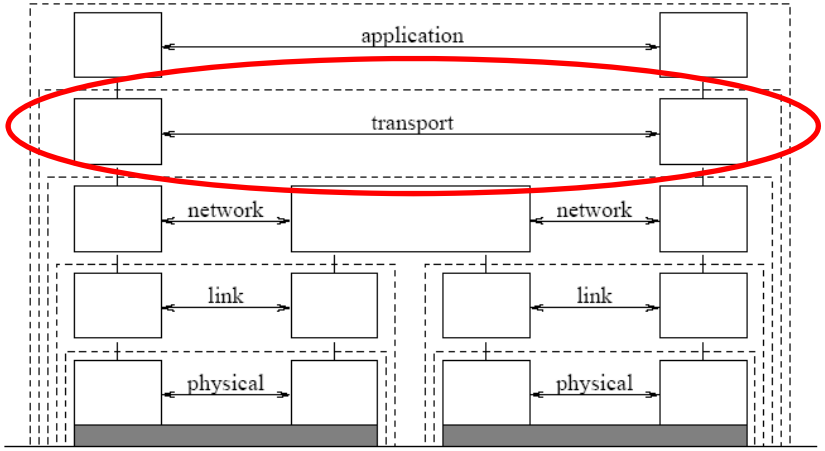
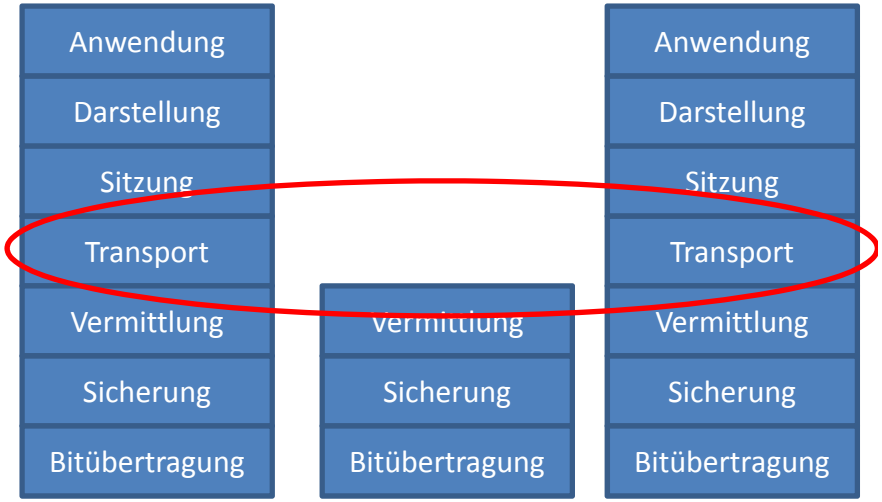
Die Transportschicht

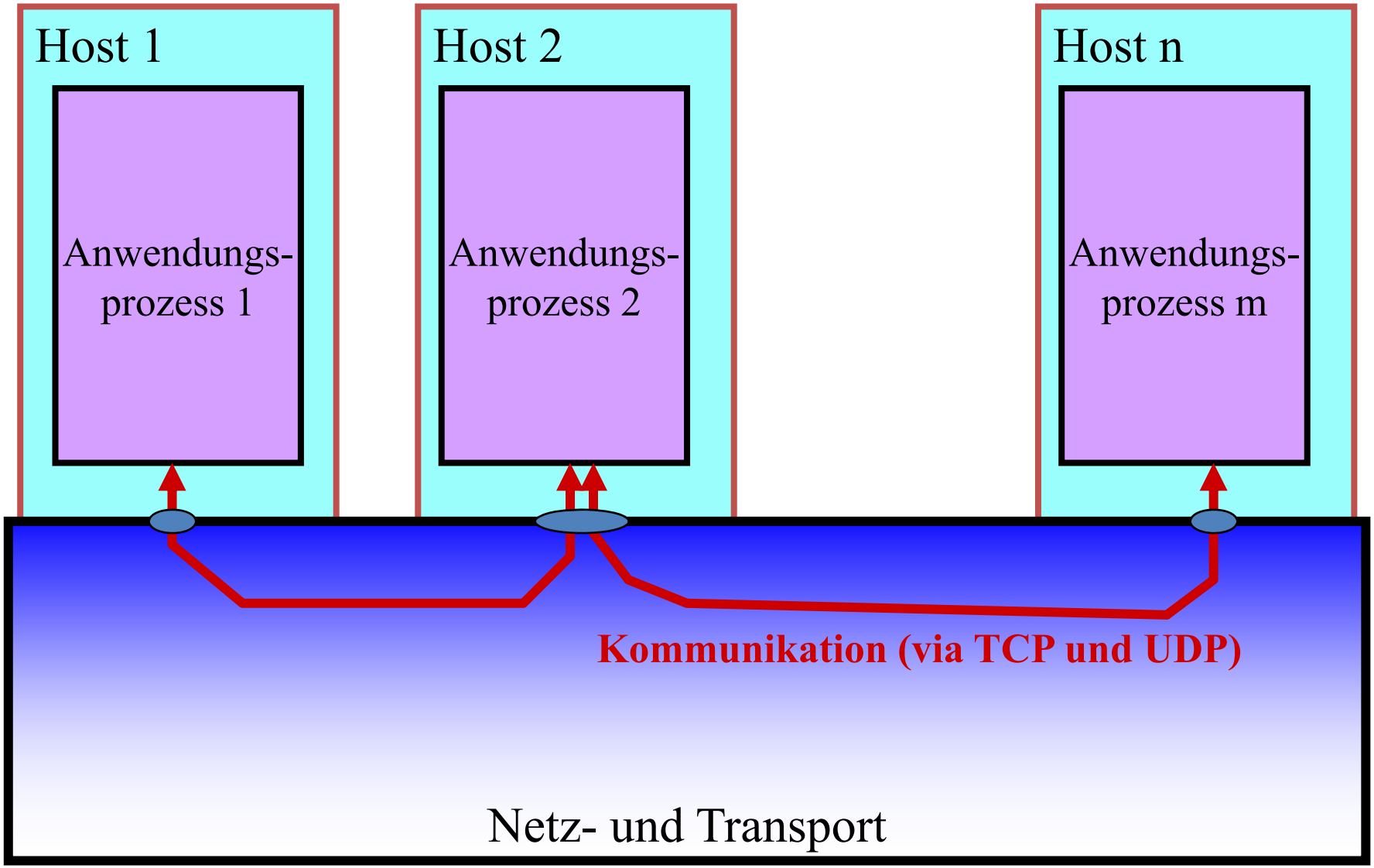
Gliederung

- Dienste der Transportschicht
- Multiplexen und Demultiplexen
- Verbindungsloser Transport UDP
- Protokollmechanismen
- Verbindungsorientierter Transport TCP
- Lastkontrolle und Überlastabwehr

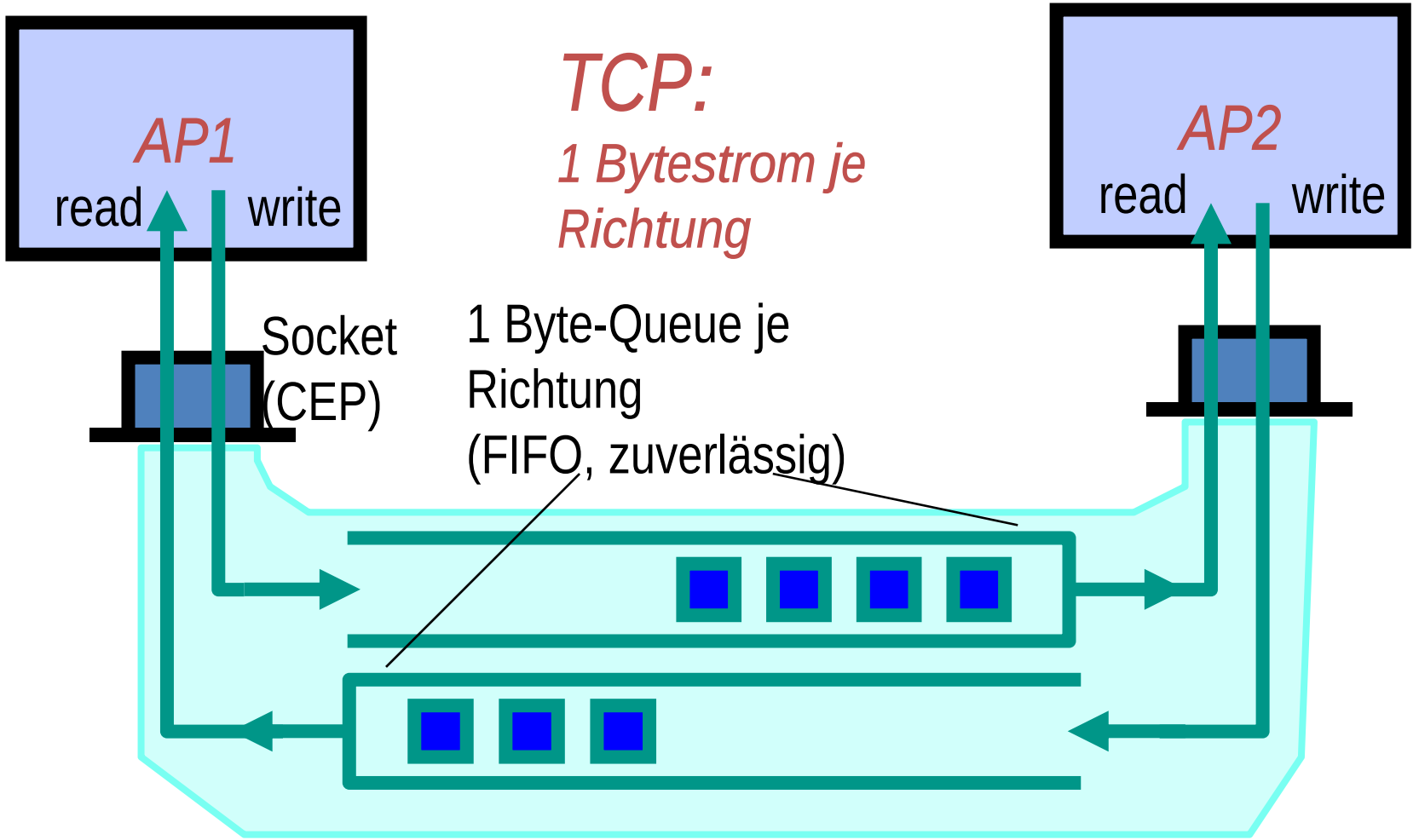




Transport: Struktur



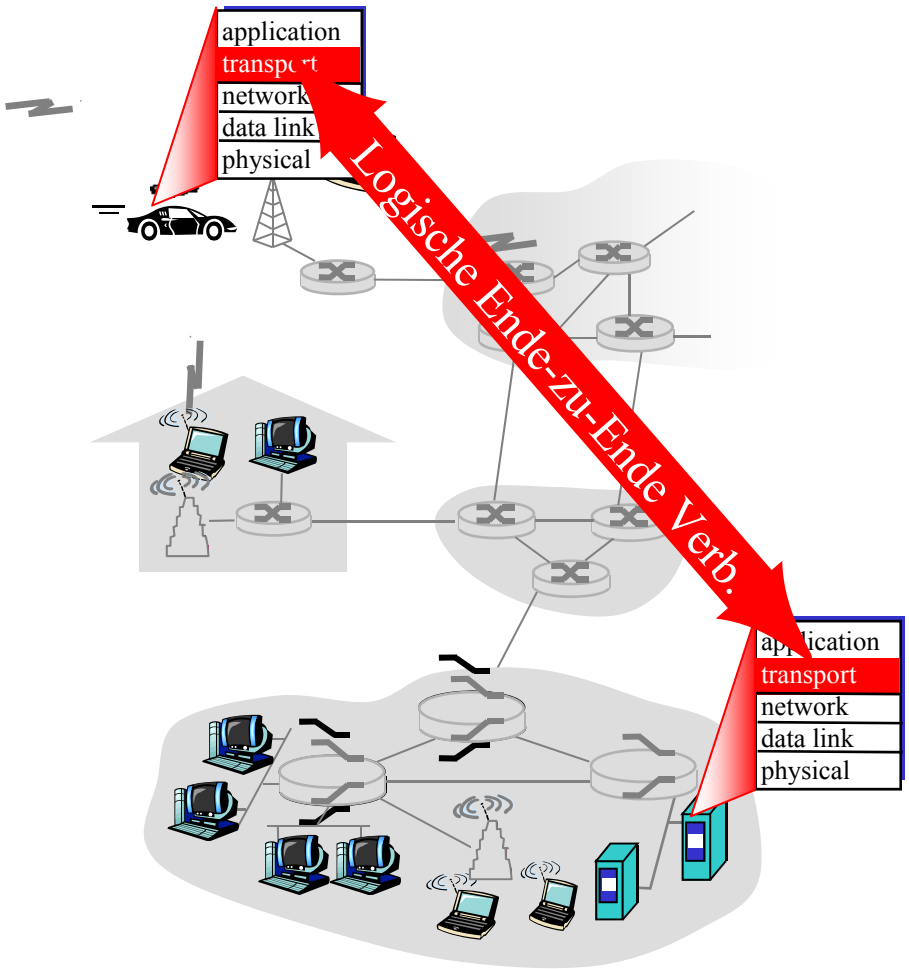
Transport: Anwendungsprozess-Kommunikation



offene TCP-Verbindung:
beide Richtungen offen

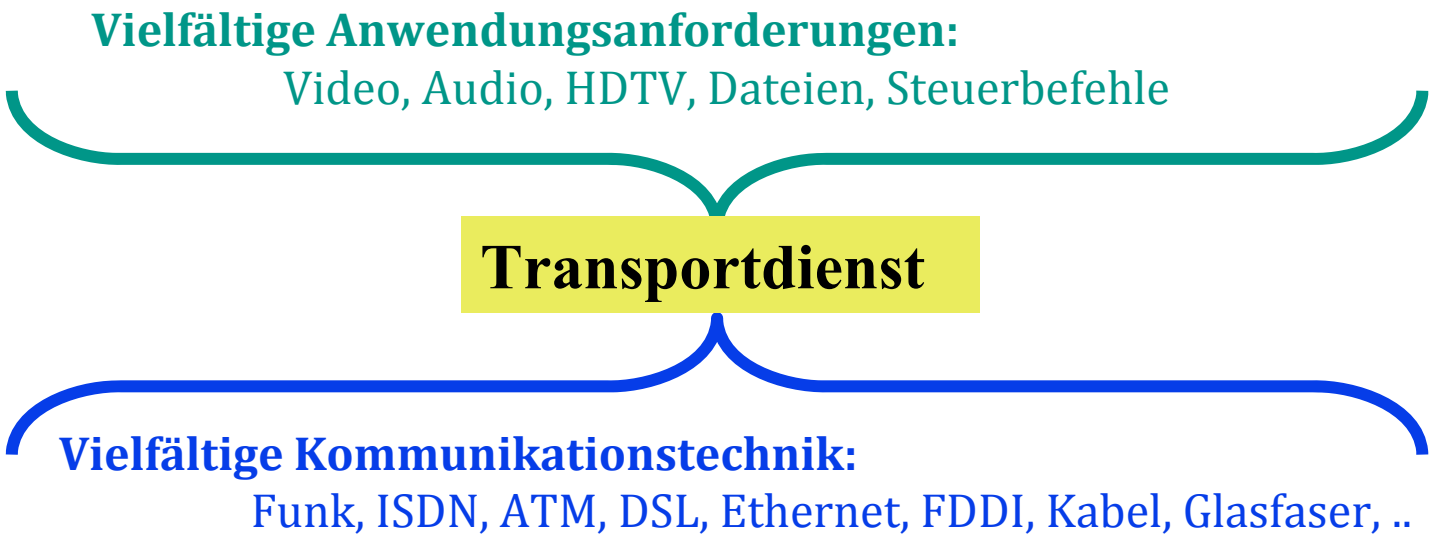
Transport: Anwendungsprozess-Kommunikation

- Funktionalität „**Ende-zu-Ende**“ Anwendungsprozesse kommunizieren:
- Adressierung
- Anwendungsnachrichten
- Anwendungsanforderungen
- Kommunikation
  - mit der Anwendungsschicht durch Bereitstellung eines Transportdienstes
  - mit der Vermittlungsschicht zur Realisierung des Transports über die Knoten im Netz



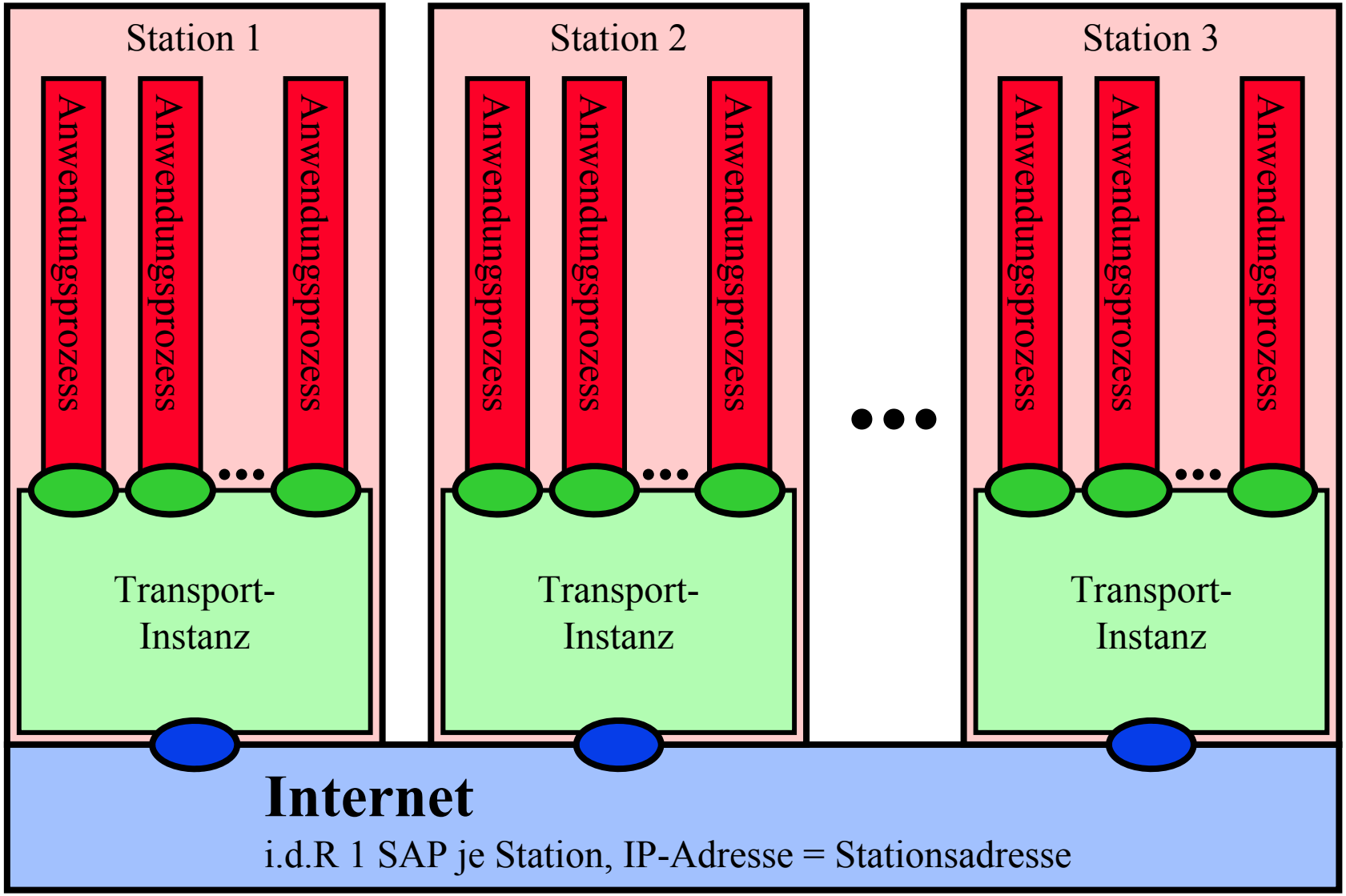
© Kurose/Ross 2009

Transport: Anwendungsprozess-Kommunikation



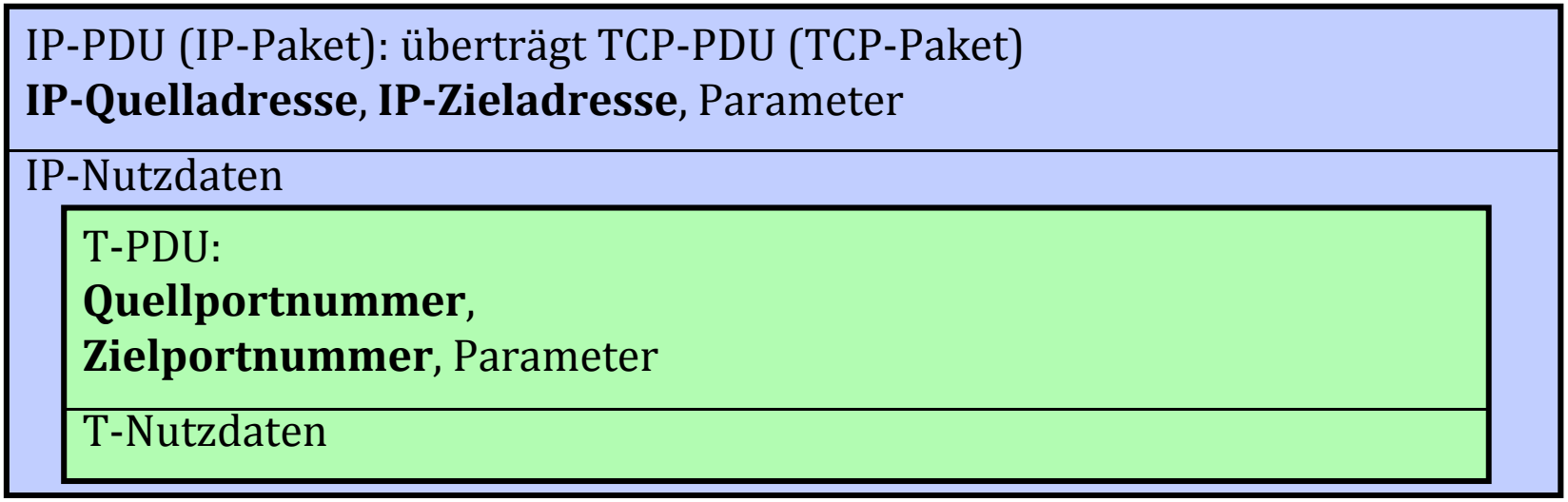
- „Universeller“ Transportdienst
  Anwendungs- und Technikunabhängigkeit
  - Dienstgüte / Quality of Service

Transport: Stations- und Prozessadressen / Multiplexen



Transportschicht: Internet

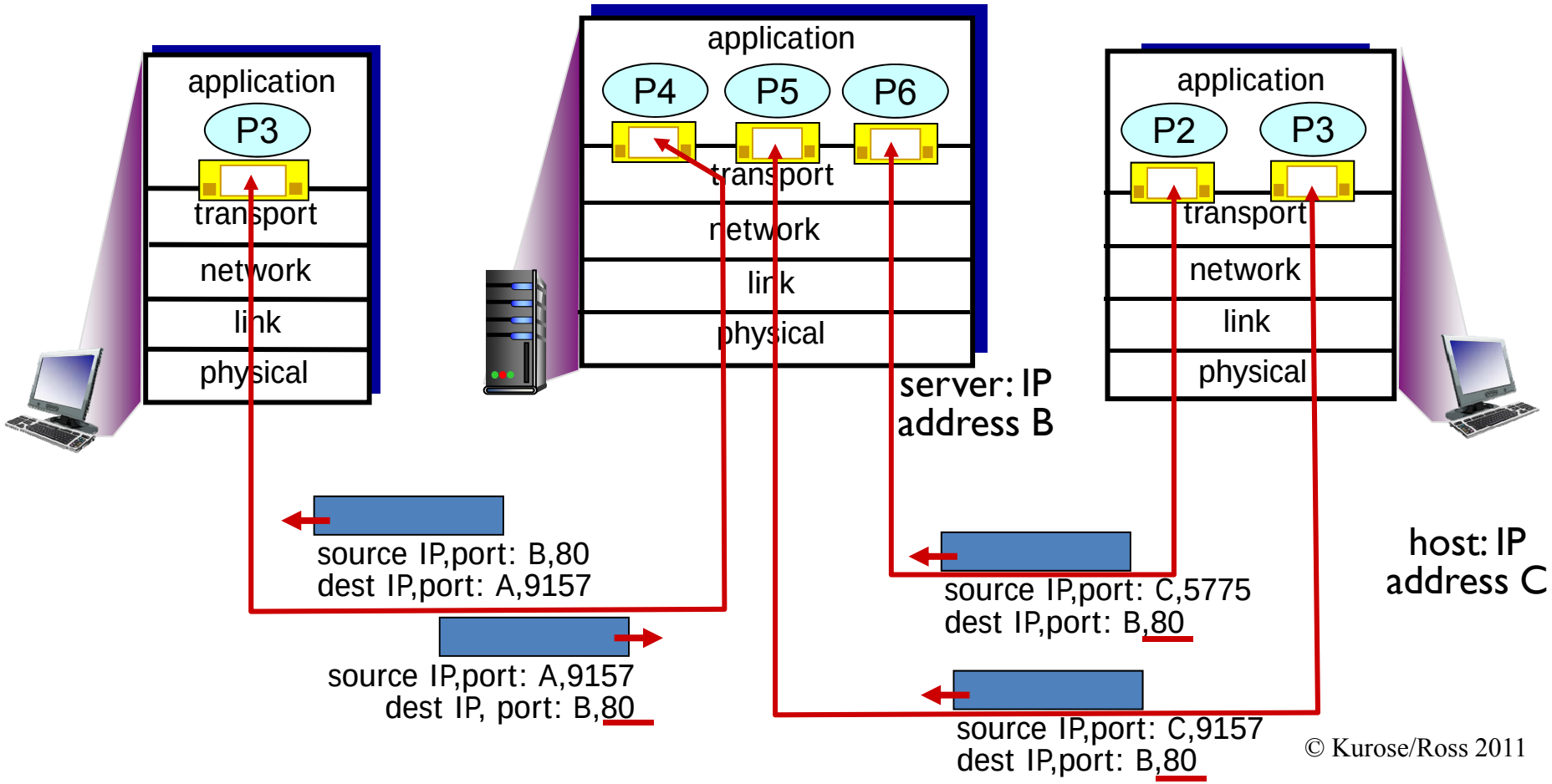
- IP-Adresse: Stationsadresse
- Transportadresse: Stationsadresse & Dienstart & Portnummer
- 2 Dienste, 2 Protokolle:
  UDP - Datagramm, TCP - Verbindungen
- Qualitäts- und Kostenkontrolle
- Multiplexen/Demultiplexen



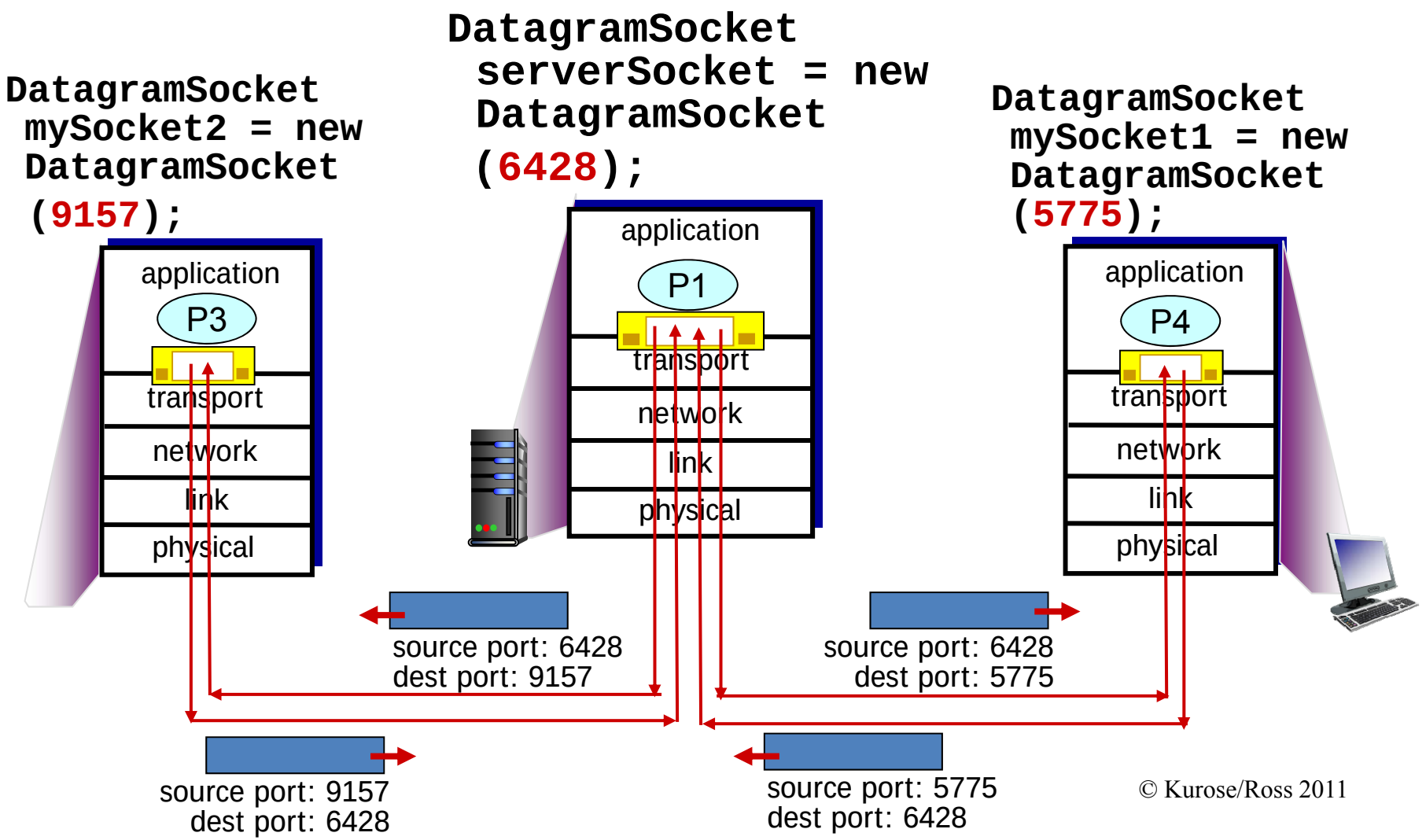
Demultiplexing: Verbindungorientiert

- Ein TCP-Socket wird durch die folgenden vier Komponenten bestimmt:
  - Quell-IP-Adresse
  - Quell-Port
  - Ziel-IP-Adresse
  - Ziel-Port
- Server betreiben simultan viele Sockets
  z.B. Web-Server:
  - Ein Socket pro Client
  - Im nicht persistenten Fall, ein Socket pro Anforderung

Demultiplexing: Verbindungorientiert



Demultiplexing: Verbindungslos

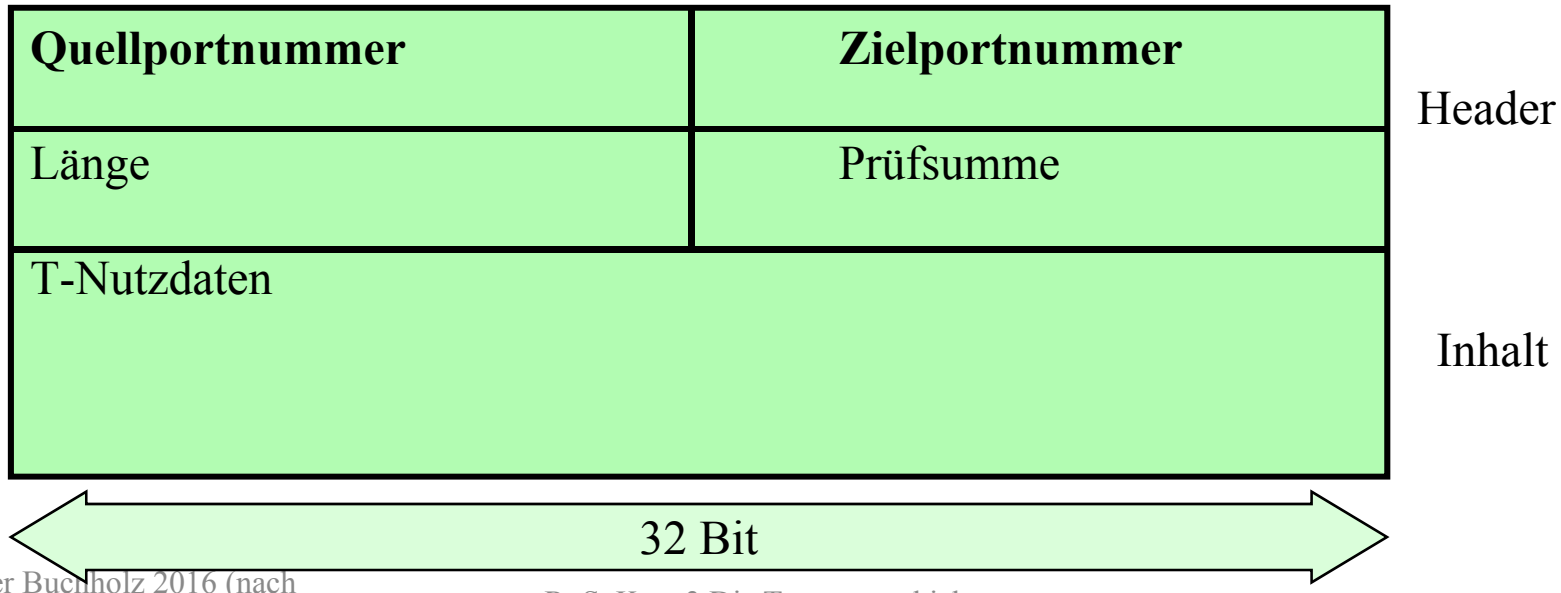


© Kurose/Ross 2011

UDP – User Datagram Protocol

- Verbindungslos, unzuverlässig, nicht reihenfolgetreu
- einfacher Dienst:

  SendeRequest (Zieladr., Daten) →
  EmpfangIndication (Quelladr., Daten)
- einfaches Protokoll, wenig Overhead:
  i.d.R. 1:1-Weitergabe per IP-Paket (Segmentierung möglich)
- PDU-Format: 1 UDP-Segment



Internet-Prüfsumme: Verfälschungserkennung

Das Verfahren nach **RFC1071** wird eingesetzt zur Berechnung der Prüfsummen von UDP-Datagrammen, TCP-Segmenten und IP-Datagrammen.

UDP-Datagramme (optional)

Prüfsumme über

- alle realen Headerfelder (Source-, Destination-Port, Length),
- Pseudo-Headerfelder (IP-Source-, Destination-Address, . . .),
- Datenfeld.

TCP-Segmente

Prüfsumme über

- alle Headerfelder (Source-, Destination-Port, Sequence-, Ack-Number, . . . , Options),
- Pseudo-Headerfelder (IP-Source-, Destination-Address, . . .),
- Datenfeld.

IP-Datagramme

Prüfsumme über

- alle Headerfelder (Version, IHL, . . . , Source-, Destination-Address, . . . , Options),
- jedoch **nicht** über das Datenfeld.

Internet-Prüfsumme: Verfälschungserkennung

Berechnung der Internet Prüfsumme

1. Benachbarte Octets (8 Bit-Wörter = Bytes), die in die Prüfsumme eingehen, werden paarweise zu 16 Bit-Integers zusammengefügt. Bei einer ungeraden Anzahl Octets wird mit einem Null-Octet aufgefüllt.
2. Das Prüfsummenfeld selbst ist mit Nullen gefüllt.
3. Die 16 Bit-Einerkomplement-Summe über alle beteiligten 16 Bit-Wörter wird berechnet und deren Einerkomplement in das Prüfsummenfeld geschrieben.
4. Um die Prüfsumme zu überprüfen, wird die 16Bit-Einerkomplement-Summe über dieselben Octets berechnet. Wenn alle Stellen des Ergebnisses 1 sind (111 . . . 1 = −0, s. u.), ist die Prüfung erfolgreich.

Einerkomplement-Addition „+“:

	13	1101	
+	4	0100	Stellenweise Addition, dann
=	17	1 0001	Aufaddieren des Übertrags
=	**2**	**0010**	**⇒ Prüfsumme 1101**

Internet-Prüfsumme: Verfälschungserkennung

Notation

- Sequenz von Octets: A, B, C, D, . . . , Y, Z
- [a, b]: 16 Bit-Integer 256 * a + b
- Einerkomplementsumme über die o. g. Sequenz von Octets:
  [A, B] [C, D] . . . [Z, 0]

Eigenschaften der Einerkomplementrechnung „+“

Kommutativität: [A,B] „+“ [C,D] = [C,D] „+“ [A,B]

Assoziativität: [A,B] „+“ ([C,D] „+“ [E, F])= ([A,B] „+“ [C,D]) „+“ [E, F]

➔

Unabhängigkeit von der Byte-Reihenfolge,

Parallelisierbarkeit

Anpassen an Änderungen ohne totale Neuberechnung

Internet-Prüfsumme: Teilaspekte

Optionale Prüfsummenberechnung für UDP

Wie codiert UDP, dass eine Prüfsumme nicht berechnet worden ist?
Wenn keine Prüfsumme berechnet worden ist, wird als Prüfsumme 000 . . . 0 übertragen. Falls die errechnete Prüfsumme 000 . . . 0 ist, wird das Prüfsummenfeld auf 1111 . . . 1 gesetzt. Beide bedeuten im Einerkomplement 0 bzw. −0.

Neuberechnung der IP-Datagramm-Prüfsumme an jedem Hop?

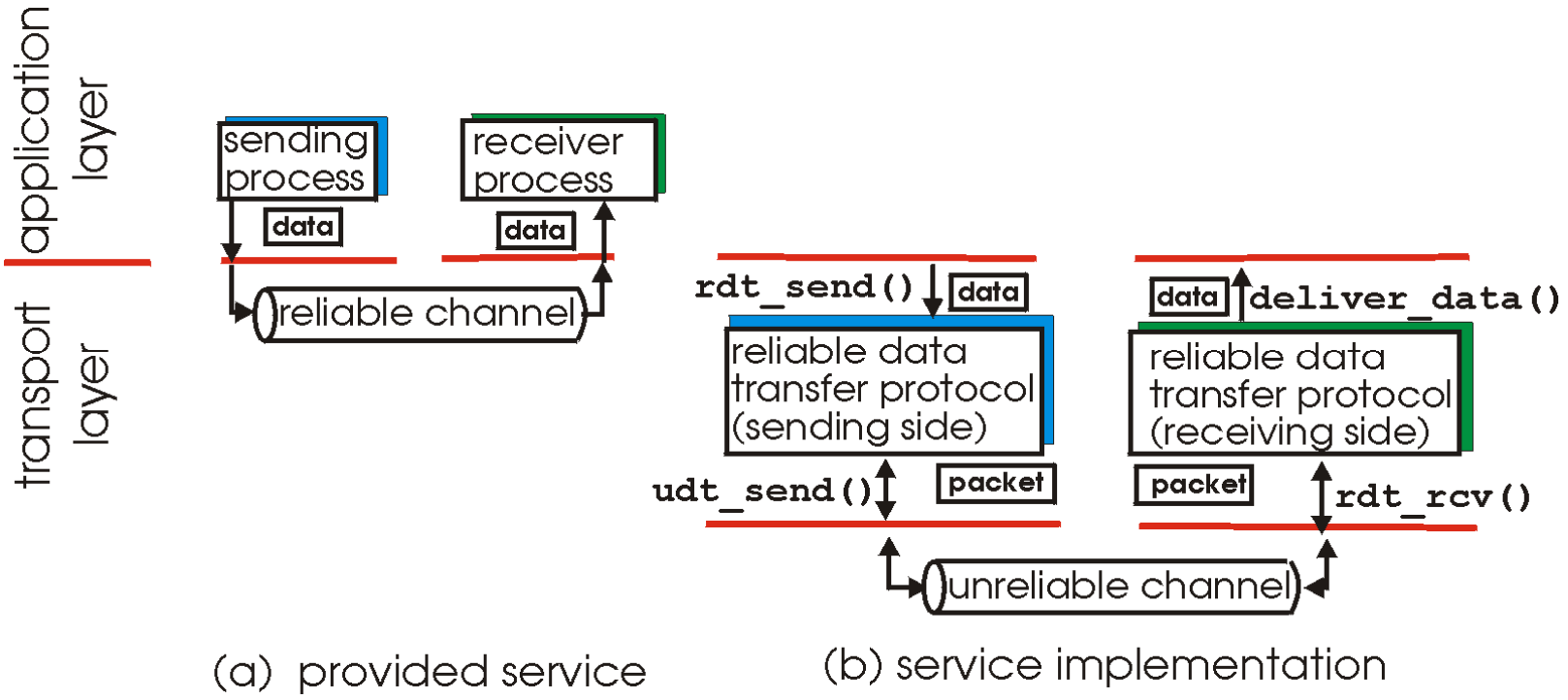
An jedem Hop wird die Time-to-Live eines IP-Datagramms dekrementiert. Damit verändert sich auch die Prüfsumme. Muss diese an jedem Hop neu berechnet werden?
Nein. Wegen der Assoziativität und der Invertierung genügt es, den Wert, um den die TTL verringert wurde, zu addieren:

$$C0 = C + (-m) + m0 = C + (m0 - m)$$

wobei C alte, C0 neue Prüfsumme, m alte und m0 neue TTL.
Weitere Information sind in der RFC 1141 zu finden.

Auf dem Weg zu TCP

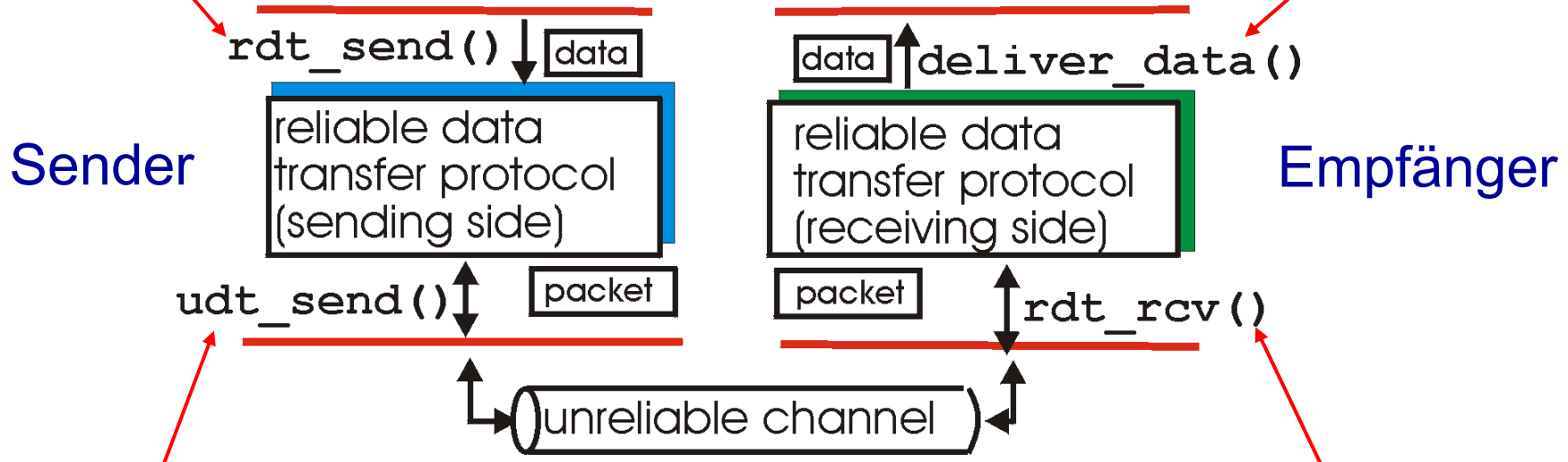
- Zuverlässige Kommunikation über unzuverlässigen Basisdienst



Schrittweise Entwicklung

`rdt_send()`: Aufruf durch darüber liegende Schicht (z.B. Anwendung), Weitergabe der zu sendenden Daten

`deliver_data()`: Aufruf durch rdt, Weitergabe der Daten an höhere Schicht

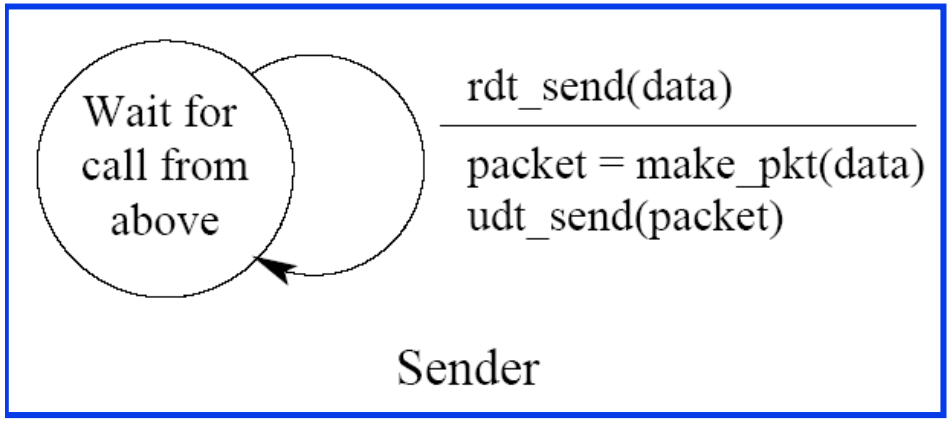


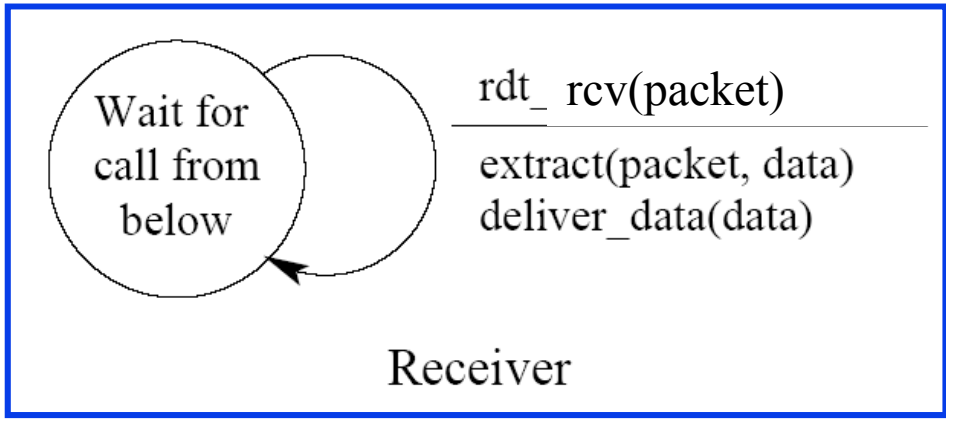
`udt_send()`: Aufruf durch rdt, Übertragung eines Pakets über den unzuverlässigen Kanal zum Empfänger

`rdt_rcv()`: Aufruf, wenn ein paket auf Empfängerseite ankommt

rdt1.0: Zuverlässiger Basisdienst

- Einfaches Weitergeben genügt





- Wenn Basisdienst schon zuverlässig ist:
  - Verfälschungsfrei
  - Verlustfrei
  - Phantomfrei
  - Duplikatfrei
  - Vertauschungsfrei

rdt2.0: Basisdienst nicht verfälschungsfrei (Bitfehler)

- Fehlerbehandlung generell
  1. ERKENNEN
  2. BEHEBEN

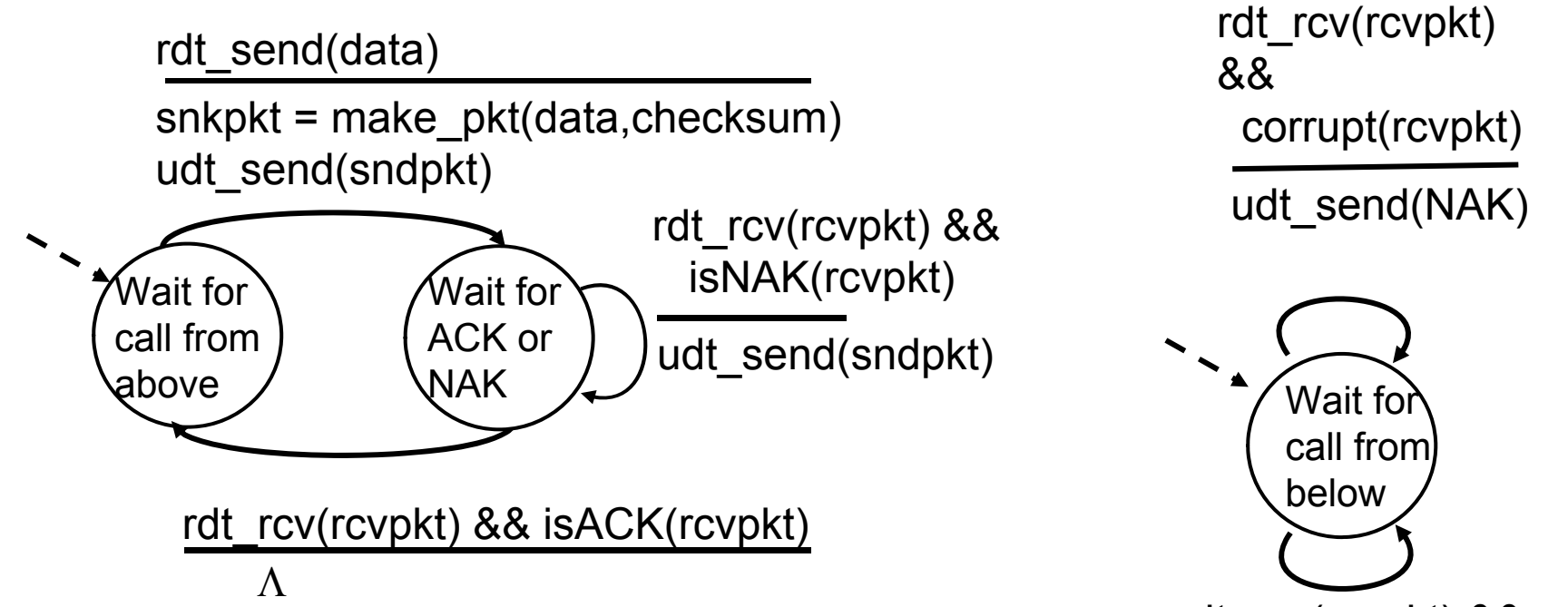
Verfälschungsbehandlung

- **A) ARQ (Automatic Repeat Request)** Verfahren
  - Fehlererkennung: Fehlererkennender Code
  - Fehlerbehebung: Wiederholung, eingeleitet durch Rückmeldungen

- **B) FEC (Forward Error Correction) Verfahren**
  - Fehlerkorrigierender Code

Wir nutzen erst einmal Variante A)

rdt2.0: Basisdienst nicht verfälschungsfrei (Bitfehler)



Sender

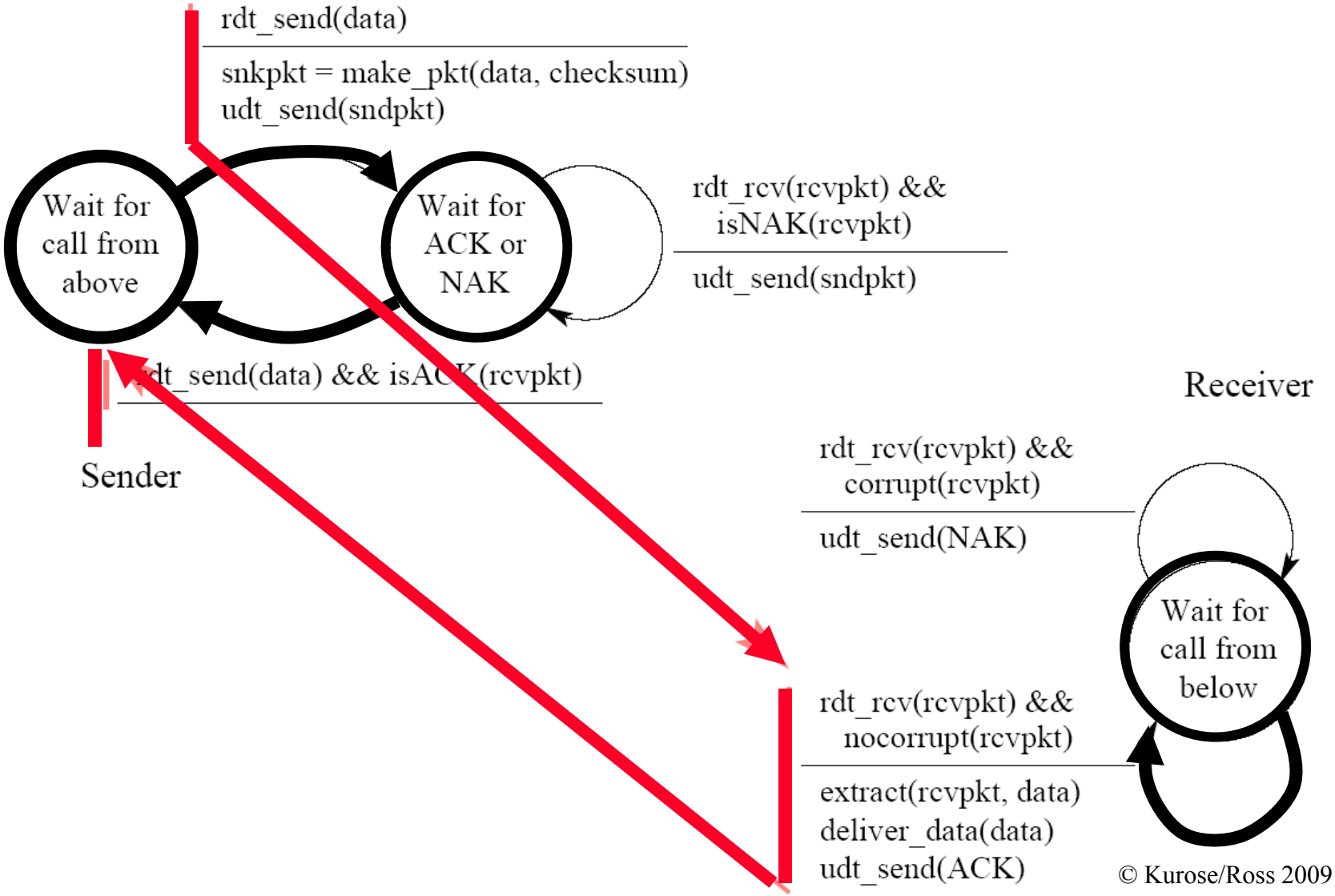
Stop and Wait – Protokoll:

1. **Sende** Nachricht
2. **Warte** auf Quittung (ACK / NAK)

Weiter bei 1.

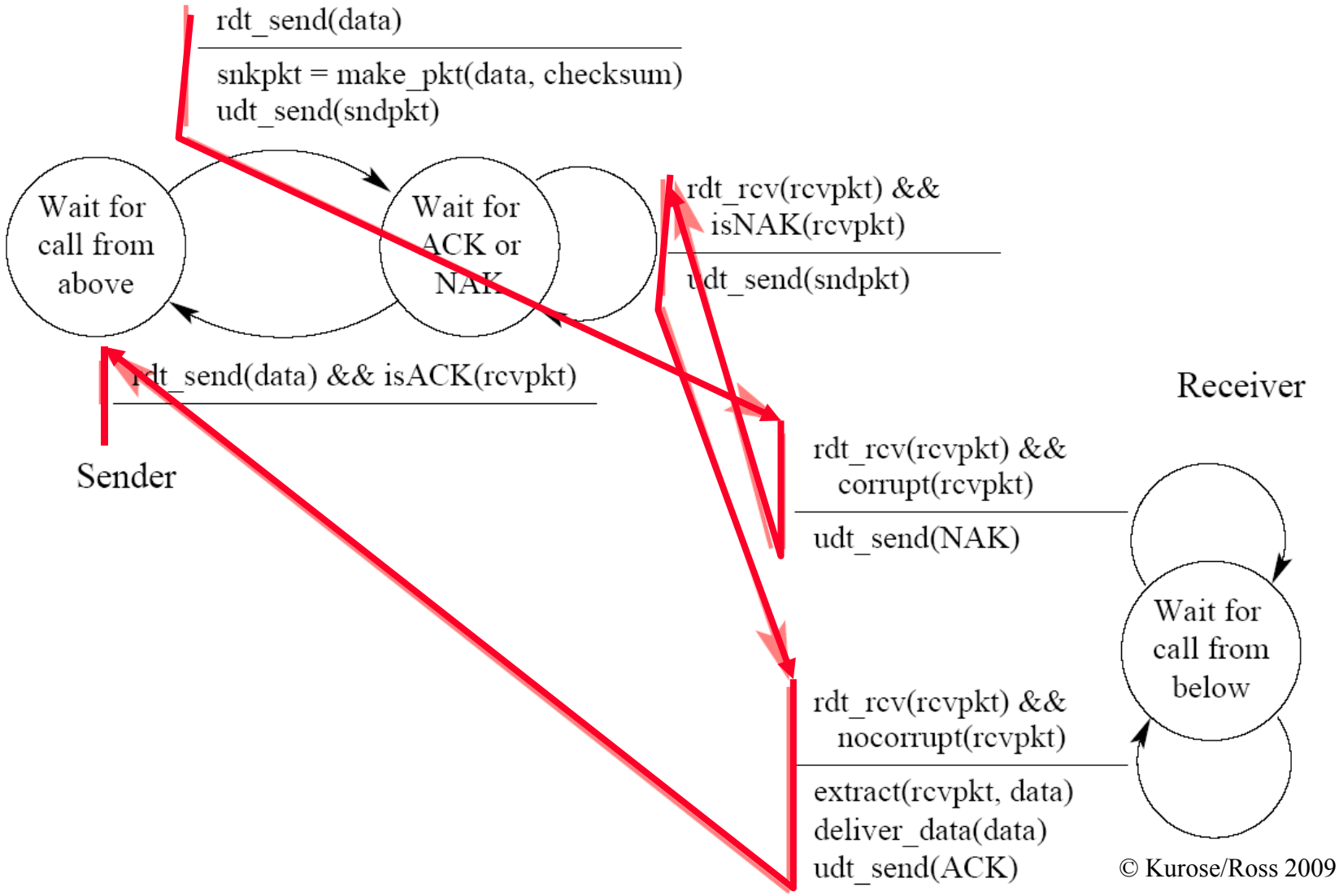
rdt2.0: Basisdienst nicht verfälschungsfrei (Bitfehler)

Szenario ohne Fehler



rdt2.0: Basisdienst nicht verfälschungsfrei (Bitfehler)

Szenario mit Fehlern



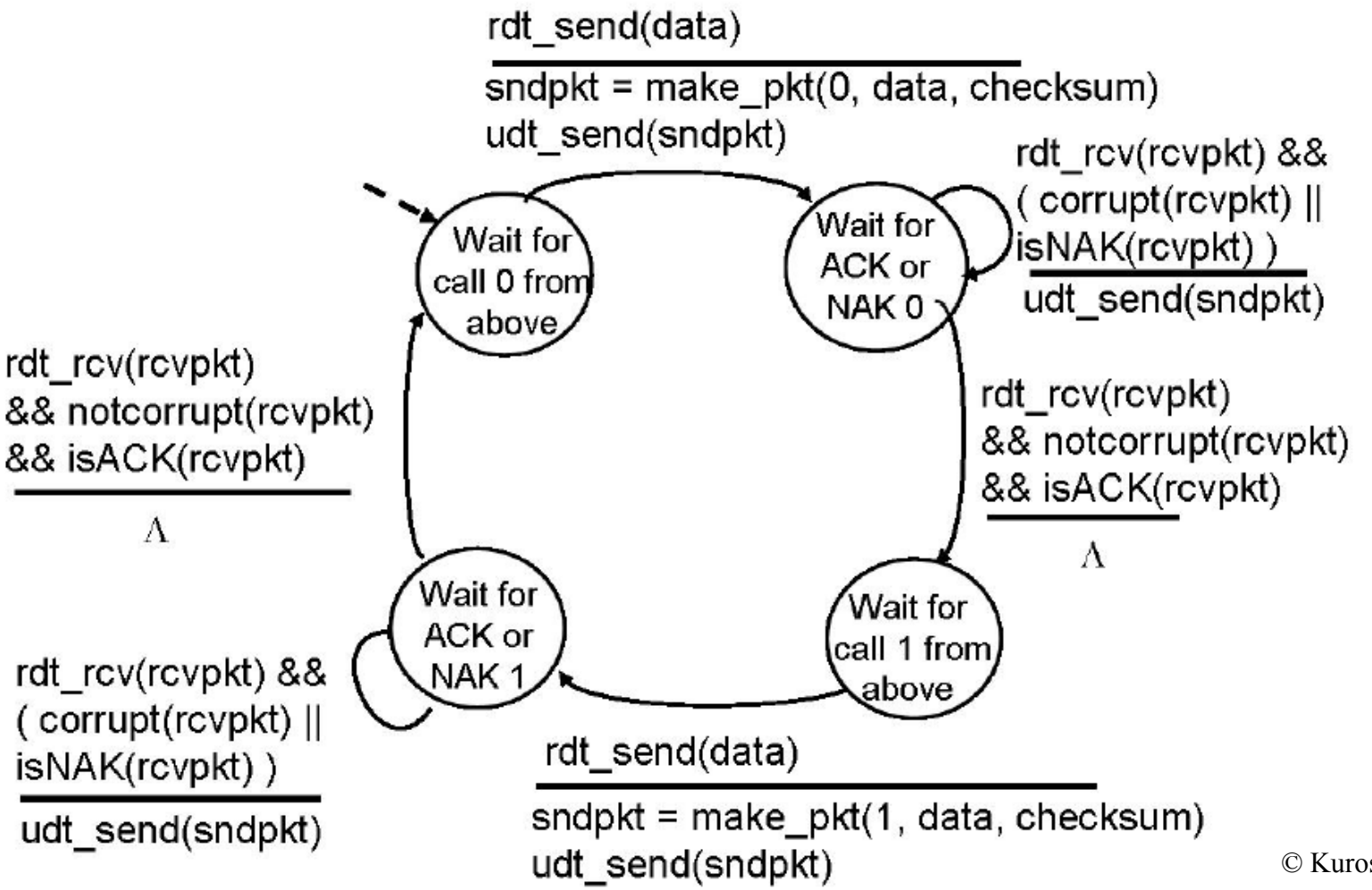
rdt2.0: Basisdienst nicht verfälschungsfrei (Bitfehler)

Entscheidender Mangel von rdt2.0

- Was passiert, wenn ACK/NAK fehlerhaft?
  - Sender weiß nicht, was empfangen wurde
  - Neusenden nicht möglich, wegen evtl. Doppelübertragungen
- Lösung des Problems?
  - Paket erneut senden, falls ACK/NAK beschädigt
  - Evtl. Doppelübertragungen korrekt empfangener Pakete
- Behandlung von Doppelübertragungen?
  - Sender nummeriert Pakete (Sequenznummern)

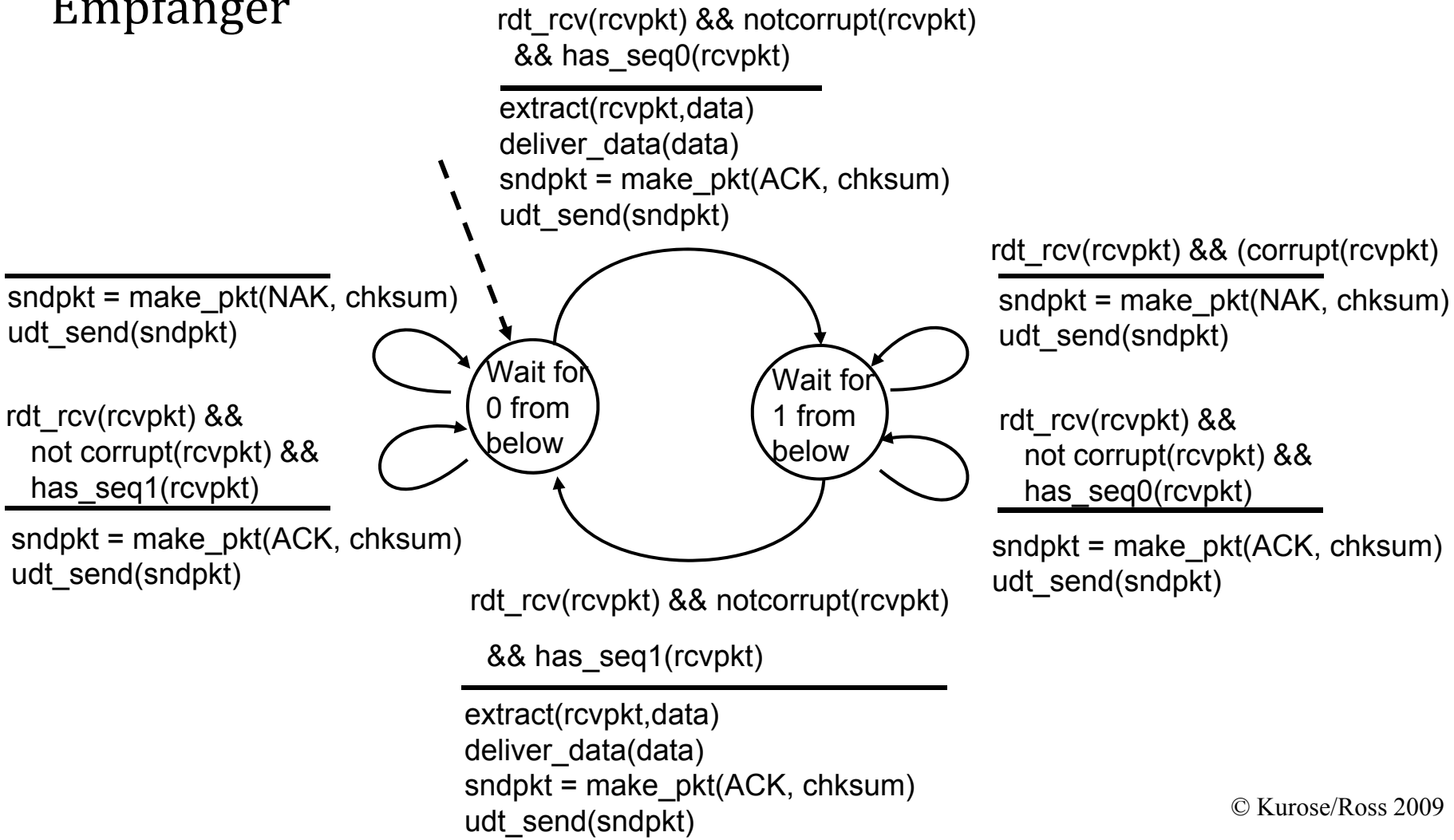
rdt2.1: Sequenzzahlen (0..1)

rdt 2.1 (Sender)



rdt2.1: Sequenzzahlen (0..1)

Empfänger



rdt2.1: Sequenzzahlen (0..1)

Bewertung von rdt2.1

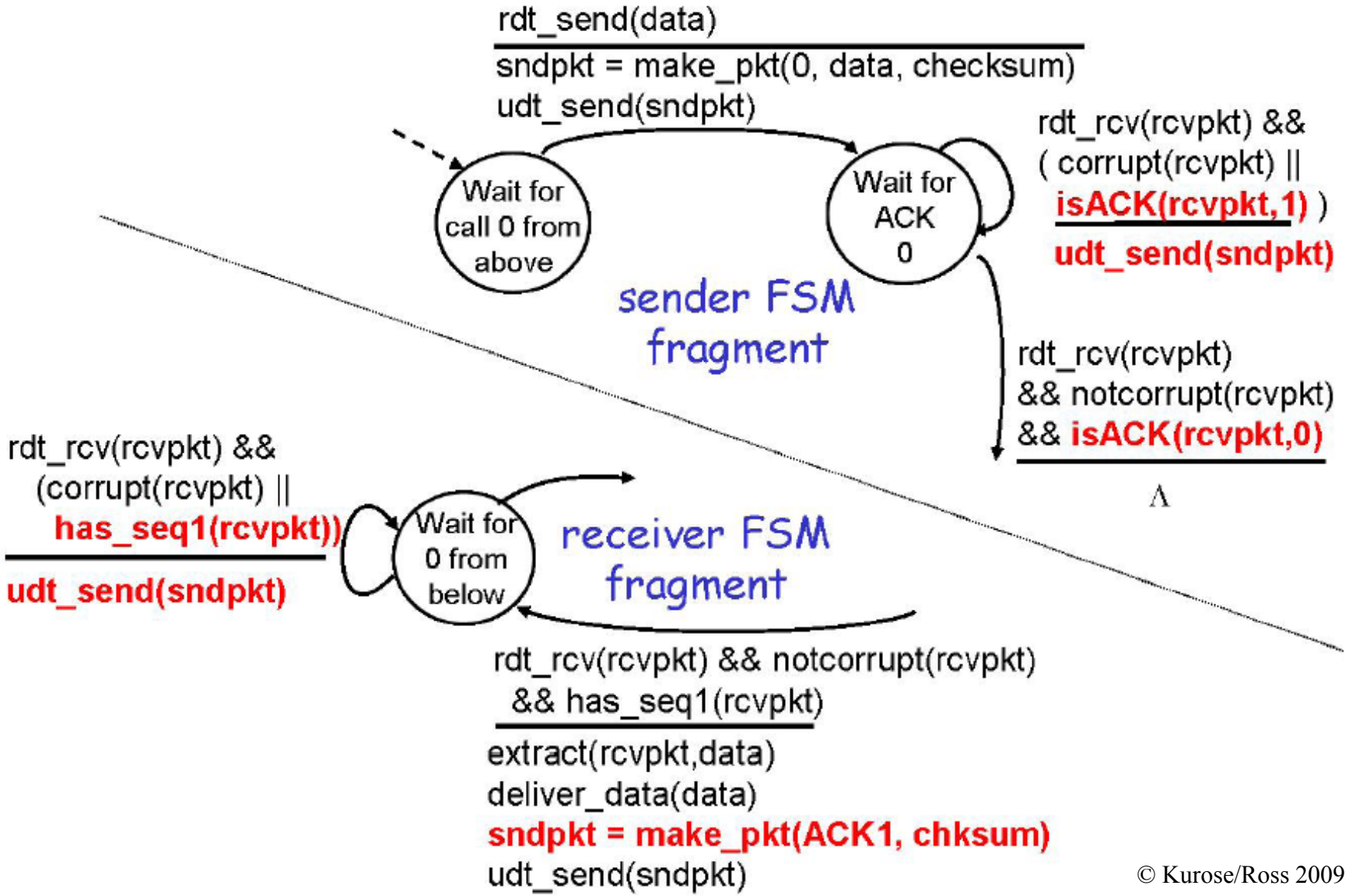
- Sender
  - Fügt Sequenznummer zu jedem Paket hinzu
    Reichen zwei Nummer (0,1) aus? Warum!?
  - Muss jedes empfangene ACK/NAK auf Korrektheit überprüfen
  - Hat die doppelte Anzahl von Zuständen
- Empfänger
  - Muss Pakete auf doppelte Übertragung untersuchen
  - Weiß nicht, ob der Sender das ACK/NAK empfangen hat

rdt2.2: Nur positive Quittungen

rdt 2.2 (Protokoll ohne NAKs)

- Funktionalität wie rdt 2.1, ohne NAKs zu nutzen
- Statt eines NAKs, sendet der Empfänger ein ACK für das letzte fehlerfrei empfangene Paket
- Auf doppelte Acks reagiert der Sender wie auf NAKs: Erneutes Senden das aktuellen Pakets

rdt2.2: Nur positive Quittungen



rdt3.0: Basisdienst auch verlustbehaftet

rdt 3.0: Kanäle mit Fehlern und Verlusten

Neue Annahme

Der zugrundeliegende Kanal kann Pakete (Daten oder ACKs) verlieren

Wie wird mit Verlusten umgegangen?

- Sender wartet, bis er sicher ist, dass Daten oder ACK verloren sind und sendet das Paket erneut
- Wie lange muss er warten, bis er sicher sein kann?

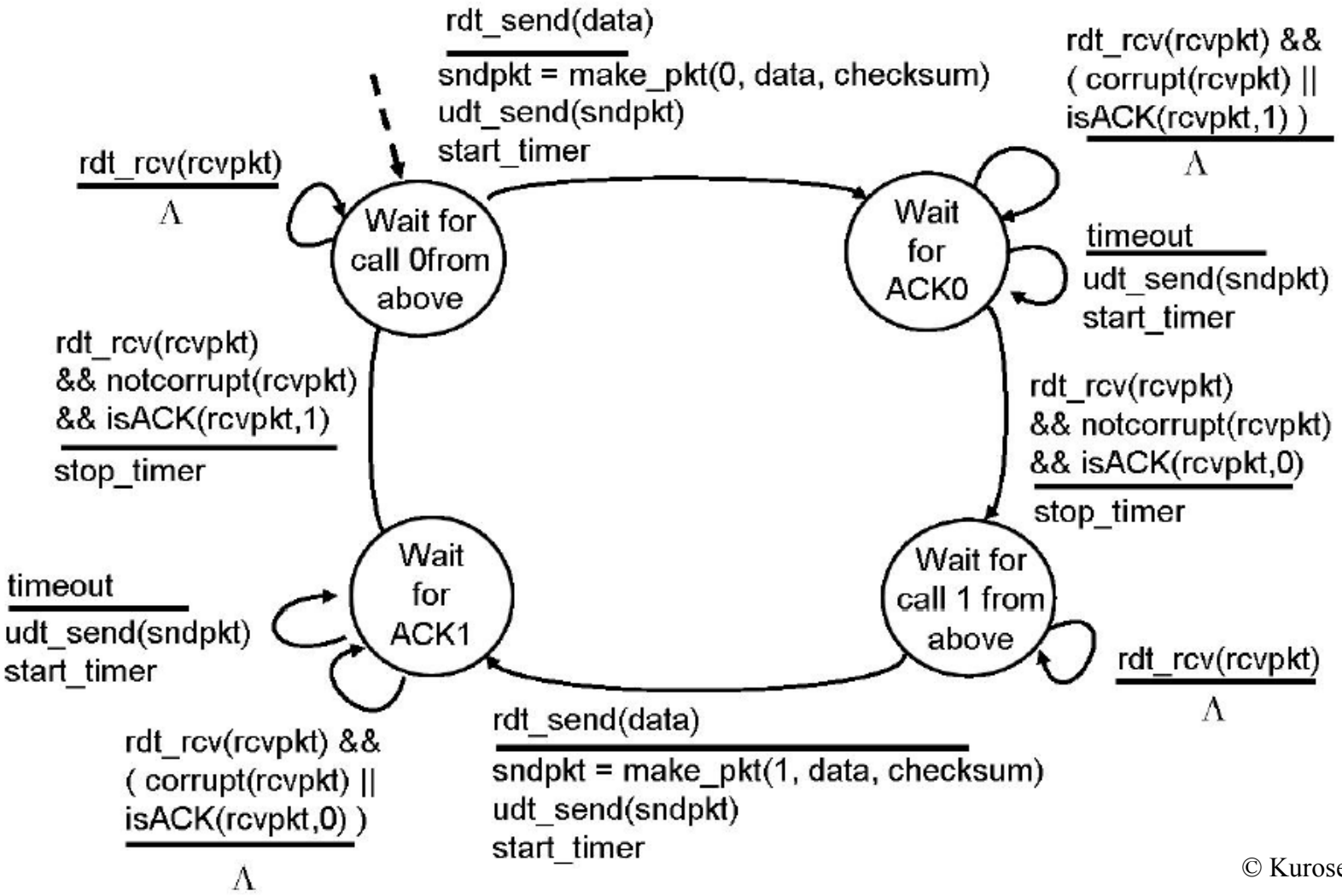
rdt3.0: Time-Out Mechanismus

- Sender wartete nur eine festgelegte Zeit auf ein ACK
  → Timer
- Erneutes Senden, wenn bis dahin kein ACK empfangen wurde
- Wenn ein Paket oder ACK nicht verloren, sondern nur verzögert, wird das Paket doppelt gesendet.
  → wird wegen Sequenznummer erkannt
- Sequenznummer muss auch im ACK angegeben werden

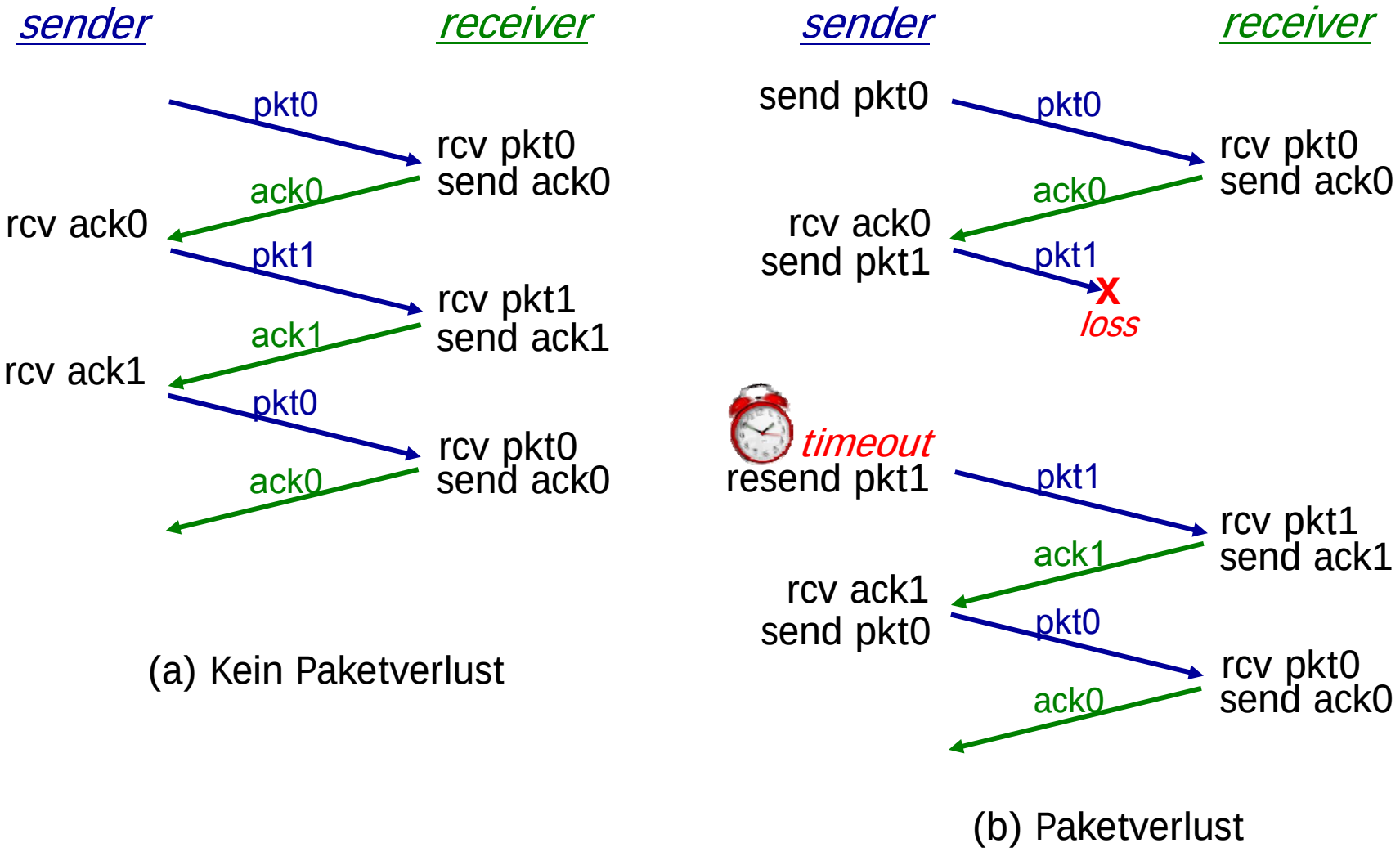
1. Beim Senden einer Nachricht: Kurzzeitwecker starten
2. Warten auf Quittung ODER Weckeralarm
3. Wecker stoppen
4. Weckeralarm wie negative Quittung behandeln: Übertragung wiederholen

rdt3.0

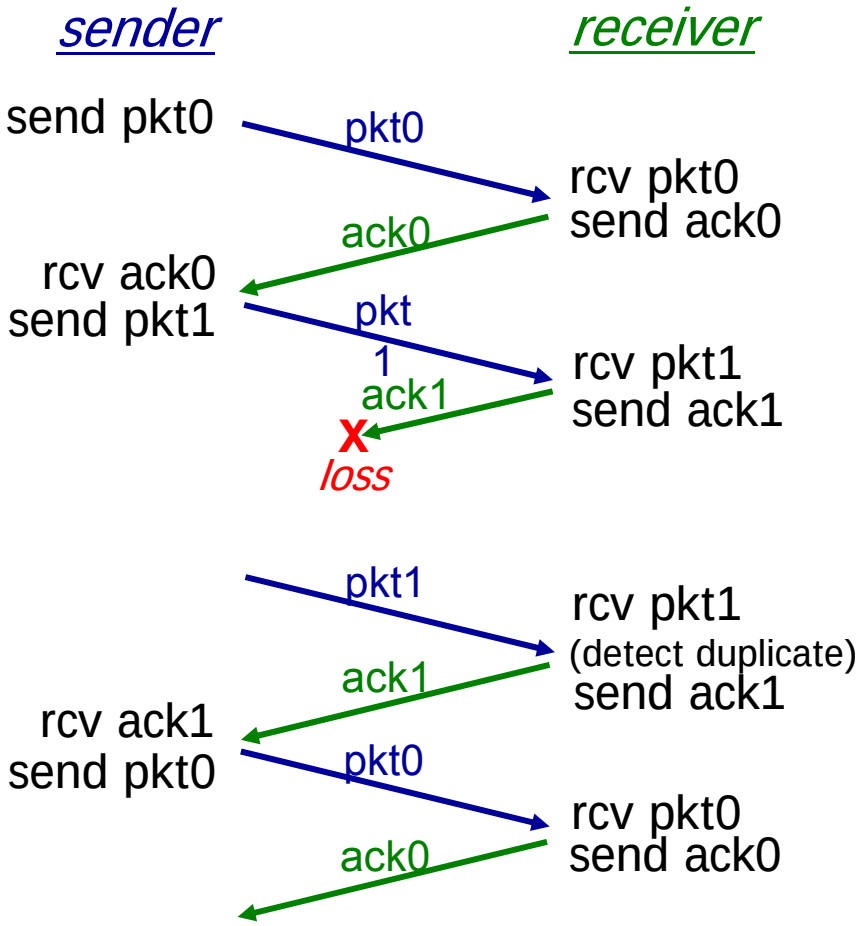
rdt 3.0 Sender



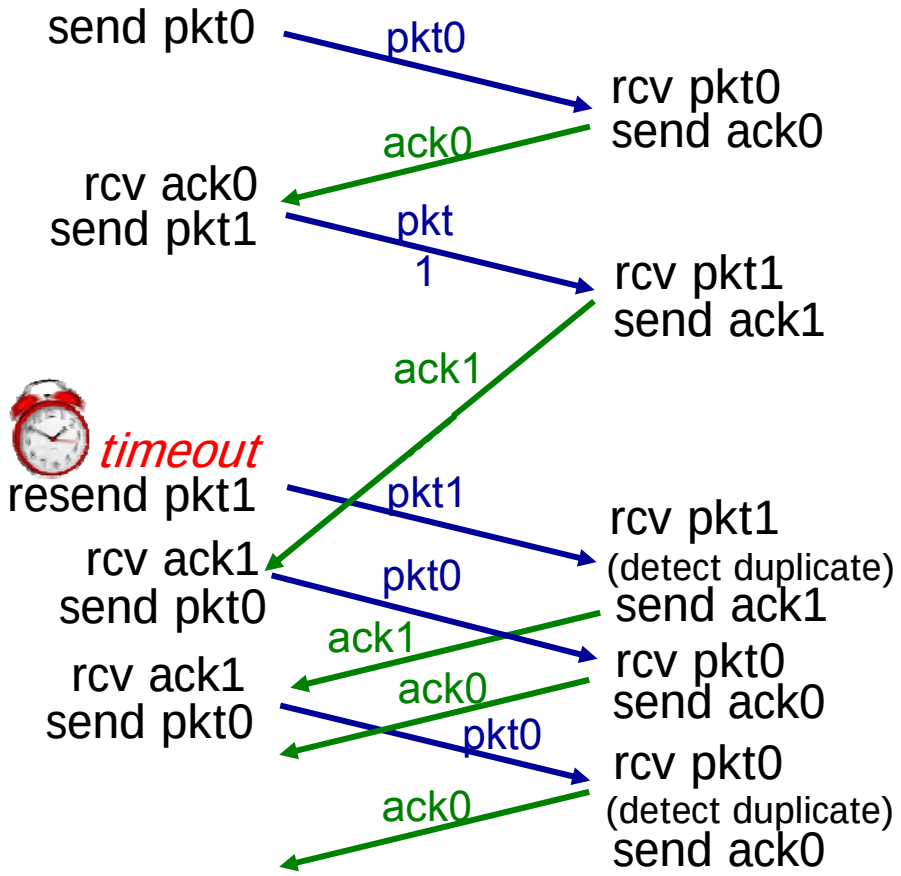
rdt3.0: Abläufe im Weg/Zeitdiagramm



rdt3.0: Abläufe im Weg/Zeitdiagramm

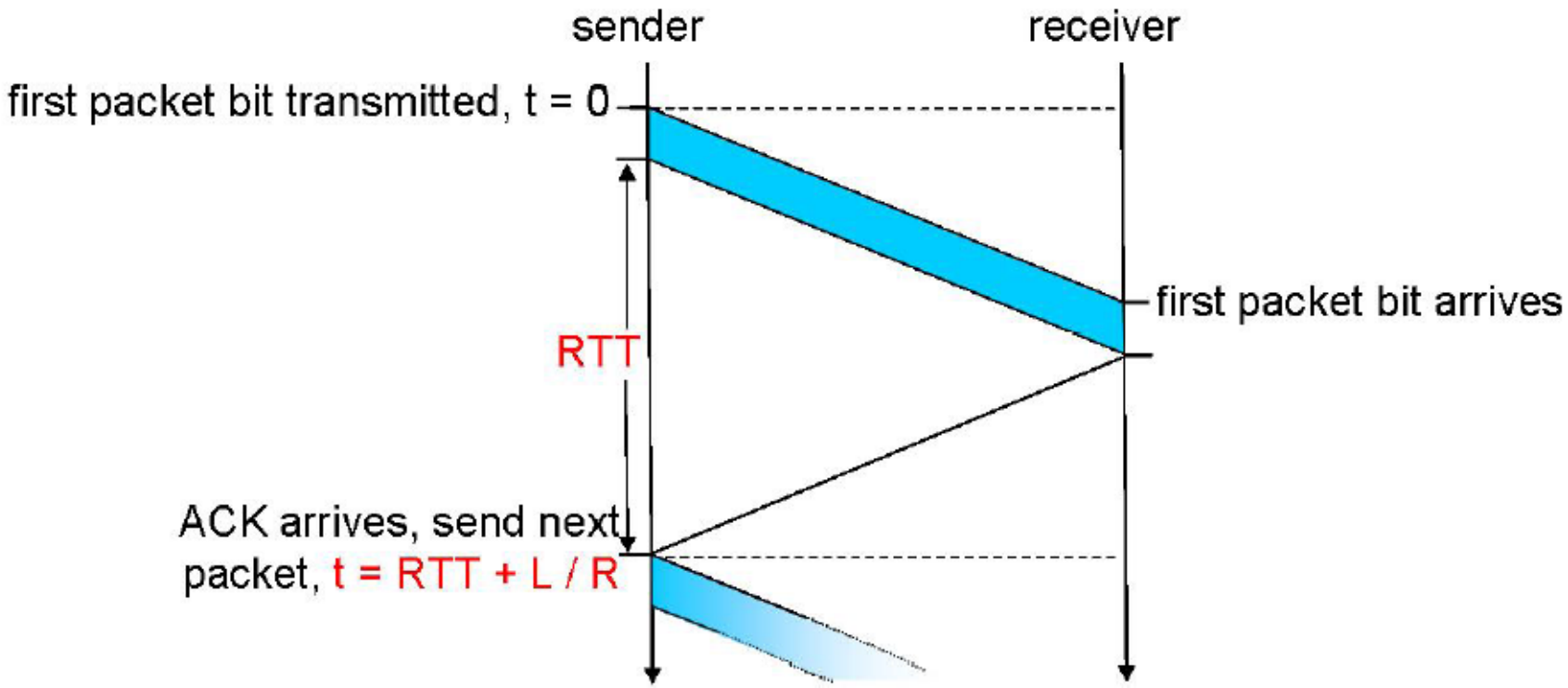


(c) ACK-Verlust



(d) Verzögertes ACK/Timerablauf

rdt3.0: Problem „Stop and Wait“

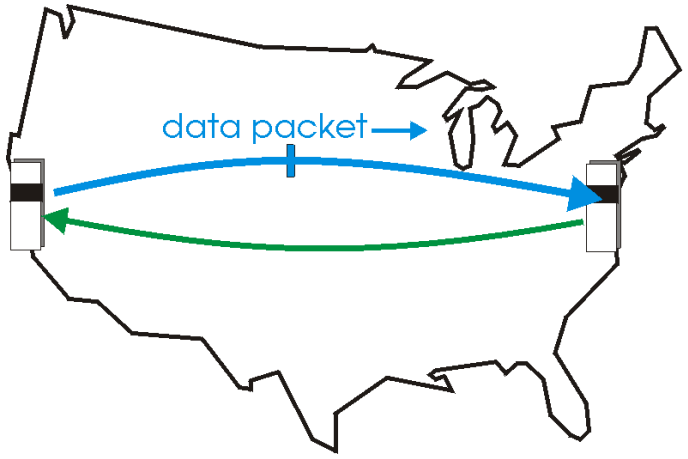


L Paketlänge, R Bandbreite der Verbindung

rdt3.0: Leistungsfähigkeit (Performance)

- rdt3.0 bietet die angestrebte Funktionalität
  (zuverlässige Übertragung über ein unzuverlässiges Medium)
- die Leistung ist grauenvoll
  (vorhandene Ressourcen werden nicht genutzt!)

Eine einfache Beispielrechnung:



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

RTT = 30 ms (Lichtgeschwindigkeit!)

R = 1 Gbps (10^9 bps) (Übertragungsrate)

L = 1000 bytes (Paketlänge)

t_{trans} = 8 µs (Übertragungszeit)

Auslastung des Senders

U_{Sender} = (L/R) / (RTT+L/R)
= 0.008/30.008 = 0.00027

Erreichbarer Durchsatz

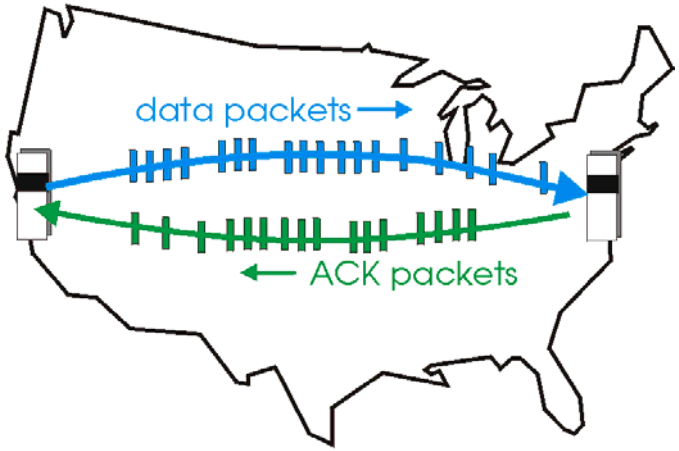
1000 bytes in 30.0008 ms = 33.3 kB/sec

Über eine 1 Gbps Leitung!

Pipeline Protokolle

Wie geht es besser?

- Neues Paket schon senden, bevor ACK eingetroffen
- Aber nicht beliebig viele, da Pakete gespeichert werden müssen
- Umfang der Sequenznummer erhöhen (Einführung eines Kreditrahmens bzw. Fensters)



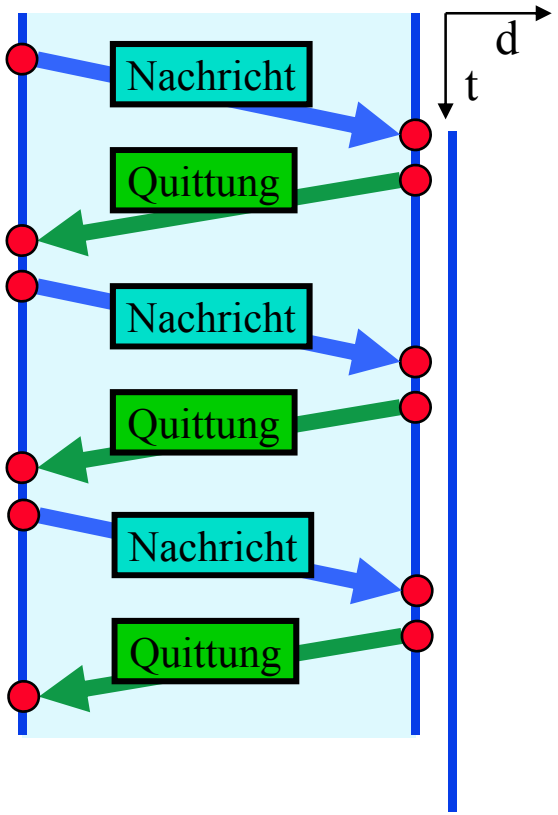
(b) a pipelined protocol in operation

Zwei Realisierungsmöglichkeiten:

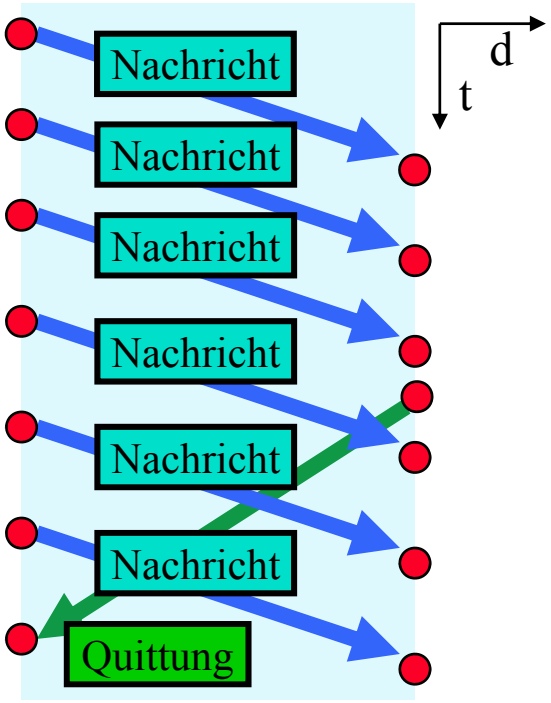
- Go-back-n
- Selective-repeat

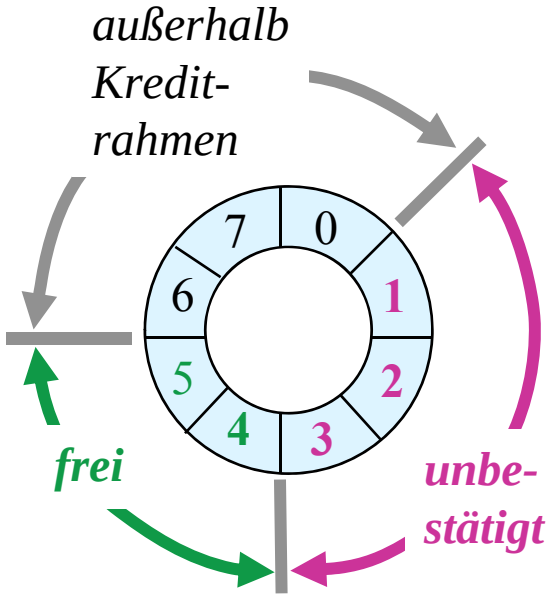
Pipeline Protokolle

- Stop and Go - Protokolle

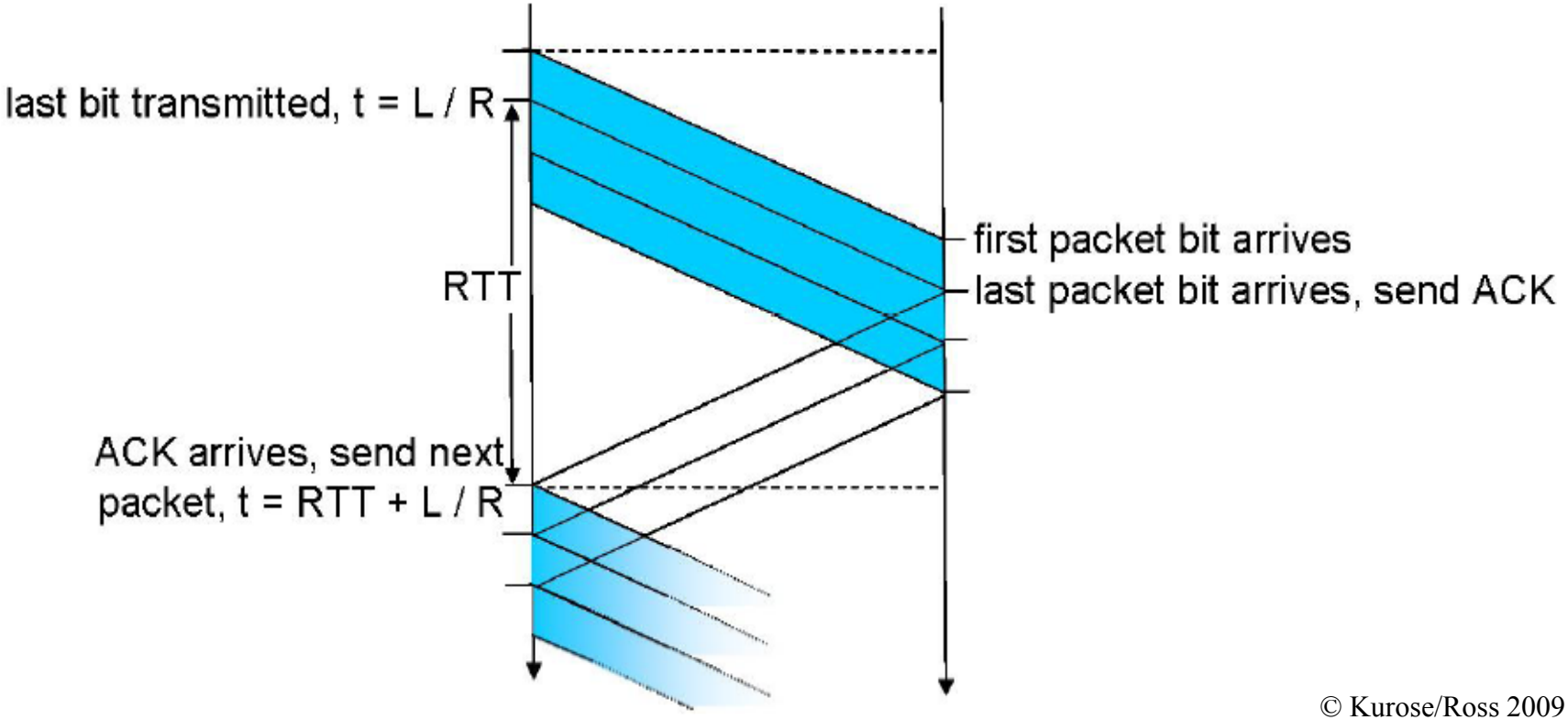


- Sliding-Window - Protokolle
  - Sendekredit
  - Summenquittung
  - Wiederkehrende Laufnummern





Pipeline Protokolle: Bessere Kanalausnutzung



© Kurose/Ross 2009

Unser Beispiel für Fenstergröße 3:

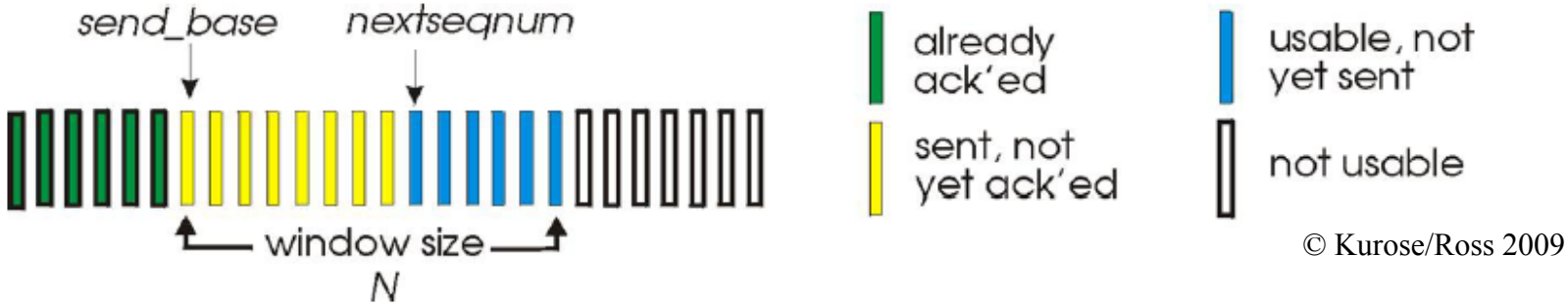
$U_{Sender} = (3L/R)/(RTT+L/R) = 0.024 / (30.008) = 0.0008$

bzw. Übertragungsrate 100 Kps

Pipeline Protokolle: Go back n

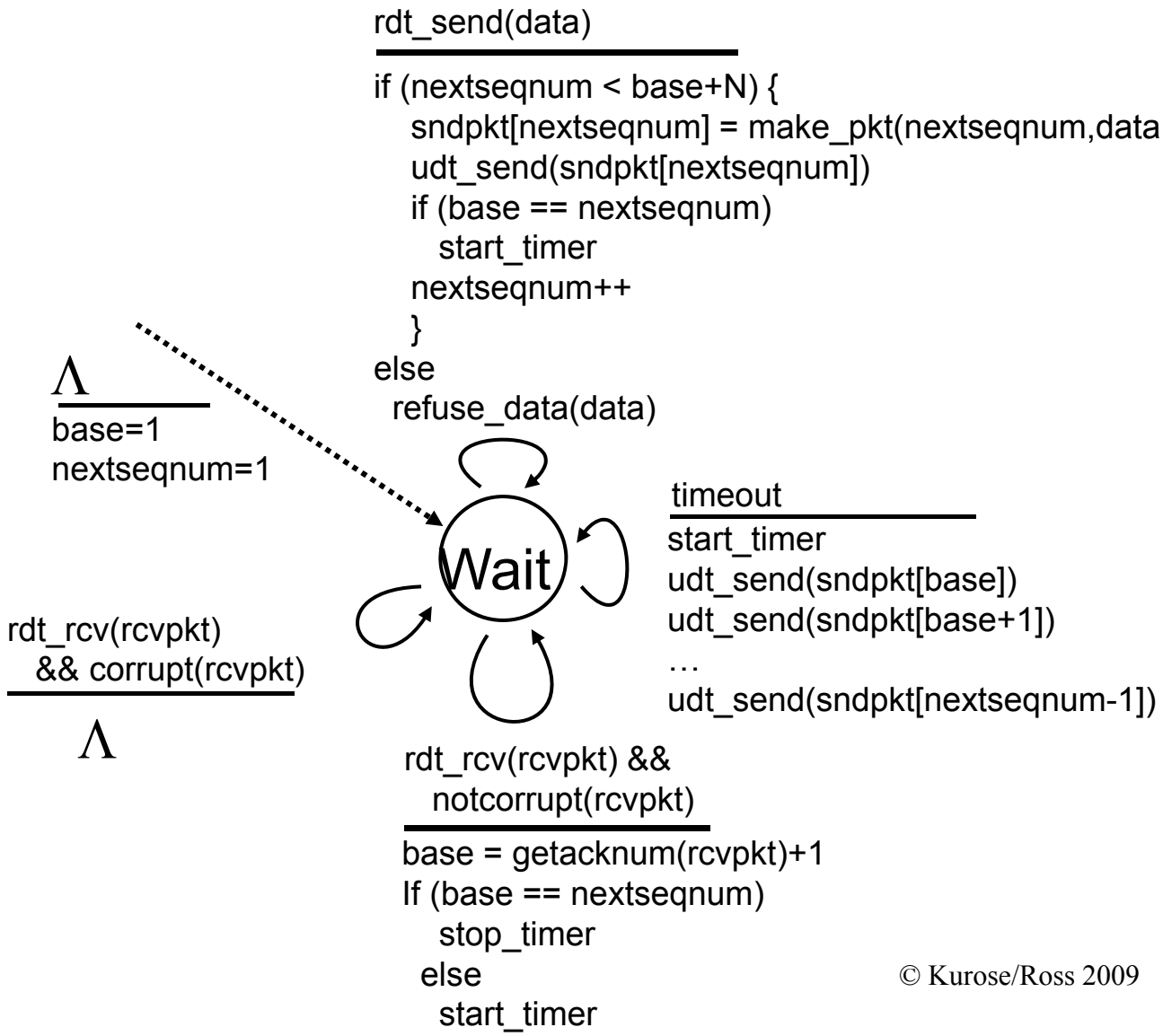
Sender

- K-bit Sequenznummer im Paket-Header
  (Adressarithmetik modulo 2^K)
- Fenster von bis zu N nicht bestätigten Paketen

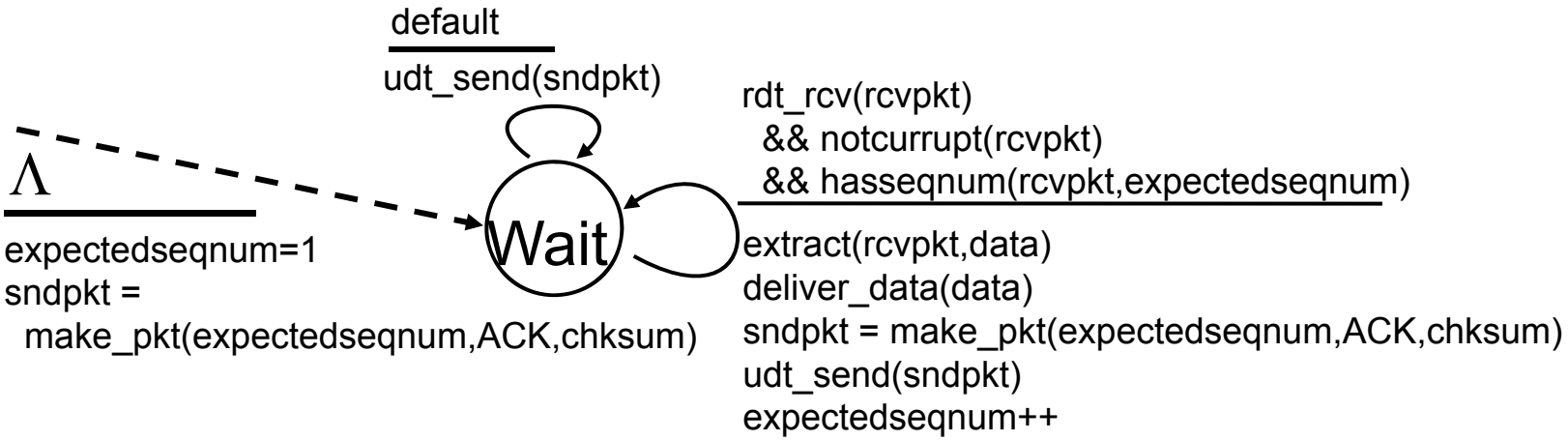


- ACK für Paket n bestätigt alle Pakete die bis zu Paket n gesendet wurden
  (Sammel-ACK bzw. Cumulative ACK)
- Ein Timer läuft jeweils für das älteste nicht bestätigte Paket
  bei Eintreffen eines ACKs wird Timer neu gestartet
  (sofern noch nicht alle ACKs eingetroffen)
- Bei Timerablauf Wiederholung aller noch nicht bestätigten Pakete

Pipeline Protokolle: Go back n - Sender

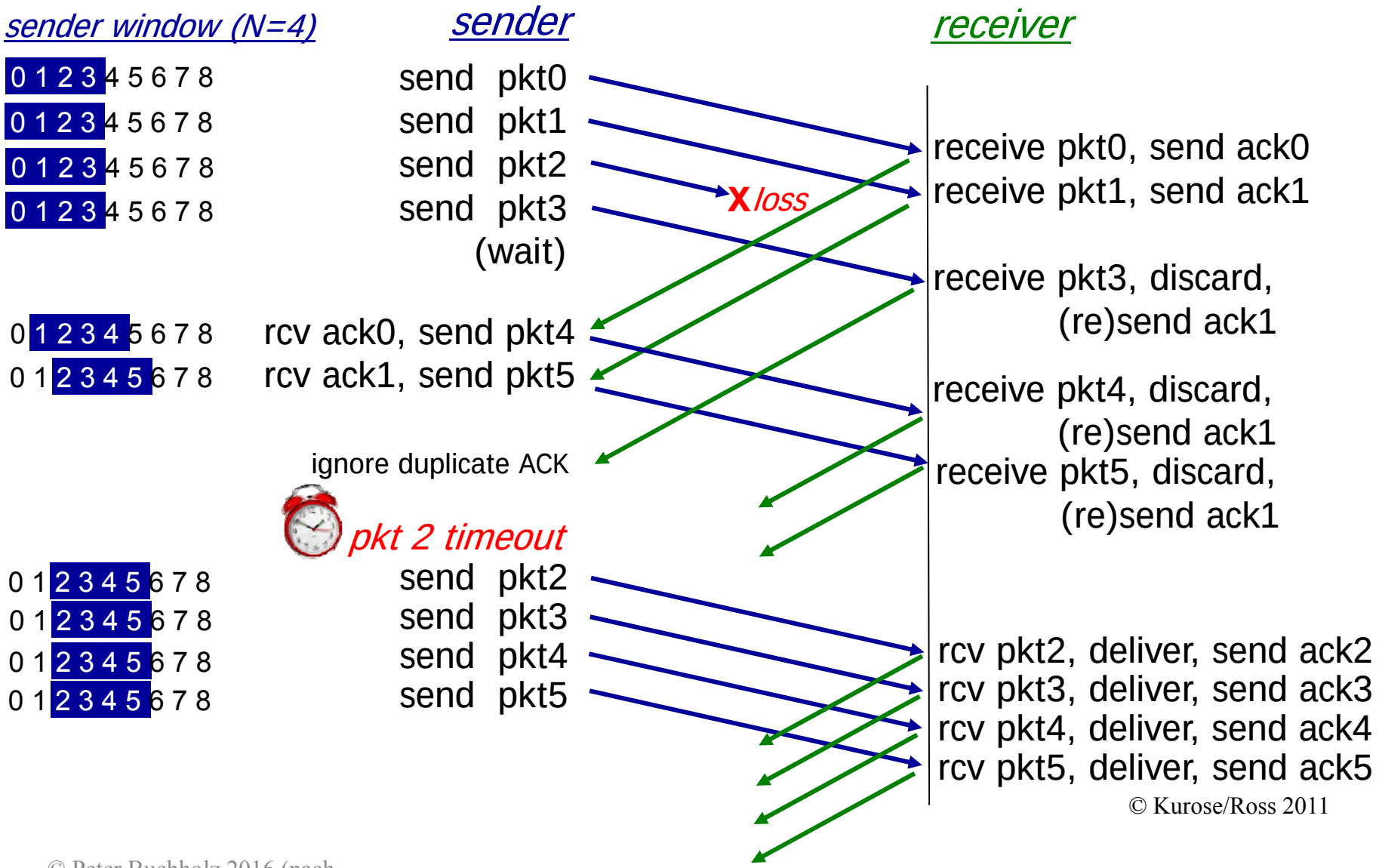


Pipeline Protokolle: Go back n - Empfänger



- ACK für das korrekt empfangene Paket mit der größten Sequenznummer
- Empfang von Paketen in falscher Reihenfolge: wegwerfen, ACK für Paket mit größter Sequenznummer, das in richtiger Reihenfolge empfangen wurde

Pipeline Protokolle: Go back n - Ablauf



Pipeline Protokolle: Selective Repeat

Empfänger

- bestätigt jedes korrekt empfangene Paket
- puffert Pakete, die in falscher Reihenfolge empfangen wurden

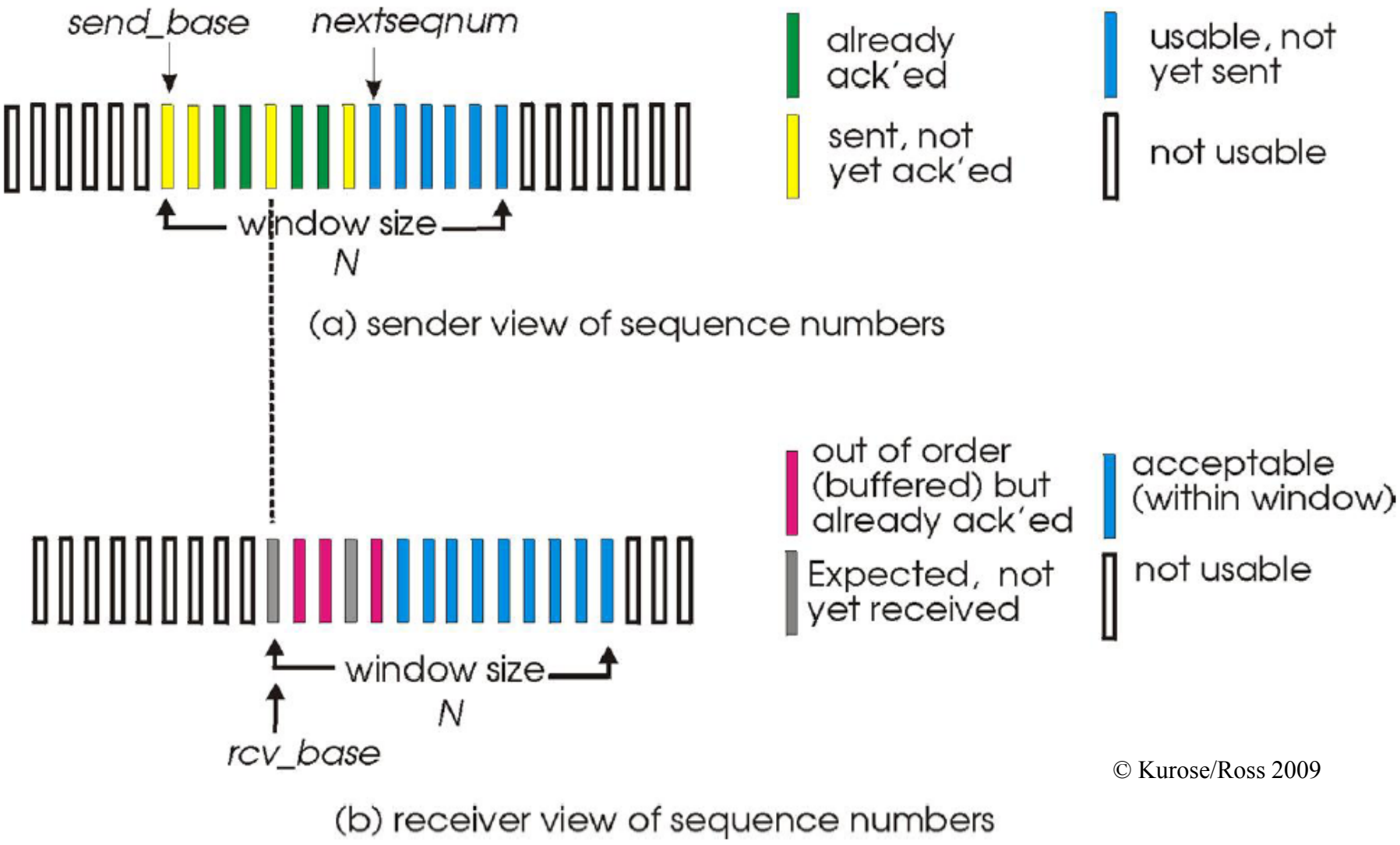
Sender

- sendet nur unbestätigte Pakete erneut
- setzt einen Timer für jedes Paket

Sende-Fenster

- besteht aus N konsekutiven Sequenznummern
- begrenzt die Anzahl gesendeter unbestätigter Pakete

Pipeline Protokolle: Selective Repeat - Beispielfenster



Pipeline Protokolle: Selective Repeat

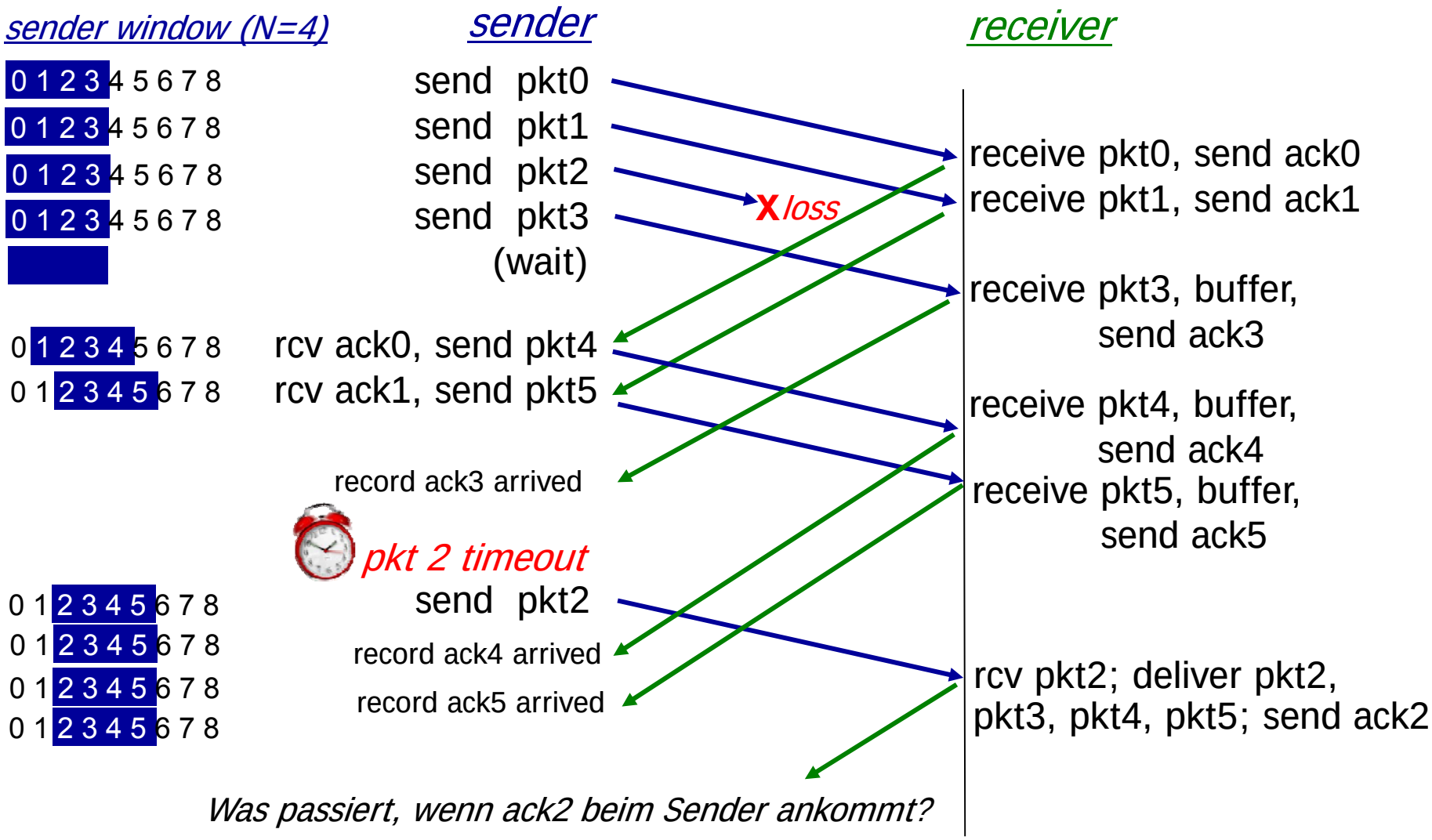
Sender

- sendet Paket, wenn nächste Sequenznummer im Fenster liegt
- sendet bei Timeout Paket nochmals und startet Timer neu
- markiert bestätigte Pakete
- verschiebt das Fenster, wenn Paket mit kleinster Sequenznummer bestätigt wird

Empfänger

- bestätigt Pakete, deren Sequenznummern im Fenster liegen
- ignoriert andere Pakete
- puffert Pakete, die in falscher Reihenfolge empfangen wurden
- liefert Pakete aus, die in richtiger Reihenfolge empfangen wurden

Pipeline Protokolle: Selective Repeat - Ablauf

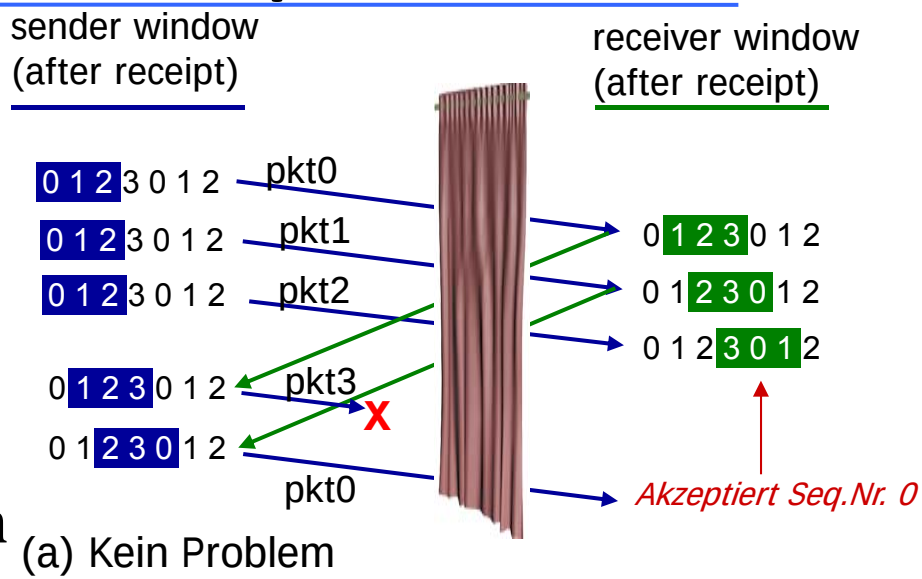


Pipeline Protokolle: Selective Repeat - Ablauf

Problem-Szenario

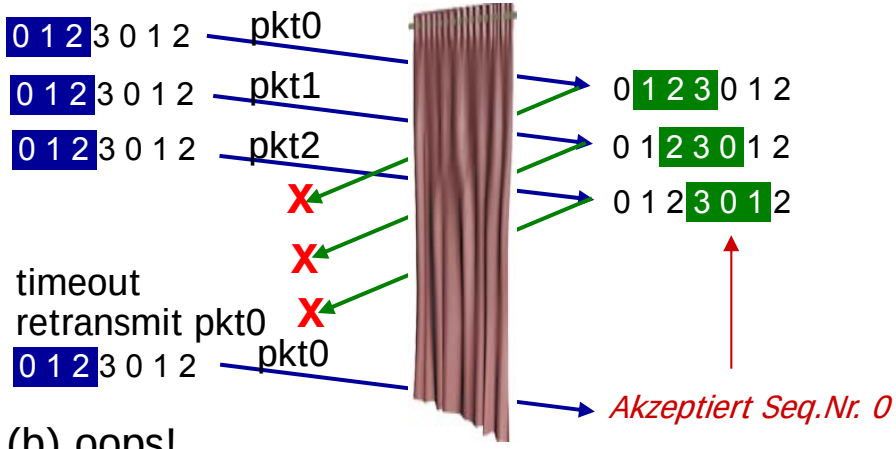
- Sequenznummern 0,1,2,3
- Fenstergröße = 3
- Empfänger sieht keinen Unterschied zwischen beiden Szenarien
- In (a) wird wiederholtes Paket als neues empfangen

Welche Beziehung muss zwischen der Anzahl der Sequenznummern und der Fenstergröße bestehen?



(a) Kein Problem

Sender weiß nicht, was beim Empfänger passiert



(b) oops!

Zuverlässige Übertragung

Mechanismen zur Unterstützung zuverlässiger Übertragungen:

Mechanismus	Benutzung
Prüfsummen	Fehlererkennung
Timer	Verluste, aber doppelte Übertragungen möglich
Sequenznummern	Lücken im Paketstrom und doppelte Übertragungen können erkannt werden
ACKs	Bestätigung des korrekten Empfangs unter Nutzung der Sequenznummer Auch kumulativ möglich
NAKs	Empfänger teilt Sender mit, dass bestimmtes Paket nicht korrekt empfangen wurde Alternative ausbleibendes ACK + Timeout
Fenster, Pipelining	Effizienzsteigerung und Flusskontrolle

Verbindungsorientierter Transport im Internet: TCP

Zieldienst: **Zuverlässiger Duplex-Bytestrom-Transfer**

Basisdienst:

Internet – Unzuverlässiger Transfer von IP-Paketen

→

Protokollmechanismen zur zuverlässigen Übertragung a la **rdt3.0**

Weiterhin

- Verbindungsverwaltung (verbindungsorientiert):
  Aufbau über 3-Wege-Handshake, Abbau über *close* je Richtung
- Voll-Duplex und Piggy-Backing

Sowie später erklärt

- Stau- und Flusskontrolle

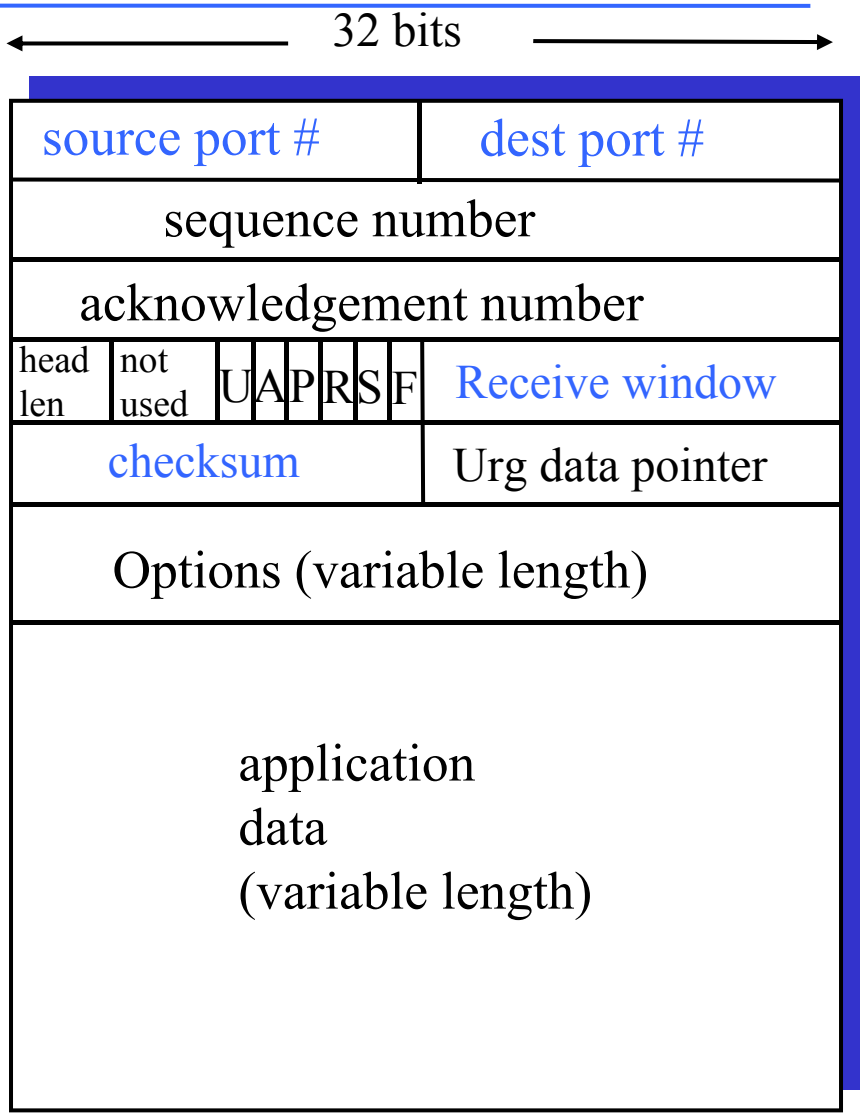
TCP: PDU-Format

Source port (16 bit):
Nummer des Quellports

Destination port (16 bit):
Nummer des Zielports

checksum (16 bit): wie bei UDP beschrieben

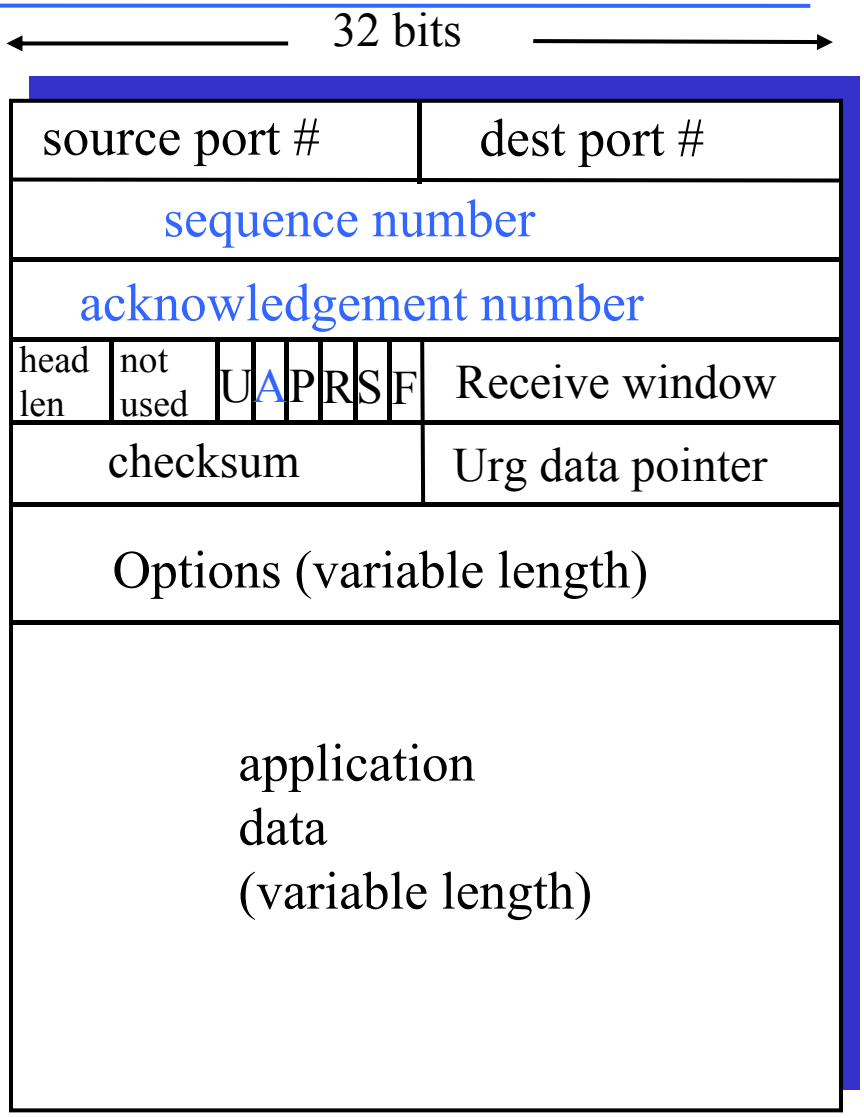
Receive window (16 bit):
Größe des Empfangsfensters, wird zur Flusskontrolle verwendet



TCP: PDU-Format

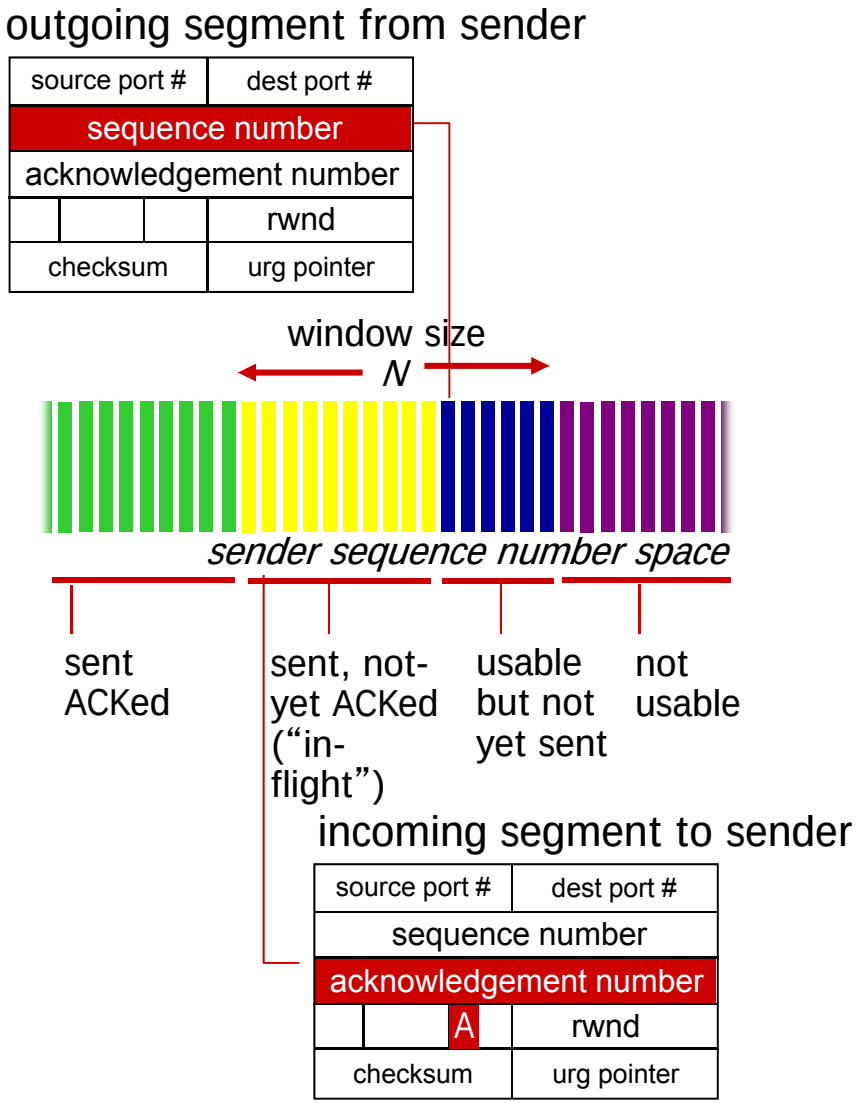
Sequence Number (32 bit):
Nummer des ersten Datenocktets dieses Segments

Acknowledgement Number (32 bit):
Nummer des nächsten vom Sender erwarteten Segments
nur gültig, wenn A Bit gesetzt

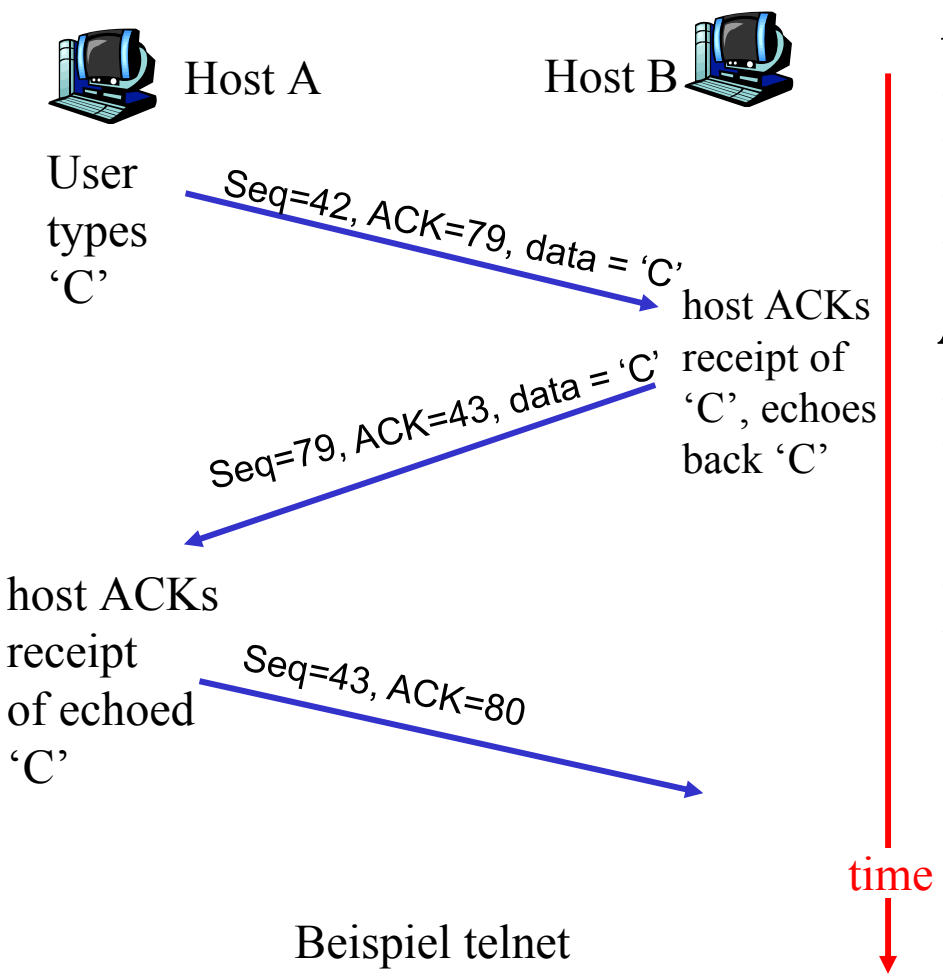


TCP: Sequenz- und ACK-Nummern

- Daten werden als Strom von Bytes interpretiert
- Sequenznummer des Sender := erstes Byte das gesendet wird
- Ack-Nummer des Empfängers (nur gültig, falls A-Bit gesetzt) nächstes erwartetes Byte (alle Bytes davor wurden empfangen)
- Zufällige Auswahl der initialen Sequenznummern
- Behandlung von Nachrichten außerhalb der Reihenfolge im RFC nicht festgelegt



Sequenz- und ACK-Nummer: Ein Beispiel



Beispiel telnet

Initial erwartet

- Host A Byte 79
- Host B Byte 42

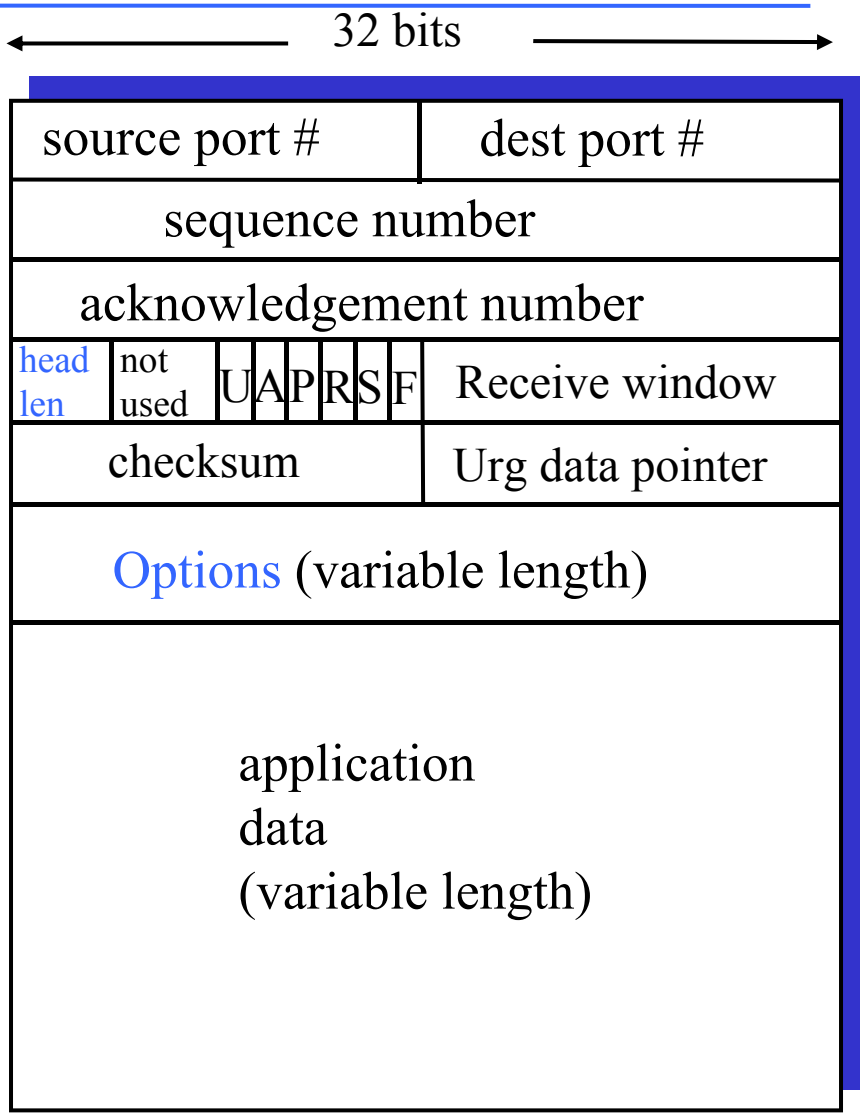
Ablauf

- Host A initiiert Kommunikation mit Seq.Nr. 42
- Host B antwortet mit 79 und bestätigt in diesem Paket 42 (ACK piggy-backed)

TCP: PDU-Format

Head len (4 bit): Länge des Headers in 32 Bit Worten (Länge des Headers kann variieren, da Optionen variable Länge haben können)

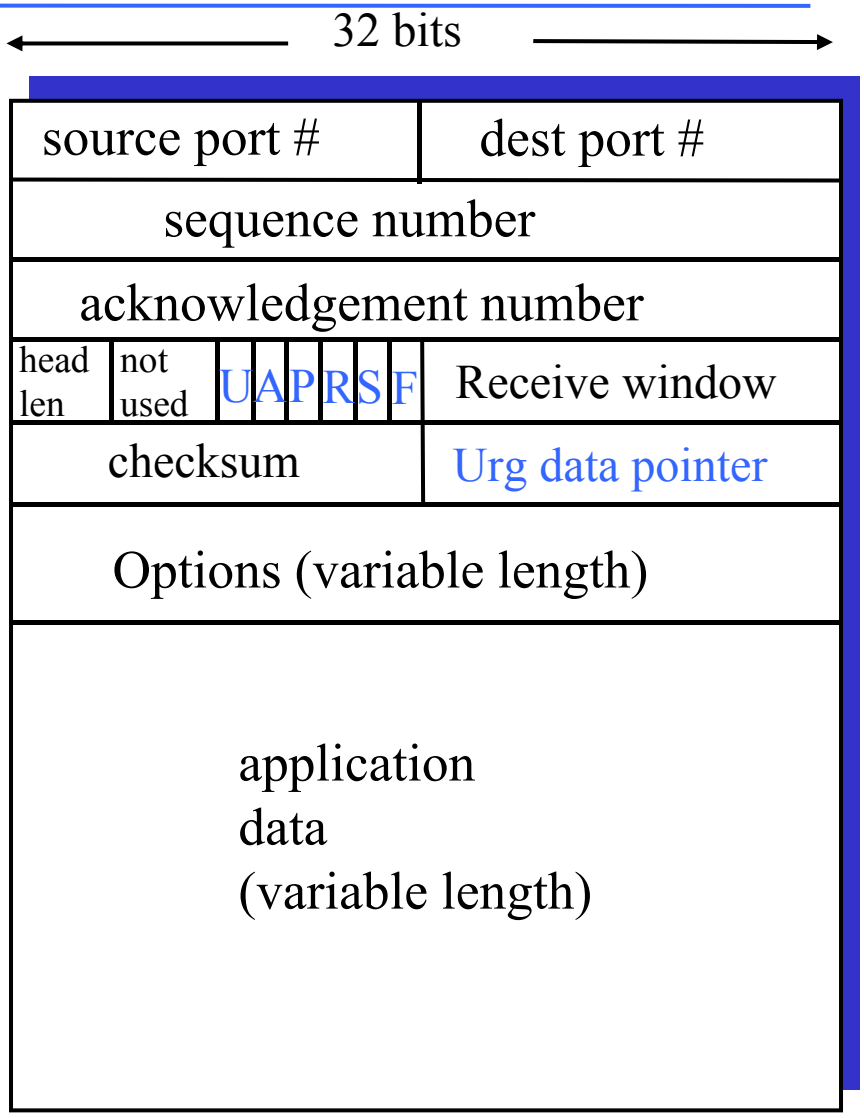
Options (variable Länge): Optionen werden für zusätzliche Kontrollinformationen genutzt



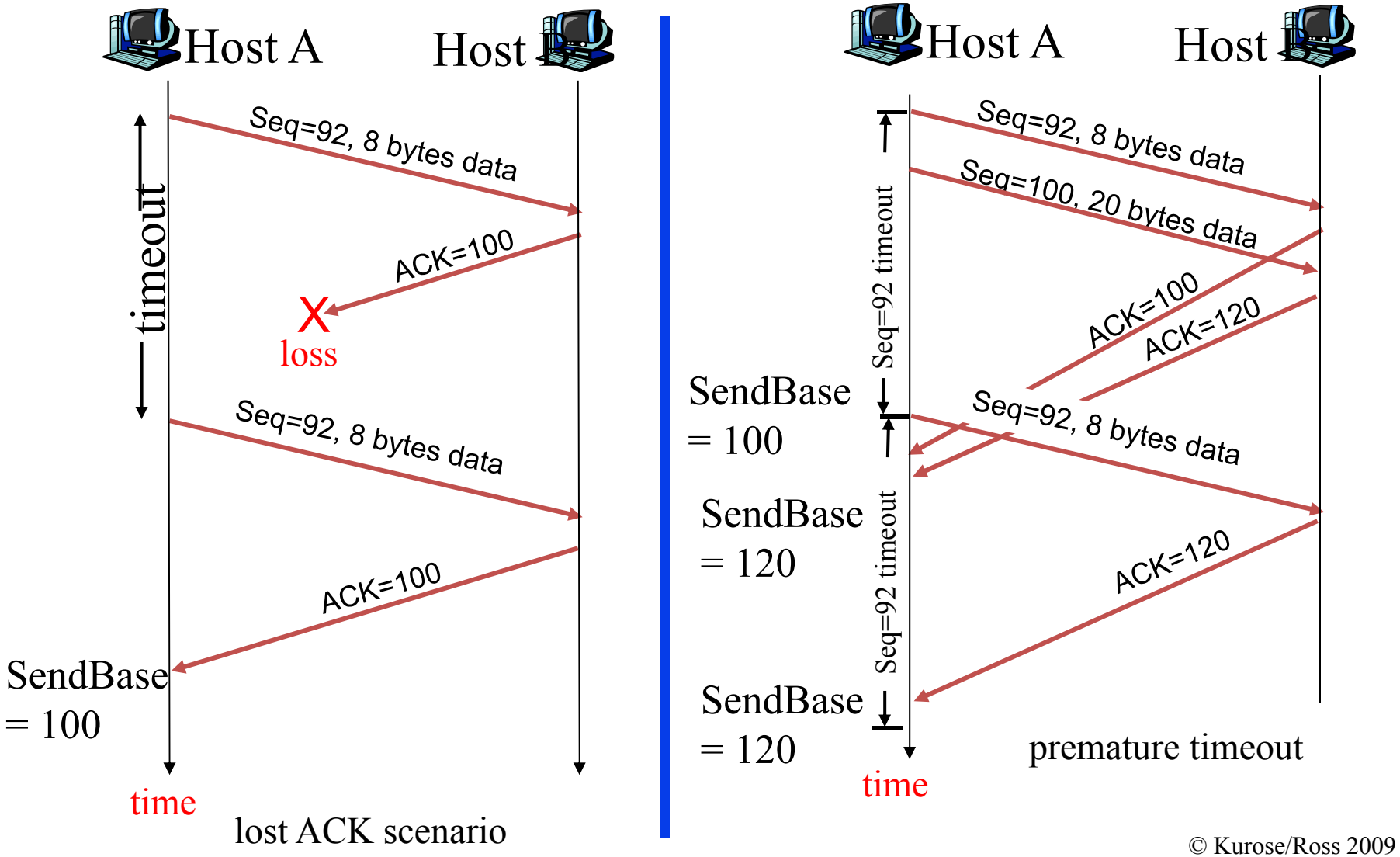
TCP: PDU-Format

Urg data pointer (16 bit):
Position des letzten dringenden Datenbytes im Segment (Flag U gesetzt)

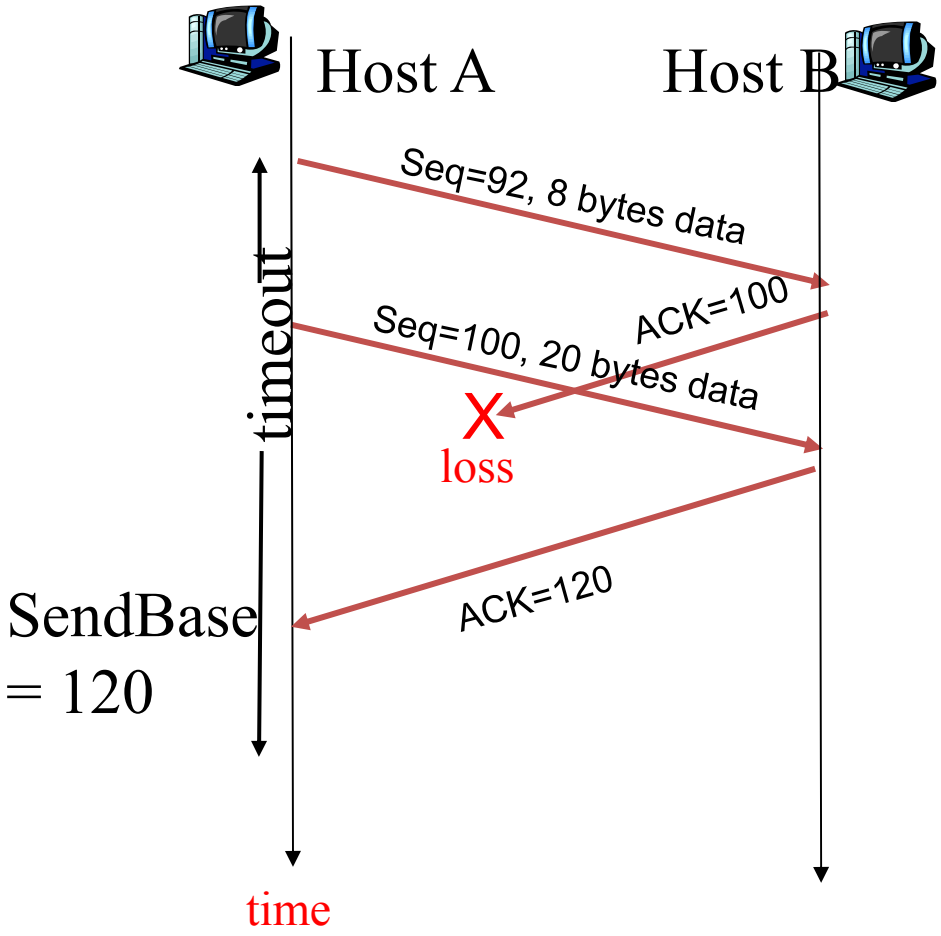
Flags (8 Bit):
A Acknowledgement gesetzt
P Daten sollen sofort an die Anwendungsschicht weitergereicht werden



TCP: Einige Szenarien



TCP: Einige Szenarien



Cumulative ACK scenario

TCP: Time-Out-Wert und RTT-Schätzung

Wie setzt TCP den Timeout-Wert?

- Zu klein → unnötige Wiederholungen
- Zu groß → unnötiges Warten, langsame Reaktion
- der Wert muss größer als RTT sein, aber nicht wesentlich größer! (aber RTT ändert sich)

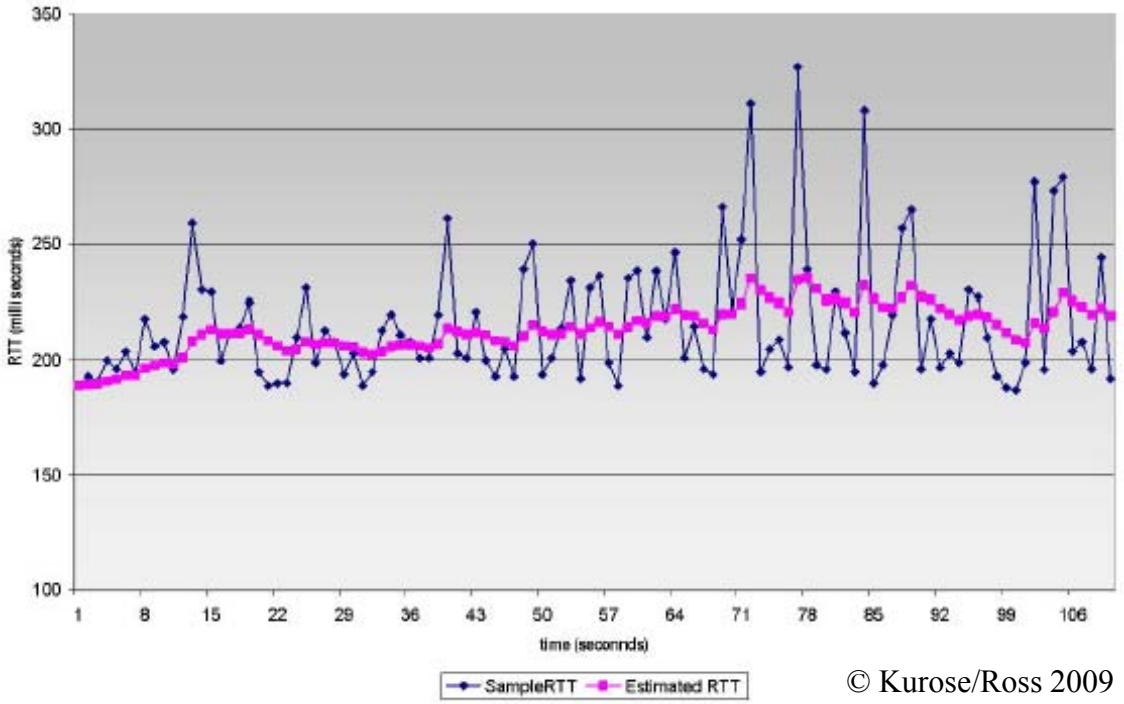
Wie kann RTT geschätzt werden?

- SampleRTT: Zeitspanne vom Absenden eines Segments bis zum Empfang des Acks
  (ohne Berücksichtigung von Wiederholungen)
- SampleRTT: ändert sich bei jedem Paket. Geht es besser?
  → (gewichteter) Durchschnitt über die letzten RTTs
  (moving average)

TCP: RTT und Timeout

Die Timeout-Zeitkonstante soll an Hand der Round Trip Time festgelegt werden:

t_{to}= RTT+ Sicherheits-abstand



© Kurose/Ross 2009

RTT Schätzung:

EstimatedRTT := (1 − α) * EstimatedRTT + α * SampleRTT

- Exponentiell gewichteter Durchschnitt: *Einfluss alter Messungen fällt exponentiell*
- typischer Wert: α = 0.125
- Genutztes RTT für TimeOut := EstimatedRTT * Faktor (Faktor ≈ 2-4)

TCP: Sendung von Acks

Ereignis beim Empfänger	Aktion des Empfängers
Ankunft eines Segments mit der erwarteten Seq.Nr., alle vorher empfangenen Daten wurden bereits bestätigt	Warte bis zu 500ms auf ein neues zu sendendes Segment. Falls kein Segment kommt sende ACK
Ankunft eines Segments mit der erwarteten Seq.Nr., vorher empfangenen Daten wurden noch nicht bestätigt	Bestätige alle noch nicht bestätigten Segmente sofort
Ankunft eines Segments mit einer zu großen Seq.Nr., Lücke entdeckt	Sende sofort ein *duplicate ACK*, Mit der nächsten erwarteten Seq.Nr.
Ankunft eines Segements, welches eine Lücke füllt	Falls das Segment zu Beginn der Lücke startet, sende sofort ein ACK

TCP: Fast retransmit

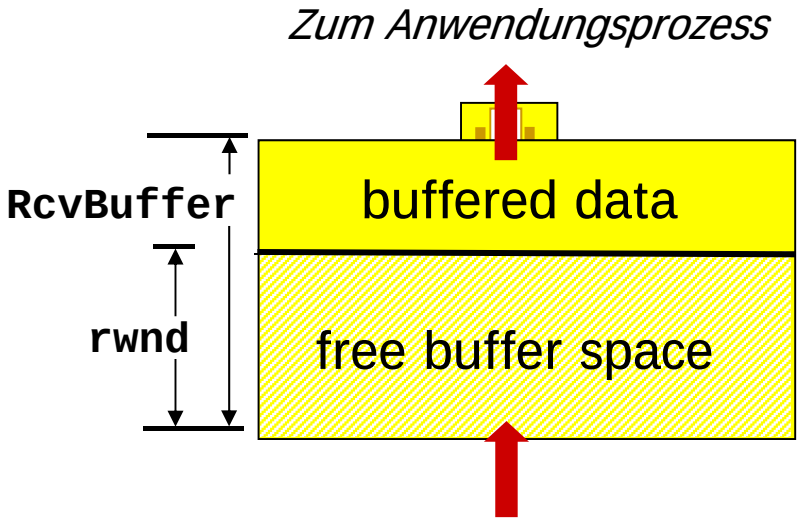
- Timeout ist oft relativ lang:
  - Lange Verzögerung bis ein Segment erneut gesendet wird
- Erkennung wahrscheinlich verloren gegangener Segmente durch duplizierte ACKs.
  - Sender sendet viele Segmente hintereinander
  - Verlust eines einzelnen Segments führt zu duplizierten ACKs für dieses Segment
- Wenn eine Sender 3 ACKs für das selbe Segment empfangen hat, sendet er das nachfolgende Segment noch einmal, auch wenn der Timer noch nicht abgelaufen ist (fast retransmit)

TCP: Flusskontrolle - „*Sendekredit bei Empfänger*“

Abstimmung der Senderate mit der Empfangs-(/Lese-)Rate

- Empfänger sendet mit jedem Segment den Wert von Window
- Der Sender richtet sich mit dem Senden nach LastByteSent – LastByteAcked ≤ Window

Daten zum Empfänger unterwegs



Senderseitige Pufferung

TCP: Verbindungsverwaltung

TCP Sender und Empfänger etablieren eine "Verbindung" bevor sie Daten austauschen

- Initialisierung von TCP-Variablen:
  - seq. #s
  - buffers, flow control info (e.g. **`RcvWindow`**)
- *client:* Initiator der Verbindung

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

- *server:* vom Client kontaktiert

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

3-Wege-Handshake:

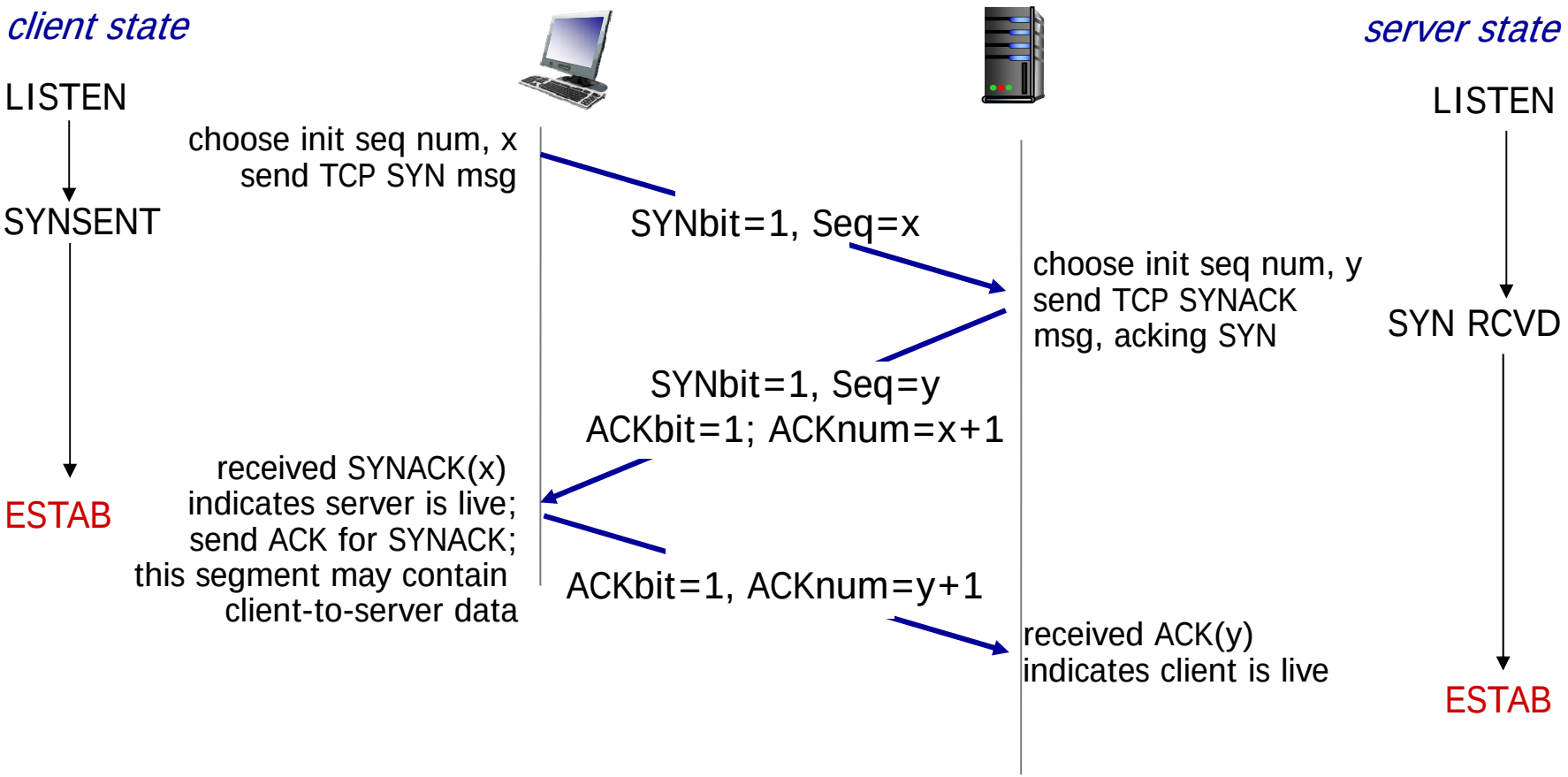
Schritt 1: client Host sendet TCP SYN Segment zum Server
  - spezifiziert initiale seq #
  - Ohne Daten

Schritt 2: server Host empfängt SYN, antwortet mit SYNACK Segment
  - Server allokiert Pufferplatz
  - Spezifiziert seine initiale seq. #

Schritt 3: client empfängt SYNACK, antwortet mit einem ACK Segment, das Daten enthalten kann

TCP: Verbindungsverwaltung

Öffne Verbindung über 3-Wege-Handshake



TCP: Verbindungsverwaltung

Schließen einer Verbindung

Socket.close() ;

Schritt 1 Client sendet TCP FIN-Kontrollsegment an Server

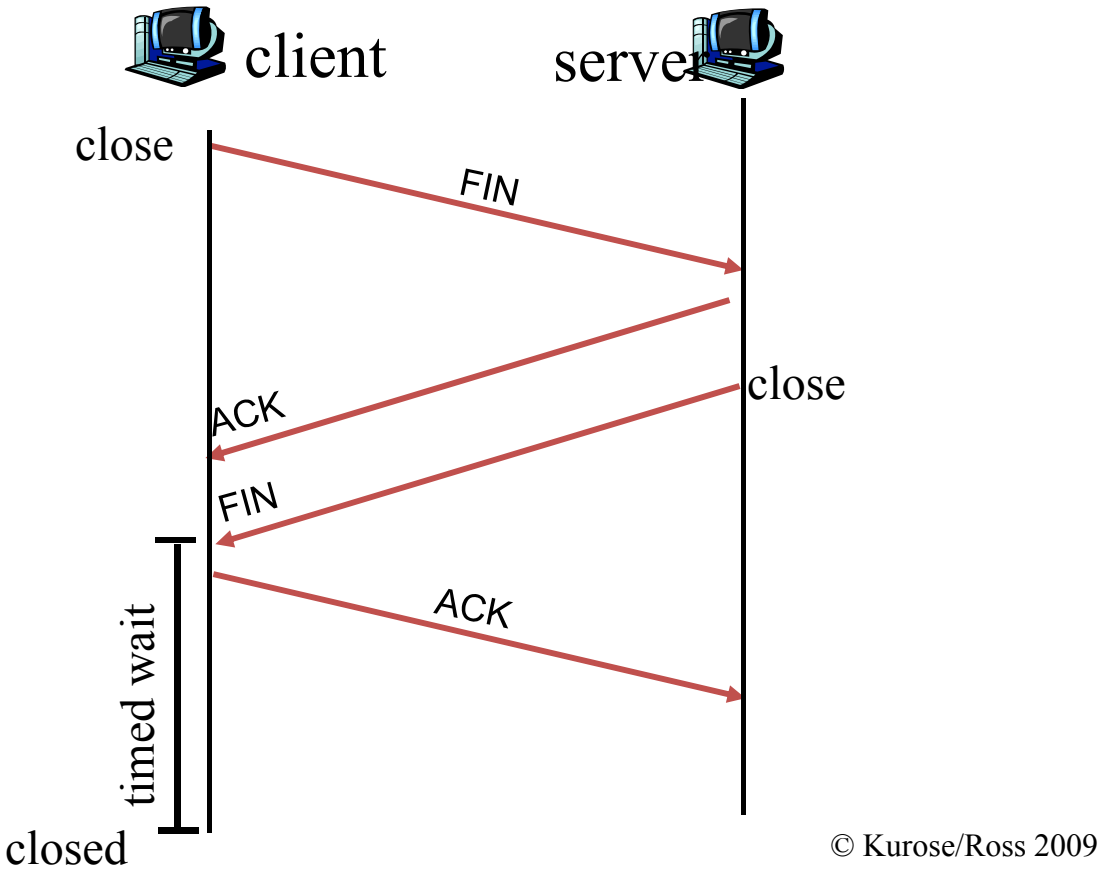
Schritt 2 Server empfängt FIN, sendet ein ACK, beendet die Verbindung und sendet ein FIN

Schritt 3 Client empfängt FIN, antwortet ACK. Geht in den Zustand „Time Wait“

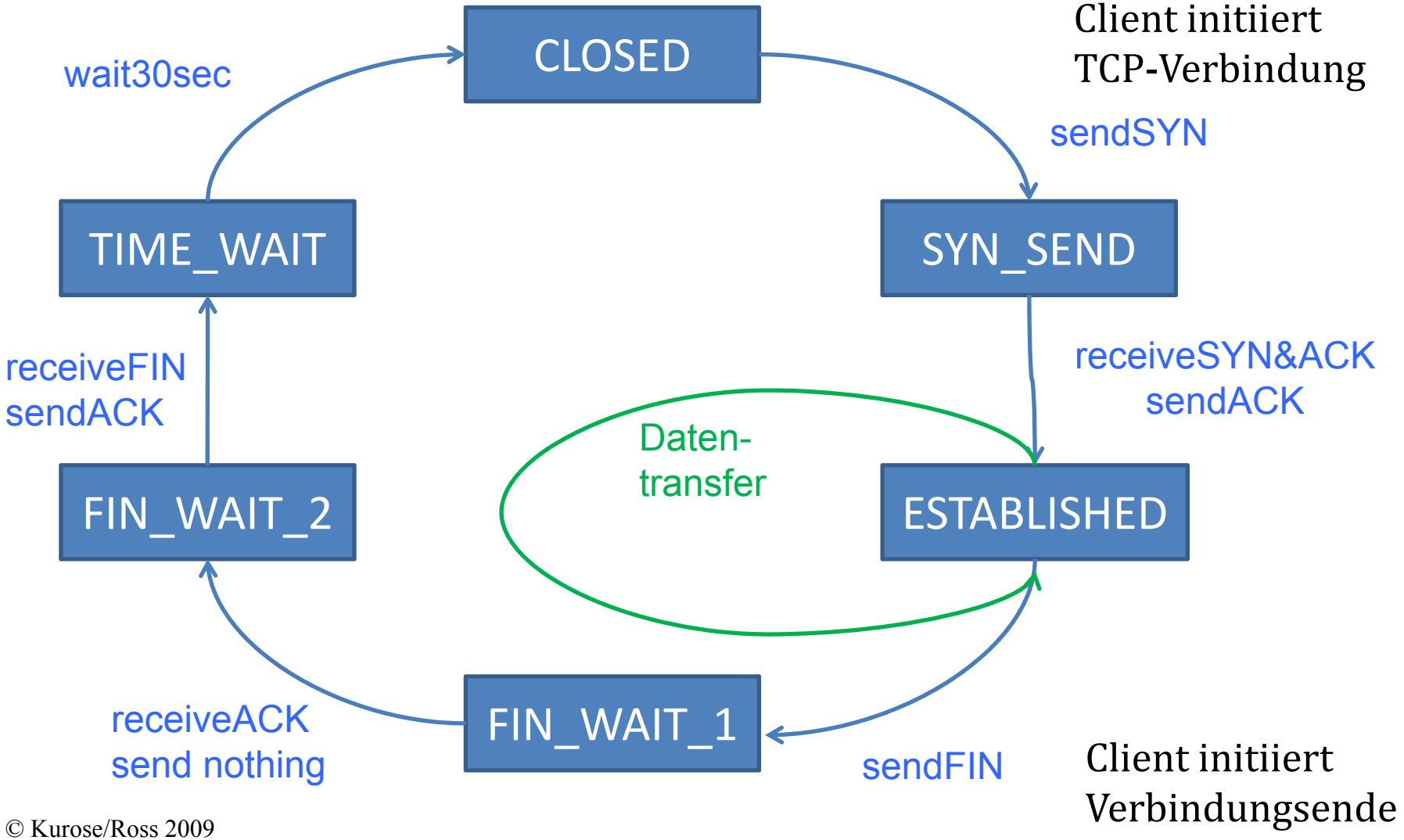
Schritt 4 Server empfängt ACK. Verbindung beendet

Mit kleinen Änderungen können simultane ACKs verarbeitet werden

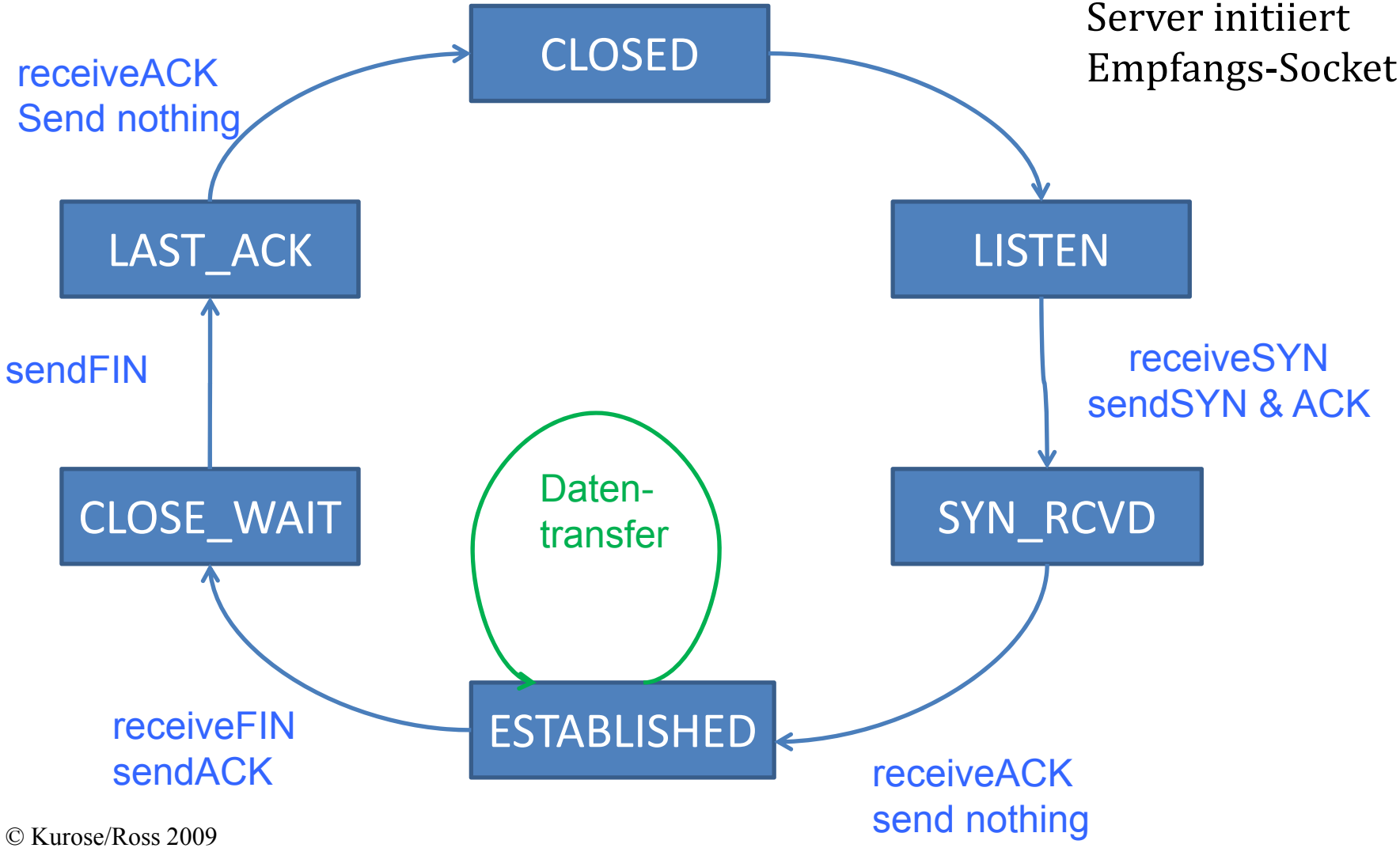
TCP: Verbindungsverwaltung



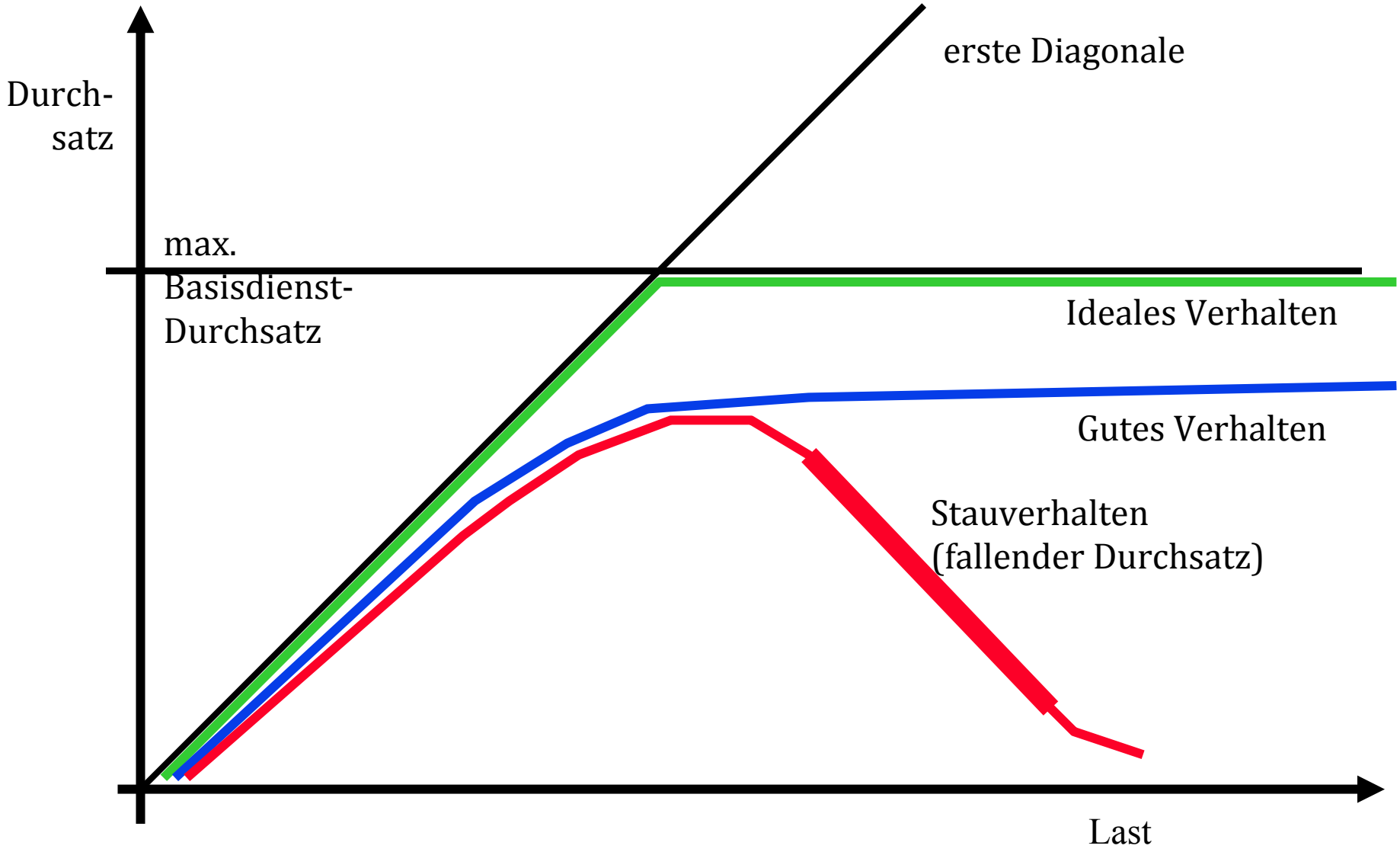
TCP: Client - Ablauf



TCP: Server - Ablauf



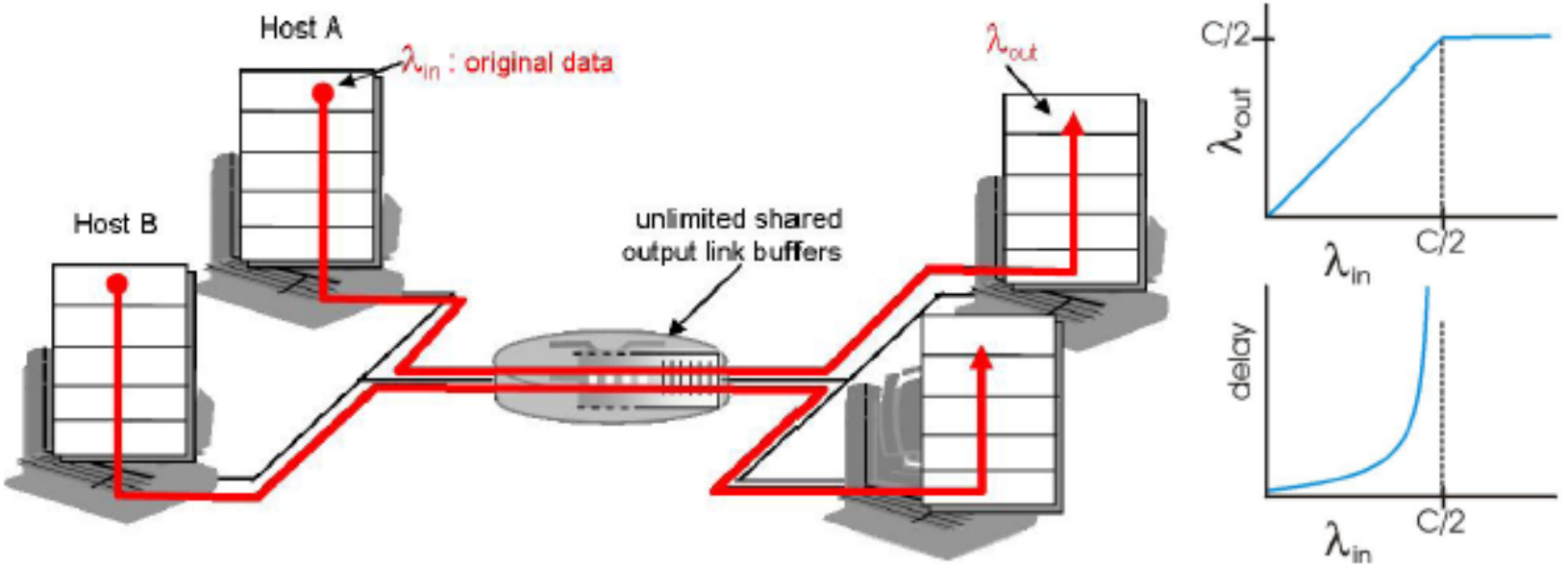
Überlastkontrolle (Staukontrolle) allgemein



Überlastkontrolle versus Flusskontrolle

- Überlastkontrolle und Flusskontrolle sind zwei **verschiedene** Dinge !
- Flusskontrolle regelt die Senderate
- Überlastkontrolle verhindert Stauverhalten
- Überlastkontrolle verwendet Flusskontrolle:
  - Messen der momentanen Belastung
  - Berechnen des möglichen Flusses
  - Entsprechendes Drosseln des Flusses
  - Iteration ...

Staus im Internet

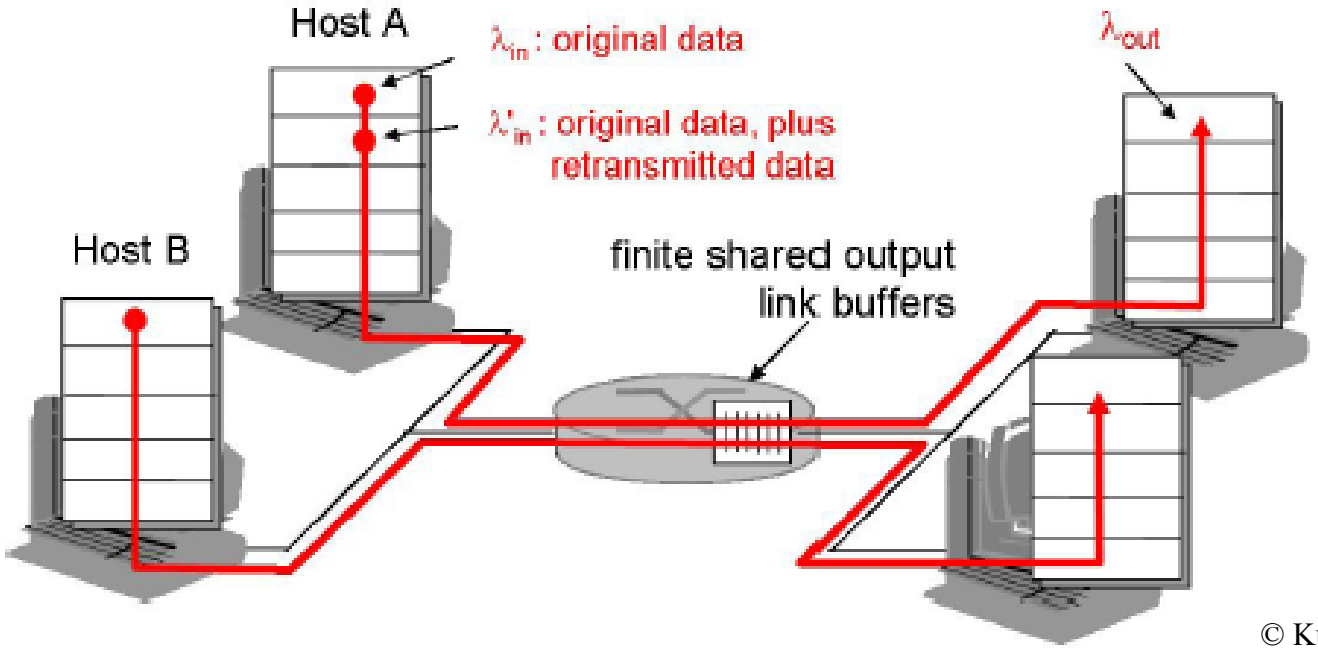


© Kurose/Ross 2009

Ursachen für Überlast (Szenario 1 unbeschränkter Puffer)

Größere Verzögerung im Router während der Übermittlung bei wachsender Übertragungsrate und wachsendem Hintergrundverkehr!!

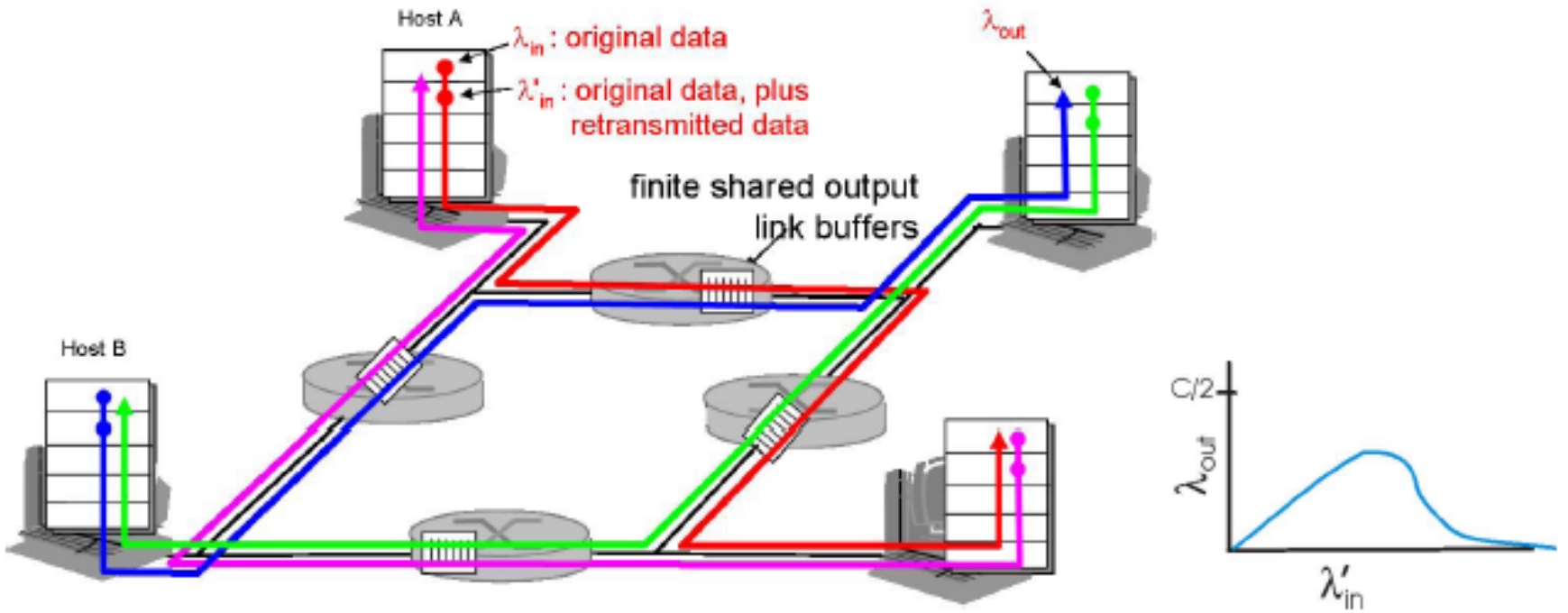
Staus im Internet



Ursachen für Überlast (Szenario 2 beschränkter Puffer)

Segmente gehen durch Pufferüberläufe verloren, wiederholte Übertragungen erzeugen zusätzliche Last und damit zusätzliche Verkehr!!

Staus im Internet



© Kurose/Ross 2009

Ursachen für Überlast (Szenario 2 beschränkter Puffer)

Gemeinsam genutzte Routen führen zu Paketverlusten bei allen Verbindungen und damit auch zu Wiederholungen auf allen Verbindungen!!

Überlastkontrolle - Ansätze

Ende-zu-Ende Überlastkontrolle

- Keine explizite Unterstützung durch die Vermittlungsschicht
- Überlastung wird durch Paketverlust und –verzögerung festgestellt und kann die Eingangsrate drosseln (d.h. Fenstergröße verkleinern)
- TCP muss diesen Ansatz verfolgen

Überlastkontrolle im Netz

- Komponenten der Vermittlungsschicht (Router) geben dem Sender explizit Feedback über Überlastzustände
- Z.B. DECnet, TCP/IP ECN, ATM ABR
- Sender bekommt eine explizite Senderate zugeteilt

Überlastkontrolle im Netz (ATM ABR)

ABR: available bit rate:

- Elastischer Dienst
- Wenn der Pfad vom Sender zum Empfänger nicht ausgelastet ist:
  - Soll der Sender der verfügbare Bandbreite nutzen
- Wenn der Pfad vom Sender zum Empfänger überlastet ist:
  - Darf der Sender nur eine minimale garantierte Bandbreite nutzen

Realisierung durch

RM (resource management) Zellen:

- Sender sendet RM Zellen zwischen den Datenzellen
- RM Zellen enthalten Informationen über den Netzzustand und werden von Switches gesetzt ("*network-assisted*")
  - NI bit: Rate nicht erhöhen (leichte Überlast)
  - CI bit: Überlast
- RM werden vom Empfänger zum Sender zurück geschickt

Überlastkontrolle in TCP

Steuerung der Übertragungsrate durch Anpassung der Sendefenstergröße

w = Anzahl übertragener unbestätigter Segmente
= LastByteSent – LastByteAcked

Es muss gelten

w ≤ aktuelle Sendefenstergröße

d.h., ein Sender darf nur dann ein neues Paket senden, wenn die Fenstergröße noch nicht erschöpft ist.

Die aktuelle **Sendefenstergröße** wird als Minimum aus 2 Werten bestimmt

- Das Receiver-Window **RcvWin** entspricht dem vom Empfänger aktuell zugeteilten Kredit.
- Das Congestion-Window **CongWin** wird vom Sender entsprechend der Staukontroll-Mechanismen bestimmt.

Sendefenstergröße = min (RcvWin, ConWin)

Überlastkontrolle in TCP

Der TCP-Algorithmus ist darauf angelegt, Überlast möglichst zu vermeiden bzw., wenn das nicht geht, möglichst schnell den Überlastzustand im Netz zu überwinden.

Algorithmus in 3 Schritten

Für jeden Knoten wird ein Grenzwert (Threshold) dynamisch verwaltet, so dass

1. Zu Anfang der **Grenzwert** auf einen passenden Wert (integer) festgelegt. Die Anfangsgröße von CongWin ist 1 MSS (Maximal Segment Size)
   Für die nacheinander gesendeten Segmente gilt nun:
   Bei jeder Bestätigung eines Segments, das vor dem Timeout ankommt, wird die Größe von CongWin um MSS erhöht, solange CongWin ≤ Threshold sonst Übergang zu 2. (bei Bestätigung) oder 3. (bei Timeout)
   (**Slow-Start-Phase**, exponentielles Wachstum von CongWin)

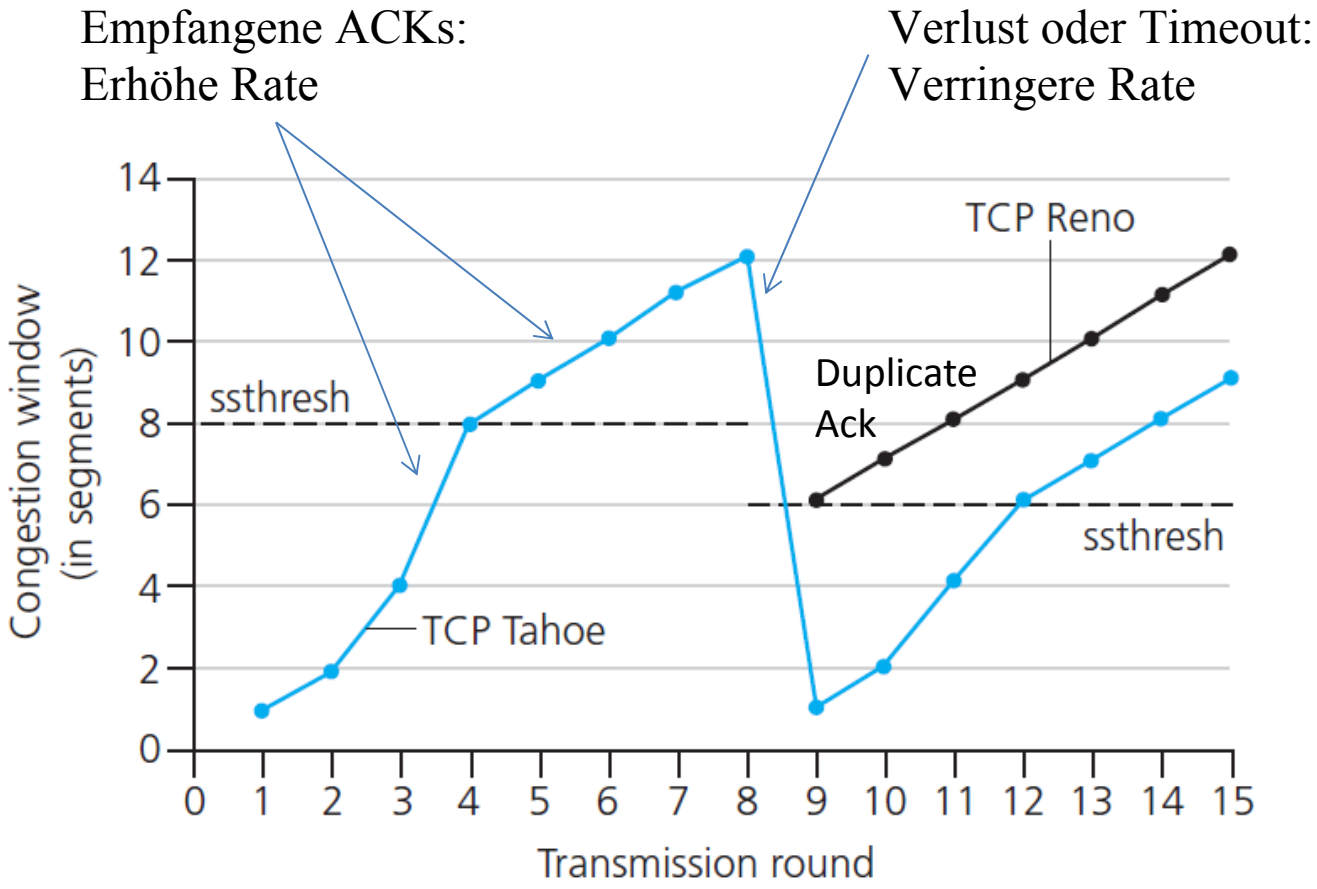
Überlastkontrolle in TCP

2. Sobald (bei rechtzeitiger Ankunft eines ACKs) die Größe von CongWin den Grenzwert übersteigt, wird die Größe von CongWin jeweils nur um MSS/CongWin erhöht
   (Überlastvermeidung, additives Wachstum von CongWin)
3. Sobald auf eine Sendung eines Segments ein Timeout erfolgt, wird der Grenzwert auf die Hälfte des aktuellen Wertes von CongWin gesetzt und die Größe von CongWin auf 1,
   anschließend wird wie bei Schritt 1 fortgefahren
   Bei drei doppelten Acks wird CongWin halbiert und wächst dann linear

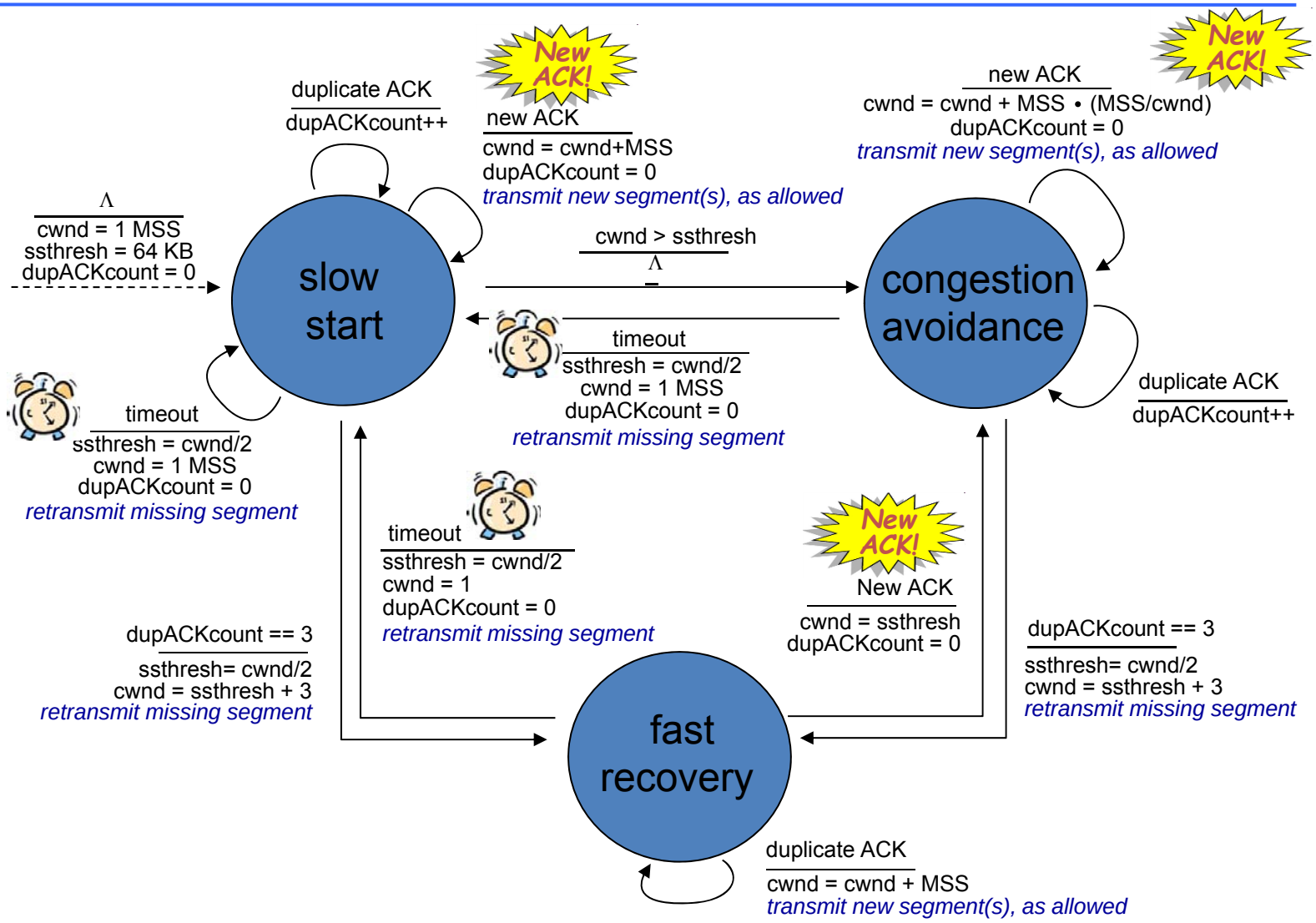
Es gibt unterschiedliche TCP-Varianten, deren Überlastkontrolle sich in Details unterscheiden.

TCP: Überlastkontrolle

Typisches Verhalten der TCP-Überlastkontrolle



TCP-Überlastkontrolle



© Kurose/Ross 2011

TCP-Durchsatz

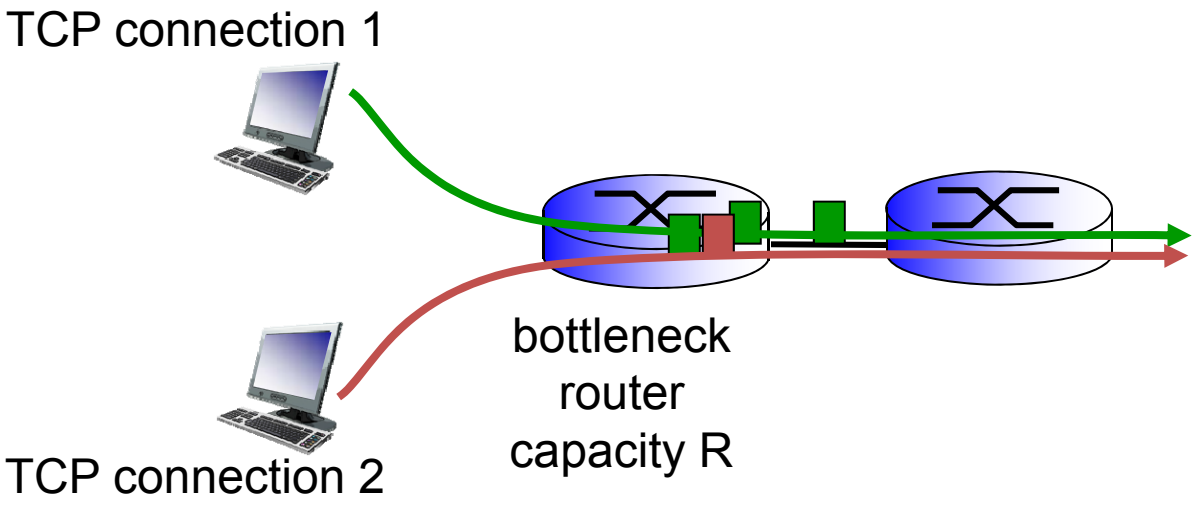
Welchen Durchsatz erreicht TCP?

- Bei Fenstergröße W: W / RTT
- Nach einem Paketverlust (dup. ACK): 0.5W / RTT
- Durch Sägezahnmuster ≈ 0.75W / RTT

TCP in zukünftigen Netzen:

- Parameter
  - Hoher Durchsatz
  - Segmentgröße bleibt
  - RTT bleibt
- Notwendige Fenstergröße muss sehr groß sein
- Verlustwahrscheinlichkeit darf nur sehr klein sein
- oder TCP muss erweitert werden

TCP-Fairness



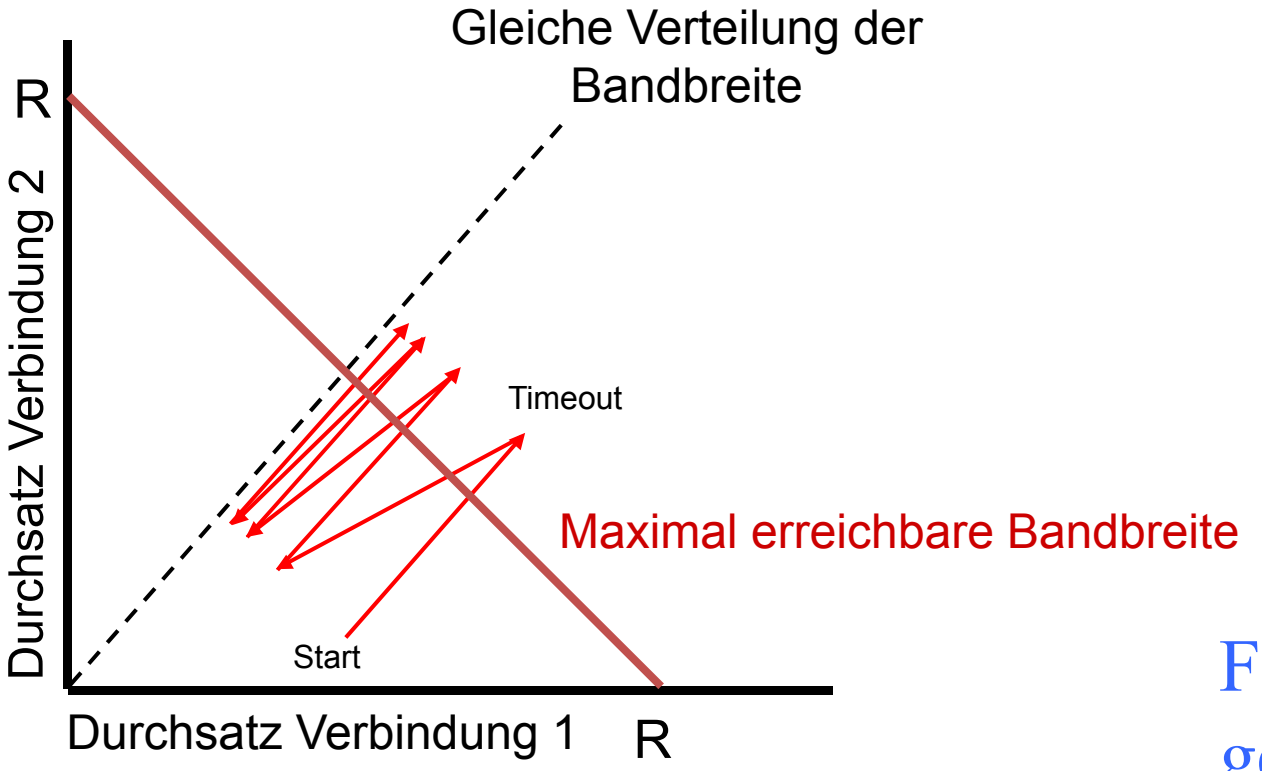
© Kurose/Ross 2011

Fairness-Ziel:
Wenn K TCP-Verbindungen sich eine Leitung mit Kapazität R teilen, die zum Flaschenhals wird, dann sollte jede Verbindung eine Kapazität von ungefähr R/K erhalten

TCP-Fairness

Zwei konkurrierende Verbindungen:

- Additive Vergrößerung der Fenstergröße führt zu linearem Wachstum
- Multiplikative Verkleinerung reduziert Fenstergröße proportional



Fairness ist gegeben!

Fairness: UDP, TCP und vielfache Verbindungen

Fairness in UDP

- Multimedia-Anwendungen nutzen oft UDP
  (da sie Ratenreduzierung durch TCP vermeiden)
- UDP erlaubt es beliebig viele Pakete in das Netz zu pumpen
- Damit ist UDP nicht fair, auch nicht gegenüber TCP-Verbindungen im Netz

Fairness paralleler TCP-Verbindungen

- Anwendungen kann mehrere TCP-Verbindungen gleichzeitig eröffnen (üblich bei Browsern)
- Bsp.: Kapazität R wird von 9 Verbindungen geteilt
  Neue Anwendung erhält
  - R/10 bei 1 Verbindung
  - R/2 bei 9 Verbindungen
- Parallele TCP-Verbindungen sind nicht fair

Zusammenfassung

- Prinzipien der Dienste in der Transportschicht:
  - Multiplexing, Demultiplexing
  - Zuverlässige Datenübertragung
  - Flusskontrolle
  - Überlastkontrolle
- Realisierung im Internet
  - UDP
  - TCP

Nächste Schritte:

- Wir verlassen den Rand des Netzes (Anwendungen, Netzzugang)
- Gehen in das Netz