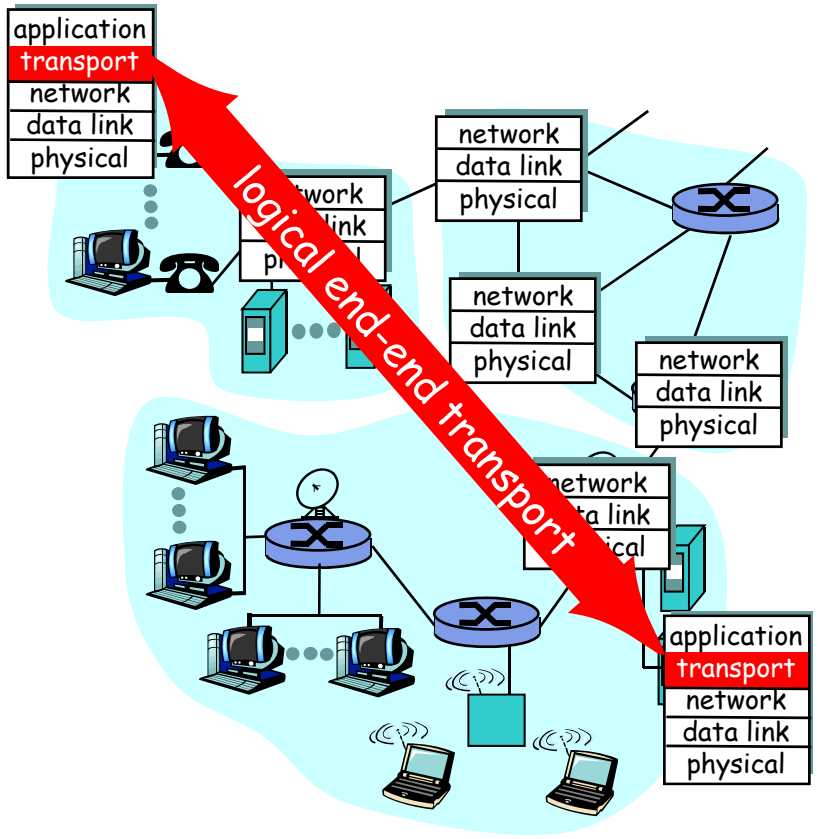
Kapitel 3
Transportschicht

1. Dienste und Prinzipien auf der Transportschicht
2. Multiplexen und Demultiplexen von Anwendungen
3. Verbindungsloser Transport: UDP
4. Prinzipien des zuverlässigen Datentransfers
5. Verbindungsorientierter Transport: TCP
6. TCP – Staukontrolle

Transport-Dienste

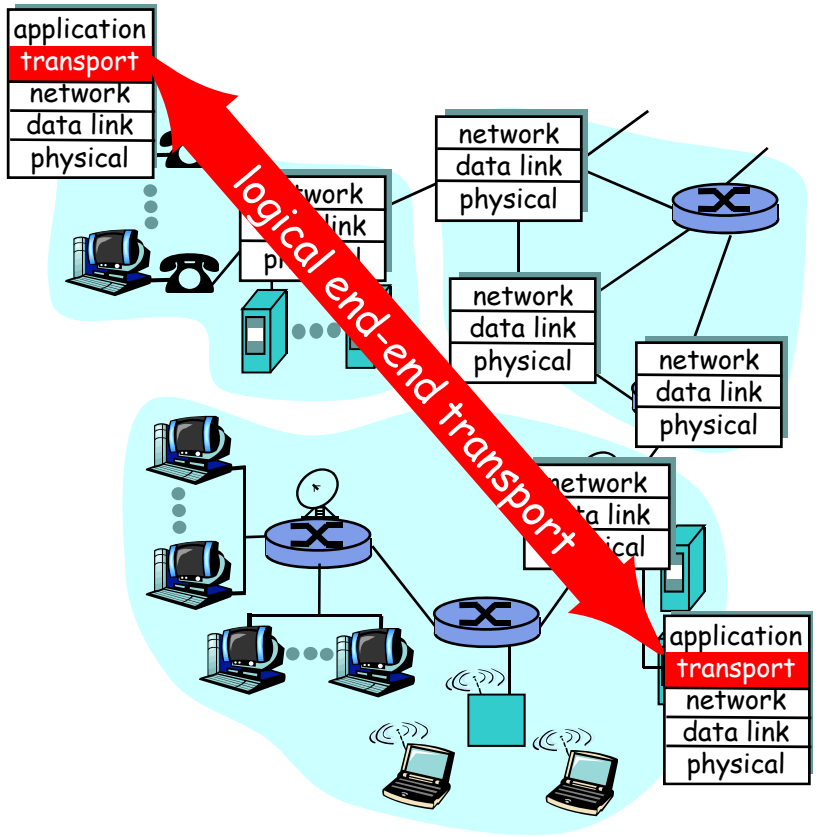
- Stellen eine *logische Kommunikation* zwischen Anwendungsprozessen her , die auf unterschiedlichen Hosts laufen
- Transportprotokolle laufen nur auf den Endsystemen
- Transportdienste / Netzwerkschichtdienste:
- *Netzwerkschicht:* Datentransfer zwischen Hosts (Rechnern)
- *Transportschicht:* Datentransfer zwischen Prozessen
  - Nutzt und erweitert Dienste der Netzwerkschicht



Transportschicht-Protokolle

Internet-Transportdienste:

- Verlässlicher, reihenfolge-erhaltender Punkt-zu-Punkt Transport: TCP
  - Staukontrolle
  - Flusskontrolle
  - Verbindungsaufbau
- Unzuverlässiger ("best-effort"), ungeordneter Punkt-zu-Punkt und Multicast-Transport: UDP
- Nicht verfügbare Dienste:
  - Realzeit
  - Bandbreitengarantie
  - Verlässlicher Multicast



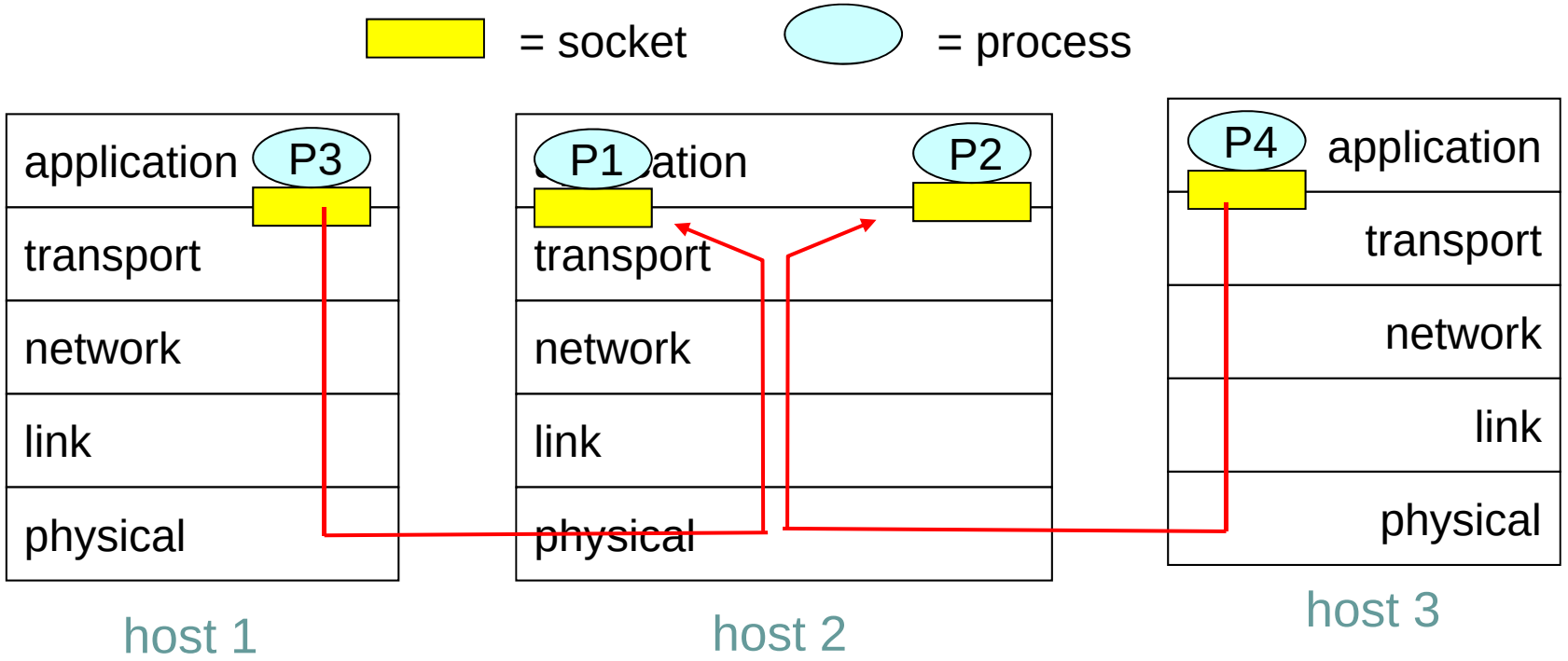
Kapitel 3
Transportschicht

1. Dienste und Prinzipien auf der Transportschicht
2. Multiplexen und Demultiplexen von Anwendungen
3. Verbindungsloser Transport: UDP
4. Prinzipien des zuverlässigen Datentransfers
5. Verbindungsorientierter Transport: TCP
6. TCP – Staukontrolle

Multiplexen/Demultiplexen

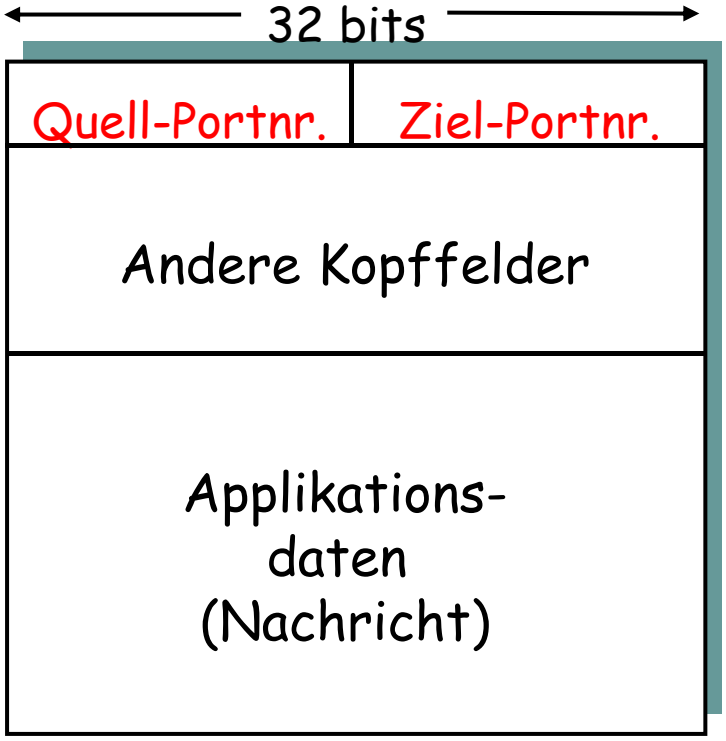
Multiplexen:
Einsammeln der Daten von mehreren Anwendungsschicht-Prozessen, Einpacken der Daten mit Steuerinformationen (→ Segmente)

Demultiplexen:
Abliefern der empfangenen Segmente / Daten beim *richtigen* Anwendungsschicht-Prozess



Wie funktioniert “Demultiplexen”?

- Der Empfänger erhält IP-Datagramme
  - Jedes IP-Datagram enthält eine IP-Quelladresse und IP-Zieladresse
  - Jedes IP-Datagramm transportiert genau ein Segment der Transportschicht
  - Jedes Segment hat eine Quell-Portnummer und Ziel-Portnummer
- Der Empfänger benutzt die IP-Adressen und Port-Nummern, um das Segment dem korrekten Socket zu übergeben



TCP/UDP Segment Format

Verbindungsloses Demultiplexen (UDP)

- UDP-Sockets werden eindeutig identifiziert durch (IP-Adresse, Portnummer)
- Aktionen beim Empfang eines UDP-Segments:
  - Auswertung der Zielportnummer im UDP-Segment
  - Weiterleitung des UDP-Segments zum Socket mit dieser Portnummer (falls vorhanden)

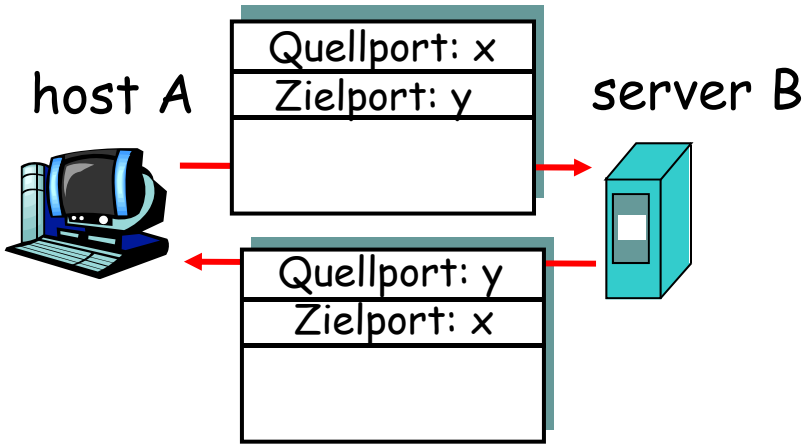
→ Segmente mit unterschiedlicher Quell-IP-Adresse / Quellportnummer werden bei gleicher Zielportnummer an dasselbe UDP-Socket weitergeleitet

- Wozu dient bei UDP die Quellportnummer?

Verbindungsorientiertes Demultiplexen (TCP)

- TCP-Sockets 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!

Multiplexen/Demultiplexen: Beispiele



Port-Nutzung: Einfache Anwendung

TCP-Port-Nutzung:
Web Server

TCP-/UDP-Portnummern

- Welche Portnummern 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 TCP-Ports:
  - 20+21: FTP
  - 23: Telnet
  - 25: SMTP
  - 53: DNS
  - 80: HTTP
  - 110: POP3
  - 443: HTTPS
  - …
- IT-Sicherheit:
  Welche Informationen können „Port-Scanner“ liefern?

http://www.iana.org/assignments/port-numbers

Kapitel 3
Transportschicht

1. Dienste und Prinzipien auf der Transportschicht
2. Multiplexen und Demultiplexen von Anwendungen
3. Verbindungsloser Transport: UDP
4. Prinzipien des zuverlässigen Datentransfers
5. Verbindungsorientierter Transport: TCP
6. TCP – Staukontrolle

UDP: User Datagram Protocol [RFC 768]

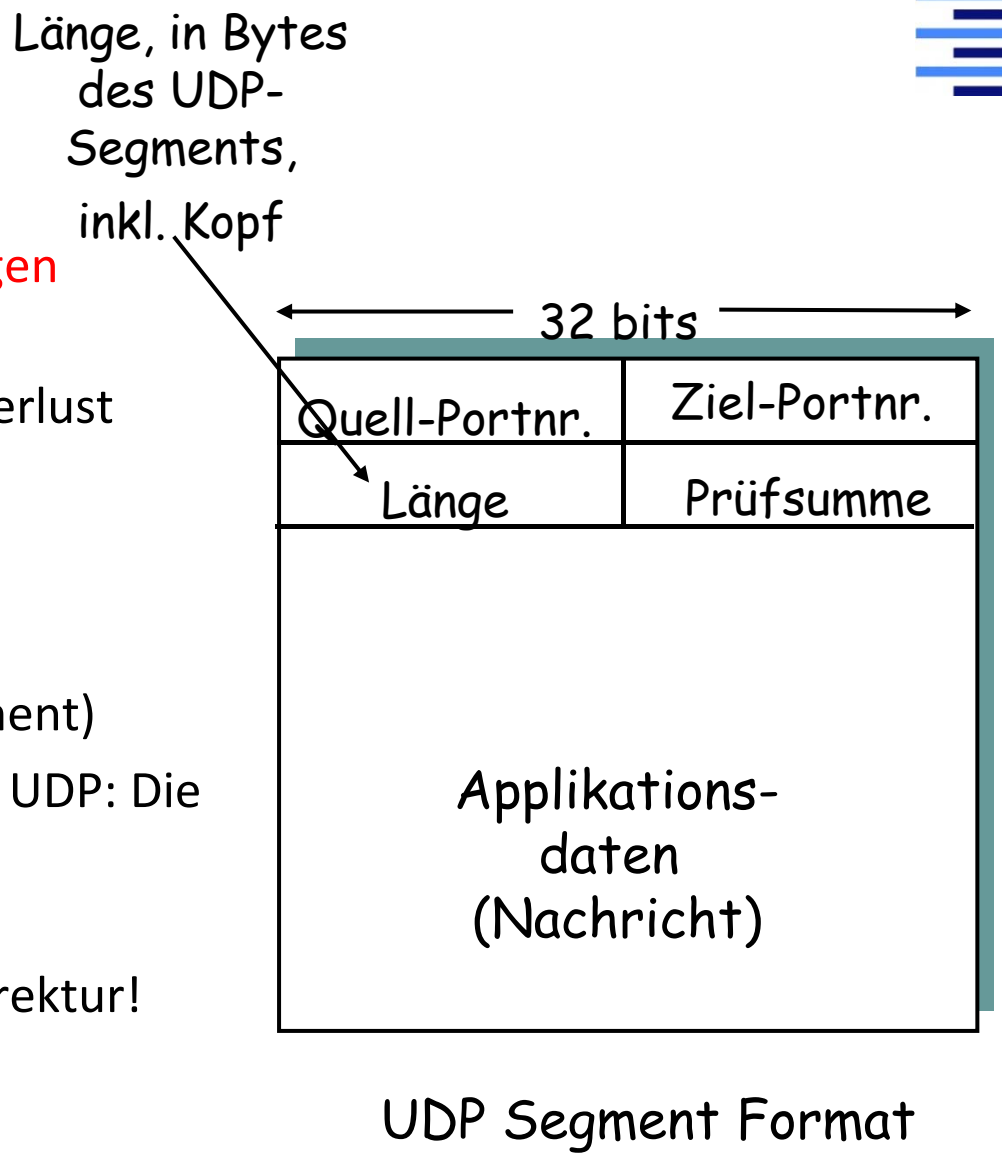
- "Nacktes" Internet Transport Protokoll
- "Best effort" Dienst, UDP Segmente können:
  - verloren gehen
  - in falscher Reihenfolge
  - oder doppelt geliefert werden
- *Verbindungslos:*
  - kein "handshaking" zwischen UDP-Sender und Empfänger
  - Jedes UDP-Segment wird unabhängig von anderen transportiert

Warum gibt es UDP?

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

UDP: mehr

- Oft für Multimedia-Anwendungen (Streaming) verwenden
  - 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!



UDP Segment Format

UDP-Prüfsumme

Ziel: Fehlerentdeckung im übertragenen Segment
(z.B. falsche Bits)

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.

    Könnten da doch noch Fehler sein ? ….

UDP-Prüfsumme: Beispiel (vereinfacht)

Sender

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

1. Addition der beiden ersten:
   …0110
   …0101
   …1011

2. Addition des dritten:
   …1011
   …1111
   …1010

3. Einerkomplement-Bildung:
   …1010 → …0101

4. Speichern der Prüfsumme:
   …0101

Empfänger

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

1. Addition der beiden ersten:
   …0110
   …0101
   …1011

2. Addition des dritten:
   …1011
   …1111
   …1010

3. Addition der Prüfsumme:
   …1010
   …0101
   …1111 ←

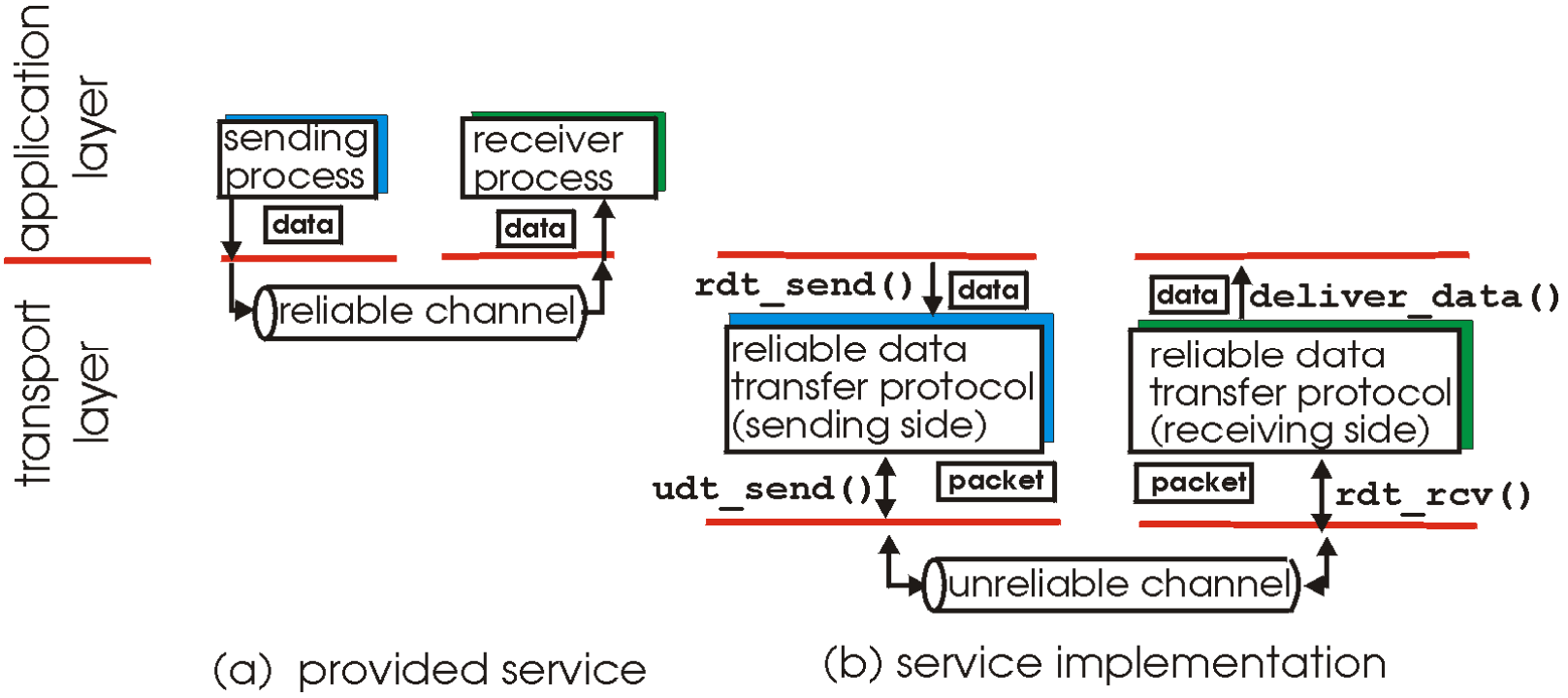
Wenn Ergebnis != FF → Fehler!

Kapitel 3
Transportschicht

1. Dienste und Prinzipien auf der Transportschicht
2. Multiplexen und Demultiplexen von Anwendungen
3. Verbindungsloser Transport: UDP
4. Prinzipien des zuverlässigen Datentransfers
5. Verbindungsorientierter Transport: TCP
6. TCP – Staukontrolle

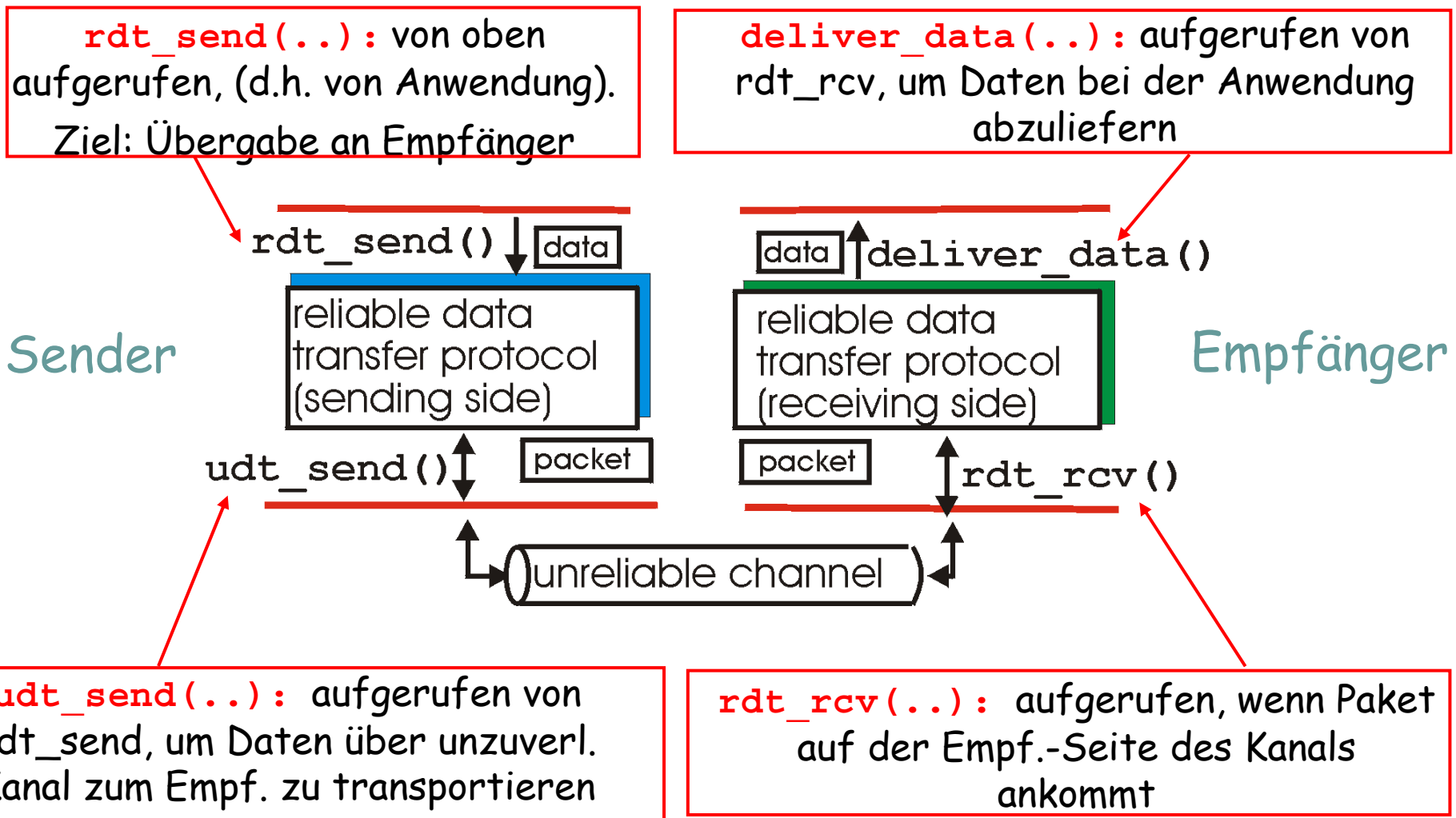
Prinzipien des zuverlässigen Datentransfers

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



- Die Eigenschaften des unzuverlässigen Kanals bestimmen die Komplexität des zuverlässigen Kommunikationsprotokolls (rdt)

Zuverlässiger Datentransfer: Grundmodell

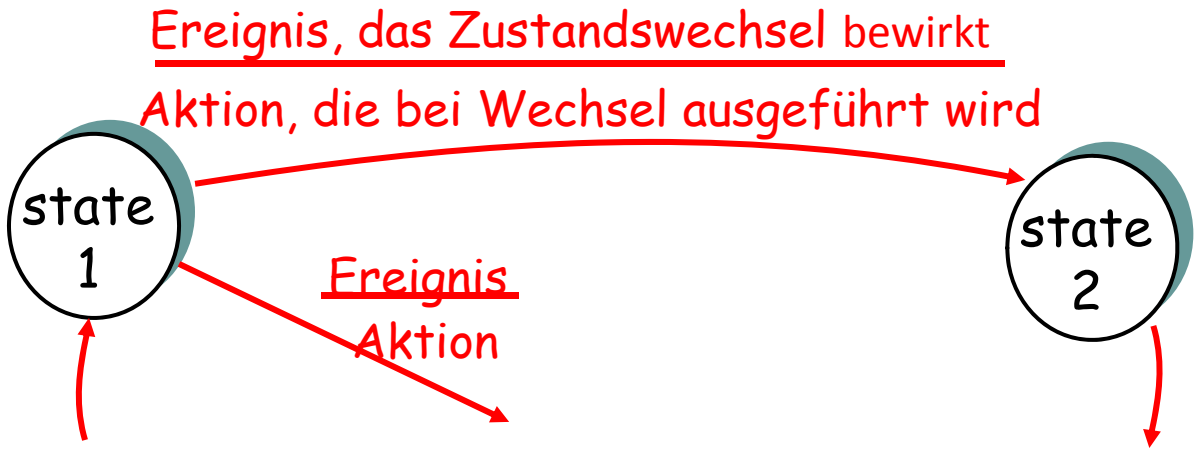


Weitere Vorgehensweise

Wir werden ..

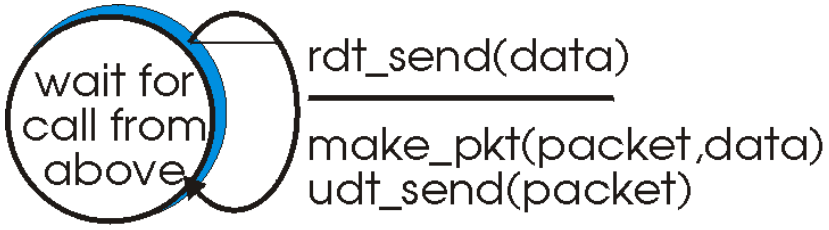
- das Protokoll des zuverlässigen Datentransfers (**rdt**) schrittweise entwickleln
- nur unidirektionalen Datentransfer betrachten
  - aber Steuerinformationen fließen in beide Richtungen!
- einen endlichen Automaten (FSM) benutzen, um Sender und Empfänger zu spezifizieren:

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

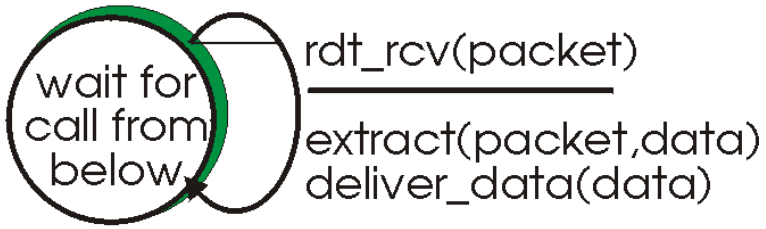


rdt 0.1: zuverlässiger Transfer über zuverl. Kanal

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



(a) rdt 0.1 : sending side



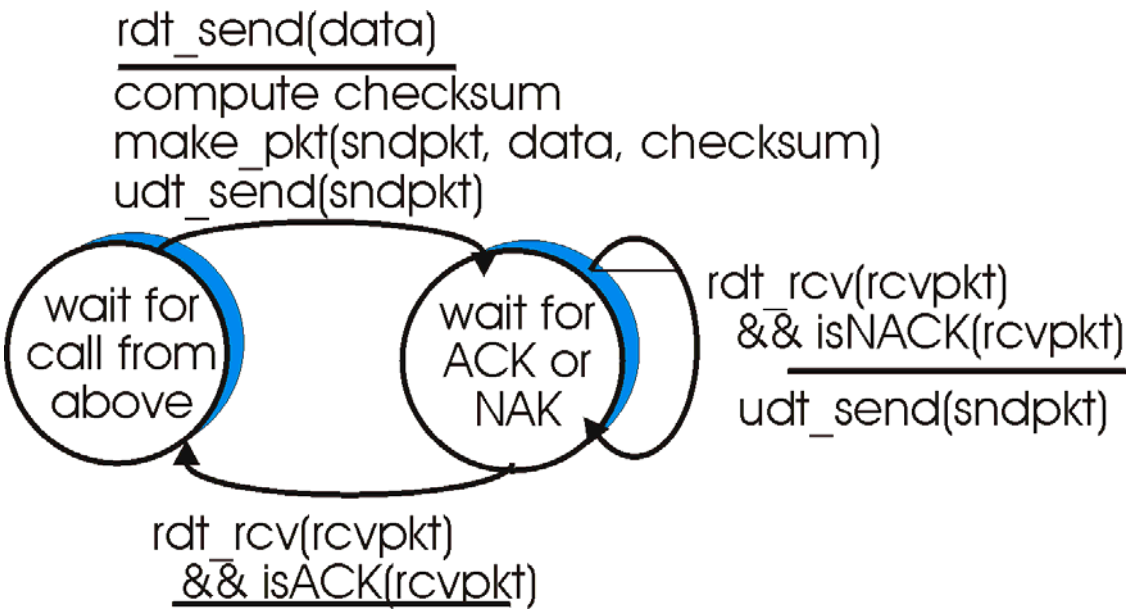
(b) rdt 0.1 : receiving side

rdt 0.2: Kanal mit Bitfehlern

- Kanal verfälscht Bits des Pakets
  - aber ohne Totalverlust eines Pakets
- *Die* Frage: Wie bei Fehlern reagieren?
  - *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
  - Sender wiederholt Übertragung nach Empfang von NAK
  - Menschliches Szenario mit ACKs, NAKs ?
- Neue Mechanismen **`rdt 0.2`** :
  - Fehlerentdeckung
  - Empfänger-Rückmeldung: Kontroll-Nachrichten (ACK, NAK) vom Empfänger an den Sender

„ARQ-Protokolle“
(Automatic Repeat reQuest)

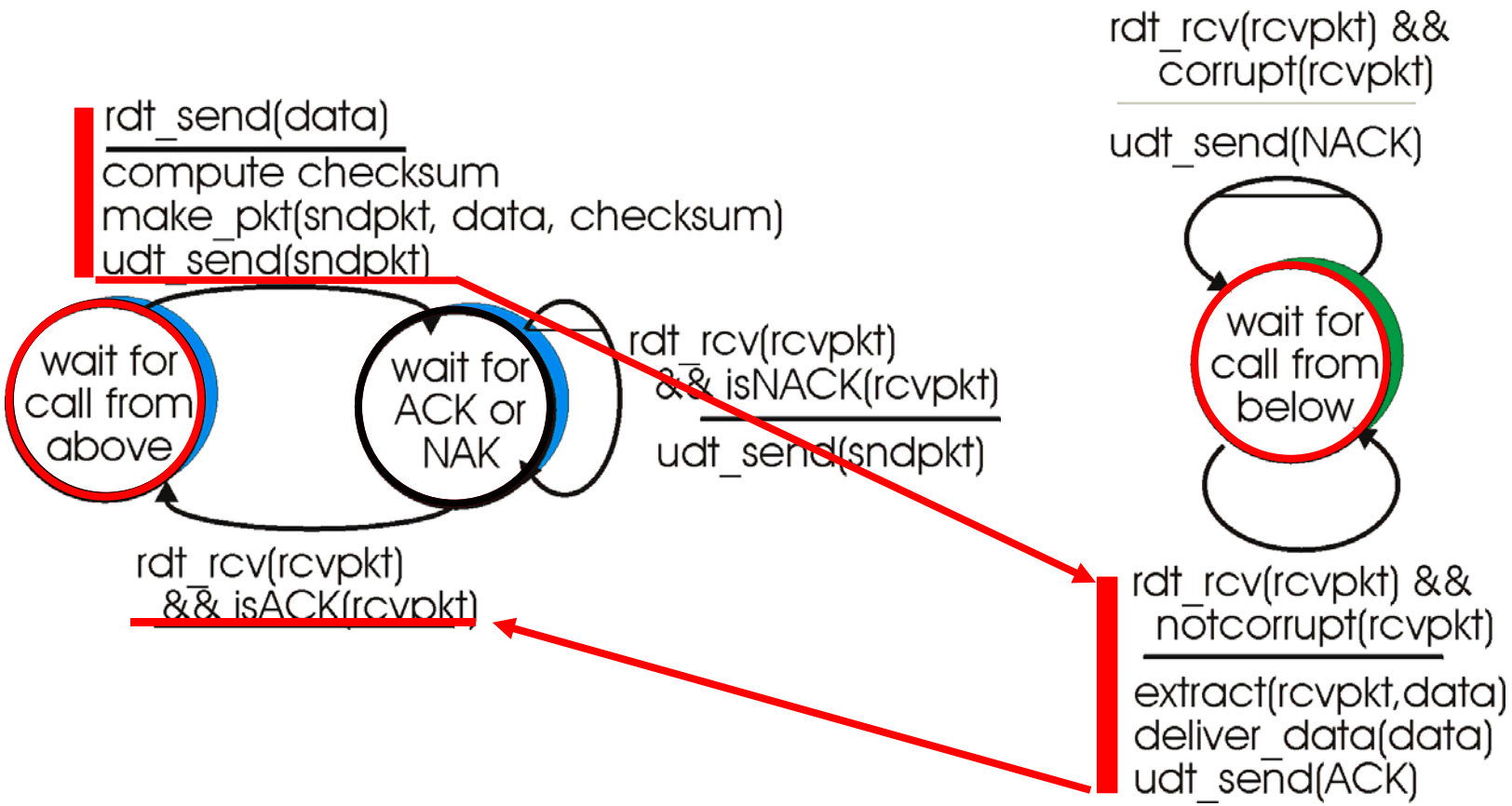
rdt 0.2: FSM-Spezifikation



Sender-FSM

Empfänger-FSM

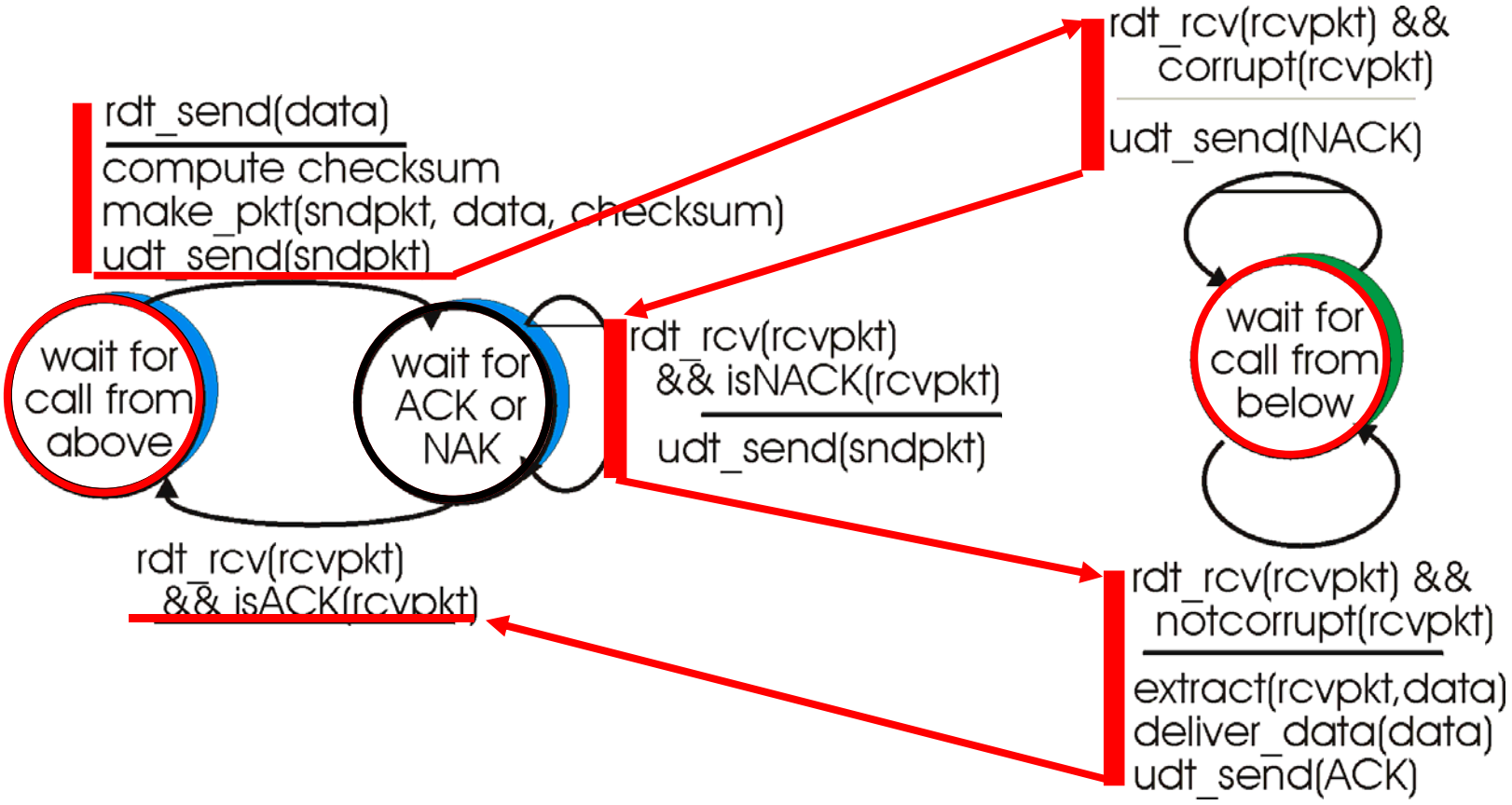
rdt 0.2: In Aktion (ohne Fehler)



Sender FSM

Empfänger FSM

rdt 0.2: In Aktion (Fehlerszenario)



Sender FSM

Empfänger FSM

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 das könnte eine unerkannte Wiederholung korrekt übertragener Pakete ("Duplikate") bewirken !

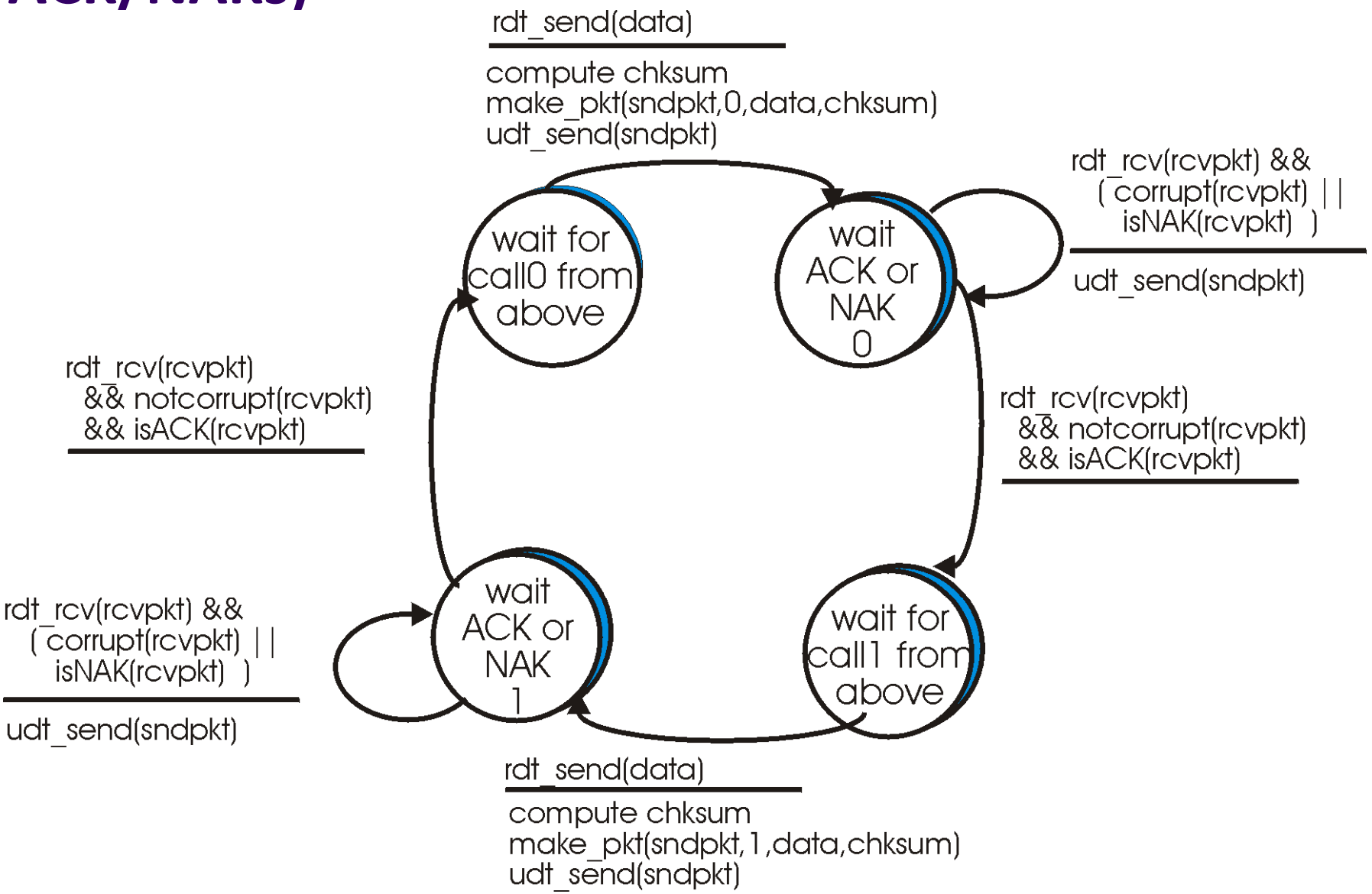
→ Duplikate erkennen!

- Sender fügt *Sequenznummer* bei jedem Paket hinzu
- Sender wiederholt aktuelles Paket, wenn ACK/NAK verfälscht ist
- Empfänger verwirft Duplikate (liefert sie nicht bei der Anwendung ab)

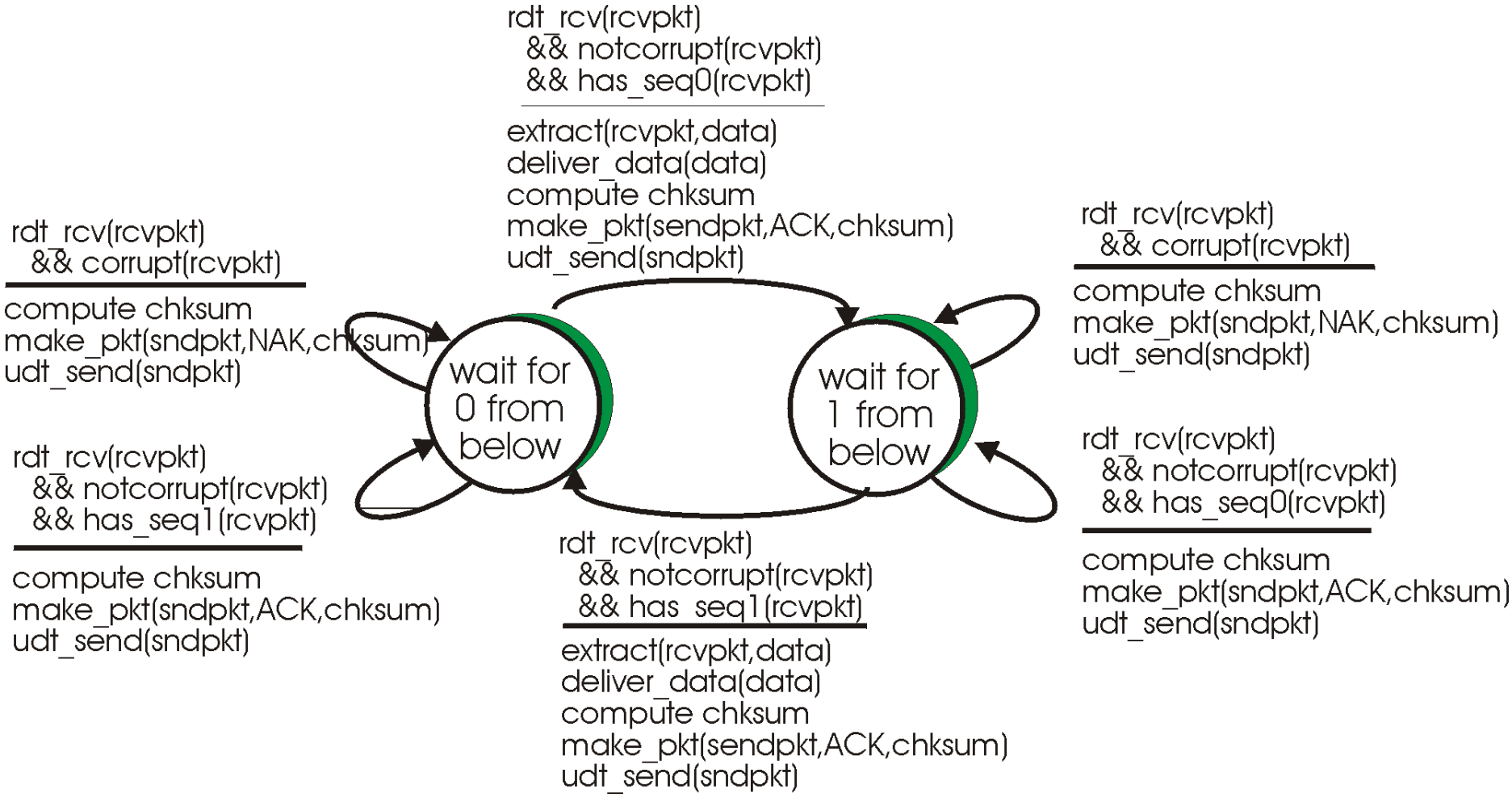
stop and wait

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
  - Zustand muss erinnern, ob aktuelles Paket die Sequenznummer 0 oder 1 hat

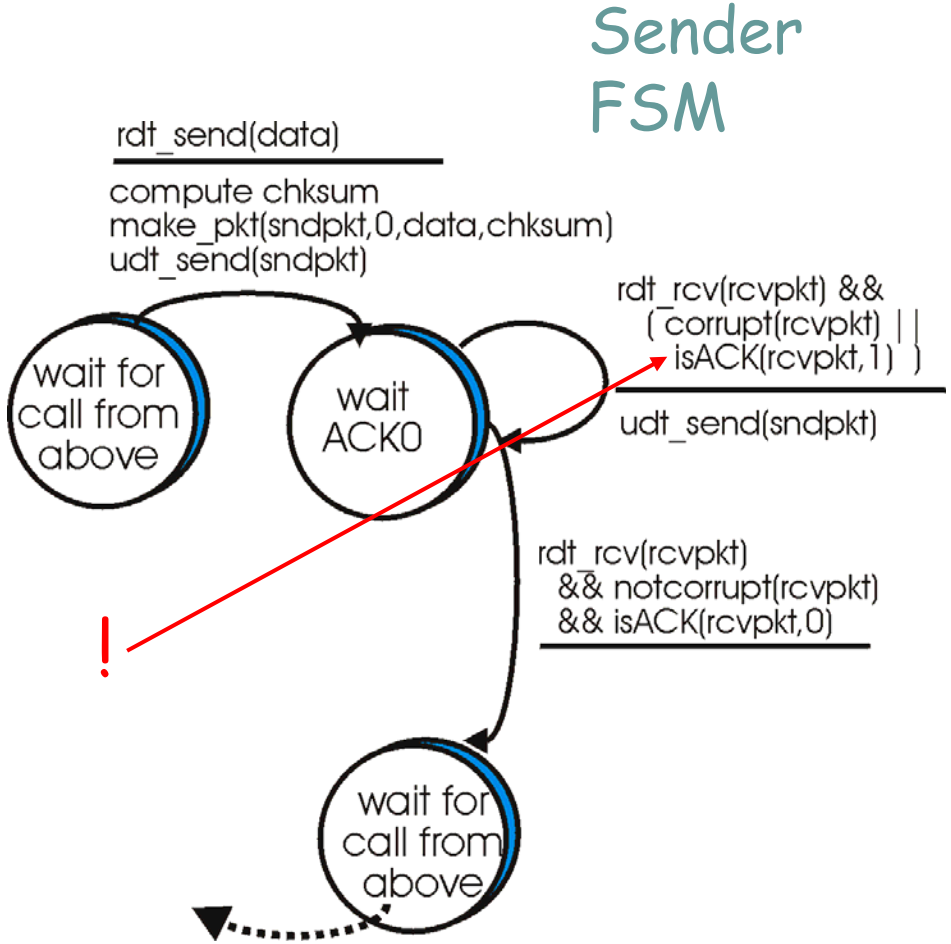
Empfänger:

- … 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

- 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



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

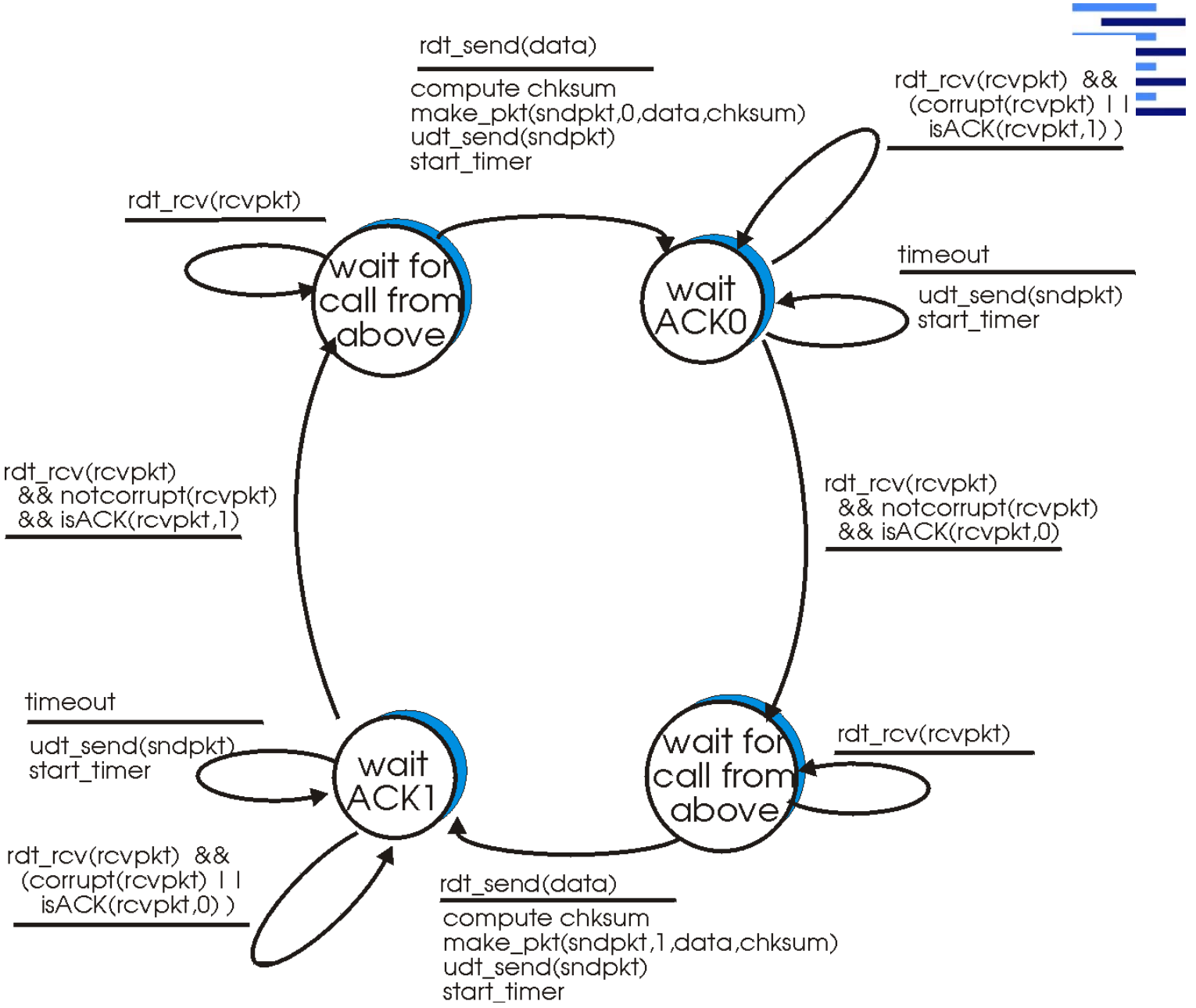
Frage:

- Wie erkennen und behandeln wir Verluste ?

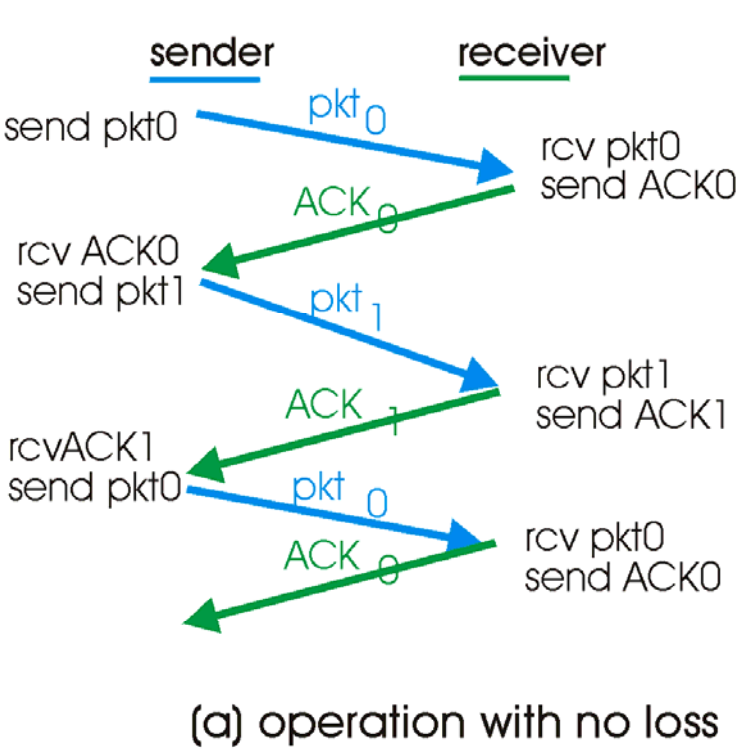
Lösungsversuch: Sender wartet eine "vernünftige" Zeitdauer auf das ACK

- 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

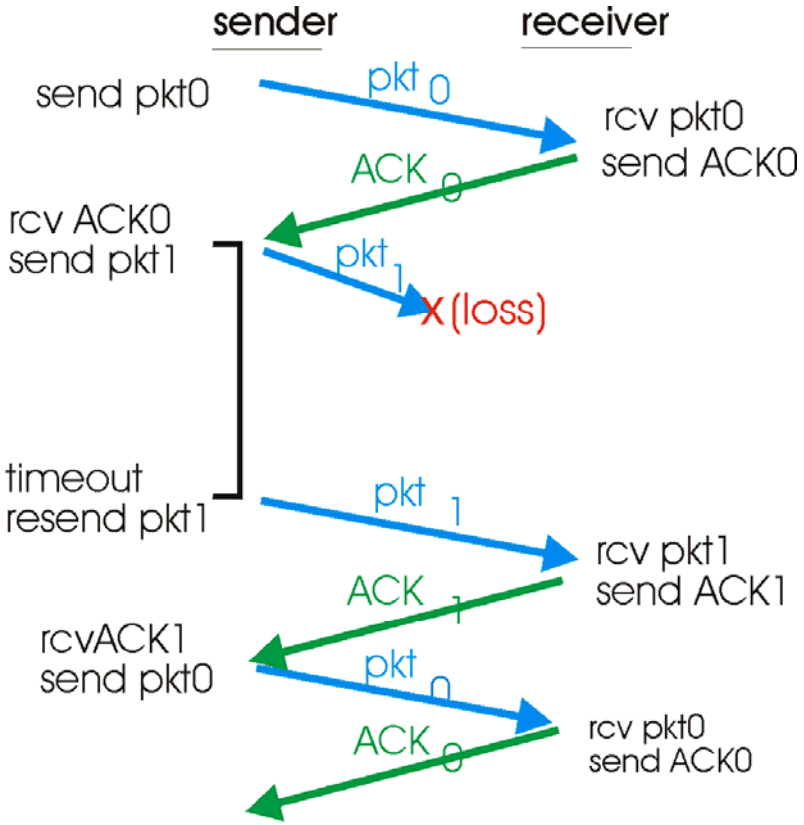
rdt 0.3 Sender



rdt 0.3 in Aktion

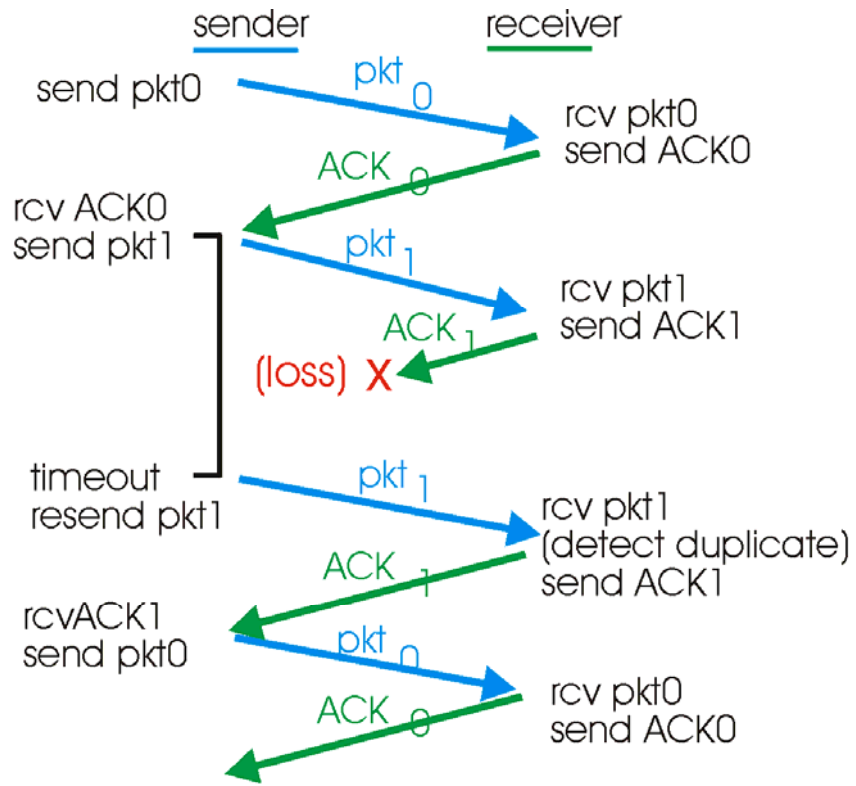


(a) operation with no loss

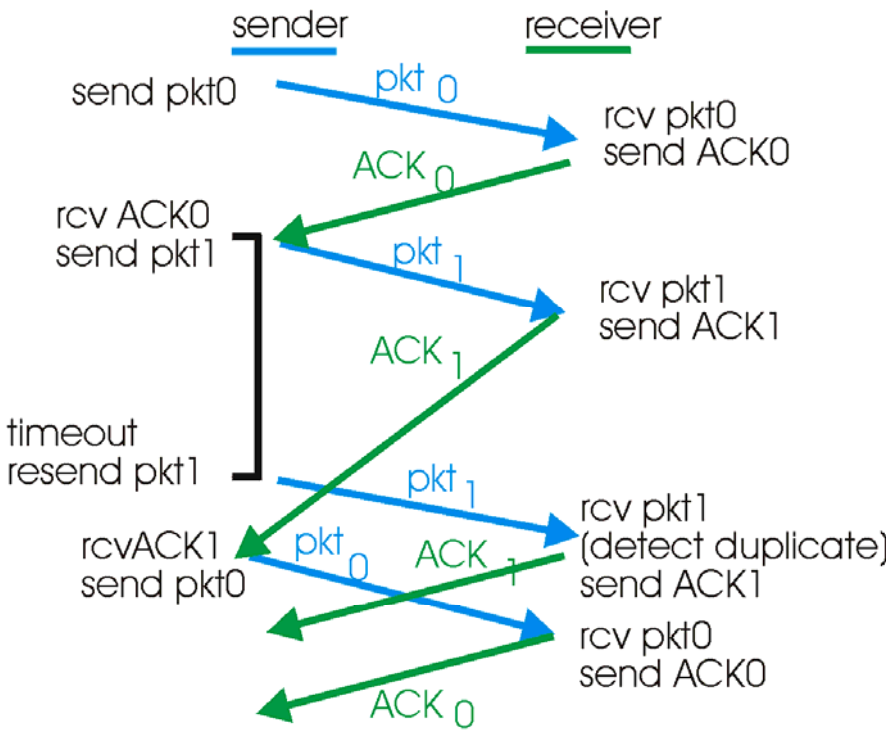


(b) lost packet

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
  Übertragungsverzögerung $T_{transmit}$?

$$T_{transmit} = \frac{L}{R} = \frac{8000\ bit}{10^{**}9\ bit/s} = 0{,}000008\ s = 0{,}008\ ms$$

Bei einer Ausbreitungsverzögerung T_{prop} = 15 ms ergibt sich für die Zeit RTT, bis die Quittung für ein Paket vorliegt („Round Trip Time“)

RTT = 15 ms + 15 ms = 30 ms (ACK-Sendeverzögerung hier vernachlässigbar)

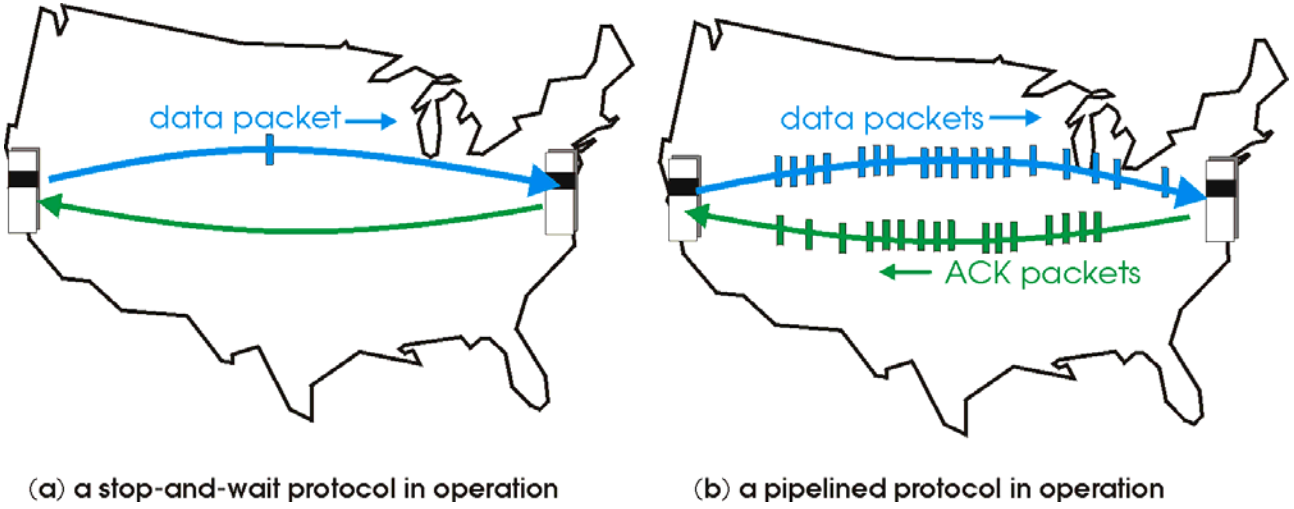
$$\text{Nutzungsgrad} = \frac{\text{Bruchteil der Zeit, die Sender sendet}}{} = \frac{0{,}008\ ms}{30{,}008\ ms} = 0{,}000267 = 0{,}0267\ \%$$

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

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



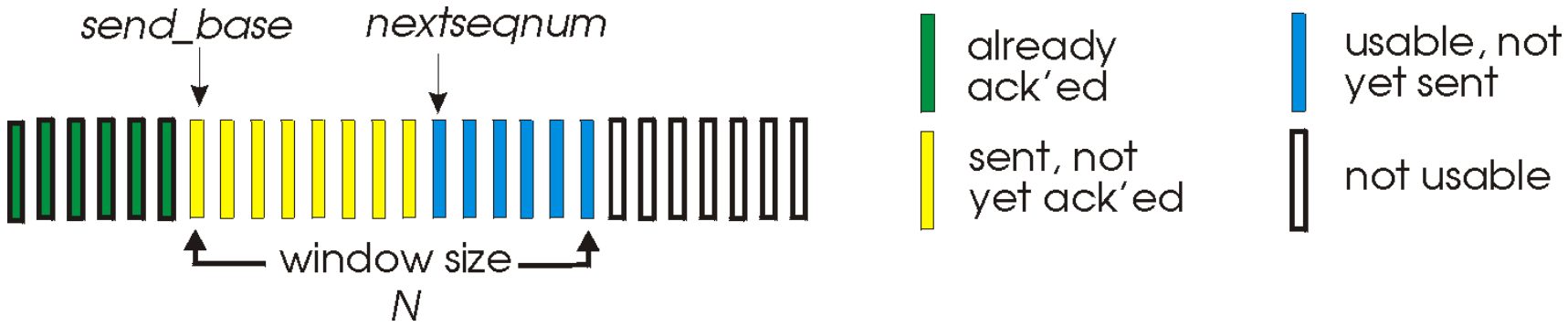
(a) a stop-and-wait protocol in operation (b) a pipelined protocol in operation

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

Go-Back-N (rdt 1.0)

Sender:

- k-bit Sequenznummer im Paket-Header
- Fenster (“window”) von bis zu N aufeinanderfolgend nicht bestätigten Pakten erlaubt

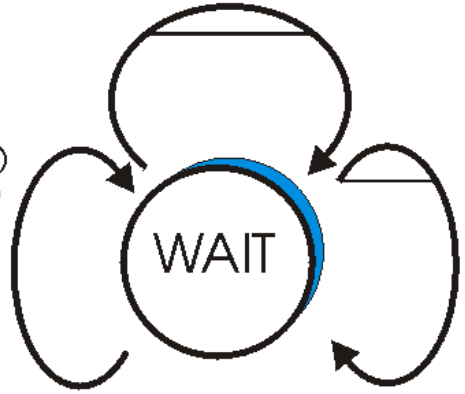


- ACK(n): bestätigt alle Pakete bis einschließlich Sequenznummer n - “Kumulatives ACK”
- Timer für das älteste nicht bestätigte Paket (send_base n)
- *timeout(n):* Sendewiederholung von Paket n und aller Pakete mit höherer Sequenznummer im Fenster

Go-Back-N: Erweiterte FSM des Senders

rdt_send(data)

```
if (nextseqnum < base+N) {
   compute chksum
   make_pkt(sndpkt(nextseqnum)),nextseqnum,data,chksum)
   udt_send(sndpkt(nextseqnum))
   if (base == nextseqnum)
     start_timer
   nextseqnum = nextseqnum + 1
   }
else
  refuse_data(data)
```



rdt_rcv(rcv_pkt) && notcorrupt(rcvpkt)

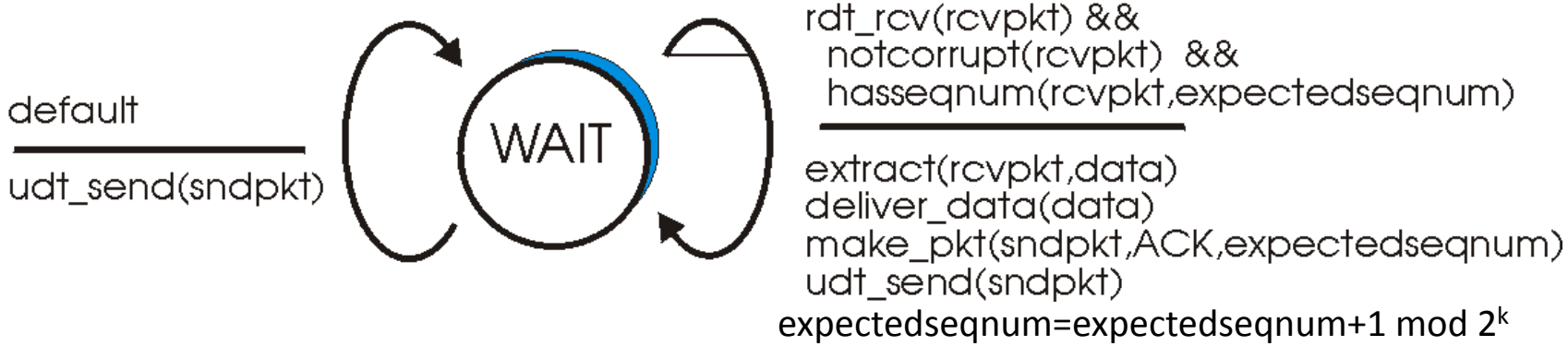
```
if (getacknum(rcvpkt)+1 > base)
   base=getacknum(rcvpkt)+1
   if (base==nextseqnum)
       stop_timer
   else
       start_timer
```

timeout

```
start_timer
udt_send(sndpkt(base))
udt_send(sndpkt(base+1)
……
udt_send(sndpkt(nextseqnum-1))
```

Für alle Additionsoperationen gilt: mod 2^k

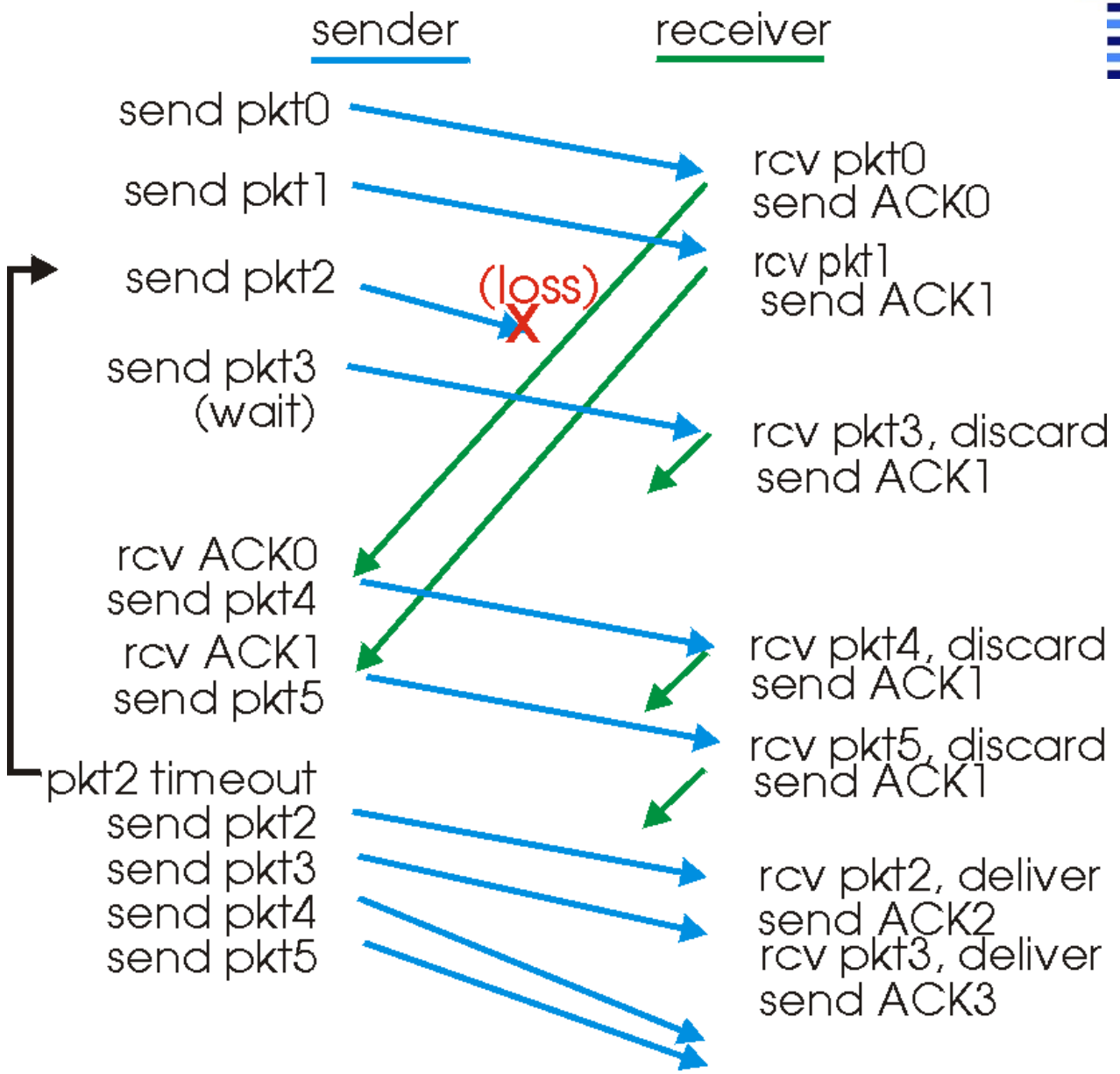
Go-Back-N: Erweiterte FSM des Empfängers



Der Empfänger ist einfacher:

- 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

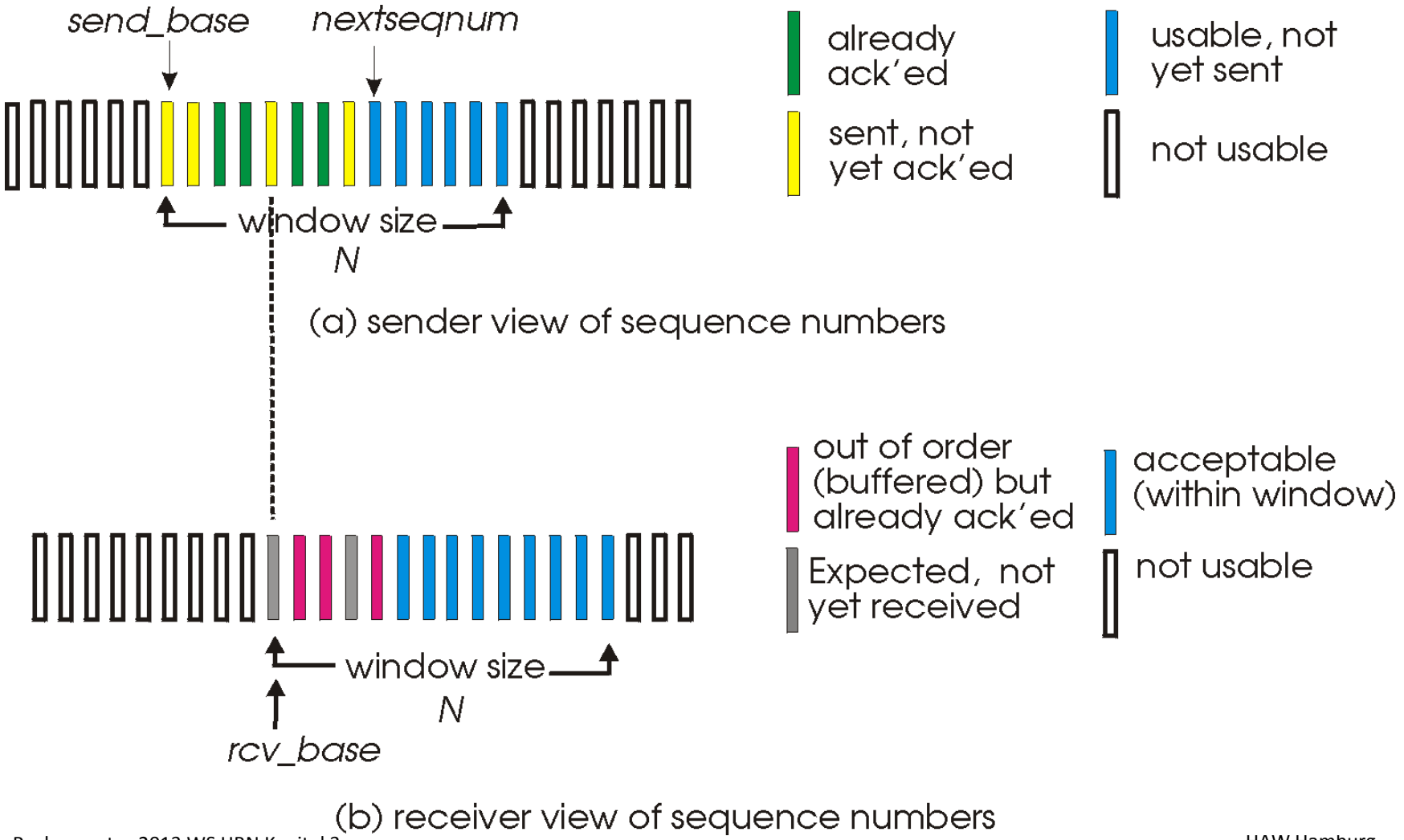
Go-Back-N in Aktion



Selective Repeat (rdt 2.0)

- 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: Sender / Empfängerfenster



Selective repeat

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

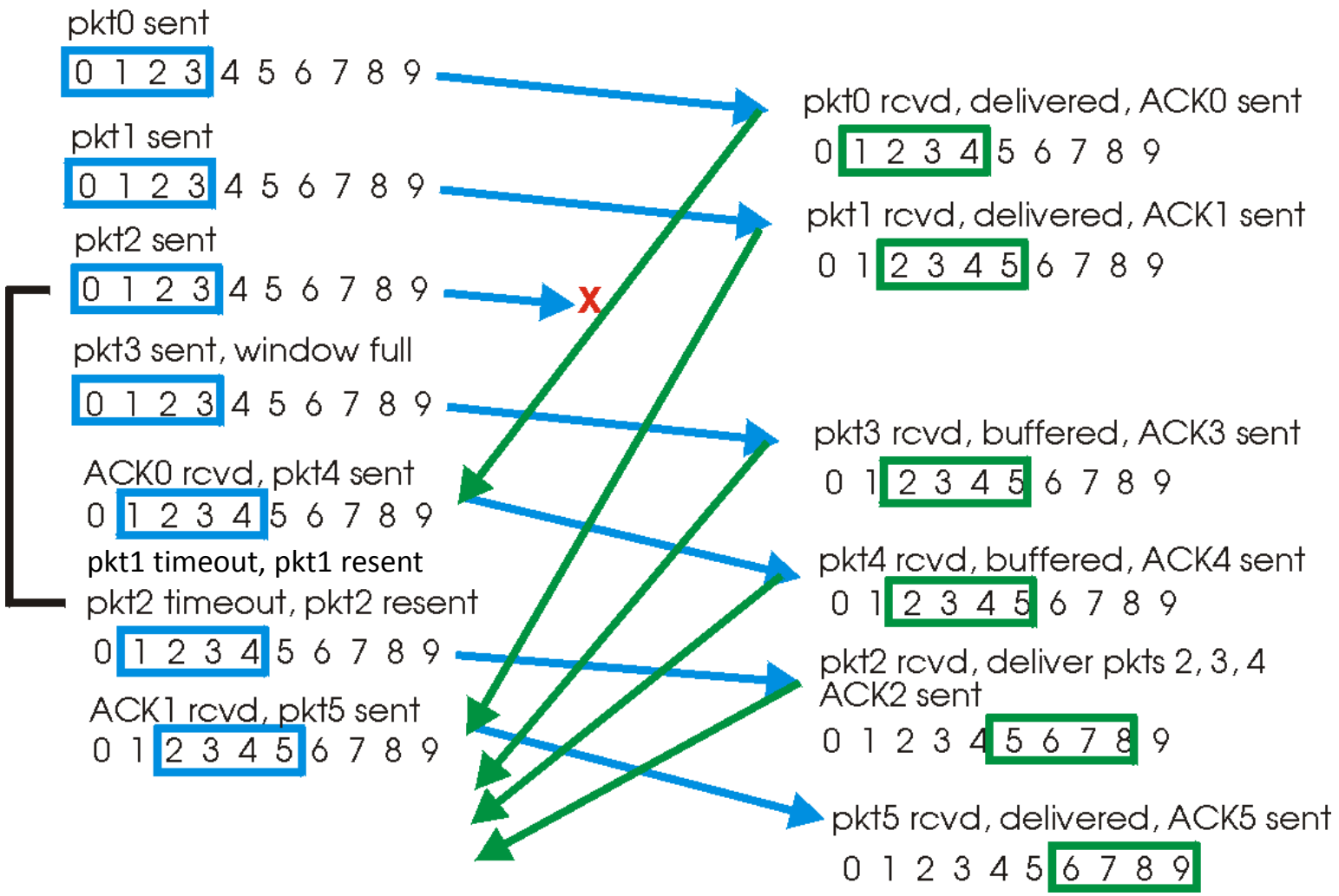
Paket n in [rcvbase-N,rcvbase-1]

- ACK(n)

sonst:

- ignorieren

Selective repeat in Aktion

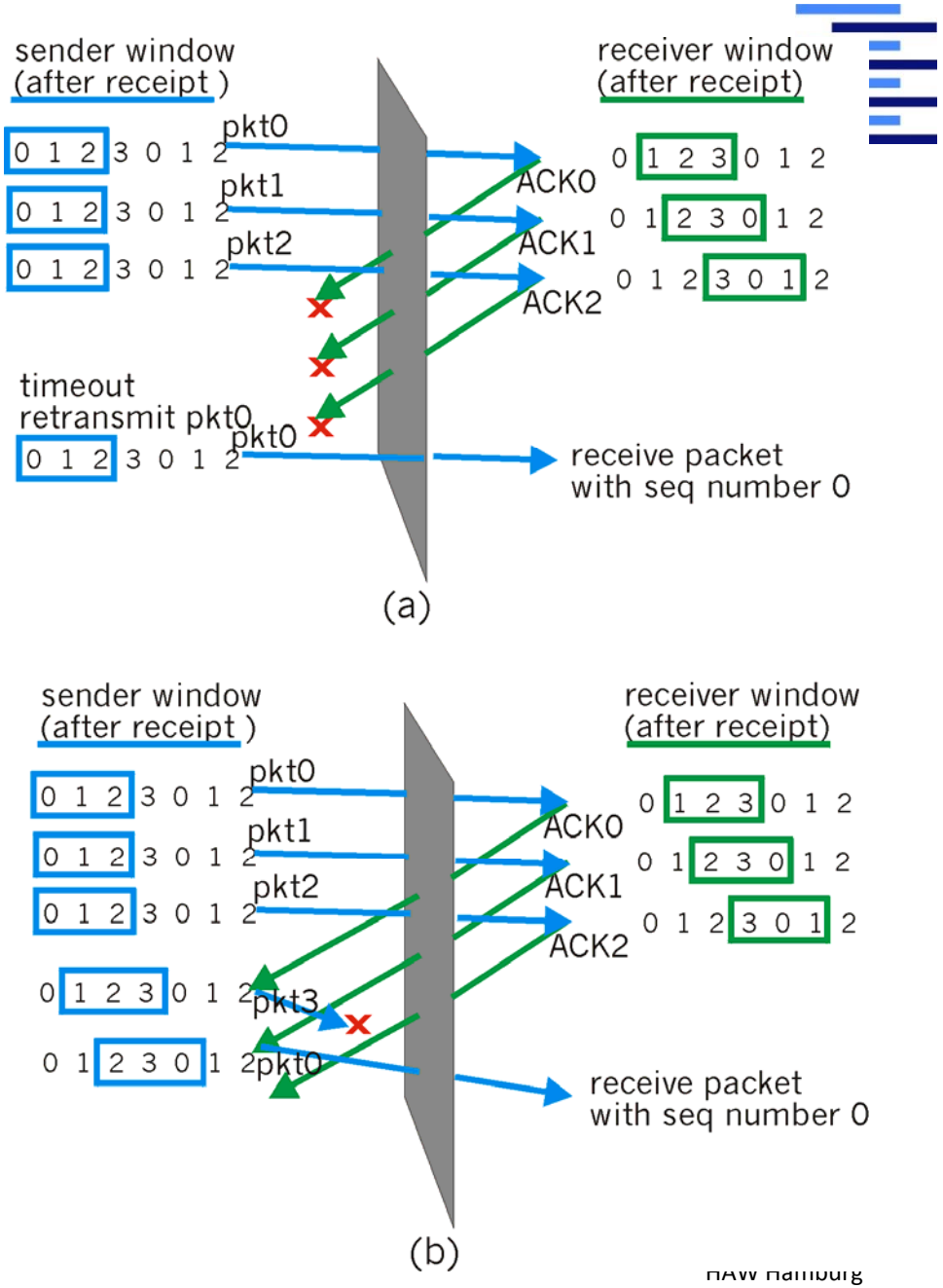


Selective repeat: Dilemma

Beispiel:

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

Was ist die Beziehung zwischen Sequenznummernbereich und Fenstergröße?



Zusammenfassung: Prinzipien des zuverlässigen Datentransfers

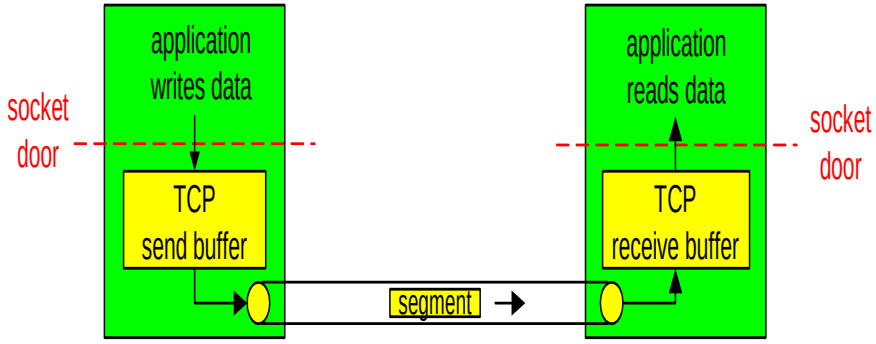
- Erkennen eines verfälschten Pakets beim Empfänger:
  - durch eine Prüfsumme
- Melden des Empfängerzustands an den Sender:
  - durch eine Quittung (ACK)
- Reparieren von Fehlern beim Sender:
  - durch wiederholtes Senden eines Pakets
- Entdecken von Duplikaten / fehlenden Paketen beim Empfänger:
  - durch Sequenznummern
- Entdecken komplett verloren gegangener Pakete beim Sender:
  - durch Timer
- Anpassen der Sendegeschwindigkeit an den Empfänger:
  - durch Window (Sendefenster)

Kapitel 3
Transportschicht

1. Dienste und Prinzipien auf der Transportschicht
2. Multiplexen und Demultiplexen von Anwendungen
3. Verbindungsloser Transport: UDP
4. Prinzipien des zuverlässigen Datentransfers
5. Verbindungsorientierter Transport: TCP
6. TCP – Staukontrolle

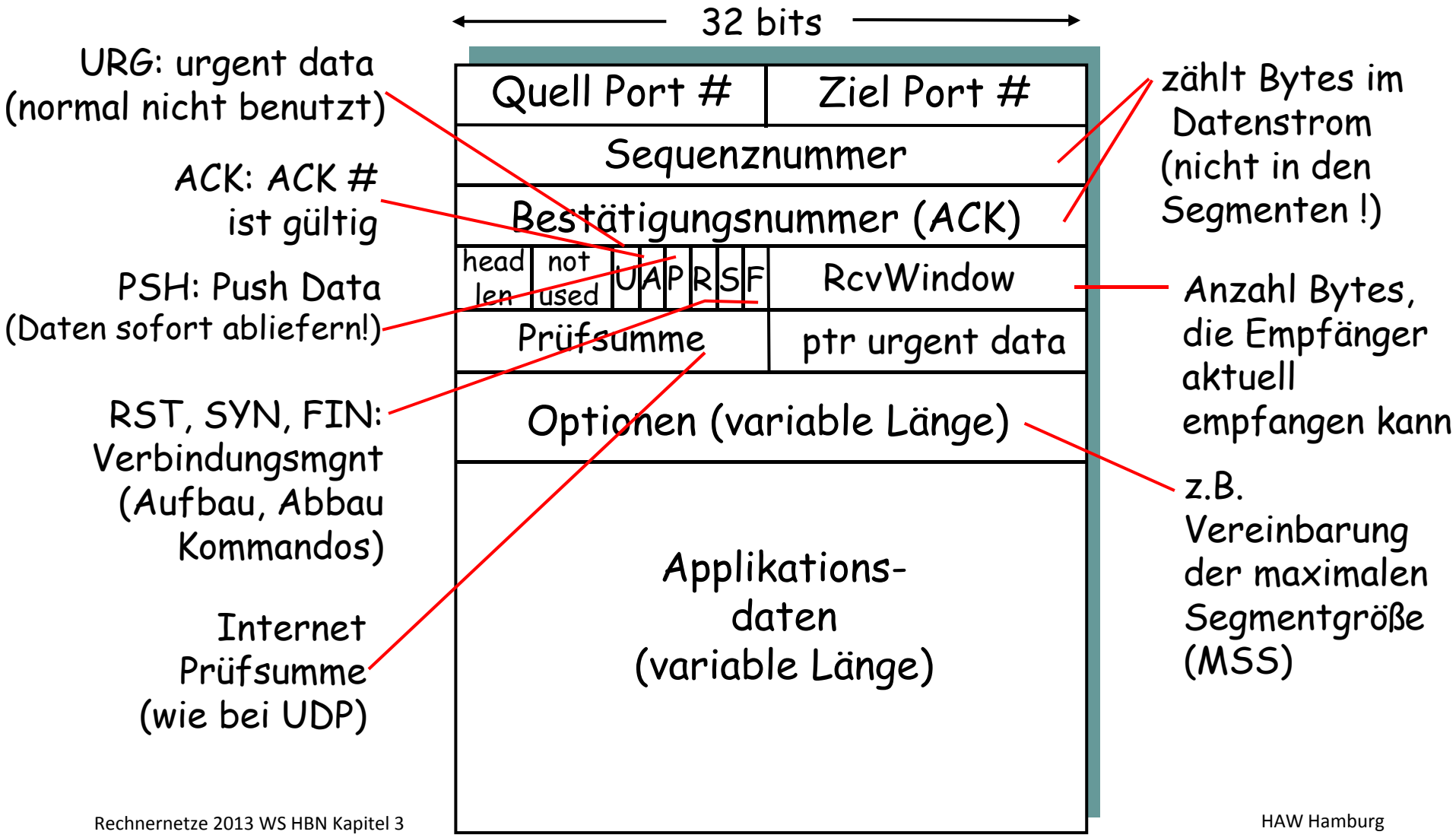
TCP: Überblick RFCs: 793, 1122, 1323, 2018, 2581

- Punkt zu Punkt - Verbindung:
  - ein Sender, ein Empfänger
- Empfängt/liefert zuverlässigen, reihenfolge-bewahrenden *Byte-Strom:*
  - erzeugt selbst Pakete ("Segmente")
- Voll-Duplex Datentransfer:
  - Bidirektionaler Datenfluss in derselben Verbindung
  - MSS: maximum segment size
- *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)
- Staukontrolle:
  - Steigere die Übertragungs-geschwindigkeit ("Slowstart") , bis ein Paket verloren geht (durch Sendefenster)
- Pipeline-Protokoll:
  - TCP Stau- und Flusskontrolle bestimmen die Fenstergröße

TCP Segment Struktur



TCP-Sequenznummern und ACKs

Sequenznummern:

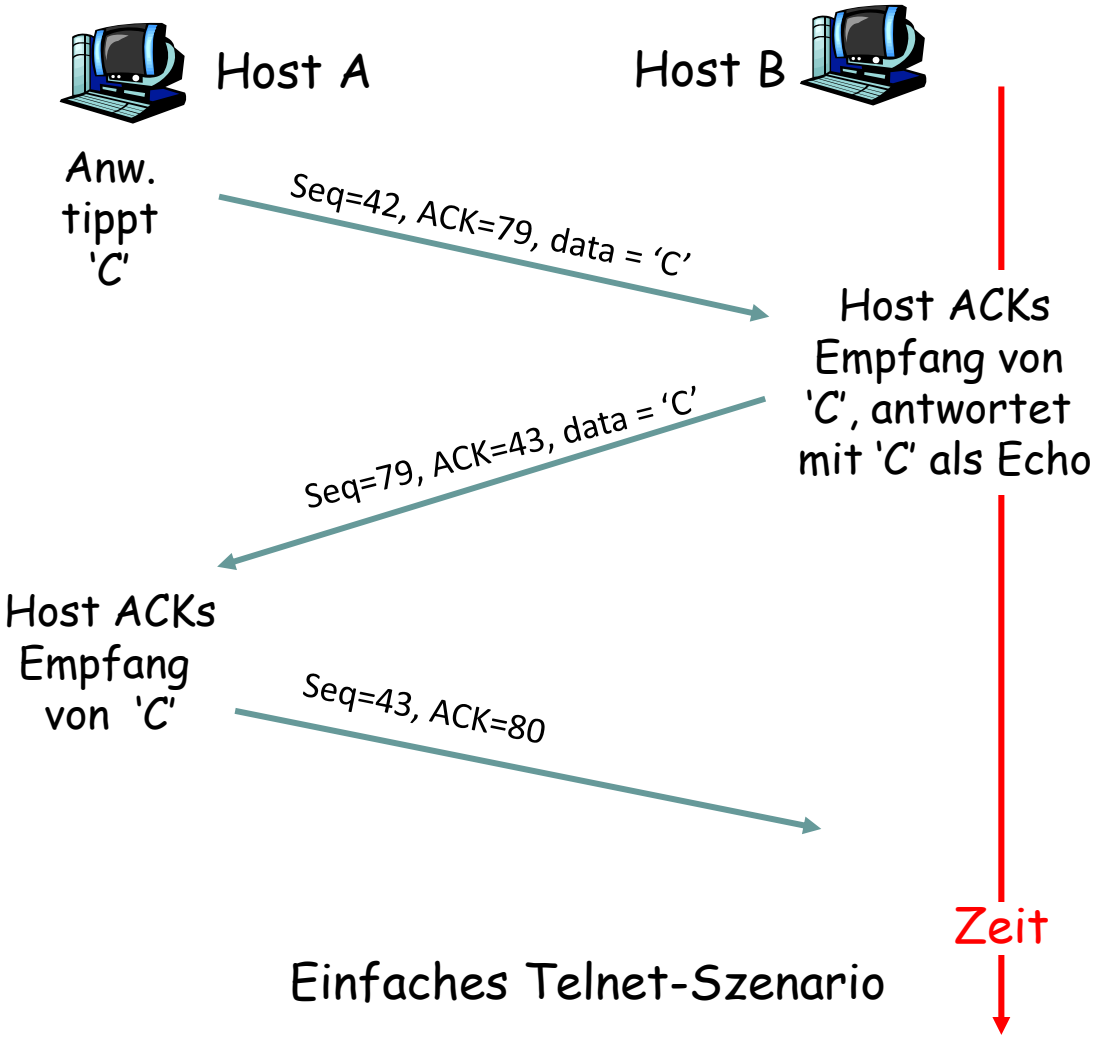
- Byte-Strom- “Zähler” des ersten Byte in den Segmentdaten

ACKs:

- Sequenz-Nr. des nächsten Byte, das von der anderen Seite erwartet wird
- Kumulatives ACK (→ *Go-Back-N)*

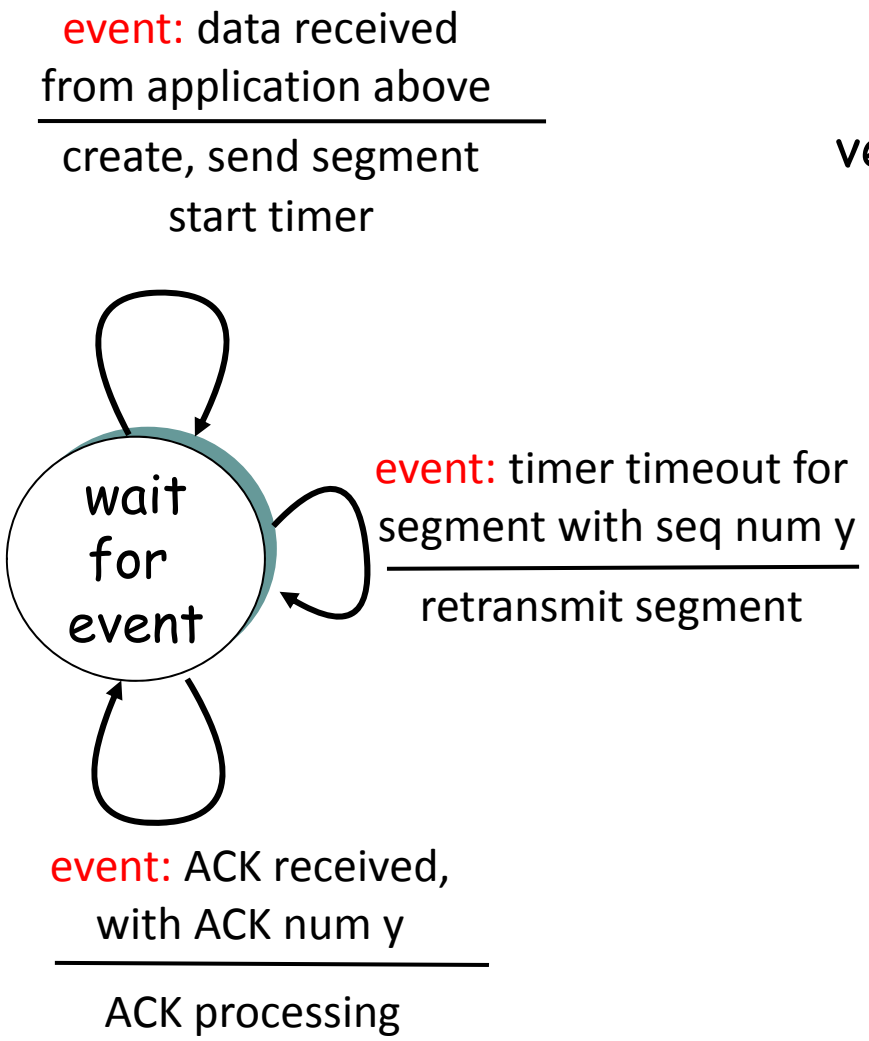
Frage: Wie behandelt Empfänger Semente, die außerhalb der Reihenfolge ankommen?

- A: TCP Spezifikation legt das nicht fest (können ggf. gespeichert werden → *Selective Repeat*)



Einfaches Telnet-Szenario

TCP: zuverlässiger Datentransfer



vereinfachter Sender, angenommen:

- Daten nur in eine Richtung
- keine Fluss- und Staukontrolle

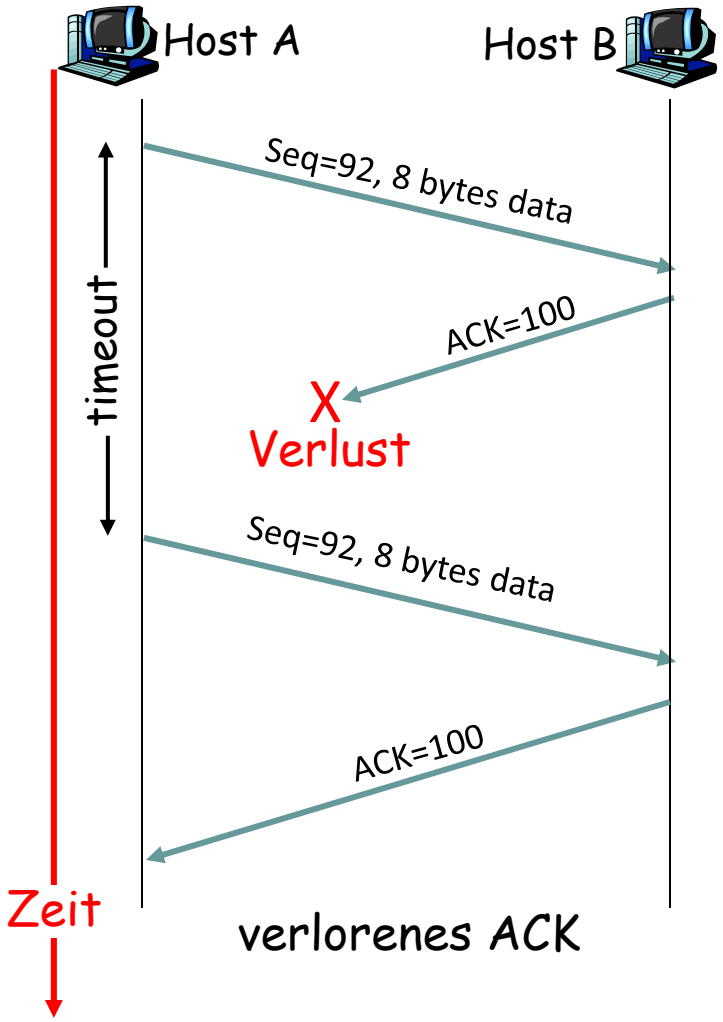
Vereinfachter TCP-Sender

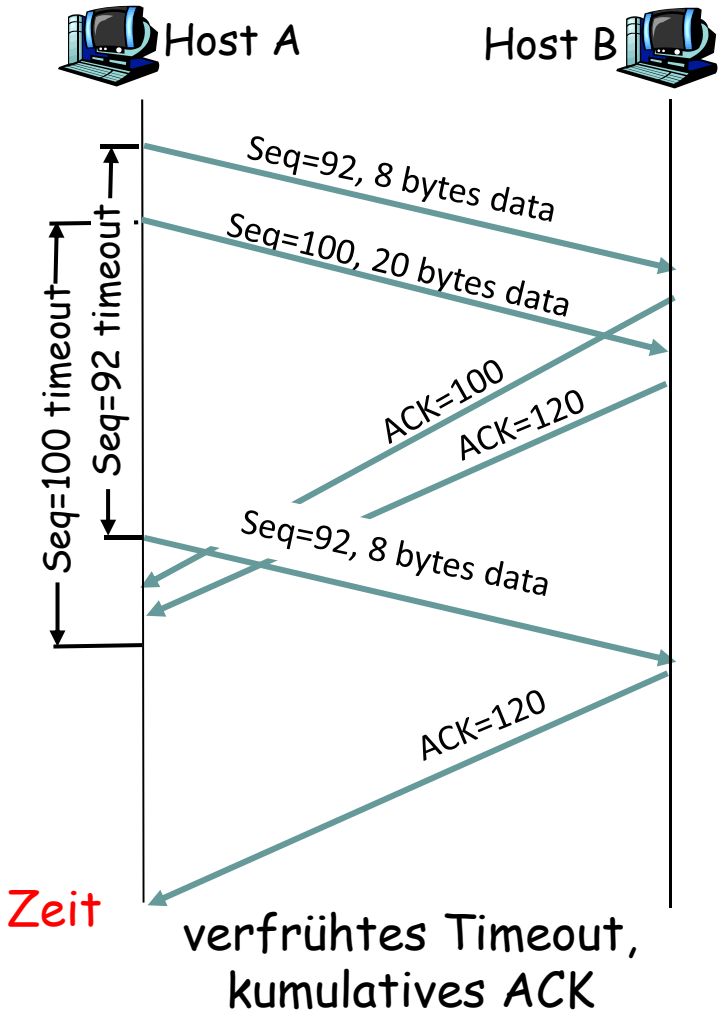
```
00  sendbase = initial_sequence number
01  nextseqnum = initial_sequence number
02
03   loop (forever) {
04    switch(event)
05    event: data received from application above
06         create TCP segment with sequence number nextseqnum
07         start timer for segment nextseqnum
08         pass segment to IP
09         nextseqnum = nextseqnum + length(data)
10     event: timer timeout for segment with sequence number y
11         retransmit segment with sequence number y
12         compute new timeout interval for segment y
13         restart timer for sequence number y
14     event: ACK received, with ACK field value of y
15         if (y > sendbase) { /* cumulative ACK of all data up to y */
16            cancel all timers for segments with sequence numbers < y
17             sendbase = y
18             }
19         else { /* a duplicate ACK for segment y = sendbase */
20             increment number of duplicate ACKs received for y
21             if ((number of duplicate ACKS received for y) == 3) {
22                /* TCP fast retransmit */
23               resend segment with sequence number y
24               restart timer for segment y
25            }
26    }  /* end of loop forever */
```

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

Ereignis	TCP Empfängeraktion
Ankunft in richtiger Reihenfolge, ohne Lücken (Seq.-Nr == erw. Seq-Nr) alles sonst schon mit ACK quittiert	verzögertes 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: Übertragungswiederholung





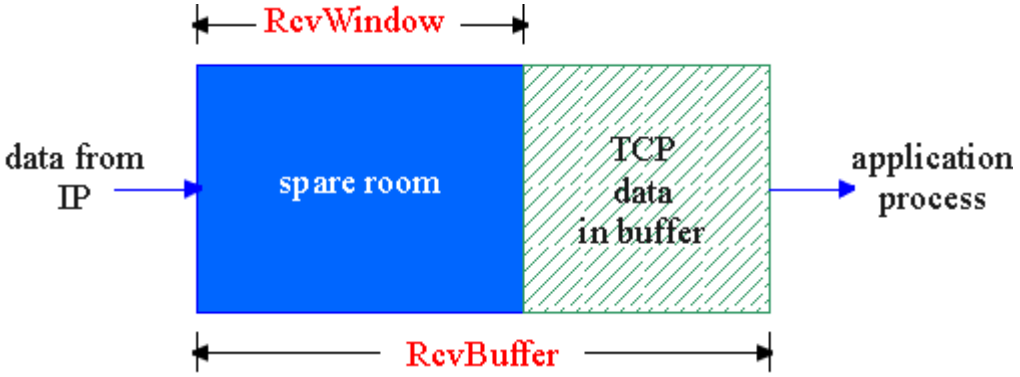
TCP Flusskontrolle

Flusskontrolle

Sender soll die Puffer des Empfängers nicht durch zu schnelles Senden überfüllen

`RcvBuffer` = Größe des TCP Empfangspuffers

`RcvWindow` = aktuell freier Platz im Empfangspuffer



receiver buffering

Empfänger: informiert den Sender explizit über die aktuelle Größe des freien Puffers (ändert sich dynamisch)

- **`RcvWindow`** Feld im TCP Segment

Sender: hält die Anzahl gesendeter, nicht quittierter Daten kleiner als den aktuell erhaltenen Wert für **`RcvWindow`**

Round Trip Time (RTT) und Timeout

Frage: Wie setzt TCP den Timeout-Wert?

- größer als die RTT (Round Trip Time)
  - Achtung: RTT ändert sich ständig
- zu kurz: verfrühtes Timeout
  - unnötiges Neuversenden
- 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 Wieder-holungen 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-Timeoutbestimmung

```
EstimatedRTT = (1-x)*EstimatedRTT + x*SampleRTT
```

- Exponentiell gewichteter Durchschnittswert
- Einfluss einer Messung sinkt exponentionell
- Typischer Wert von x: 0,1

Bestimmen des Timeout

- **`EstimatedRTT`** plus “sicherer Abstand” (Deviation)
- große Variationen von **`EstimatedRTT`** ➔ größerer Sicherheitsabstand

```
Timeout = EstimatedRTT + 4*Deviation

Deviation = (1-x)*Deviation +
            x*|SampleRTT-EstimatedRTT|
```

TCP Verbindungsmanagement

Zur Erinnerung: TCP-Sender und Empfänger bauen eine Verbindung auf, bevor Daten ausgetauscht werden!

- Initializierung der TCP-Variablen:
  - Sequenznummern
  - Puffer, Flusskontroll-Info (d.h. **`RcvWindow`**)

- *Client:* Initiatior der Verbindung

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

- *Server:* durch Client kontaktiert

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

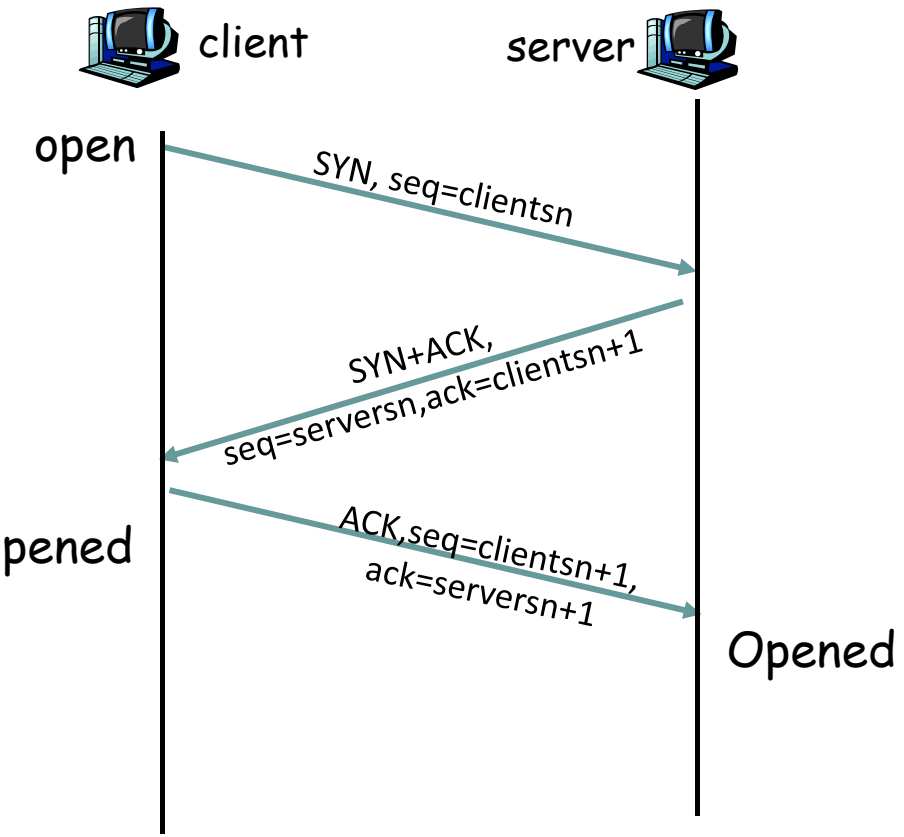
TCP Verbindungsmanagement (Open)

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 Kontrollsegment und initialer Server-Seq-Nr.

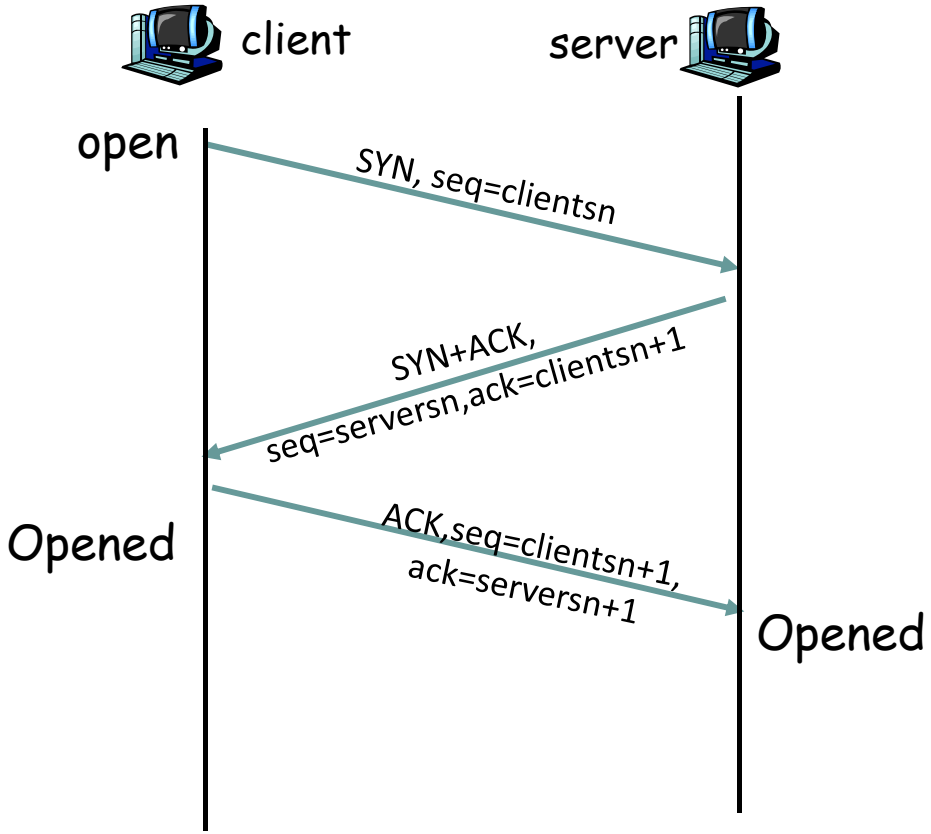


TCP Verbindungsmanagement (Open)

Schritt 3:

Client belegt Puffer, antwortet mit ACK (SYN = 0)

- bestätigt empfangene SYN+ACK
- bestätigt Server- und Client-Seq-Nr + 1
- Segment kann bereits Daten enthalten!



TCP Verbindungsmanagement (close)

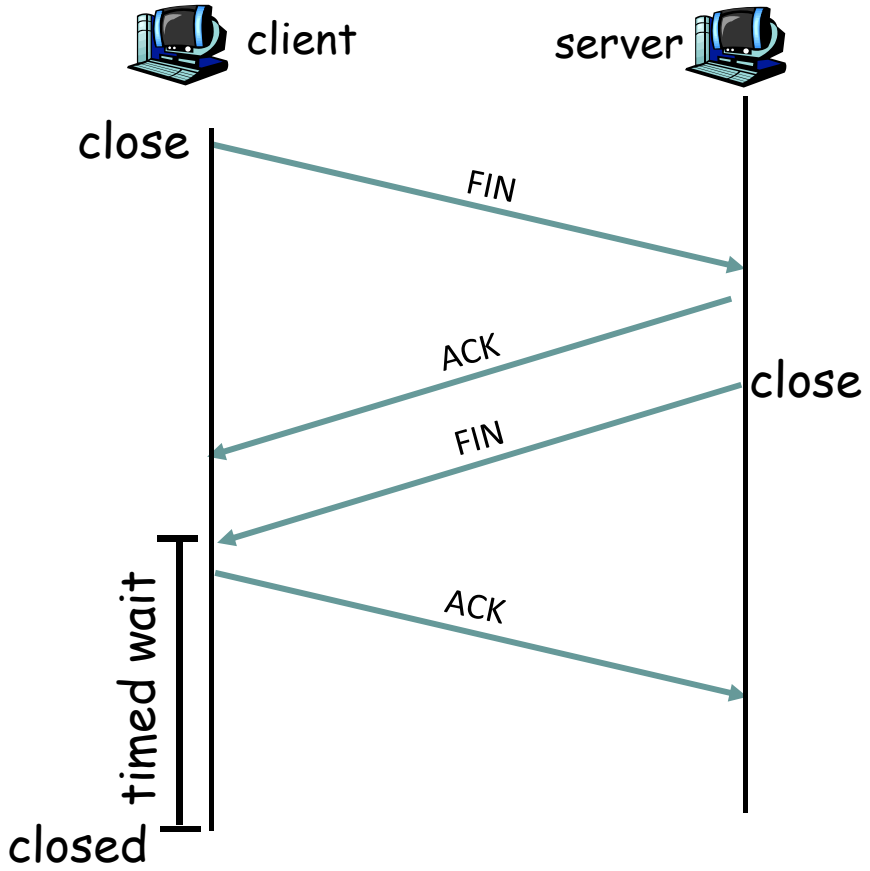
Schließen einer Verbindung:

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.



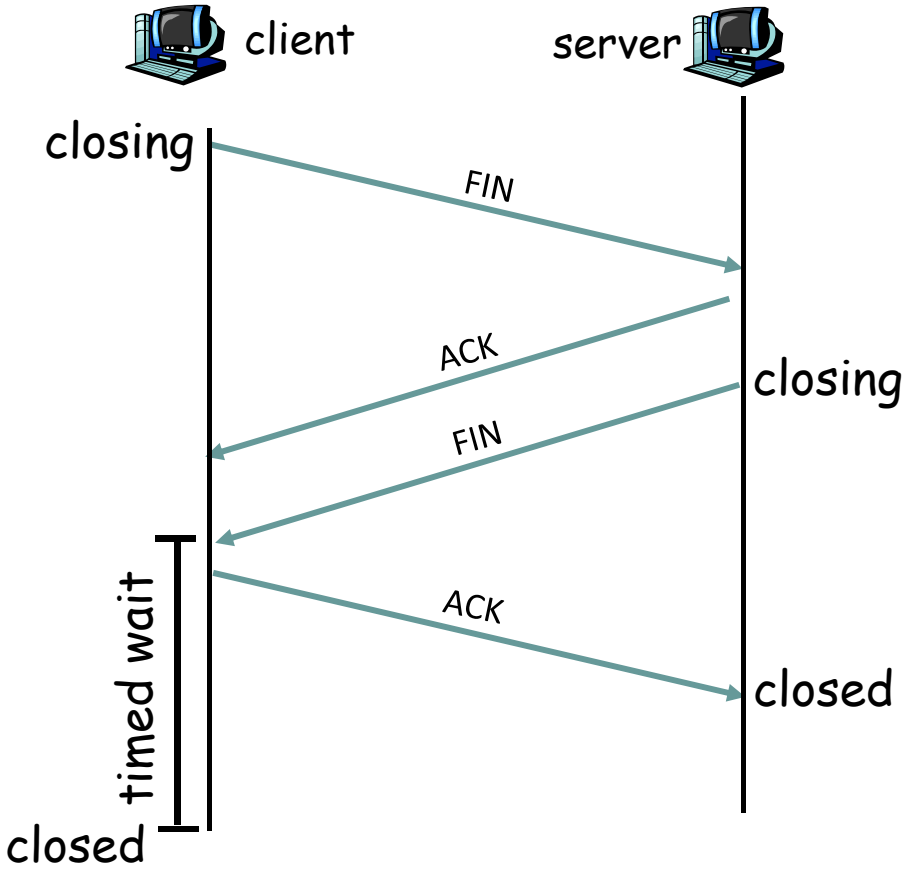
TCP Verbindungsmanagement (close)

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 !



Kapitel 3
Transportschicht

1. Dienste und Prinzipien auf der Transportschicht
2. Multiplexen und Demultiplexen von Anwendungen
3. Verbindungsloser Transport: UDP
4. Prinzipien des zuverlässigen Datentransfers
5. Verbindungsorientierter Transport: TCP
6. TCP – Staukontrolle

Prinzipien der Staukontrolle (Überlastkontrolle)

Stau:

- praktische Sicht: “zu viele Quellen senden zu viele Daten zu schnell, das *Netzwerk* kann sie nicht alle bearbeiten”
- anders als Flusskontrolle (Sender → Empfänger)!
- feststellbar durch:
  - verlorene Pakete (Pufferüberlauf in den Routern)
  - große Verzögerungen (große Queues in den Puffern der Router)
- ein wichtiges Problem *des Netzes* !

Kosten von Staus

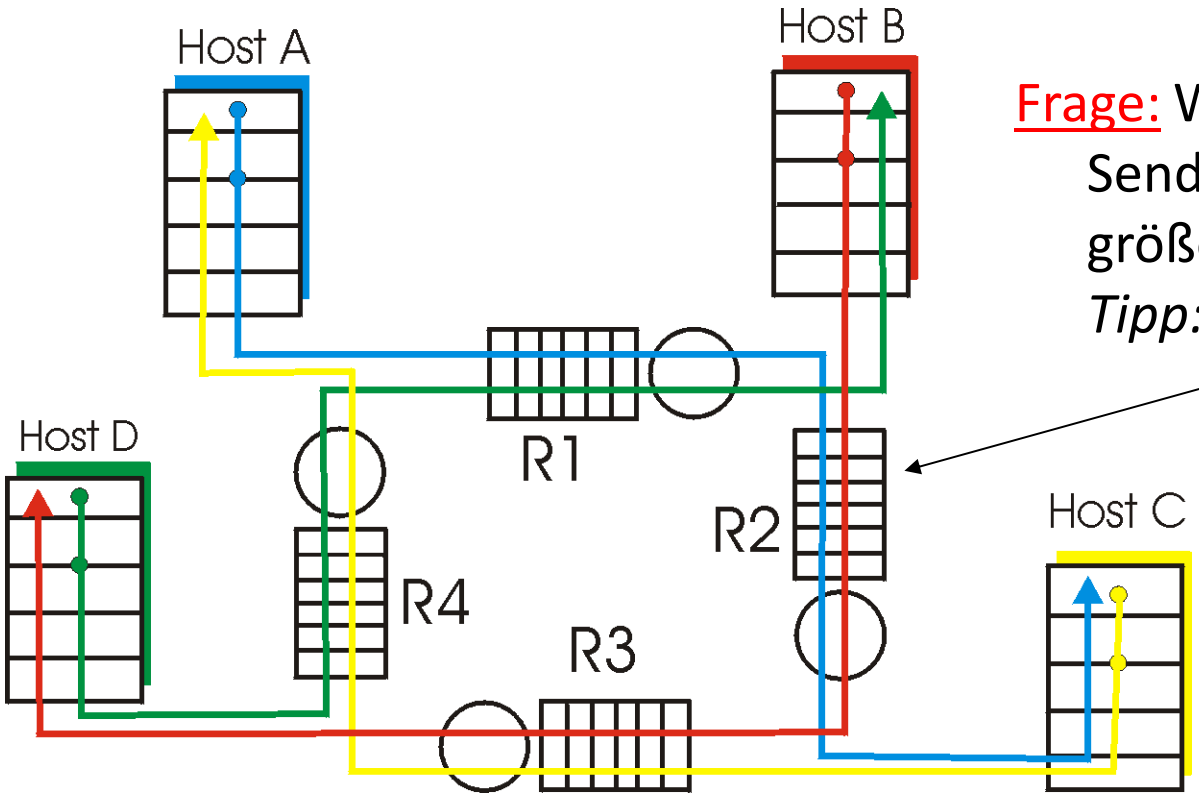
- Verlorene Pakete müssen wiederholt werden
  - Evtl. unnötiges Übertragen von Paketen, die in einem der folgenden Router verloren gehen
  - Evtl. unnötige Wiederholung von Paketen aufgrund von Verzögerungen

➔ Starke Verringerung der Übertragungsrate (“Durchsatz”)
(Tendenz: → 0 bei dauerhafter Überlast)

➔ Große Paketverzögerungen
(Tendenz: → ∞ bei dauerhafter Überlast)

Kosten von Staus: Beispiel

- 4 Sender mit identischer Senderate
- 4 Router mit identischer Übertragungskapazität
- Multi-Hop-Pfade (A-C und B-D)
- Timeout + Retransmit



Frage: Was passiert, wenn die Senderate für alle Hosts größer wird ?

Tipp: Router 2 betrachten!

Ansätze zur Staukontrolle

Ende-zu-Ende Staukontrolle:

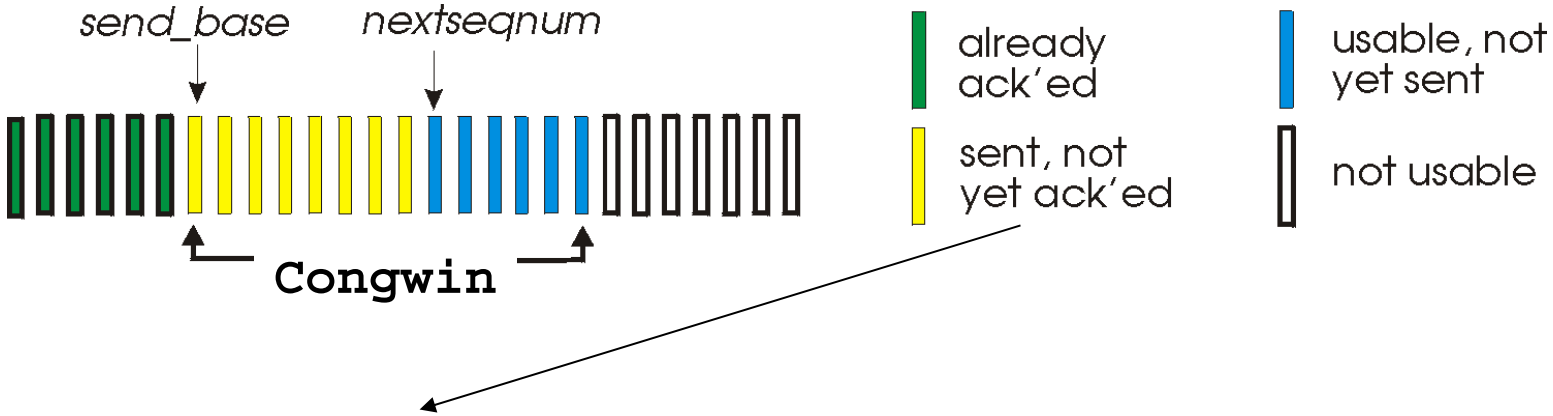
- keine explizite Kommunikation mit dem Netzwerk über Staukontrolle
- Stausituation wird gefolgert aus dem beobachteten Ende-zu-Ende-Verhalten (Paketverlust und Verzögerungen)
  - von TCP so durchgeführt

Netzwerk-gesteuerte Staukontrolle:

- Router liefern Feedback an die Endgeräte
  - 1 Bit zeigt Stau an (SNA, DECNet, ATM, TCP-ECN)
  - oder: explizite Datenrate, die ein Sender noch produzieren darf

TCP - Staukontrolle

- Ende-zu-Ende Kontrolle (keine Assistenz durch das Netzwerk)
- Übertragungsrate wird limitiert durch die Staufenstergröße (congestion window size)
  `Congwin` (zusätzlicher Parameter)

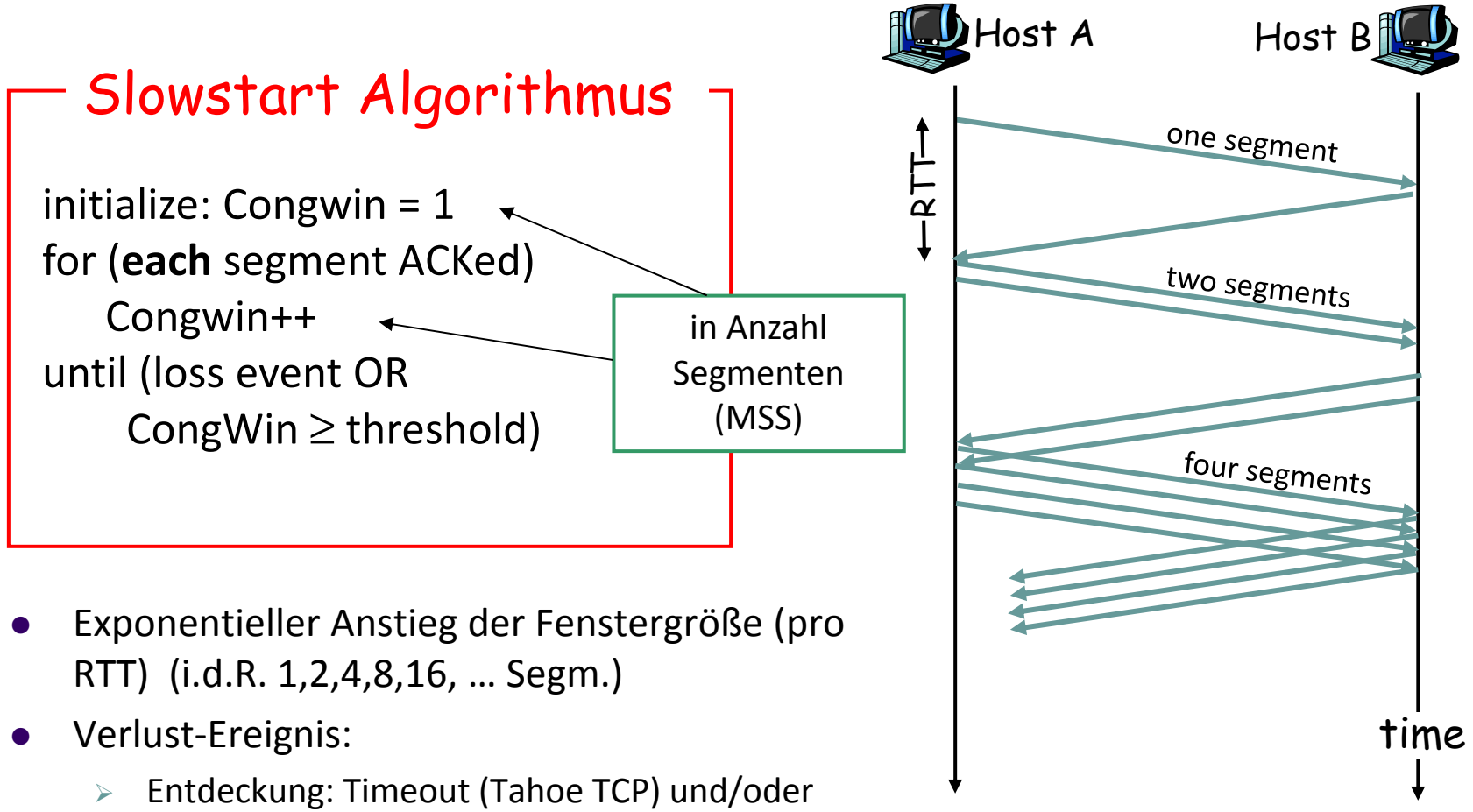


LastByteSent – LastByteAcked ≤ min {CongWin, RcvWin}

TCP - Staukontrolle:

- "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 (→ Annahme: Stau im Netz!)
    3. Bei Paketverlust: *Erniedrige* **`Congwin`** stark und beginne wieder mit 2.
- 2 "Phasen"
  - "Slow Start"
  - Stauvermeidung
- wichtige Variable:
  - **`Congwin`**
  - **`threshold:`** definiert für Congwin eine Schwelle zwischen "slow start"-Phase und Stauvermeidungs-phase

TCP Slowstart - Phase



- Exponentieller Anstieg der Fenstergröße (pro RTT) (i.d.R. 1,2,4,8,16, … Segm.)
- Verlust-Ereignis:
  - Entdeckung: Timeout (Tahoe TCP) und/oder drei gleiche ACKs (Reno TCP)
  - Starte Slowstartphase erneut

TCP - Stauvermeidungsphase

Stauvermeidung

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

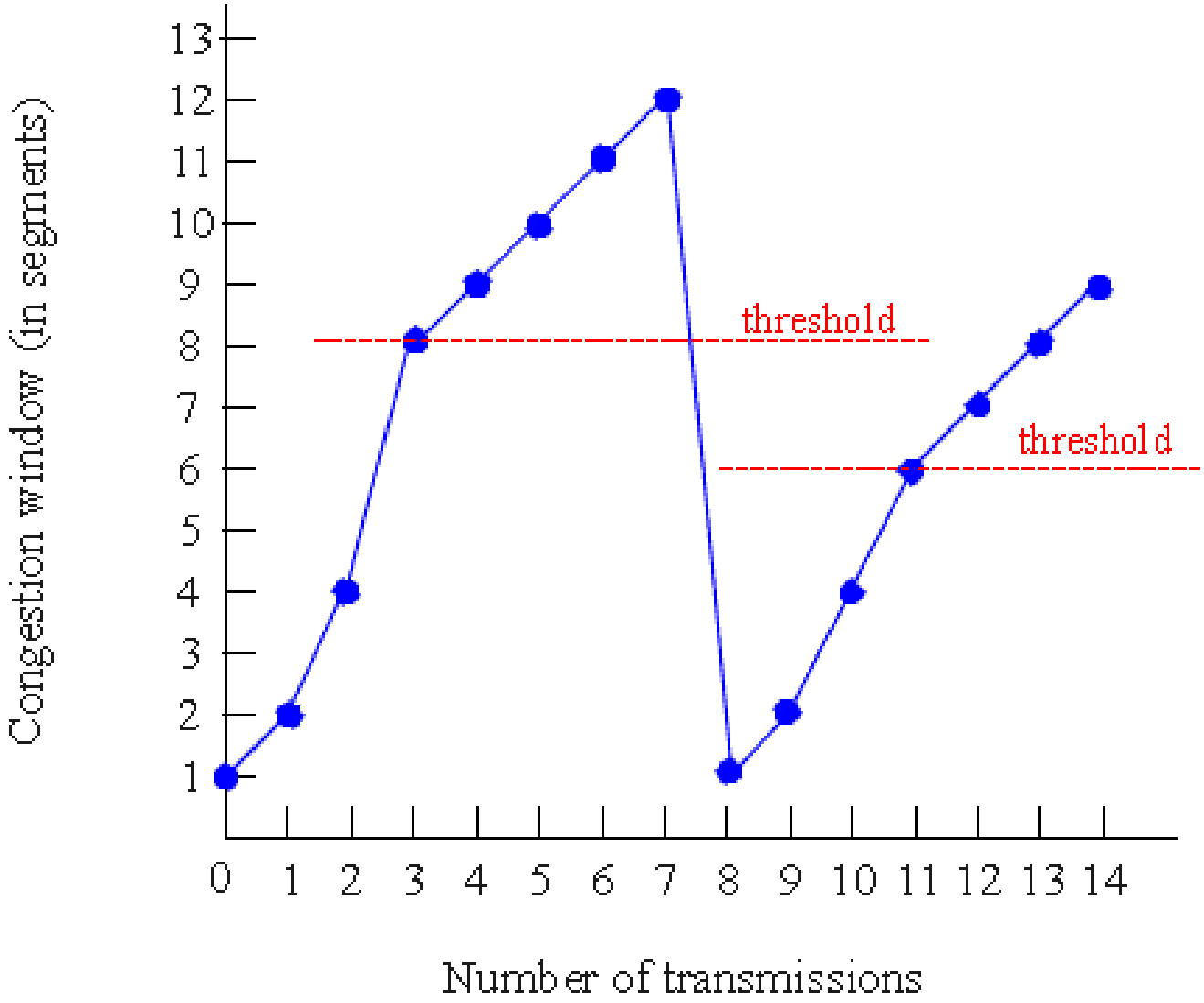
AIMD-Prinzip:

additive increase, multiplicative decrease

- erhöhe Fenster um 1 pro RTT
- erniedrige Threshold um den Faktor 2 bei einem Verlust-Ereignis

a) TCP Reno überspringt Slowstart nach drei Duplikat-ACKs (“fast retransmit / fast recovery”)
→ Info: kein Stau, sondern Übertragungsfehler!!

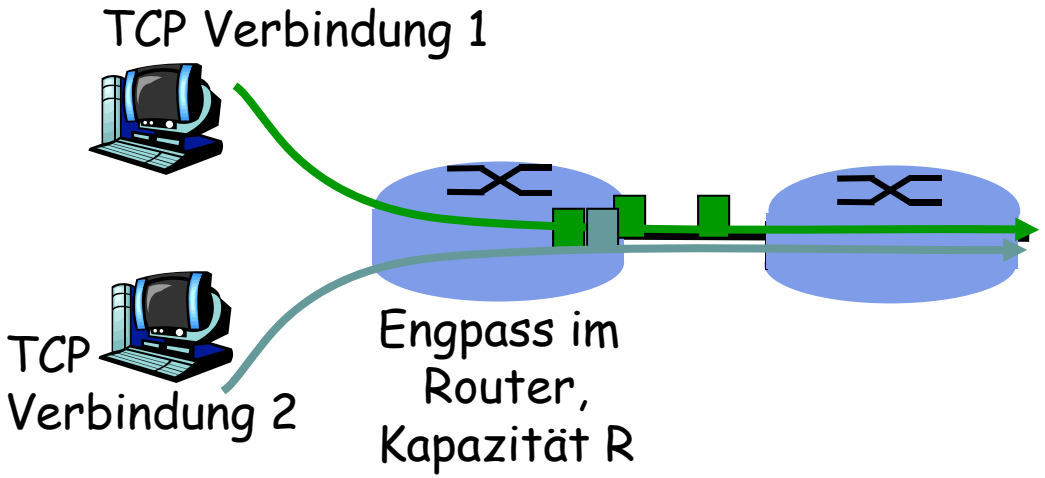
TCP – Staukontrolle: Beispiel



TCP Fairness

Ziel:

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



Trick (→ HTTP):

- Baue parallel mehrere TCP-Verbindungen auf!

Ende des 3. Kapitels:
Was haben wir geschafft?

Transportschicht

1. Dienste und Prinzipien
2. Multiplexen und Demultiplexen von Anwendungen
3. Verbindungsloser Transport: UDP
4. Prinzipien des zuverlässigen Datentransfers
5. Verbindungsorientierter Transport: TCP
6. TCP – Überlastkontrolle

Jetzt:

- wir verlassen den Netzwerk-”Rand”
- treten in den Netzwerk - ”Kern” ein