

**Folien zur Veranstaltung
Rechnernetze in der AI
Wintersemester 2018
(Teil 4)**

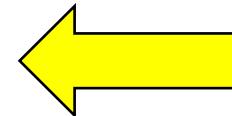
Prof. Dr. Franz Korf
Franz.Korf@haw-hamburg.de

Basierend auf der RN Vorlesung von M. Hübner

Kapitel 4: Transportschicht

Gliederung

- Dienste und Prinzipien auf der Transportschicht
- Multiplexen und Demultiplexen von Anwendungen
- Verbindungsloser Transport: UDP
- Prinzipien des zuverlässigen Datentransfers
- Verbindungsorientierter Transport: TCP
- TCP – Überlastkontrolle (Staukontrolle)
- Zusammenfassung



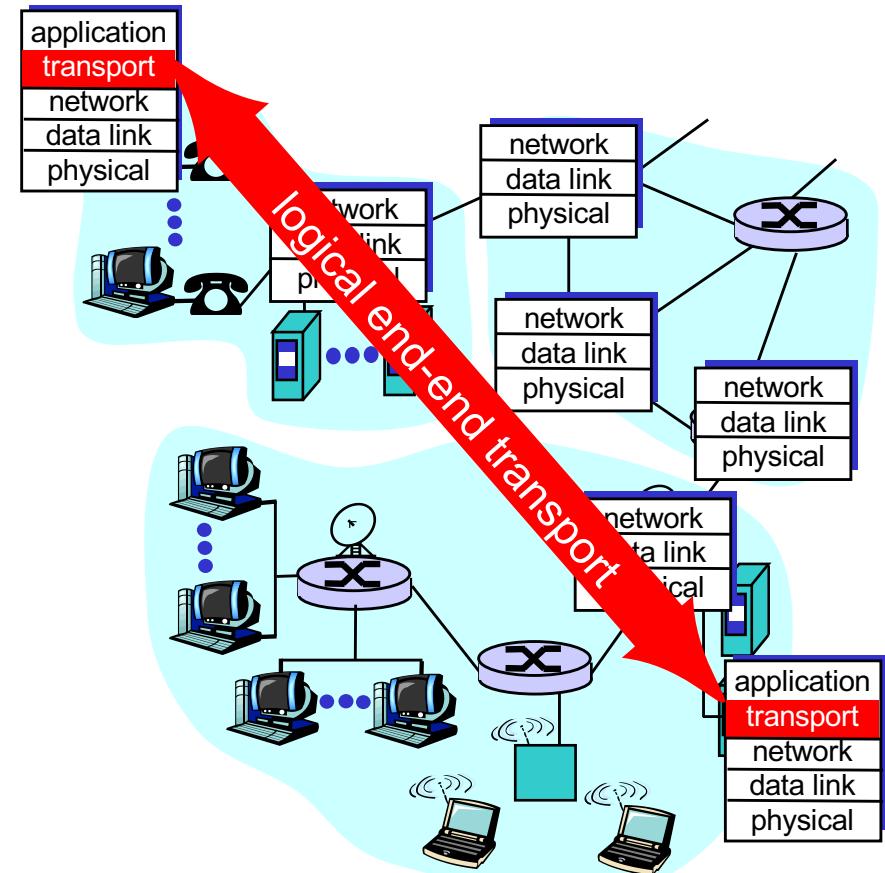
Textbuch zu diesem Kapitel: J. Kurose & K. Ross: Computernetzwerke – Der Top-Down-Ansatz, Kapitel 3

Folien und Abbildung teilweise aus:

J. Kurose & K. Ross: Computernetzwerke – Der Top-Down-Ansatz

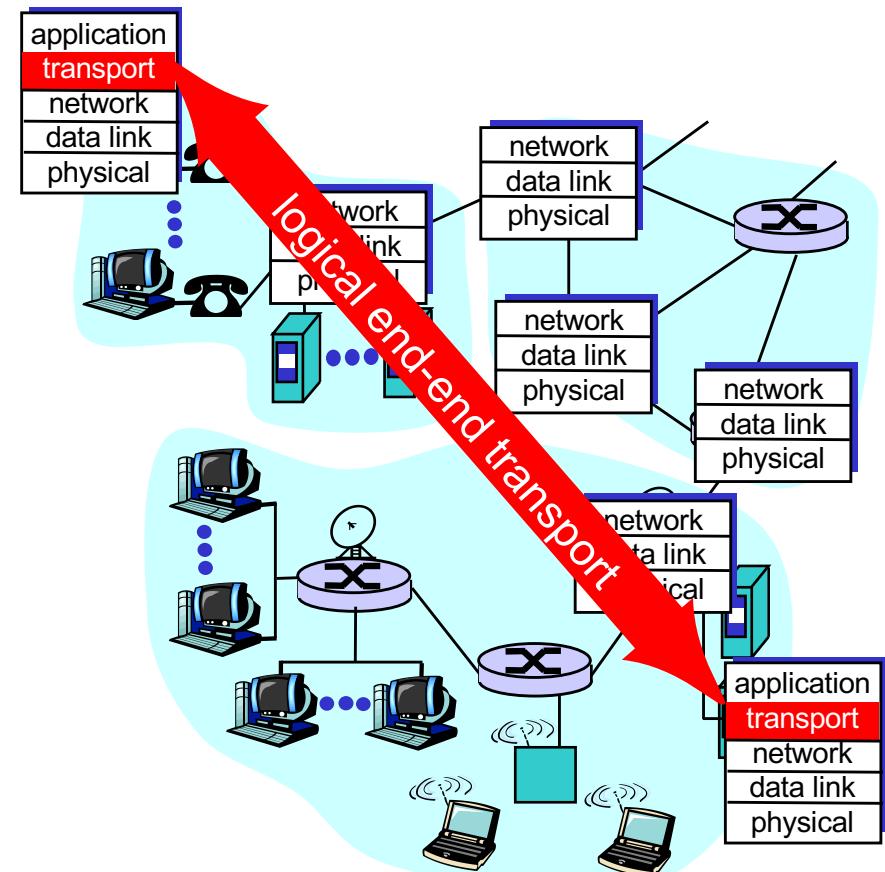
Transportschicht-Dienste

- Stellen eine **logische Kommunikation** zwischen Anwendungsprozessen her, die auf unterschiedlichen Hosts laufen
- **Transportprotokolle laufen nur auf den Endgeräten**
 - Sender (auf Transportschicht): Erhält Daten von Anwendungen (Prozessen), teilt diese in Segmente ein und gibt sie an die Netzwerkschicht weiter
 - Empfänger (auf Transportschicht): Erhält Segmente von der Netzwerkschicht und gibt die Daten an den zugehörigen Anwendungsprozess weiter



Transportschicht-Dienste

- Identifikation des zugehörigen Prozesses über die "Portnummer"
- Analogie:
 - Mehrfamilienhaus mit Adresse (= Host)
 - Postfächer/Briefkästen (= Portnummern)
 - (Teilweise sind die Briefkästen nicht nur mit Port Nummer des Empfängers beschriftet)
- Transportschicht nutzt die Dienste der Netzwerkschicht.



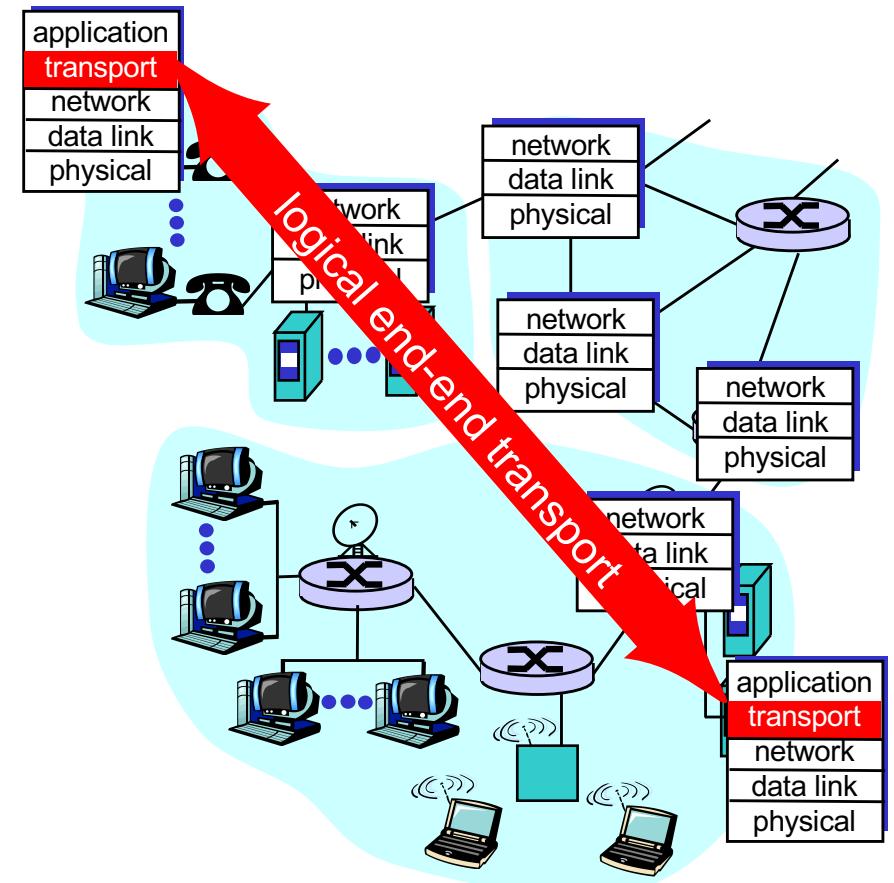
Transportschicht-Dienste

Transportschicht vs. Netzwerkschichtdienste:

- Transportschichtdienste: Bieten Datentransfer zwischen Prozessen an
- Netzwerkschichtdienste: Bieten Datentransfer zwischen Hosts (Rechnern)

Internet-Transportprotokolle:

- **TCP**: Verlässlicher, reihenfolge-erhaltender Punkt-zu-Punkt-Transport (Unicast Transport)
 - Wie kann man ein verlässliches Protokoll auf einen unzuverlässigen Protokoll aufbauen?
- **UDP**: Unzuverlässiger (“best-effort”), ungeordneter Punkt-zu-Punkt- und Multicast-Transport



Transportschicht-Dienste

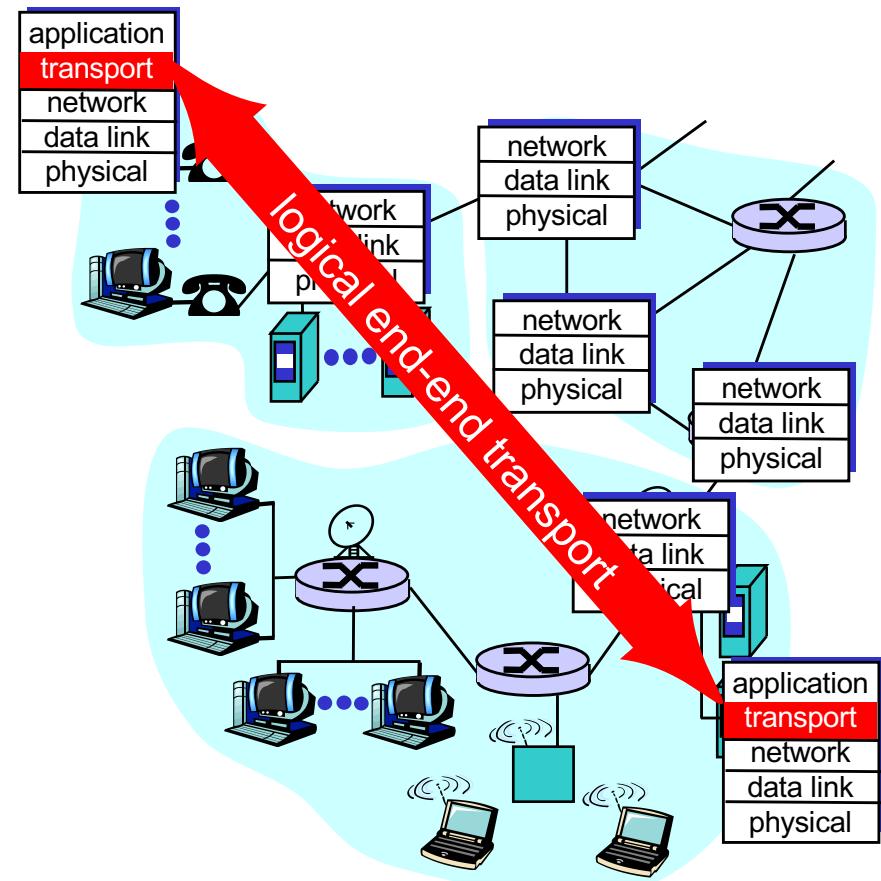
(Erstmal) nicht unterstützt werden:

- Garantierte Latenzen
 - Garantierte Bandbreiten
- Kann auf höheren Protokollebenen nicht „nachbilden“ werden – untere Protokollsichten müssen diese Garantien liefern.

Multiplexen und Demultiplexen:

- Verteilen der Messages Botschaften auf Ports

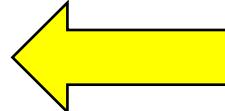
Flusskontrolle und Überlastkontrolle



Kapitel 4: Transportschicht

Gliederung

- Dienste und Prinzipien auf der Transportschicht
- Multiplexen und Demultiplexen von Anwendungen
- Verbindungsloser Transport: UDP
- Prinzipien des zuverlässigen Datentransfers
- Verbindungsorientierter Transport: TCP
- TCP – Überlastkontrolle (Staukontrolle)
- Zusammenfassung



Textbuch zu diesem Kapitel: J. Kurose & K. Ross: Computernetzwerke – Der Top-Down-Ansatz, Kapitel 3

Folien und Abbildung teilweise aus:

J. Kurose & K. Ross: Computernetzwerke – Der Top-Down-Ansatz

Multiplexen / Demultiplexen

Multiplexen:

- Einsammeln der Daten von mehreren Anwendungsschicht-Prozessen,
- Einpacken der Daten mit Steuerinformationen (→ Segmente)

Demultiplexen:

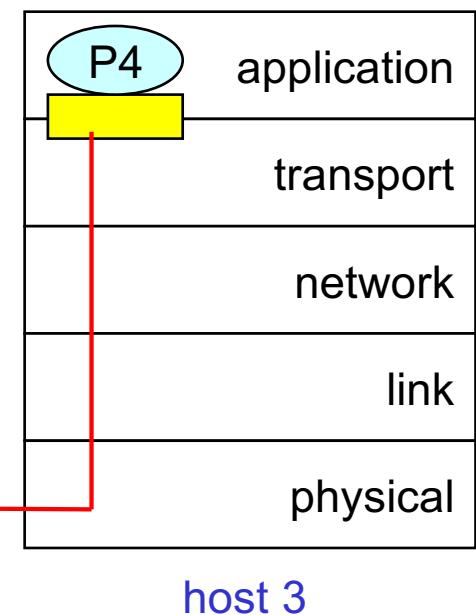
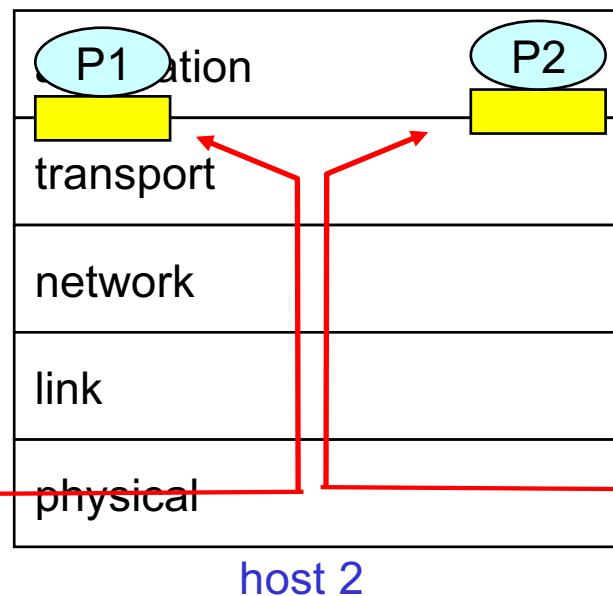
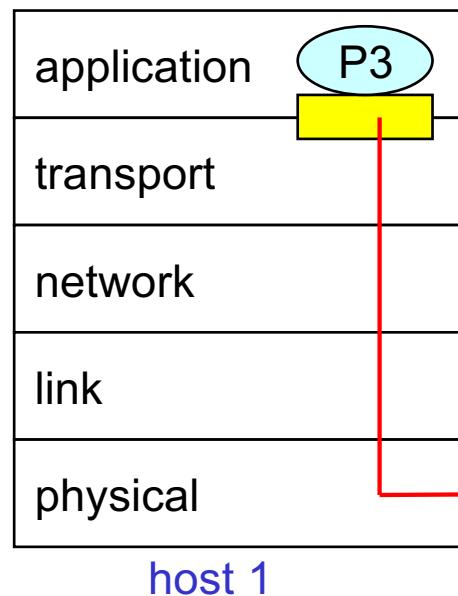
- Abliefern der empfangenen Segmente / Daten beim richtigen Anwendungsschicht-Prozess



= socket



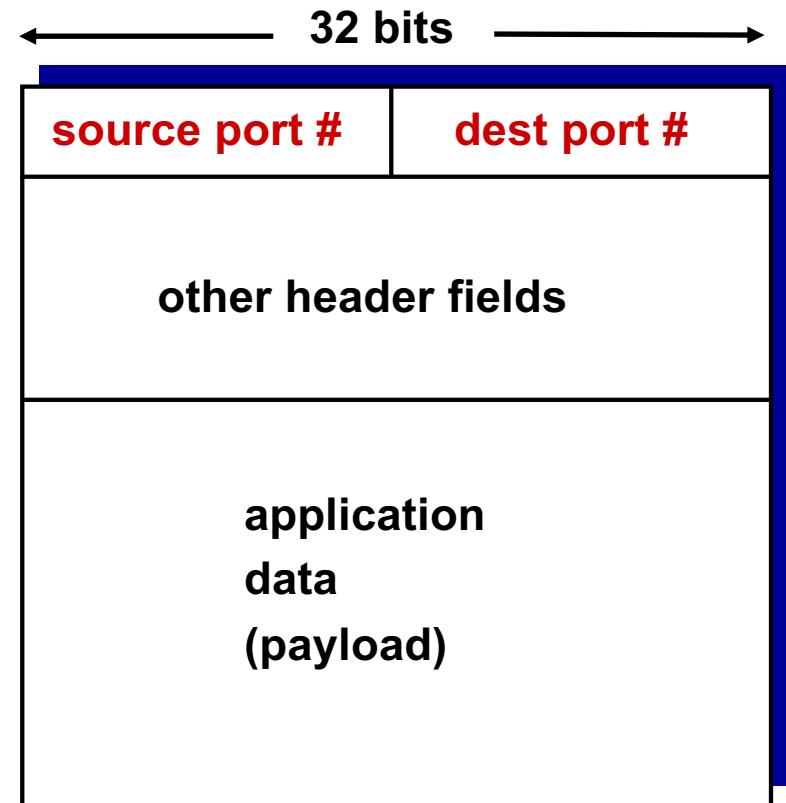
= process



Wie funktioniert “Demultiplexen”?

Der Empfänger erhält IP-Datagramme

- Jedes IP-Datagramm enthält eine IP-Quelladresse und IP-Zieladresse
- Jedes IP-Datagramm transportiert genau ein Segment der Transportschicht
- Jedes Segment hat eine Quell-Portnummer (als Absender-ID) und Ziel-Portnummer (als Empfänger-ID)
- Der Empfänger benutzt die IP-Adressen und Port-Nummern, um das Segment dem korrekten Socket zu übergeben
- UDP und TCP unterschiedliches Demultiplexen



TCP/UDP segment format

Verbindungsloses Demultiplexen (UDP)

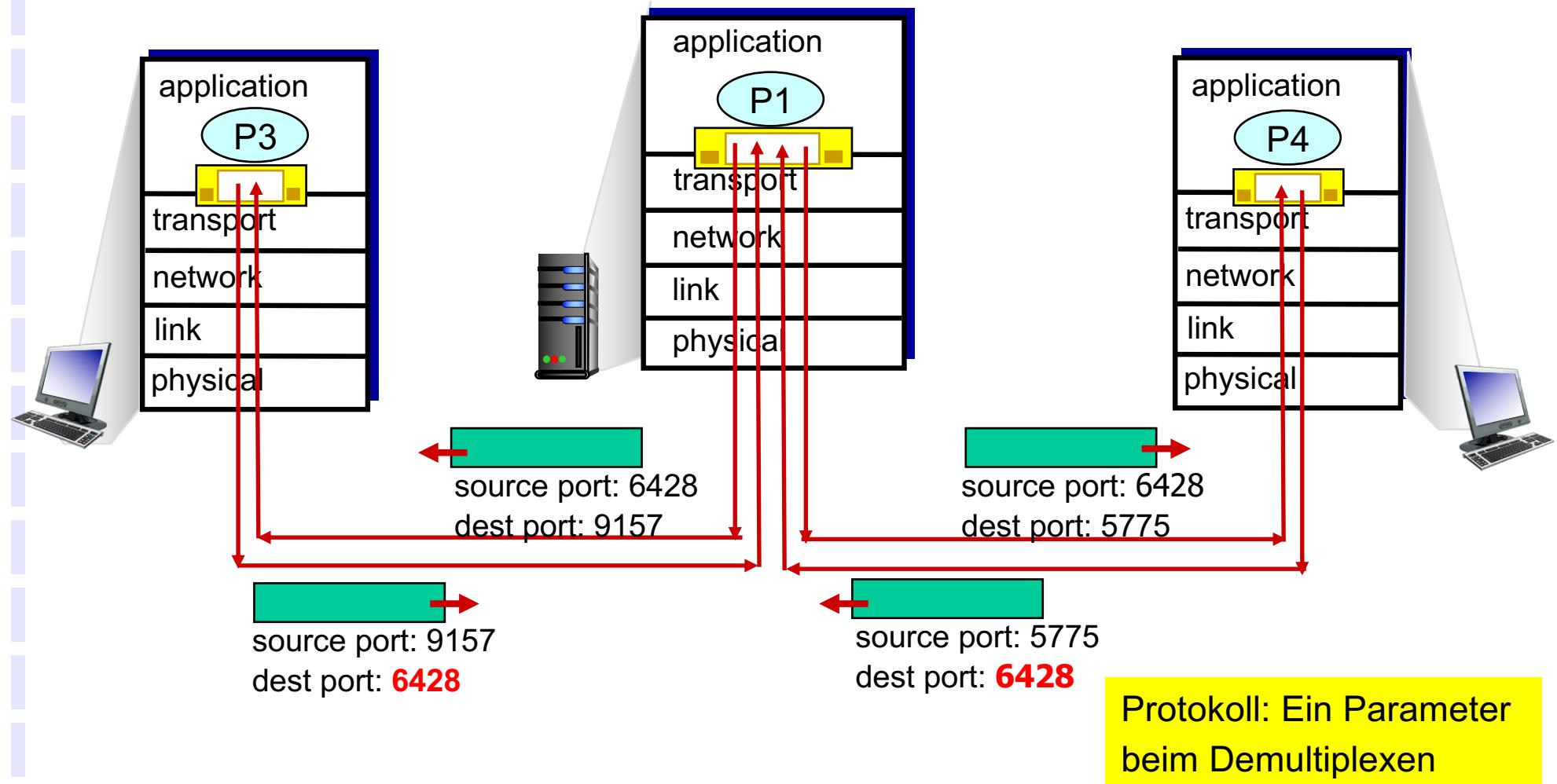
- Kleine Wiederholung:
 - Erzeugung eines UDP Sockets: `sock = DatagramSocket(UDP_Port)`
 - Schickt man ein UDP Segment an ein UDP Socket, dann muss man Ziel-IP-Adr und Ziel-Port angegeben.
- UDP-Sockets werden eindeutig durch (IP-Adresse, Portnummer) identifiziert
- Aktionen beim Empfang eines UDP-Segments:
 - Auswertung der Zielportnummer im UDP-Segment
 - Weiterleitung des UDP-Segments zum Socket mit dieser Portnummer (falls vorhanden)
- Segmente mit unterschiedlicher Quell-IP-Adresse / Quellportnummer werden bei gleicher Zielportnummer an dasselbe **UDP-Socket** weitergeleitet
- Wozu dient bei UDP die Quellportnummer?

Beispiel: Verbindungsloses Demultiplexen (UDP)

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

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

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

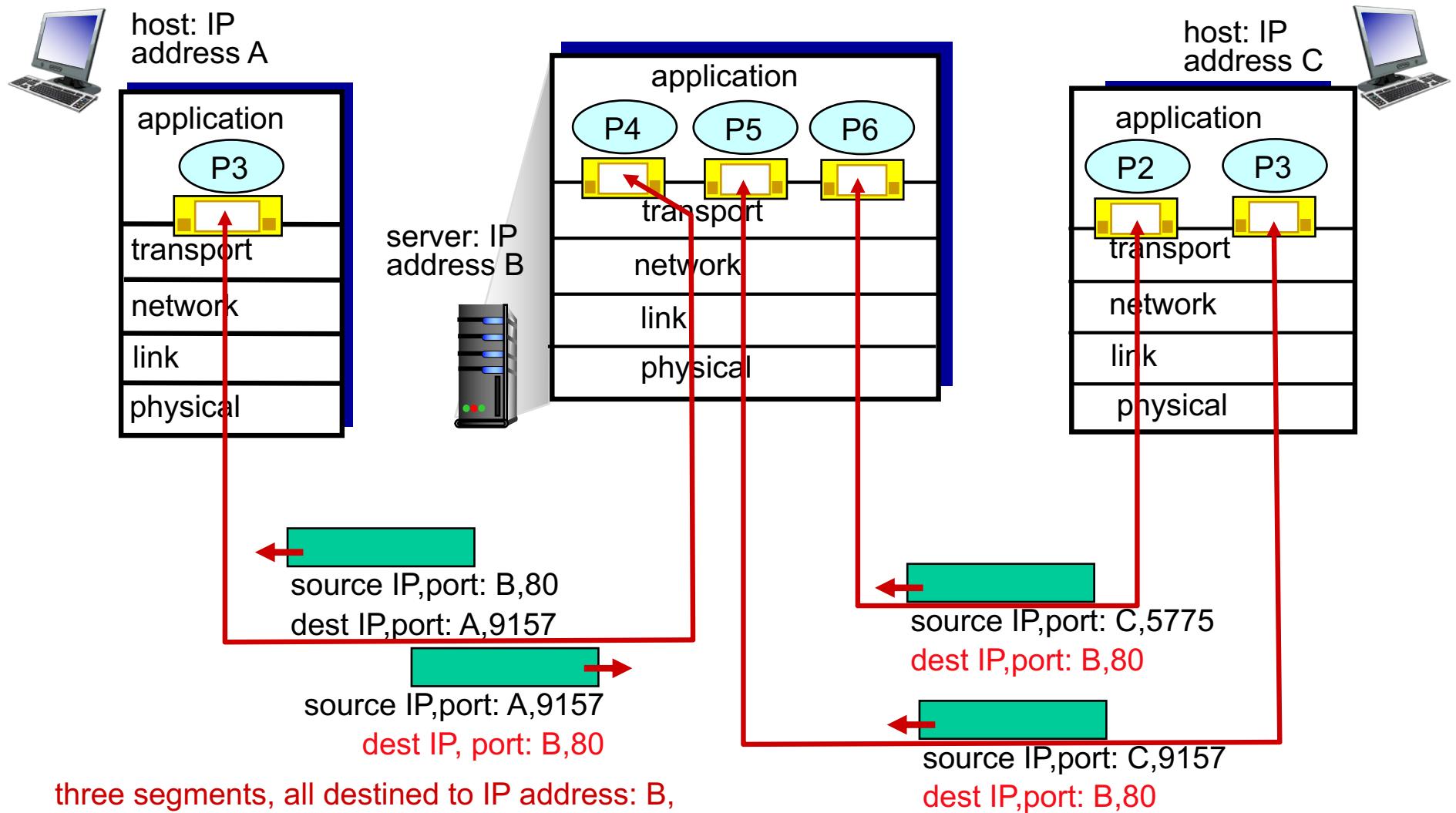


Verbindungsorientiertes Demultiplexen (TCP)

- TCP-Verbindungen werden eindeutig identifiziert durch
(Quell-IP-Adresse, Quellportnummer, Ziel-IP-Adresse, Zielportnummer)
- Aktionen beim Empfang eines TCP-Segments:
 - Falls Verbindungsanfrage → Weiterleitung an ServerSocket für die Zielportnummer (falls vorhanden)
 - Sonst: Auswertung der Quell-IP-Adresse im IP-Datagramm, Quellportnummer und Zielportnummer im TCP-Segment und Hinzufügen der eigenen IP-Adresse
 - Weiterleitung der TCP-Segmentdaten zum Socket mit dieser Identifikation (falls vorhanden)
- **Jedes TCP-Socket repräsentiert eine Verbindung**
 - mehrere verschiedene Verbindungen für eine Zielportnummer können gleichzeitig existieren!

Protokoll: Parameter
beim Demultiplexen

Verbindungsorientiertes Demultiplexen (TCP)



three segments, all destined to IP address: B,
dest port: 80 are demultiplexed to different sockets

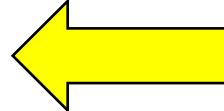
TCP-/UDP-Portnummern

- Welche Portnummern (0 .. 65535) darf eine Anwendung verwenden?
 - Vermeidung von Konflikten!
 - Portnummernbereiche, definiert durch die IANA (Internet Assigned Numbers Authority)
 - **Well Known Ports:** 0 -1023 Well Known ports MÜSSEN bei der IANA registriert werden.
 - **Registered Ports:** 1024 - 49151 Registered ports SOLLTEN bei der IANA registriert werden.
 - **Dynamic and/or Private Ports:** 49152 – 65535
 - Bekannte Ports:

20+21: FTP	23: Telnet	25: SMTP	53: DNS
80: HTTP	110: POP3	443: HTTPS	...
- <http://www.iana.org/assignments/port-numbers>

Kapitel 4: Transportschicht

Gliederung

- Dienste und Prinzipien auf der Transportschicht
- Multiplexen und Demultiplexen von Anwendungen
- Verbindungsloser Transport: UDP 
- Prinzipien des zuverlässigen Datentransfers
- Verbindungsorientierter Transport: TCP
- TCP – Überlastkontrolle (Staukontrolle)
- Zusammenfassung

Textbuch zu diesem Kapitel: J. Kurose & K. Ross: Computernetzwerke – Der Top-Down-Ansatz, Kapitel 3

Folien und Abbildung teilweise aus:

J. Kurose & K. Ross: Computernetzwerke – Der Top-Down-Ansatz

UDP: User Datagram Protocol [RFC 768]

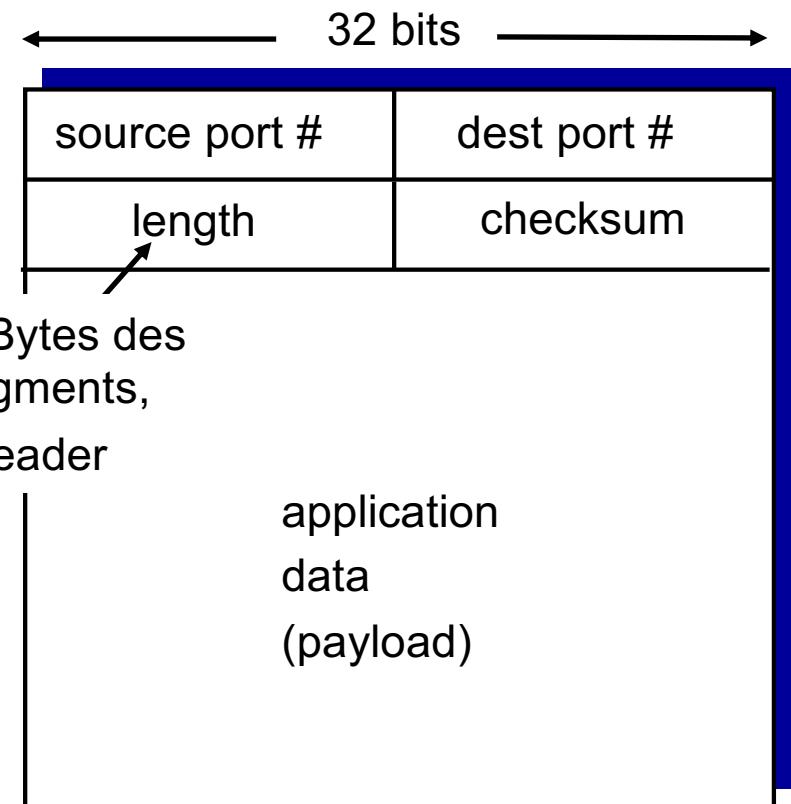
- “Nacktes” Internet Transport Protokoll
- “Best effort” Dienst, UDP Segmente können:
 - verloren gehen
 - in falscher Reihenfolge
 - oder doppelt geliefert werden
- Verbindungslos:
 - Kein “handshaking” zwischen UDP-Sender und Empfänger
 - Jedes UDP-Segment wird unabhängig von anderen transportiert

Warum gibt es UDP?

- Kein Verbindungsaufbau (der Verzögerung bedeutet)
- Einfach: kein Verbindungszustand bei Sender und Empfänger
- Kleiner Segment-Header
- Keine Fluß- und Überlastkontrolle: UDP kann so schnell es geht senden

UPD (Fortsetzung)

- Oft für Multimedia-Anwendungen verwendet
 - Tolerant gegenüber Paketverlust
 - Datenrate ist wichtiger
- Wozu sonst wird UDP genutzt:
 - DNS
 - SNMP (Netzwerk-Management)
- Verlässlicher Datentransfer mit UDP: Die Anwendung muss das leisten!
 - Anwendungsspezifische Fehlerentdeckung und -korrektur und Überlastkontrolle
 - Beispiel: QUIC-Protokoll (*Google*)
<https://en.wikipedia.org/wiki/QUIC>



UDP segment format

UDP-Prüfsumme

- **Ziel:** Fehlerentdeckung im übertragenen Segment (z.B. falsche Bits)
- **Warum ist eine Prüfsumme sinnvoll?**
- **Funktionsweise:**

Absender:

- Betrachte den Segmentinhalt als Sequenz von 16-bit Integerzahlen
- Prüfsumme: Addition des Segmentinhalts
- Sender speichert Einerkomplement der Prüfsumme in das entspr. Datenfeld

Empfänger:

- Berechne Prüfsumme des empfangenen Segments
- Überprüfe, ob berechnete Summe der übertragenen entspricht:
 - NEIN – Fehler gefunden
 - JA – kein Fehler gefunden.
 - Werden alle Fehler erkannt?

UDP-Prüfsumme: Beispiel (vereinfacht)

Sender

Drei 16-bit-Blöcke: ...0110,
...0101,
...1111

1. Addition der beiden ersten:

$$\begin{array}{r} \dots0110 \\ \dots0101 \\ \hline \dots1011 \end{array}$$

2. Addition des dritten:

$$\begin{array}{r} \dots1011 \\ \dots1111 \\ \hline \dots1010 \end{array}$$

3. Einerkomplement-Bildung:

$$\dots1010 \rightarrow \dots0101$$

4. Speichern der Prüfsumme: ...0101

Carry wird durchgeschoben

Empfänger

Drei 16-bit-Blöcke: ...0110,
...0101,
...1111

1. Addition der beiden ersten:

$$\begin{array}{r} \dots0110 \\ \dots0101 \\ \hline \dots1011 \end{array}$$

2. Addition des dritten:

$$\begin{array}{r} \dots1011 \\ \dots1111 \\ \hline \dots1010 \end{array}$$

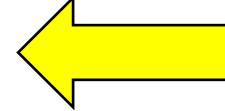
3. Addition der Prüfsumme:

$$\begin{array}{r} \dots1010 \\ \dots0101 \\ \dots1111 \\ \hline \end{array}$$

Wenn Ergebnis != FF → Fehler!

Kapitel 4: Transportschicht

Gliederung

- Dienste und Prinzipien auf der Transportschicht
- Multiplexen und Demultiplexen von Anwendungen
- Verbindungsloser Transport: UDP
- Prinzipien des zuverlässigen Datentransfers 
- Verbindungsorientierter Transport: TCP
- TCP – Überlastkontrolle (Staukontrolle)
- Zusammenfassung

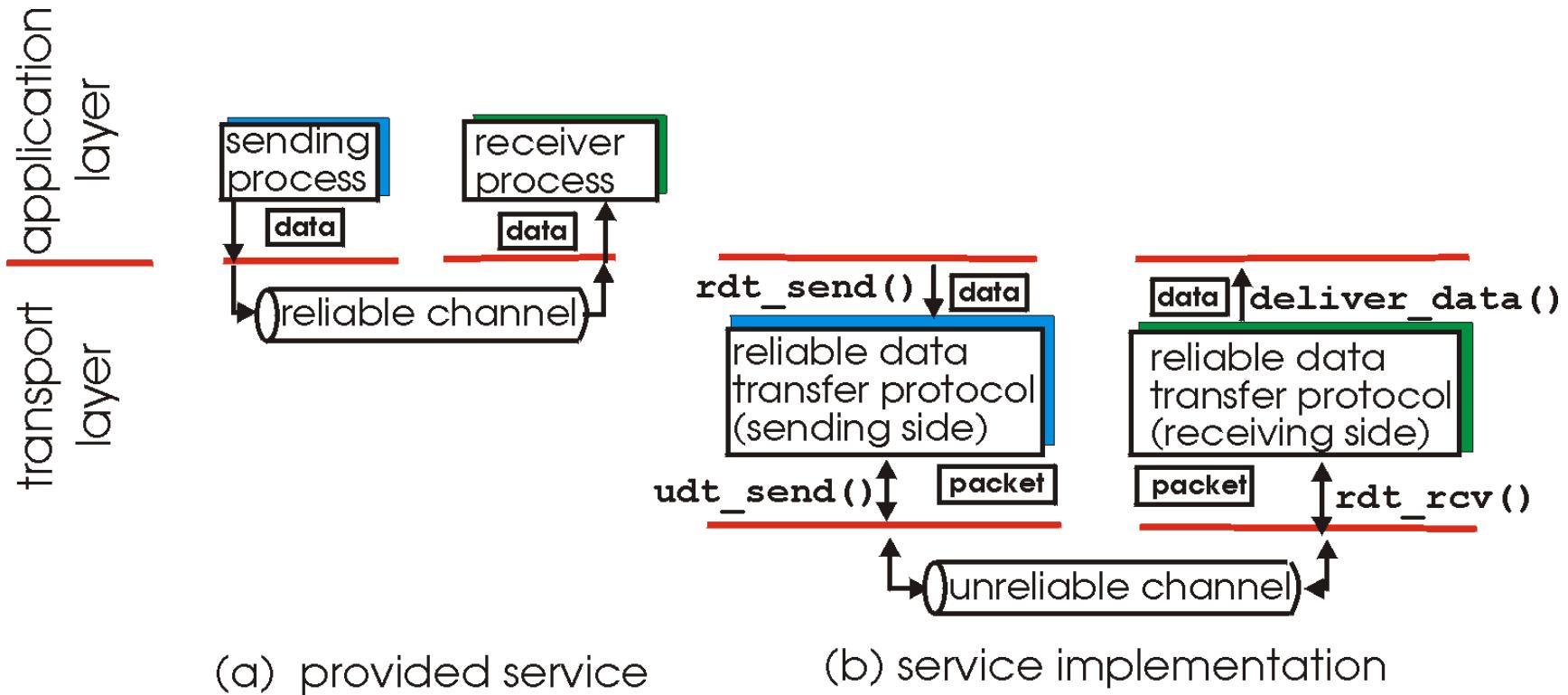
Textbuch zu diesem Kapitel: J. Kurose & K. Ross: Computernetzwerke – Der Top-Down-Ansatz, Kapitel 3

Folien und Abbildung teilweise aus:

J. Kurose & K. Ross: Computernetzwerke – Der Top-Down-Ansatz

Prinzipien des zuverlässigen Datentransfers

- Wichtig für Anwendungs-, Transport- und Sicherungsschicht
- Wichtige Funktionalität in Rechnernetzen!

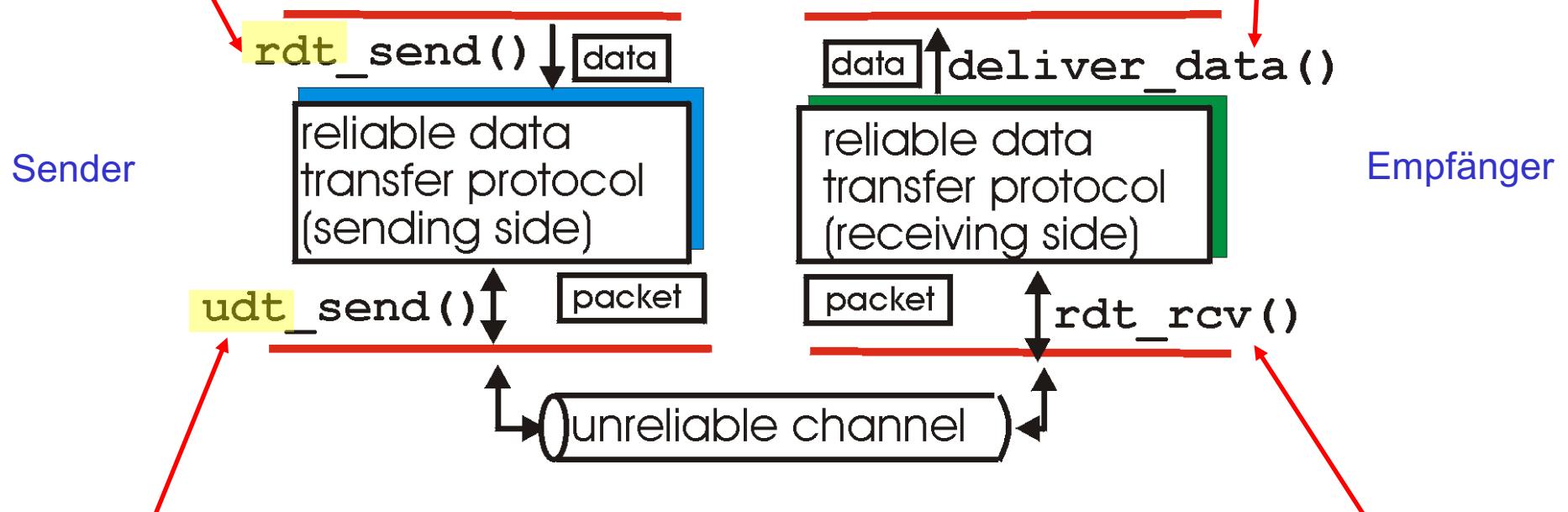


- Die Eigenschaften des unzuverlässigen Kanals bestimmen die Komplexität des zuverlässigen Kommunikationsprotokolls

Zuverlässiger Datentransfer: Grundmodell

rdt_send(..): von oben aufgerufen, (d.h. von Anwendung).
Ziel: Übergabe an Empfänger

deliver_data(..): aufgerufen von rdt_rcv, um Daten bei der Anwendung abzuliefern



udt_send(..): aufgerufen von rdt_send, um Daten über unzuverl. Kanal zum Empf. zu transportieren

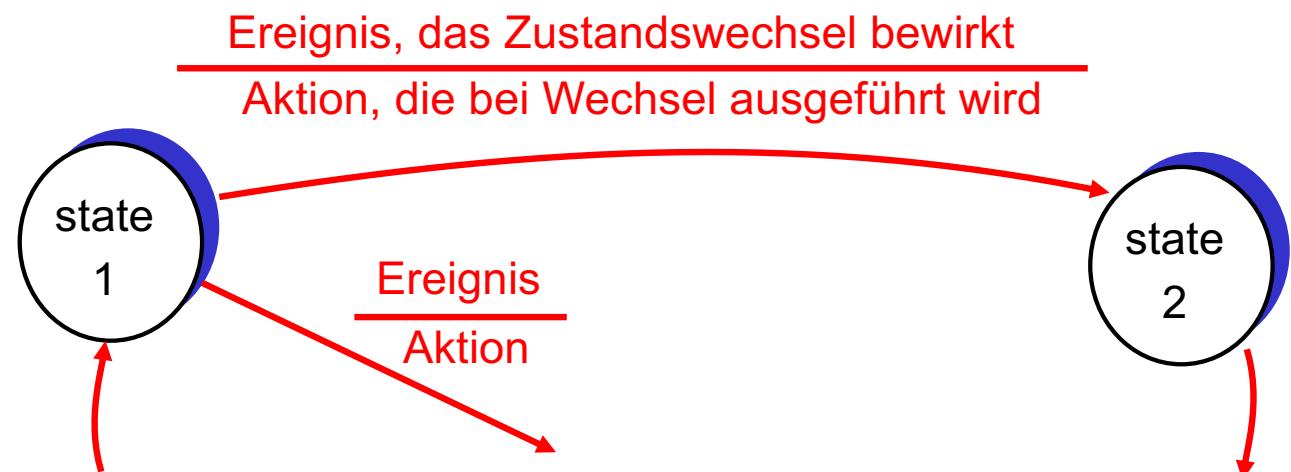
rdt_rcv(..): aufgerufen, wenn Paket auf der Empf.-Seite des Kanals ankommt

Weiteres Vorgehen

Wir werden ..

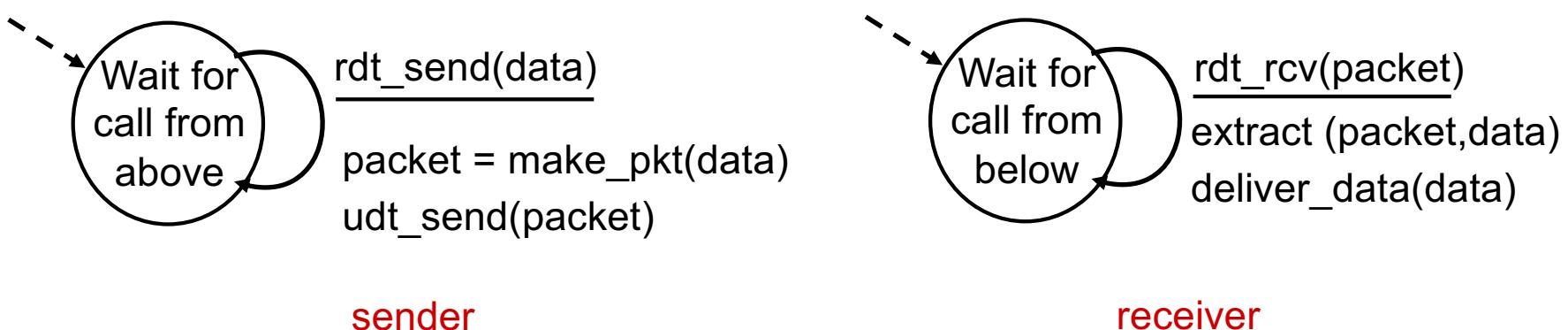
- das Protokoll des zuverlässigen Datentransfers (**rdt**) schrittweise entwickeln
- nur unidirektionalen Datentransfer betrachten
 - aber Steuerinformationen fließen in beide Richtungen!
- einen Finite State Machine (FSM) benutzen, um Sender und Empfänger zu spezifizieren

Zustand: wenn in einem Zustand, wird Übergang eindeutig bestimmt durch nächstes Ereignis



rdt 0.1: zuverlässiger Transfer über zuverlässigen Kanal

- Kanal ist perfekt zuverlässig
 - keine Bitfehler
 - kein Verlust von Paketen
 - Reihenfolge bleibt erhalten
- Getrennte Automaten für Sender und Empfänger:
 - Sender senden in darunterliegenden Kanal
 - Empfänger liest Daten vom darunterliegenden Kanal



rdt 0.2: Kanal mit Bitfehlern

Eigenschaften des unzuverlässigen Kanals: verfälscht Bits des Pakets

- aber ohne Totalverlust eines Pakets
- Die Reihenfolge der Pakete bleibt erhalten

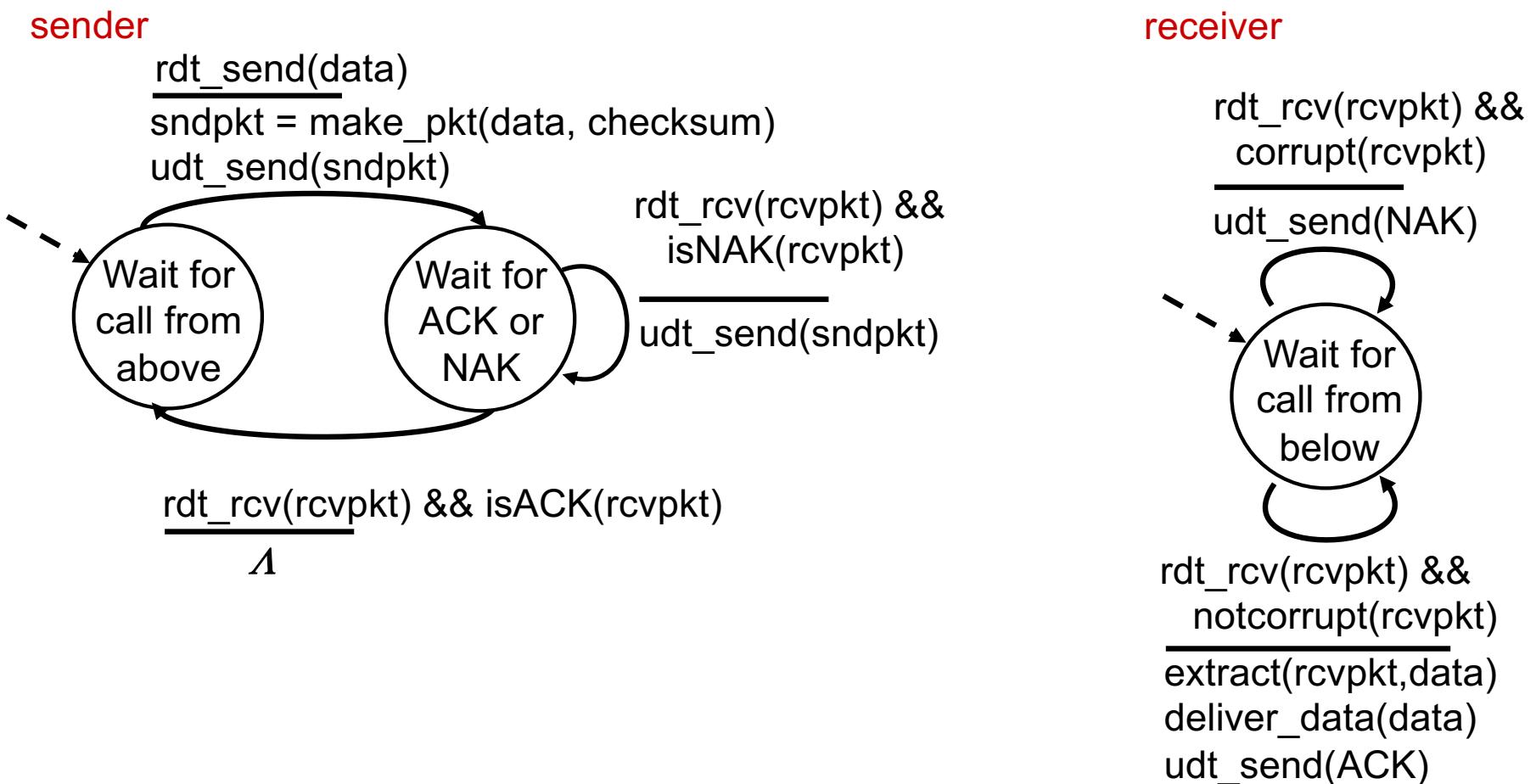
Ansatz: ARQ-Protokoll (Automatic Repeat reQuest)

- Empfänger muss erkennen können, dass ein Fehler vorliegt.
- Quittungen (acknowledgements - ACKs): Empfänger teilt Sender explizit mit, dass das empfangene Paket OK war.
- Negative Quittungen (NAKs): Empfänger teilt Sender mit, dass ein Paket Fehler hatte
 - Fehlerfall: Sender wiederholt Übertragung nach Empfang von NAK

Neue Mechanismen rdt 0.2 :

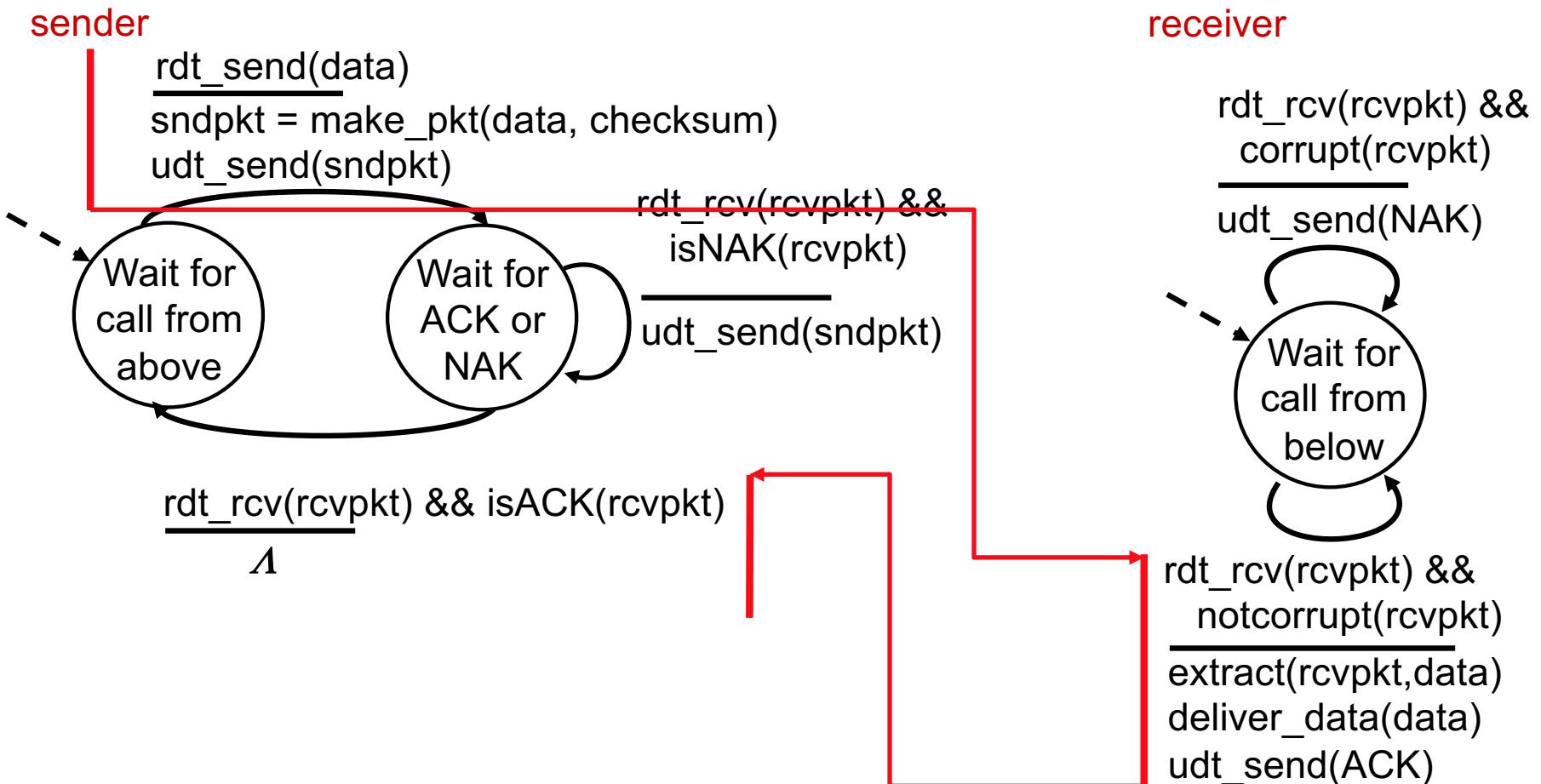
- Fehlerentdeckung
- Empfänger-Rückmeldung: Kontroll-Nachrichten (ACK, NAK) vom Empfänger an den Sender

rdt 0.2: FSM-Spezifikation (1. Versuch)



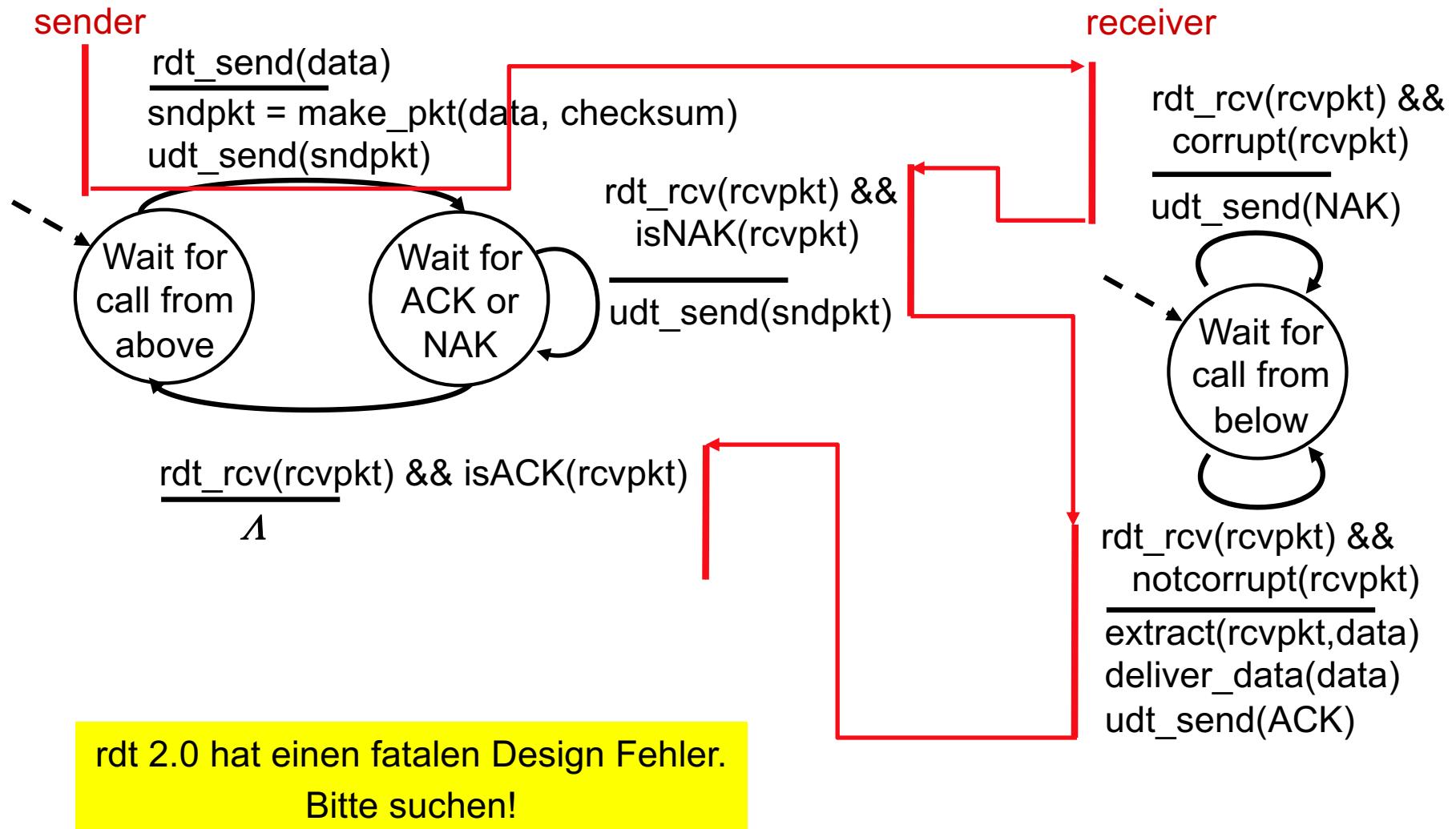
rdt 0.2: FSM-Spezifikation (1. Versuch)

In Aktion – ohne Fehler



rdt 0.2: FSM-Spezifikation (1. Versuch)

In Aktion – Fehlerszenario



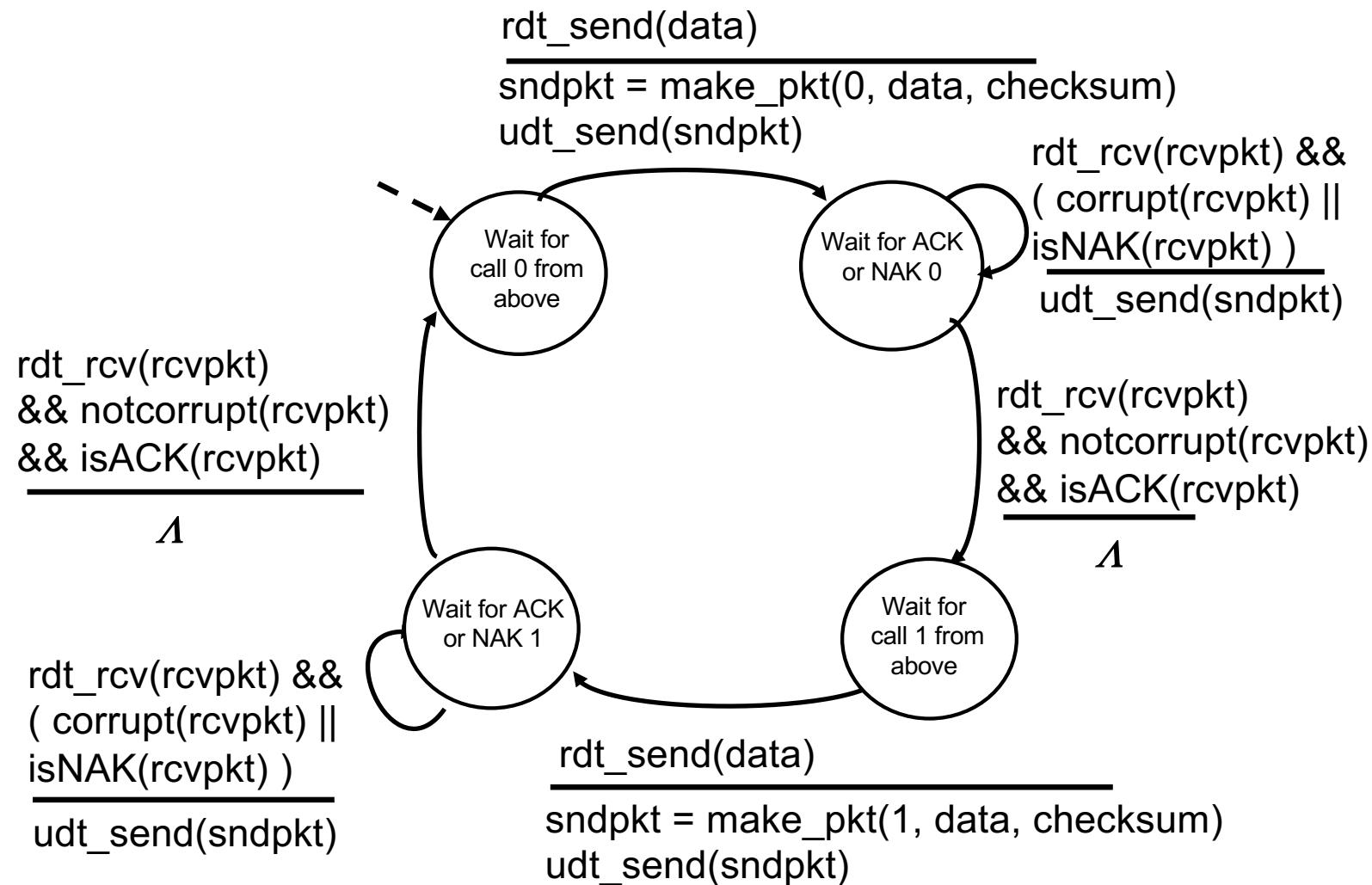
rdt 0.2 hat einen fatalen Design-Fehler!

- Was passiert, wenn ACK / NAK verfälscht sind?
- Sender weiß nicht, was beim Empfänger passiert ist!
- Was tun? Soll der Sender ...
- die ACK/NAK des Empfängers quittieren? Was passiert aber, wenn die Sender-Quittungen (ACK/NAK) verloren gehen?
- das Paket einfach wiederholen? Aber dann muss das Wiederholung eines korrekt übertragenen Pakets (Duplikat) erkannt werden.
- Duplike erkennen!
- Sender fügt Sequenznummer bei jedem Paket hinzu
- Sender wiederholt aktuelles Paket, wenn ACK/NAK verfälscht ist
- Empfänger verwirft Duplike (liefert sie nicht bei der Anwendung ab)

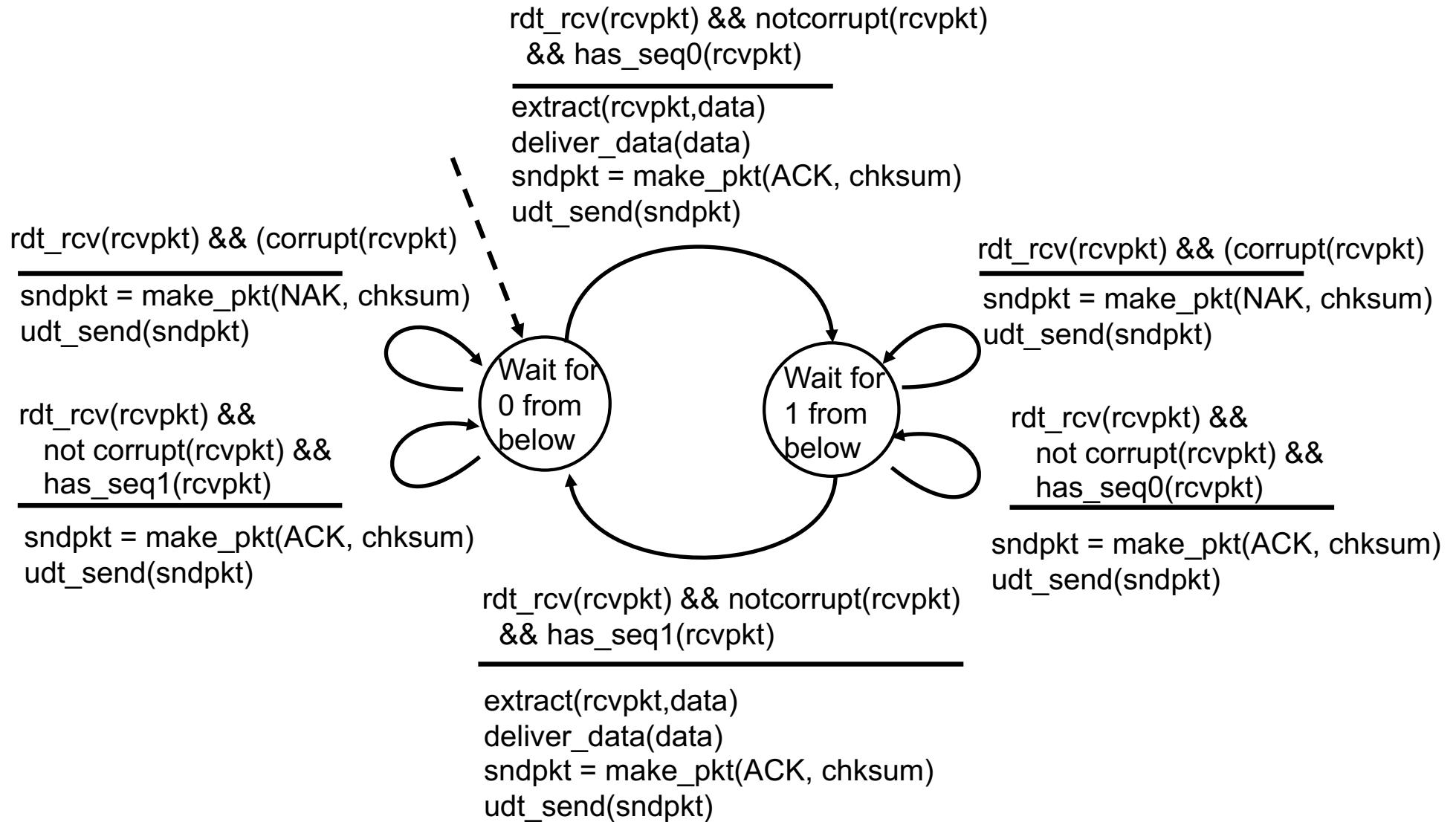
stop and wait Protokoll

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

rdt 0.2.1: Sender (behandelt verfälschte ACK/NAKs)



rdt 0.2.1: Empfänger (behandelt verfälschte ACK/NAKs)



rdt 0.2.1: Diskussion

Sender:

- ... fügt Sequenznummer zum Paket hinzu (2 Sequenz-nummern 0/1 reichen aus)
- ... muss prüfen, ob empfangenes ACK/NAK verfälscht ist
- ... hat doppelt so viele Zustände, denn ein Zustand muss erinnern, ob aktuelles Paket die Sequenznummer 0 oder 1 hat

Empfänger:

- ... muss prüfen, ob empfangenes Paket ein Duplikat ist
- Zustand kennzeichnet, ob die Sequenznummer 0 or 1 erwartet wird
- Bemerkung: Der Empfänger kann nicht wissen, ob das letzte ACK/NAK beim Sender richtig empfangen wurde

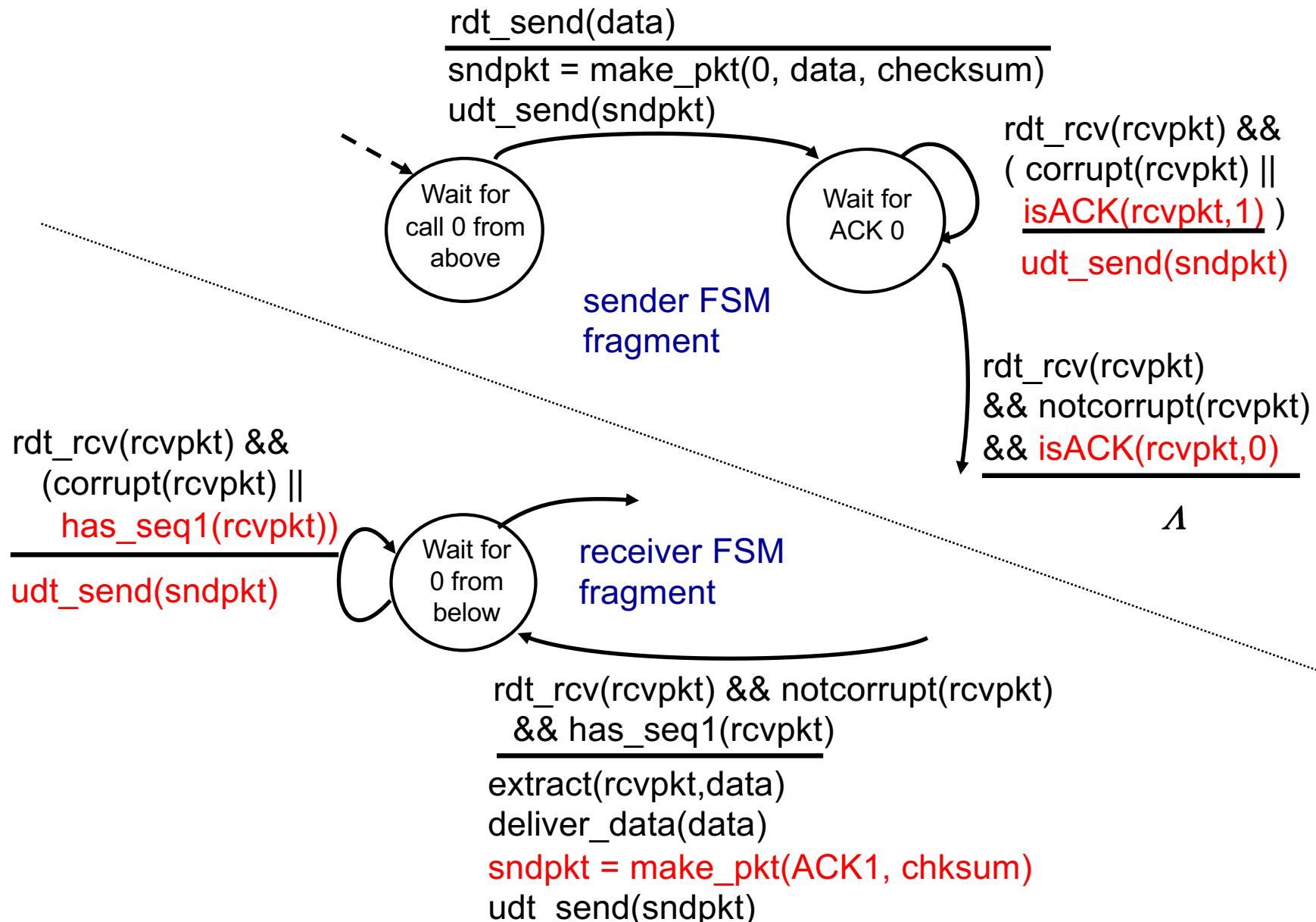
rdt 0.2.2: ein NAK-freies Protokoll

Ansatz:

- dieselbe Funktionalität wie rdt0.2.1, nur mit ACKs
- Empfänger sendet ACK für das letzte richtig empfangene Paket (anstelle eines NAK)
- Empfänger muss explizit die Sequenznr. des quittierten Pakets in das ACK einfügen
- Duplikate eines ACK beim Sender führen zur selben Aktion wie das NAK: erneutes Versenden des aktuellen Pakets

Bitte passen die die FSMs aus rdt 0.2.1 entsprechend an.

rdt 0.2.2: FSMs



rdt 0.3: Kanal mit Fehlern und Paketverlust

Neue Annahme:

- Kanal kann zusätzlich Pakete verlieren (Daten oder ACKs)
- Prüfsumme, Sequenznummern, ACKs, Wiederholungen helfen, sind aber nicht genug

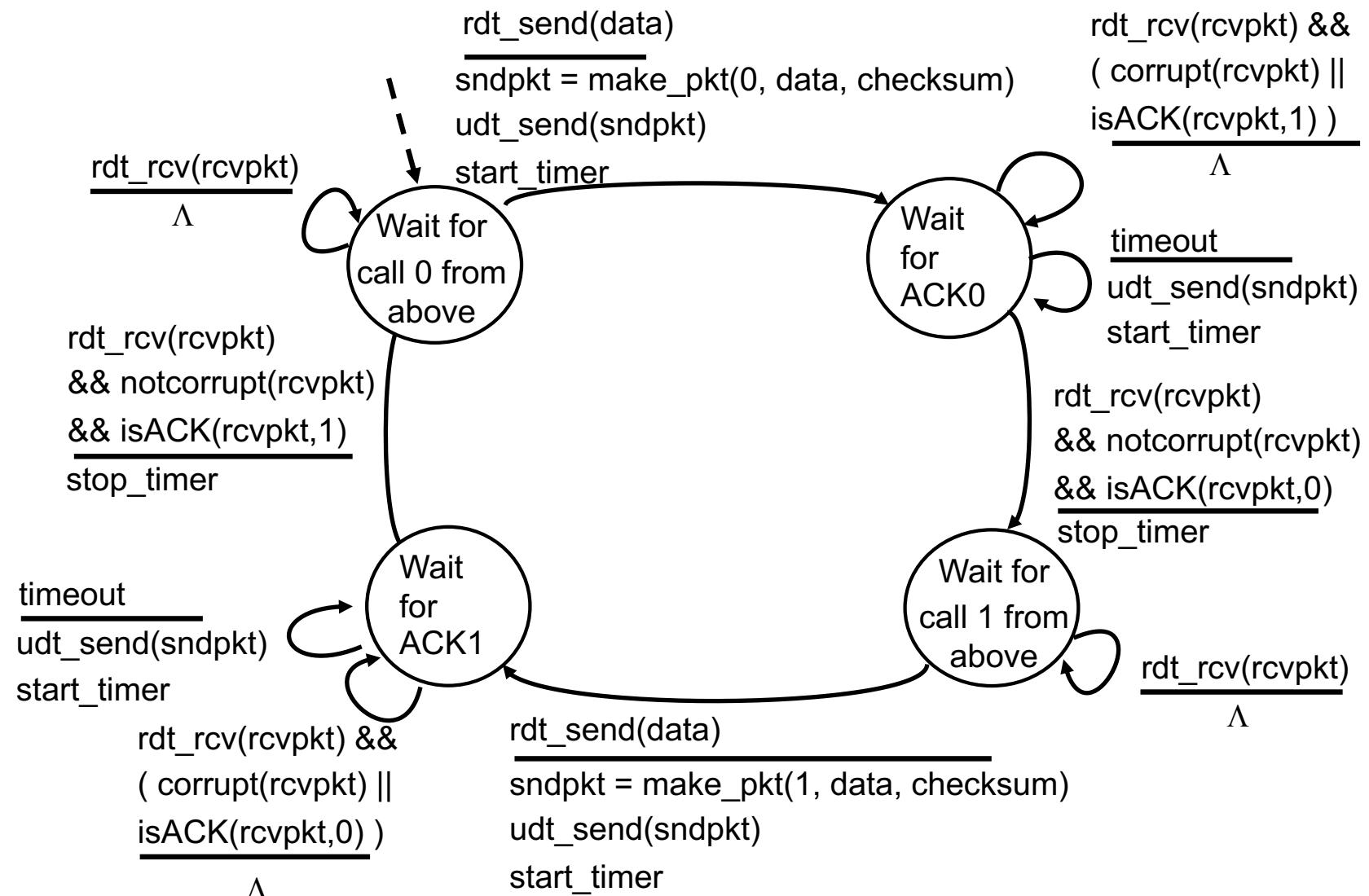
Frage:

- Wie erkennen und behandeln wir Verluste ?

Lösungsversuch:

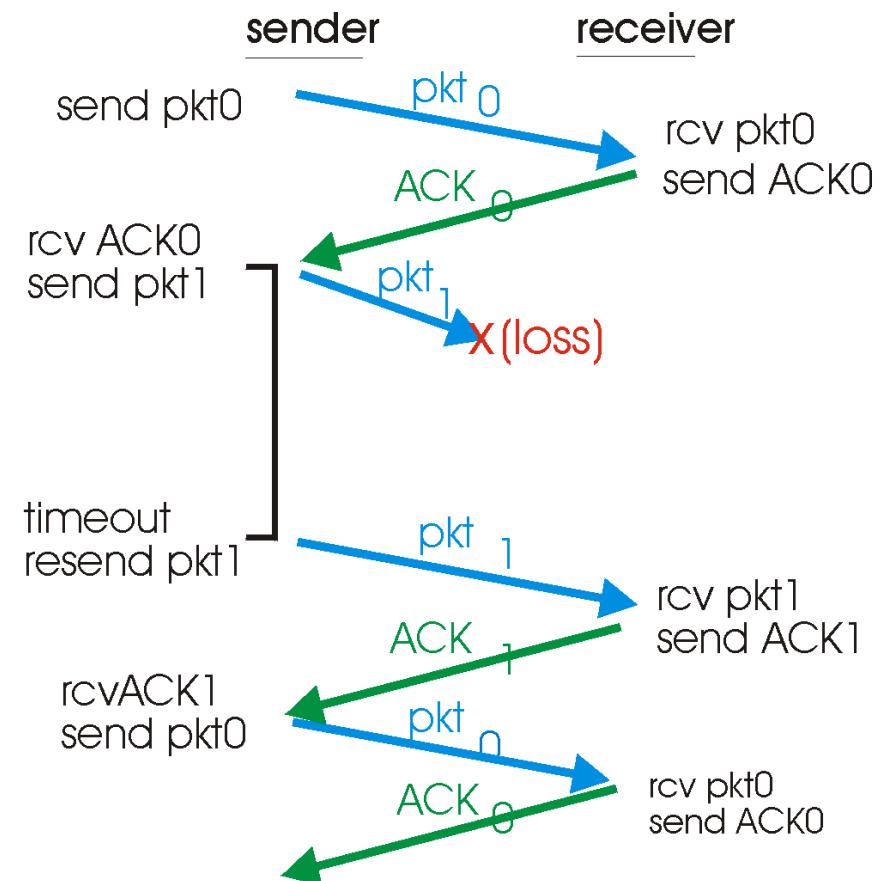
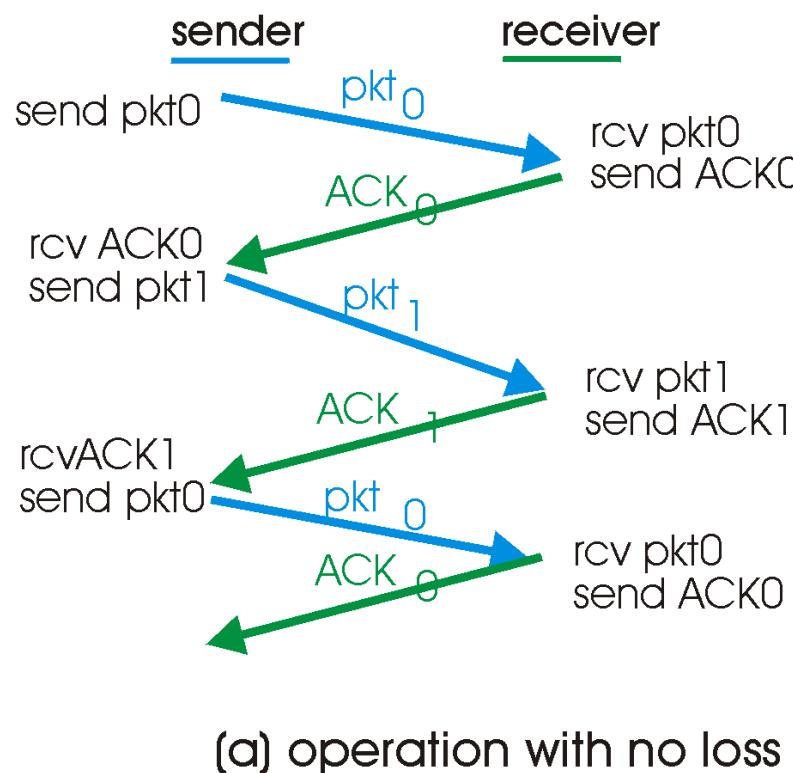
- Sender wartet eine “vernünftige” Zeitdauer auf das ACK
- Erfordert Timer (countdown)
- Sender wiederholt Übertragung, wenn ACK nicht innerhalb dieses Zeitintervalls empfangen wurde
- Wenn Paket (oder ACK) verzögert wurde (kein Verlust): Wiederholung produziert ein Duplikat, aber mithilfe der Sequenznummer kann das erkannt werden

rdt 0.3: Sender

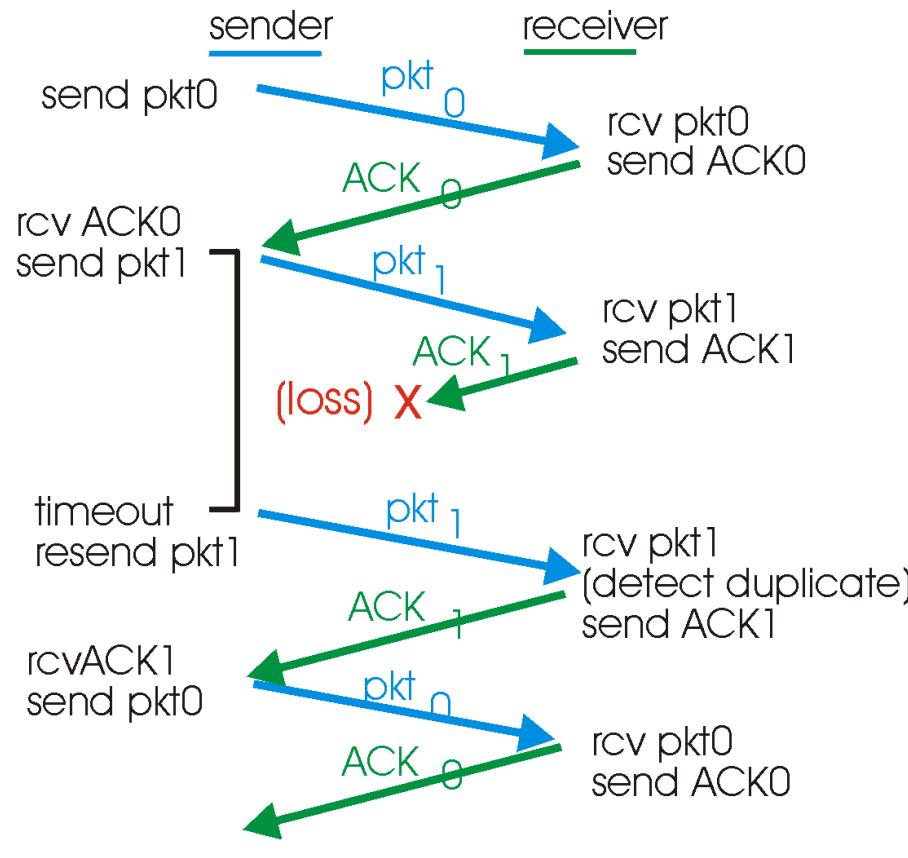


Was muss am Empfänger geändert werden?

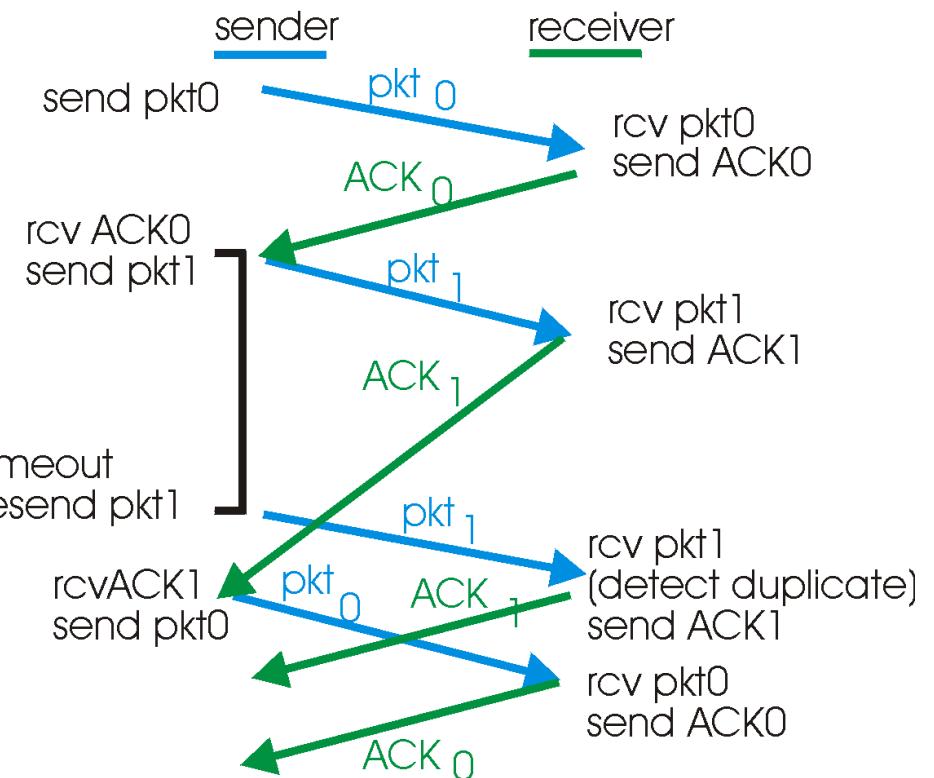
rdt 0.3: In Aktion



rdt 0.3: In Aktion



(c) lost ACK



(d) premature timeout

Performanz von rdt 0.3

- **rdt 0.3 funktioniert, aber die Performance ist schlecht!!**
- **Beispiel:** R = 1 Gbit/s Übertragungsrate, L = 1 KB Paketlänge

$$T_{\text{transmit}} = \frac{L}{R} = \frac{8000 \text{ bit}}{10^{12} \text{ bit/s}} = 0,000008 \text{ s} = 0,008 \text{ ms}$$

- Die Kommunikationspartner stehen an der Ost- und der Westküste der USA. Das ergibt eine Ausbreitungsverzögerung $T_{\text{prop}} = 15 \text{ ms}$.
- Die Quittung liegt nach der RTT vor.

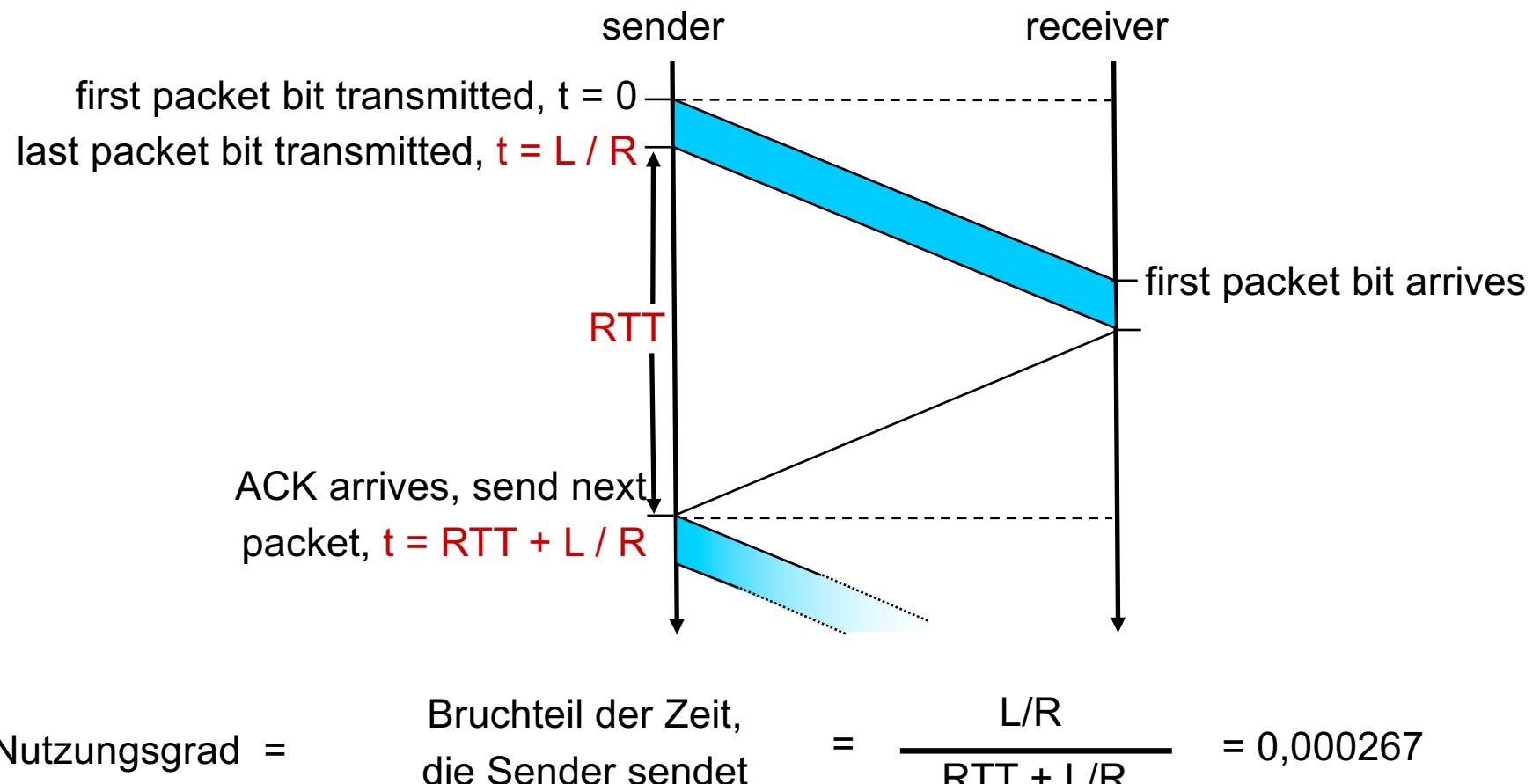
$$\text{RTT} = 15 \text{ ms} + 15 \text{ ms} = 30 \text{ ms} \quad (\text{ACK-Sendeverzögerung etc. hier vernachlässigbar})$$

$$\text{Nutzungsgrad} = \frac{\text{Bruchteil der Zeit, die Sender sendet}}{30,008 \text{ ms}} = \frac{0,008 \text{ ms}}{30,008 \text{ ms}} = 0,000267$$

- 1 KB pro 30 ms → 33KB/s Durchsatz über eine 1 Gbit/s Leitung!
- **Das Netzwerk-Protokoll begrenzt die Nutzung der physikalischen Ressourcen!**

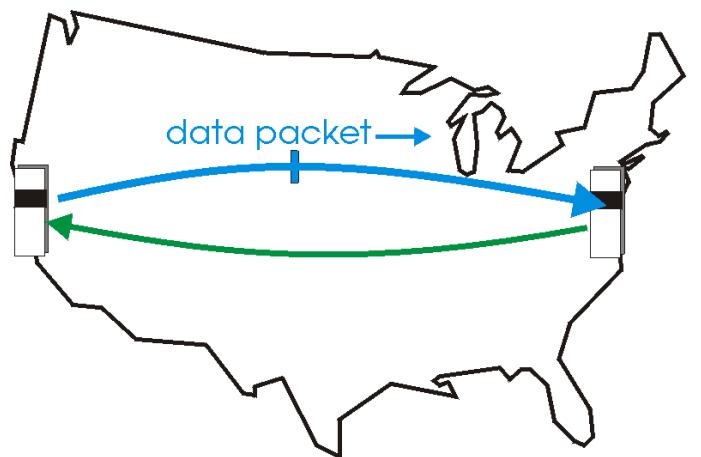
Grund für dieses Performance Problem

Stop and Wait Protokoll

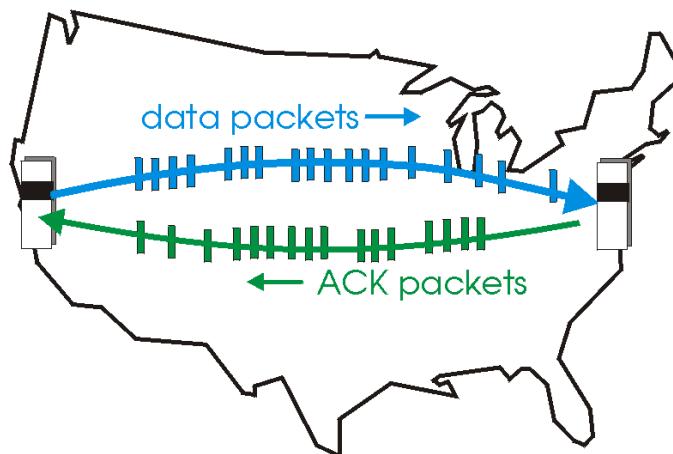


Pipeline-Protokolle

- **Pipelining:** Sender erlaubt, dass **mehrere** Pakete noch zu bestätigen, d.h. "unterwegs" sind
 - Sequenznummernbereich muss vergrößert werden
 - Puffer beim Sender und / oder Empfänger erforderlich



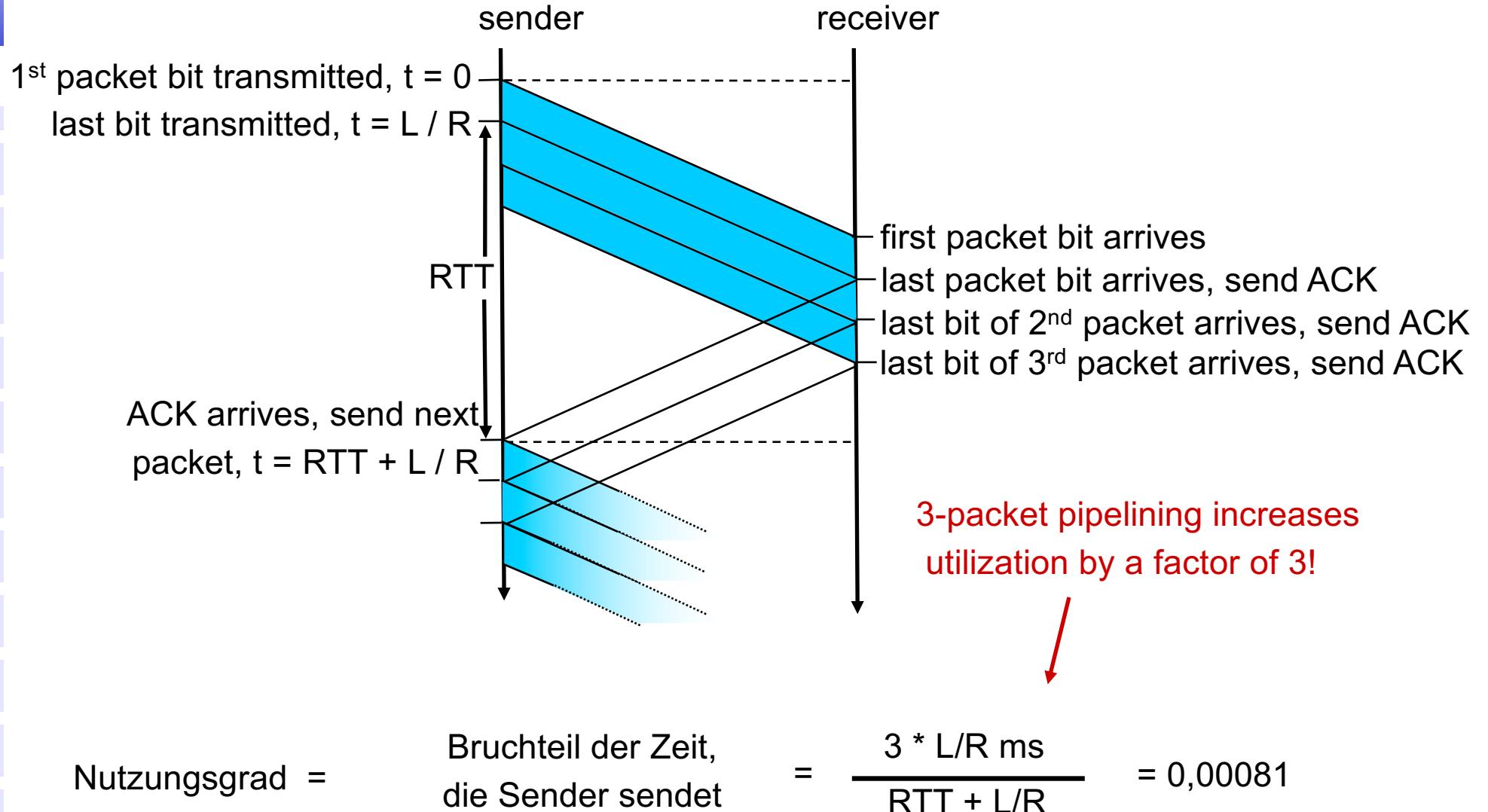
(a) a stop-and-wait protocol in operation



(b) a pipelined protocol in operation

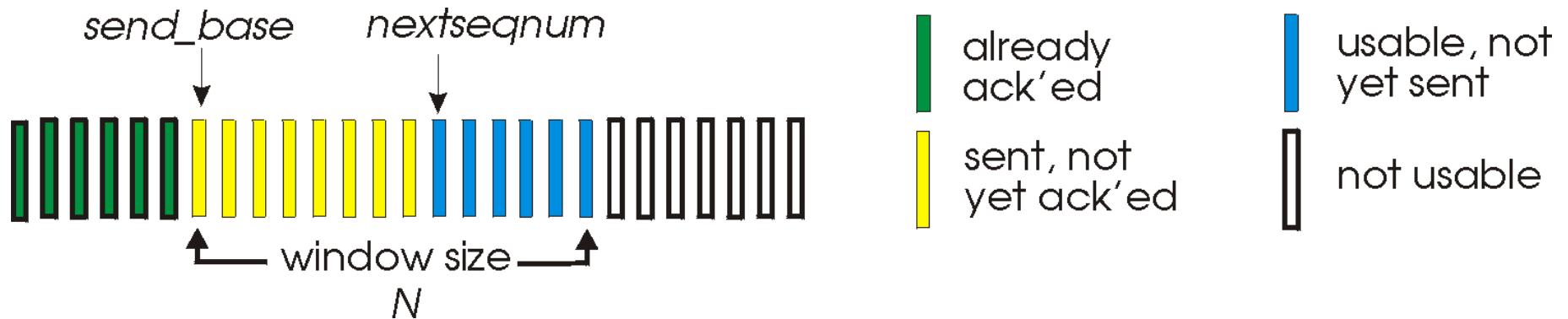
- Es gibt 2 grundsätzliche Arten von Pipeline-Protokollen:
 - *Go-Back-N*
 - *Selective Repeat*

Pipelining erhöht den Nutzunggrad



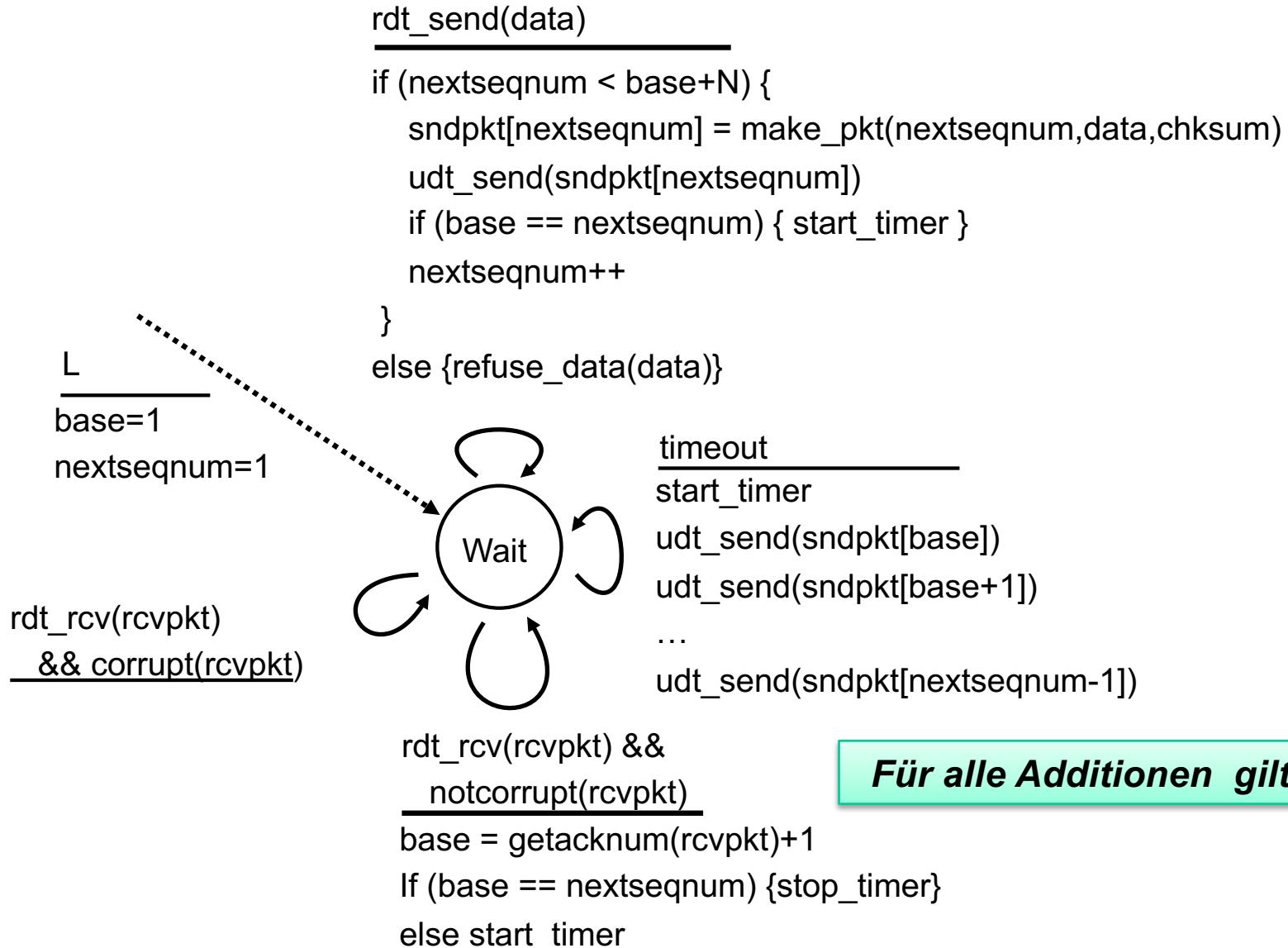
Go-Back-N Protokoll

- k-bit Sequenznummer im Paket-Header
- Fenster ("window") von bis zu N aufeinanderfolgend nicht bestätigten Paketen erlaubt

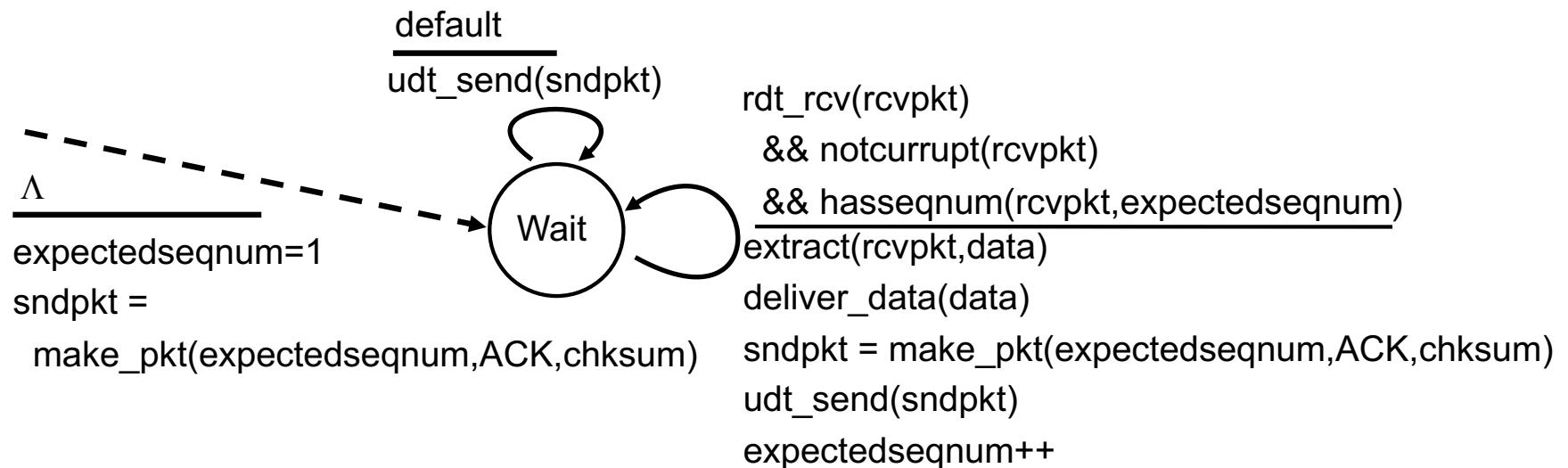


- ACK(n): bestätigt alle Pakete bis einschließlich Sequenznummer n - **Kumulatives ACK**
- Timer für das älteste nicht bestätigte Paket (*send_base* n)
- *timeout(n)*: Sendewiederholung von Paket n und aller Pakete mit höherer Sequenznummer im Fenster

Go-Back-N: FSM des Senders



Go-Back-N: FSM des Empfängers

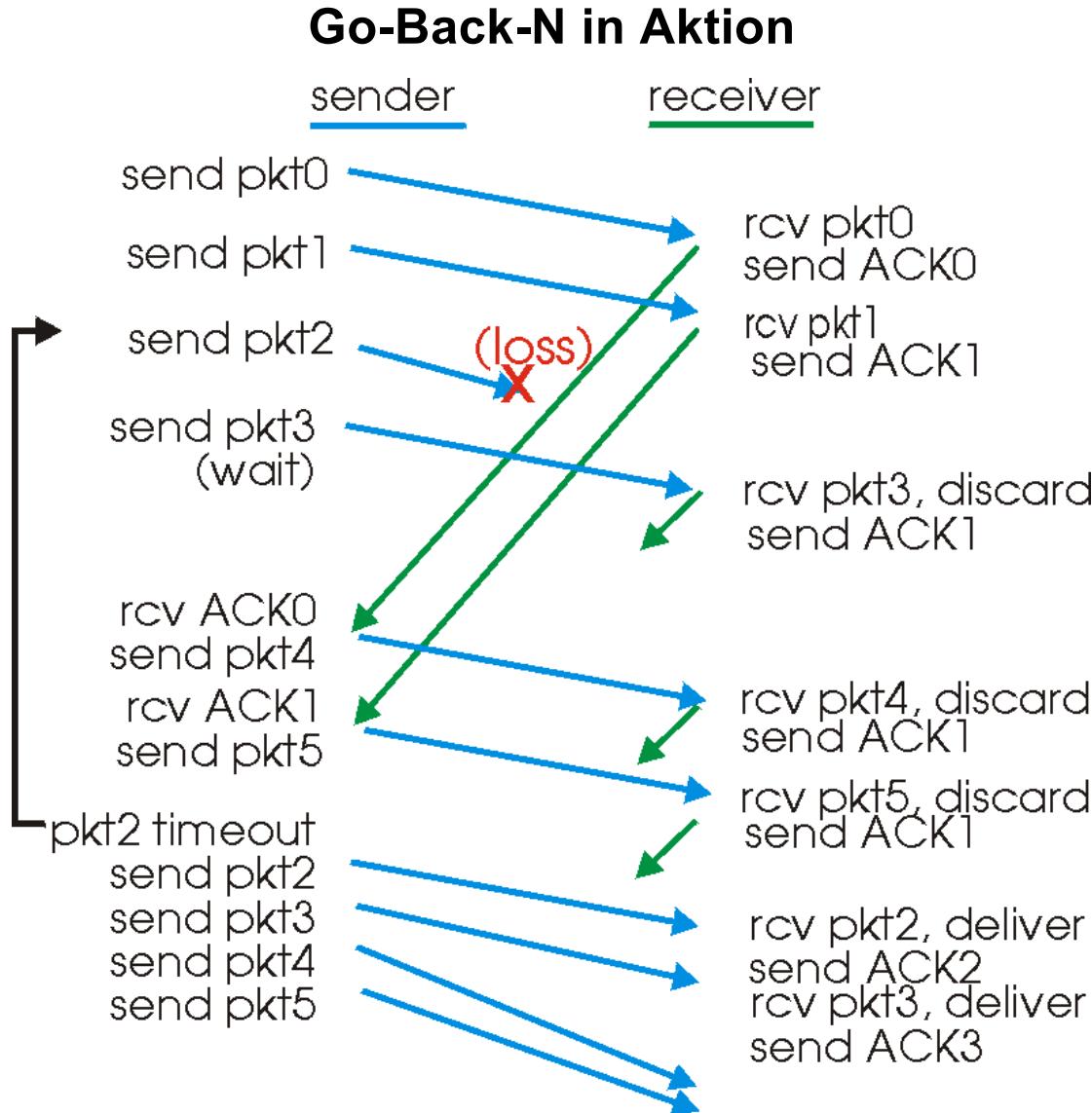


Paket korrekt und innerhalb der Reihenfolge:

- Sende ACK für das empfangene Paket (→ Sequenznummer im ACK)
- Erhöhe **expectedseqnum** (modulo 2^k)

Paket nicht korrekt oder außerhalb der Reihenfolge:

- Verwerfen (nicht puffern) → **kein Puffer auf Seiten des Empfängers!**
- ACK für das Paket mit der höchsten Sequenznummer in richtiger Reihenfolge (letztes korrektes Paket) senden
- Empfänger kann dadurch Duplikat – ACKs produzieren



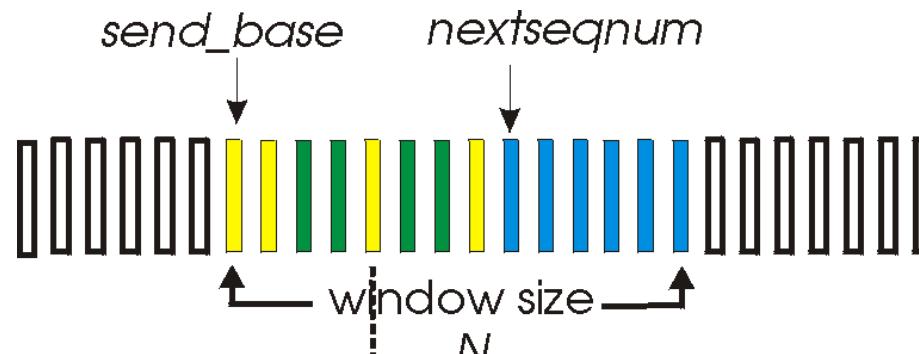
Stellen Sie bitte typische Szenarien mit den Demonstrator
http://www.ccs-labs.org/teaching/rn/animations/gbn_sr/_nach

Was passiert, wenn viele Pakete in der Pipeline sind und ein Paketfehler tritt auf?

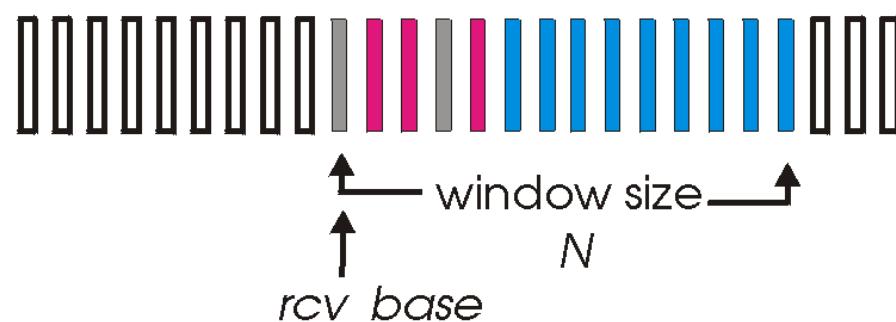
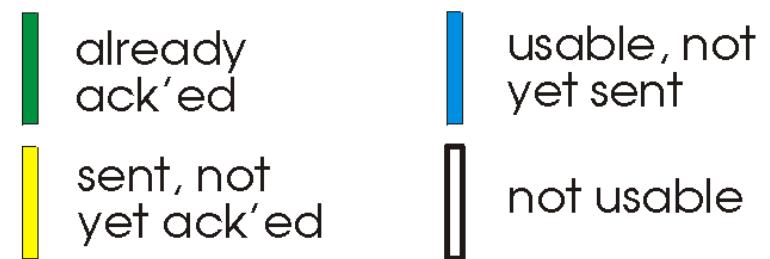
Selective Repeat Protokoll

- Empfänger bestätigt individuell alle korrekt empfangenen Pakete
 - Empfänger puffert Pakete - wenn erforderlich - zwischen, um die Pakete bei der Ablieferung an die höhere Schicht in richtiger und lückenlosen Reihenfolge übergeben zu können (Empfangspuffergröße = Sendepuffergröße)
- Sender wiederholt nur die Pakete, für die er kein ACK erhält
 - Sender braucht Timer für jedes unbestätigte Paket
- Sendefenster
 - N Pakete mit aufeinanderfolgenden Sequenznummern (wieder wird die Anzahl der gesendeten, nicht bestätigten Sequenznummern limitiert)

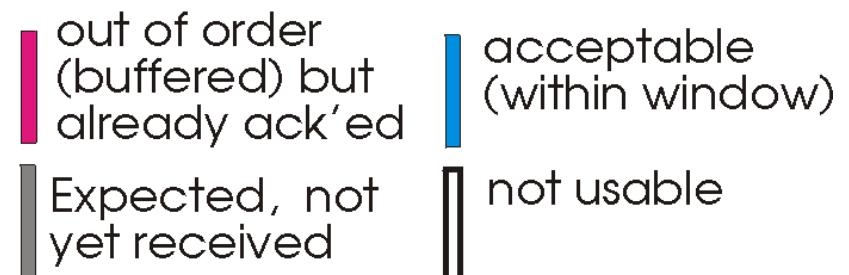
Selective Repeat Protokoll: Sender- & Empfangsfenster



(a) sender view of sequence numbers



(b) receiver view of sequence numbers



Selective Repeat Protokoll

Sender

Daten von “oben” :

- Wenn nächste Sequenznummer im Fenster liegt, sende Paket

timeout(n):

- Wiederhole Senden von Paket n, Timer neu starten

ACK(n) in [sendbase,sendbase+N-1]:

- markiere Paket n als empfangen
- wenn n die kleinste nichtbestätigte Paket- Sequenznummer ist, setze Fensterbasis (send_base) auf nächste nicht bestätigte Sequenznummer

Empfänger

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

- sende ACK(n)
- außerhalb der Reihenfolge: Zwischenspeichern
- in richtiger Reihenfolge: Abliefern (mit allen bisher nicht gelieferten, aber gespeicherten Paketen, die dann in der richtigen Reihenfolge sind).
- Schiebe Fenster auf nächstes nicht empfangenes Paket vor

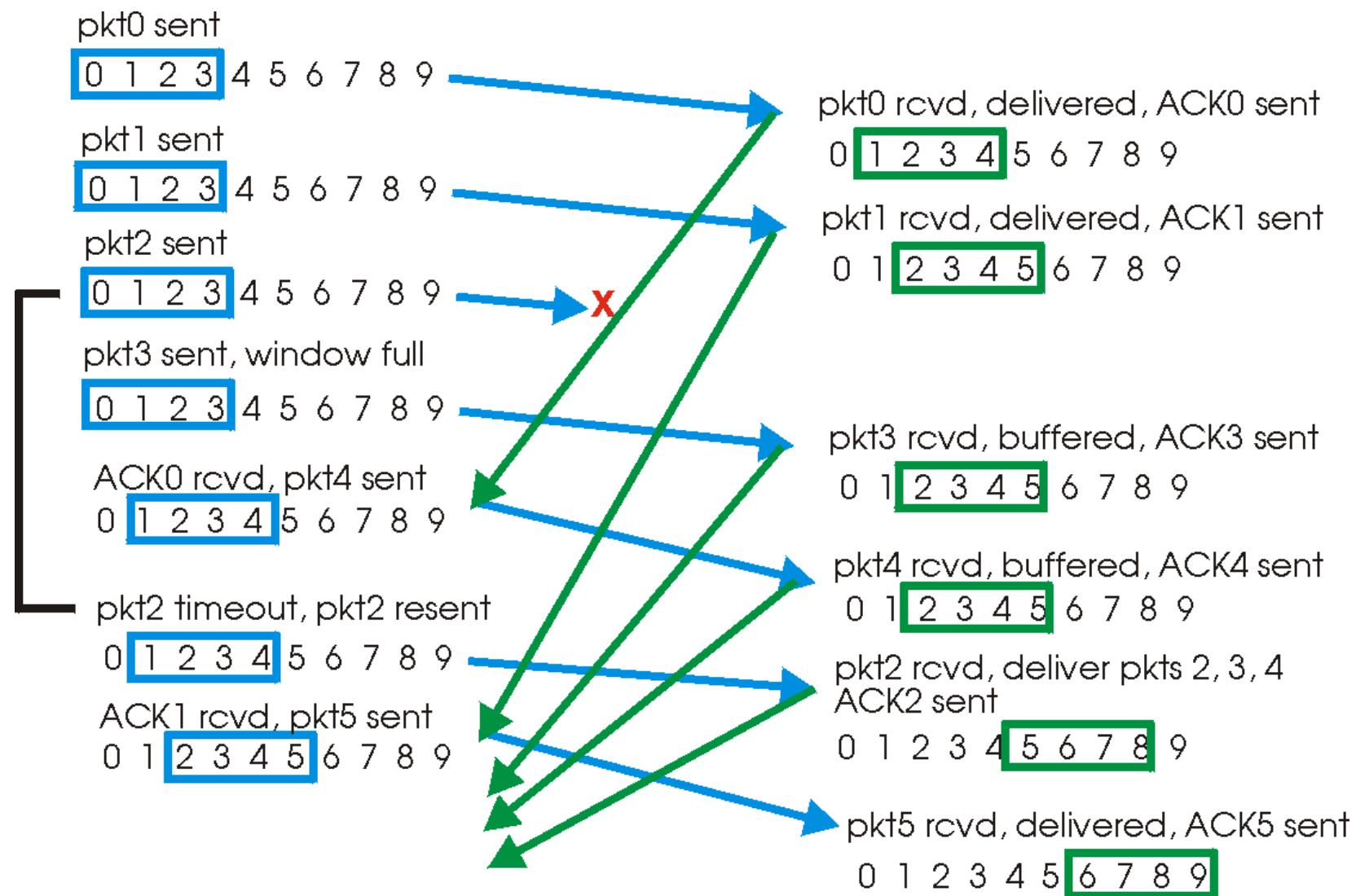
Paket n in [rcvbase-N,rcvbase-1]

- ACK(n)

sonst:

- ignorieren

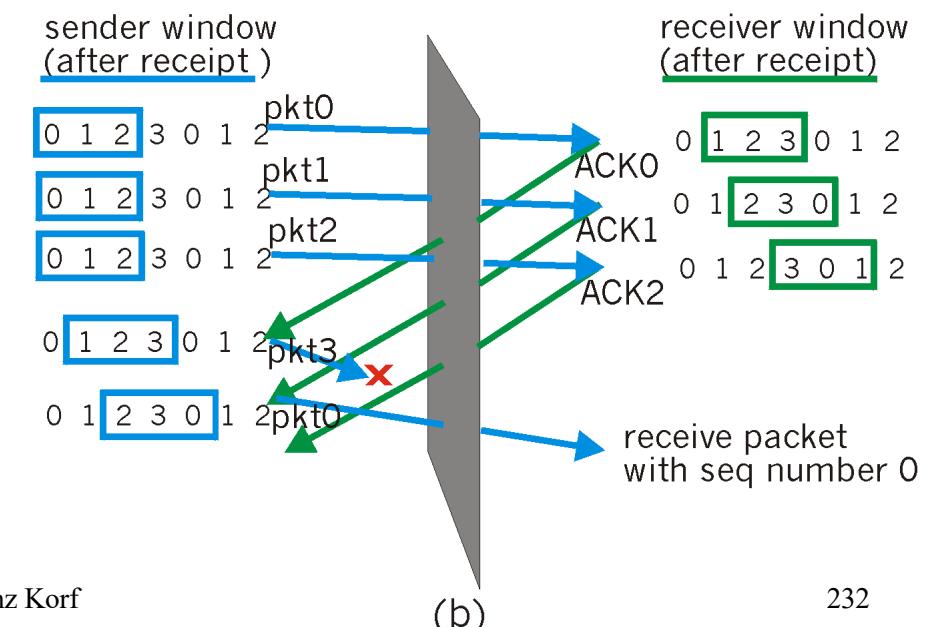
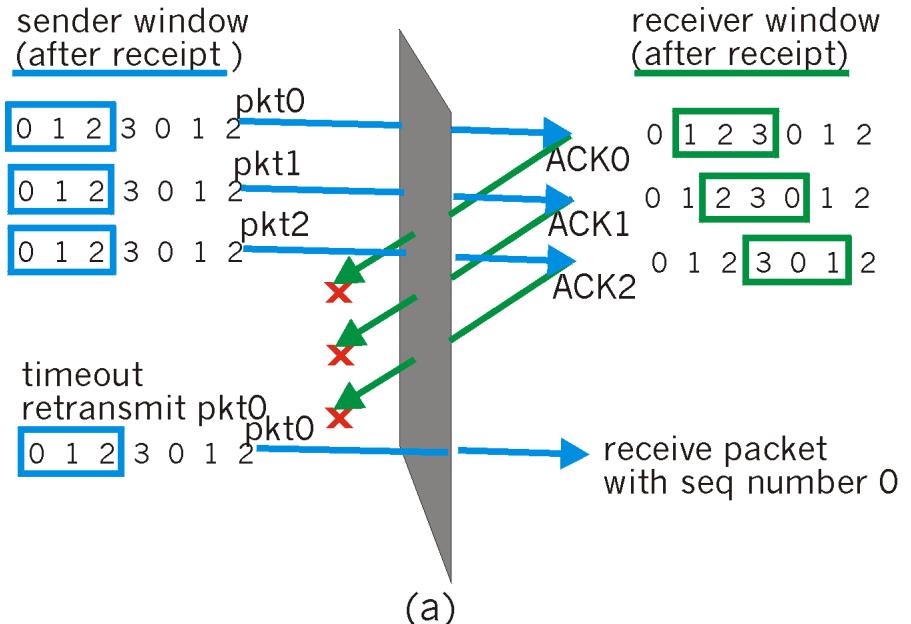
Selective repeat in Aktion



Stellen Sie bitte typische Szenarien mit den Demonstrator
http://www.ccs-labs.org/teaching/rn/animations/gbn_sr/ nach

Selective repeat Dilemma

- Sequenznummer 0, 1, 2, 3
- Fenstergröße = 3
- Empfänger sieht keinen Unterschied zwischen beiden Szenarien !
- Empfänger liefert Duplikate fälschlicherweise als neue Daten ab (a)



Zusammenfassung: Prinzipien des zuverlässigen Datentransfers

Erkennen eines verfälschten Pakets beim Empfänger:

- durch eine **Prüfsumme**

Melden des Empfängerzustands an den Sender:

- durch eine **Quittung (ACK)**

Reparieren von Fehlern beim Sender:

- durch **wiederholtes Senden** eines Pakets

Entdecken von Duplikaten / fehlenden Paketen beim Empfänger:

- durch **Sequenznummern**

Entdecken komplett verloren gegangener Pakete beim Sender:

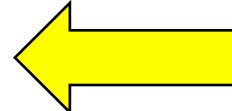
- durch **Timer**

Anpassen der Sendegeschwindigkeit an den Empfänger:

- durch **Window-Mechanismus (Sendefenster)**

Kapitel 4: Transportschicht

Gliederung

- Dienste und Prinzipien auf der Transportschicht
- Multiplexen und Demultiplexen von Anwendungen
- Verbindungsloser Transport: UDP
- Prinzipien des zuverlässigen Datentransfers
- Verbindungsorientierter Transport: TCP 
- TCP – Überlastkontrolle (Staukontrolle)
- Zusammenfassung

Textbuch zu diesem Kapitel: J. Kurose & K. Ross: Computernetzwerke – Der Top-Down-Ansatz, Kapitel 3

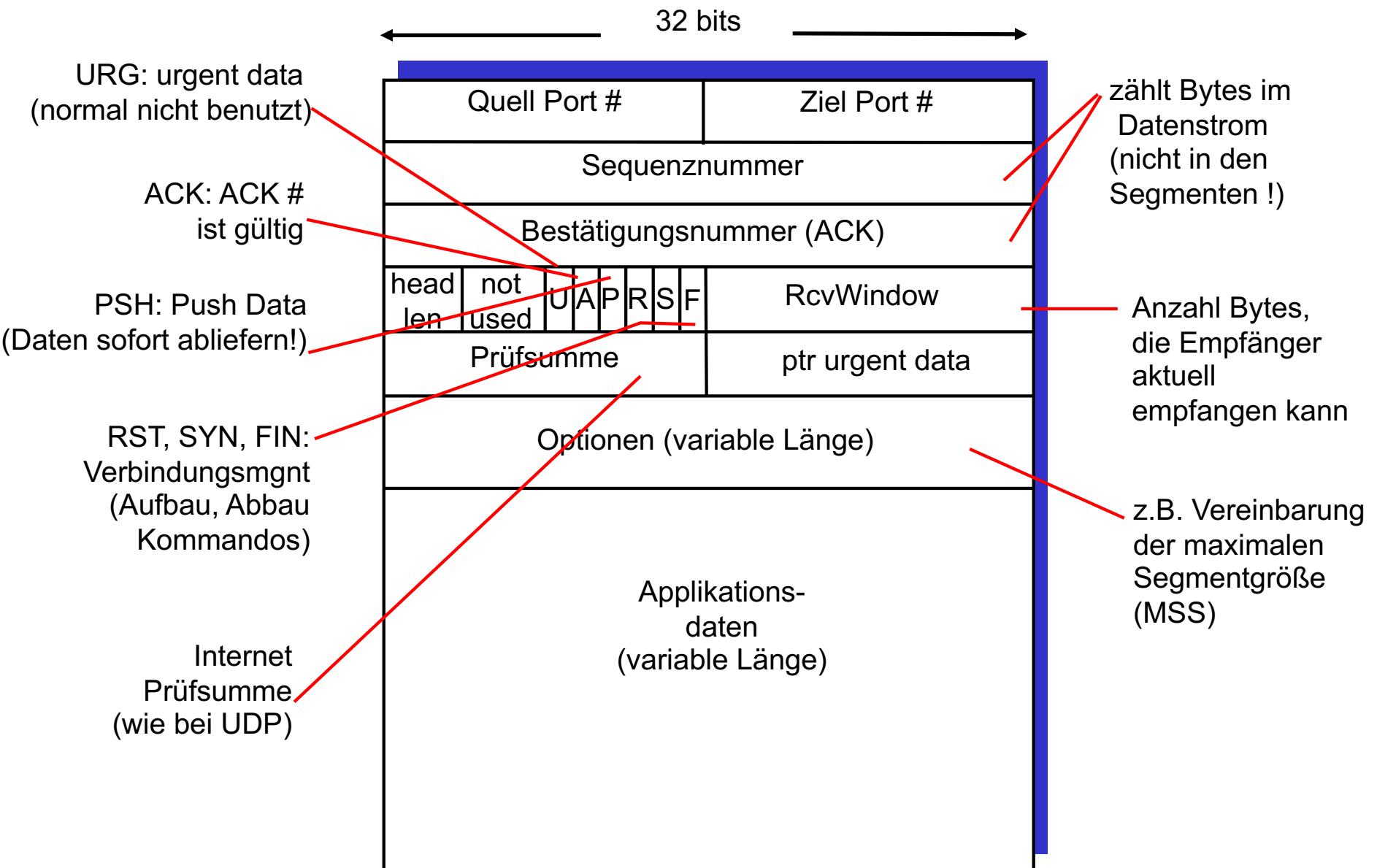
Folien und Abbildung teilweise aus:

J. Kurose & K. Ross: Computernetzwerke – Der Top-Down-Ansatz

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

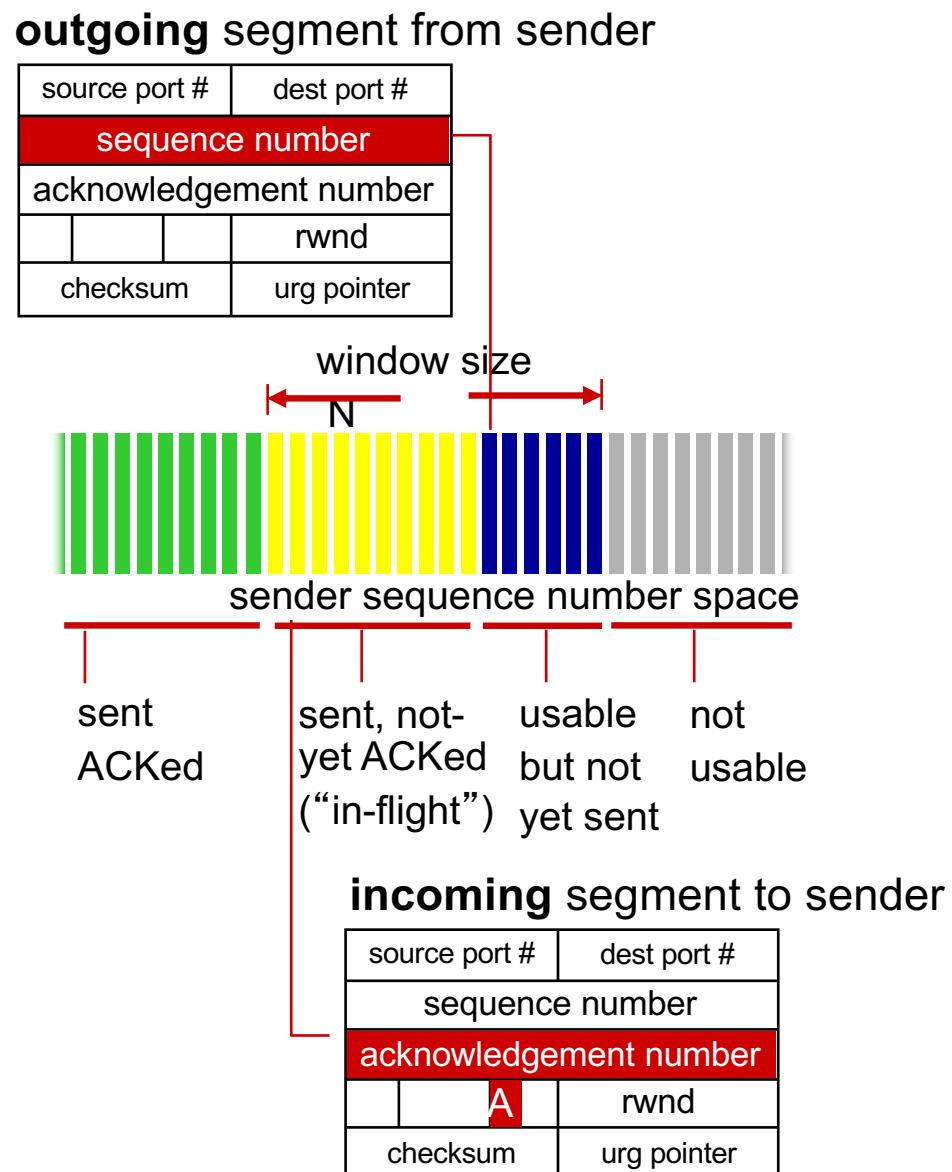
- Punkt zu Punkt - Verbindung: ein Sender, ein Empfänger
- Empfängt/liefert zuverlässigen, reihenfolge-bewahrenden Byte-Strom: erzeugt selbst Pakete ("Segmente")
- Voll-Duplex Datentransfer: Bidirektonaler Datenfluss in derselben Verbindung
- MSS: maximum segment size: Wird in der Regel aus der MTU (Maximum Transmission Unit) des lokalen Hosts bestimmt wird.
- Puffer im Sender & Empfänger
- Verbindungsorientiert: Handshaking (Austausch von Kontrollnachrichten) initialisiert Sender- und Empfängerzustand vor dem Datentransfer
- Flusskontrolle: Sender kann den Empfänger nicht überfluten (durch Sendefenster)
- Überlast/Staukontrolle: Steigere die Übertragungsgeschwindigkeit ("Slowstart"), bis ein Paket verloren geht (durch Sendefenster)
- Pipeline-Protokoll: TCP Stau- und Flusskontrolle bestimmen die Fenstergröße

TCP Segment Struktur

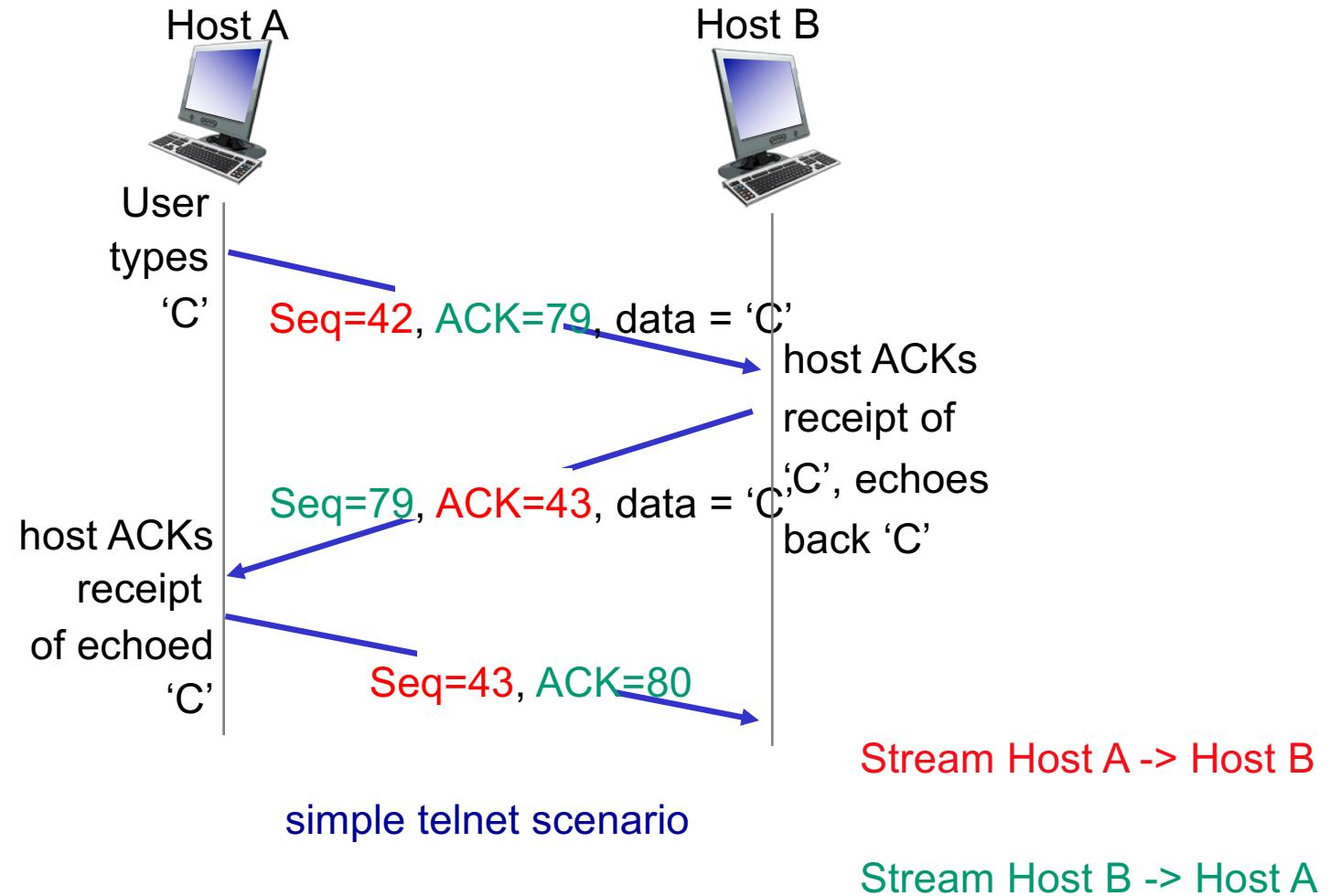


TCP Sequenznummern

- Es wird ein Strom von Bytes übertragen. Diese sind durchnummieriert.
 - Sequenznummer: Die kleinste Bytenummer, die im Paket gesendet wird.
 - Acknowledgement Nummer: Das nächste Byte im Strom das erwartet wird.
 - Frage: Wie behandelt Empfänger Segmente, die außerhalb der Reihenfolge ankommen?
TCP Spezifikation legt das nicht fest



TCP Sequenznummer: Telnet Beispiel



TCP: Round Trip Time (RTT) und Timeout

TCP verwendet einen **Timeout** Mechanismus um verlorene Segmente oder Ack's wiederherzustellen.

Frage: Wie setzt TCP den Timeout-Wert?

- Der Timeout-Wert sollte größer als die RTT sein.
 - Problem RTT ändert sich ständig
- Ist die Timeout Zeit zu kurz - verfrühtes Timeout => unnötiges Neuversenden
- Ist die Timeout Zeit zu lang => langsame Reaktion auf Verlust von Segmenten

Frage: Wie wird die RTT geschätzt?

- **SampleRTT**: gemessene Zeit vom Versenden eines Segments bis zur Bestätigung durch ACK
 - Ignorieren von Wiederholungen und kumulativ quittierte Segmente
- **SampleRTT** ändert sich dynamisch, RTT sollte jedoch sich nur langsam ändern
 - Benutzung mehrerer aktueller Messungen, nicht nur den letzten Wert von SampleRTT

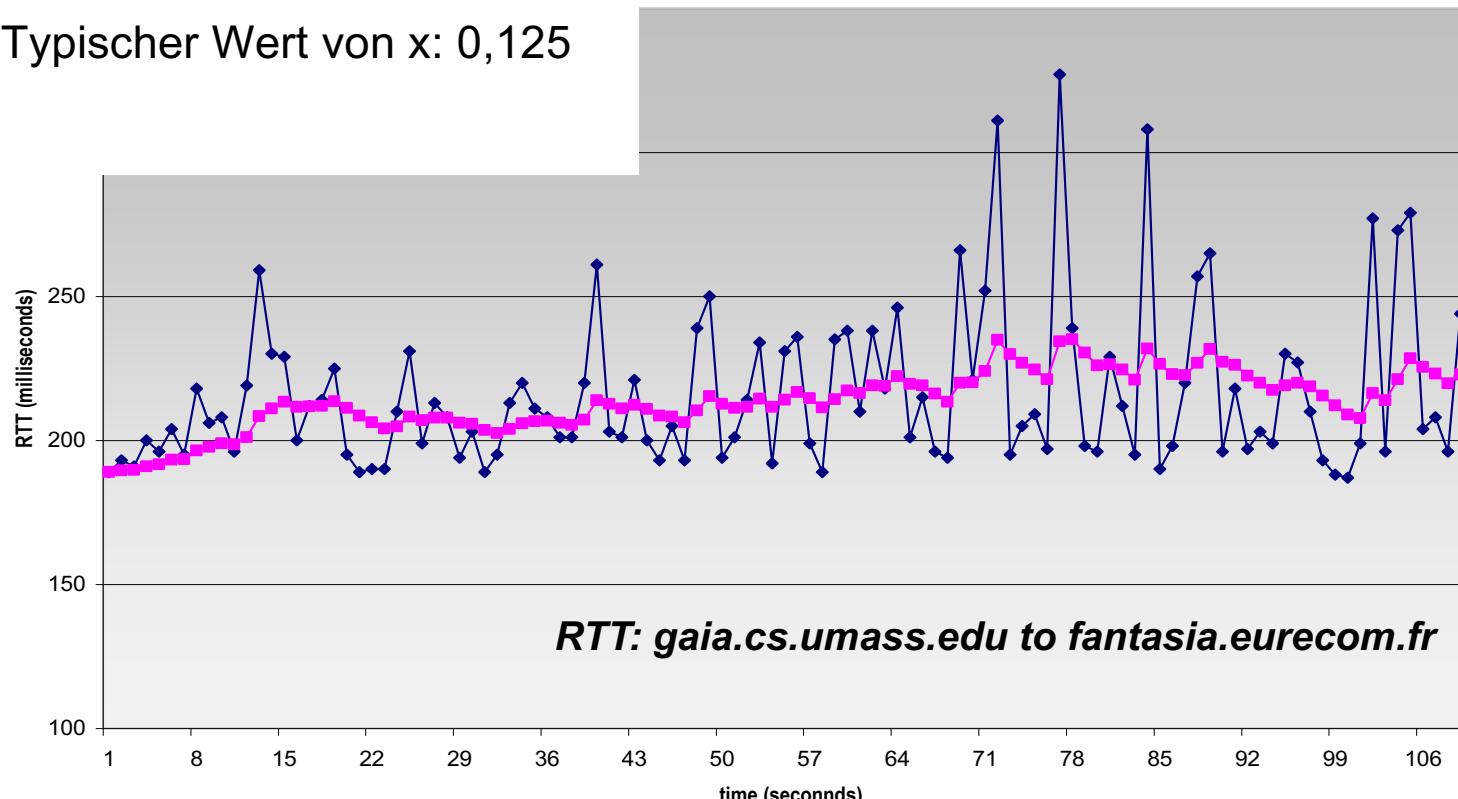
TCP: Abschätzung der RTT

RTT schwankt aufgrund von wechselnder Netzwerklast und Last auf dem Endsystem

Konsequenz: Untypische Werte werden ausgefiltert.

$$\text{EstimatedRTT} = (1 - x) * \text{EstimatedRTT} + x * \text{SampleRTT}$$

- Typischer Wert von x: 0,125



TCP: Abschätzung der Variabilität RTT

Dieses Maß gibt an, wie stark der SampleRTT vom EstimatedRTT abweicht.

$$\text{DevRTT} = (1 - y) * \text{DevRTT} + y * |\text{SampleRTT} - \text{EstimatedRTT}|$$

- Typischer Wert von $y = 0,25$

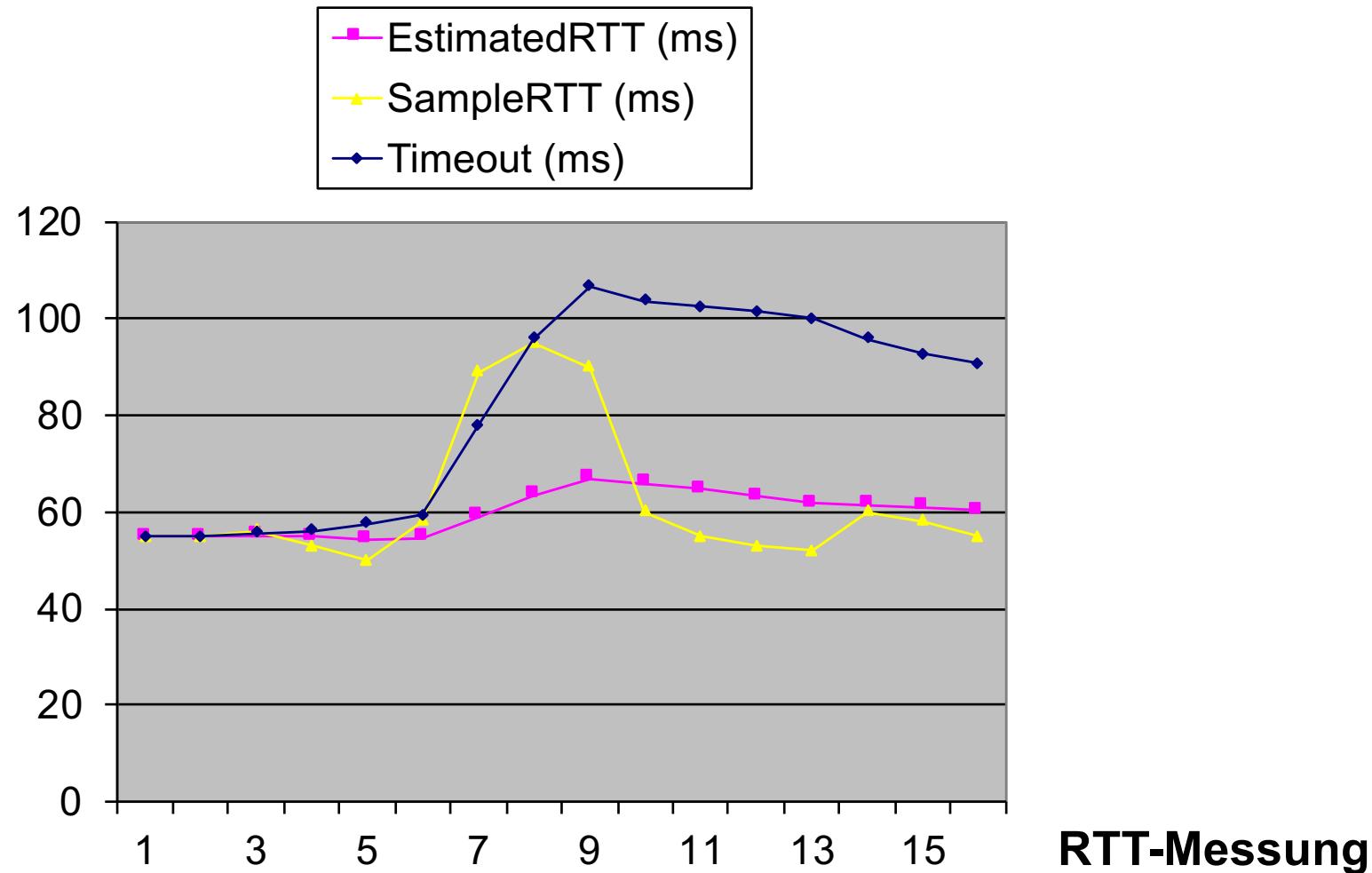
TCP: Bestimmung des Timeout

- $\text{Timeout} = \text{EstimatedRTT} + \text{Sicherheitsabstand}$
- Das Timeout soll größer als EstimatedRTT sein, damit unnötige Übertragungswiederholungen vermieden.
- Das Timeout soll nicht viel größer als EstimatedRTT sein, damit bei einem Segmentverlust die Neuübertragung zugig stattfindet.
- Das zum Timeout dazu addierte Sicherheitsabstand muss an Schwanken der RTT angepasst sein.

$$\text{Timeout} = \text{EstimatedRTT} + 4 * \text{DevRTT}$$

- Startwert: $\text{Timeout} = 1$
- Ist ein Timeout aufgetreten, wird das Timeout verdoppelt.

TCP: Beispielberechnung des Timeouts

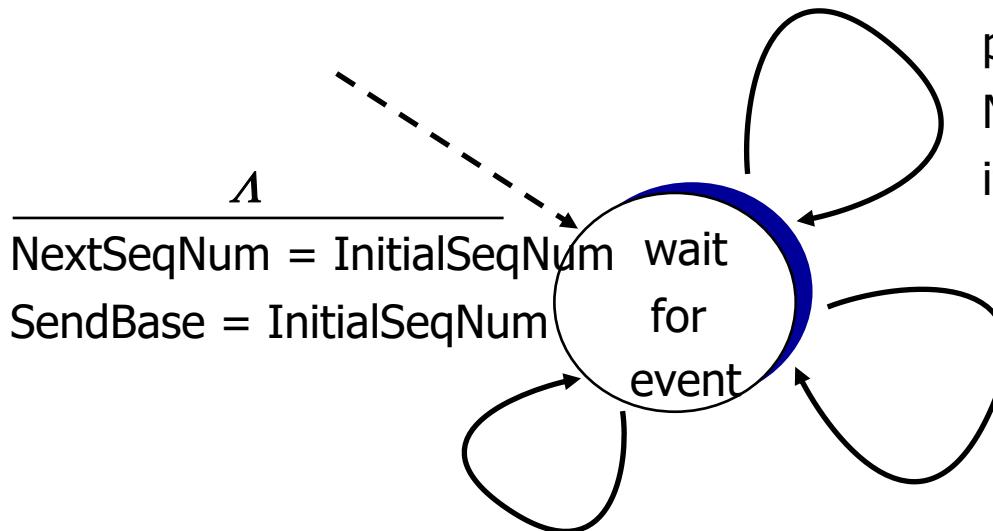


TCP: Zuverlässiger Datentransfer

- TCP realisiert einen zuverlässigen Datentransfer oberhalb des unzuverlässigen Best-Effort-Dienst von IP.
- Das empfohlene TCP Timer-Management (RFC 6298) verwendet nur einen Timer für die Wiederholung von Übertragungen – auch bei mehreren nicht bestätigten Segmenten.
- Pipeline Ansatz
- Kumulatives ACK
- Auslösung der Neuübertragung eines Segments durch
 - Timeout
 - Doppelte ACKs

TCP: Zuverlässiger Datentransfer

SendBase–1: last cumulatively ACKed byte



```

NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum
if (y > SendBase) {
    SendBase = y
    if (there are currently not-yet-acked segments)
        start timer
    else
        stop timer
}
    
```

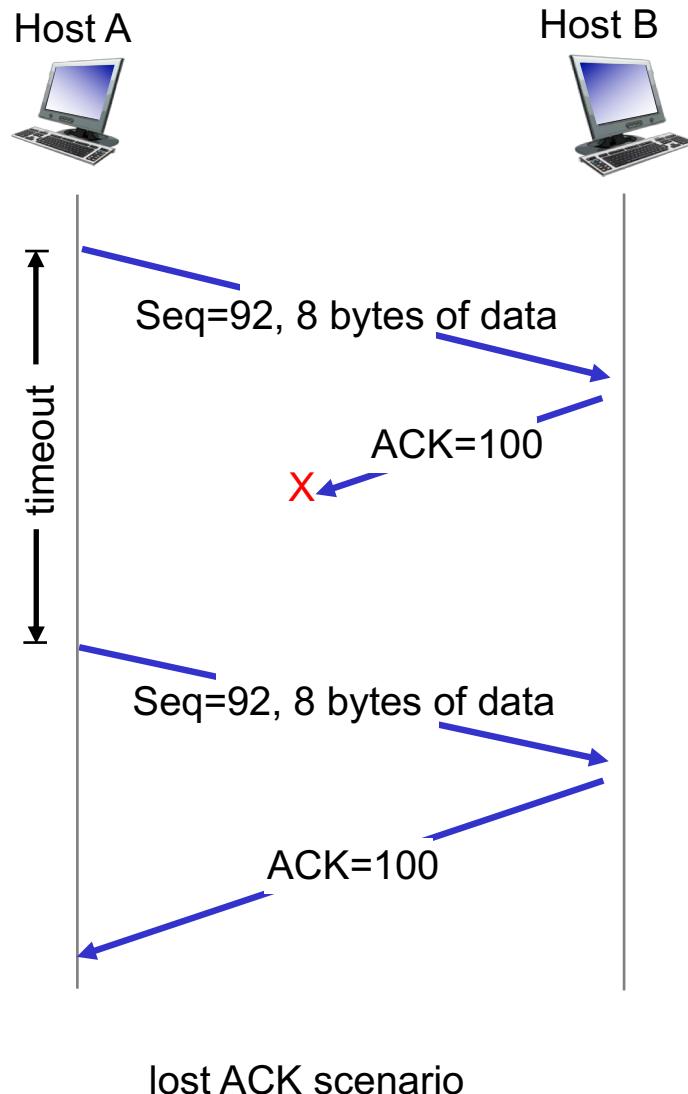
data received from application
 create segment, seq. #: NextSeqNum
 pass segment to IP (i.e., “send”)
 $\text{NextSeqNum} = \text{NextSeqNum} + \text{length(data)}$
 if (timer currently not running)
 start timer

 timeout
 retransmit not-yet-acked segment
 with smallest seq. #
 start timer

Hier vereinfachter Sender:

- Daten nur in eine Richtung
- keine Überlastkontrolle

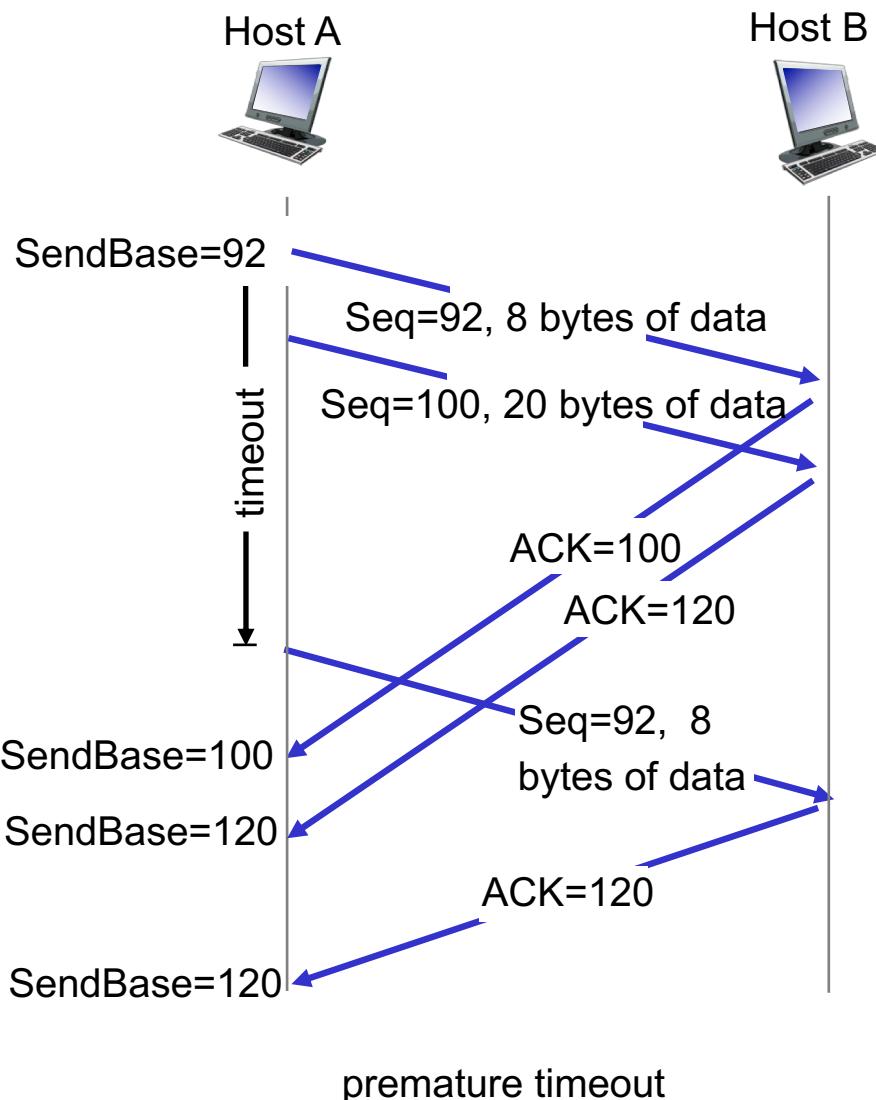
Interessante Szenarien



Kommentar:

- Host B erkennt, dass das zweite Paket ein Duplikat ist und verwirft es.

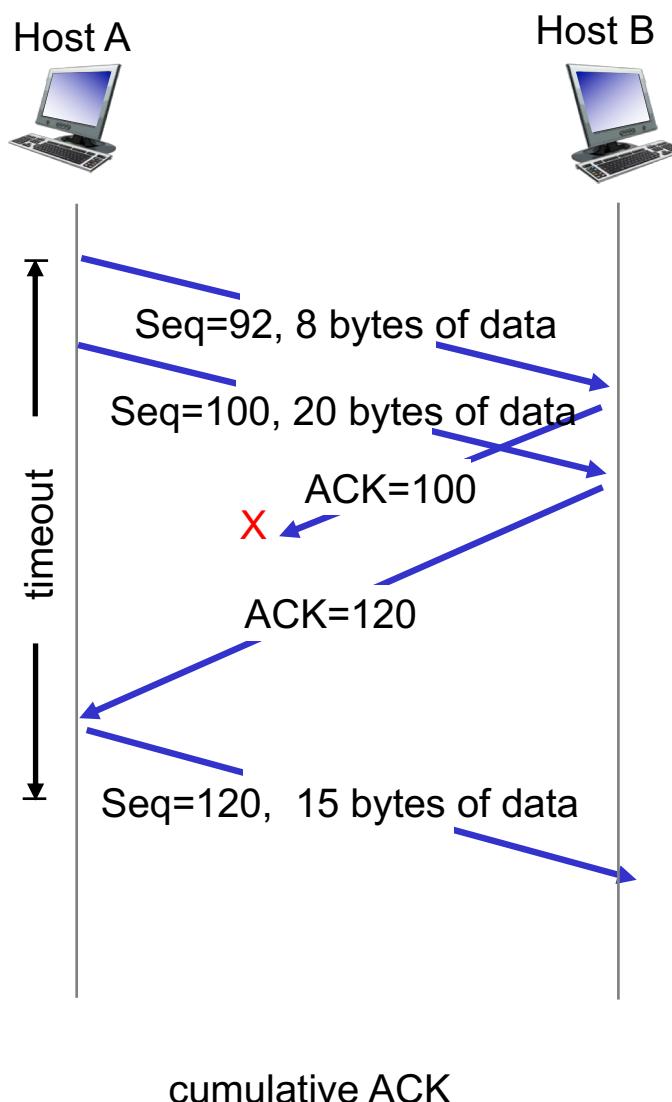
Interessante Szenarien (Fortsetzung)



Kommentare:

- Pipelining
- Alle Bestätigungen treffen nach dem Timeout ein.
- Nur das erste Segment wird erneut übertragen.
- Das zweite Segment wird nicht erneut übertragen, da das passende Ack vor dem zweiten Timeout eintrifft.

Interessante Szenarien (Fortsetzung)



Kommentar:

- Das verloren gegangene Ack 100 hat keine Auswirkungen, da das darauffolgende Ack noch rechtzeitig vor dem Timeout eintrifft.

Diskussion: Verlängerung des Timeout Intervalls

Oft verwendete in TCP Implementierungen:

Ist ein Timeout aufgetreten, wird das Timeout verdoppelt.

Dann wird das älteste nicht erfolgreich übertragene Segment erneut übertragen.

- Bei mehreren aufeinander folgenden Timeouts wächst das Timeout exponentiell ein.
- Meistens treten die Timeouts aufgrund von der Verzögerung in den Switches ein.
- Einfach Form der Überlastkontrolle / Staukontrolle.

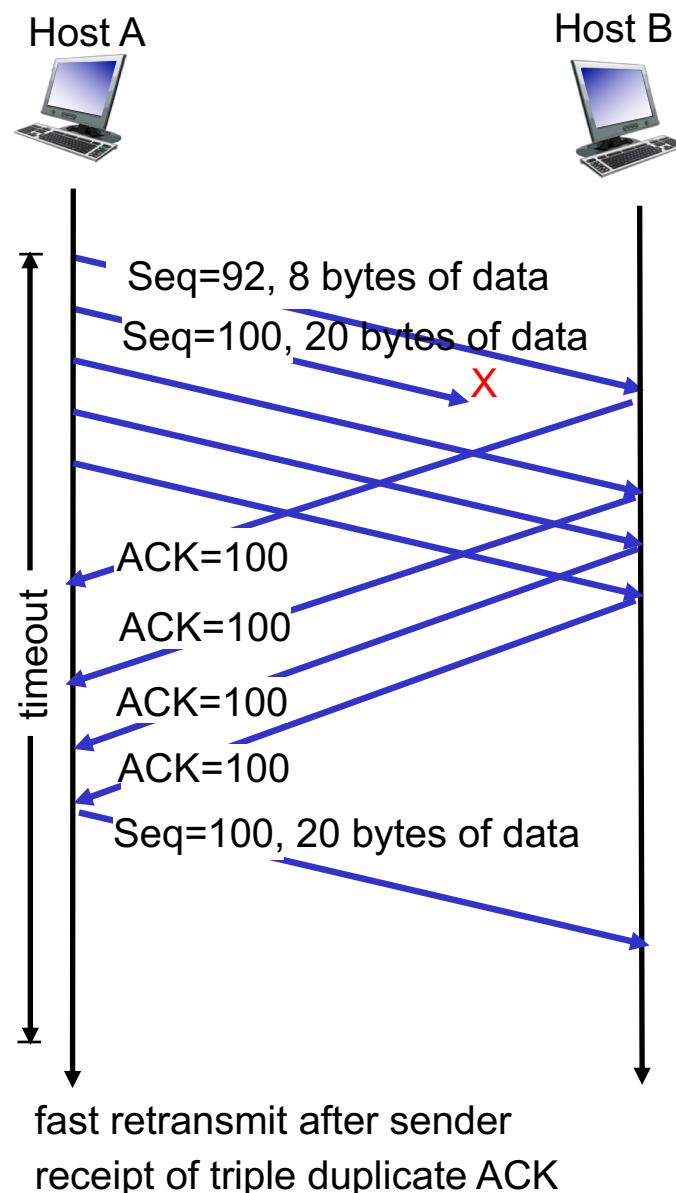
TCP: Fast Retransmit

- **Verlust eines Segments:** Das relativ lange Timeout erhöht die Ende-zu-Ende Latenz.
- Beobachtung: Erkennen von Paketverlusten durch doppelte ACKs
 - Sender schickt häufig viele Segmente direkt hintereinander
 - Wenn ein Segment verloren geht, führt dies zu vielen doppelten ACKs, weil für alle nachfolgenden Segmente vom Empfänger ein ACK mit der Sequenznr. vor dem verlorenen Segments gesendet wird
- Umsetzung der Beobachtung: Fast Retransmit [RFC 1122, RFC 2581]
 - Wenn der Sender 3 Duplikate eines ACK erhält, dann nimmt er an, dass das Segment verloren gegangen ist:
 - Segment erneut schicken, bevor der Timer abläuft
 - Deutliche Performancesteigerung, weil nicht unnötig auf Timerablauf gewartet warten muss!

TCP ACK-Erzeugung beim Empfänger bei Fast Retransmit

Ereignis	TCP Empfängeraktion
Ankunft in richtiger Reihenfolge, ohne Lücken (Seq.-Nr == erw. Seq-Nr) alles sonst schon mit ACK quittiert	verzögere ACK. Wartet bis zu 500ms auf das nächste Segment für den Sender. Wenn keins in dieser Zeit kommt, sende ACK
Ankunft in richtiger Reihenfolge, ohne Lücken (Seq.-Nr == erw. Seq-Nr) ein verzögertes ACK steht aus	sende sofort ein einzelnes kumulatives ACK
Ankunft außerhalb der Reihenfolge Seq-Nr. höher als erwartet (Seq.-Nr > erw. Seq-Nr) Lücke entdeckt	sende Duplikat-ACK, Verweis auf die nächste erwartete Seq-Nr. (das nächste erwartete Byte)
Ankunft eines Segments, das teilweise oder ganz eine Lücke füllt	sofortiges ACK, wenn das Segment am unteren Ende der Lücke startet

TCP: Beispiel Fast Retransmit

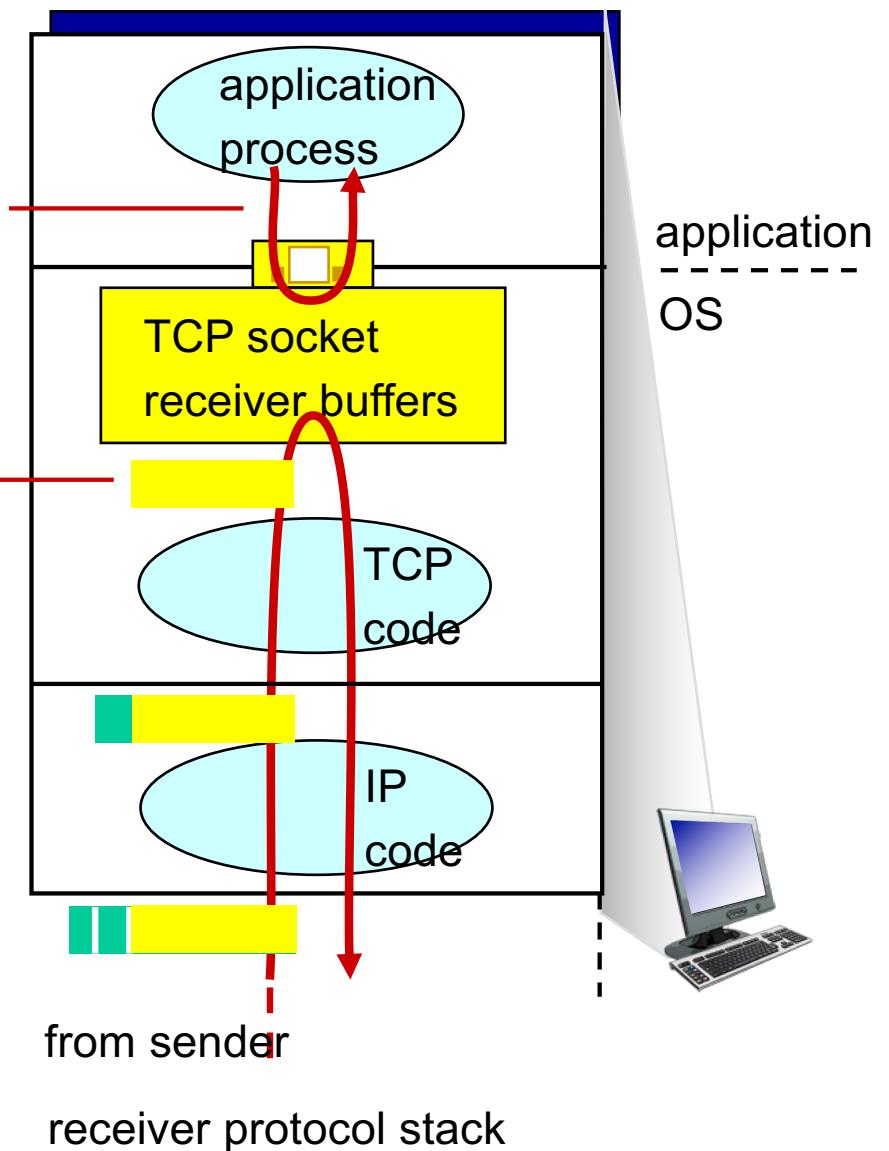


TCP: Flusskontrolle

Was steuert die Flusskontrolle?

- Kontrolle, dass der Puffer zwischen Anwendung und OS nicht überläuft.
- Anpassung der Sendegeschwindigkeit des Senders an die Konsumgeschwindigkeit des Verbrauchers.

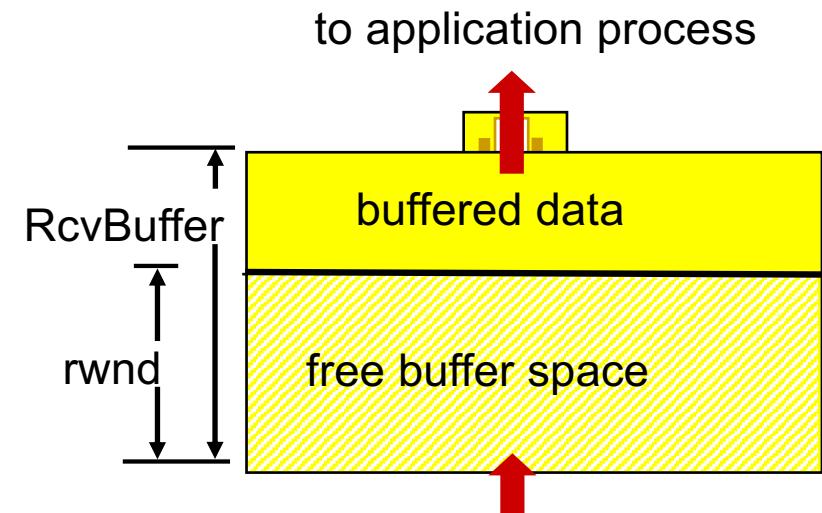
application may remove data from TCP socket buffers
....
... slower than TCP receiver is delivering (sender is sending)



TCP Flusskontrolle

Funktionsweise der Flusskontrolle:

- Empfänger: informiert den Sender explizit über die aktuelle Größe des freien Puffers (ändert sich dynamisch)
RcvWindow-Feld im TCP-Header
- Sender: hält die Anzahl gesendeter, nicht quittierter Datenbytes kleiner als den aktuell erhaltenen Wert für **RcvWindow** (Sendefenster-Mechanismus)
- Frage: Was passiert, wenn der Empfänger **RcvWindow = 0** meldet?
- **Typische RcvBuffer size = KiB (socket Parameter)**
Viel OS passen RvcBuffer Size dynamisch an.



to application process

buffered data

free buffer space

TCP segment payloads

receiver-side buffering

RcvBuffer = Größe des TCP Empfangspuffers

RcvWindow = aktuell freier Platz im Empfangspuffer

TCP Verbindungsmanagement

Wiederholung:

- TCP-Sender und Empfänger bauen eine Verbindung auf, bevor Daten ausgetauscht werden!
- Initialisierung der TCP-Variablen:
 - Sequenznummern
 - Puffer, Flusskontroll-Info (d.h. **RcvWindow**)
- Client: Initiator der Verbindung

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

- Server: durch Client kontaktiert

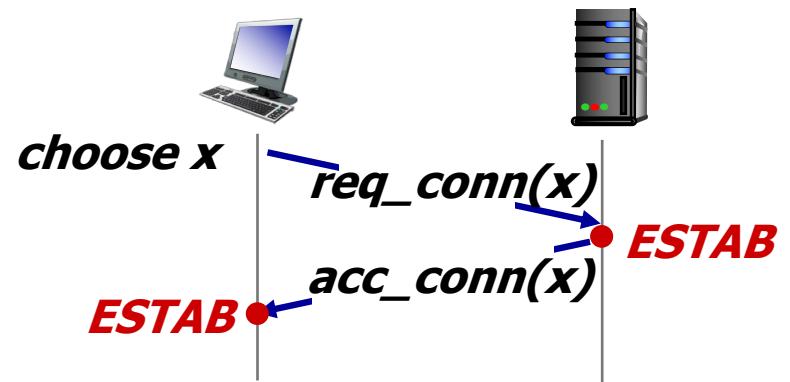
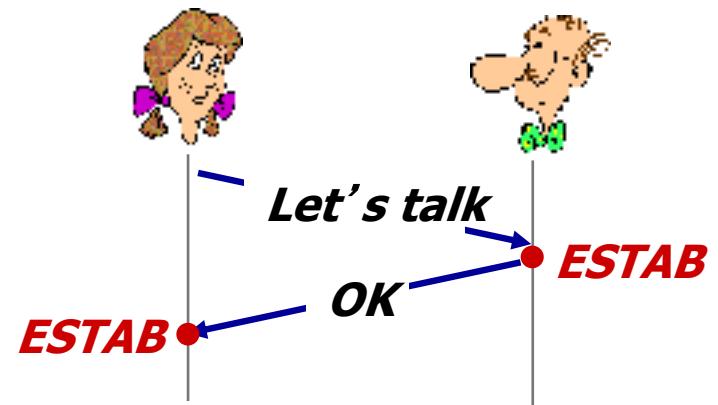
```
Socket connectionSocket = welcomeSocket.accept();
```

Warum verwendet TCP ein 3-Wege Handshake Protokoll?

Unsicherer Kommunikationskanal

- Wenn Bob keine Antwort sendet, dann weiß Alice nicht, ob Bob die Nachricht nicht erreicht hat, ob Bob überhaupt kommunikationsbereit ist oder ob die Quittung von Bob verloren gegangen ist.
- Bob weiß nicht, ob Alice seine Quittung erhalten hat.
- Dies ist am Anfang besonders kritisch, da nicht klar ist, ob der Empfänger überhaupt kommunikationsbereit ist – ein Timeout hilft hier nicht weiter.

2-Wege Handshake



TCP Verbindungsmanagement (open)

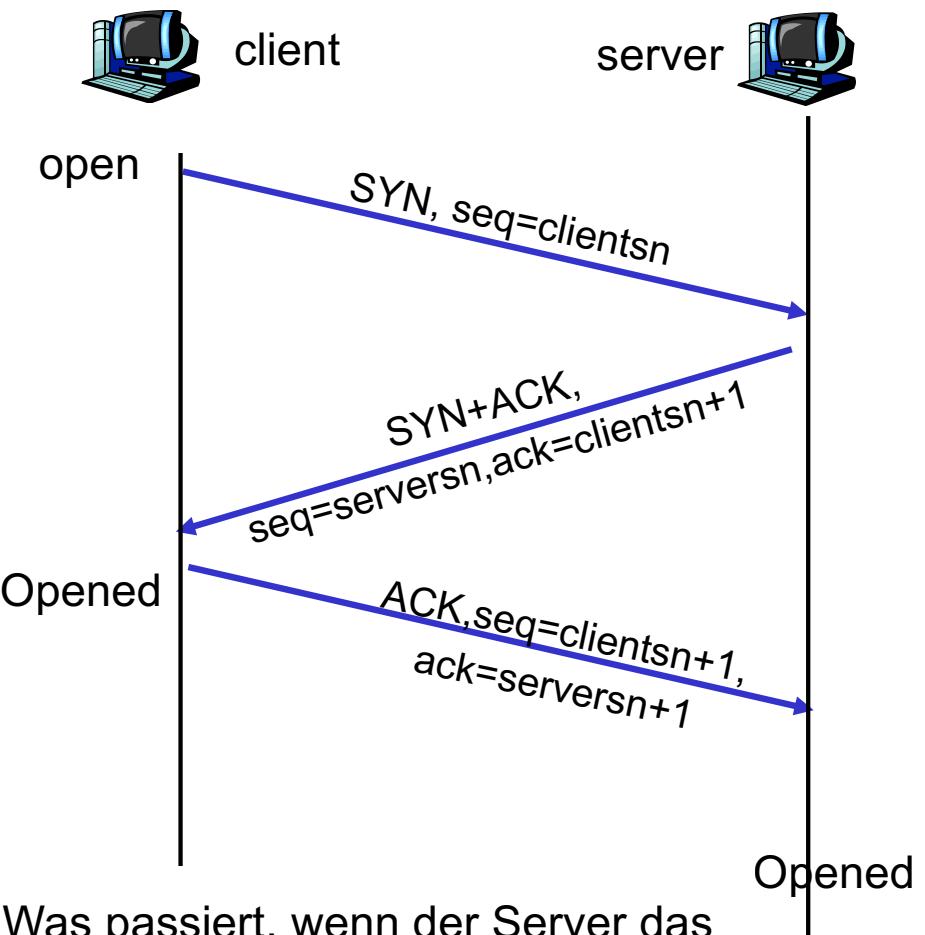
Aufbau einer Verbindung:

3 Wege-Handshake !

Schritt 1: Client-System sendet TCP-Segment mit SYN-Flag = 1 und initialer Client-Seq-Nr.

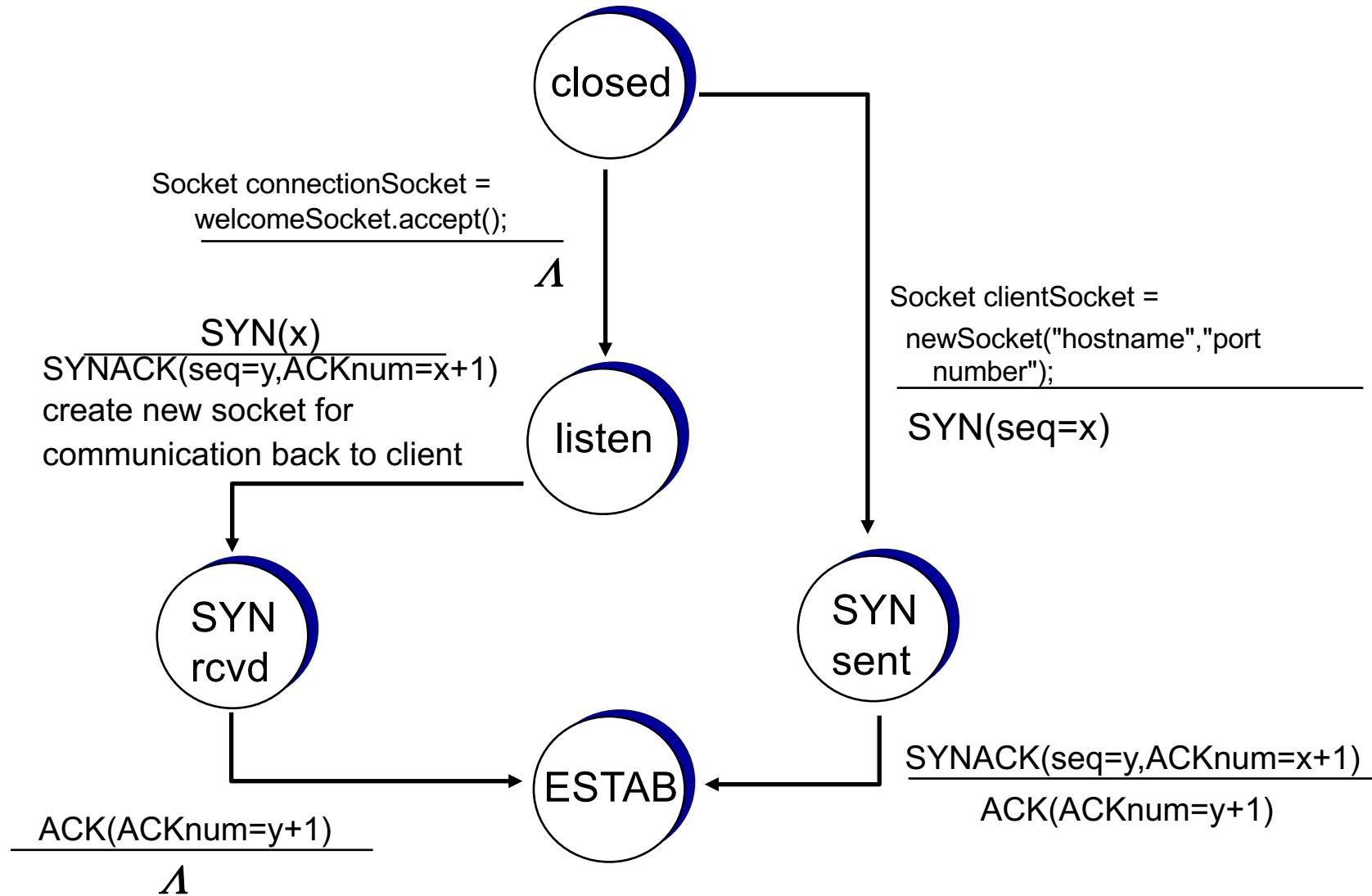
Schritt 2: Server-System empfängt SYN, belegt Puffer und antwortet mit SYN+ACK - Segment und initialer Server-Seq-Nr.

Schritt 3: Client empfängt SYN+ACK, legt Puffer an und antwortet mit einem ACK – dieses Segment darf bereits Daten beinhalten



Was passiert, wenn der Server das Empfangsport gar nicht bedient?
Antwort mit RST Flag

TCP 3 Wege Handshake als gemeinsame FSM



TCP Verbindungsmanagement (close)

Schließen einer Verbindung:

Client schließt socket: clientSocket.close();

Schritt 1: Client-System sendet TCP FIN
Kontrollsegment zum Server

Schritt 2: Server empfängt FIN, antwortet mit ACK.

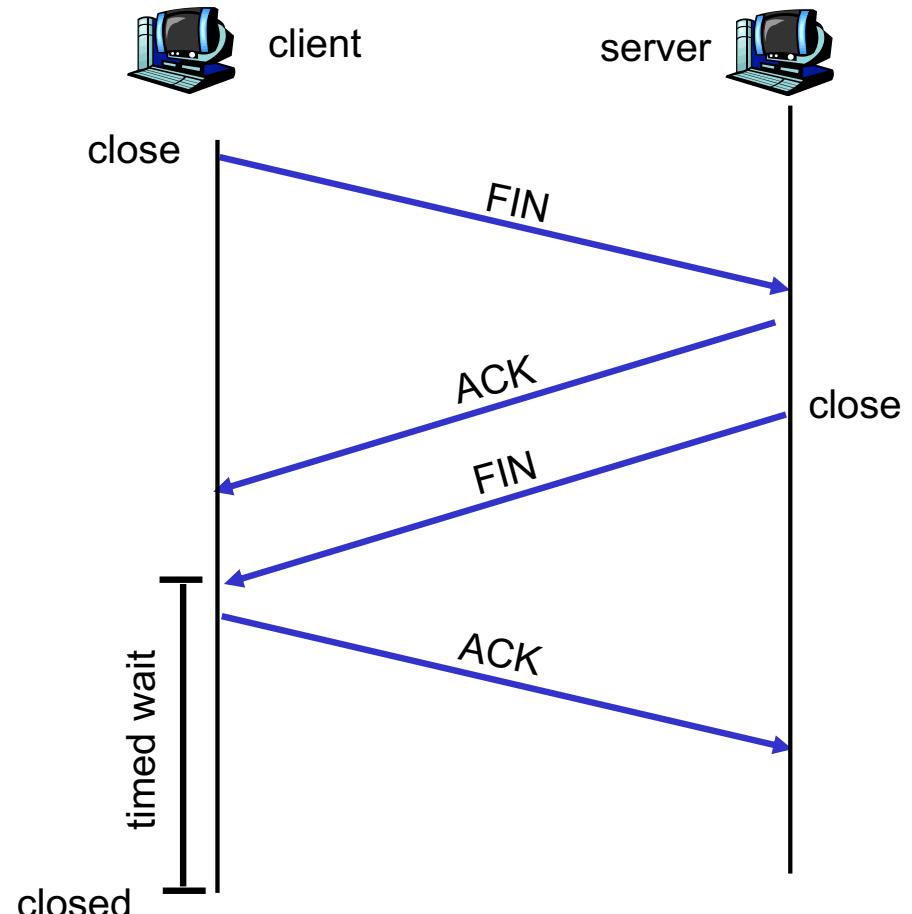
Schließt Verbindung, sendet FIN.

Schritt 3: Client empfängt FIN, antwortet mit ACK.
Wartet definierte Zeit ab, antwortet mit

ACK auf weitere empfangene FINs

Schritt 4: Server empfängt ACK. Verbindung ist geschlossen.

Four Way Handshake !



Übungsaufgabe

Wie könnte ein Port Scanner arbeiten (nmap)?

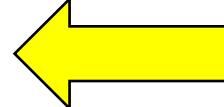
- Ein Port Scanner prüft offene Ports / ob ein Server auf diesen Port lauscht.

Mögliche Arbeitsweise

- Versuch des Aufbaus einer TCP Verbindung auf Port x von Rechner y
- Antwort enthält SYN Flag => Der Dienst ist aktiv
- Antwort enthält RST Flag => Der Weg bis zum Rechner ist freigeschaltet, aber auf dem Port lauscht kein Server.
- Kein Antwort: Die Anfrage wurde auf dem Weg zum Server abgefangen (z.B. Fire Wall)

Kapitel 4: Transportschicht

Gliederung

- Dienste und Prinzipien auf der Transportschicht
- Multiplexen und Demultiplexen von Anwendungen
- Verbindungsloser Transport: UDP
- Prinzipien des zuverlässigen Datentransfers
- Verbindungsorientierter Transport: TCP
- TCP – Überlastkontrolle (Staukontrolle) 
- Zusammenfassung

Textbuch zu diesem Kapitel: J. Kurose & K. Ross: Computernetzwerke – Der Top-Down-Ansatz, Kapitel 3

Folien und Abbildung teilweise aus:

J. Kurose & K. Ross: Computernetzwerke – Der Top-Down-Ansatz

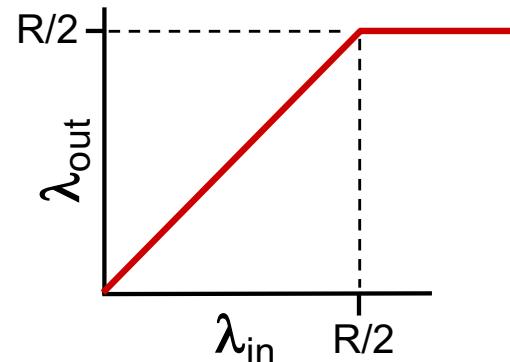
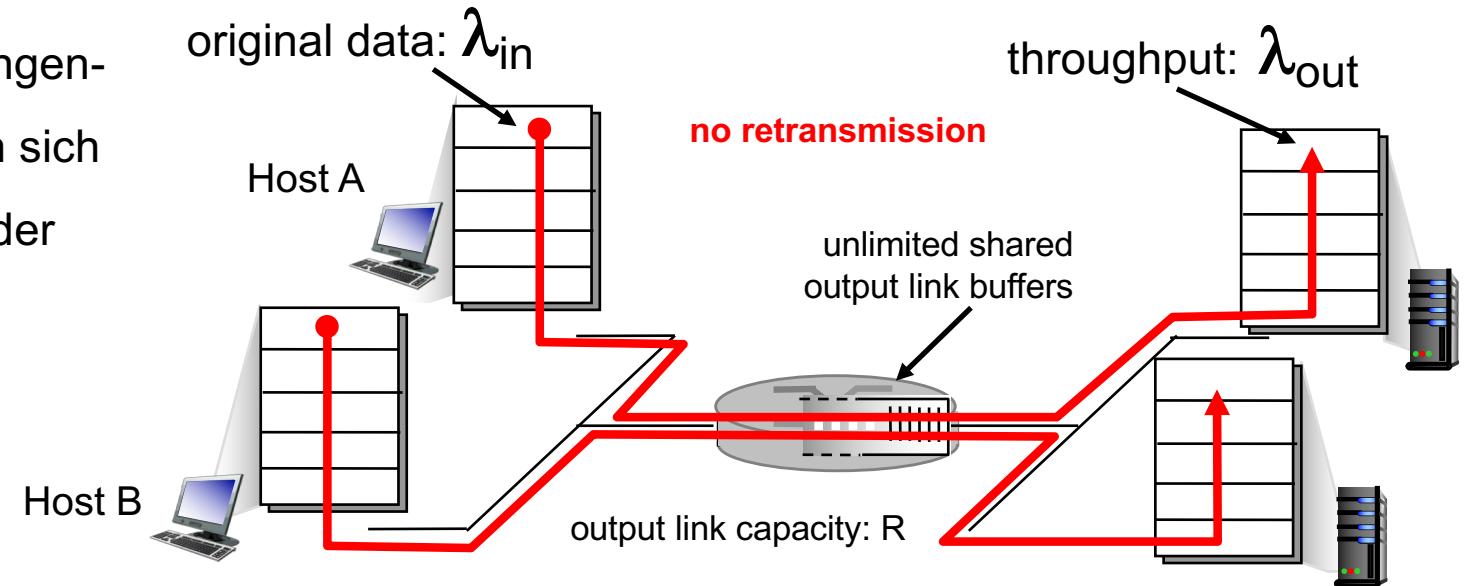
Überlastkontrolle / Staukontrolle (congestion control)

Stau:

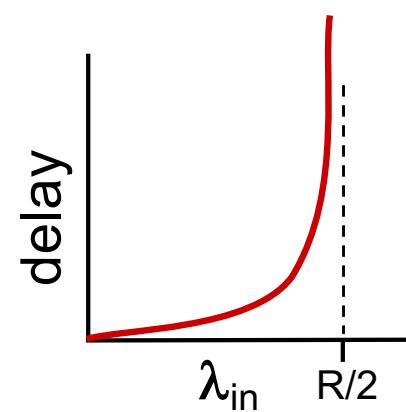
- praktische Sicht: “zu viele Quellen senden zu viele Daten zu schnell, das Netzwerk kann sie nicht alle bearbeiten”
- Unterschied zur Flusskontrolle (Sender → Empfänger)!
- Stau ist erkennbar an:
 - verlorene Pakete (Pufferüberlauf in den Routern)
 - große Verzögerungen (große Queues in den Puffern der Router)
- Die Vermeidung von Staus ist ein zentrales Netzwerkproblem

Stausituation Szenario 1

Große Warteschlangenverzögerung, wenn sich die Empfangsrate der Leitungskapazität annähert.



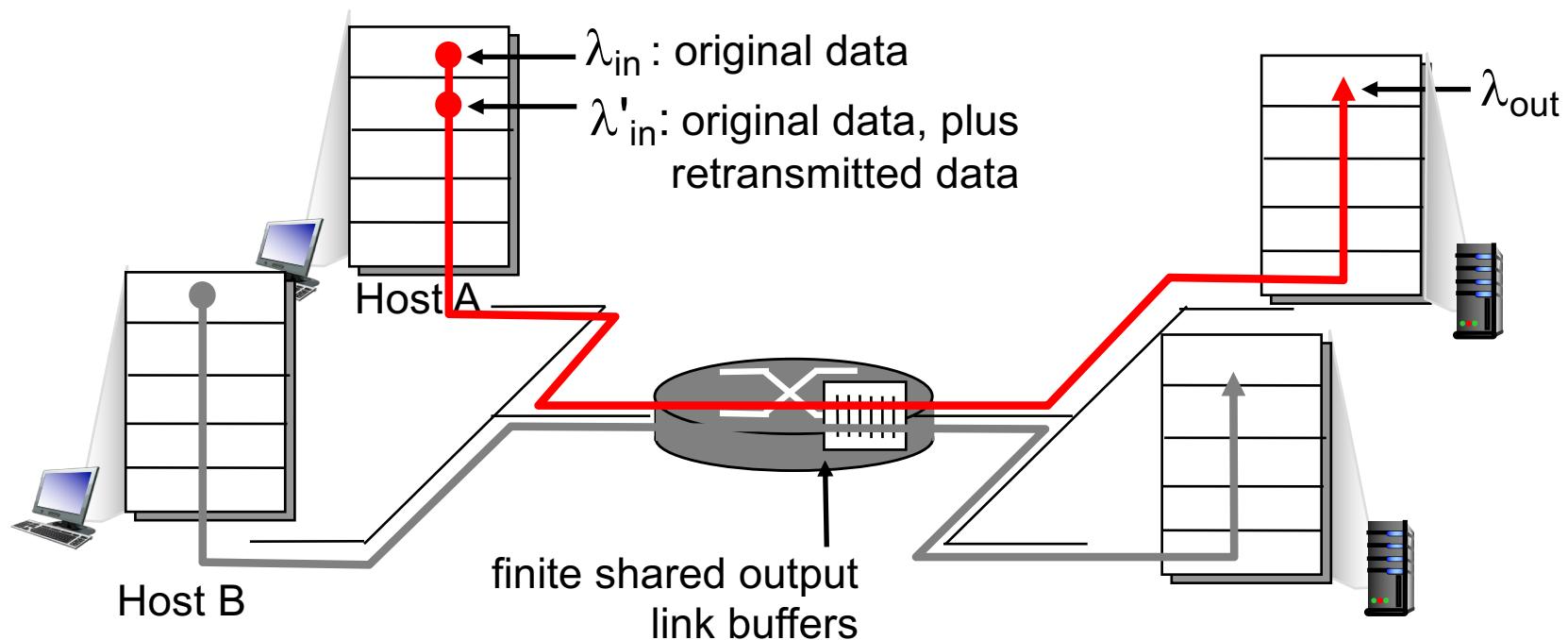
Maximaler Durchsatz pro Verbindung: $R/2$



Die Verzögerung wächst stark an, wenn sich die Ankunftsrate λ_n dem maximalen Durchsatz $R/2$ annähert.

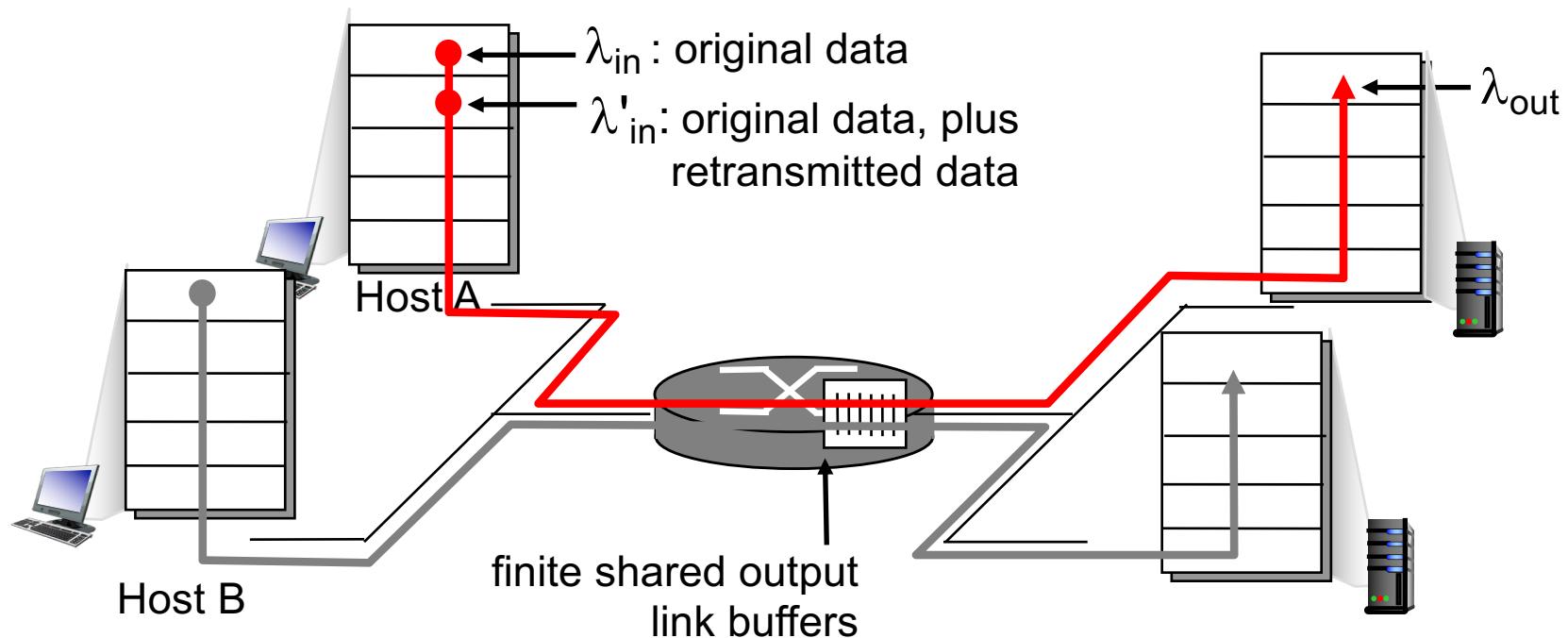
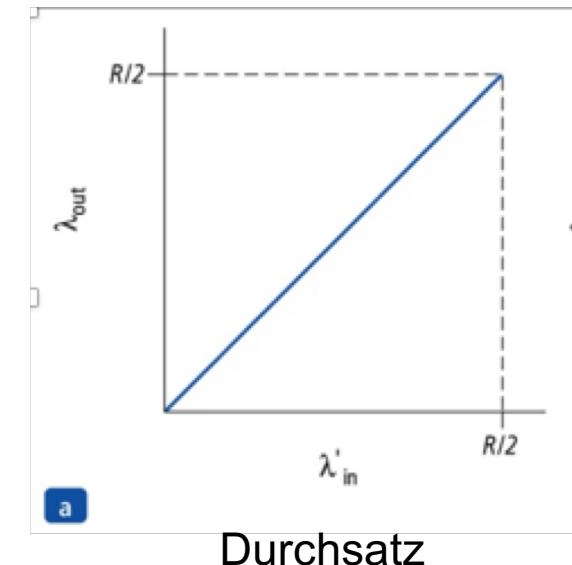
Stausituation Szenario 2

- Erneute Übertragung von timeout Paketen
 - application-layer input = application-layer output: $\lambda_{in} = \lambda_{out}$
 - transport-layer input includes *retransmissions*: $\lambda'_{in} \geq \lambda_{in}$
- Die Anzahl der neu zu übertragenden Pakete hat signifikante Auswirkungen auf den Durchsatz



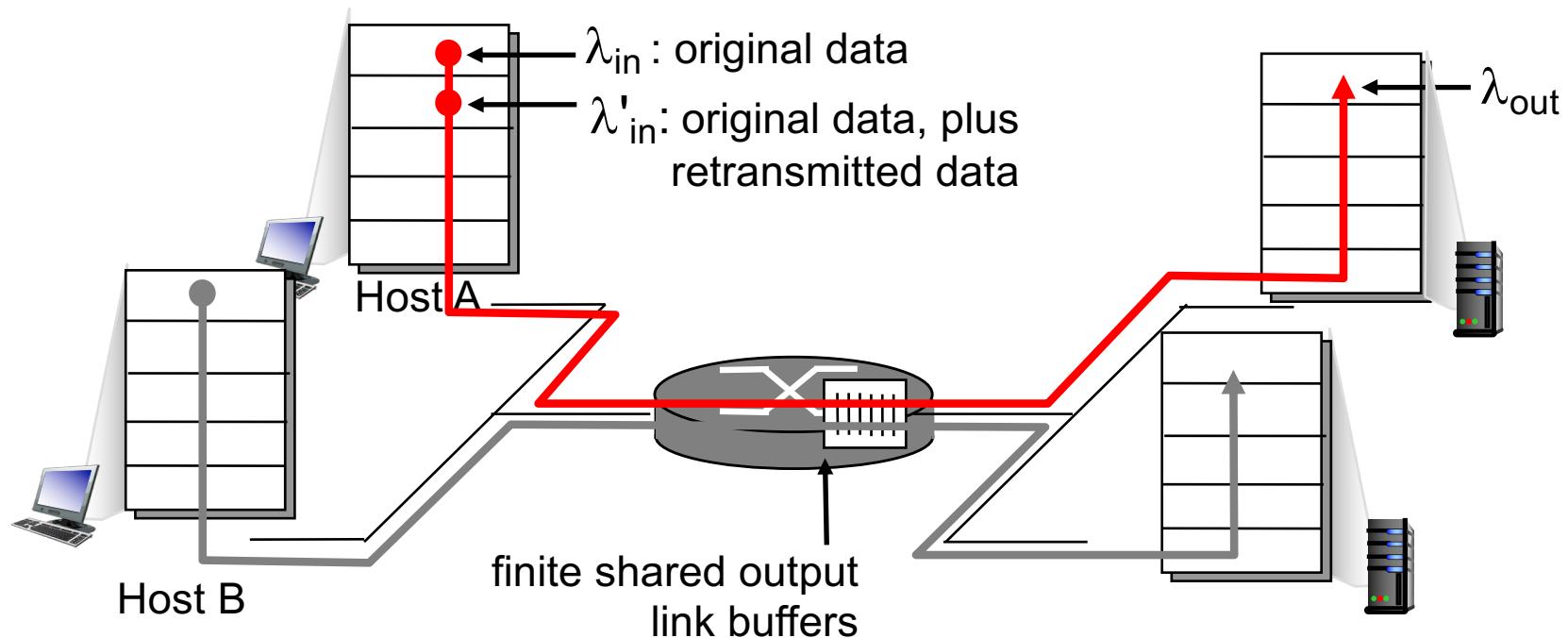
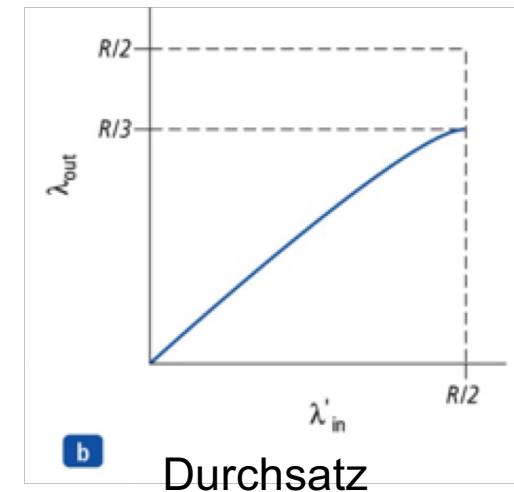
Stausituation Szenario 2

- Idealisierte Situation: Sender erkennt, ob Platz im Puffer vorhanden ist.
- Kein Paketverlust.



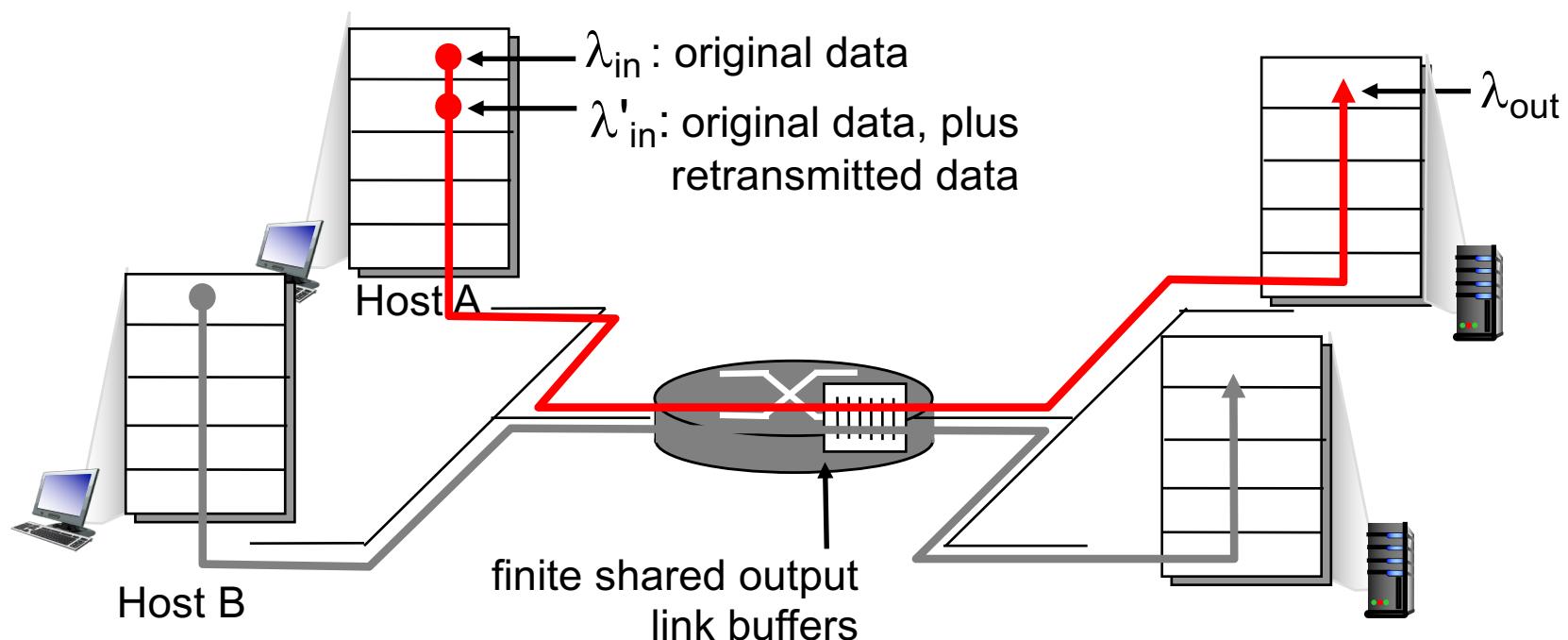
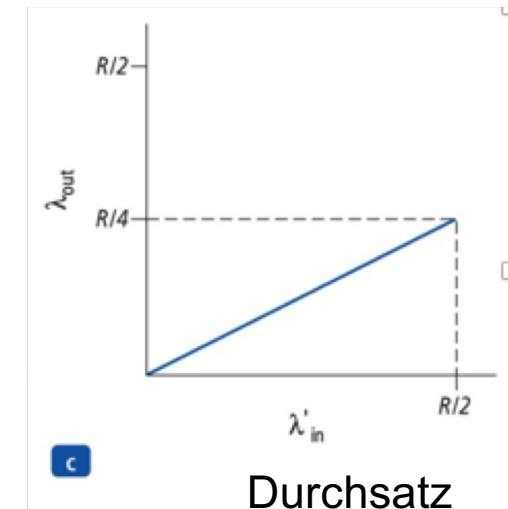
Stausituation Szenario 2

- Sender erkennt, dass ein Paket aufgrund eines Pufferüberlaufs verloren gegangen ist.
- Nur diese Pakete werden erneut übertragen.



Stausituation Szenario 2

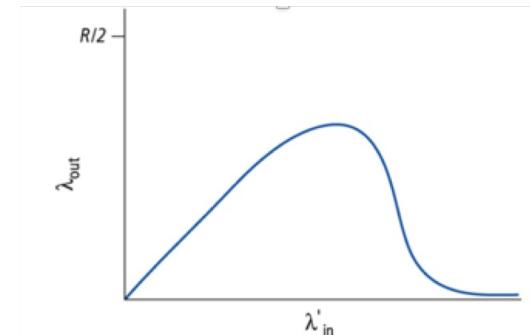
- Vorzeitige Timeouts: Somit werden auch Pakete erneut versendet, die noch in der Warteschlange liegen.
- Unnötige Übertragungswiederholungen verschwenden Bandbreite.



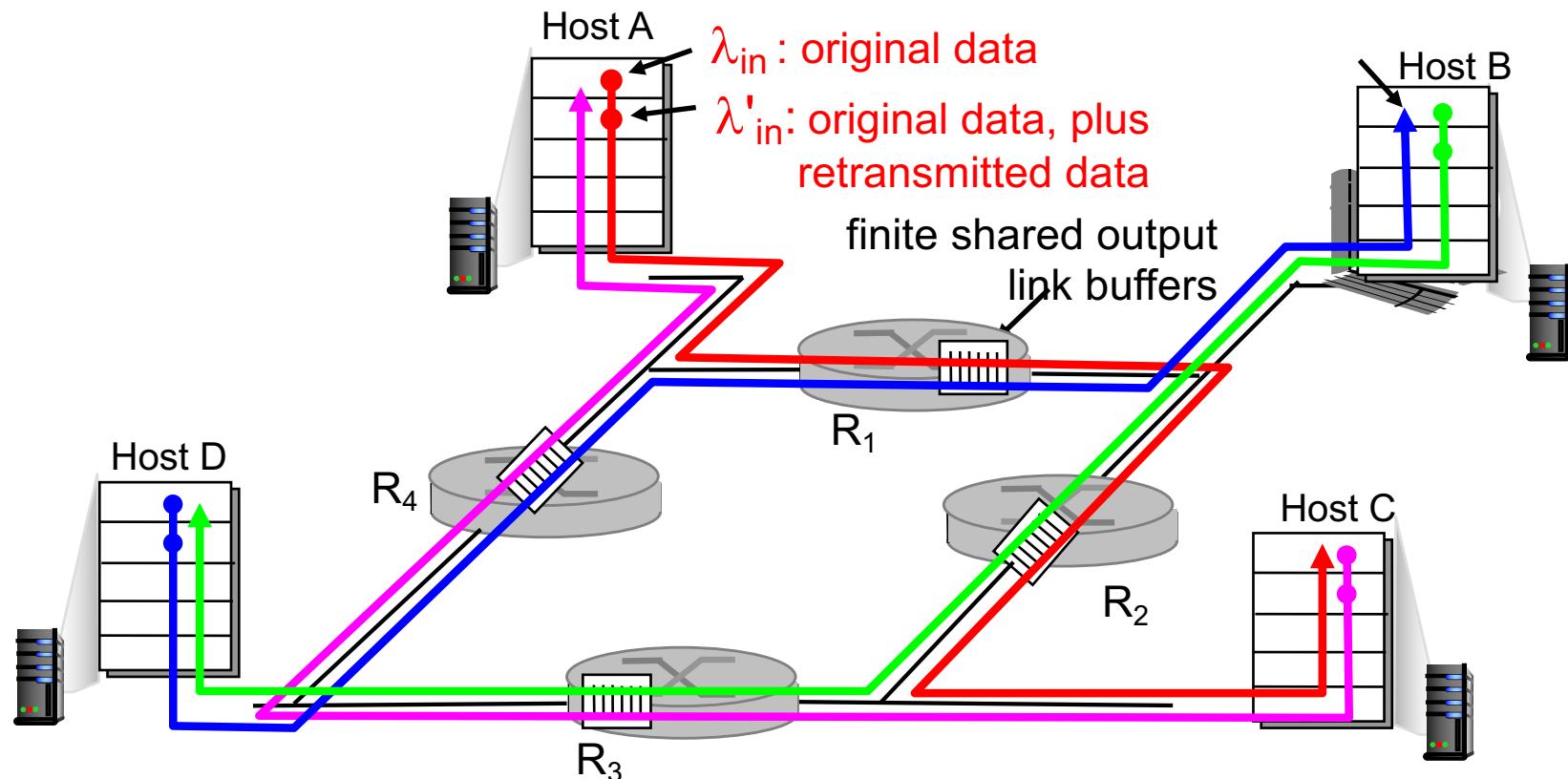
Stausituation Szenario 3

Situation: 4 Sender, Kommunikation über 2 hops, timeout & Retransmission

- Hohes Maß an “verschwendeter” Übertragungsarbeit. **Warum?**



Durchsatz bei Multihop Pfaden



Kosten von Staus

- Verlorene Pakete müssen wiederholt werden
 - Evtl. unnötiges Übertragen von Paketen, die in einem der folgenden Router verloren gehen
 - Evtl. unnötige Wiederholung von Paketen aufgrund von Verzögerungen
- Starke Verringerung der Übertragungsrate (“Durchsatz”)
(Tendenz: $\rightarrow 0$ bei dauerhafter Überlast)
- Große Paketverzögerungen
(Tendenz: $\rightarrow \infty$ bei dauerhafter Überlast)

Ansätze zur Überlastkontrolle

Ende-zu-Ende Überlastkontrolle:

- keine explizite Kommunikation mit dem Netzwerk über Überlast/Staukontrolle
- Stausituation wird gefolgert aus dem beobachteten Ende-zu-Ende-Verhalten (Paketverlust und Verzögerungen)
- von TCP so durchgeführt

Netzwerk-gesteuerte Überlastkontrolle:

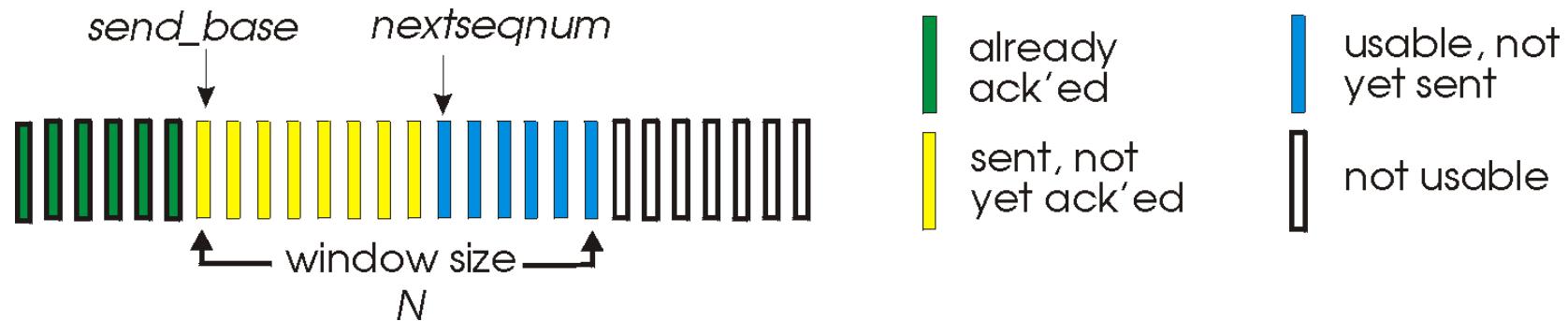
- Router liefern Feedback an die Endgeräte
 - 1 Bit zeigt Stau an (SNA, DECNet, ATM, TCP-ECN)
 - Explizite Datenrate, die ein Sender noch produzieren darf
- Zwei Arten der Rückmeldung
 - Die Netzwerkrouter verschicken Choke (Drossel) Pakete, die die Überlast eines Routers anzeigen.
 - Markierung der Überlast in den Paketen und Empfänger teilt die Info an den Sender mit
 - Lange Verzögerung

TCP Überlastkontrolle

- **Ende-zu-Ende Überlastkontrolle**
- **Ansatz:** Jeder Sender stellt seine Übertragungsrate in Abhängigkeit von der von ihm wahrgenommenen Überlast in Netz ein.
- Drei Fragen:
 - Wie begrenzt ein TCP Sender die Senderate?
 - Wie erkennt eine TCP Sender eine Überlastsituation?
 - Nach welchen Algorithmus wird das Sendetempo als Funktion über die erkannte Überlast eingestellt?

Begrenzung der TCP Senderate

- Übertragungsrate wird limitiert durch die Staufenstergröße (congestion window size) **Congwin** (zusätzlicher Parameter)



$$\text{LastByteSent} - \text{LastByteAcked} \leq \min \{\text{CongWin}, \text{RcvWin}\}$$

Für den Rest des Kapitels gehen wir davon aus, dass der TCP Empfangspuffer so groß ist, dass nur das CongWin die Senderate beschränkt (also keine Flusskontrolle)

Erkennung von Überlast

Erkenne der Überlast

- an Timeouts
- an drei doppelten ACKs
- Grund:
 - Überlast => Puffer laufen voll => Datagramme werden verworfen => Timeout oder drei doppelte ACKs

Generelle Algorithmus der Überlastkontrolle

- Jedes "neue" Ack ist ein Zeichen, dass die Übertragung problemlos gearbeitet hat.
- Erhöhe CongWin langsam
- **Selbsttaktend:** Die Frequenz, mit der die Pakete eintreffen, legen die Geschwindigkeit, mit der die CongWin und somit die Senderate erhöht wird, fest.
- Erkenne Überlast: Reduziere CongWin

TCP – Überlastkontrolle RFC 5681

Generelle Vorgehensweise: Austesten der verfügbaren Bandbreite

- Idealisiert: Sende so schnell wie möglich, ohne dass ein Paketverlust eintritt (**CongWin** möglichst groß)
 1. Starte mit kleinem **CongWin** – Wert
 2. Erhöhe **Congwin** langsam, bis ein Paketverlust auftritt
 3. Bei Paketverlust: Erniedrige **Congwin** stark und beginne wieder mit 2.

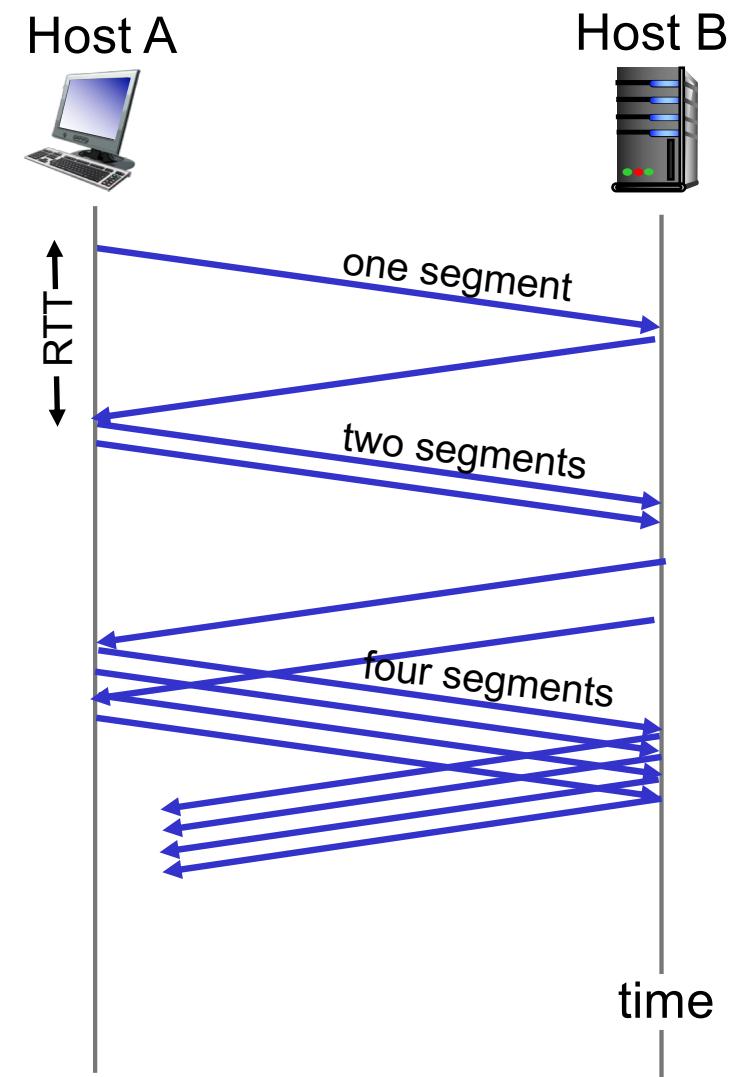
TCP – Überlastkontrolle: Slow Start

Wichtige Variablen:

- CongWin: Sendefenster für Überlast/Staukontrolle
- Threshold: Definiert für Congwin eine Schwelle zwischen "slow start"-Phase und Stauvermeidungs-Phase
- **Slow Start Algorithmus:**

```
initialize: CongWin = 1 MSS
for (each segment ACKed)
    CongWin++
until (loss event OR CongWin ≥ threshold)
```

- Start mit kleinen Wert
- Verdoppelung der Fenstergröße (pro RTT): Exponentieller Anstieg



Reaktion auf das Erkennen einer Überlast

Timeout

- **Threshold = CongWin / 2;**
- CongWin = 1 MSS;
- Erneuter Slow Start;

3 duplicate ACKs: TCP RENO

- CongWin = CongWin / 2;
- Wechsel in den Congestion Avoidance Modus (Stauvermeidungs-Phase)

3 duplicate ACKs: TCP Tahoe

- **Threshold = CongWin / 2;**
- CongWin = 1 MSS;
- Erneuter Slow Start;

Congestion Avoidance - Stauvermeidung

Algorithmus:

```
/* slowstart is over      */
/* Congwin ≥ threshold */

Until (loss event) {
    every CongWin segments ACKed:1
        CongWin++
}
threshold = Congwin/2
Congwin = 1
perform slowstart*
```

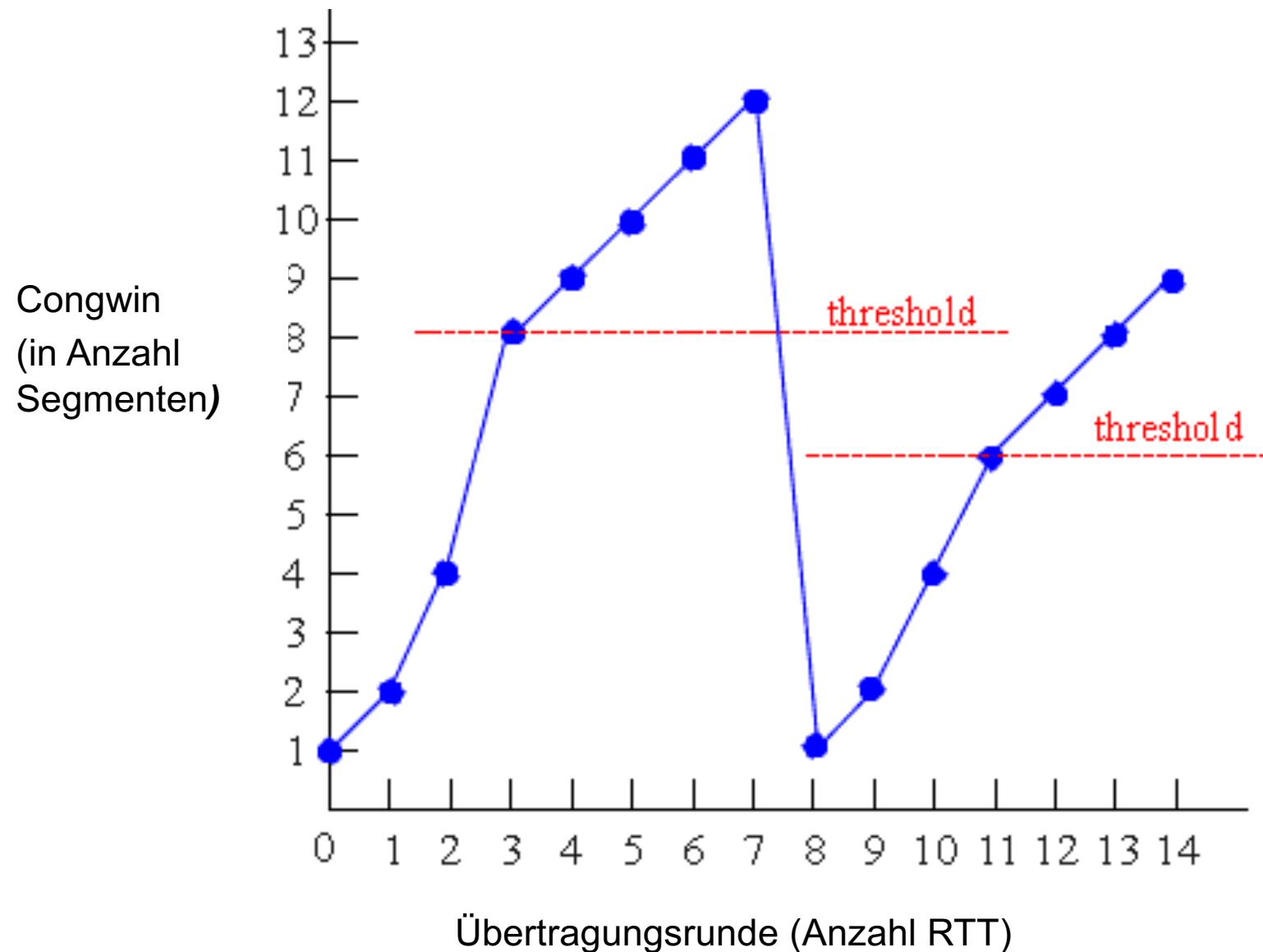
AIMD-Prinzip:

- additive increase, multiplicative decrease
- erhöhe Congwin um 1 pro RTT
- erniedrige threshold um den Faktor 2 bei einem Verlust-Ereignis

* TCP Reno überspringt Slowstart nach drei Duplikat-ACKs (“fast retransmit / fast recovery”) und setzt nur Congwin = threshold

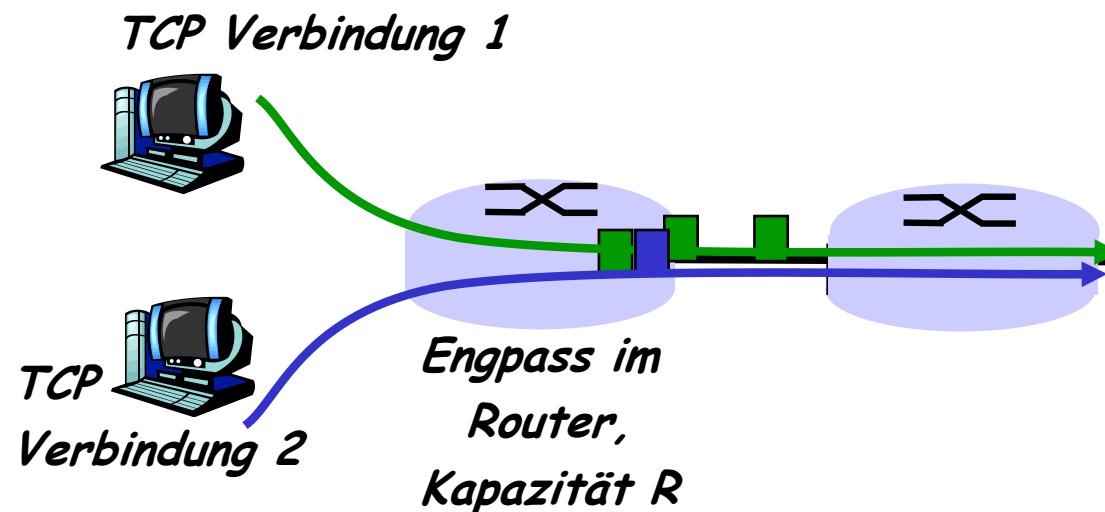
¹ Nicht pro Ack sondern nachdem das Ack für ConWin viele Segmente eingetroffen ist.

TCP – Überlastkontrolle: Beispiel



TCP Fairness

Ziel: Wenn N TCP-Verbindungen sich denselben beschränkten Netzwerk-Pfad teilen, sollte jede $1/N$ der Kapazität des Pfades erhalten

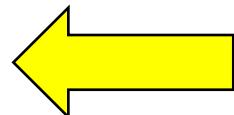


Trick (\rightarrow HTTP): Baue parallel mehrere TCP-Verbindungen auf!

Kapitel 4: Transportschicht

Gliederung

- Dienste und Prinzipien auf der Transportschicht
- Multiplexen und Demultiplexen von Anwendungen
- Verbindungsloser Transport: UDP
- Prinzipien des zuverlässigen Datentransfers
- Verbindungsorientierter Transport: TCP
- TCP – Überlastkontrolle (Staukontrolle)
- Zusammenfassung



Textbuch zu diesem Kapitel: J. Kurose & K. Ross: Computernetzwerke – Der Top-Down-Ansatz, Kapitel 3

Folien und Abbildung teilweise aus:

J. Kurose & K. Ross: Computernetzwerke – Der Top-Down-Ansatz

Zusammenfassung

