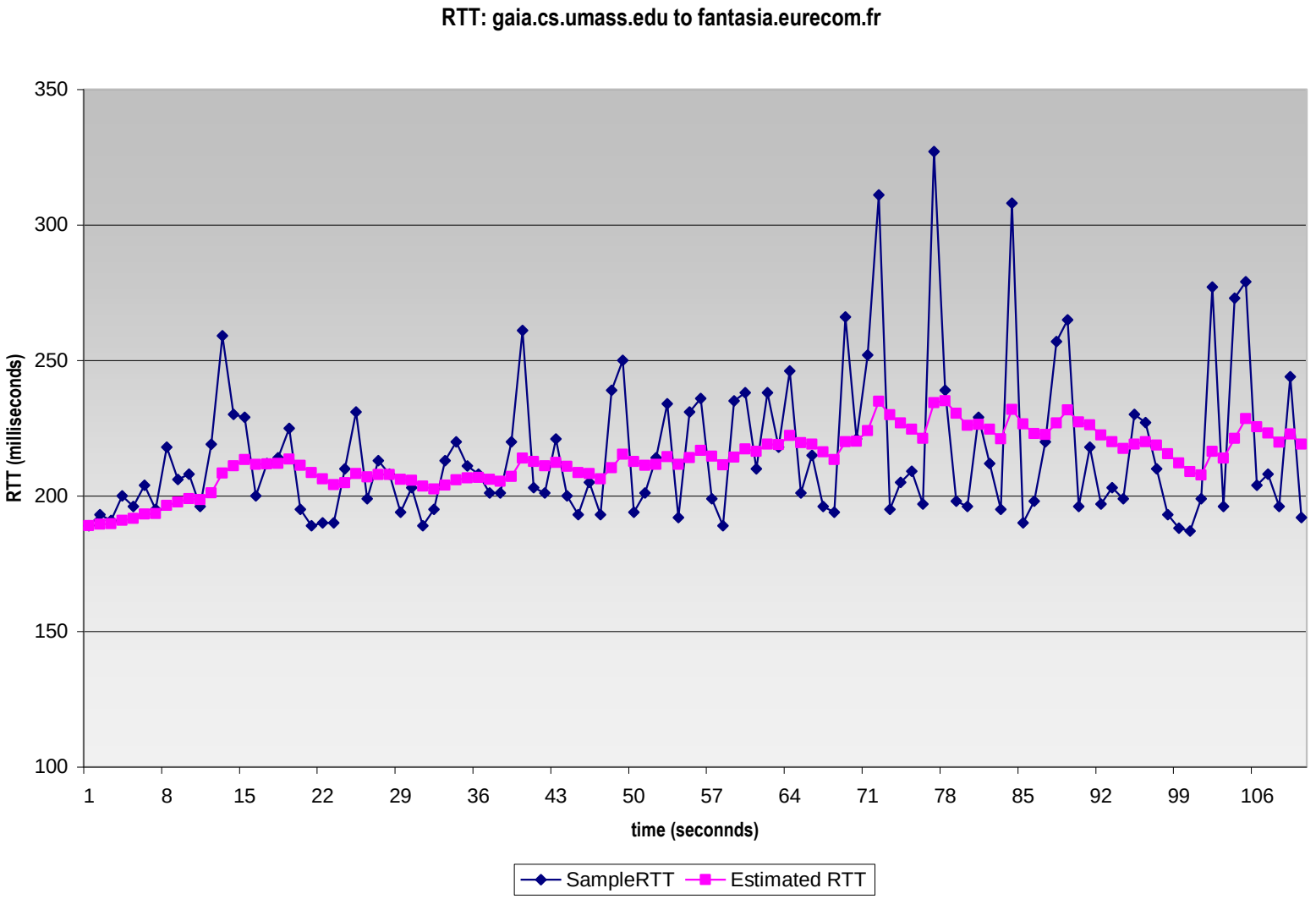
Wenn eine neue **`SampleRTT`** gemessen wird, aktualisiert TCP den Wert **`EstimatedRTT`** entsprechend der folgenden Formel:

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

Der neue Wert von **`EstimatedRTT`** ist eine gewichtete Kombination des vorherigen Wertes von **`EstimatedRTT`** und der neuen **`SampleRTT.`**

- Einfluss vergangener Messungen verringert sich exponentiell schnell
- Üblicher Wert: $\alpha = 0.125$ (RFC 2988)

3.5.3 Beispiel für die Rundlaufzeit Bestimmung



3.5.3 TCP Rundlaufzeit und Timeout

Wie bestimmt TCP den Wert für den Timeout?

- **`EstimatedRTT`** plus "Sicherheitsabstand"
  Größere Schwankungen von **`EstimatedRTT`** → größerer Sicherheitsabstand
- Formel zur Bestimmung des Maßes der Variabilität der **`RTT`**:
  (Abschätzung wie sehr die **`SampleRTT`** typischerweise von der **`EstimatedRTT`** abweicht)

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

(üblicherweise: $\beta = 0.25$)

Daraus wird der Timeout folgendermaßen bestimmt:

`TimeoutInterval = EstimatedRTT + 4*DevRTT`

3.5.4 Zuverlässiger Datentransfer mit TCP

- TCP stellt einen zuverlässigen Datentransfer über den unzuverlässigen Datentransfer von IP zur Verfügung
- Pipelining von Segmenten
- Kumulative ACKs
- TCP verwendet einen einzigen Timer für Übertragungswiederholungen
- Übertragungswiederholungen werden ausgelöst durch:
  - Timeout
  - Doppelte ACKs

Zu Beginn betrachten wir einen vereinfachten TCP-Sender:

- Ignorieren von doppelten ACKs
- Ignorieren von Fluss- und Überlastkontrolle

3.5.4 Zuverlässiger Datentransfer mit TCP

TCP Ereignisse im Sender:

- Daten von Anwendung erhalten:
  - Erzeuge Segment mit geeigneter Sequenznummer
    → 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

3.5.4 TCP-Sender (vereinfacht)

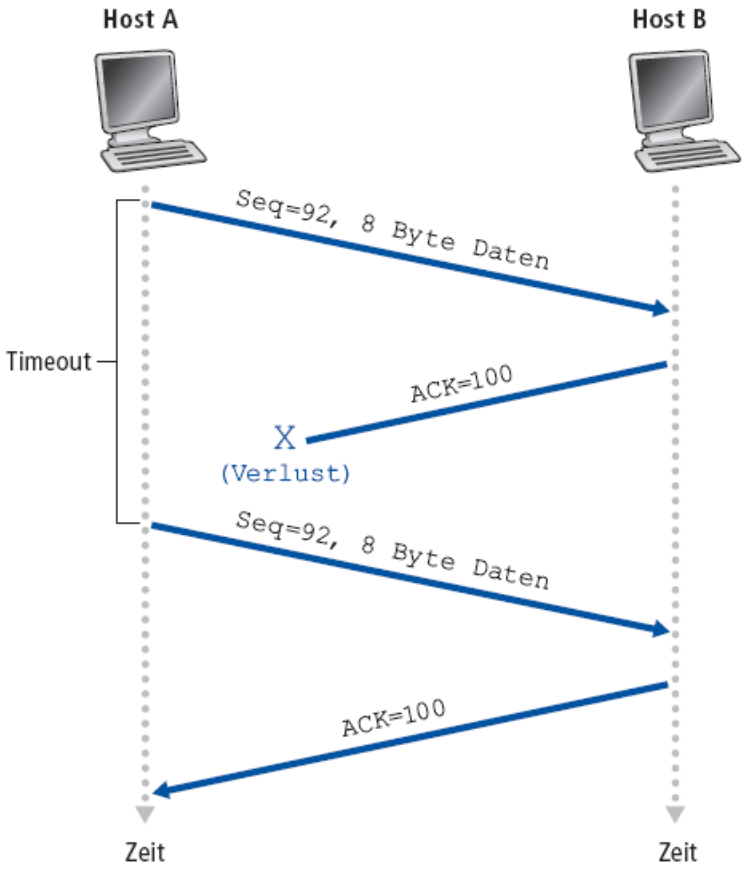
```
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

loop (forever) {
  switch(event)

  event: Daten von der Anwendung
      erzeuge TCP Segment mit Sequenznummer NextSeqNum
      if (Timer läuft nicht)
          starte Timer
      gib Segment an IP weiter
      NextSeqNum = NextSeqNum + length(data)

  event: Timeout
      übertrage das unbestätigte Segment mit der kleinsten Sequenznummer erneut
      starte Timer

  event: ACK empfangen für Sequenznummer y
      if (y > SendBase) {
          SendBase = y
          if (es gibt noch unbestätigte Segmente)
              starte Timer
      }

} /* end of loop forever */
```

Anm.:
• SendBase-1 = das letzte erfolgreich bestätigte Byte

Beispiel:
• SendBase-1 = 71;
• ACK Wert y = 73 kommt an;
Ist y > SendBase, dann bestätigt dieser ACK Wert y ein oder mehrere zuvor unbestätigte Segmente. (*Kumulative ACKs! → hier bestätigt y Segement 72 und 73*)

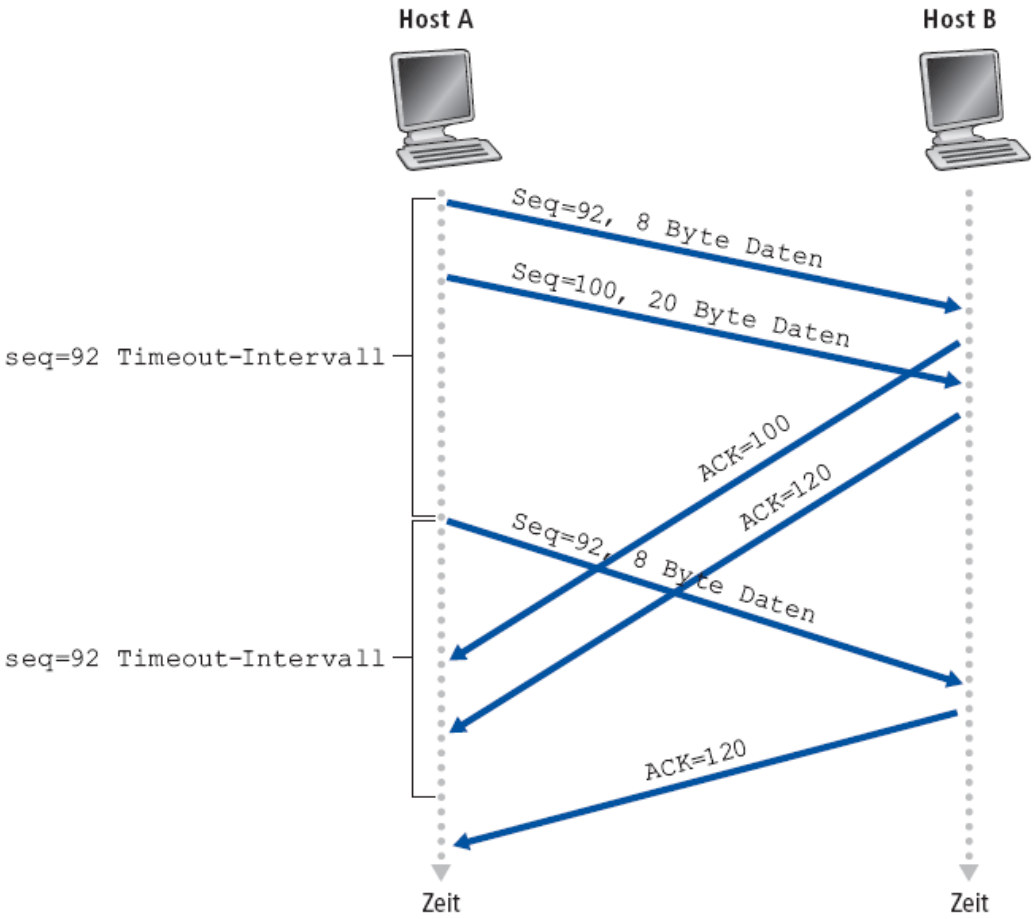
3.5.4 Beispiele für Übertragungswiederholungen: Paketverlust

Erneute Übertragung aufgrund eines verloren gegangenen ACKs:



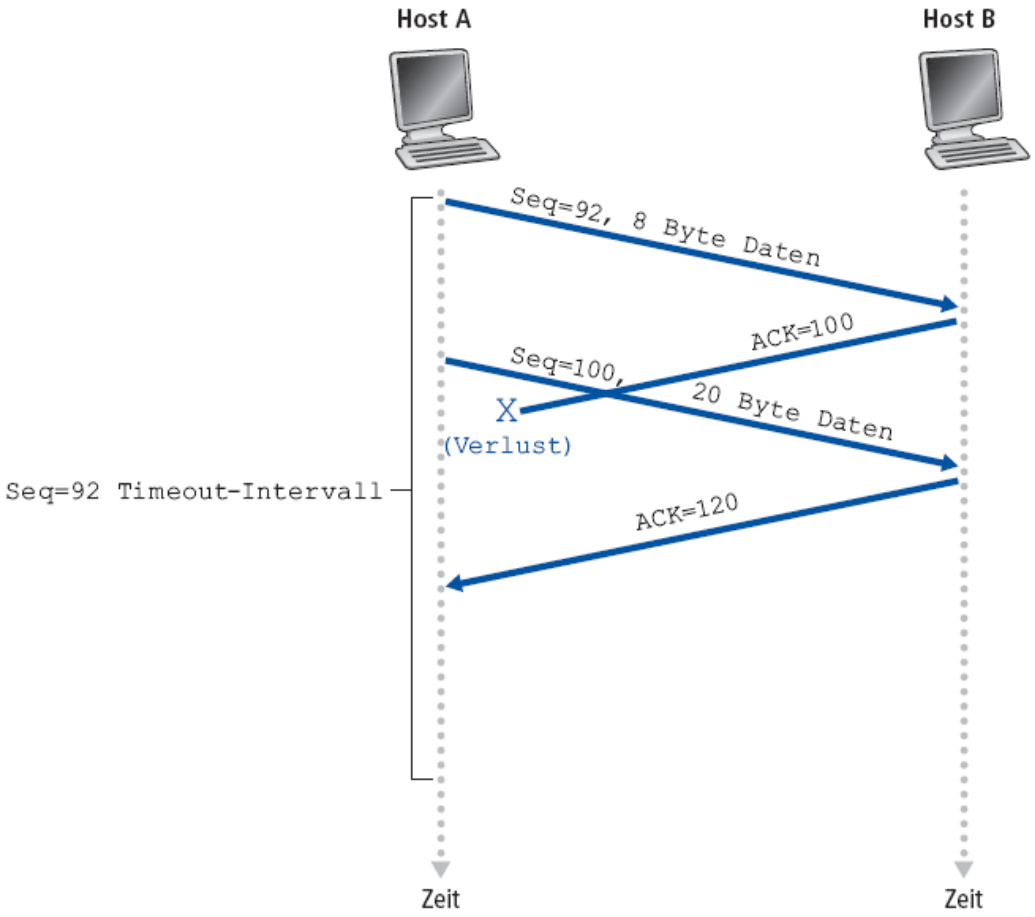
3.5.4 Beispiele für Übertragungswiederholungen: Verfrühte Timeouts

Segment 100 wird nicht erneut übertragen:



3.5.4 Beispiele für Übertragungswiederholungen: Kumulative ACKs

Das kumulative ACK verhindert die erneute Übertragung des ersten Segments:



3.5.4 TCP ACK-Erzeugung

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.

(Definiert in RFC 1122 und RFC 2581)

3.5.4 Fast Retransmit

- Zeit für Timeout ist häufig sehr lang:
  → Große Verzögerung vor einer Neuübertragung
- Erkennen von Paketverlusten durch doppelte ACKs:
  – Sender schickt häufig viele Segmente direkt hintereinander
  → Wenn ein Segment verloren geht, führt dies zu vielen doppelten ACKs

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

3.5.4 Fast Retransmit Algorithmus

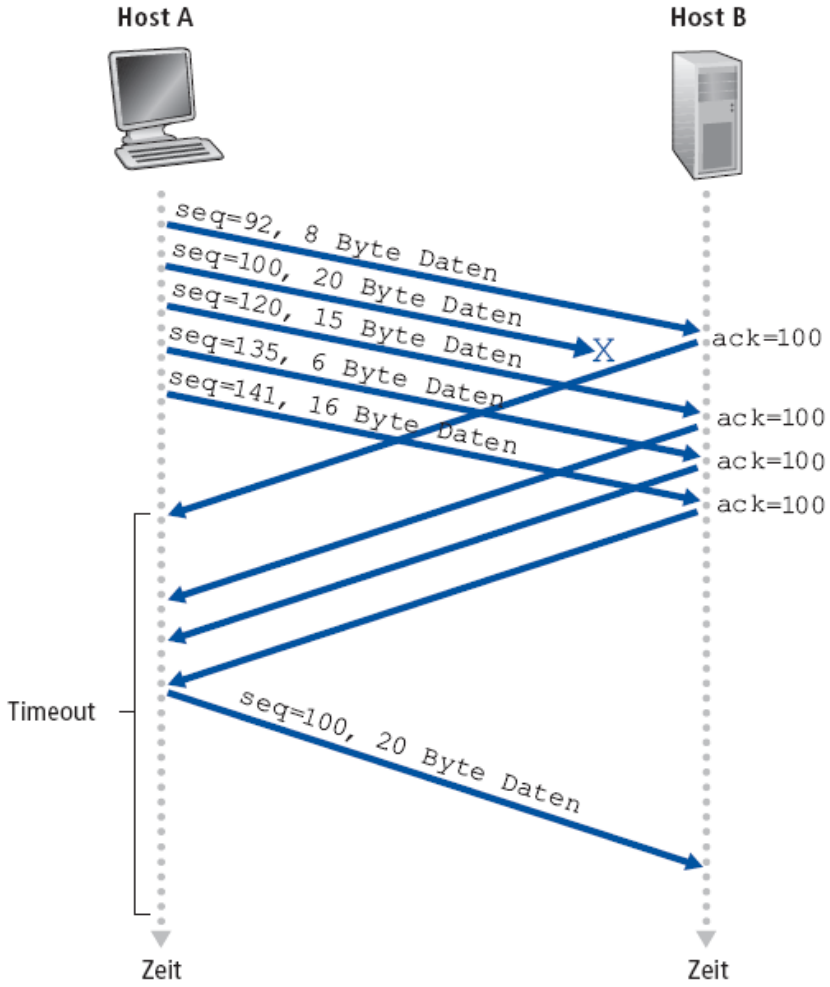
```
event: ACK empfangen, Acknowledgement-Nummer ist y
        if (y > SendBase) {
            SendBase = y
            if (wenn es noch unbestätigte Segmente gibt)
                starte Timer
        }
        else {
            erhöhe den Zähler für doppelte ACKs für y um eins
            if (Zähler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

Ein doppeltes ACK für ein bereits bestätigtes Segment

Fast Retransmit

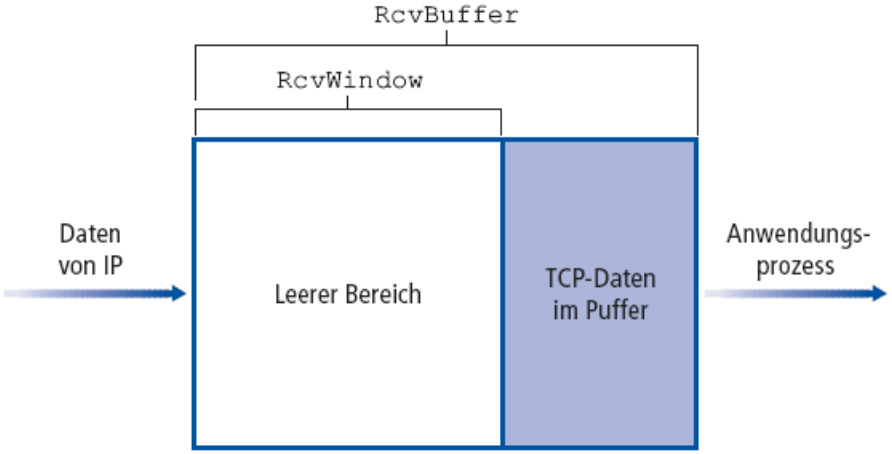
3.5.4 Fast Retransmit

Erneute Übertragung des fehlenden Segmentes, bevor der Timer des Segmentes abläuft:



3.5.4 TCP Flusskontrolle

Empfängerseite von TCP hat einen Empfängerpuffer:



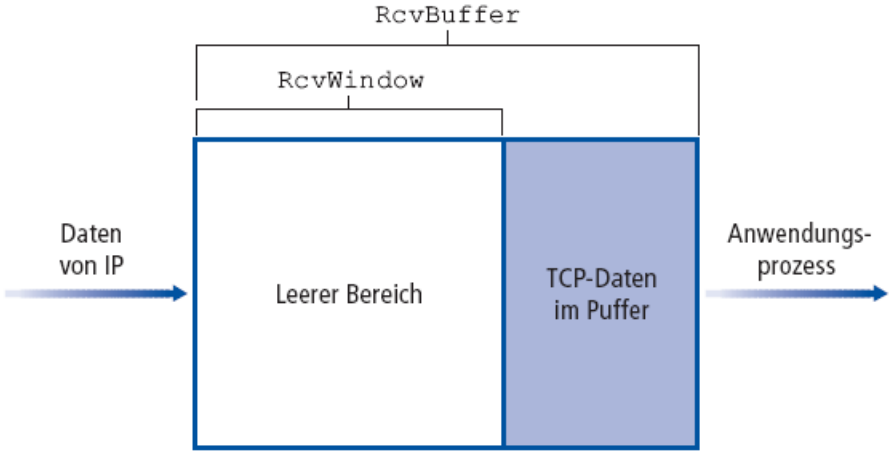
→ Die Anwendung kommt unter Umständen nicht mit dem Lesen hinterher.

TCP Flusskontrolle bedeutet der Sender schickt nicht mehr Daten, als der Empfänger in seinem Puffer speichern kann.

Die Flusskontrolle ist ein Dienst zum Angleich von Geschwindigkeiten: Senderate wird an die Verarbeitungsrate der Anwendung auf der Empfängerseite angepasst.

3.5.4 TCP Flusskontrolle: Funktionsweise

Annahme: Empfänger verwirft Segmente, die außer der Reihe ankommen.



- Platz im Puffer ausgedrückt durch Variable **`RcvWindow`**
  `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

3.5.4 TCP Verbindungsmanagement

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

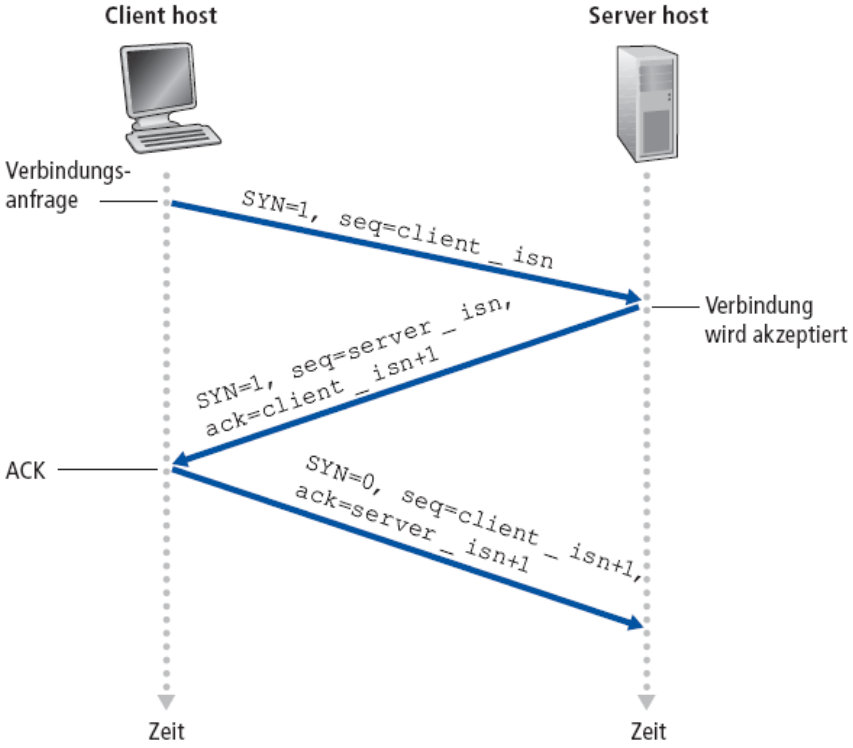
- Initialisieren der TCP-Variablen:
  Sequenznummern, Informationen für Flusskontrolle (z.B. **`RcvWindow`**)
- *Client:* (Initiator)
  ```
  Socket clientSocket = new   Socket("hostname","port number");
  ```
- *Server:* (vom Client kontaktiert)
  ```
  Socket connectionSocket = welcomeSocket.accept();
  ```

3.5.4 TCP Verbindungsmanagement

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.

3.5.4 TCP Verbindungsmanagement



<u>Drei-Wege-Handshake:</u>

<u>Schritt 1:</u> Client sendet TCP-SYN-Segment an den Server

- Initiale Sequenznnummer (Client->Server)
- keine Daten

<u>Schritt 2:</u> Server empfängt SYN und antwortet mit SYNACK

- Server legt Puffer an
- Initale Sequenznummer (Server->Client)

<u>Schritt 3:</u> Client empfängt SYNACK und antwortet mit einem ACK – dieses Segment darf bereits Daten beinhalten.

3.5.4 TCP Verbindungsmanagement

Schließen einer Verbindung:

Client schließt socket: **`clientSocket.close();`**

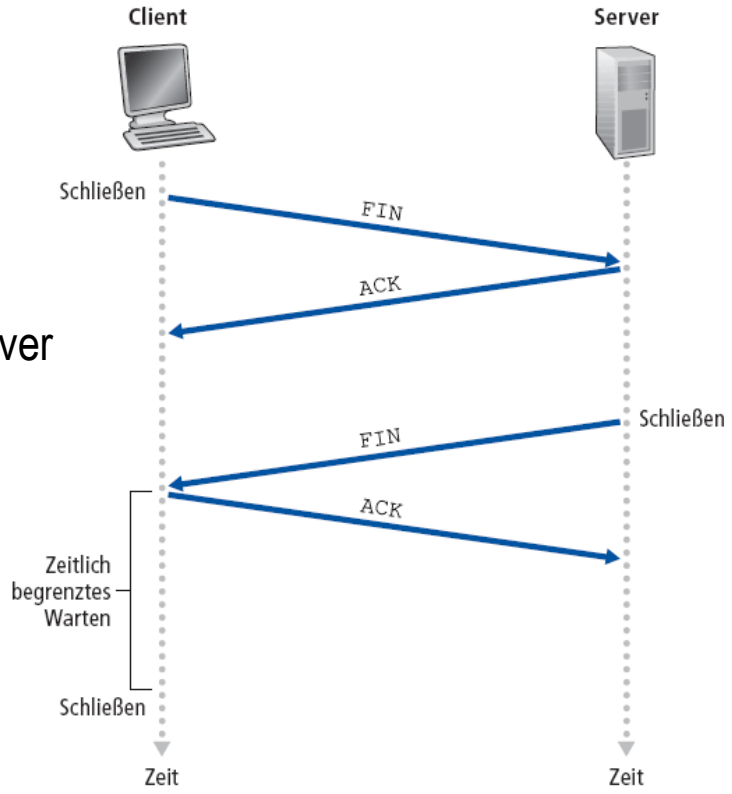
Schritt 1: Client sendet ein TCP-FIN-Segment an den Server

Schritt 2: Server empfängt FIN, antwortet mit ACK; dann sendet er ein FIN (kann im gleichen Segment erfolgen)

Schritt 3: Client empfängt FIN und antwortet mit ACK

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

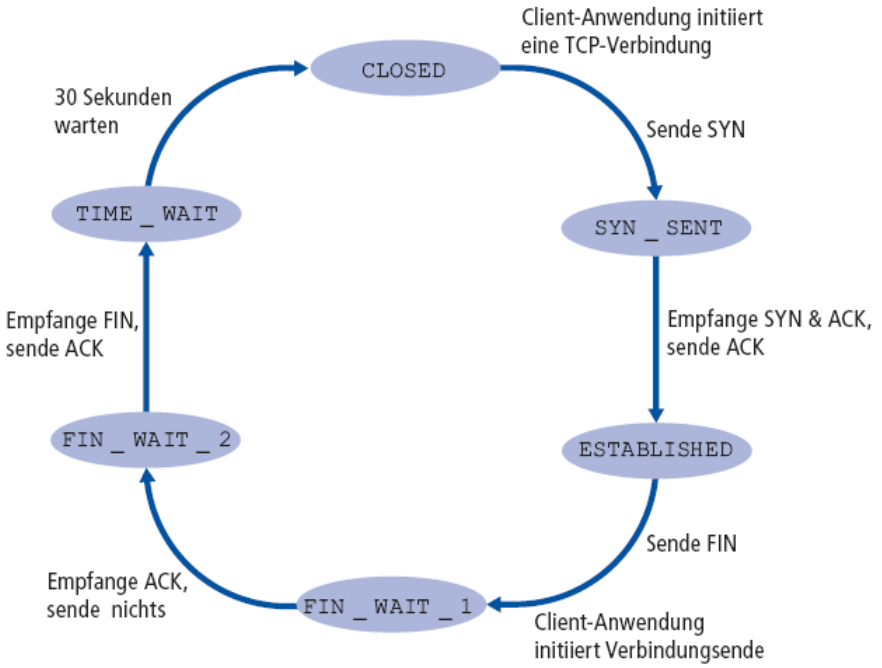
Schritt 4: Server empfängt ACK und schließt Verbindung



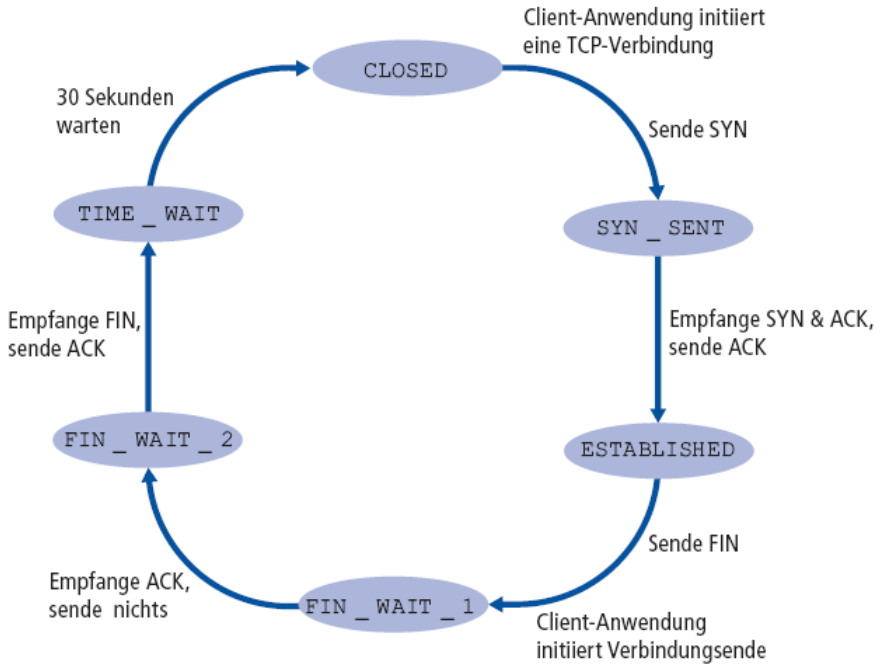
Anm.: Mit kleinen Änderungen können so auch gleichzeitig abgeschickte FINs behandelt werden.

3.5.4 TCP Verbindungsmanagement

TCP-Client-Lebenszyklus:



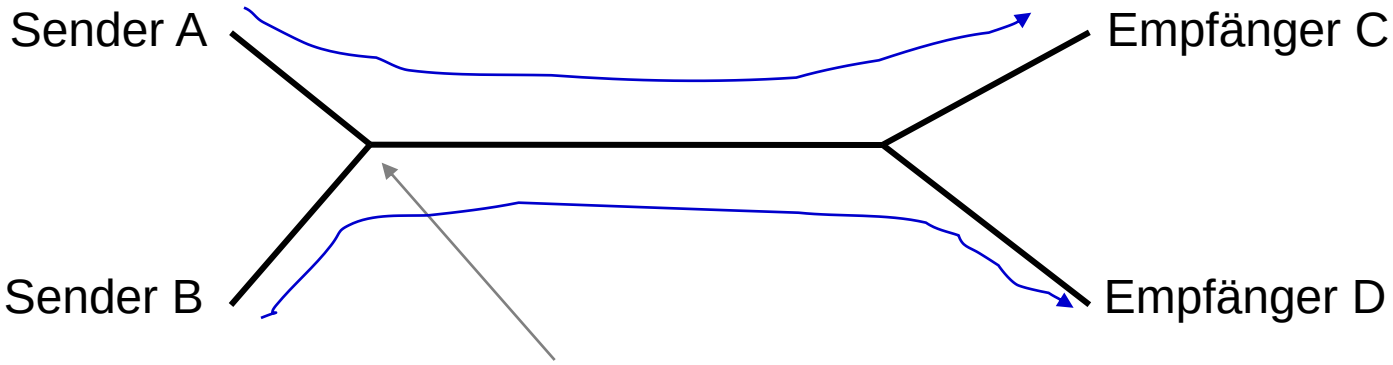
TCP-Server-Lebenszyklus:



3.6 Grundlagen der Überlastkontrolle

Problem:

Wenn alle Sender im Netz immer so viele Pakete losschicken, wie bei den Empfängern in den Puffer passen, dann kann es zu Überlast (congestion) im Netz kommen, da nicht auf die momentane Situation im Netz Rücksicht genommen wird.



wenn alle Verbindungen die gleiche Bandbreite haben und sowohl A als auch B mit der Bandbreite der Verbindung senden, dann kommt es hier zu Netzwerküberlast (Congestion)!

3.6 Grundlagen der Überlastkontrolle

Überlast:

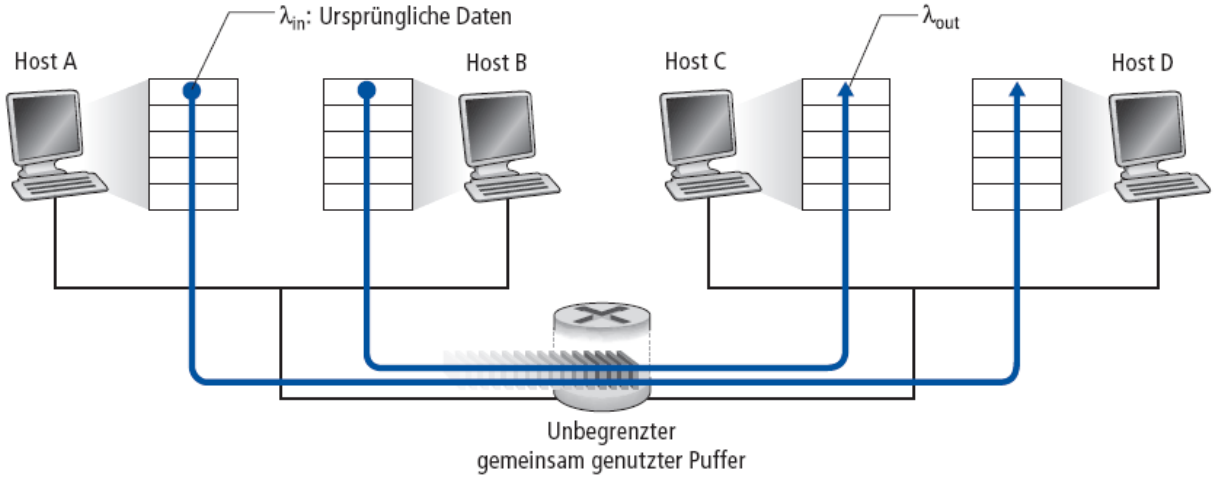
- Informell: „Zu viele Systeme senden zu viele Daten, das Netzwerk kann nicht mehr alles transportieren“
- Nicht zu verwechseln mit Flusskontrolle!
- Erkennungsmerkmale:
  - Paketverluste (Pufferüberlauf in den Routern)
  - Lange Verzögerungen (Warten in den Routern)

→ Eines der zentralen Probleme in Computernetzwerken!

3.6.1 Ursachen und Kosten von Überlast

Betrachtung von 3 zunehmend komplexer werdenden Szenarien um die Ursache und die Auswirkungen von Überlast zu verdeutlichen:

Szenario 1: zwei Quellen, ein Router mit unendlichen Puffern



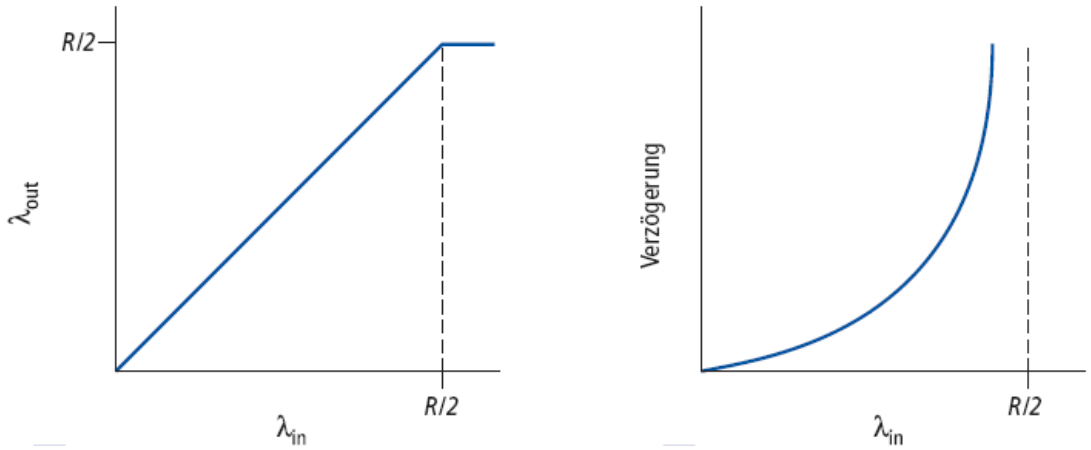
- Anwendungen auf Hosts A und B senden Daten mit der durchschnittlichen **Rate λ_{in} Byte/Sek**
- Keine Fehlerkorrektur (z.B. Übertragungswiederholungen), Flusskontrolle oder Überlastkontrolle
- Pakete der Hosts A und B gehen durch einen gemeinsamen Router und verlassen diesen über dieselbe Verbindung der **Kapazität R**

3.6.1 Ursachen und Kosten von Überlast

Szenario 1: zwei Quellen, ein Router mit unendlichen Puffern

- Linker Graph: Durchsatz pro Verbindung (die Anzahl der Bytes pro Sekunde beim Empfänger) als Funktion der Senderate
  - Für Senderaten zwischen 0 und R/2 ist der Durchsatz beim Empfänger gleich der Senderate der Quelle = alles, was vom Sender verschickt wird, wird vom Empfänger mit begrenzter Verzögerung empfangen.
  - Bei Senderaten > R/2: der Durchsatz bleibt konstant bei R/2

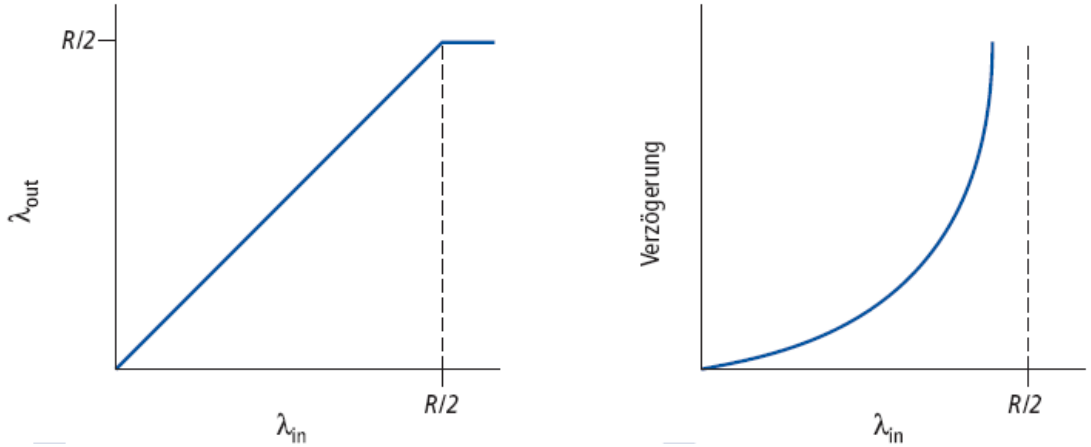
  → *Diese Obergrenze für den Durchsatz ist eine Folge des Teilens der Verbindungskapazität zwischen beiden Verbindungen!*



3.6.1 Ursachen und Kosten von Überlast

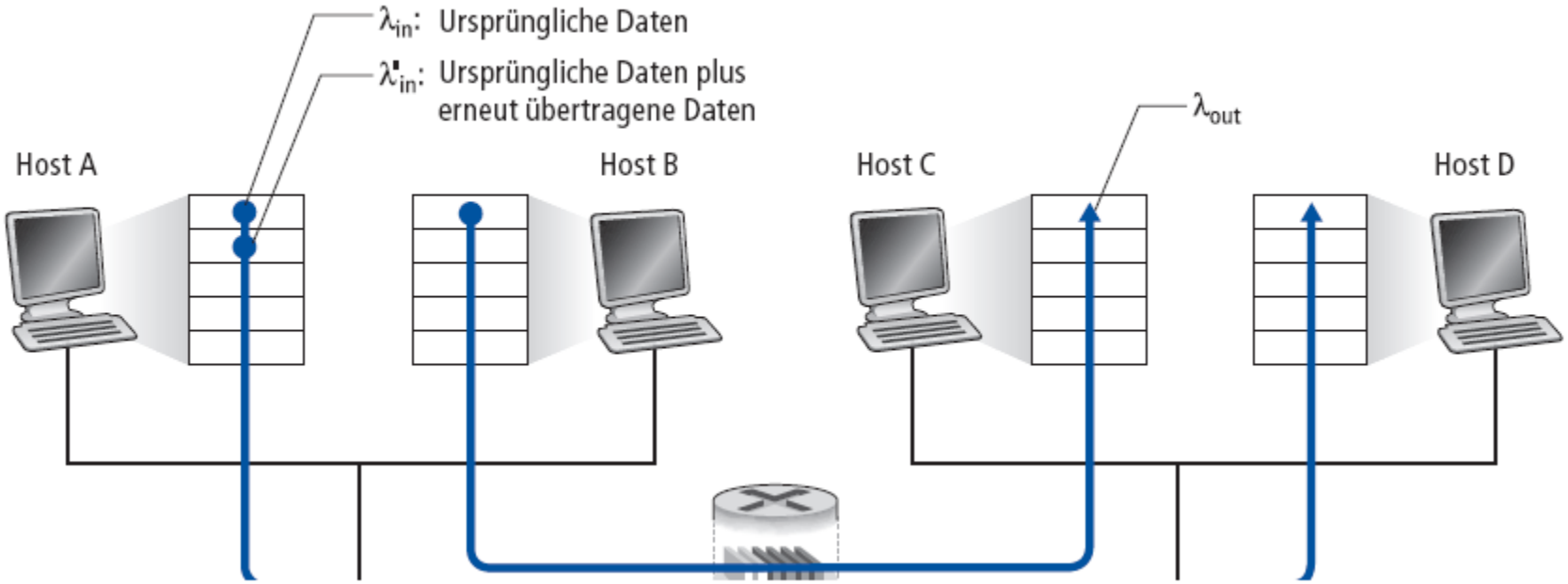
Szenario 1: zwei Quellen, ein Router mit unendlichen Puffern

- Rechter Graph: Verzögerung als Funktion der Senderate
  - Sobald sich die Senderate R/2 annähert wird die Verzögerung immer größer
  - Bei Senderaten > R/2: durchschnittliche Anzahl der Pakete in der Warteschlange des Routers ist unbegrenzt und die durchschnittliche Verzögerung zwischen Quelle und Ziel wird unendlich
  - → *Gesamtdurchsatz von beinahe R (= 2 Verbindungen mit jeweils Durchsatz R/2) scheint vom Standpunkt des Durchsatzes her durchaus ideal, da die gesamte Kapazität der Leitung ausgenutzt wird, ist vom Standpunkt der Verzögerung her aber problematisch!*



3.6.1 Ursachen und Kosten von Überlast

Szenario 2: zwei Quellen und ein Router mit beschränktem Puffer

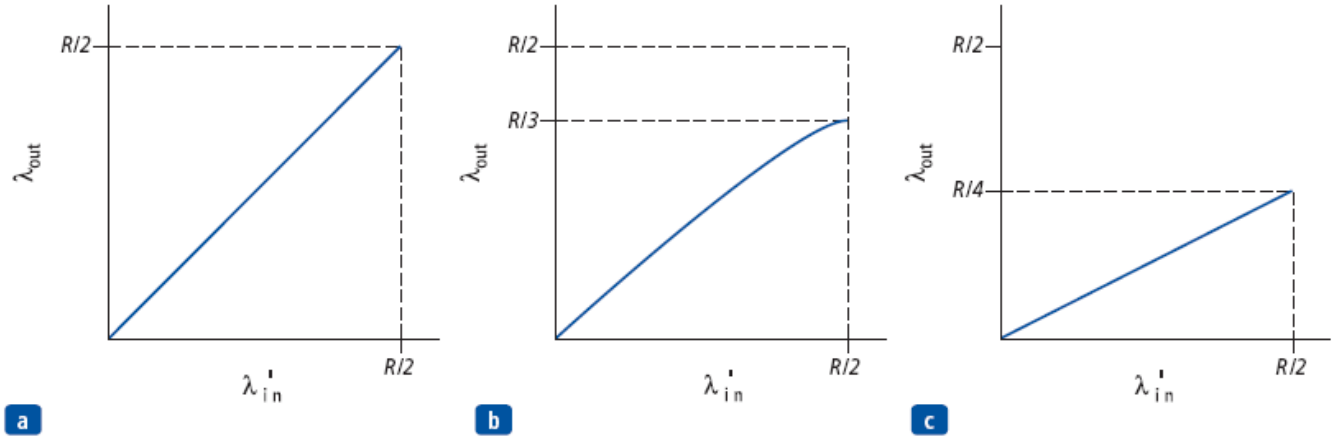


- Routerpuffer begrenzt, d.h. bei vollem Puffer werden weitere Pakete verworfen
- Übertragungswiederholung des Senders, wenn vom Router ein Paket verworfen wurde, das ein Transportschichtsegment enthielt
- **λ_{in} Byte/Sek** ist die Rate mit der eine Anwendung Daten in den Socket sendet. Die Rate mit der die Transportschicht Segmente ins Netz sendet (Originaldaten + erneut übertragene Daten) beträgt **λ'_{in} Byte/Sek (= offered load).**

3.6.1 Ursachen und Kosten von Überlast

Szenario 2: zwei Quellen und ein Router mit beschränktem Puffer

- Graph a: Stellt den unrealistischen Fall dar, daß der Host A nur dann Pakete sendet, wenn Puffer im Router frei ist.
  - Kein Verlust tritt auf: $\lambda_{in} = \lambda'_{in}$ und der Durchsatz der Verbindung ist gleich λ_{in}.
  - Die Senderate des durchschnittlichen Hosts kann R/2 nicht überschreiten, da vorrausgesetzt wird, daß Paketverlust nie stattfindet.
  - → *Vom Standpunkt des Durchsatzes ist die Leistung ideal: alles was verschickt wird, wird auch empfangen.*

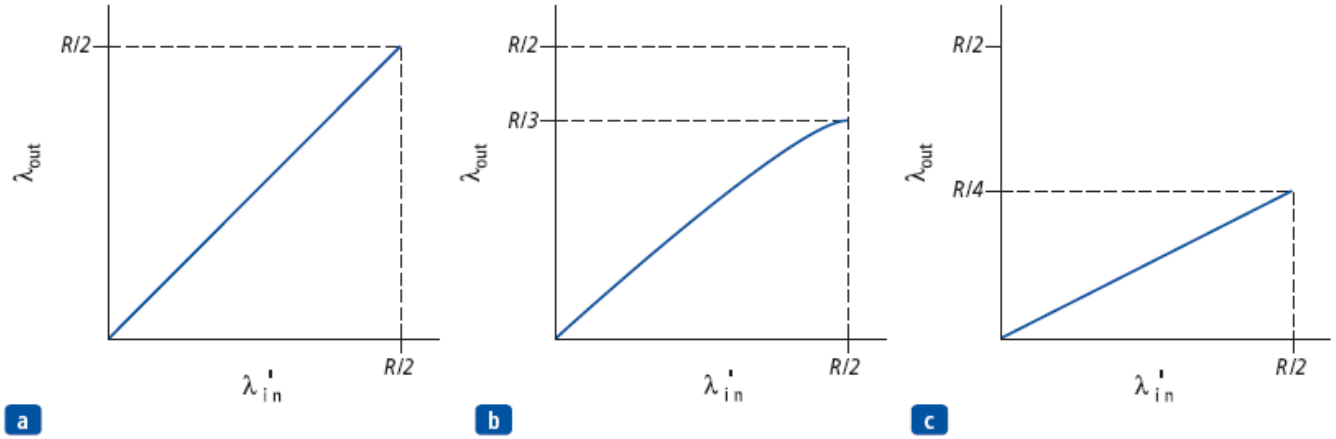


3.6.1 Ursachen und Kosten von Überlast

Szenario 2: zwei Quellen und ein Router mit beschränktem Puffer

- Graph b: Stellt den etwas realistischeren Fall dar, daß der Sender nur dann die Übertragung wiederholt, wenn er sicher weiß, daß ein Paket verloren gegangen ist.
  - Wenn die angebotene Last λ'_{in}= R/2 ist, dann ist die Rate λ_{out} , mit der die Daten an die empfangende Anwendung übermittelt werden, R/3.
  - Daher enthalten die 0,5*R gesendeten Daten durchschnittlich: 0,333*R Byte/Sek an Originaldaten und 0,166*R Byte/Sek an wiederholt übertragenen Daten.

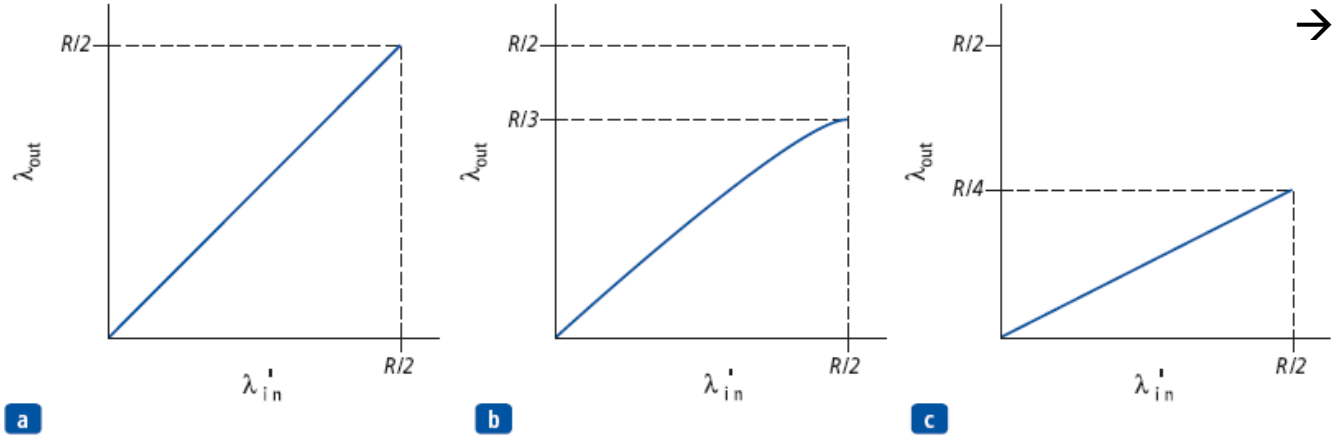
  → *Kostenfaktor eines überlasteten Netzwerkes: Wiederholtes Übertragen wegen Pufferüberläufen/zu hohen Latenzzeiten*



3.6.1 Ursachen und Kosten von Überlast

Szenario 2: zwei Quellen und ein Router mit beschränktem Puffer

- Graph c: Stellt den realistischen Fall dar, daß beim Sender vorzeitig Timer auslaufen und so Übertragungswiederholungen für Pakete auftreten können, die zwar in der Warteschlange verzögert wurden, aber nicht wirklich verloren gegangen sind.
  - Wenn sowohl Originaldatenpaket als auch Übertragungswiederholung den Empfänger erreichen verwirft dieser die Übertragungswiederholung. Der Router hätte die Übertragungskapazität der Verbindung besser verwendet, um stattdessen ein anderes Paket weiterzuleiten.
  - Wenn von jedem Paket angenommen wird, daß es im Durchschnitt zwei Mal vom Router weitergeleitet wird hat der Durchsatz einen asymptotischen Wert von R/4, während die angebotene Last gegen R/2 geht.

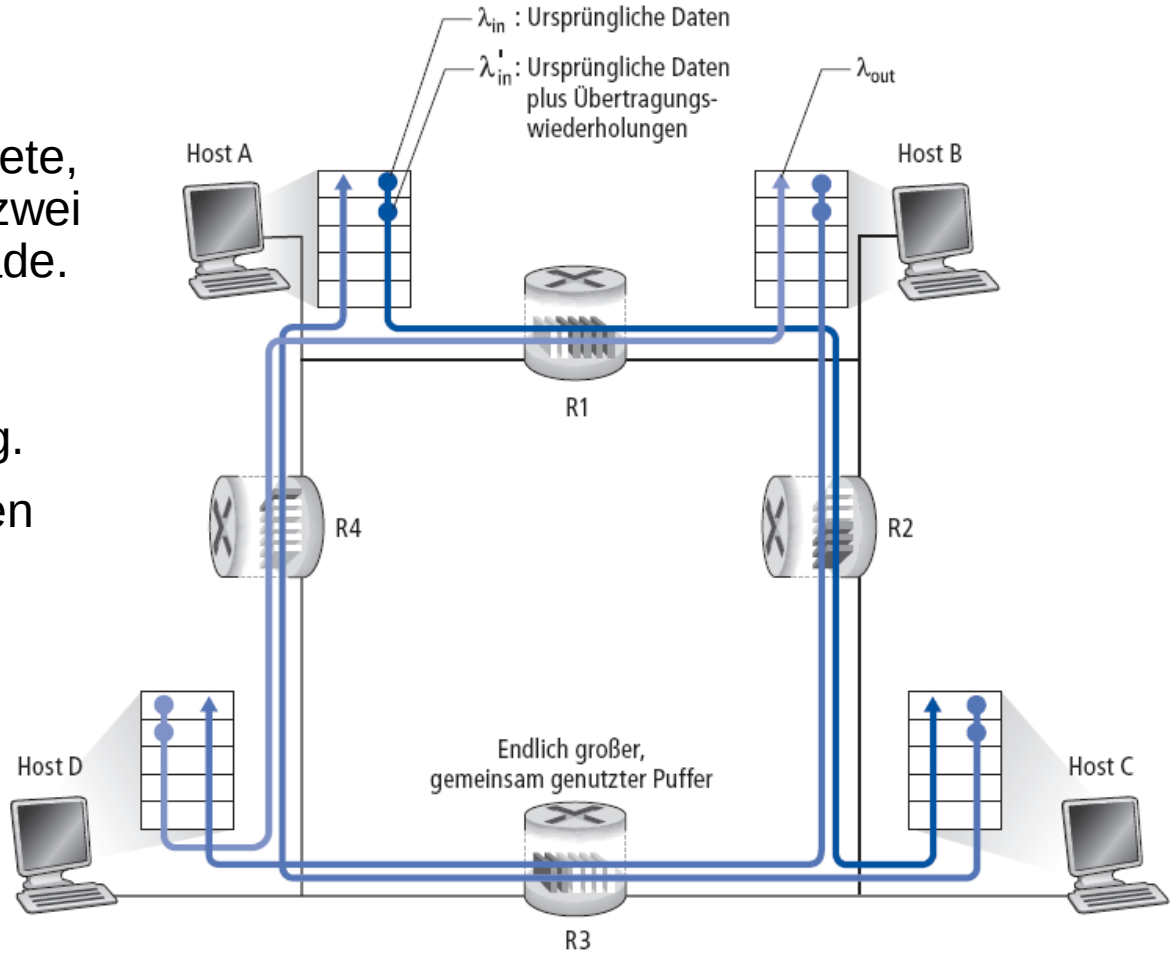


→ *Anderer Kostenfaktor: Unnötige Übertragungs-wiederholungen bei großen Verzögerungen bewirken, daß ein Router seine Verbindungsbandbreite zum Weiterleiten unnötiger Kopien eines Paketes verschwendet.*

3.6.1 Ursachen und Kosten von Überlast

Szenario 3: vier Quellen, Router mit beschränkten Puffern und Multihop-Pfade

- Vier Hosts übertragen Pakete, jeder über überlappende, zwei Router durchquerende Pfade.
- Zuverlässiger Datentransferdienst durch Übertragungswiederholung.
- Alle Hosts haben denselben Wert für λ_{in}.
- Alle Leitungen haben die Kapazität R Byte/Sek.

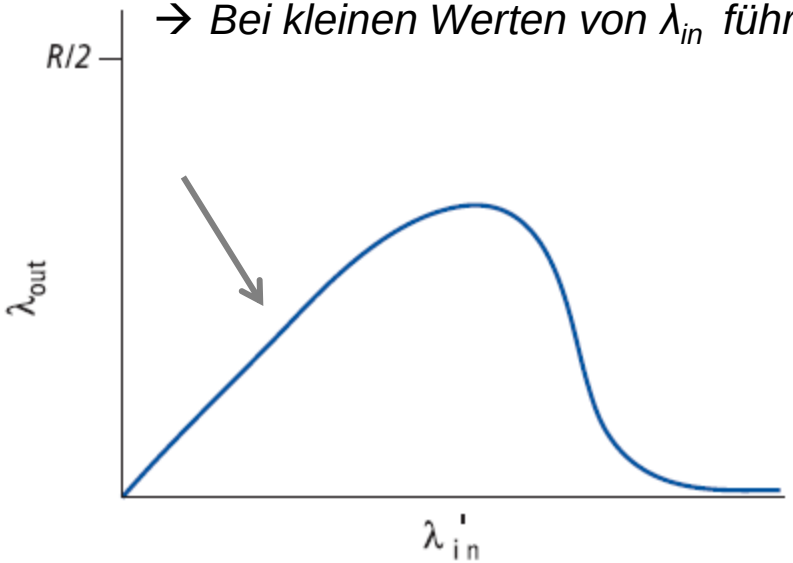


3.6.1 Ursachen und Kosten von Überlast

Szenario 3: vier Quellen, Router mit beschränkten Puffern und Multihop-Pfade

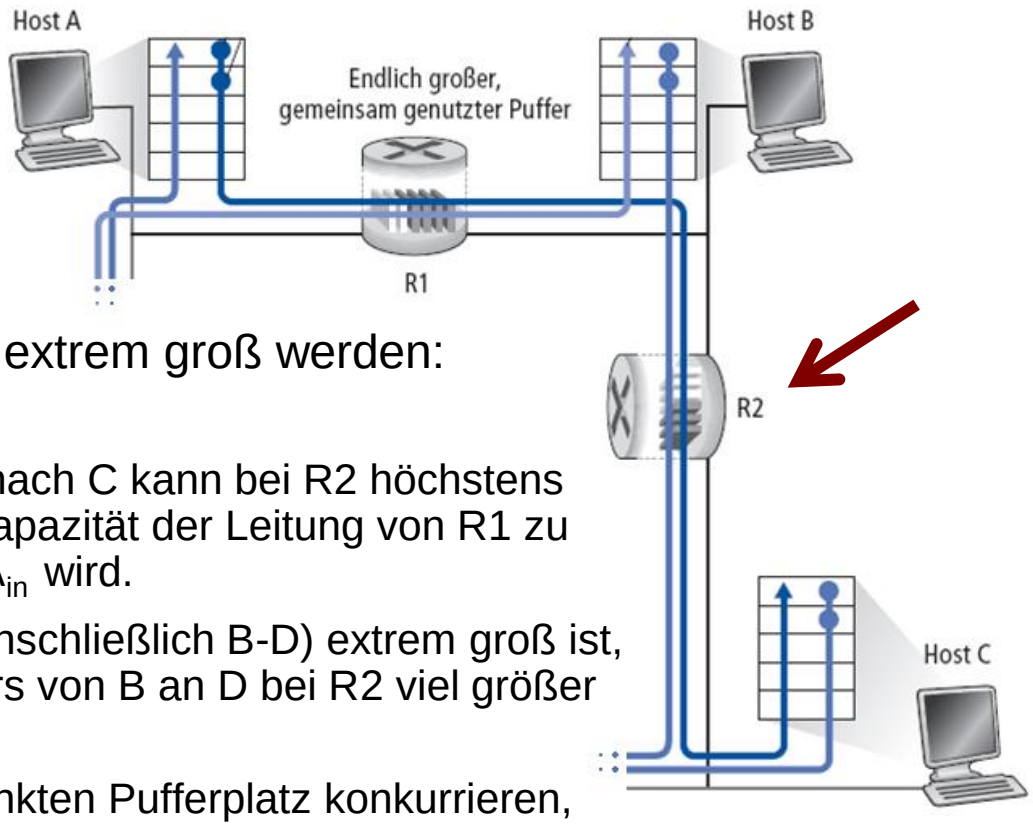
- Bsp. die Verbindung zwischen Hosts A und C teilt sich Router R1 mit mit der Verbindung D-B und Router R2 mit der Verbindung B-D.
- Fall äußerst geringen Verkehrsaufkommens:
  - Für äußerst kleine Werte λ_{in}: Pufferüberläufe sind wieder selten und der Durchsatz ist gleich der angebotenen Last.
  - Für geringfügig größere Werte λ_{in} : Pufferüberläufe immernoch selten, aber der Durchsatz ist größer, da weitere Originaldaten ins Netz gesendet und zum Zielort geliefert werden.

→ *Bei kleinen Werten von λ_{in} führt eine Zunahme von λ_{in} zu einer Zunahme von λ_{out}.*



3.6.1 Ursachen und Kosten von Überlast

Szenario 3: vier Quellen, Router mit beschränkten Puffern und Multihop-Pfade



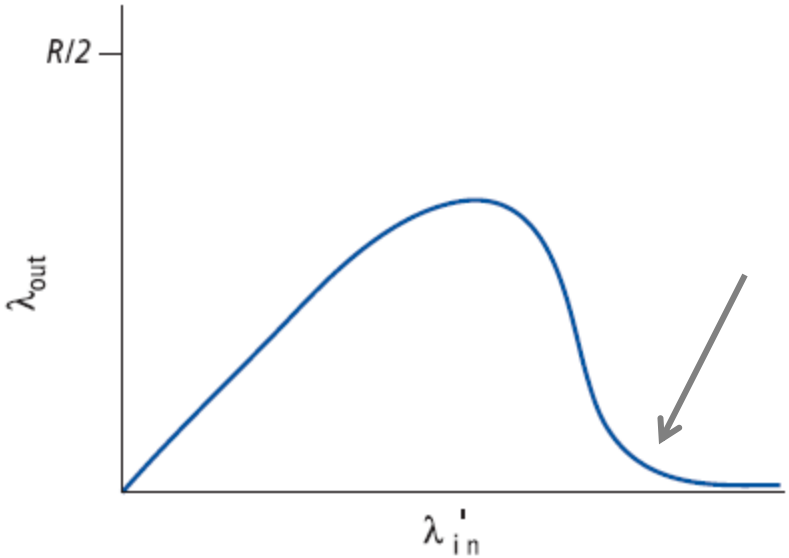
- Fall in dem λ_{in} (und daher auch λ'_{in}) extrem groß werden:
- Bsp. Router R2:
  - Der ankommende Verkehr von A nach C kann bei R2 höchstens eine Ankunftsrate von R haben (Kapazität der Leitung von R1 zu R2), unabhängig davon wie hoch λ_{in} wird.
  - Wenn λ_{in} für alle Verbindungen (einschließlich B-D) extrem groß ist, kann die Ankunftsrate des Verkehrs von B an D bei R2 viel größer sein als diejenige von A zu C.
  - Weil beide an R2 um den beschränkten Pufferplatz konkurrieren, verringert sich der erfolgreich weitergeleitete A-C-Verkehr (der also nicht durch Pufferüberlauf verloren geht) in dem Maße, in dem λ'_{in} von B-D immer größer wird.
  - Wenn λ'_{in} gegen unendlich geht: Ein leerer Puffer an R2 wird nun sofort von einem B-D Paket gefüllt und der Durchsatz der Verbindung A-C an R2 geht gegen Null (Abb: siehe folgende Folie).

3.6.1 Ursachen und Kosten von Überlast

Szenario 3: vier Quellen, Router mit beschränkten Puffern und Multihop-Pfade

- Bei hohem Verkehrsaufkommen (siehe vorige Folie) kann also der Fall auftreten, daß ein Paket das von R1 weitergeleitet wurde von R2 verworfen wird. In dieser Situation ist die von R1 beim Weiterleiten geleistete Arbeit vergeudet.
- Wenn R1 die Übertragungskapazität, die für diese Pakete aufgewendet wurde, statt dessen für ein anderes Paket genutzt hätte wäre dies profitabler gewesen!

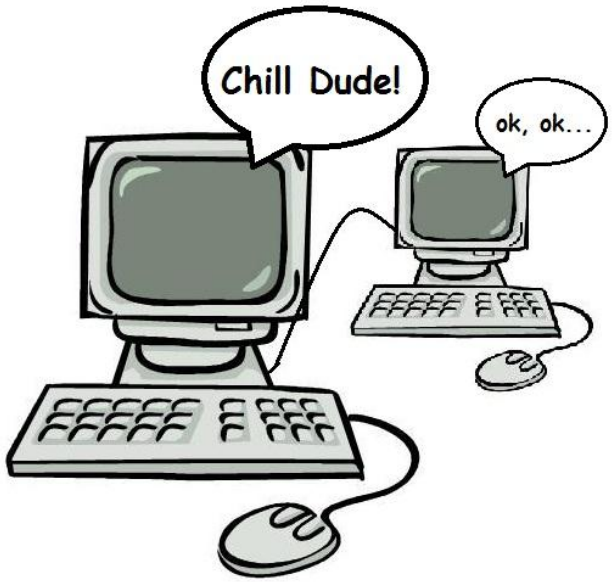
→ *Kostenfaktor beim Verwerfen eines Paketes aufgrund von Überlast: Wird ein Paket verworfen, ist die Übertragungskapazität verschwendet, die auf früheren Leitungen dafür benötigt wurde.*



3.6.2 Ansätze zur Überlastkontrolle

Ende-zu-Ende-Überlastkontrolle:

- Keine explizite Unterstützung durch das Netzwerk
- Überlast wird von den Endsystemen durch Paketverlust und erhöhte Verzögerung festgestellt
- Dies ist das Vorgehen im Internet (TCP)
- Explicit Congestion Notification (ECN) wird derzeit untersucht



3.6.2 Ansätze zur Überlastkontrolle

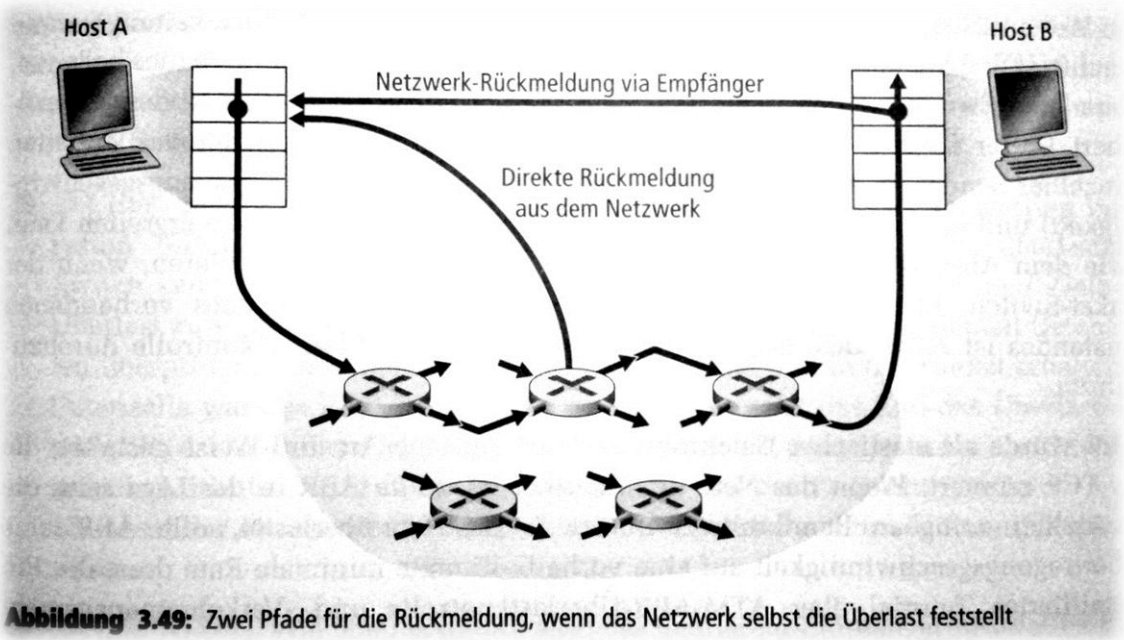


Abbildung 3.49: Zwei Pfade für die Rückmeldung, wenn das Netzwerk selbst die Überlast feststellt

Netzwerkunterstützte Überlastkontrolle:

- Router geben den Endsystemen Hinweise. Zwei Möglichkeiten:
  - Direkte Rückmeldung vom Router an den Sender (z.B. in Form eines Choke-Pakets)
  - Durch ein einzelnes Bit, welches im Paketheader von den Routern gesetzt werden kann (SNA, DECbit, TCP/IP ECN, ATM). Beim Empfang eines derart markierten Pakets informiert der Empfänger den Sender über die Überlast.

3.6.3 ATM-ABR-Überlastkontrolle

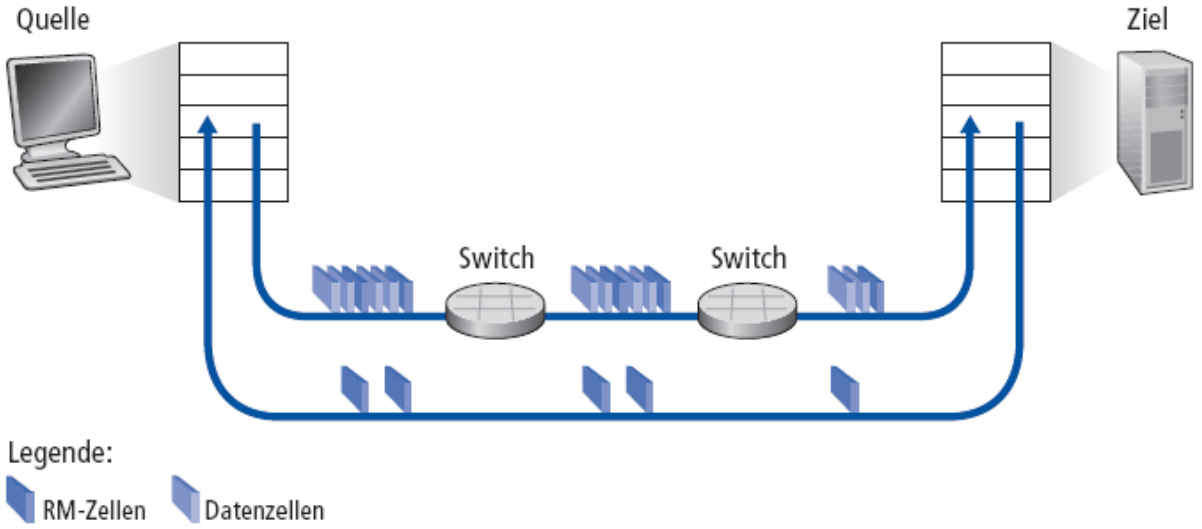
ABR (Available Bit Rate):

- Bandbreitenelastischer Dienst
- Ist der Pfad nicht belastet → Sender sollte die verfügbare Bandbreite nutzen
- Ist der Pfad überlastet → Reduktion auf die minimal garantierte Rate
- Ratenbasierter Ansatz: Sender berechnet maximale Senderate und regelt sich selbst

Spezielle Pakete: **RM-Zellen (resource management)**:

- Vom Sender abgeschickt, mit Datenzellen gemischt
- Bits in einer RM-Zelle werden durch Switches geändert (Unterstützung durch das Netzwerk)
  - **NI (No Increase) Bit**: keine Erhöhung der Rate (geringe Überlast)
  - **CI (Congestion Indication) Bit**: zeigt Überlast an
- RM-Zellen werden vom Empfänger an den Sender zurückgeschickt

3.6.3 ATM-ABR-Überlastkontrolle



- Zwei Byte langes **ER-Feld (explicit rate)** in der RM-Zelle
  - Überlasteter Switch kann den Wert in diesem Feld verringern
  - Senderate wird dann an die maximale Rate, die vom Gesamtpfad unterstützt wird, angepasst
- **EFCI-Bit** in Datenzellen: wird von überlasteten Switches auf 1 gesetzt
  - Wenn eine Datenzelle mit gesetztem EFCI-Bit ankommt, dann setzt der Empfänger das **CI-Bit** für die nächste RM-Zelle, die er an den Sender zurückschickt

3.7 TCP-Überlastkontrolle

TCP muss statt netzwerkunterstützter Überlastkontrolle (wie bei ABR) eine Ende-zu-Ende Überlastkontrolle verwenden, da die IP-Schicht den Endsystemen hinsichtlich der aktuellen Netzlast keine explizite Rückmeldung liefert!

→ Also stellt jeder Sender bei TCP seine Senderate als Funktion der wahrgenommenen Überlast selbst ein.

Das wirft folgende Fragen auf:

1. *Wie begrenzt ein TCP Sender die Rate, mit der er Daten über seine Verbindung sendet?*
2. *Wie erkennt ein TCP-Sender, daß es Überlast auf dem Pfad zwischen sich und dem Sender gibt?*
3. *Welchen Algorithmus sollte der Absender verwenden um sein Sendetempo als Funktion der erkannten Überlastzwischen den Endpunkten zu ändern?*

3.7 TCP-Überlastkontrolle

1. *Wie begrenzt ein TCP Sender die Rate, mit der er Daten über seine Verbindung sendet?*
   - Jede Seite einer TCP-verbindung besteht aus einem Eingangspuffer, einem Sendepuffer, mehreren Variablen (LastByteRead, RcvWindow usw.) und der Variable **CongWin** (Congestion Window – Überlastfenster).
   - Das CongWin beschränkt die Rate mit der ein TCP-Sender Verkehr ins Netz senden kann. Die Menge an unbestätigten Daten eines Senders darf das Minimum von CongWin nicht übersteigen:

     `LastByteSent – LastByteAcked <= min {`*`CongWin`*`}`

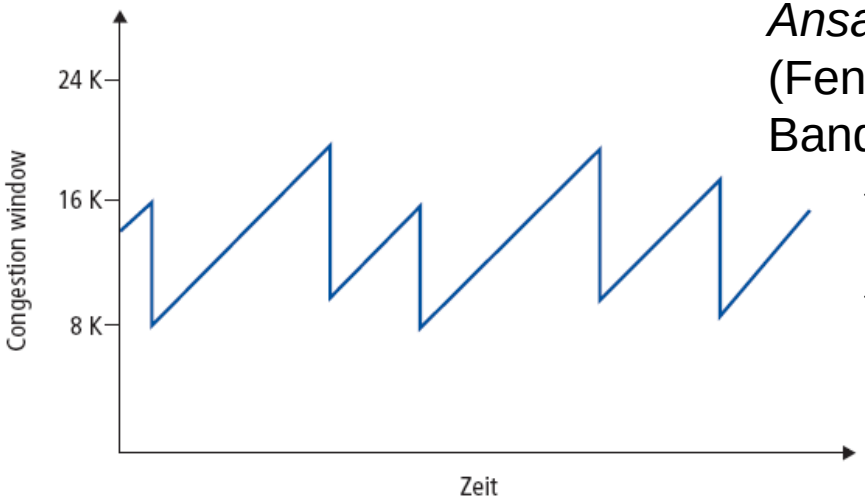
2. *Wie erkennt ein TCP-Sender, daß es Überlast auf dem Pfad zwischen sich und dem Sender gibt?*
   - Überlast wird am TCP-Sender auf zwei mögliche Arten wahrgenommen: durch das Auftreten eines Timeouts oder durch den Erhalt von drei doppelten ACKs
   - Tritt kein Verlustereignis auf, empfängt der TCP-Sender ACKs für seine zuvor unbestätigten Segmente und vergrößert sein CongWin (und damit die Übertragungsgeschwindigkeit).

3.7 TCP-Überlastkontrolle

3. Welchen Algorithmus sollte der Absender verwenden um sein Sendetempo als Funktion der erkannten Überlastzwischen den Endpunkten zu ändern?

→ Der **TCP-Überlastkontroll-Algorithmus** besteht aus drei Hauptbestandteilen:

- Additive-Increase, Multiplicative-Decrease
- Slow Start und
- Reaktion auf Timeout-Ereignisse



<u>Additive-Increase, Multiplicative-Decrease:</u>

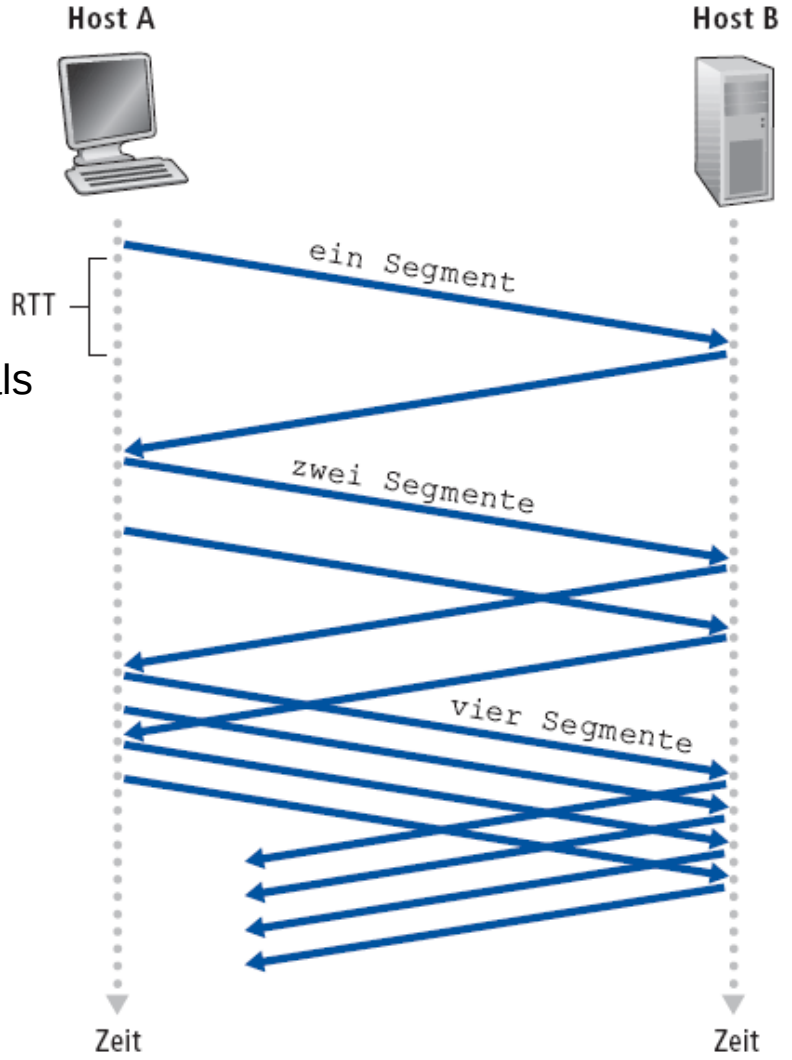
Ansatz: Erhöhe die Übertragungsrate (Fenstergröße), um nach überschüssiger Bandbreite zu suchen, bis ein Verlust eintritt

- **Additive Increase**: Erhöhe CongWin um eine MSS pro RTT, bis Verlust erkannt wird
- **Multiplicative Decrease**: Halbiere CongWin, wenn ein Verlust erkannt wird

3.7 TCP-Überlastkontrolle

Slow Start (SS ~ langsamer Anfang)

- Bei Verbindungsbeginn:
  `CongWin` = 1 MSS (Maximum Segment Size)
  - *Beispiel:* MSS = 500 Byte & RTT = 200 ms
  - Initiale Rate = 20 kBit/s
  - Verfügbare Bandbreite kann aber viel größer als MSS/RTT sein!
- Die Rate sollte sich schnell der verfügbaren Rate anpassen, deshalb bei Verbindungs-beginn: Erhöhe die Rate exponentiell schnell bis zum ersten Verlustereignis
  - Angestrebt: Verdoppeln von **`CongWin`** in jeder RTT
  - Realisiert: **`CongWin`** um 1 für jedes erhaltene ACK erhöhen
- Ergebnis: *Initiale Rate ist gering, wächst aber exponentiell schnell!*



3.7 TCP-Überlastkontrolle

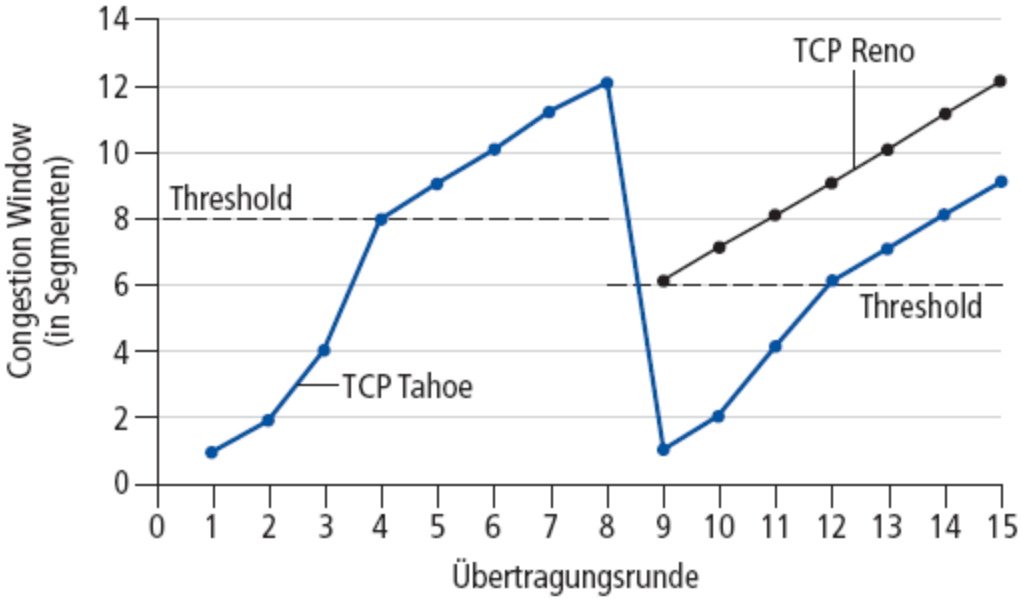
Slow Start (SS ~ langsamer Anfang)

- *Wann soll vom exponentiellen Wachstum zum linearen Wachstum übergegangen werden?*

→ Wenn **`CongWin`** die Hälfte des Wertes erreicht, den es vor dem Verlustereignis hatte.

Implementierung:

- Variabler `Threshold`
- Bei einem Verlustereignis wird `Threshold` auf die Hälfte des Wertes gesetzt, den es vor dem Verlustereignis hatte.



3.7 TCP-Überlastkontrolle

Reaktion auf Timeouts

TCP-Überlastkontrolle reagiert unterschiedlich auf die zwei Arten der Verlustereignisse:

- Nach drei doppelten ACKs:
  - **`CongWin`** halbieren ("Fast Recovery"), `Threshold` auf diesen Wert setzen
  - Fenster wächst danach linear

- Nach Timeout:
  - **`CongWin`** auf eine MSS setzen, `Threshold` auf die Hälfte des alten Werts von **`CongWin`** setzen
  - Fenster wächst dann exponentiell, bis `Threshold` erreicht ist
  - Fenster wächst danach linear

→ Grund für unterschiedliche Reaktion: drei doppelte ACKs zeigen an, dass das Netzwerk noch in der Lage ist, Pakete auszuliefern. Bei Timeout erfolgt gar keine Rückmeldung mehr.

3.7 Zusammenfassung TCP-Überlastkontrolle

Slow-Start-Phase: **`CongWin`** kleiner als `Threshold`	Das Fenster wächst exponentiell.
Congestion-Avoidance-Phase: **`CongWin`** größer als `Threshold`	Das Fenster wächst linear.
Bei **drei doppelten ACKs**:	`Threshold` wird auf **`CongWin/2`** gesetzt und dann **`CongWin`** auf ssthresh.
Bei einem **Timeout**:	`Threshold` wird auf **`CongWin/2`** und **`CongWin`** auf 1 MSS gesetzt.

3.7 Zusammenfassung TCP-Überlastkontrolle

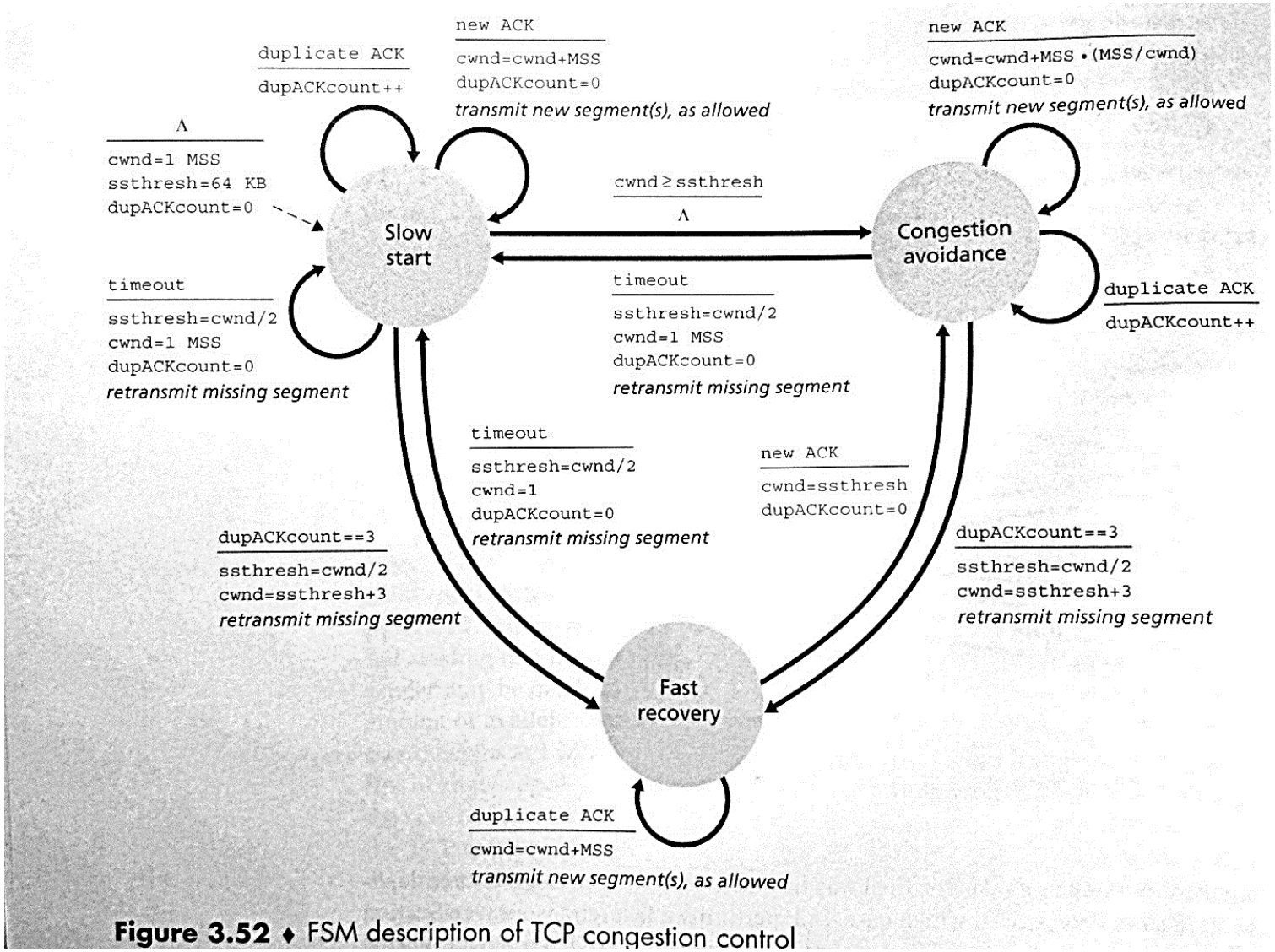


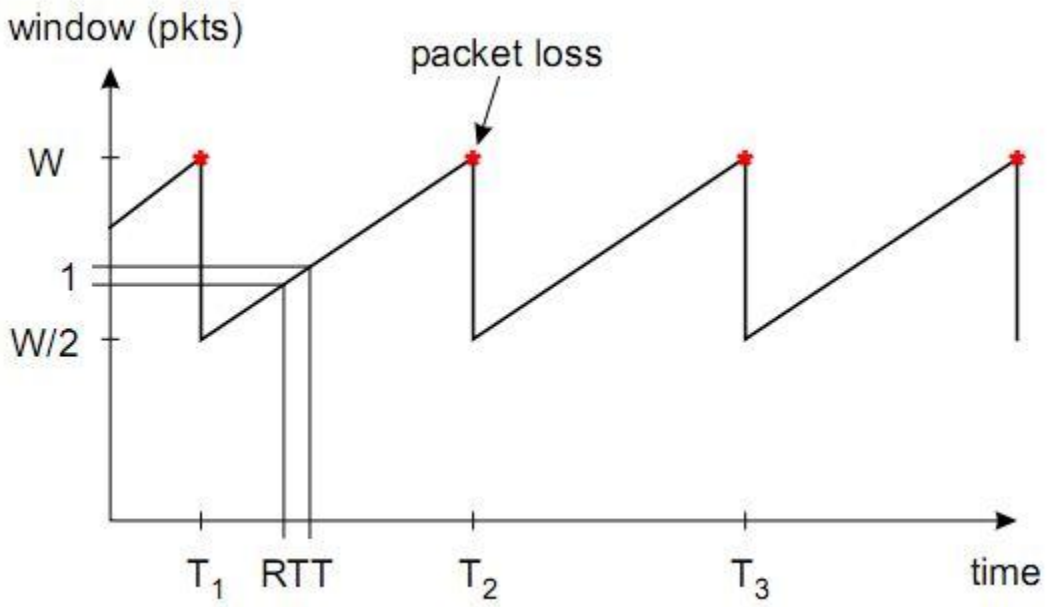
Figure 3.52 ♦ FSM description of TCP congestion control

3.7 Zusammenfassung TCP-Überlastkontrolle

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.

Die dargestellten Zustandswerte sind die Zustände des TCP-Senders unmittelbar bevor die beschriebenen Ereignisse eintreten.

3.7 TCP Durchsatzanalyse



3.7 TCP-Durchsatz

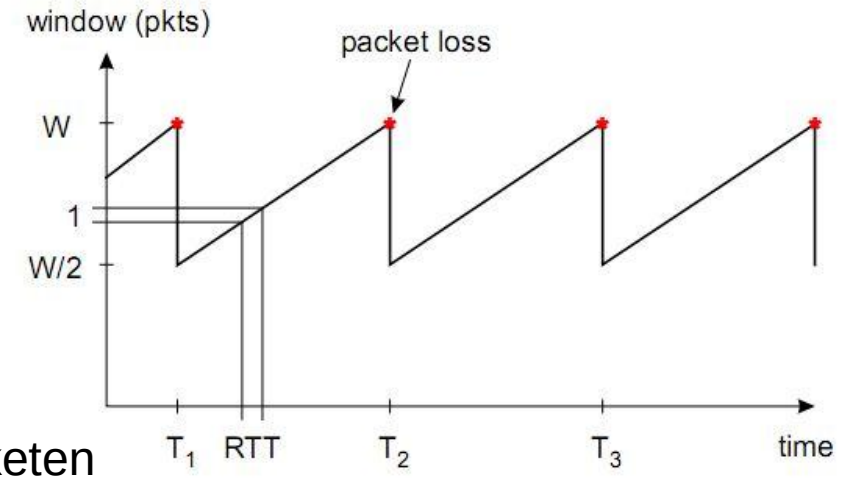
Frage nach dem durchschnittlichen TCP-Durchsatz durch Sägezahnverhalten der Senderate schwierig.

→ Wiederholte Slow-Start-Phasen nach Timeouts werden im Folgenden vernachlässigt, da normalerweise sehr kurz.

- 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 0.5 W/RTT

→ Durchschnittlicher Durchsatz: 0.75 W/RTT

3.7 TCP Durchsatzanalyse



Anzahl der gesendeten Pakete zwischen zwei verworfenen Paketen:
(z.B. zwischen T1 und T2)

$$\frac{W}{2} + (\frac{W}{2} + 1) + \cdots + W \approx \frac{3}{8}W^2$$

Wenn also von den gesendeten (3/8) W^2 Paketen wurde 1 verworfen:

$$p = \frac{8}{3W^2} \quad \Rightarrow \quad W = \sqrt{\frac{8}{3p}}$$

Die Zeit zwischen zwei verworfenen Paketen ist W/(2 RTT). Also berechnen wir den Durchsatz T folgendermaßen:

$$T = \frac{(3/8)W^2B}{(W/2)RTT} = \frac{3}{4}\frac{W \cdot B}{RTT} = \frac{\sqrt{3/2}\,B}{RTT\sqrt{p}} \approx \frac{1.22\,B}{RTT\sqrt{p}}$$

…wobei B die Größe eines einzelnen Segments in Bytes ist (=MSS).

3.7 Die Zukunft von TCP

Die TCP-Überlastkontrolle hat sich im Laufe vieler Jahre entwickelt. Die Anforderungen ändern sich, was am folgenden Beispiel verdeutlicht werden soll.

Beispiel:

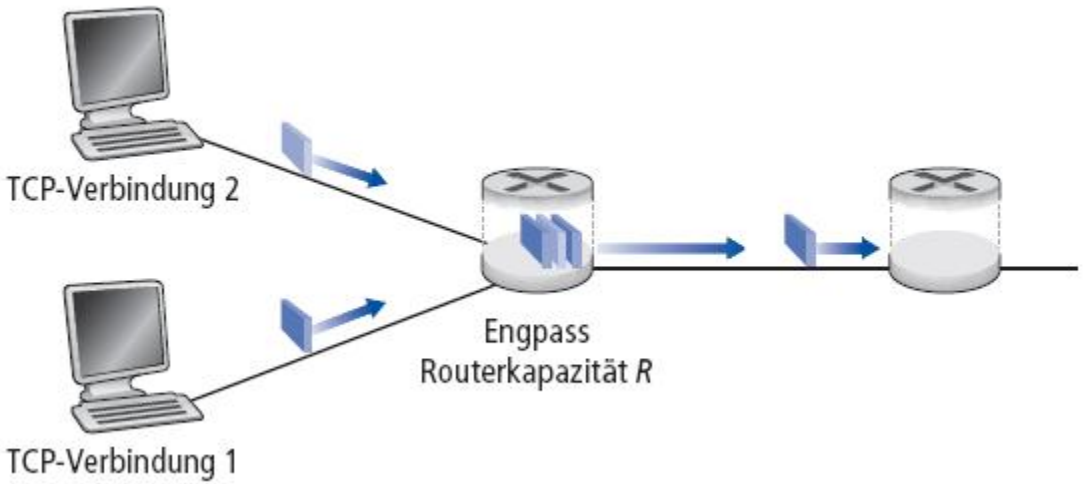
- 1500-Byte-Segmente, 100ms RTT, wir wollen 10 GBit/s als Durchsatz
- Erfordert Fenstergröße W = 83,333
- Durchsatz in Abhängigkeit von der Verlustrate:

$$\frac{1.22 \cdot MSS}{RTT\sqrt{p}}$$

→ $p = 2 \cdot 10^{-10}$ ***Wow!***

- Erfordert neue Versionen von TCP für Hochgeschwindigkeitsnetze!

3.7.1 Fairness



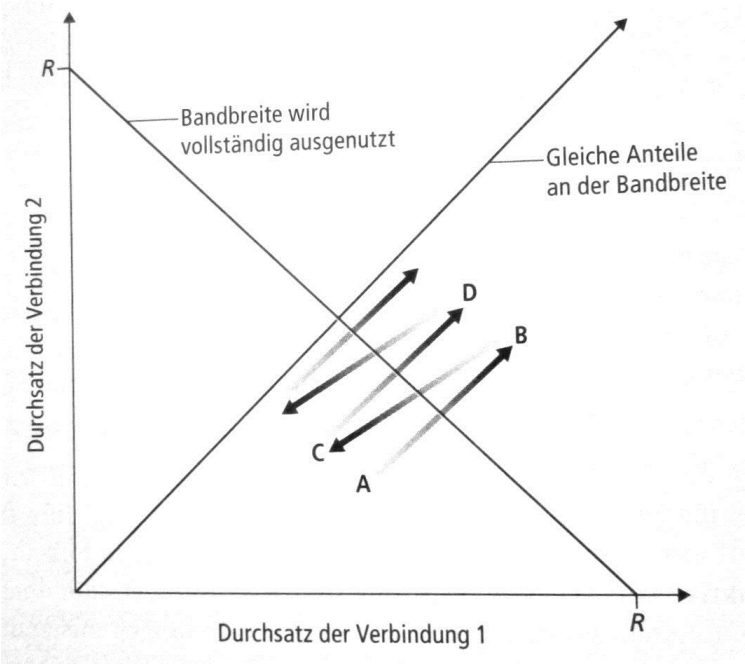
Ziel: Wenn K TCP-Sitzungen sich denselben Engpass mit Bandbreite R teilen, dann sollte jede Sitzung eine durchschnittliche Rate von R/K erhalten

3.7.1 Fairness von TCP

Warum ist TCP fair?

Bsp. Zwei Verbindungen im Wettbewerb:

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



3.7.1 Fairness

Fairness und UDP

- Viele Multimediaanwendungen verwenden kein TCP, denn sie wollen nicht, dass die Rate durch Überlastkontrolle reduziert wird
- Stattdessen Einsatz von UDP
  → Audio-/Videodaten mit konstanter Rate ins Netz leiten, Verlust hinnehmen
- TCP vermindert seine Übertragungsrate angesichts wachsender Überlast, UDP nimmt darauf keine Rücksicht
  → UDP-Quellen können den TCP-Verkehr möglicherweise verdrängen

- Forschungsgebiet: Überlastkontrollmechanismen für das Internet, die den UDP-Verkehr daran hindern den TCP-Verkehr zu verdrängen.

3.7.1 Fairness

Fairness und parallele TCP-Verbindungen

- Eine Anwendung kann zwei oder mehr parallele TCP-Verbindungen öffnen
- Webbrowser machen dies häufig
- Beispiel:
  Eine Leitung hat eine **Rate von R**, über die **neun Anwendungen** je eine TCP-Verbindung unterhalten.
  - Kommt eine neue Anwendung und legt **eine neue TCP-Verbindung** an, dann bekommt jede der insgesamt zehn Verbindungen ungefähr dieselbe Übertragungsgeschwindigkeit **R/10**
  - Wenn die neue Anwendung aber **elf neue TCP-Verbindungen** anlegt, dann erhält sie **mehr als R/2**!