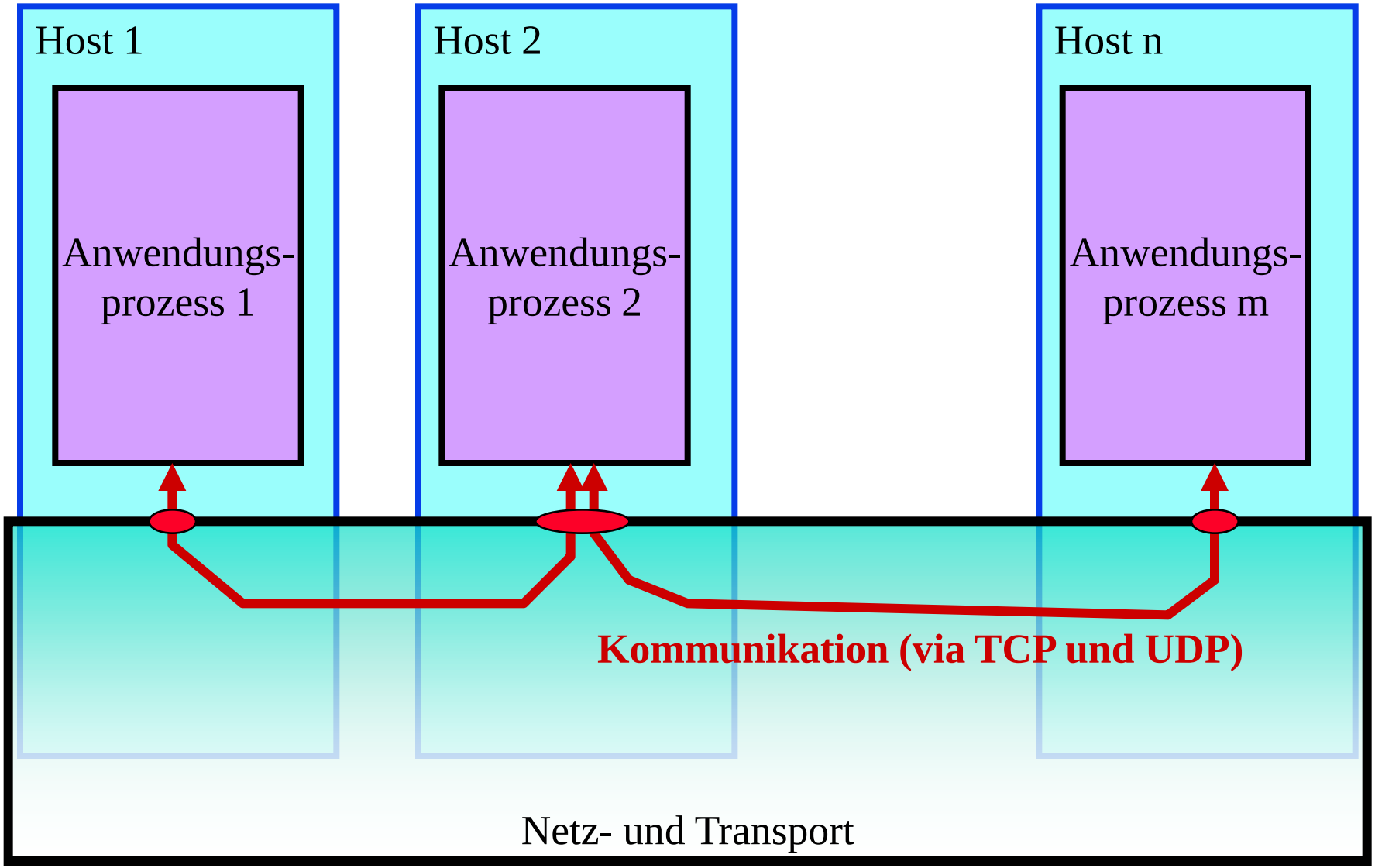
Rechnernetze und verteilte Systeme (BSRvS II)

Prof. Dr. Heiko Krumm
FB Informatik, LS IV, AG RvS
Universität Dortmund

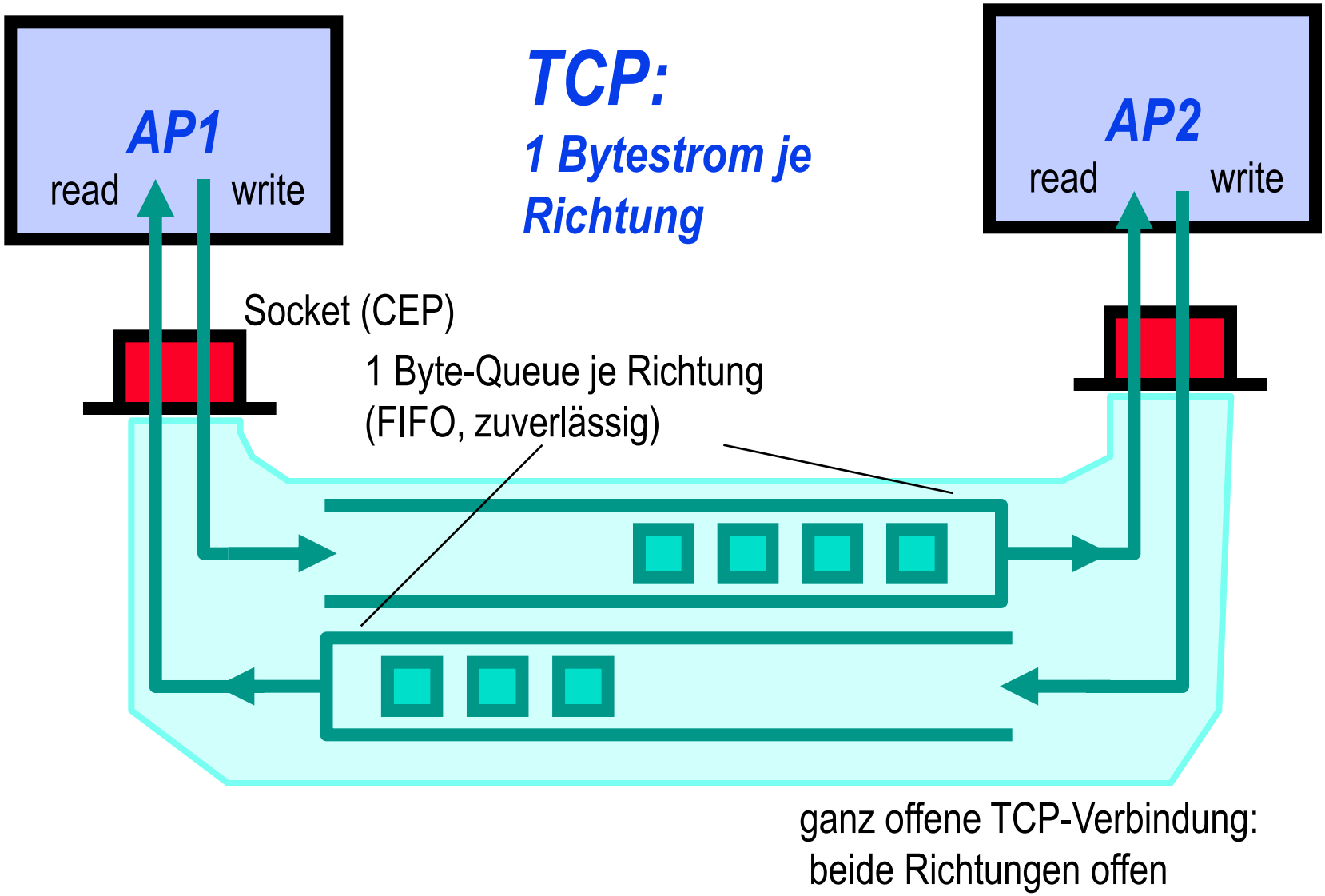
- Ende-zu-Ende-Transport
- Multiplexen und Demultiplexen
- UDP
- Protokollmechanismen
- TCP
- Lastkontrolle und Überlastabwehr

- **Computernetze und das Internet**
- **Anwendung**
- ***Transport***
- **Vermittlung**
- **Verbindung**
- **Multimedia**
- **Sicherheit**
- **Netzmanagement**
- **Middleware**
- **Verteilte Algorithmen**

Transportdienste: Anwendungsprozess-Kommunikation

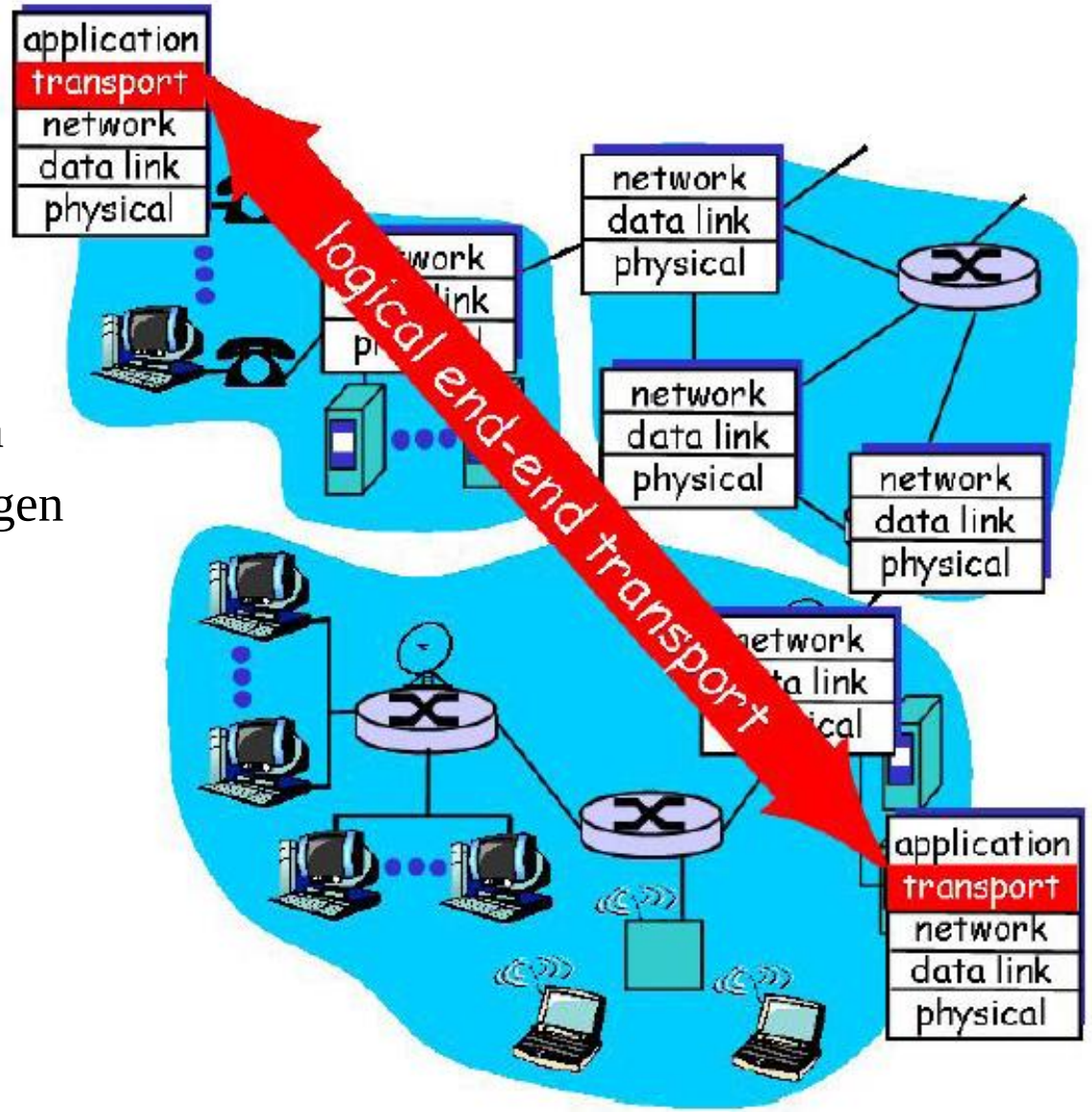


Transport: Anwendungsprozess-Kommunikation

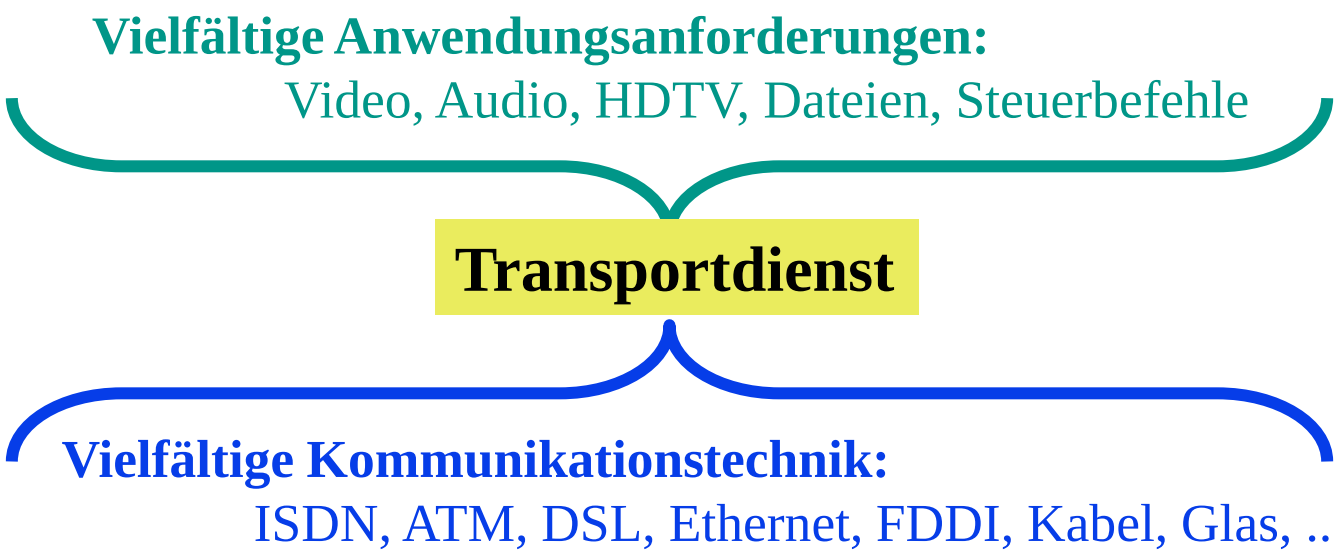


Transport: Anwendungsprozess-Kommunikation

- Funktionalität „**Ende-zu-Ende**“ Anwendungsprozesse kommunizieren:
  - Adressierung
  - Anwendungsnachrichten
  - Anwendungsanforderungen

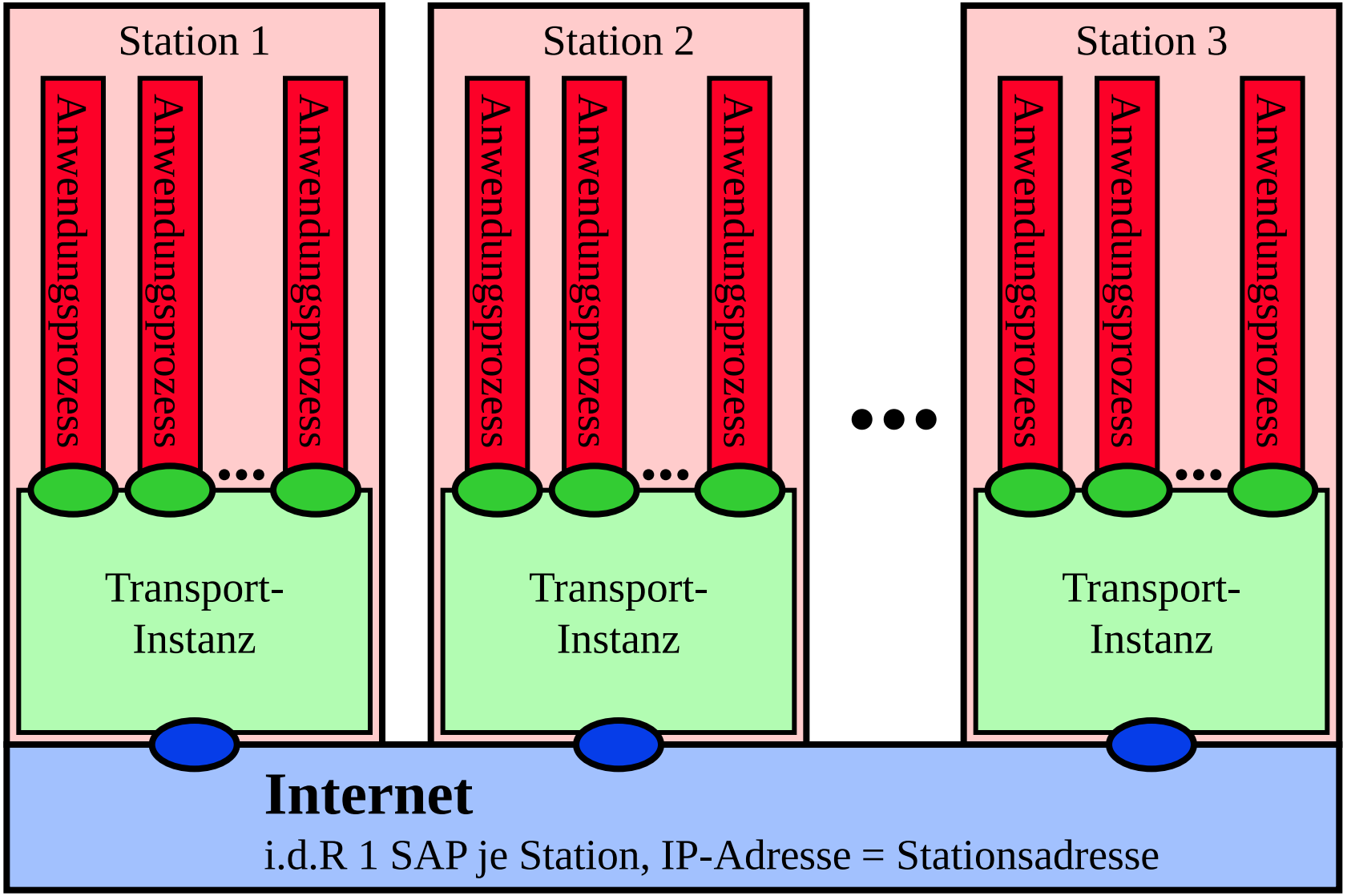


Transport: Anwendungsprozess-Kommunikation



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

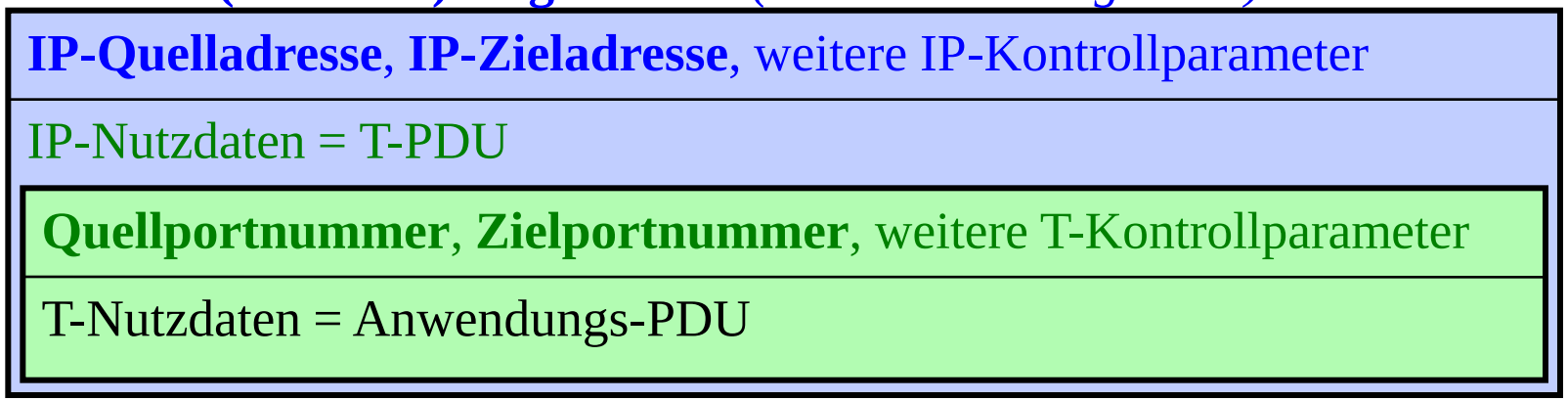
Transport: Stations- und Prozessadressen / Multiplexen



Transportschicht

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

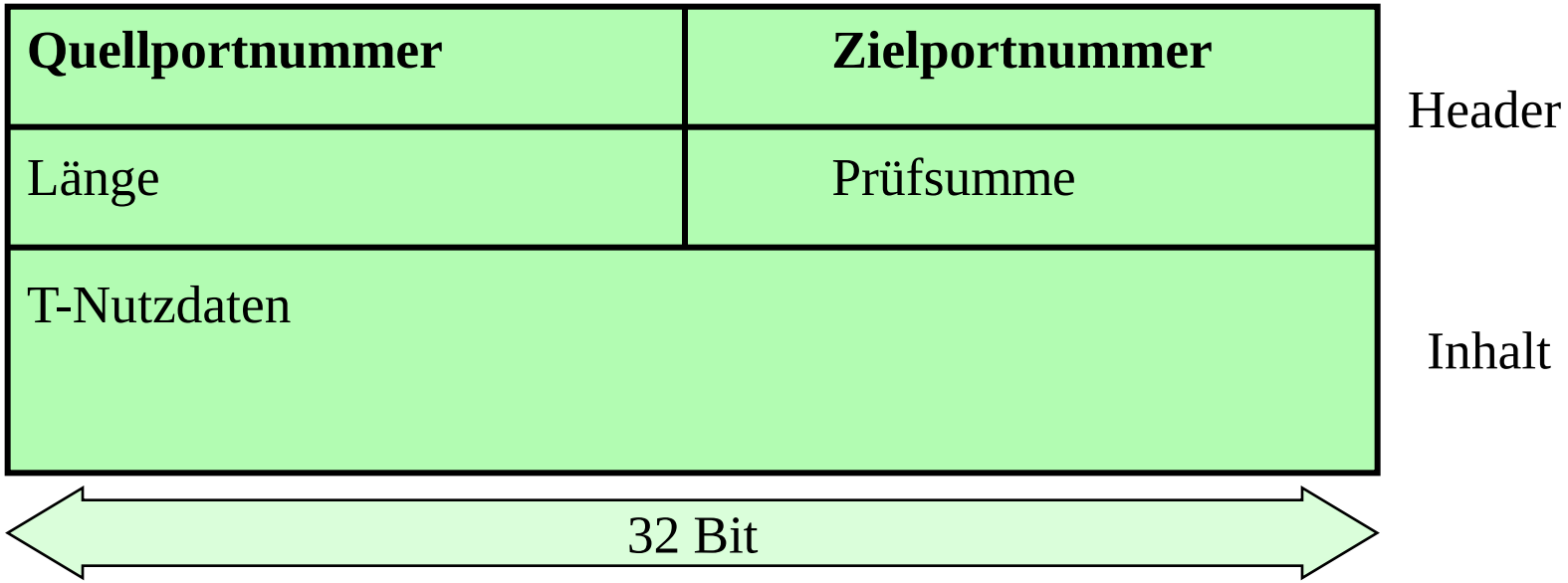
IP-PDU (IP-Paket) trägt T-PDU *(oder T-PDU-Segmente)*



UDP – User Datagram Protocol

- Verbindungslos, Relativ 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 Checksummen 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 Checksum

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

Einerkomplement-Addition „+“:

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

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 Checksummenberechnung für UDP

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

Neuberechnung der IP-Datagramm-Checksumme an jedem Hop?

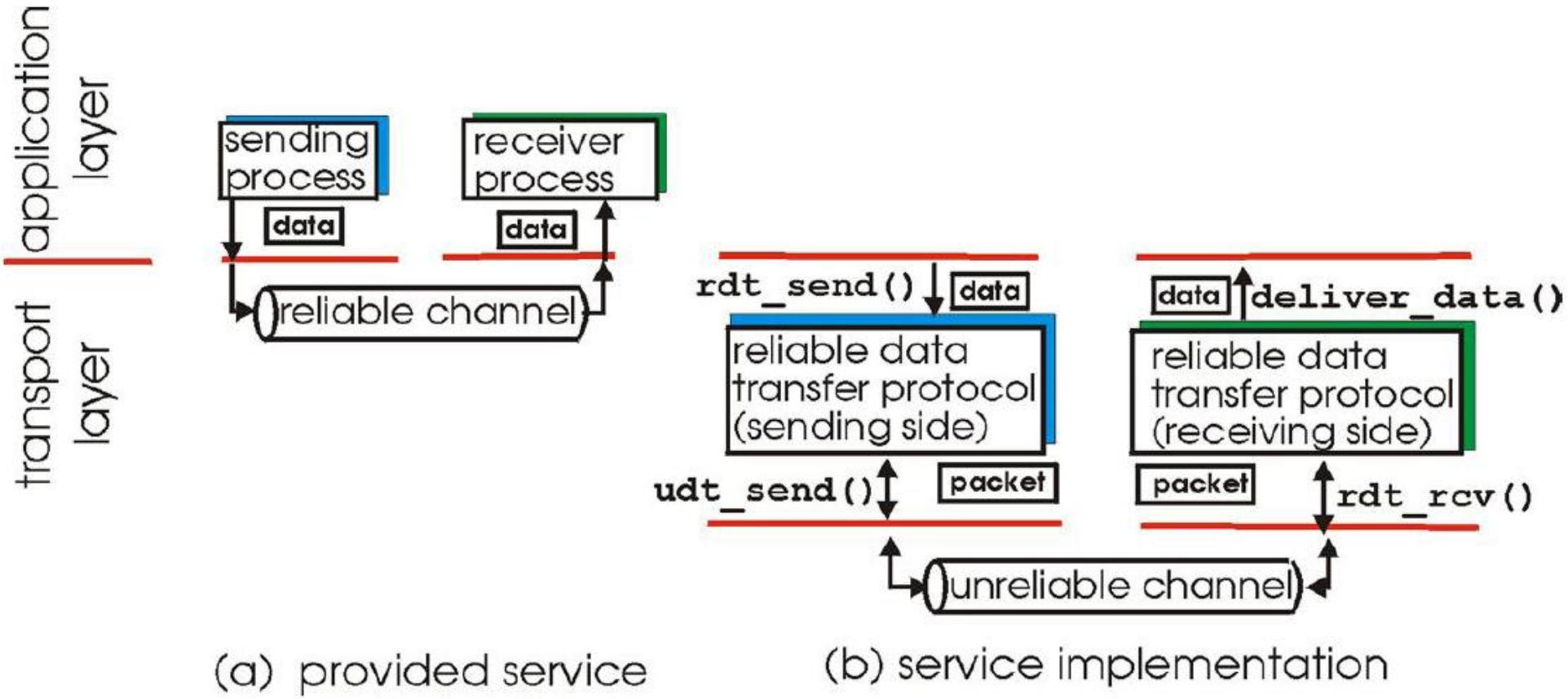
An jedem Hop wird die Time-to-Live eines IP-Datagramms dekrementiert. Damit verändert sich auch die Checksumme. Muss diese an jedem Hop neu berechnet werden?
Nein. Wegen der Assoziativität und der Invertierung genügt es, den Wert, um den die TTL verringert wurde, zu addieren:

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

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

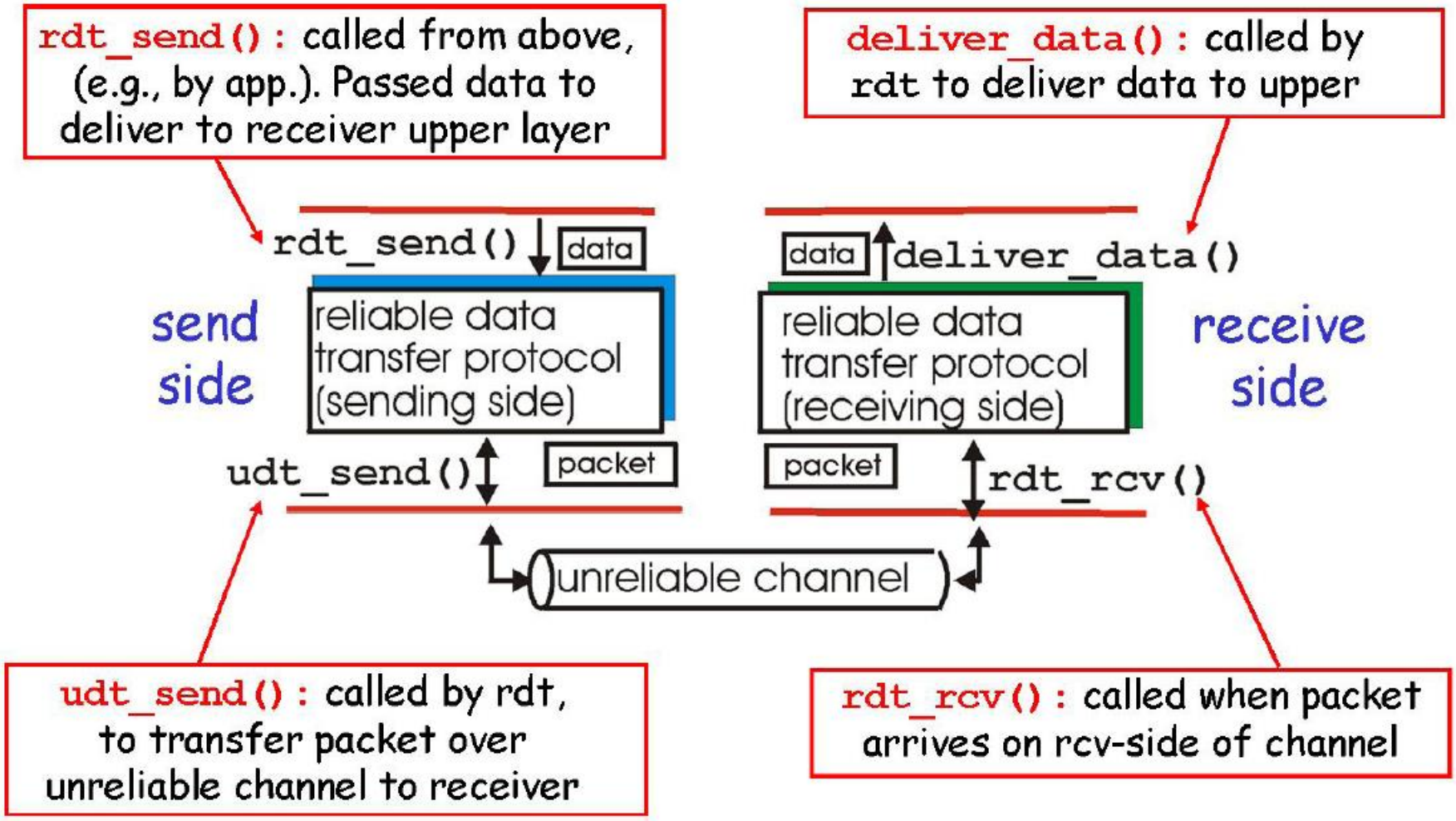
Auf dem Weg zu TCP

Zuverlässige Kommunikation über unzuverlässigen Basisdienst



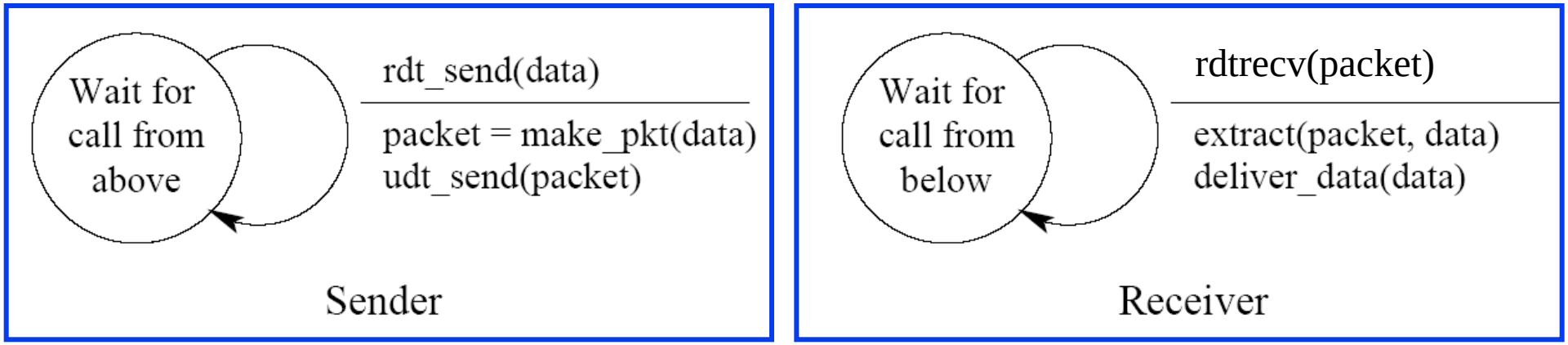
Schrittweise Entwicklung

Protokollserie: rdt0, rdt1, ..: Gemeinsamer Rahmen



rdt1.0: Zuverlässiger Basisdienst

Einfaches Weitergeben genügt



Wenn Basisdienst schon zuverlässig ist:

- Verfälschungsfrei
- Verlustfrei
- Phantomfrei
- Duplikatfrei
- Vertauschungsfrei

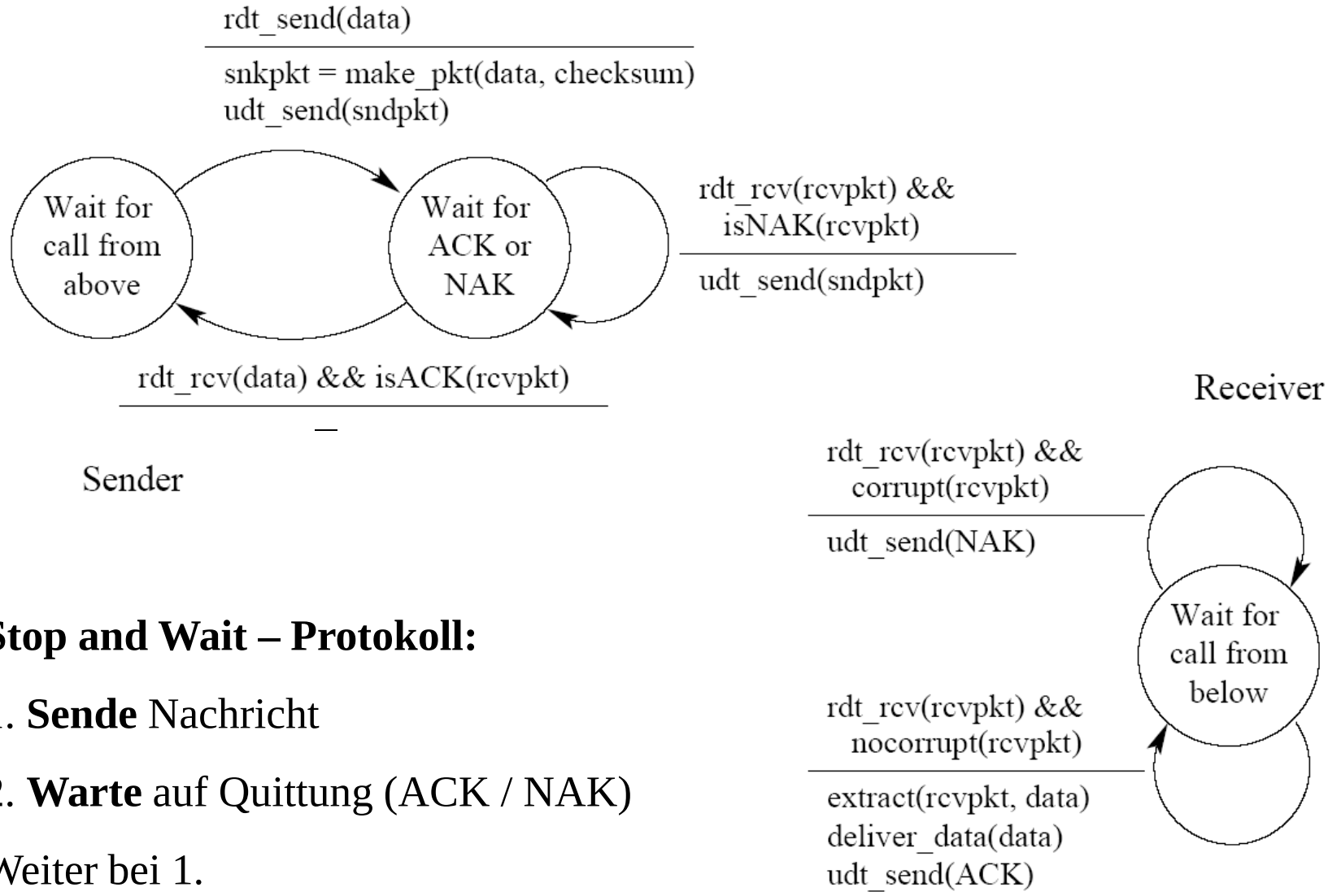
rdt2.0: Basisdienst nicht verfälschungsfrei (Bitfehler)

- Fehlerbehandlung generell
  1. ERKENNEN
  2. BEHEBEN

Verfälschungsbehandlung

- **A) ARQ (Automatic Repeat Request)** Verfahren
  - Fehlererkennung: Fehlererkennender Code
  - Fehlerbehebung: Wiederholung, eingeleitet durch Rückmelduungen
- **B) FEC (Forward Error Correction) Verfahren**
  - Fehlerkorrigierender Code
- Im folgenden: A)

rdt2.0: Basisdienst nicht verfälschungsfrei (Bitfehler)



Stop and Wait – Protokoll:

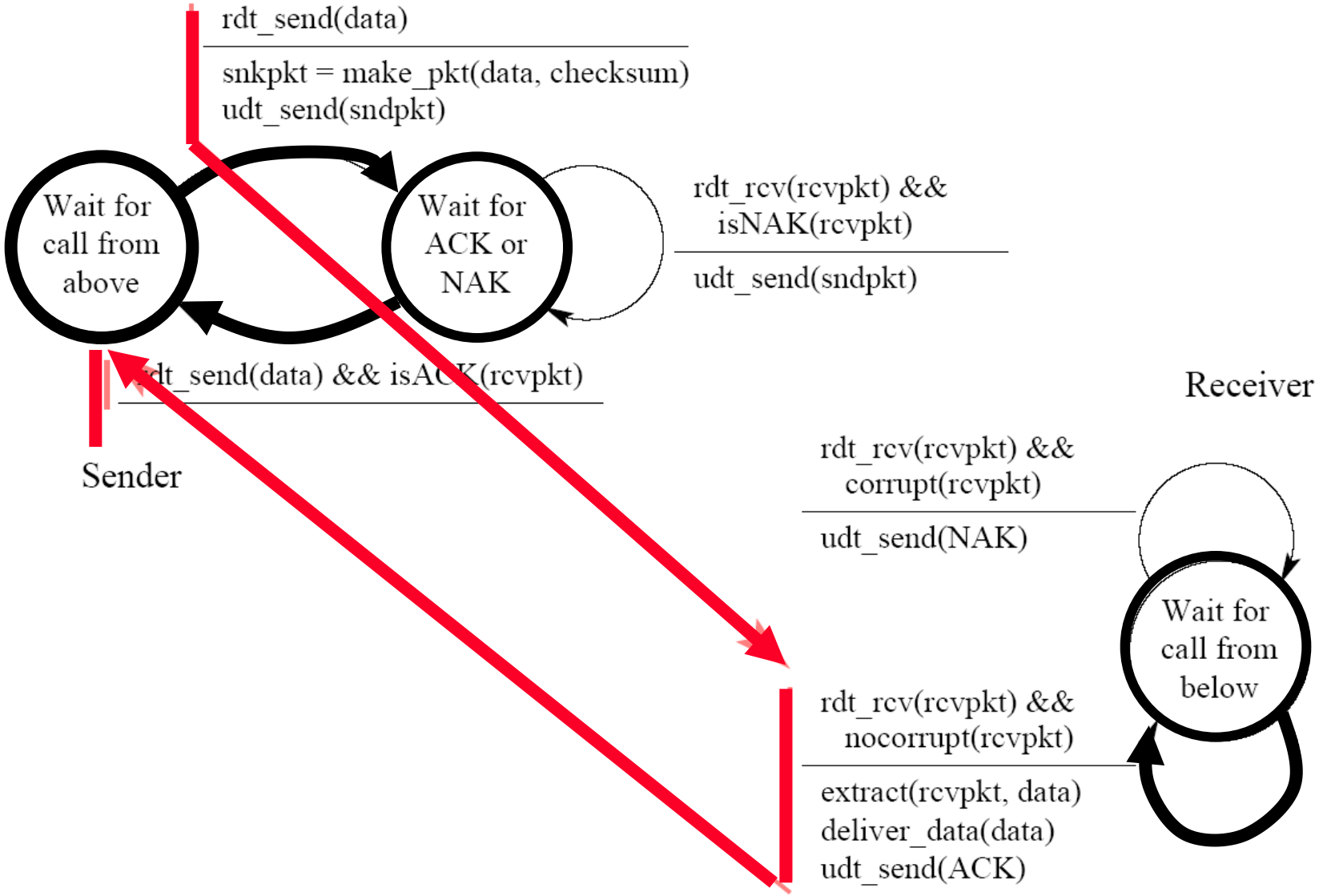
1. **Sende** Nachricht

2. **Warte** auf Quittung (ACK / NAK)

Weiter bei 1.

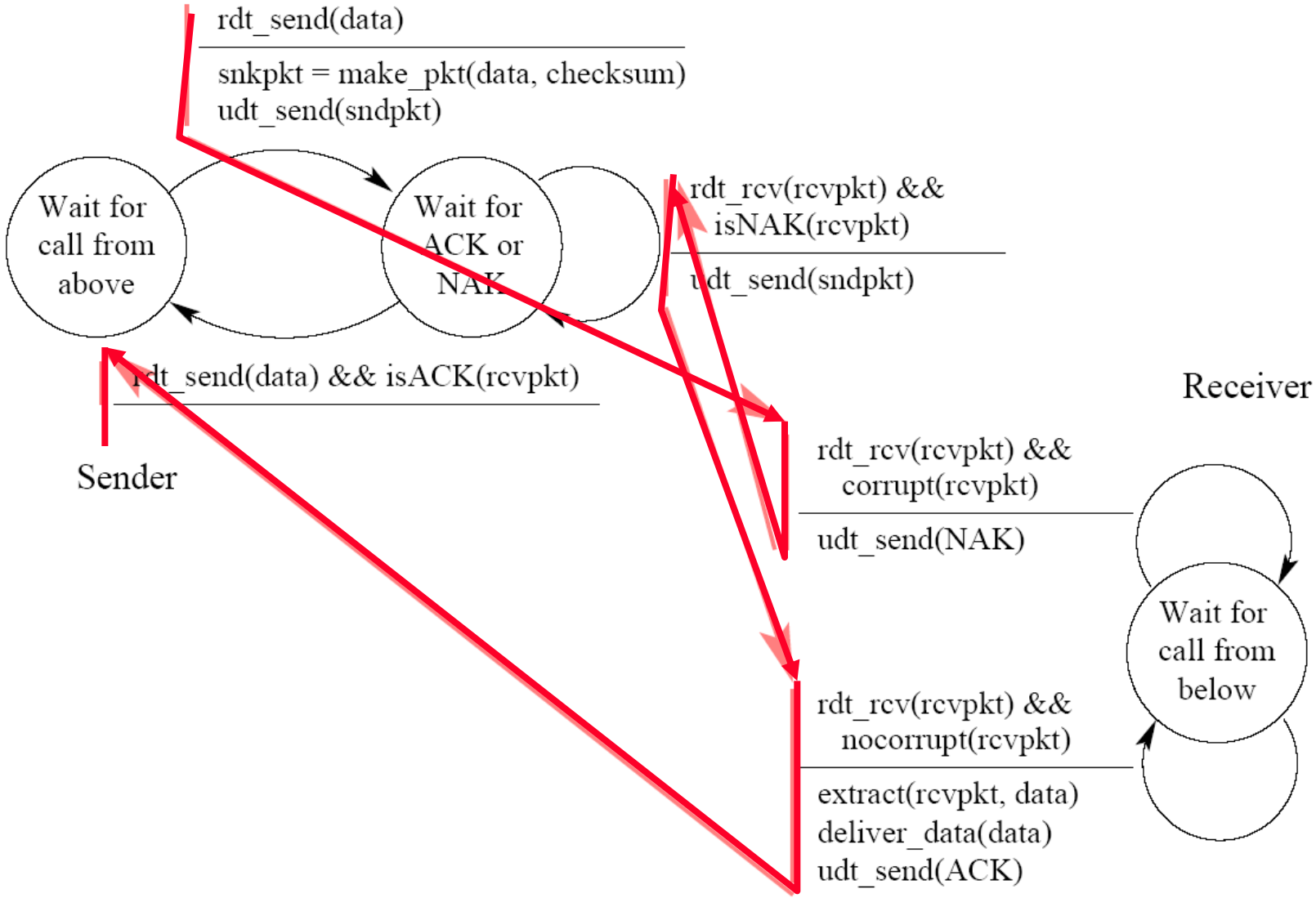
rdt2.0: Basisdienst nicht verfälschungsfrei (Bitfehler)

Szenario ohne Fehler



rdt2.0: Basisdienst nicht verfälschungsfrei (Bitfehler)

Szenario mit Fehlern



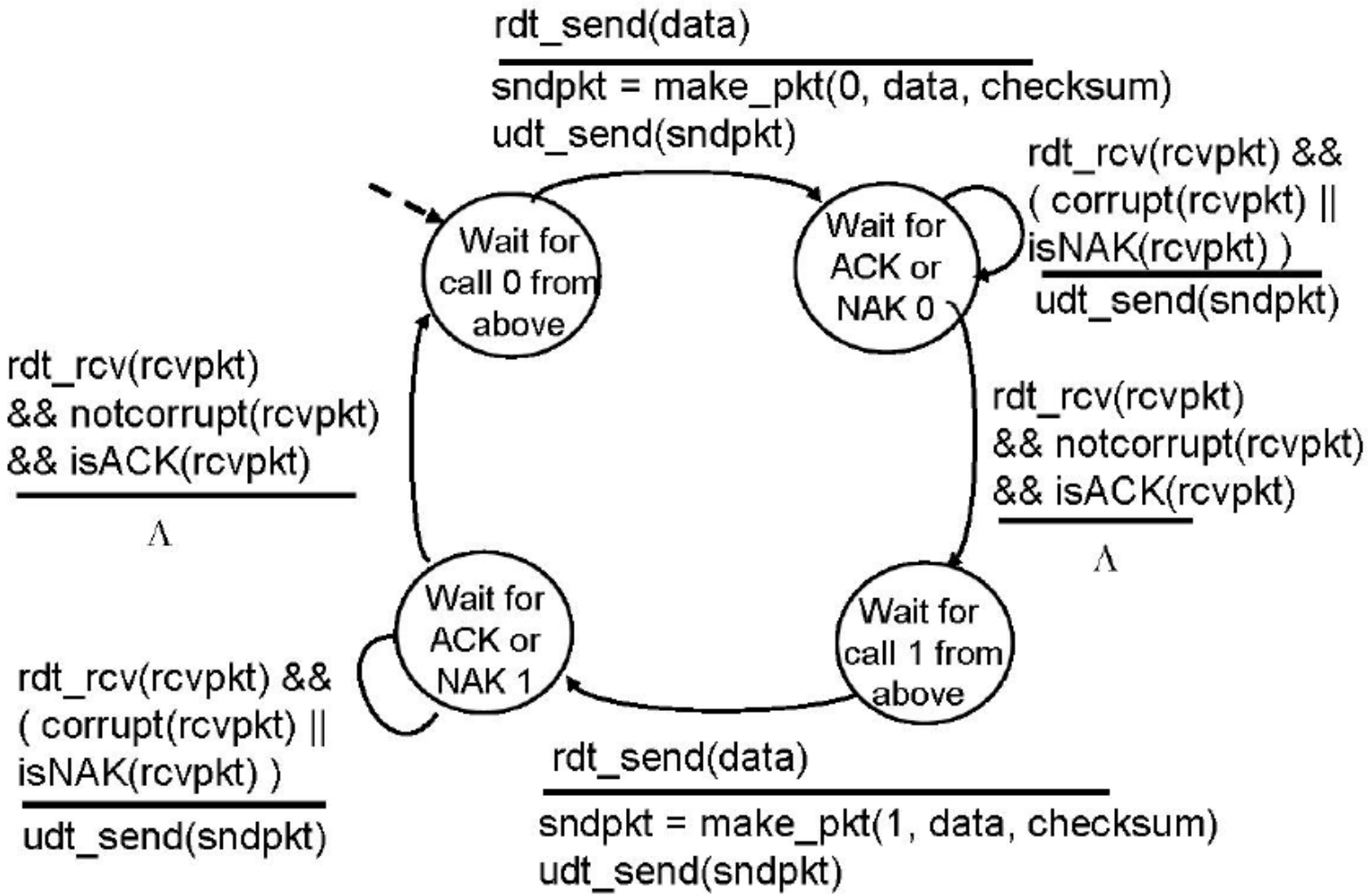
rdt2.0: Basisdienst nicht verfälschungsfrei (Bitfehler)

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)

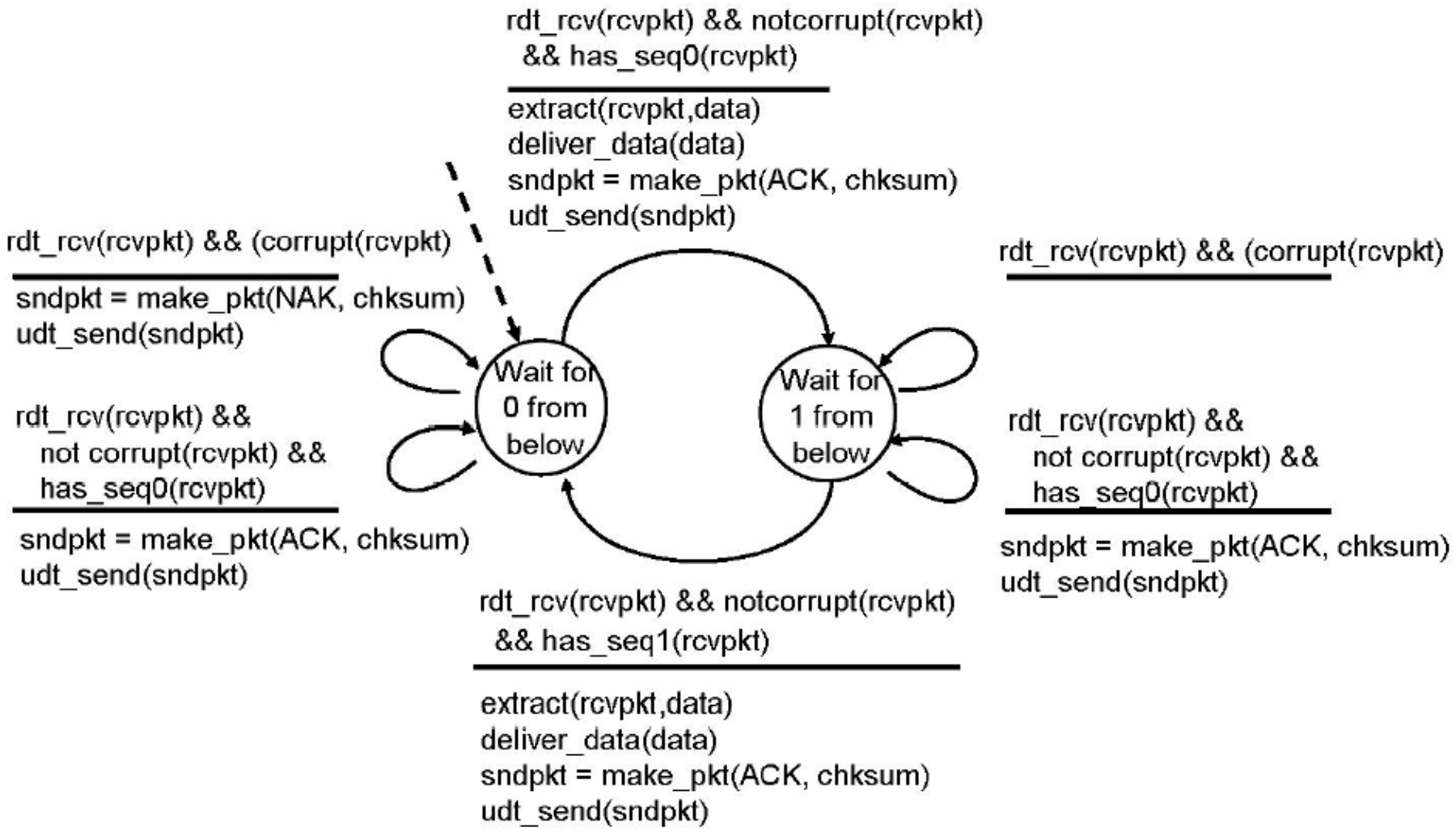
rdt2.1: Sequenzzahlen (0..1)

rdt 2.1 (Sender)



rdt2.1: Sequenzzahlen (0..1)

rdt 2.1 (Empfänger)



rdt2.1: Sequenzzahlen (0..1)

Bewertung von rdt2.1

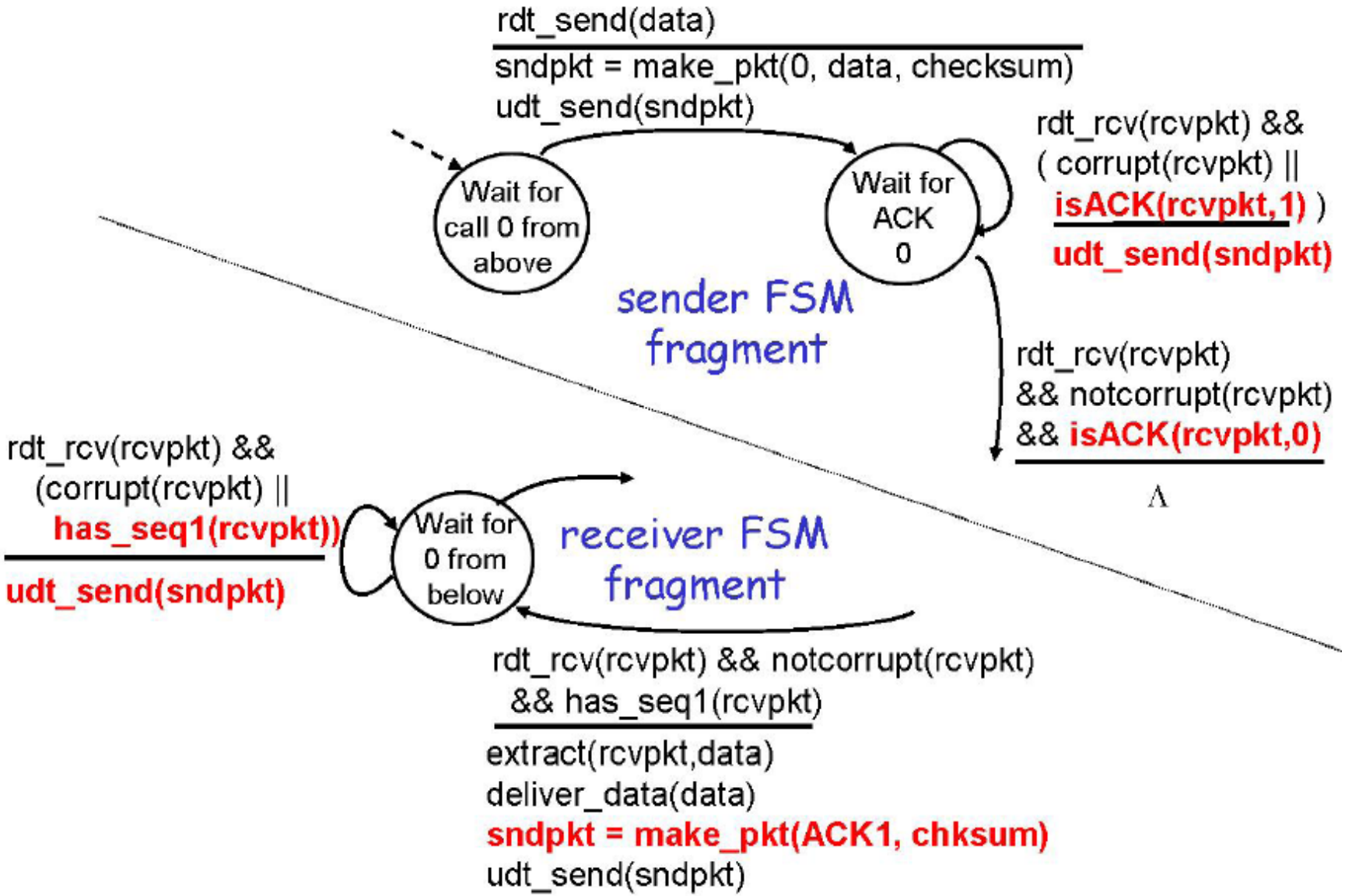
- Sender
  - Fügt Sequenznummer zu jedem Paket hinzu
  - Reichen zwei Nummer (0,1) aus? Warum!?
  - Muss jedes empfangene ACK/NAK auf Korrektheit überprüfen
  - Hat die doppelte Anzahl von Zuständen
- Empfänger
  - Muss Pakete auf doppelte Übertragung untersuchen
  - Weiß nicht, ob der Sender das ACK/NAK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

rdt2.2: Nur positive Quittungen

rdt 2.2 (Protokoll ohne NAKs)

- Funktionalität wie rdt 2.1, ohne aber 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?

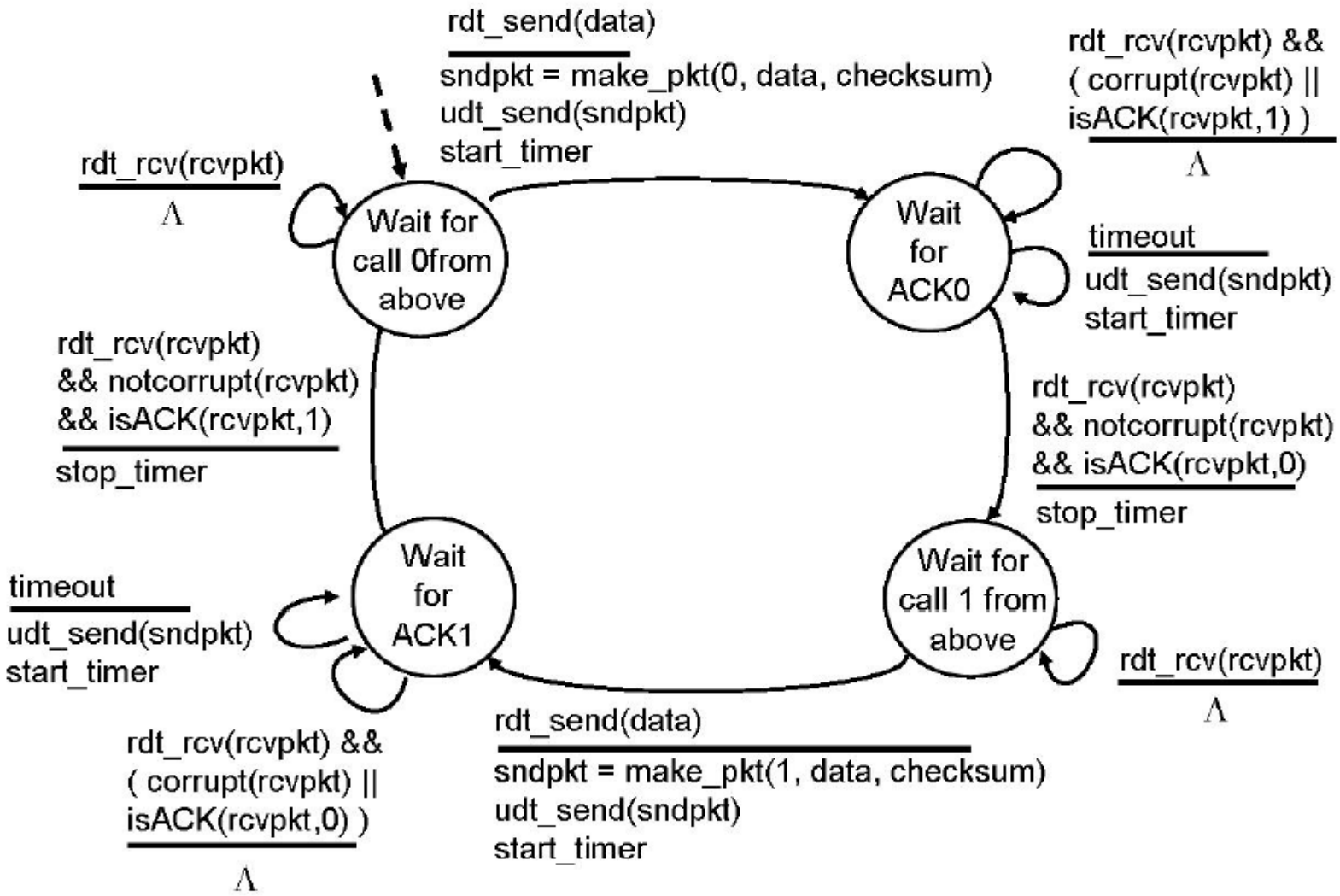
rdt3.0: Time-Out Mechanismus

- Sender wartet nur eine festgelegte Zeit auf ein ACK
  → ***Timer starten*** („Kurzzeitwecker“)
- Erneutes Senden, wenn bis ***Timeralarm*** kein ACK empfangen wurde
- Wenn ein Paket oder ACK nicht verloren waren, sondern nur verzögert, so wird das Paket doppelt gesendet
  → dies wird wegen der Sequenznummer erkannt
- Sequenznummer muss auch im ACK angegeben werden

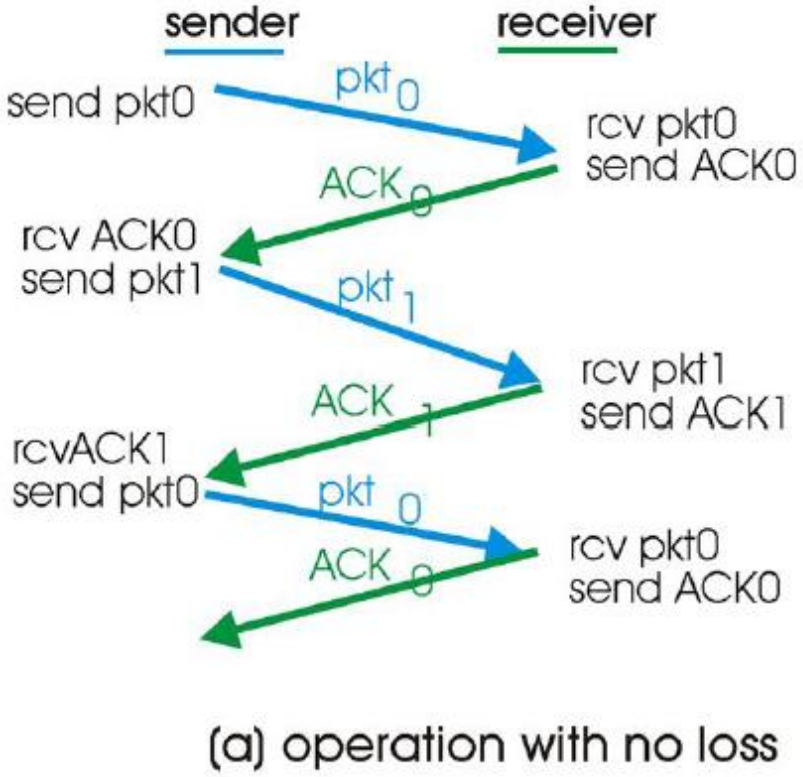
1. Beim Senden einer Nachricht: Kurzzeitwecker starten
2. Warten auf Quittung ODER Weckeralarm
3. Wecker stoppen
4. Weckeralarm wie negative Quittung behandeln: Retransmit

rdt3.0

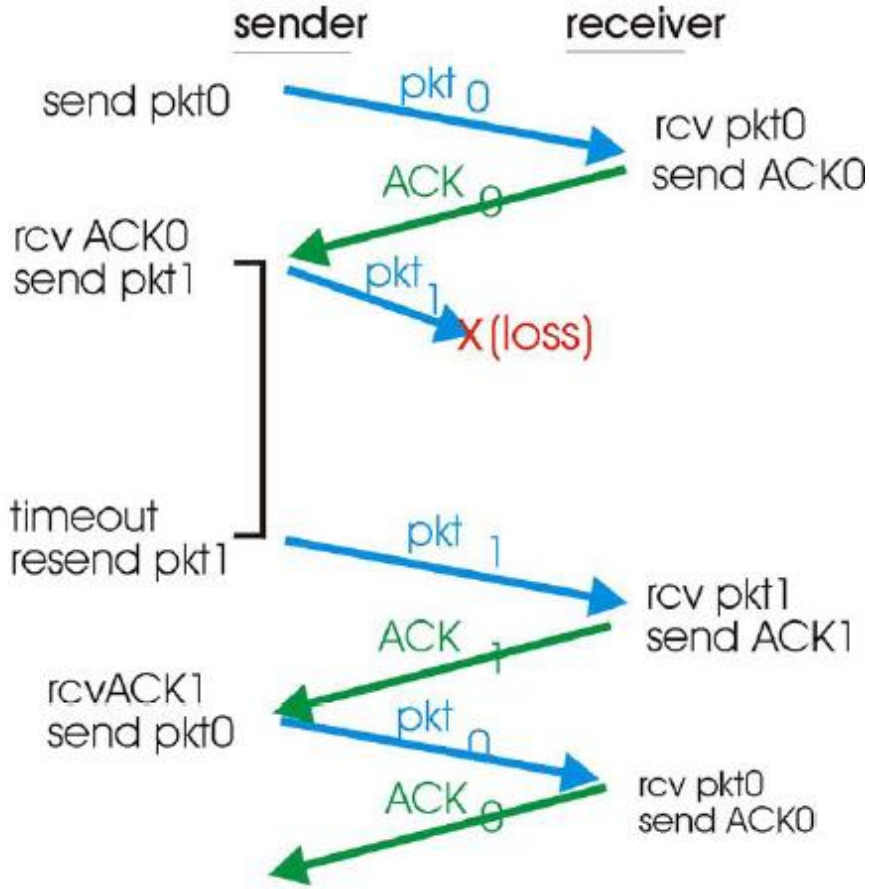
rdt 3.0 Sender



rdt3.0: Abläufe im Weg/Zeitdiagramm

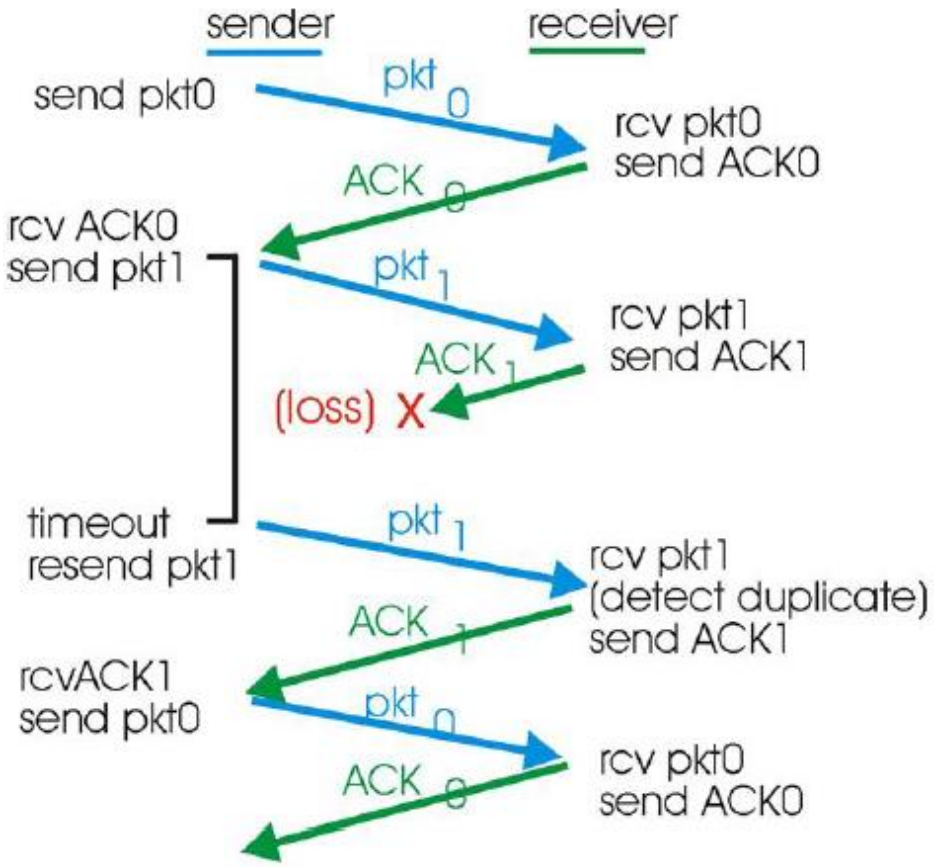


(a) operation with no loss

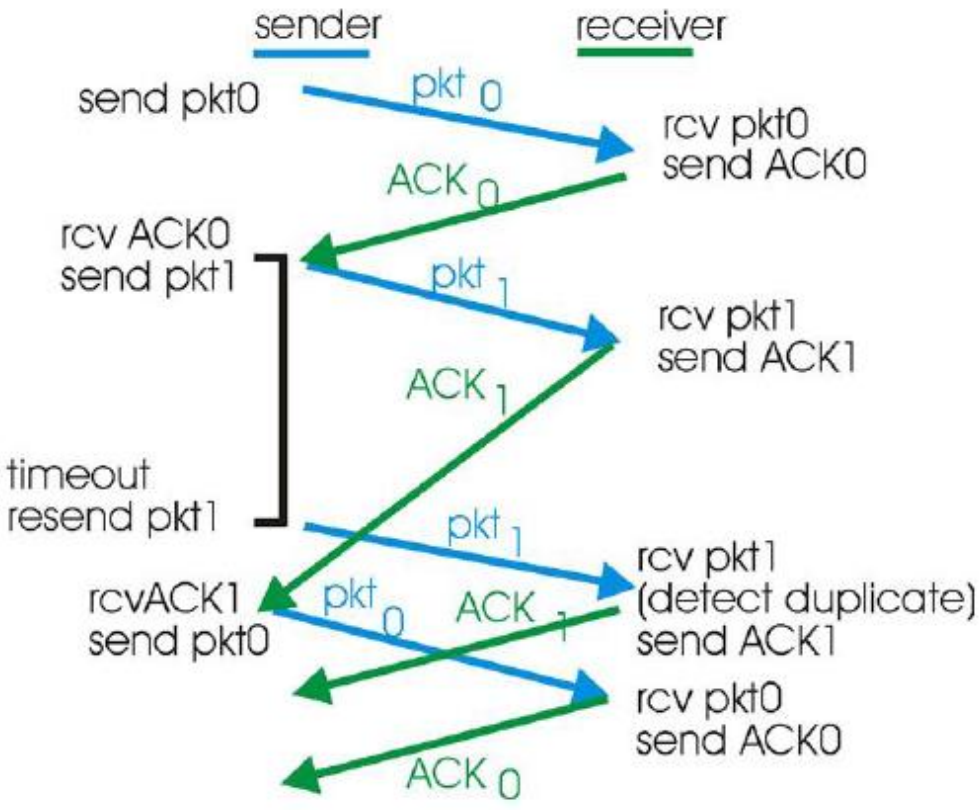


(b) lost packet

rdt3.0: Abläufe im Weg/Zeitdiagramm

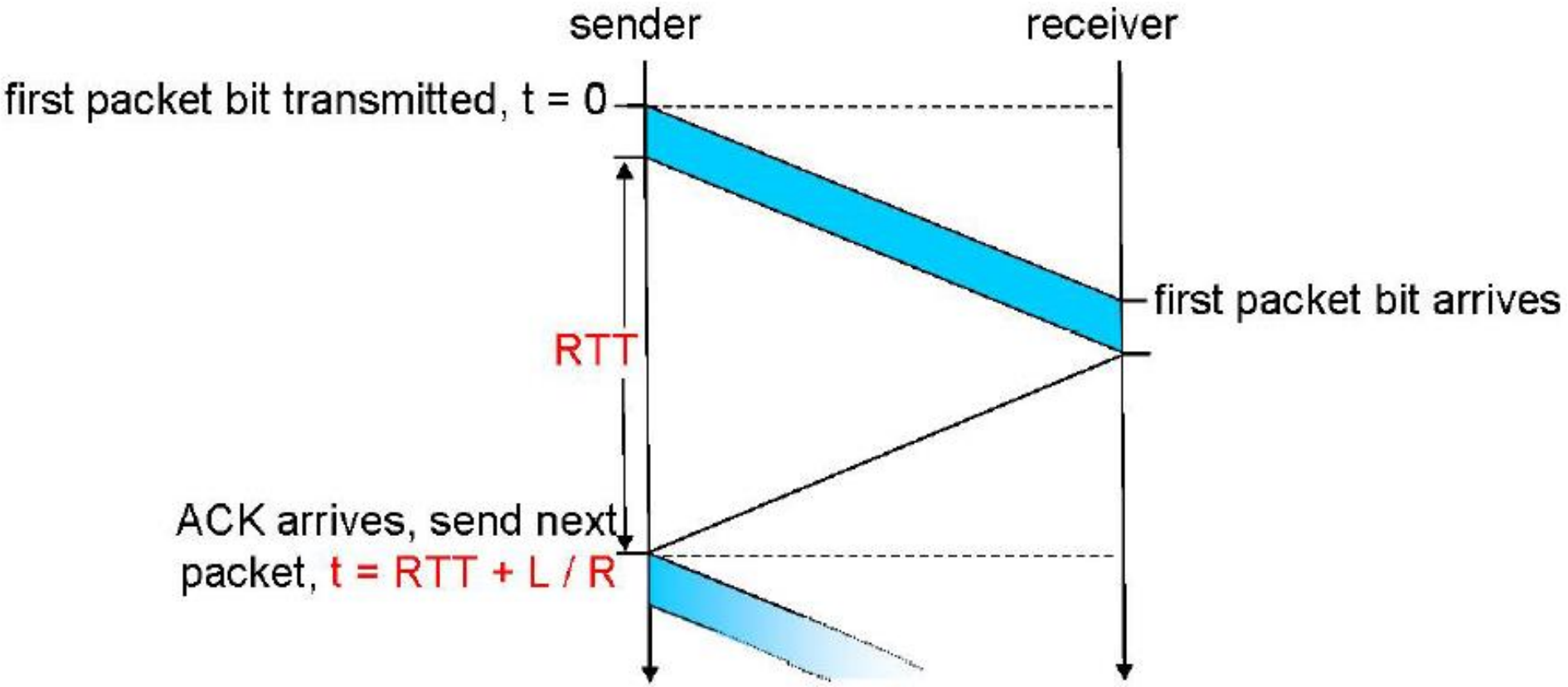


(c) lost ACK



(d) premature timeout

rdt3.0: Problem „Stop and Wait“



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!)

Beispielrechnung

RTT	= 30 ms	(Lichtgeschwindigkeit!)
R	= 1 Gb/sec	(Übertragungsrate)
L	= 1000 bytes	(Paketlänge)
t_{trans}	= 8 ms	(Übertragungszeit)

Auslastung des Senders

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

Erreichbarer Durchsatz

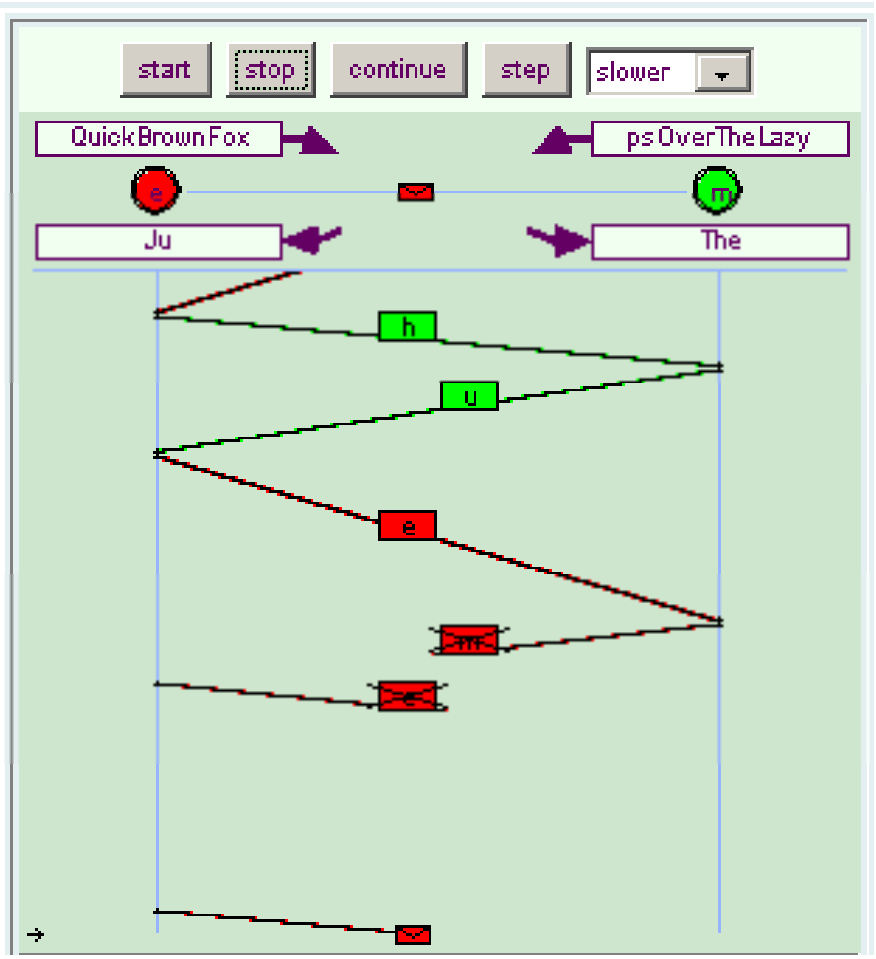
1000 bytes in 30.0008 ms = 33.3 KB/sec

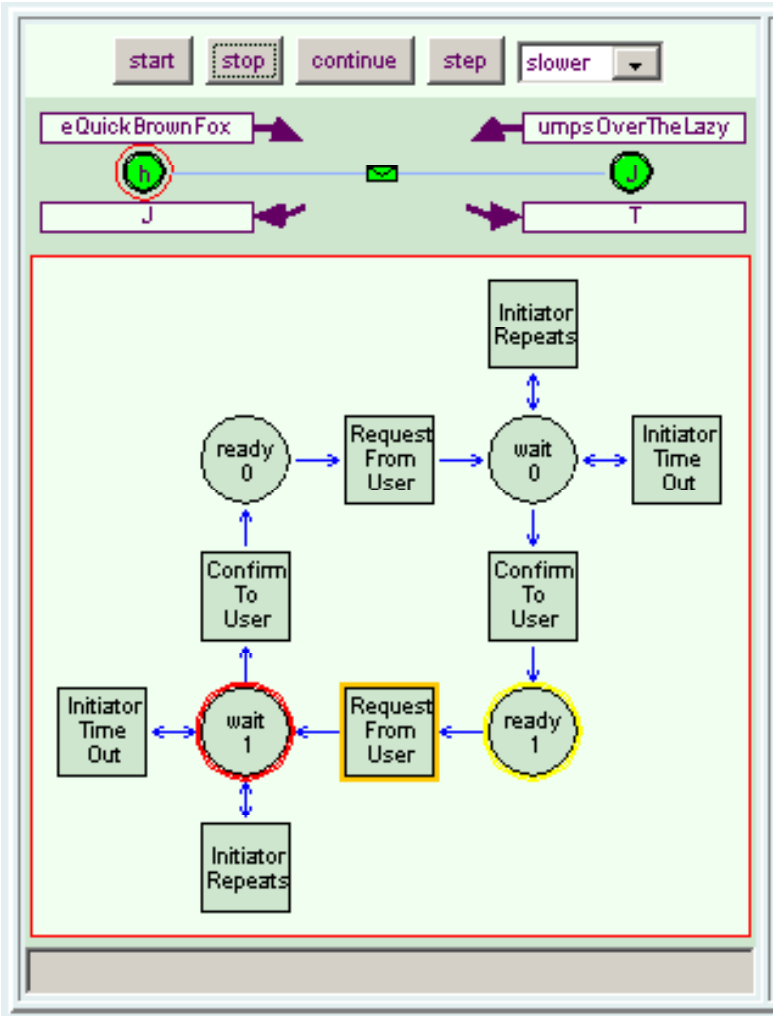
Über eine 1 Gb/sec Leitung!

Protokollablauf: Alternating Bit Protokoll

Siehe URL:

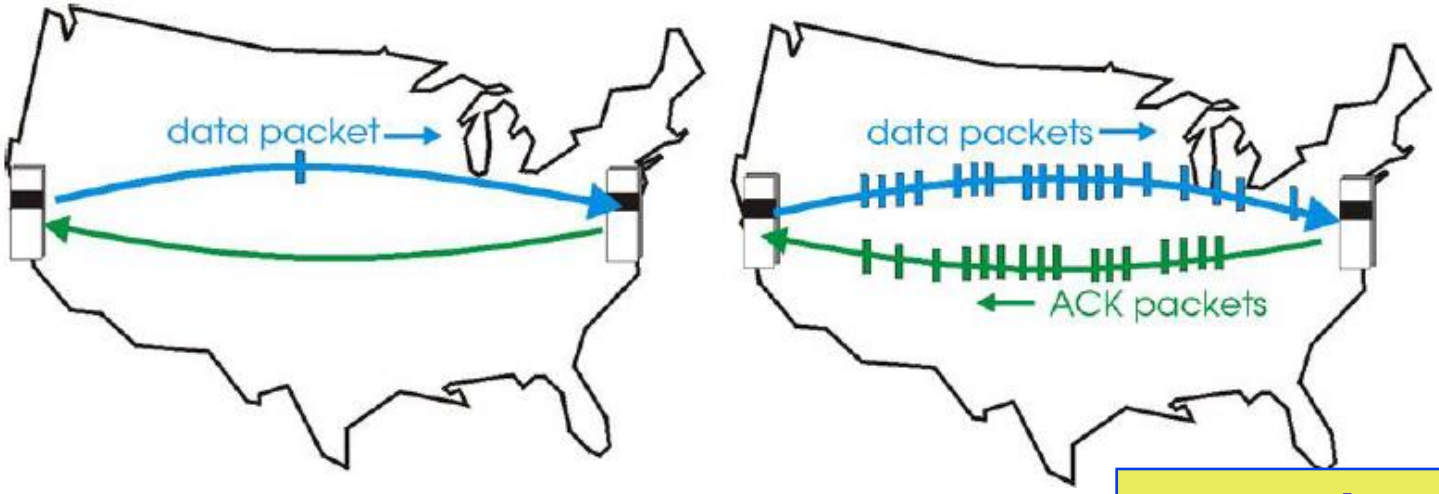
http://ls4-www.informatik.uni-dortmund.de/RVS/MA/hk/OrdnerVertAlgo/VertAlgo.html





Pipeline-Protokolle

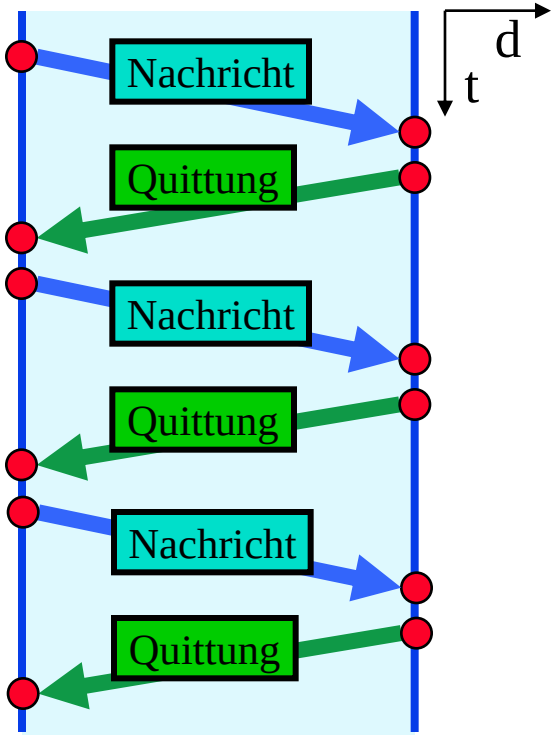
- Wie geht es besser?
  - Neues Paket schon senden, bevor ACK eingetroffen
  - Aber nicht beliebig viele, da Pakete gespeichert werden müssen
- Umfang der Sequenznummer erhöhen
  - (Einführung eines Kreditrahmens bzw. Fensters)



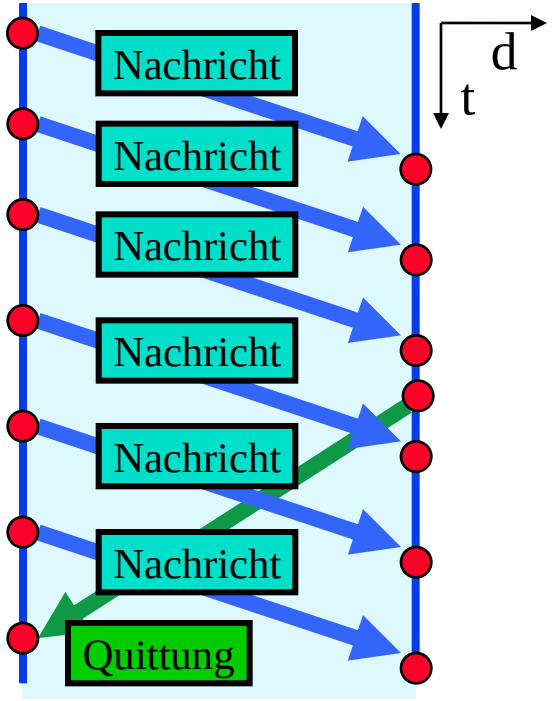
Zwei Realisierungsmöglichkeiten:
- Go-back-n
- Selective-Repeat

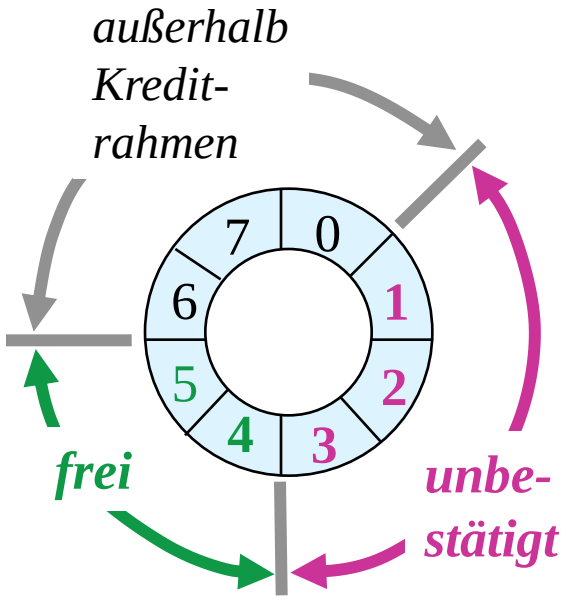
Pipeline Protokoll oft als Schiebefenster-Protokoll

- Stop and Go - Protokolle

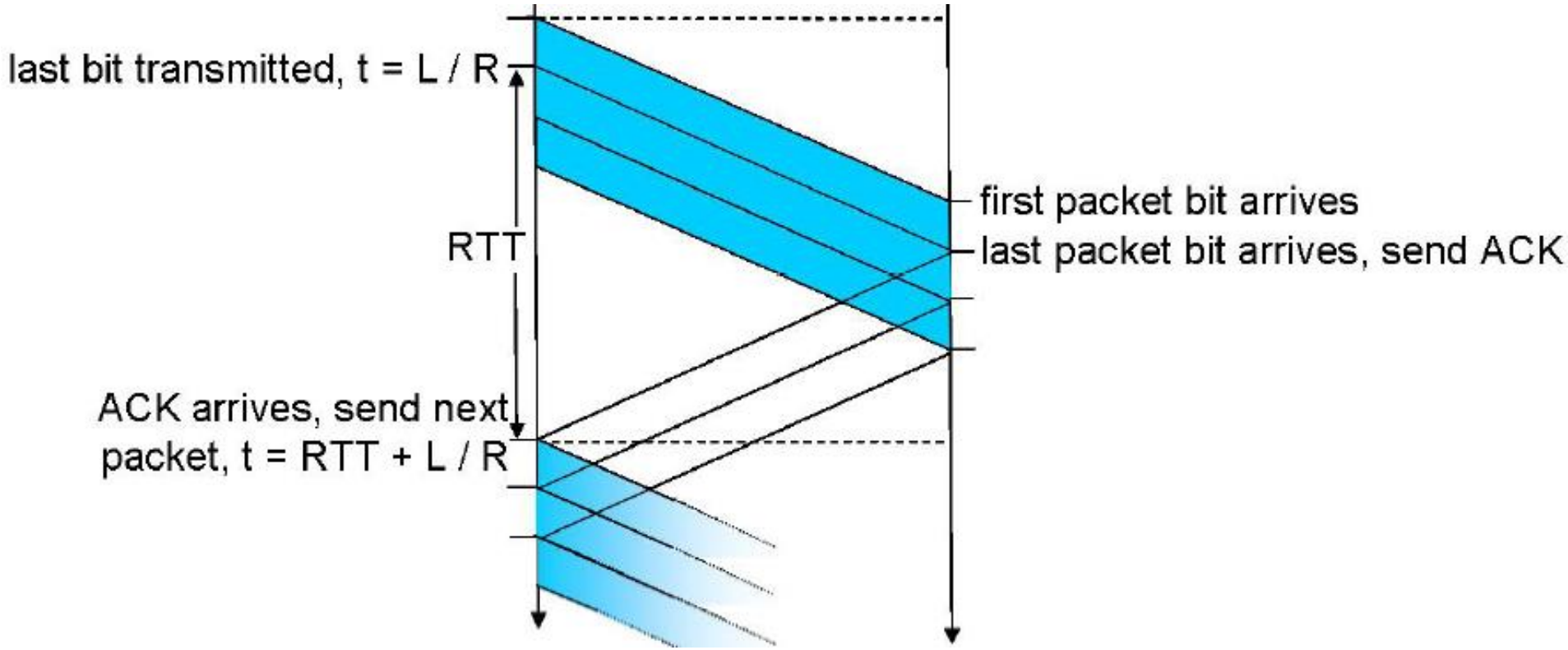


- Sliding-Window - Protokolle
  - Sendekredit
  - Summenquittung
  - Wiederkehrende Laufnummern





Pipeline Protokolle: Bessere Kanalausnutzung



Unser Beispiel für Fenstergröße 3:

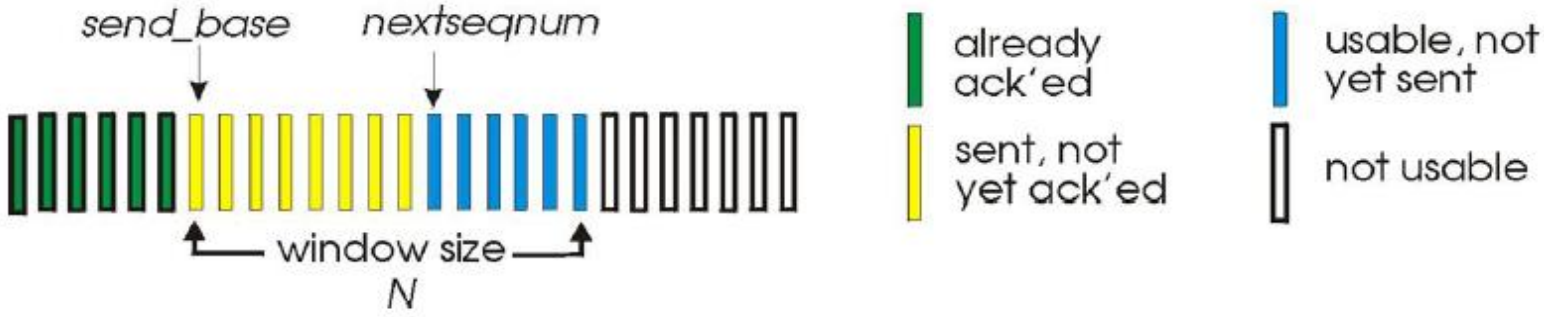
U_{Sender} = (3L/C)/(RTT+L/C) = 0.024 / (30.008) = 0.0008

bzw. Übertragungsrate 100 KB/sec

Pipeline Protokolle: Go back n

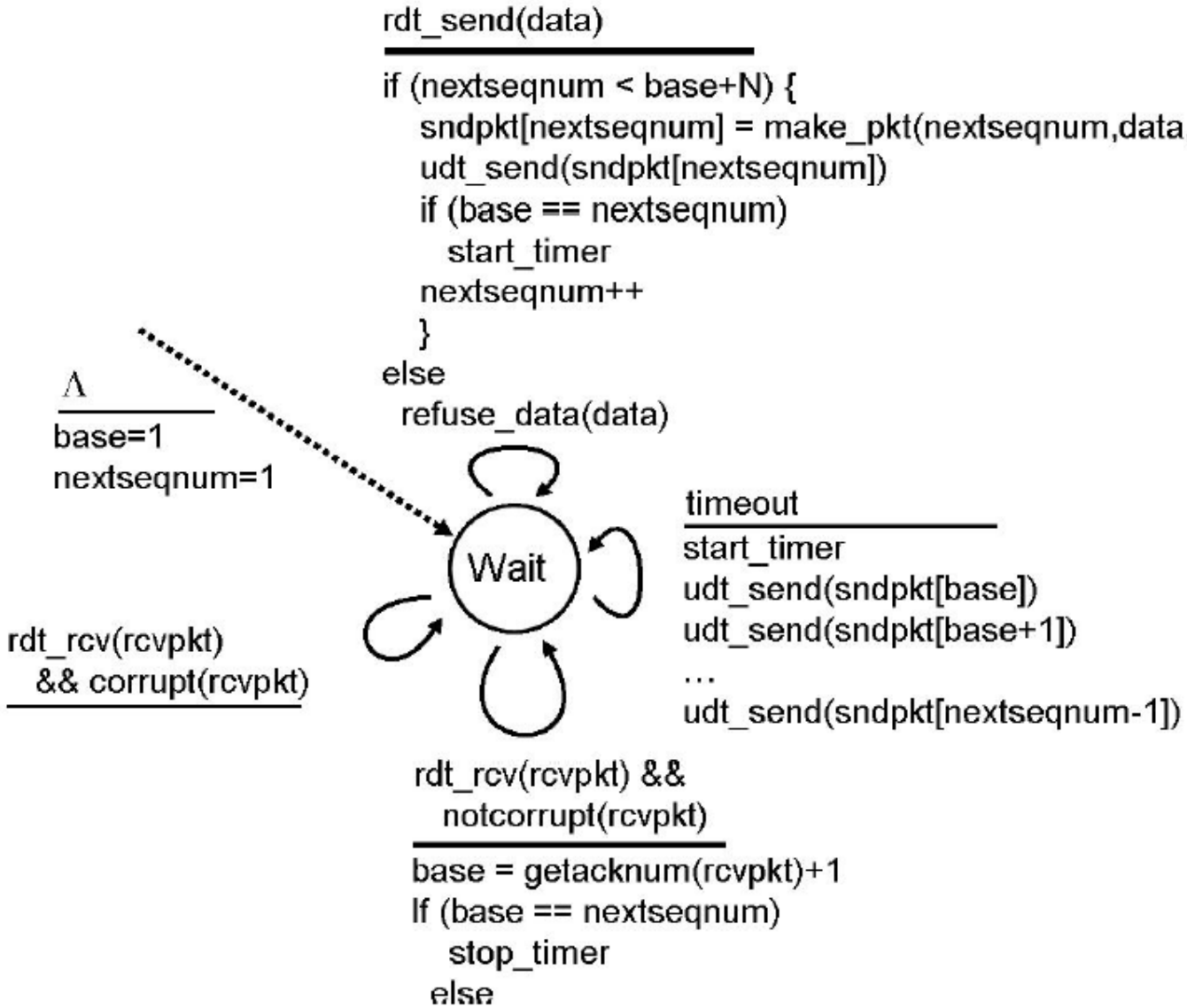
Sender

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

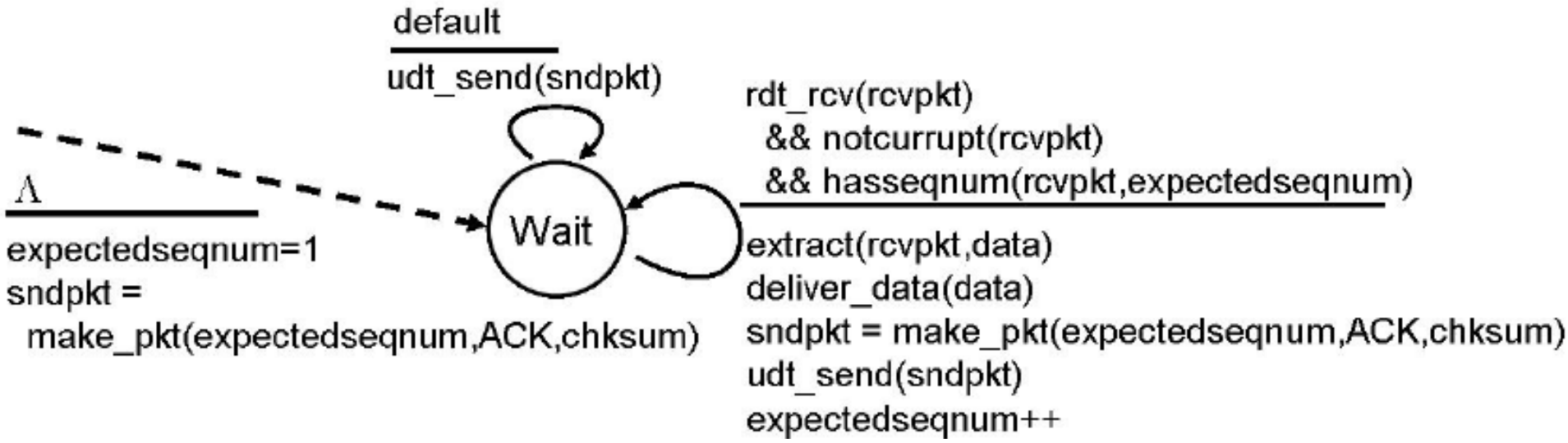


- ACK für Paket n bestätigt alle Pakete, die bis zu Paket n gesendet wurden
  (Sammel-ACK bzw. Cumulative ACK)
- Ein Timer läuft jeweils für das älteste nicht bestätigte Paket
  bei Eintreffen eines ACKs wird Timer neu gestartet
  (sofern noch nicht alle ACKs eingetroffen)
- Bei Timerablauf Wiederholung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

Pipeline Protokolle: Go back n - Sender

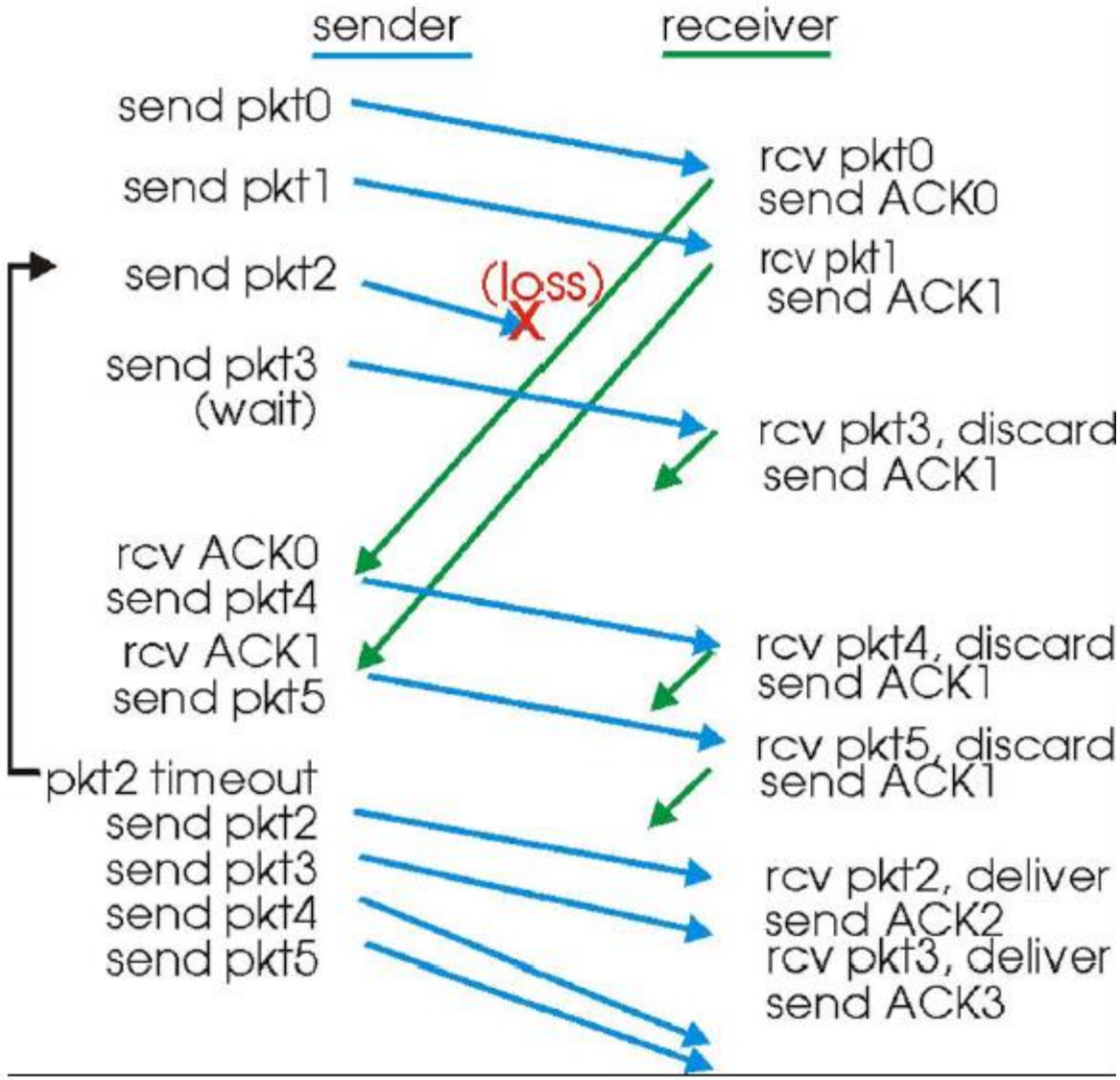


Pipeline Protokolle: Go back n - Empfänger



- 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, senden

Pipeline Protokolle: Go back n - Ablauf



Pipeline Protokolle: Selective Repeat

Empfänger

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

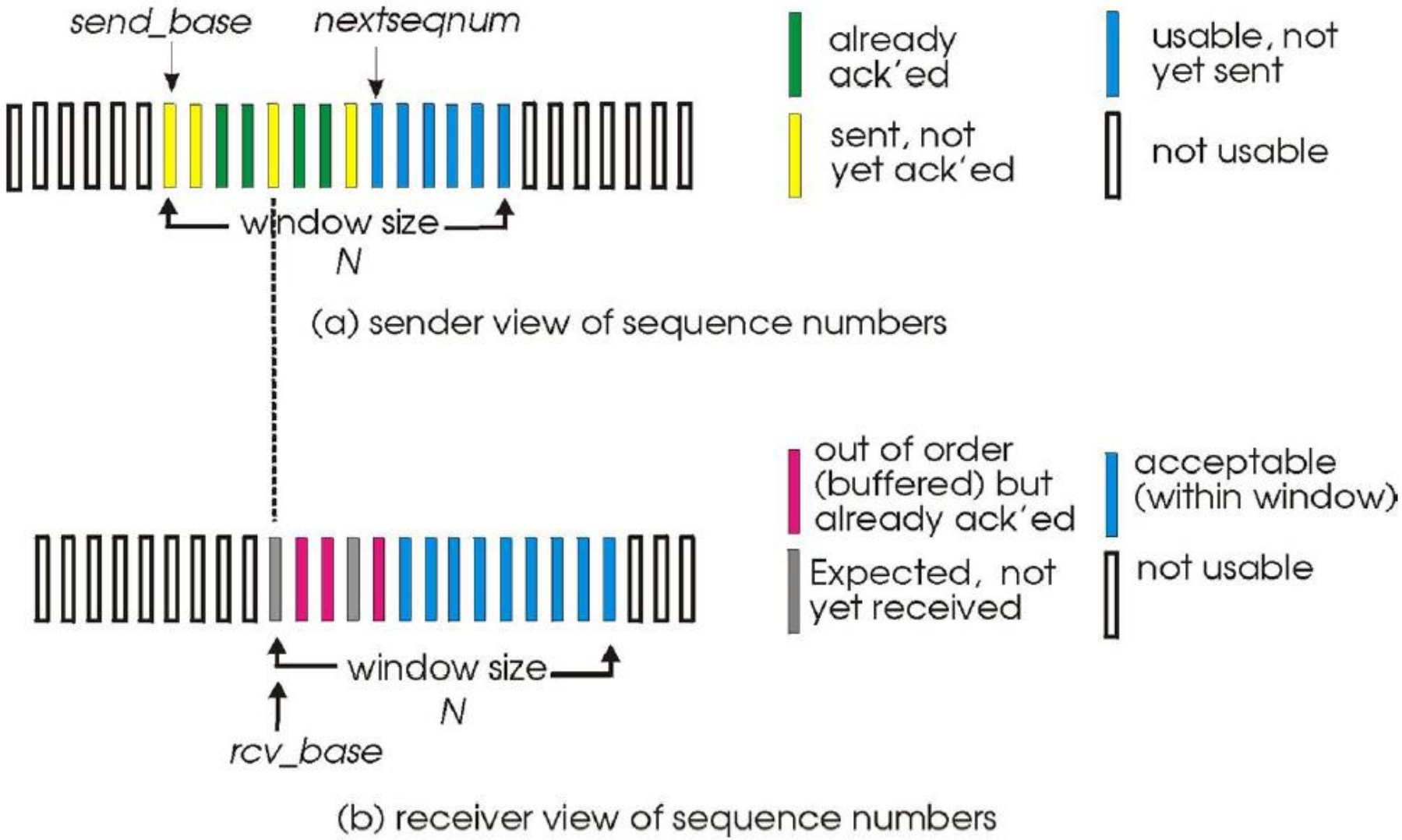
Sender

- 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

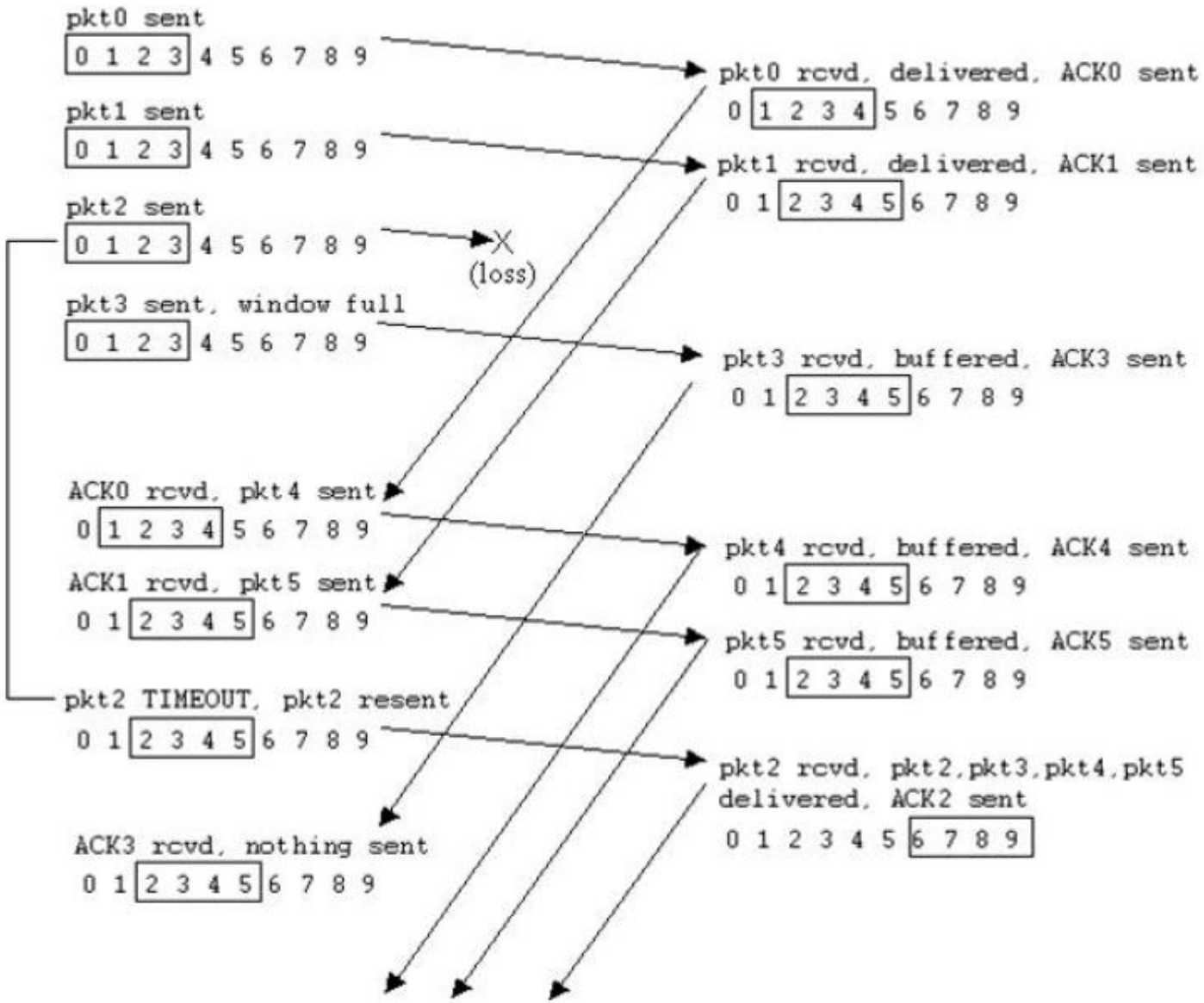
Sender

- 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

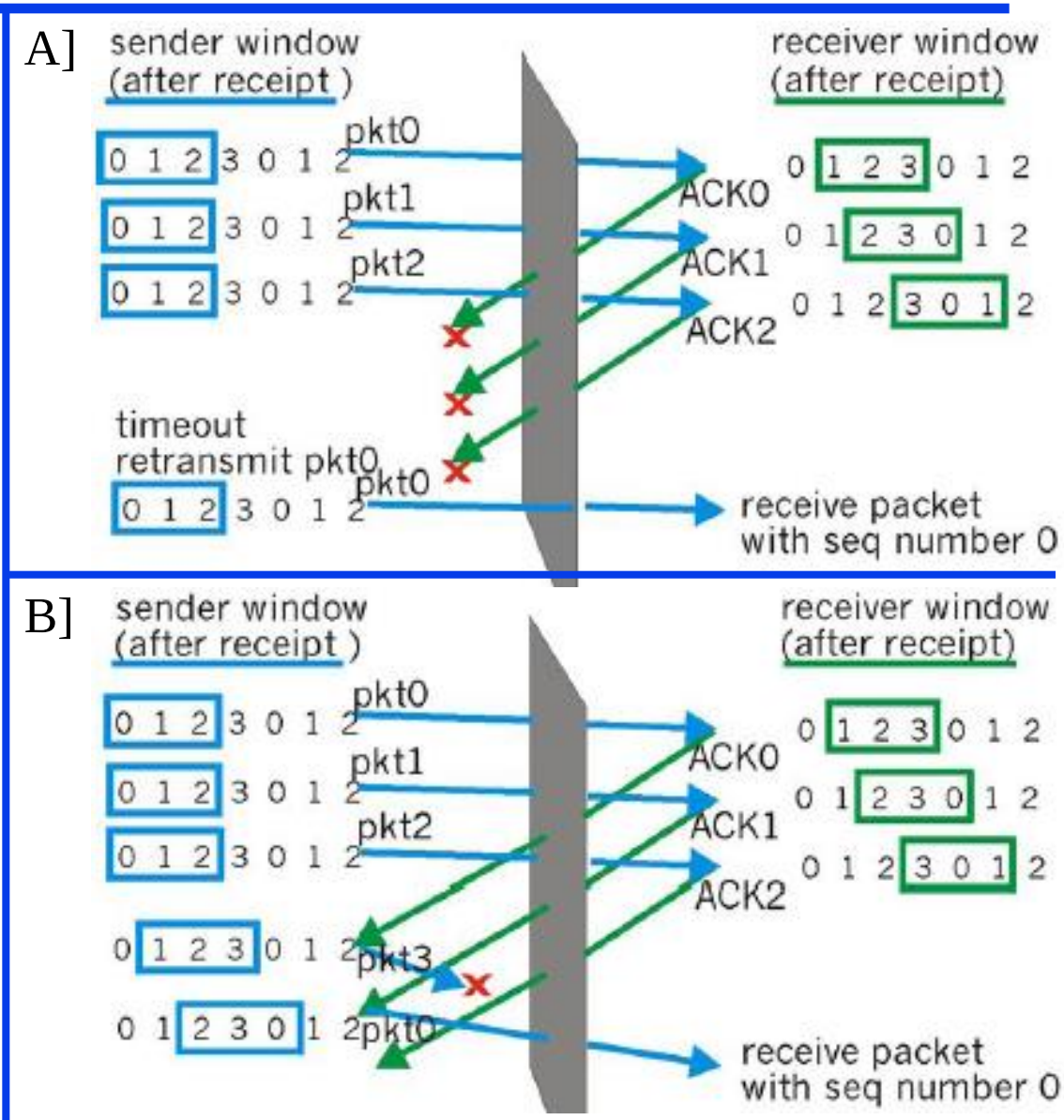
Pipeline Protokolle: Selective Repeat - Ablauf



Pipeline Protokolle: Selective Repeat - Ablauf

Problem-Szenario

- Sequenznummern 0,1,2,3
- Fenstergröße = 3
- Empfänger sieht keinen Unterschied zwischen den beiden Szenarien A und B
- In A wird wiederholtes Paket als neues empfangen
- Welche Beziehung muss zwischen der Anzahl der Sequenznummern und der Fenstergröße bestehen?



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

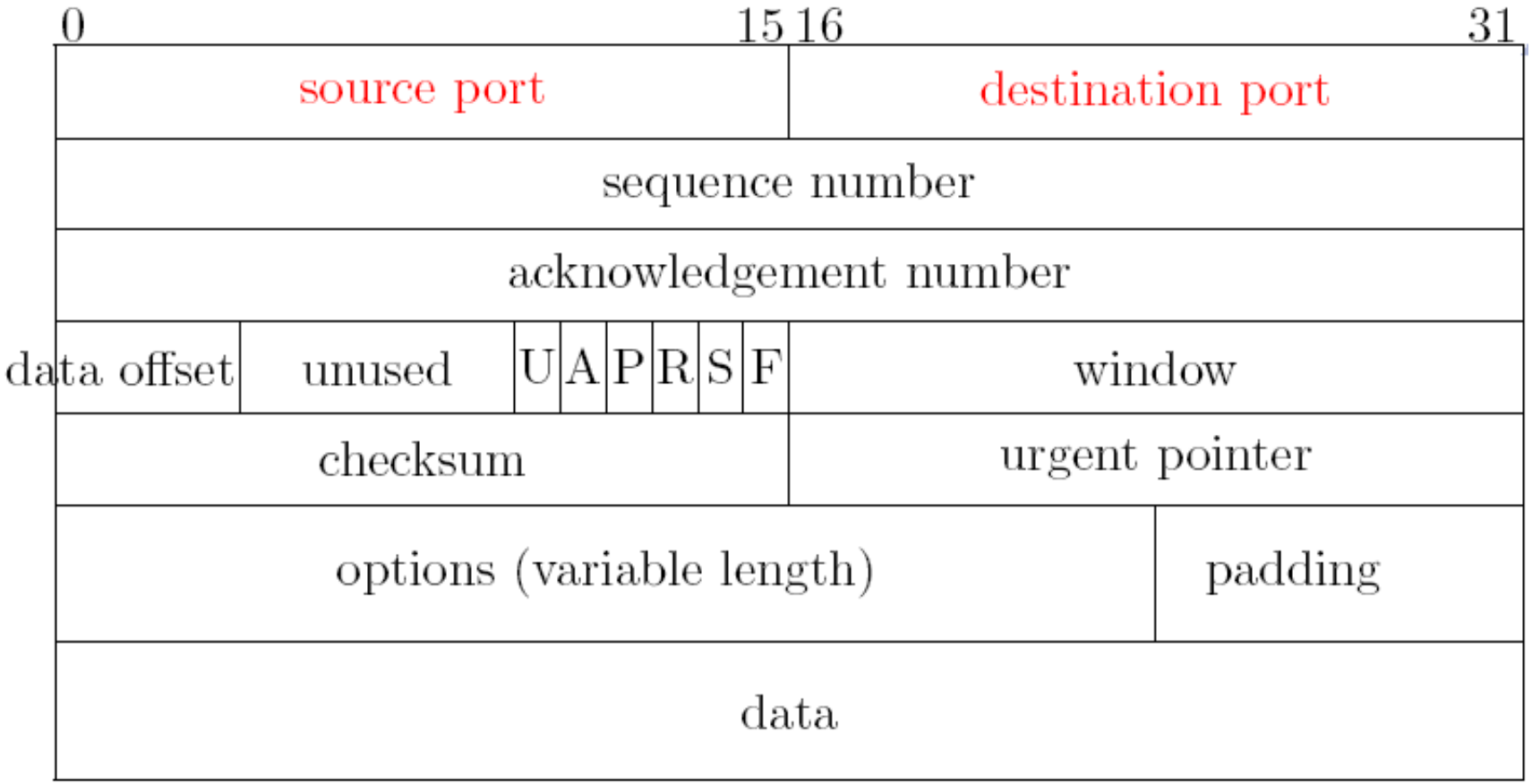
- Zieldienst: **Zuverlässiger Duplex-Bytestrom-Transfer**
- Basisdienst: **Internet – Unzuverlässiger Transfer von IP-Paketen**

→

Protokollmechanismen zur zuverlässigen Übertragung a la **rtd3.0**

- Weiterhin
  - Verbindungsverwaltung:
    Aufbau über 3-Wege-Handshake, Abbau über *close* je Richtung
  - Voll-Duplex und Piggy-Backing
- Sowie später erklärt
  - Stau- und Flusskontrolle

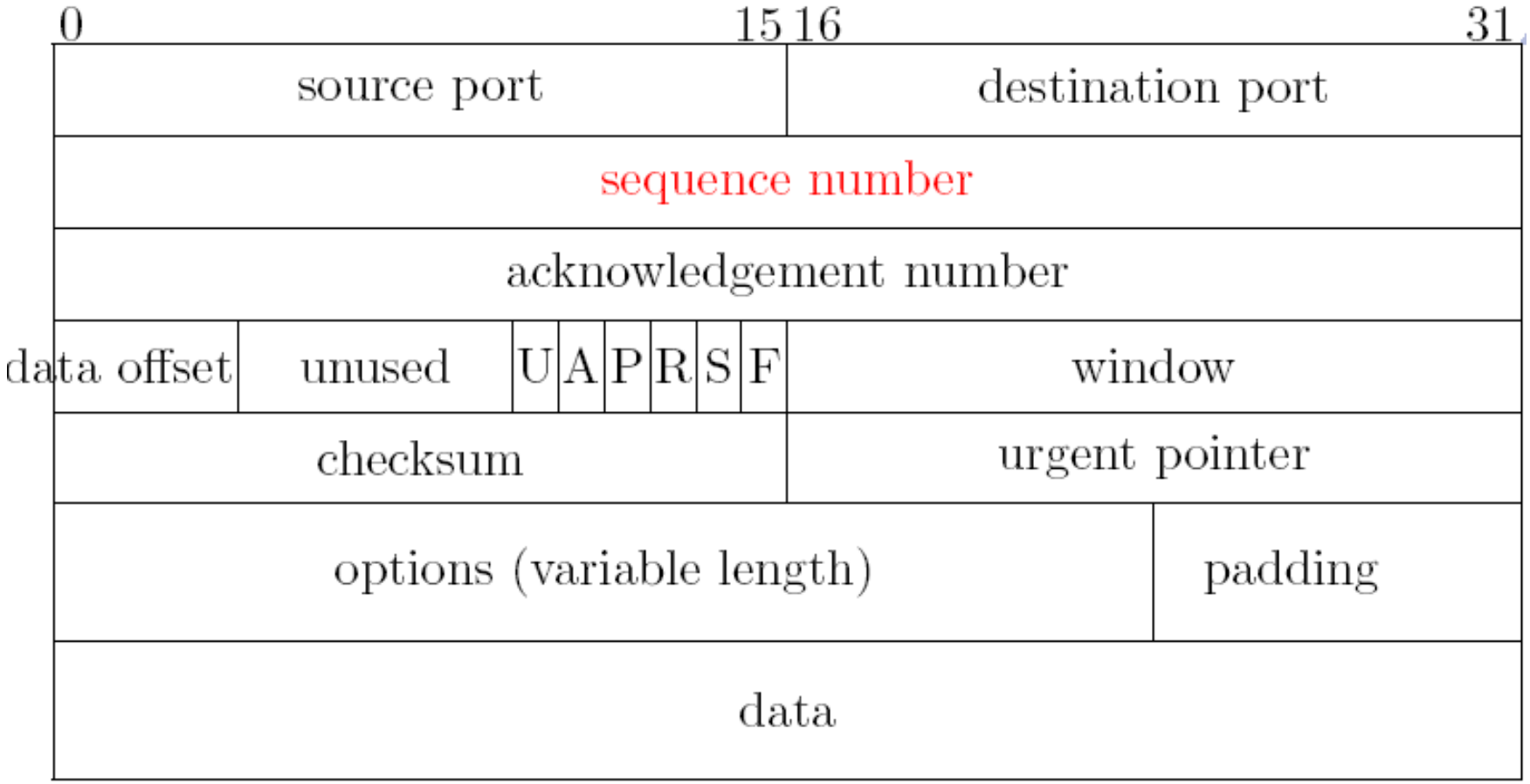
TCP: PDU-Format



Source Port (16 bit): Die Nummer des Quellports

Destination Port (16 bit): Die Nummer des Zielports

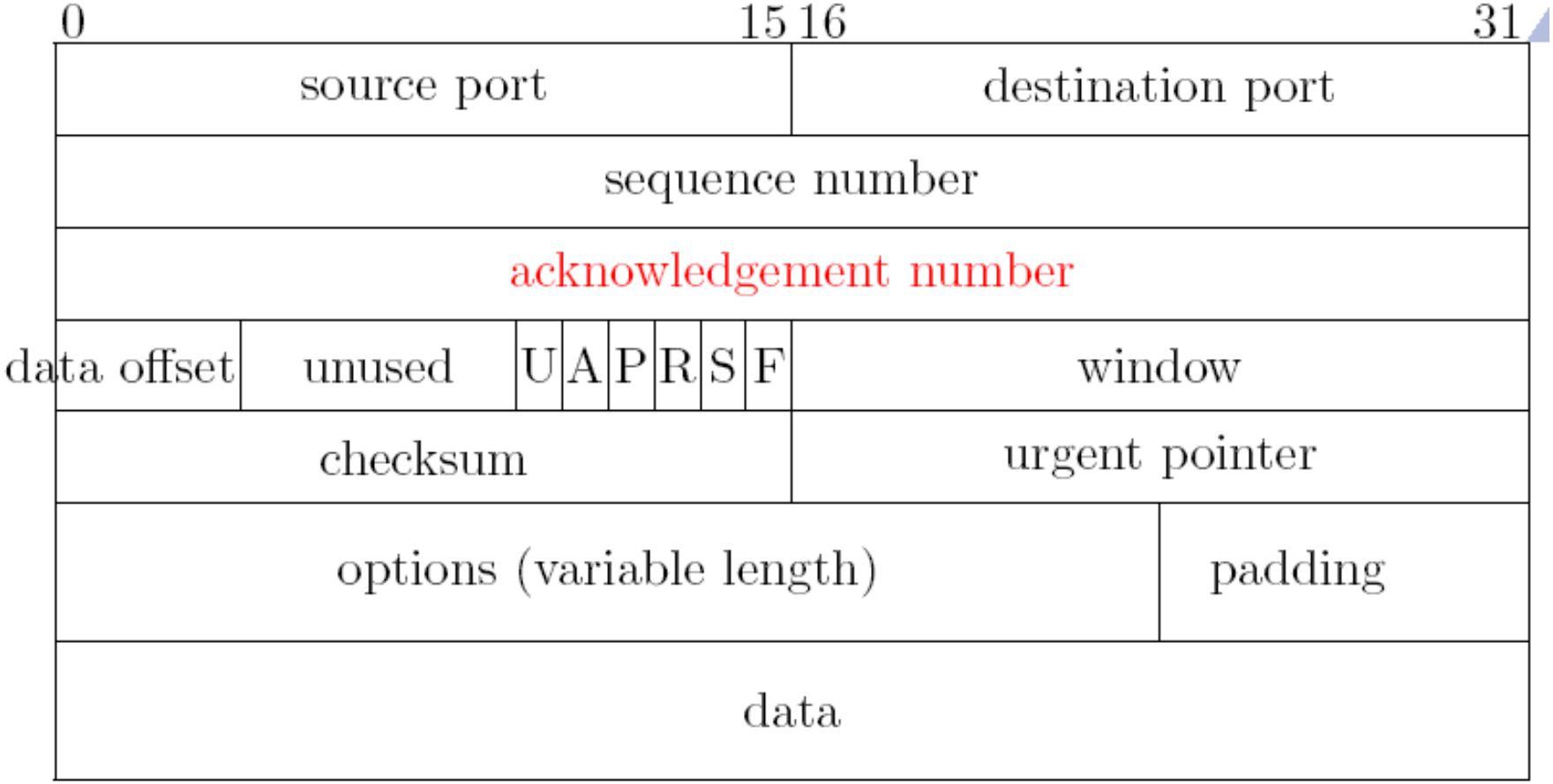
TCP: PDU-Format



Sequenznummer (32 bit):

Die Nummer des ersten Daten-Oktets dieses Segments

TCP: PDU-Format

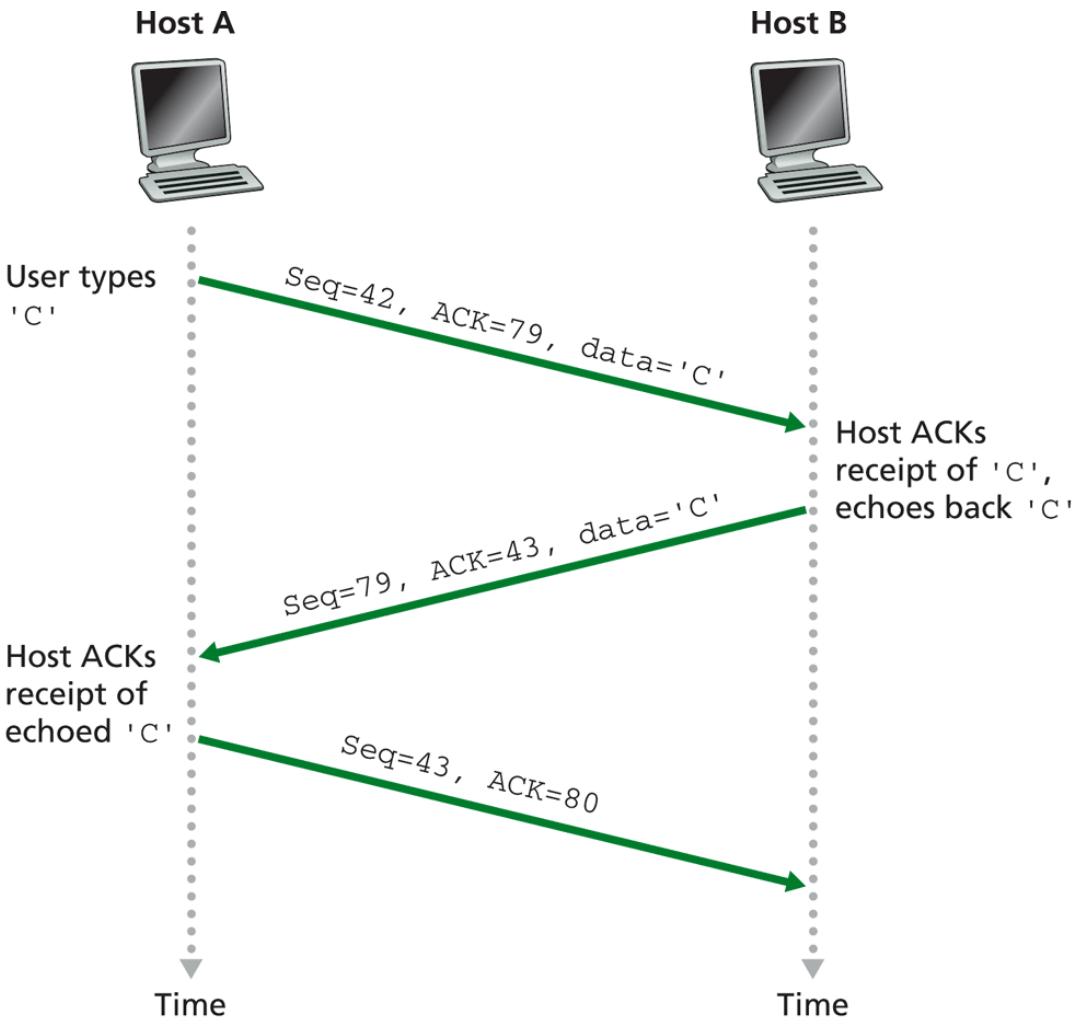


Acknowledgement-Nummer (32 bit):

Die Sequenznummer, die der Sender dieses Segments erwartet als nächstes zu empfangen (signifikant, wenn A-bit gesetzt)

Hinweis: Piggy Backing

TCP: Sequenznummern und Bestätigungen



Initial erwartet

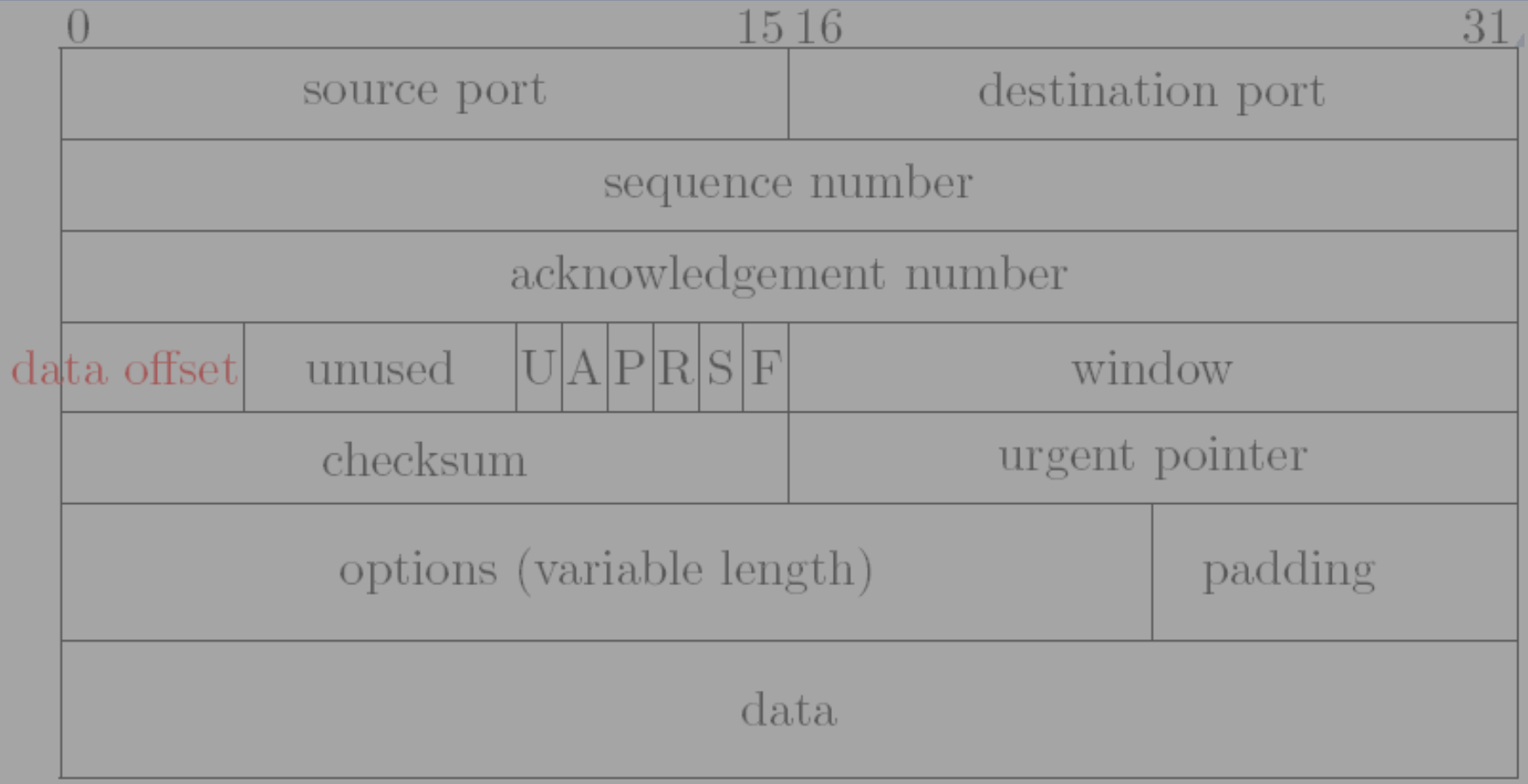
- Host A Byte 79
- Host B Byte 42

Ablauf

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

- Behandlung von Nachrichten außerhalb der Reihenfolge im RFC nicht festgelegt

TCP: PDU-Format

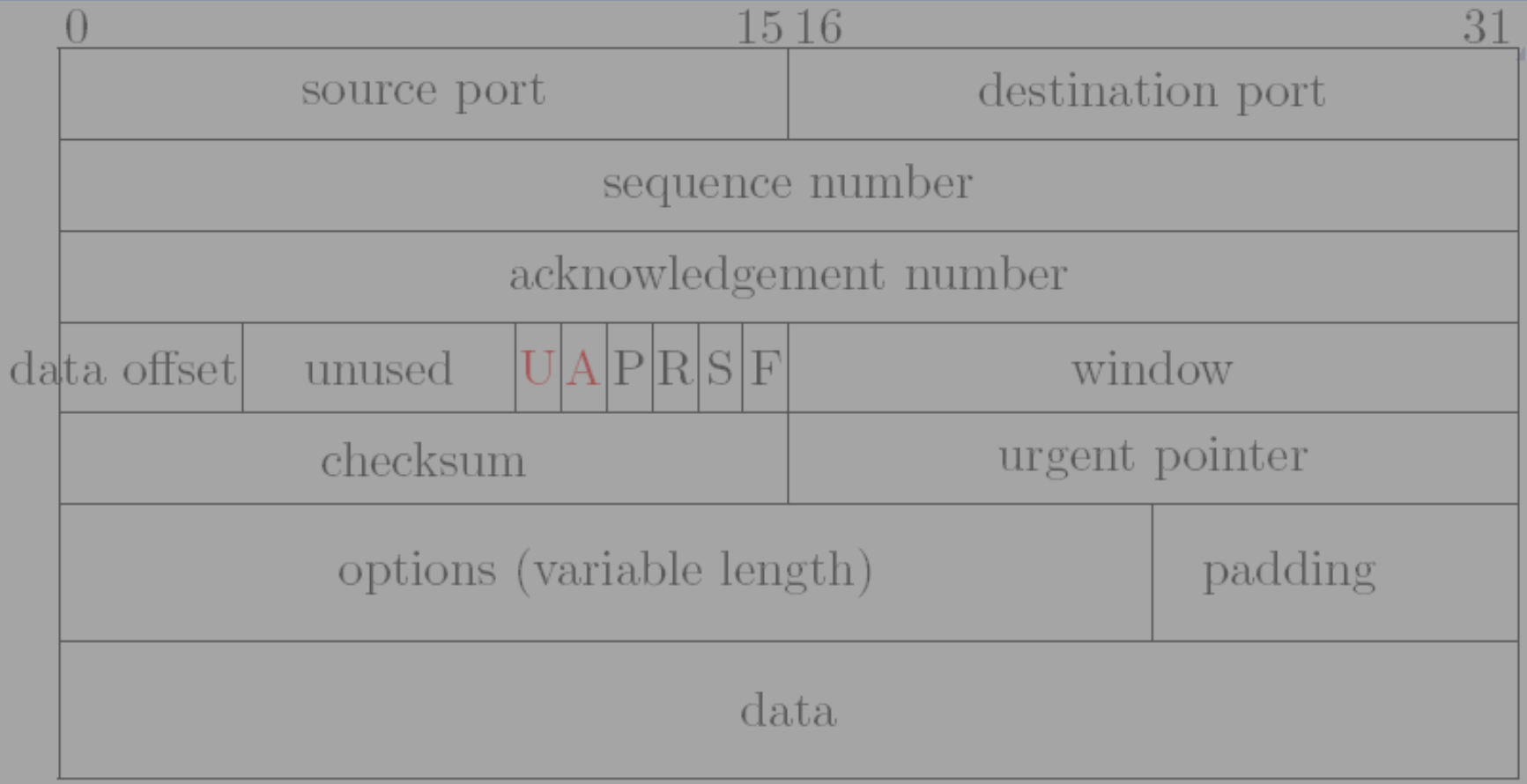


Data Offset (4 bit):

Header-Länge als Vielfaches von 32bit.

Wird benötigt wegen variabler Länge des Feldes *options*.

TCP: PDU-Format

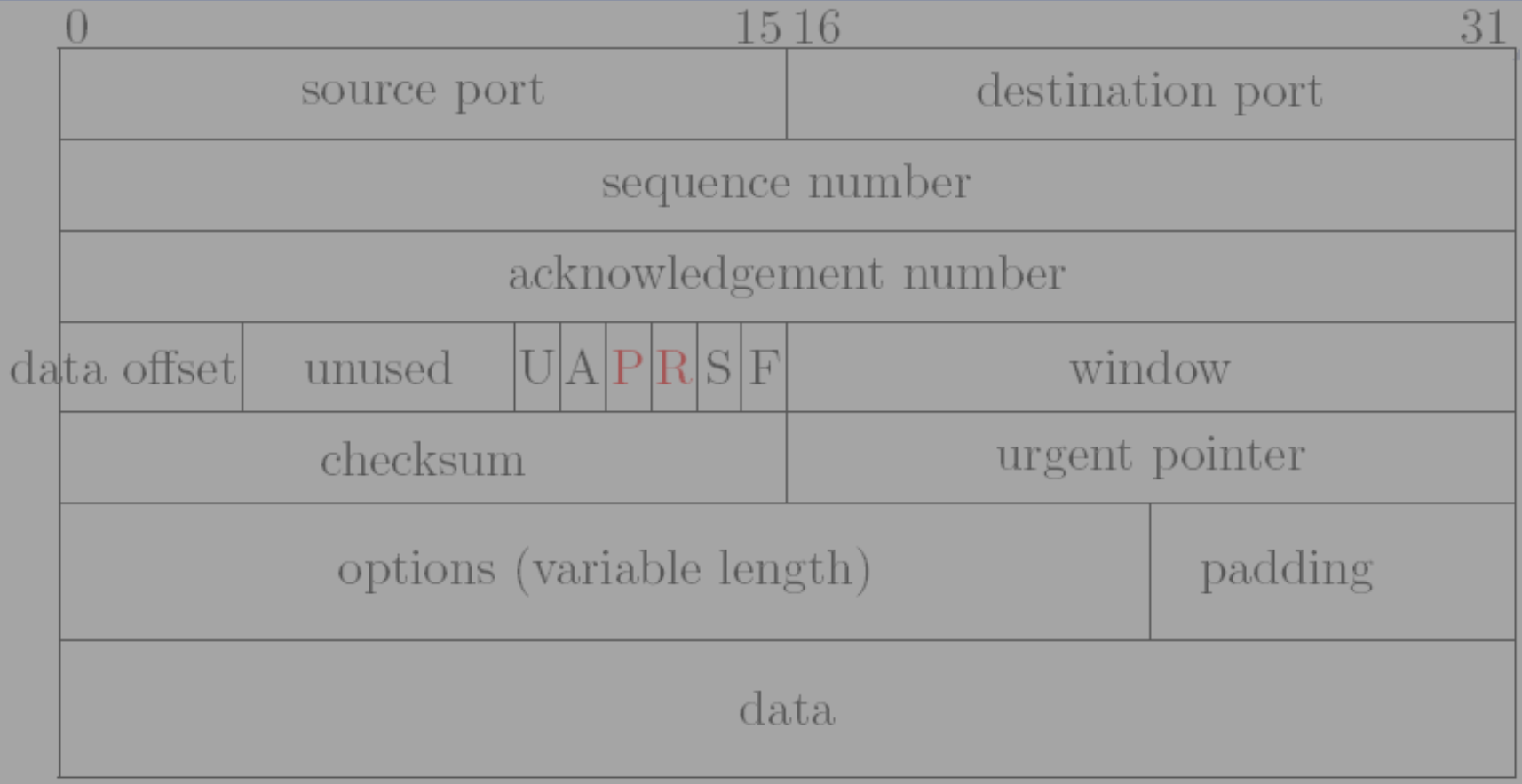


URG (1 bit): Urgent Poiner-Feld ist signifikant.

ACK (1 bit): Acknowledgement number-Feld ist signifikant.

Hinweis: Hochpriore Daten

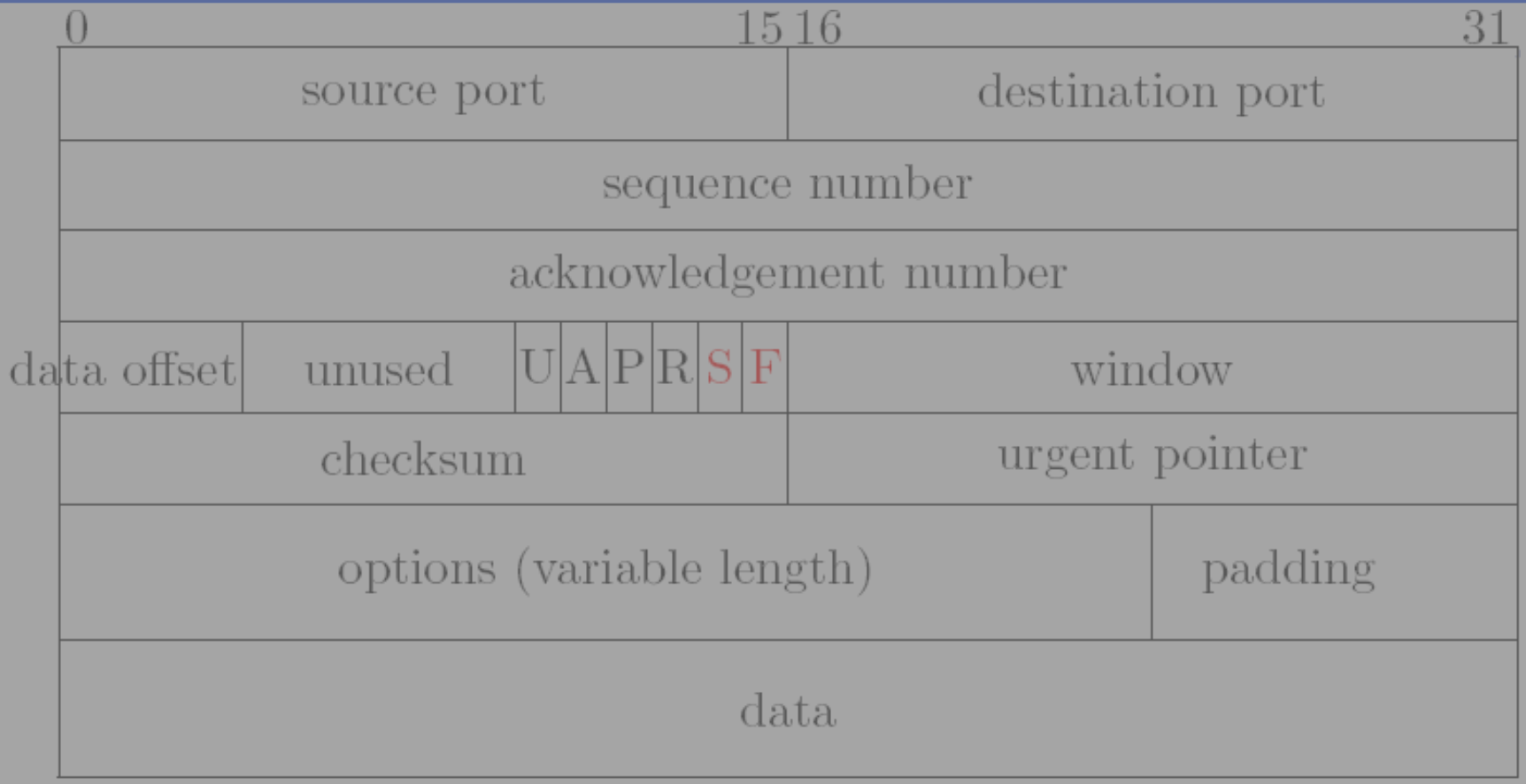
TCP: PDU-Format



PSH (1 bit): Push-Funktion, Daten sofort an höhere Schicht leiten

RST (1 bit): Reset der Verbindung

TCP: PDU-Format

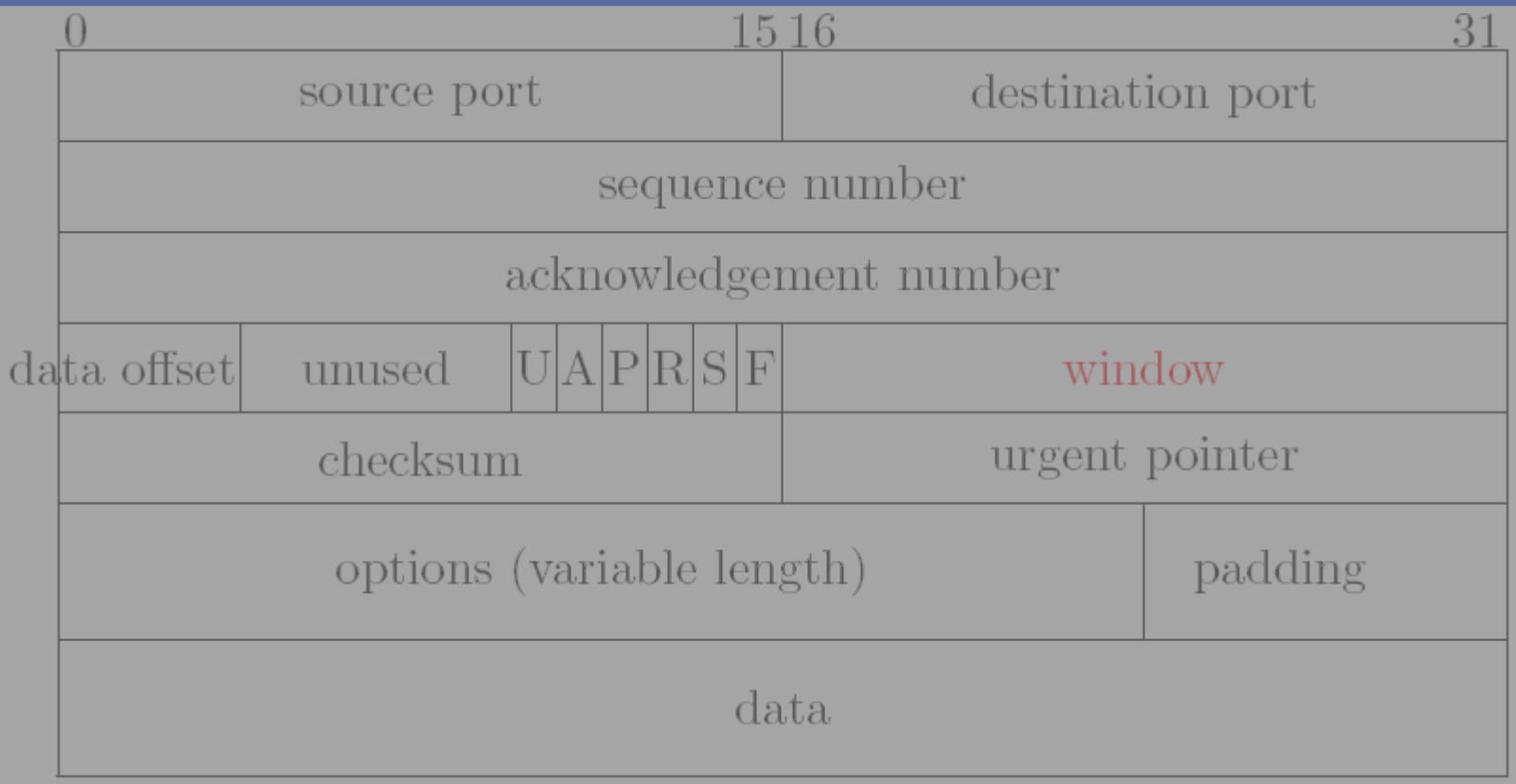


SYN (1 bit): Synchronisation der Sequenznummern

FIN (1 bit): Keine weiteren Daten

Hinweis: „close“

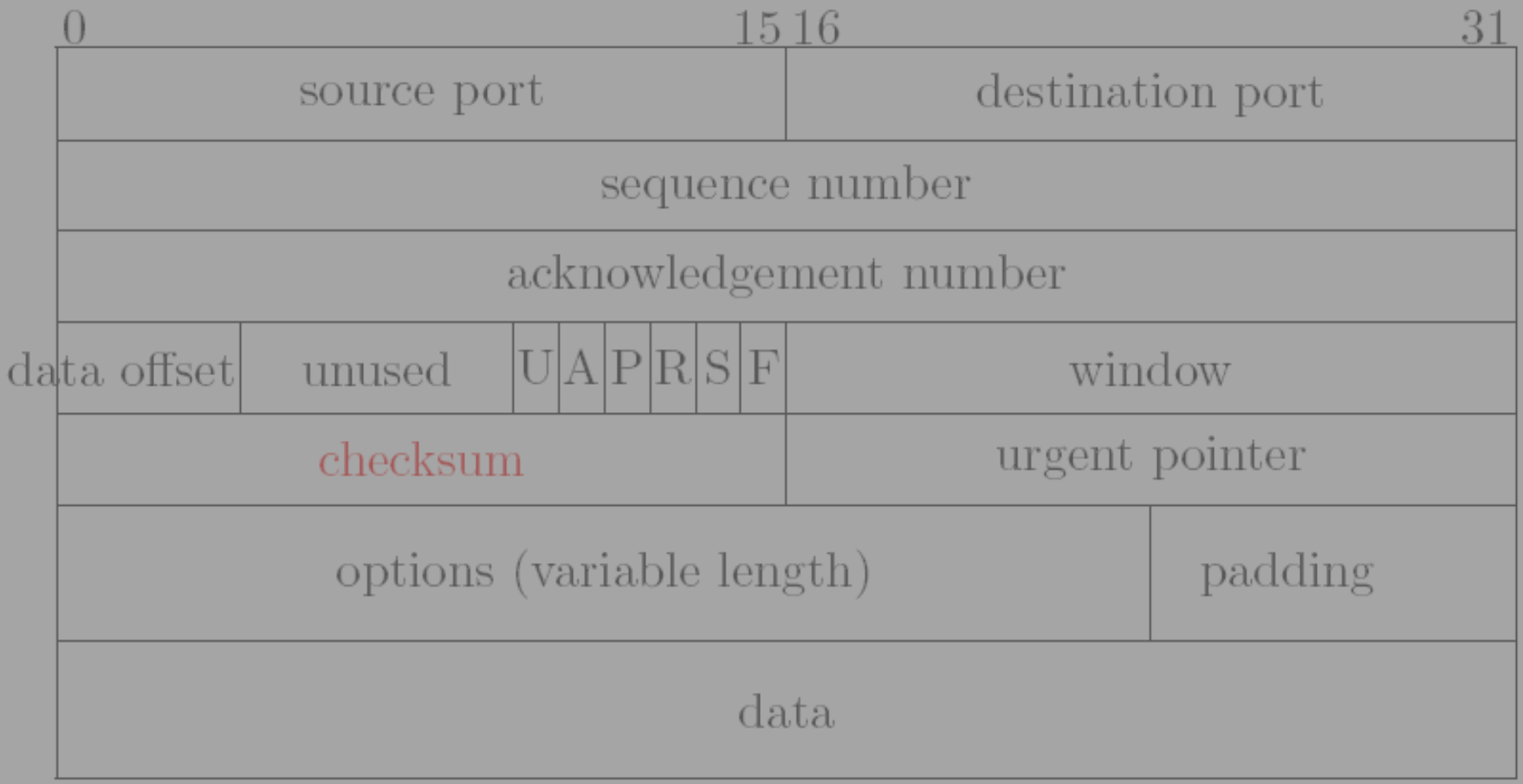
TCP: PDU-Format



window (16 bit): Die Anzahl der Oktets, die der Sender dieses Segments empfangen kann

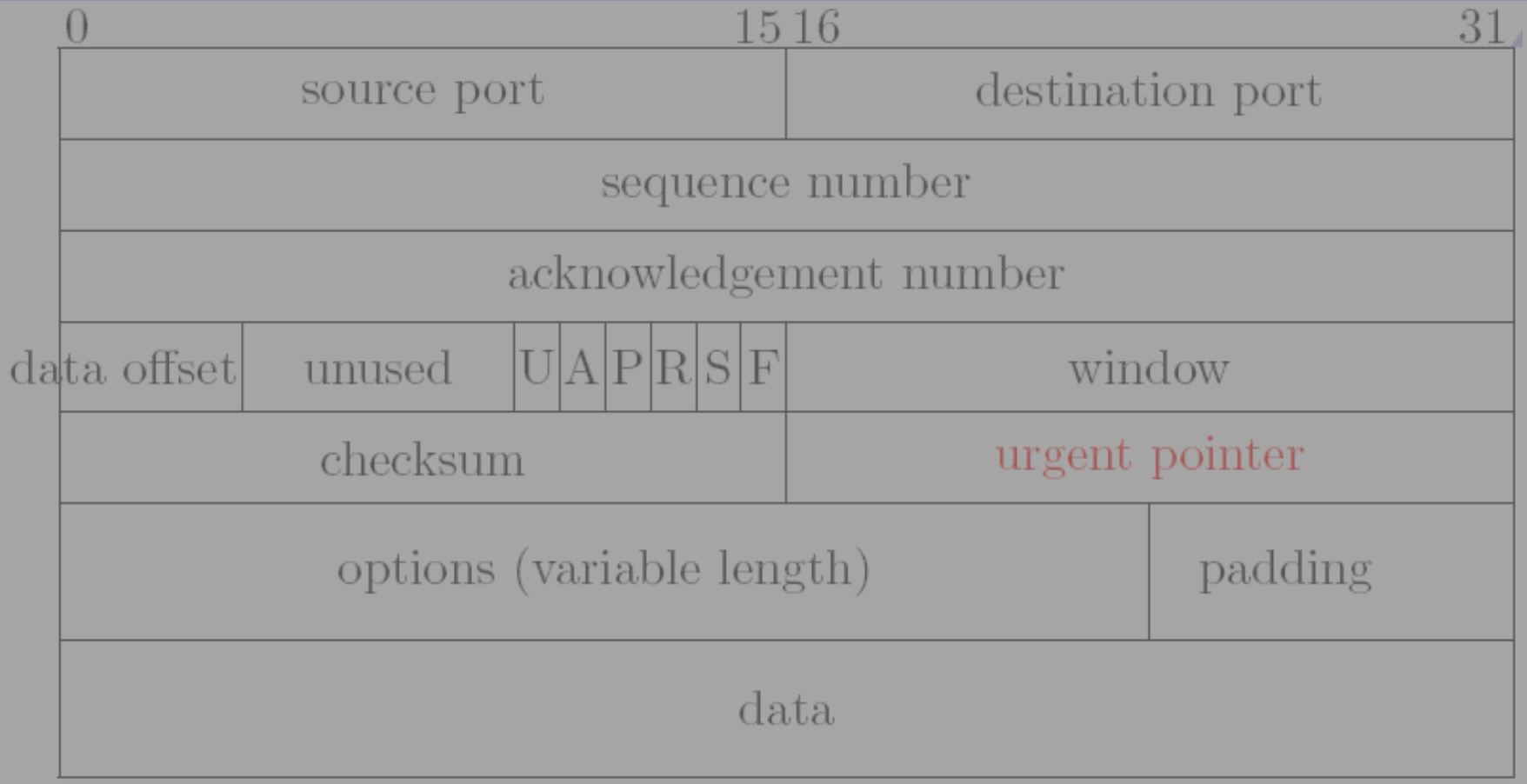
Hinweis: Sendekredit zur Flusskontrolle

TCP: PDU-Format



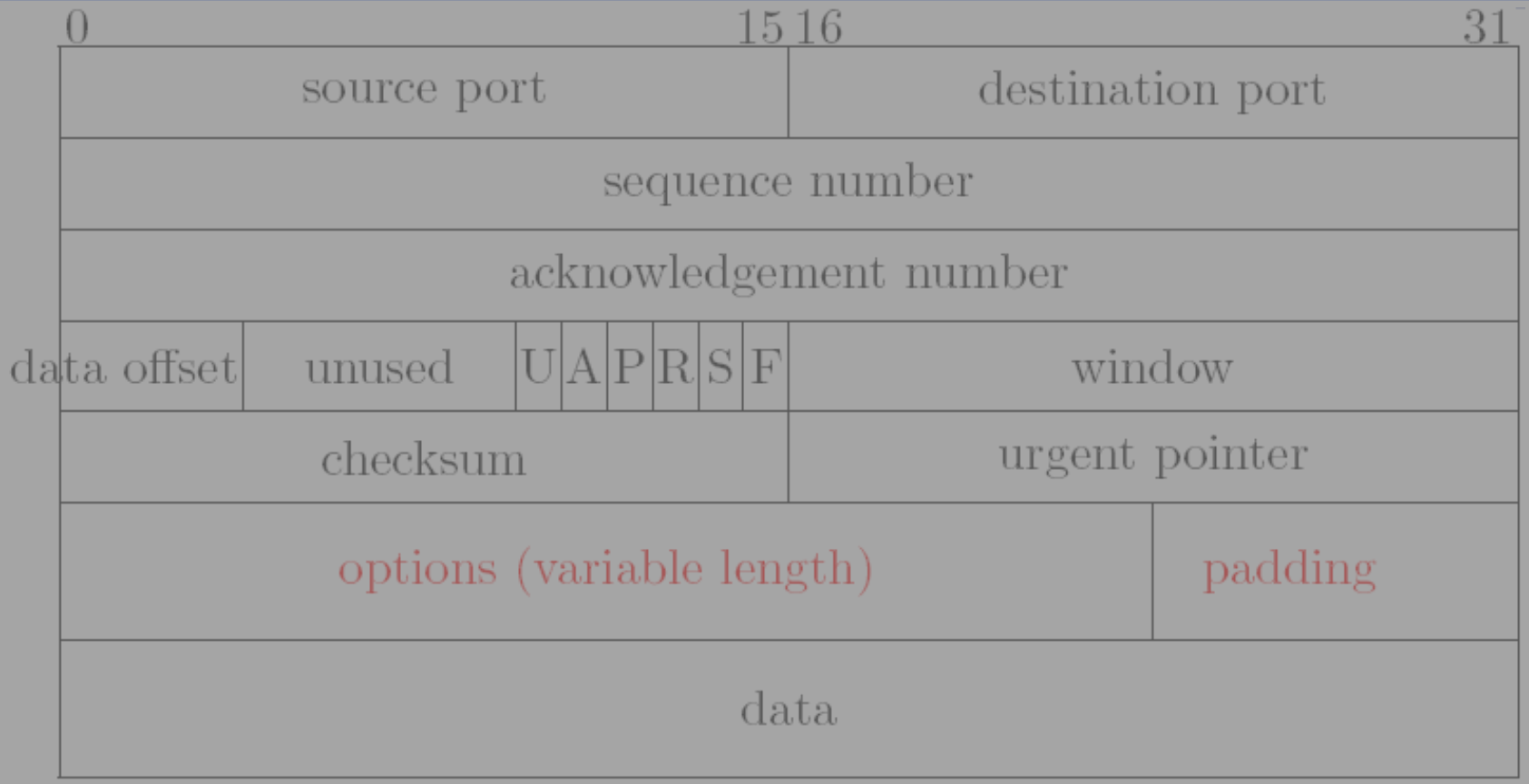
checksum (16 bit): Die Prüfsumme (Berechnung siehe Folie 11)

TCP: PDU-Format



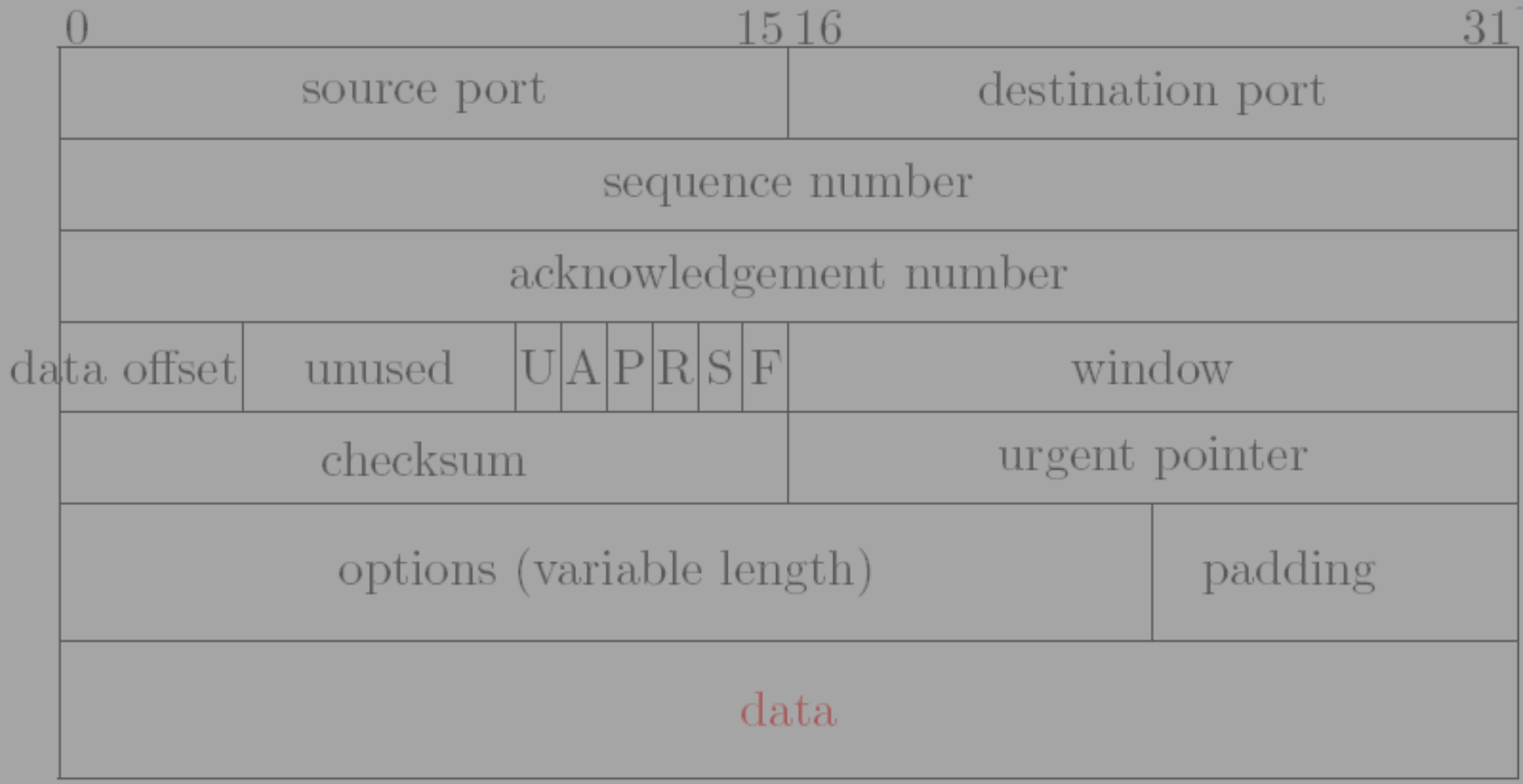
Urgent pointer (16 bit): Zeiger auf dringende Daten dieses Segments.
Nur signifikant, wenn U-Bit gesetzt

TCP: PDU-Format



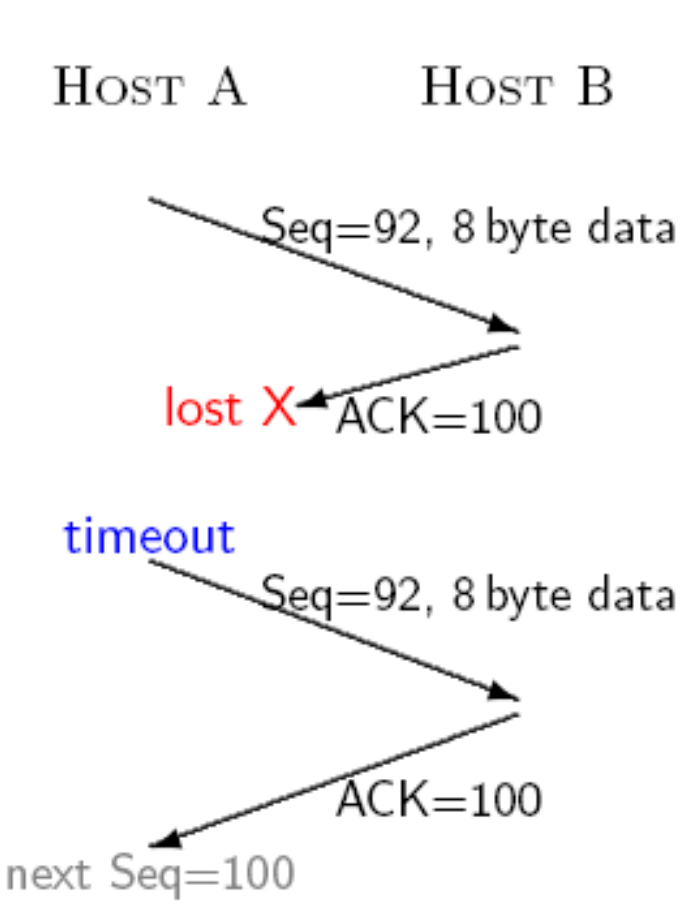
Optionen (variabel): Feld wird auf ein Vielfaches von 32 Bit aufgefüllt (mit padding Bits)

TCP: PDU-Format

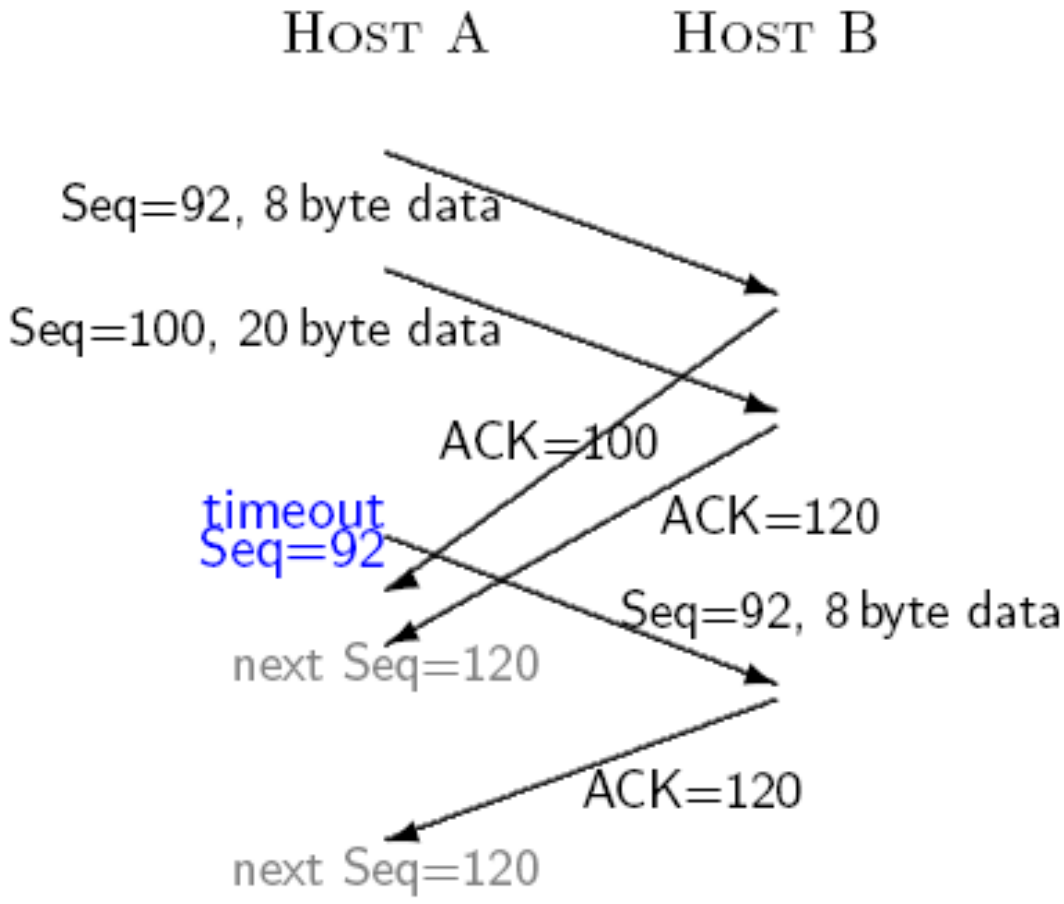


Data (variabel): Anwendungsdaten

TCP: Einige Szenarien

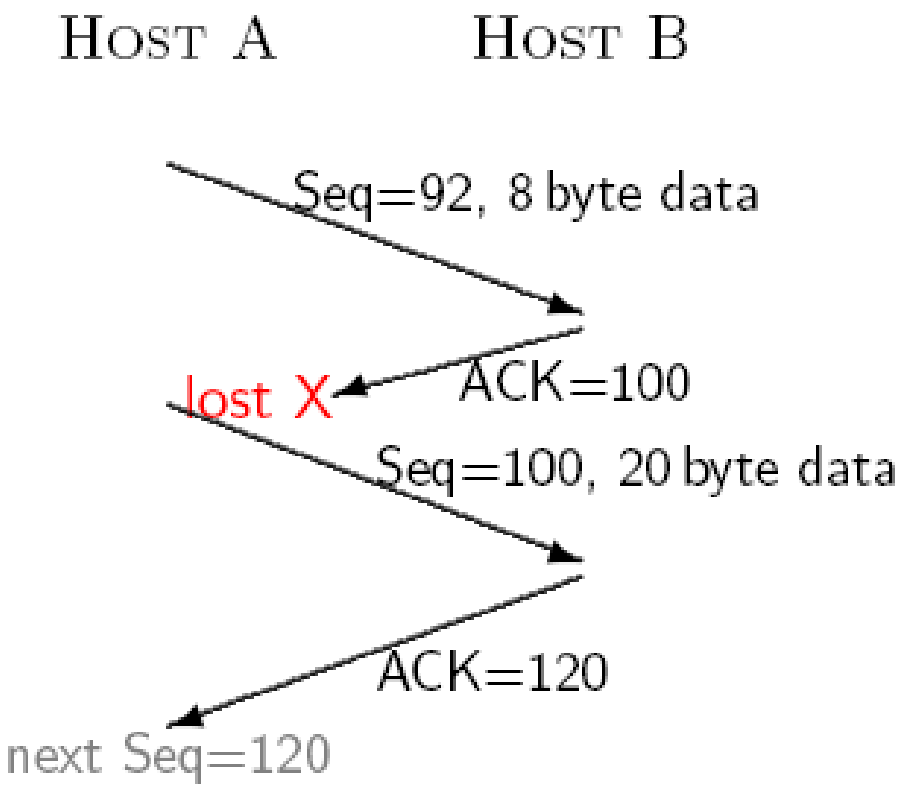


verlorenes ACK



vorzeitiger Timeout

TCP: Einige Szenarien



Kumulatives ACK

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 RTT sein, aber nicht wesentlich größer!
  (RTT ändert sich jedoch)

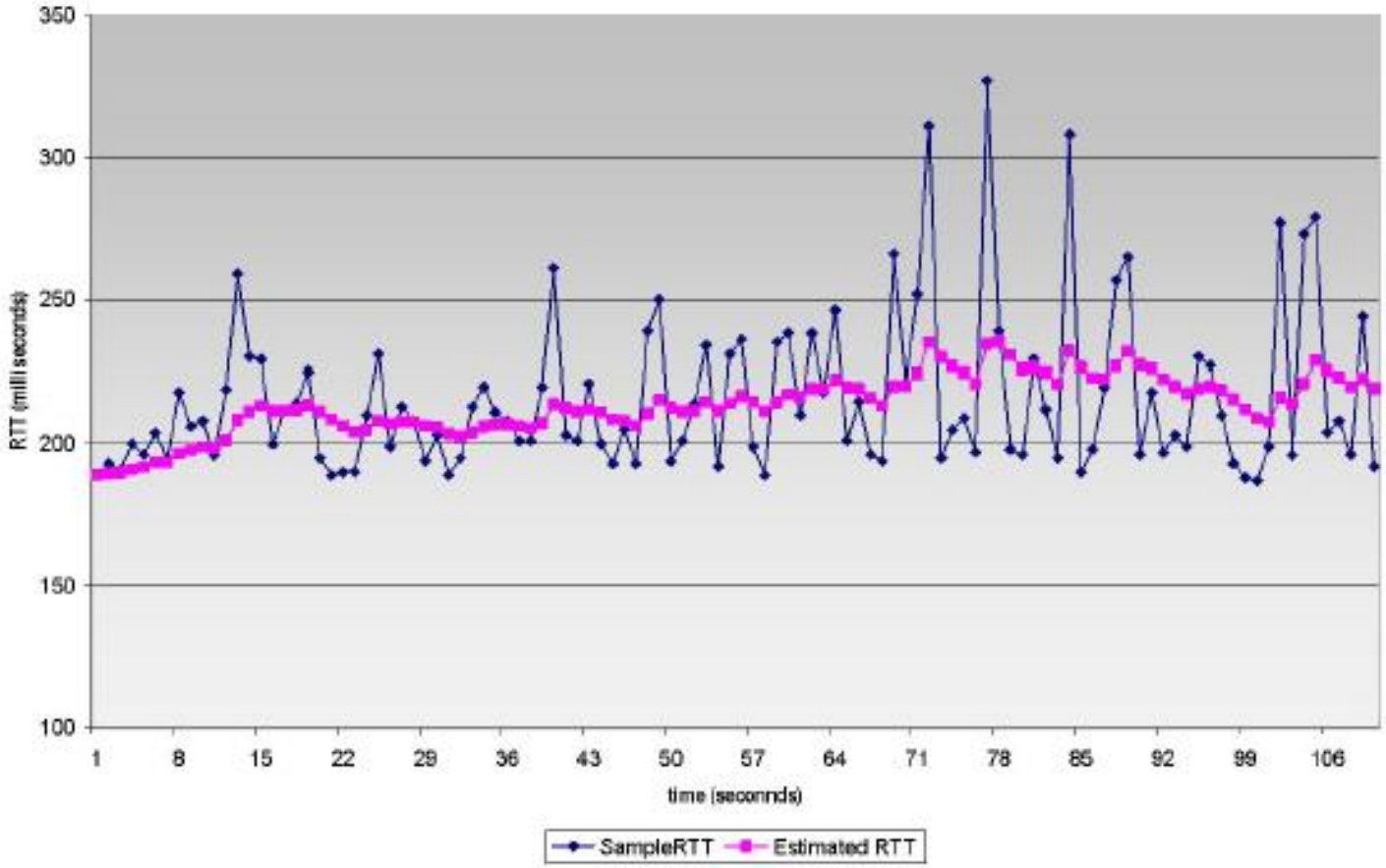
Wie kann RTT geschätzt werden?

- 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

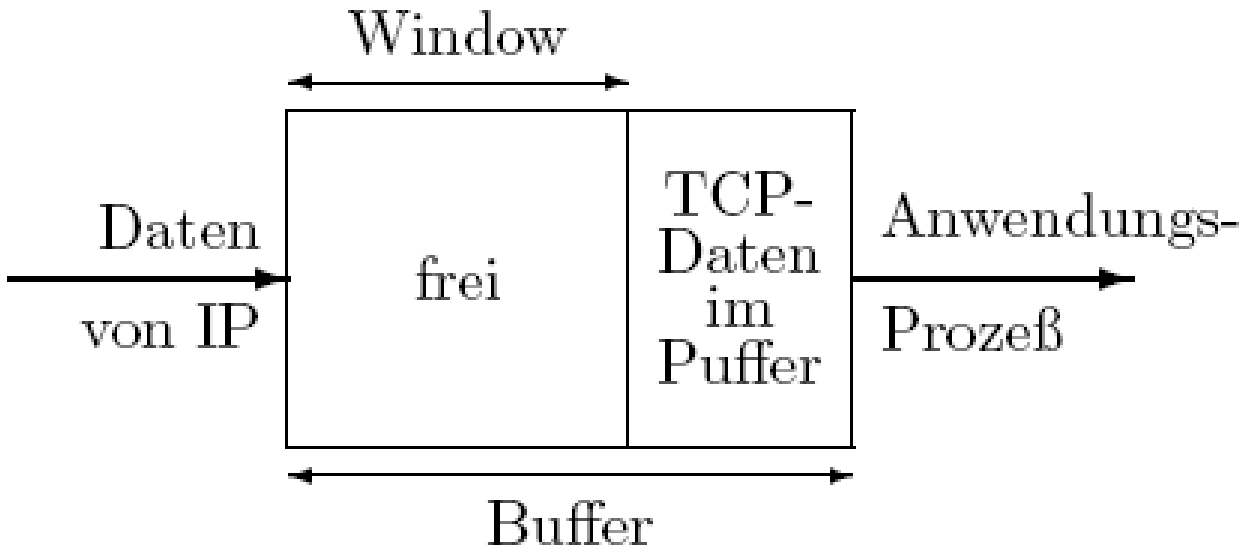


RTT Schätzung:

EstimatedRTT := $(1 - \alpha)$ * EstimatedRTT + α * SampleRTT

- Exponential weighted moving average: *Einfluß alter Messungen fällt exponentiell*
- typischer Wert: $\alpha = 0.125$

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

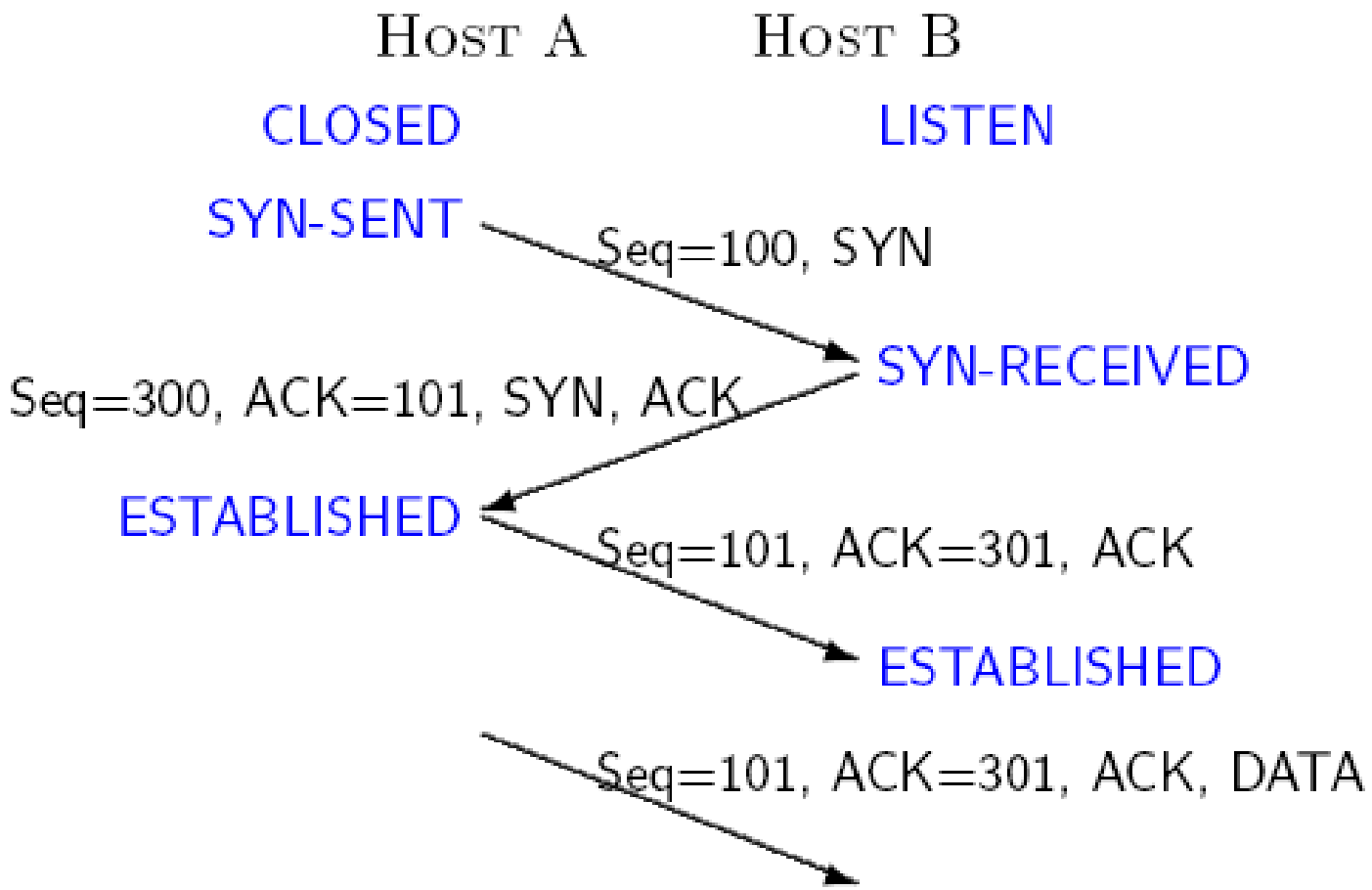
LastByteSent – LastByteAcked ≤ Window

Länge der Daten im Transit

TCP: Verbindungsverwaltung

Öffnen einer Verbindung

über 3-Wege-Handshake



TCP: Verbindungsverwaltung

Schließen einer Verbindung

via Socket.close() bei Client und Server

Schritt 1 Client ruft *Socket.close()*, dies sendet TCP FIN-Kontrollsegment an Server

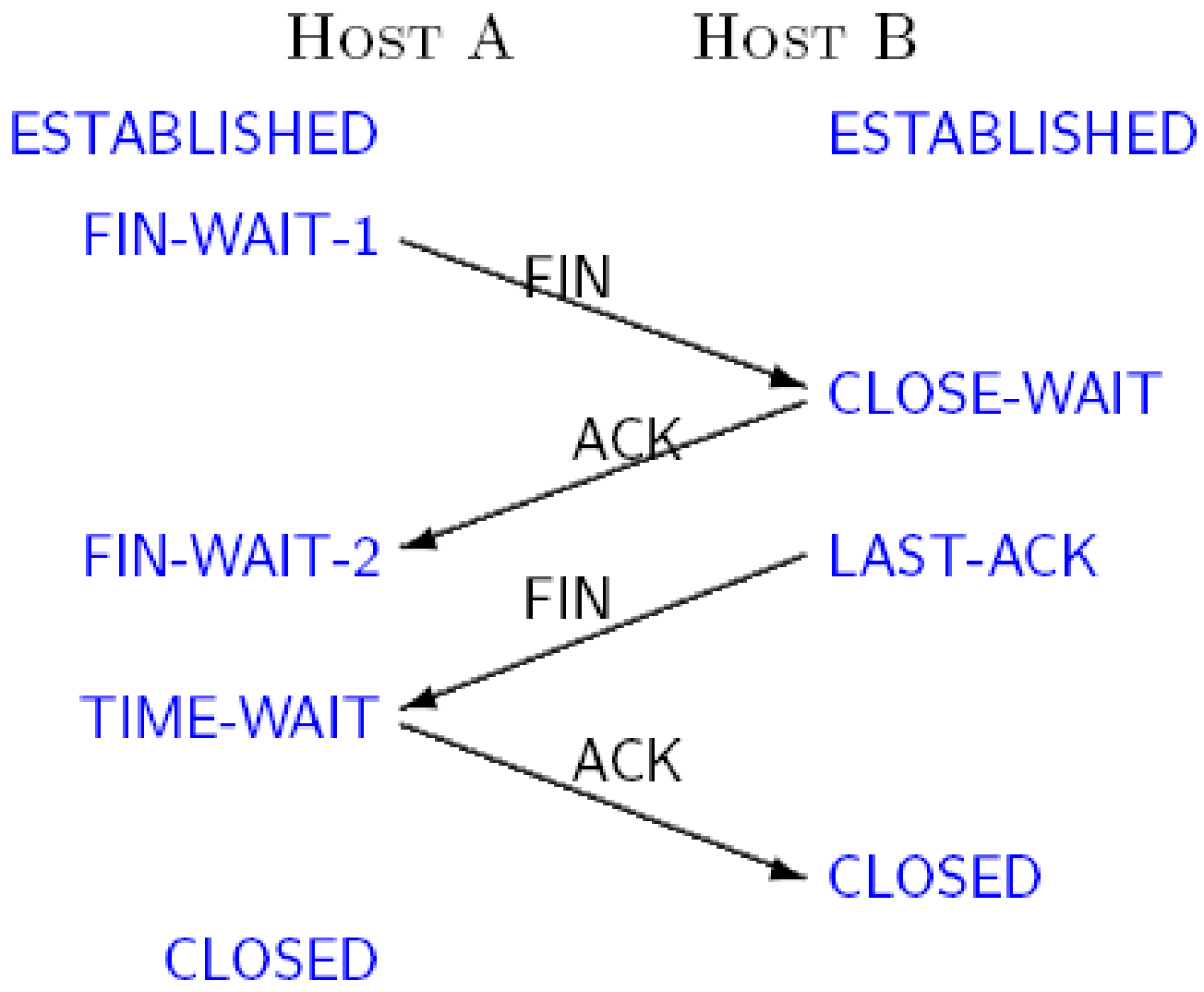
Schritt 2 Server empfängt FIN, sendet ein ACK
Server ruft *Socket.close()*, dies sendet TCP FIN-Kontrollsegment

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

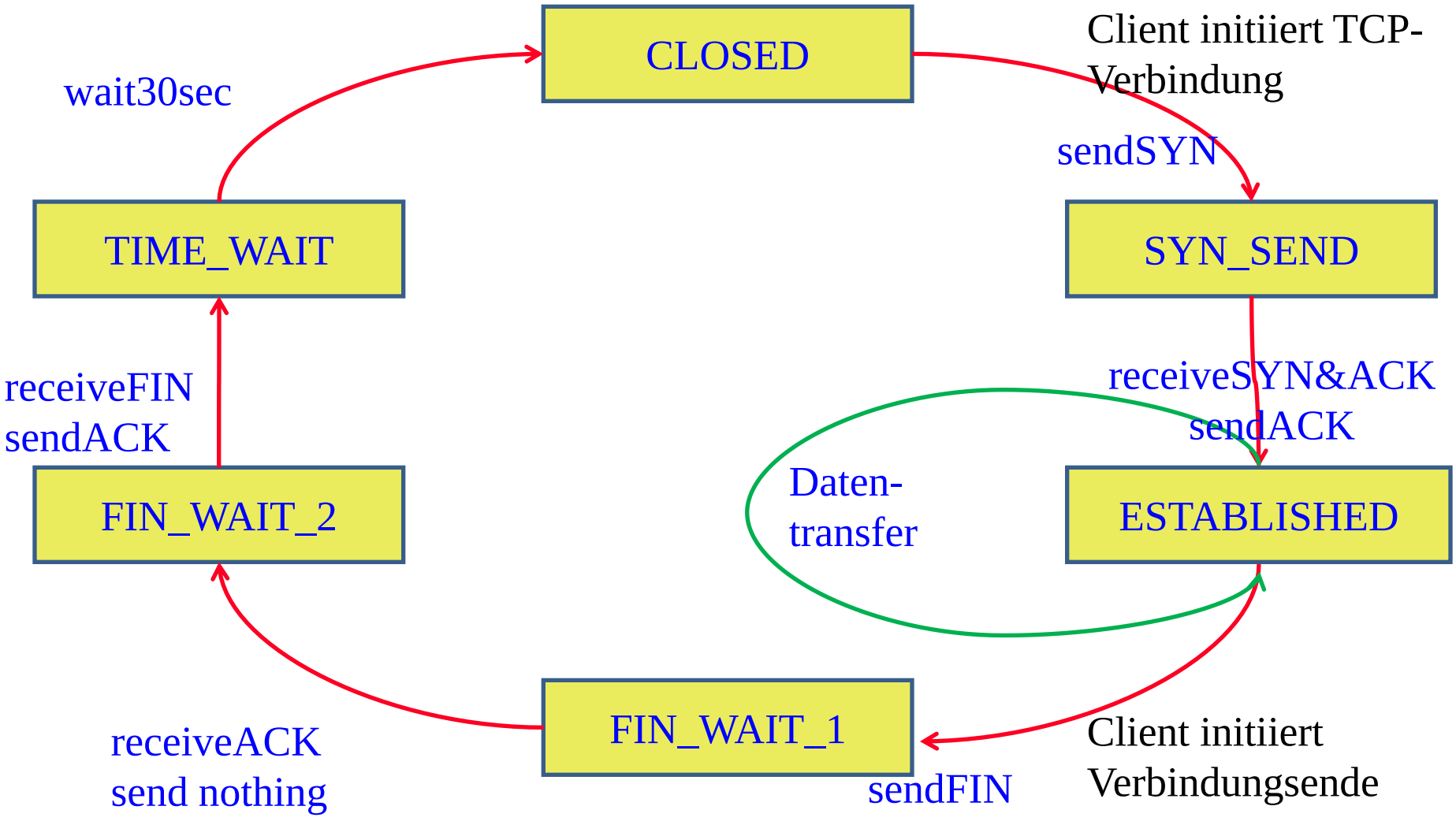
Schritt 4 Server empfängt ACK. Verbindung beendet.

Mit kleinen Änderungen können simultane ACKs verarbeitet werden

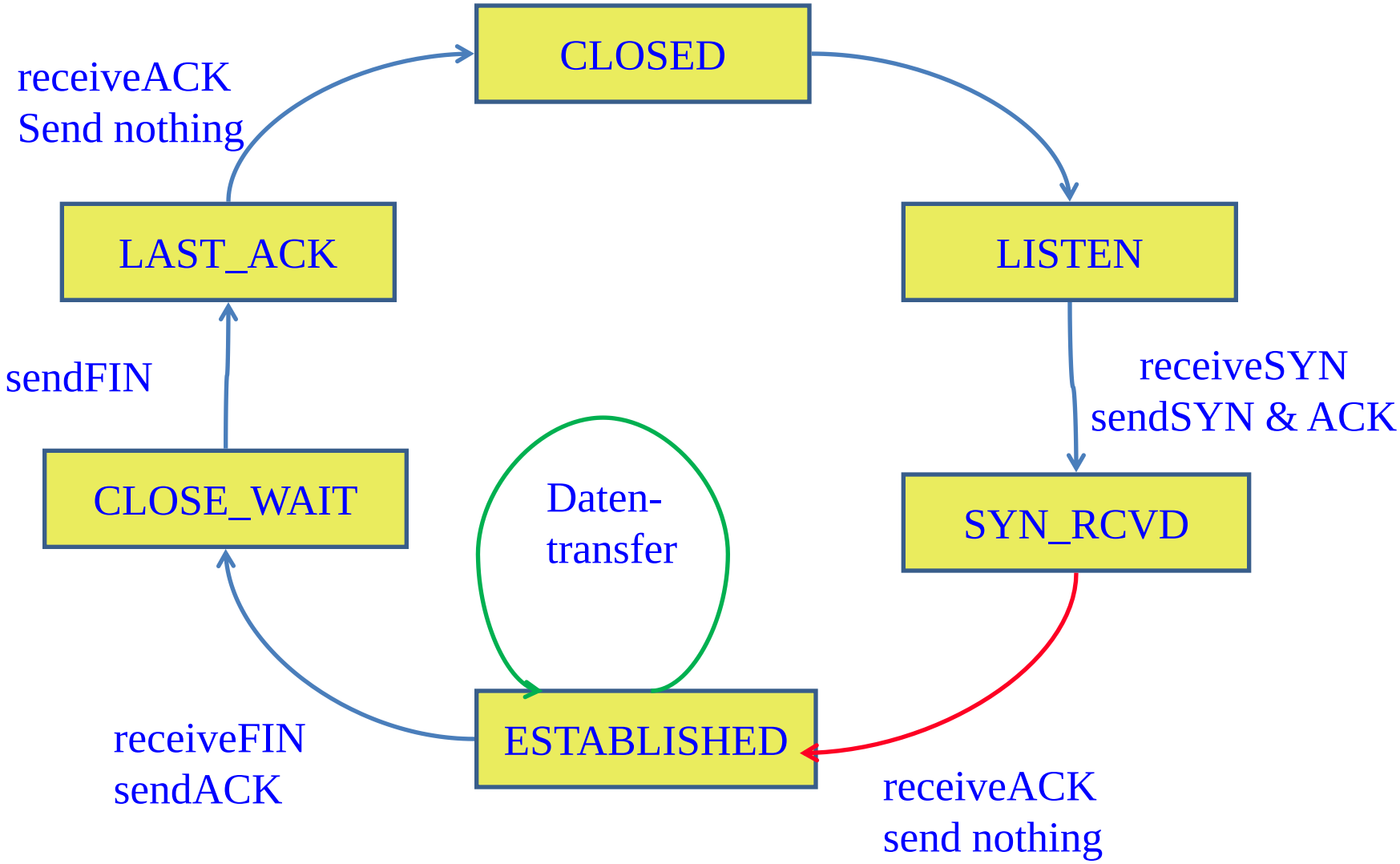
TCP: Verbindungsverwaltung



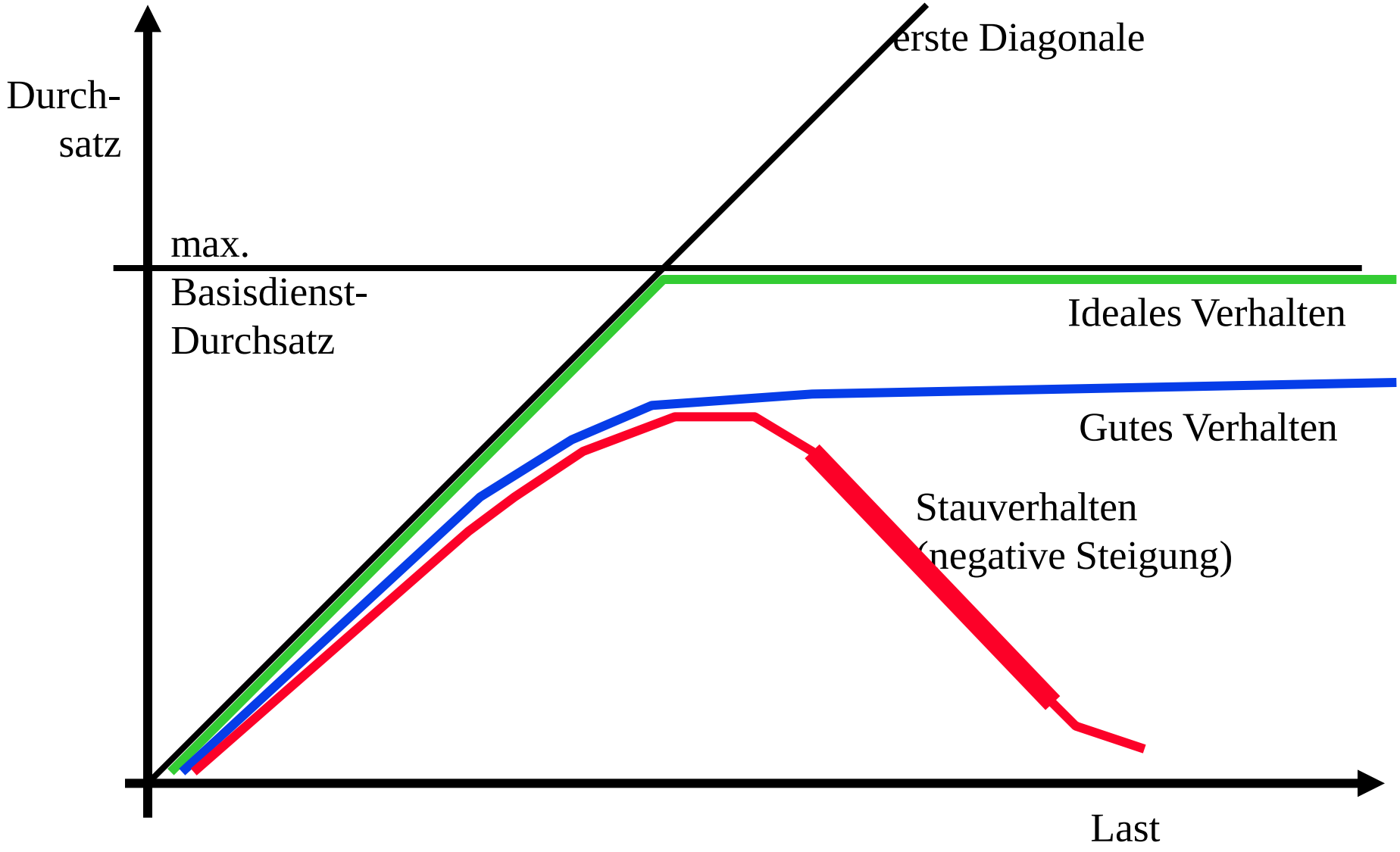
TCP: Client - Ablauf



TCP: Server - Ablauf



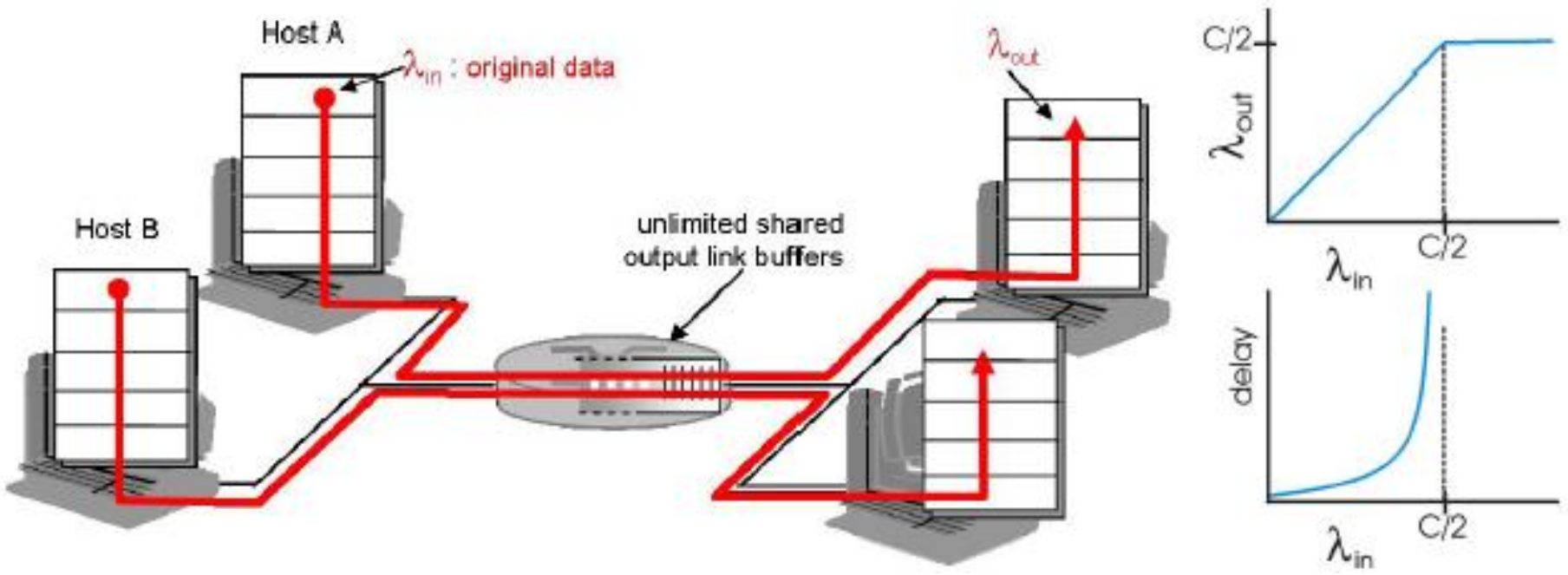
Überlastkontrolle (Staukontrolle) allgemein



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 ...

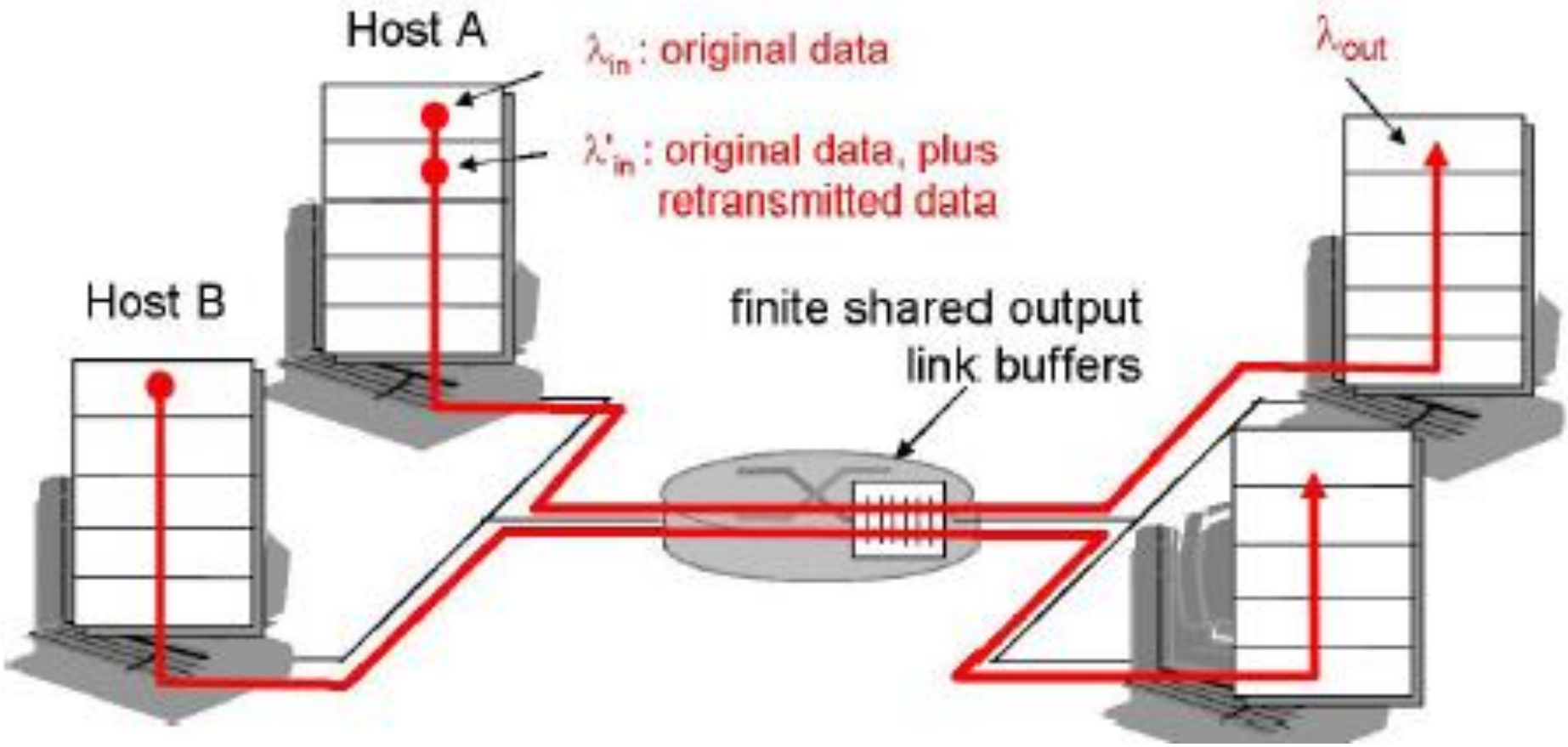
Staus nicht nur auf der B1, sondern auch im Internet



Ursachen für Überlast (Szenario 1 unbeschränkter Puffer)

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

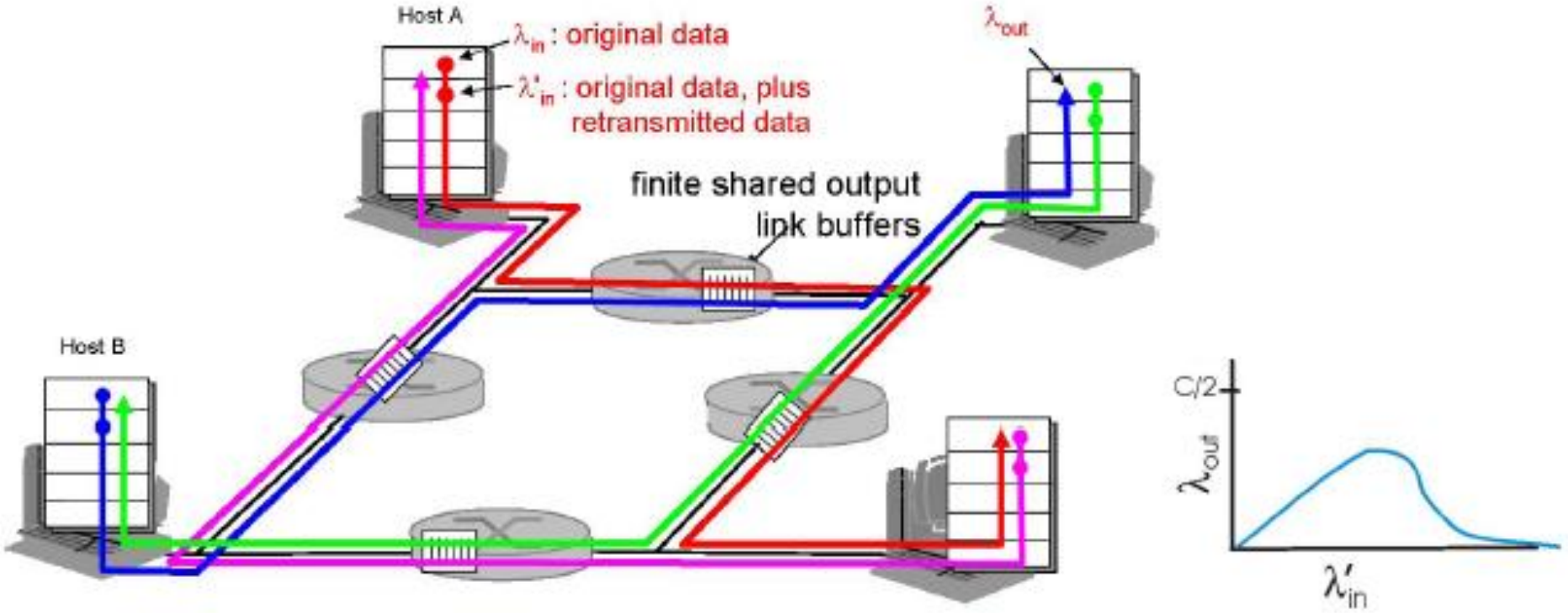
Staus im Internet



Ursachen für Überlast (Szenario 2 beschränkter Puffer)

Segmente gehen durch Pufferüberläufe verloren und werden deshalb wiederholt. Die Wiederholungen erzeugen zusätzliche Last und damit zusätzlichen Verkehr.

Staus im Internet



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 (z.B. bei ATM)

- 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

Anmerkung
Staukontrolle bedingt generell, dass die Lasterzeugung der einzelnen Teilnehmer mit steigender Gesamtlast exponentiell gedrosselt werden muss.

Überlastkontrolle in TCP

Steuerung der Übertragungsrate durch Anpassung der Sendefenstergröße

w = Anzahl übertragener unbestätigter Segmente
= LastByteSent – LastByteAcked

Es muss gelten

$w \leq$ 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

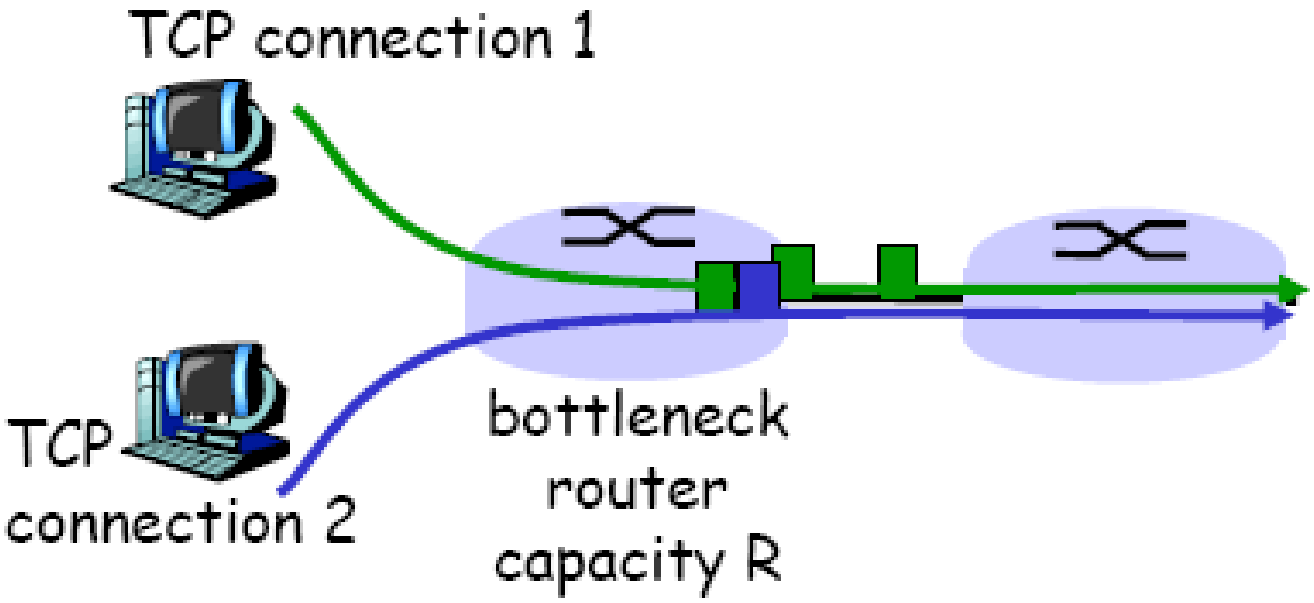
Der TCP-Algorithmus ist darauf angelegt, Überlast möglichst zu vermeiden bzw., wenn das nicht geht, möglichst schnell den Überlastzustand im Netz zu überwinden.

Algorithmus in 3 Schritten

Für jeden Knoten werden zwei Werte **CongWin** und **Threshold** verwaltet.
Anfangs ist **CongWin** = **Maximale Segmentgröße MSS** und **Threshold** ein Mehrfaches davon.
Beginn mit Phase 1.

1. Für gesendeten Segmente gilt: Bei jeder Bestätigung eines Segments, das vor dem Timeout ankommt, wird die Größe von **CongWin** verdoppelt, solange **CongWin ≤ Threshold** (**Slow-Start-Phase**, exponentielles Wachstum von **CongWin**)
   Sobald **CongWin > Threshold** wäre: Weiter mit Phase 2.
2. Bei rechtzeitiger Ankunft eines ACKs wird **CongWin** jeweils nur um 1 **MSS** erhöht (**Überlastvermeidung**, additives Wachstum von **CongWin**).
3. Wenn in Phase 1 oder Phase 2 ein Timeout auftritt, wird der **Threshold** auf **CongWin/2** und **CongWin** auf **1 MSS** gesetzt. Dann wird mit Phase 1 fortgesetzt (**Drosseln** und wieder starten).

TCP - Fairness



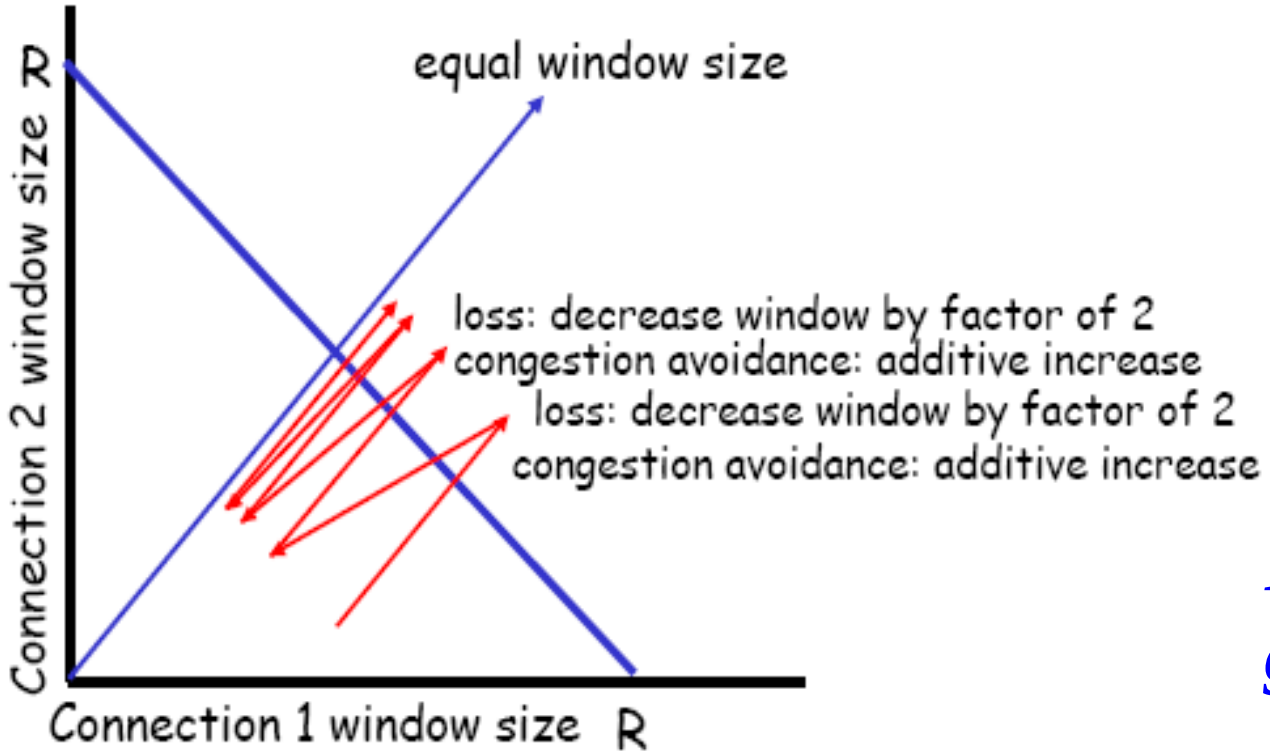
Fairness-Ziel

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

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

Parallele 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

TCP – Nachrichtenverzögerung (Latenz)

Latenz bei der Übertragung eines Datenobjekts über eine eigens dafür aufgebaute TCP-Verbindung

- Zeitraum zwischen Aufbau der TCP-Verbindung bis zum Abbau nach erfolgter Übermittlung des Objekts.
- Anforderung vom Client, Senden vom Server

2 vereinfachende Ansätze

A] statisches Überlastfenster

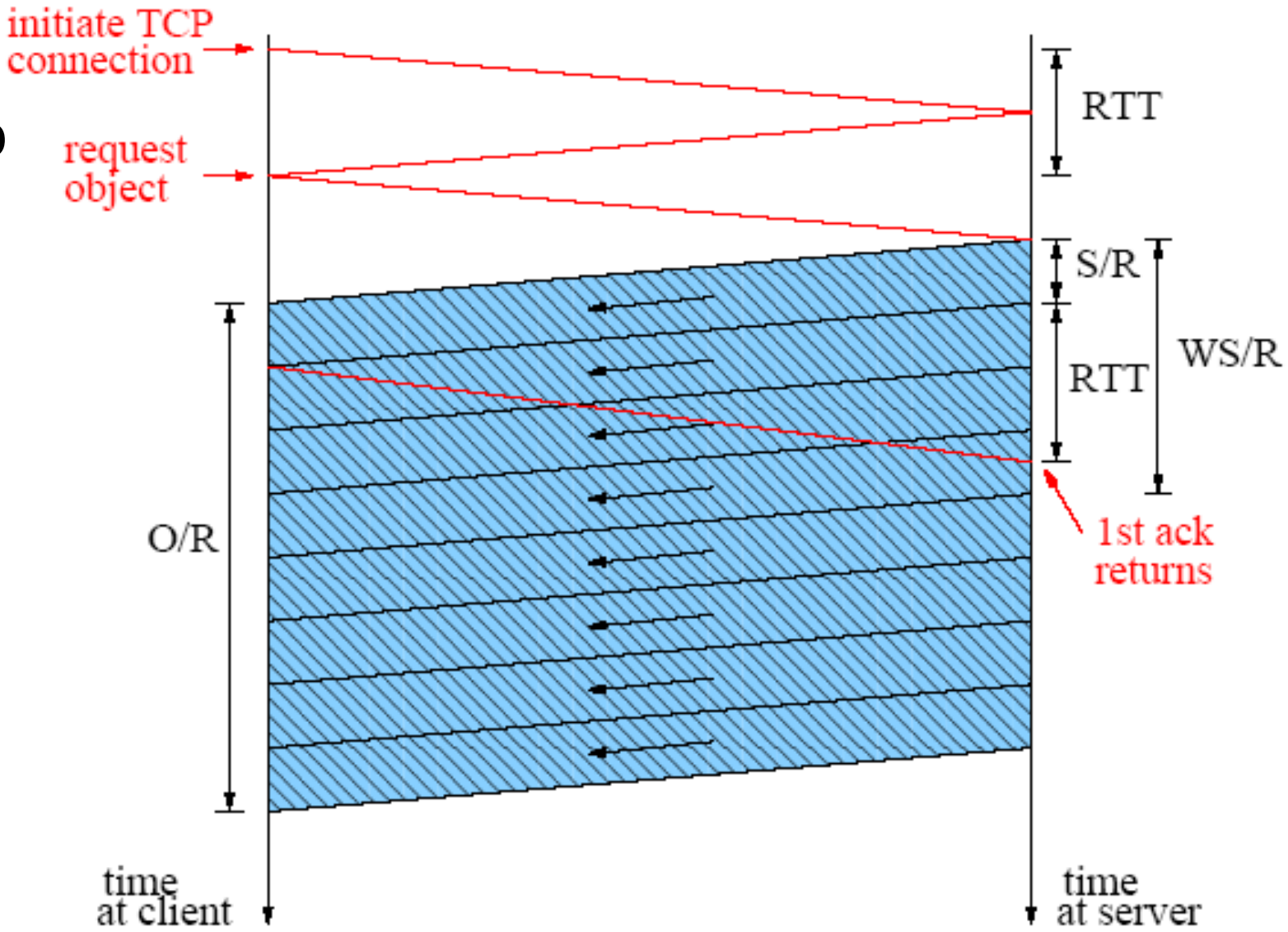
B] dynamisches Überlastfenster

Latenz: A] Statisches Überlastfenster

Objekt der Länge **O** soll bei Bitrate **R** in Segmenten der Länge **S** übertragen werden

Fall 1
großes Fenster

Weil das ACK immer rechtzeitig kommt, kann Sender ohne Unterbrechung senden.

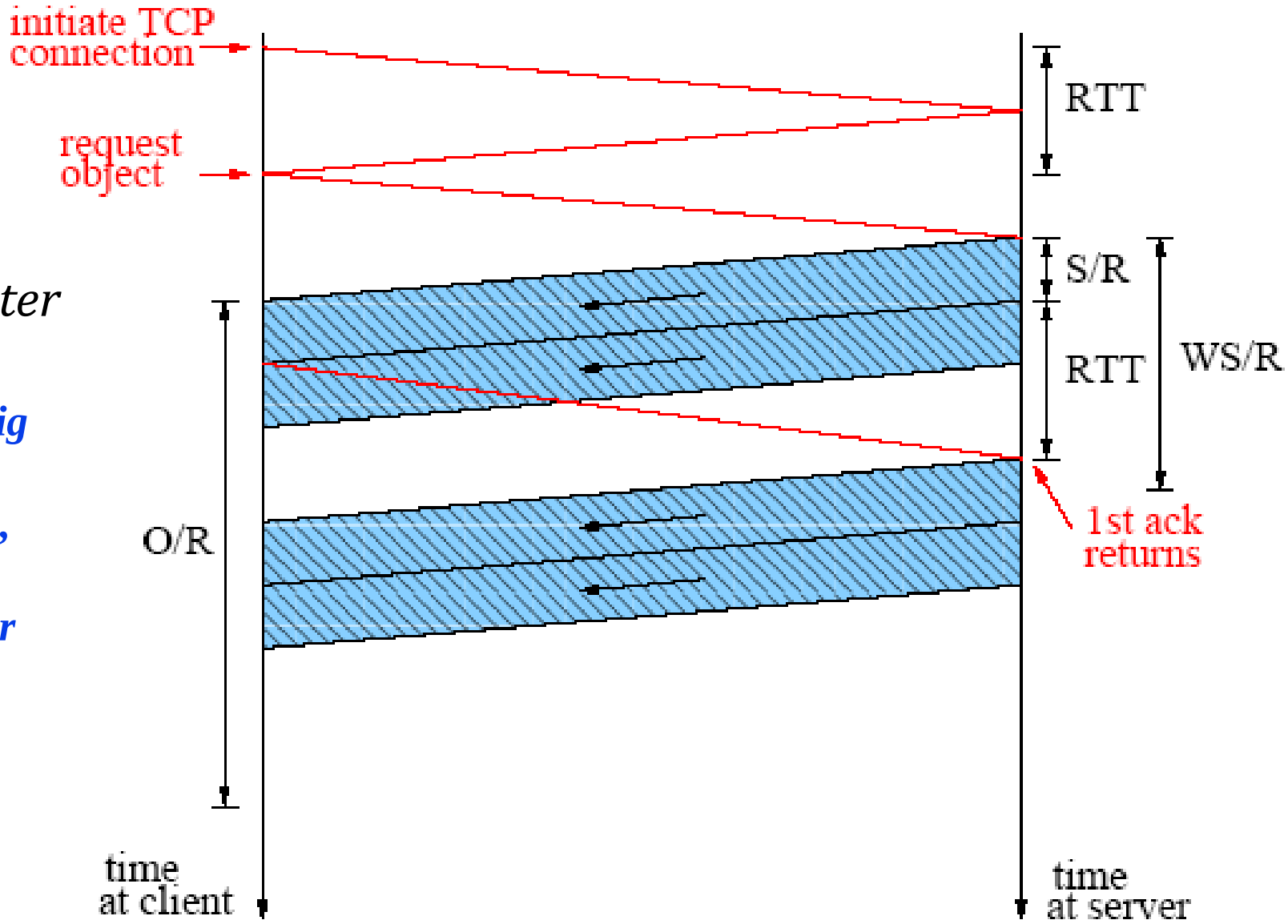


Latenz: A] Statisches Überlastfenster

Fall 2

kleines Fenster

Weil das ACK nicht rechtzeitig kommt, muss Sender warten, wenn das Überlastfenster erschöpft ist.



Latenz: B] Dynamisches Überlastfenster

Annahme
Threshold so groß, dass Protokoll im Slow-Start bleibt.

Fall 2
zunächst kleines Fenster

Nach jedem Warten kann, weil das Fenster wächst, mehr gesendet werden, bevor der Sender wieder auf ACK warten muss.