Netzwerktechnologien
3 VO

Dr. Ivan Gojmerac
ivan.gojmerac@univie.ac.at

5. Vorlesungseinheit, 17. April 2013

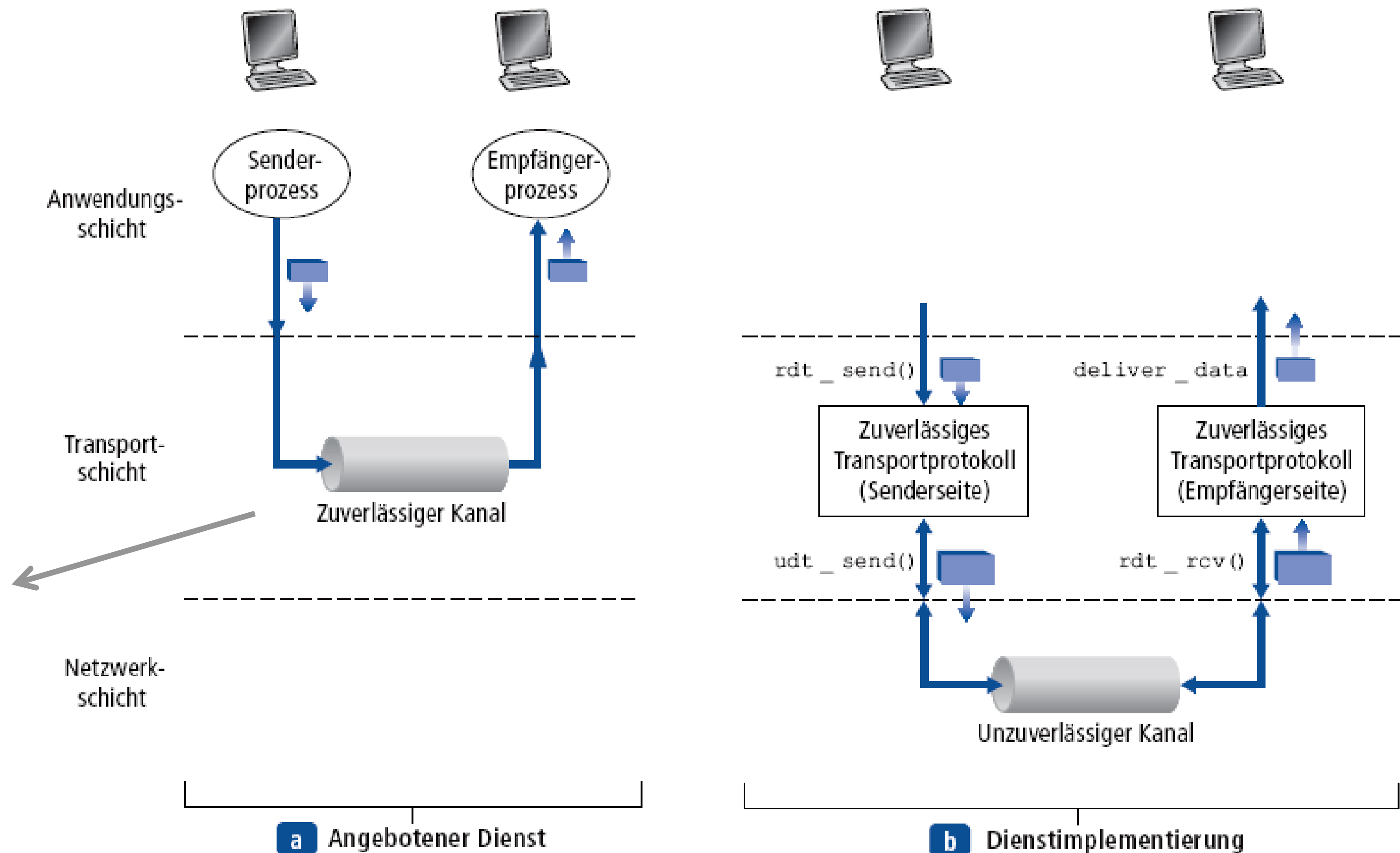
Bachelorstudium Medieninformatik
SS 2013

3.4 Zuverlässigkeit der Datenübertragung

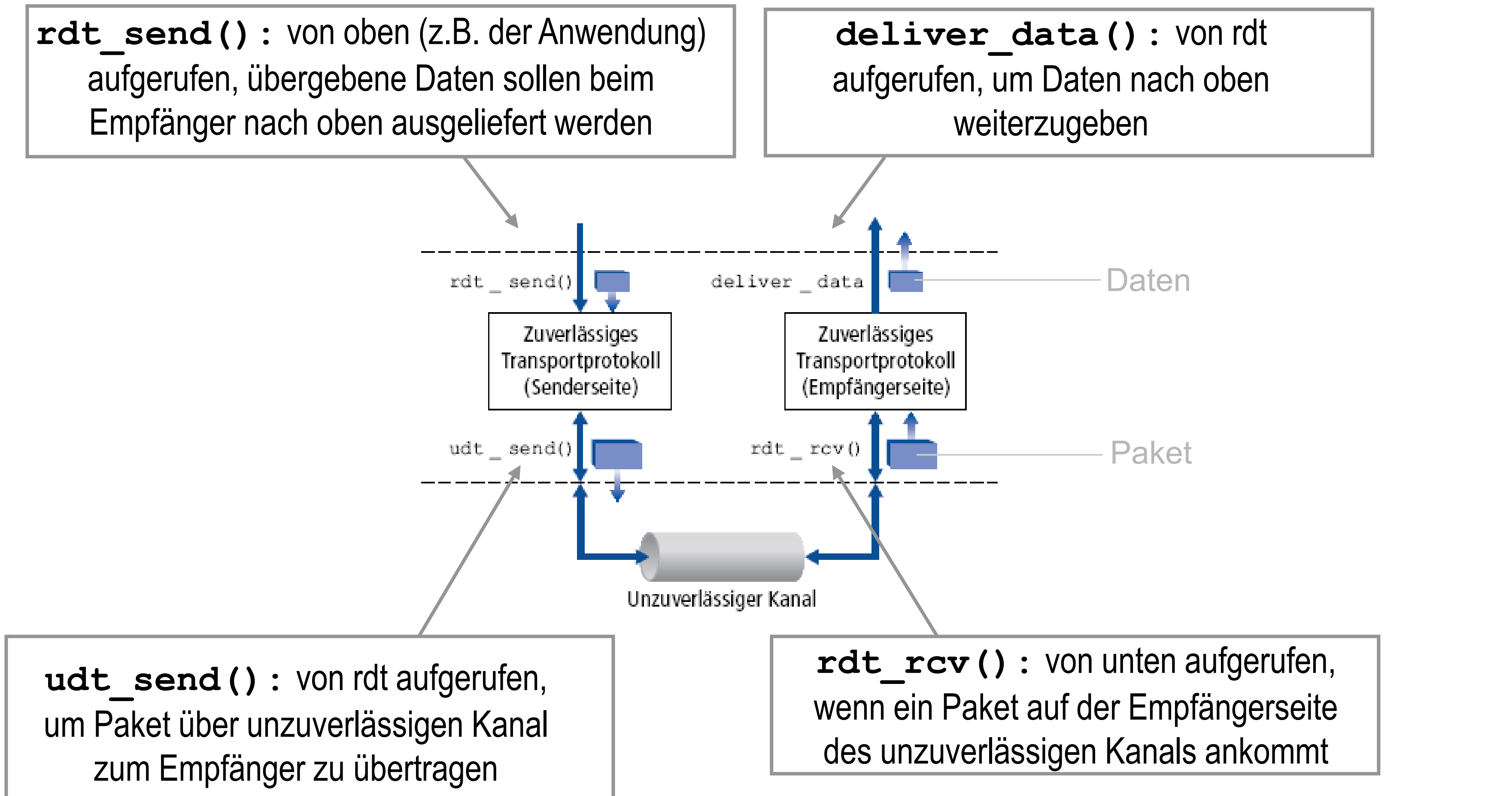
3.4 Zuverlässigkeit der Datenübertragung

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

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



3.4 Grundlagen der zuverlässigen Datenübertragung



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

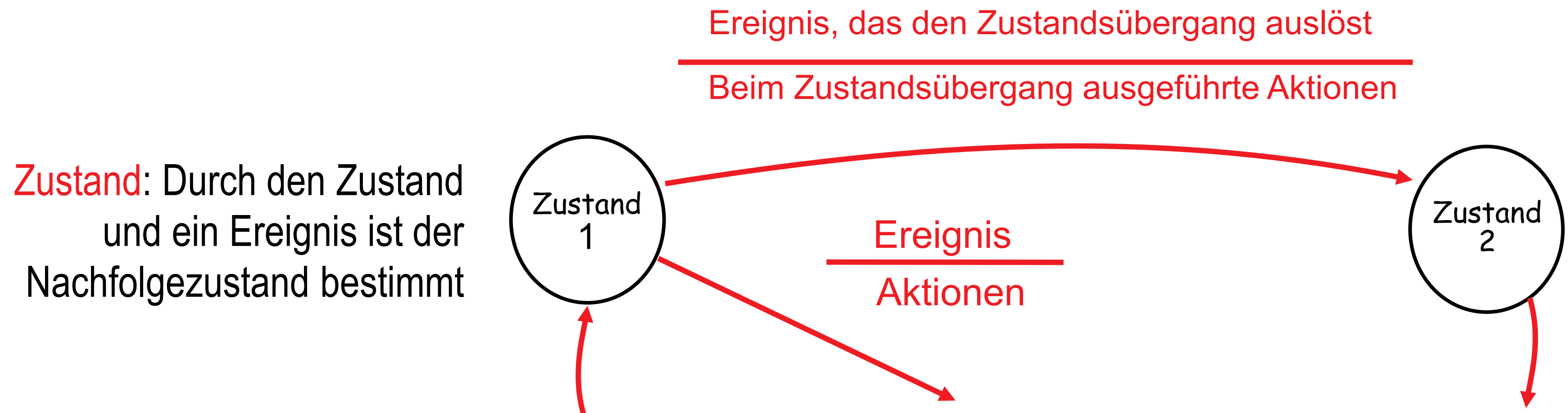
3.4.1 Aufbau eines zuverlässigen Datentransferprotokolls

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

WICHTIGE ANMERKUNG: Die Folien 7 bis 20 enthalten zu didaktischen Zwecken erfundene (d.h. fiktive) **Reliable-Data-Transfer**-Protokolle **rdt1.0** bis **rdt3.0**.

3.4.1 Endliche Automaten

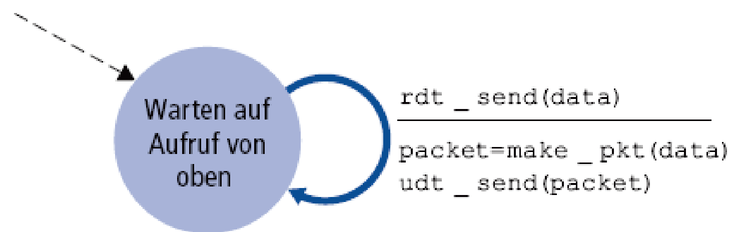
- Finite state machine = FSM
- Um Sender und Empfänger zu spezifizieren



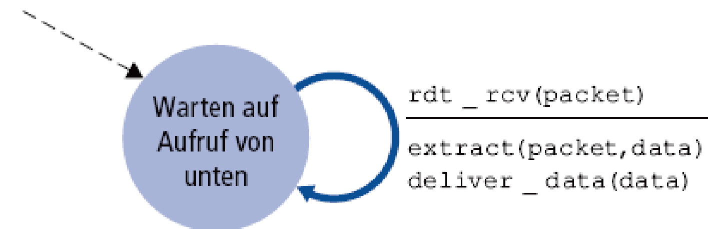
3.4.1 Aufbau eines zuverlässigen Datentransferprotokolls

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

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



a rdt1.0-Sender

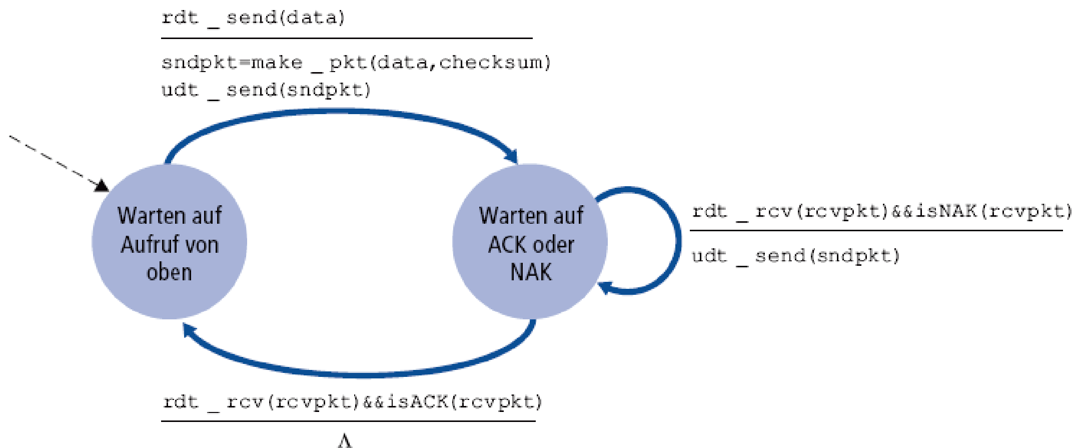


b rdt1.0-Empfänger

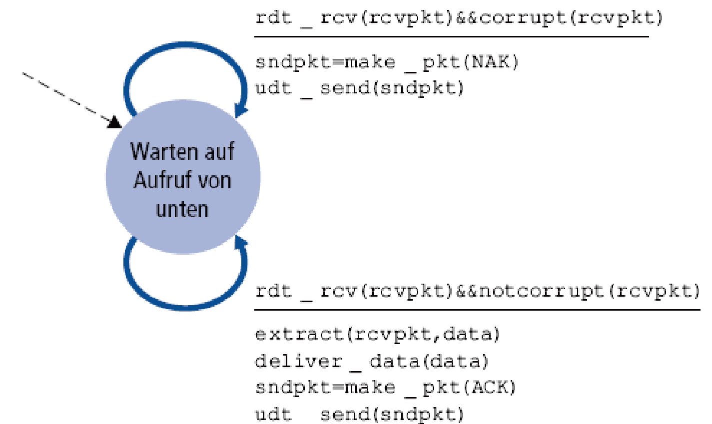
3.4.1 Aufbau eines zuverlässigen Datentransferprotokolls

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

- Verfälschte Bits sind durch eine Prüfsumme erkennbar
- Mechanismen zur zuverlässigen Datenübertragung:
  - *Acknowledgements (ACKs):*
    Empfänger sagt dem Sender explizit, dass das Paket erfolgreich empfangen wurde.
  - *Negative Acknowledgements (NAKs):*
    Empfänger sagt dem Sender explizit, dass das Paket fehlerbehaftet war. Sender wiederholt Übertragung für diese Pakete.



a rdt2.0-Sender



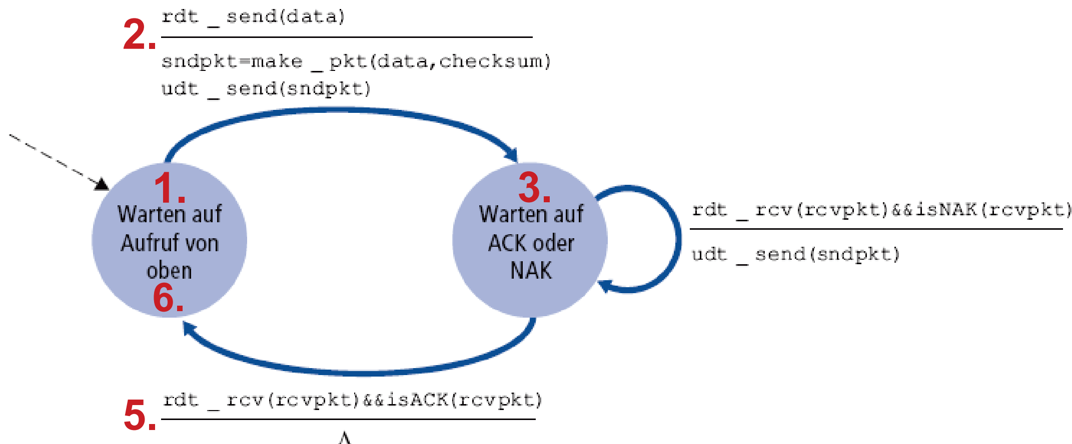
b rdt2.0-Empfänger

3.4.1 Aufbau eines zuverlässigen Datentransferprotokolls

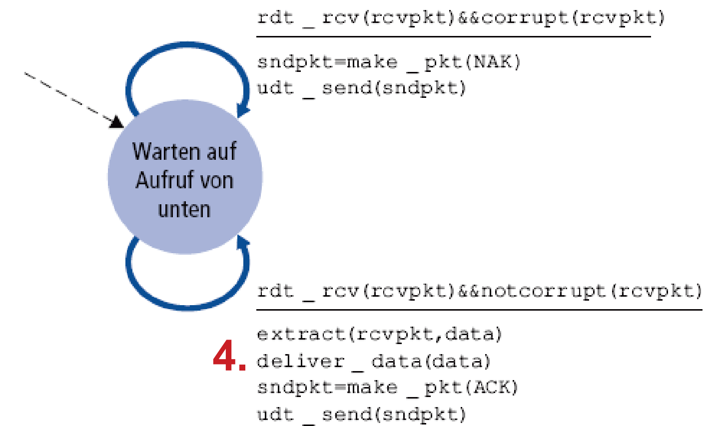
Zuverlässiger Datentransfer über einen Kanal mit Bitfehlern: **rdt2.0**

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

Ablauf **ohne** Fehler:



a rdt2.0-Sender



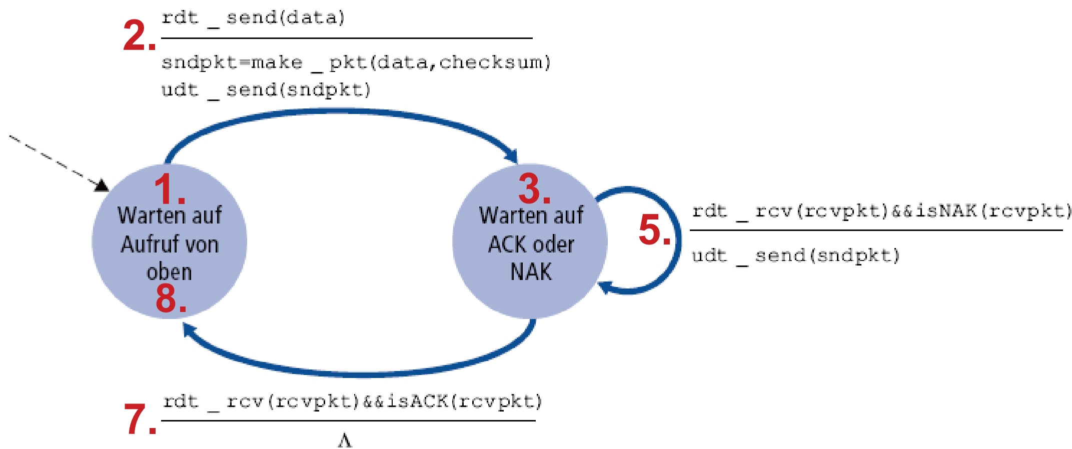
b rdt2.0-Empfänger

3.4.1 Aufbau eines zuverlässigen Datentransferprotokolls

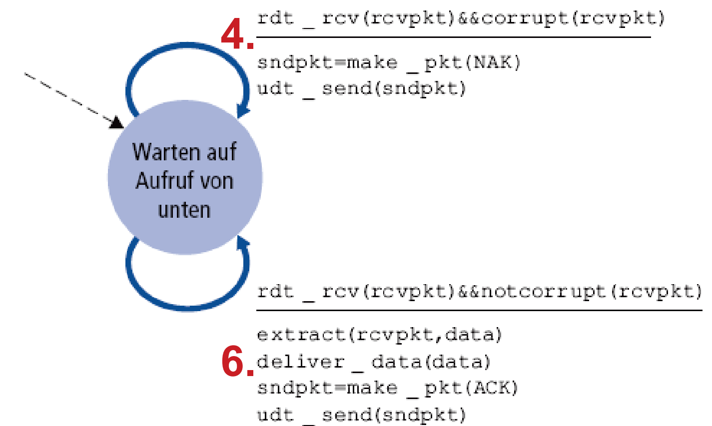
Zuverlässiger Datentransfer über einen Kanal mit Bitfehlern: **rdt2.0**

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

Ablauf **mit** Fehler:



a rdt2.0-Sender



b rdt2.0-Empfänger

3.4.1 Aufbau eines zuverlässigen Datentransferprotokolls

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

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

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

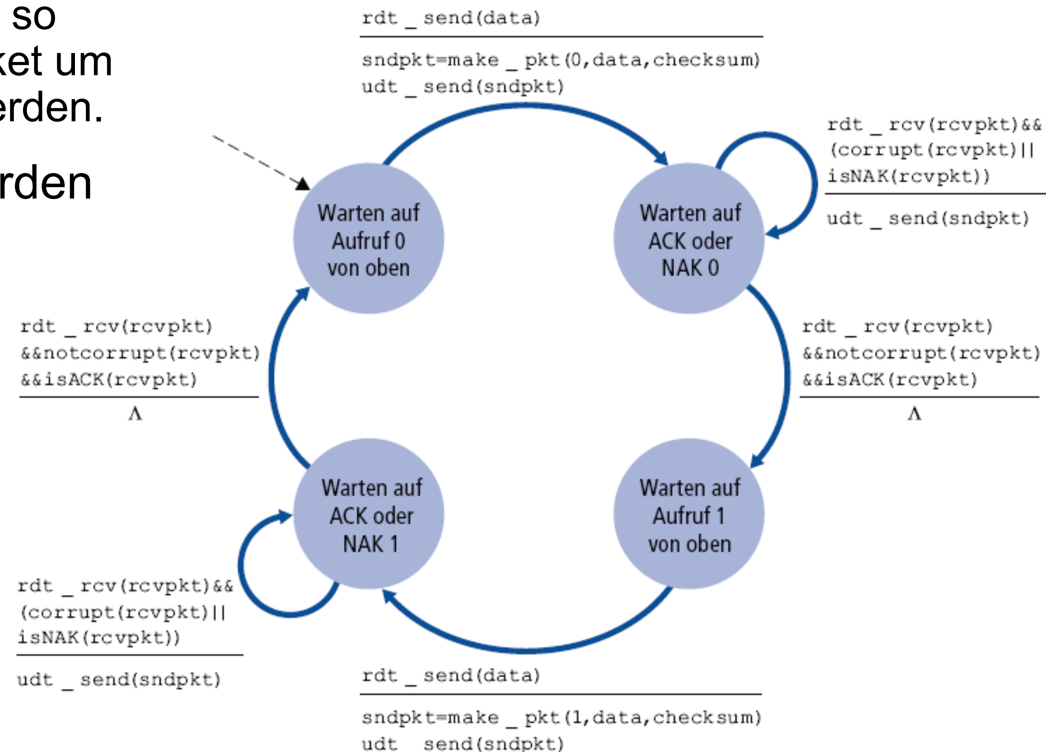
- Verbreiteter Lösungsansatz:
  Übertragungswiederholung *und* **fortlaufende Sequenznummer** in Datenpakete einfügen. → Bei Stop-and-Wait-Protokollen genügt eine 1-Bit-Zahl zum Vergleich der Sequenznummer mit der des zuletzt empfangenen Paketes.

3.4.1 Aufbau eines zuverlässigen Datentransferprotokolls

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

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

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

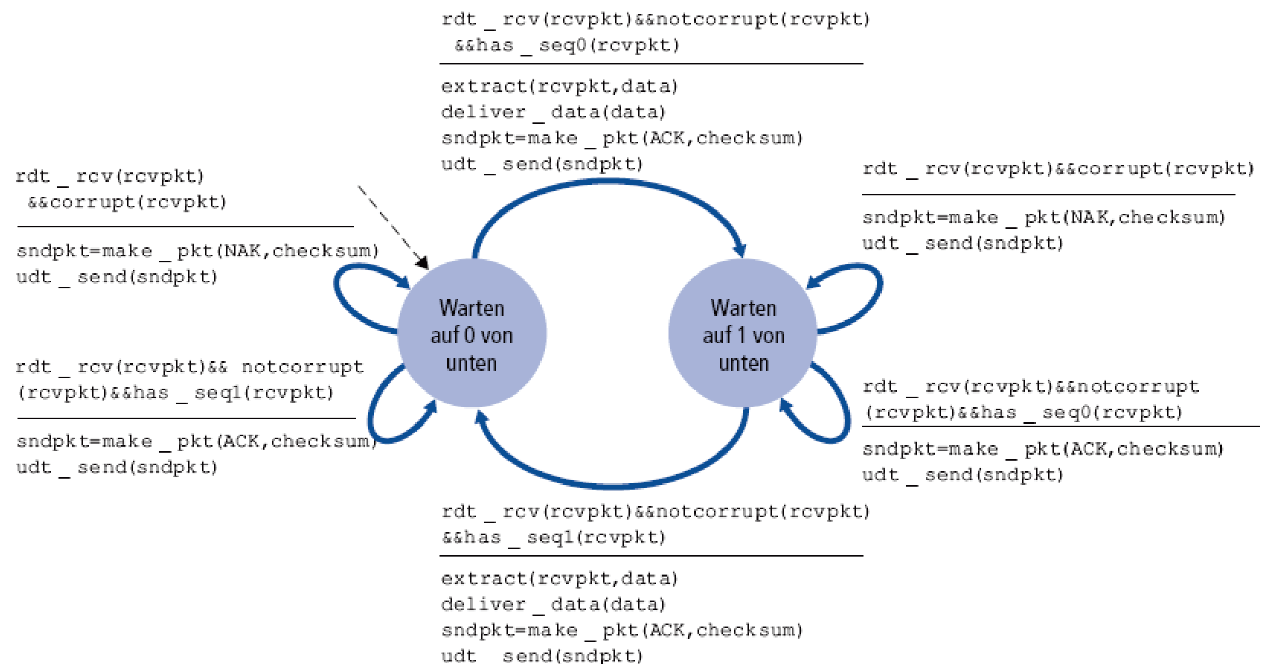


3.4.1 Aufbau eines zuverlässigen Datentransferprotokolls

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

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

→ Lässt sich erst am nächsten empfangenen Datenpaket erkennen



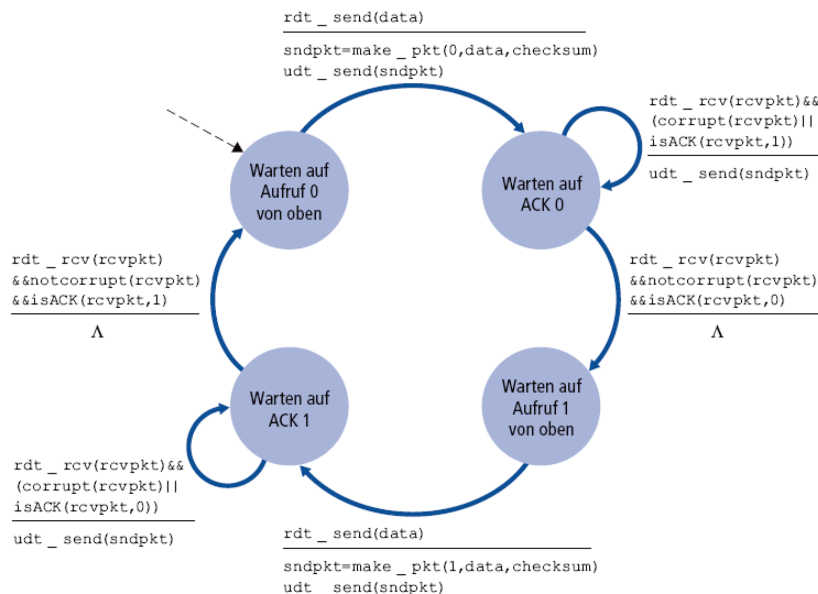
3.4.1 Aufbau eines zuverlässigen Datentransferprotokolls

rdt2.2: Elimination von NAKs

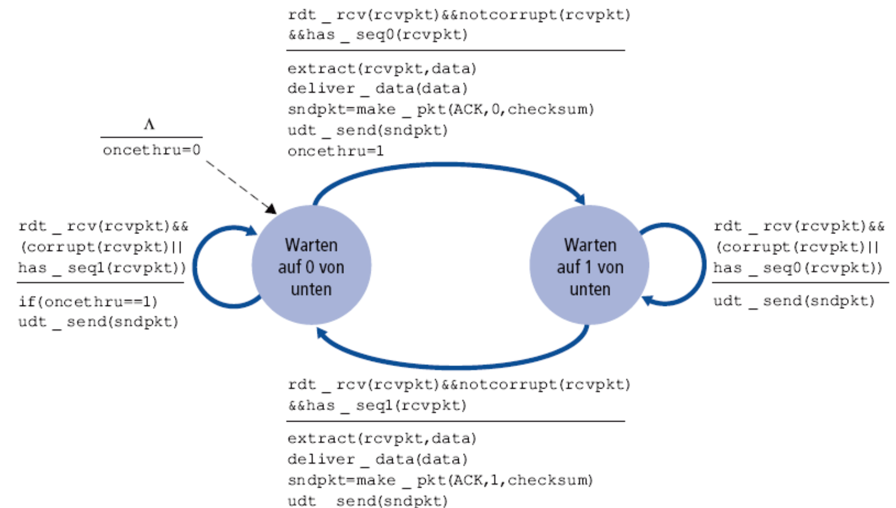
- Anstelle von NAKs: ACK für das letzte korrekt empfangene Paket

→ Empfänger muss die Sequenznummer des bestätigten Paketes im ACK mitschicken

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



a rdt2.2-Sender



rdt2.2-Empfänger

3.4.1 Aufbau eines zuverlässigen Datentransferprotokolls

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

- Der Kanal kann nun auch ganze Pakete verlieren.

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

Problem: Wie wird der Verlust von Paketen behandelt?

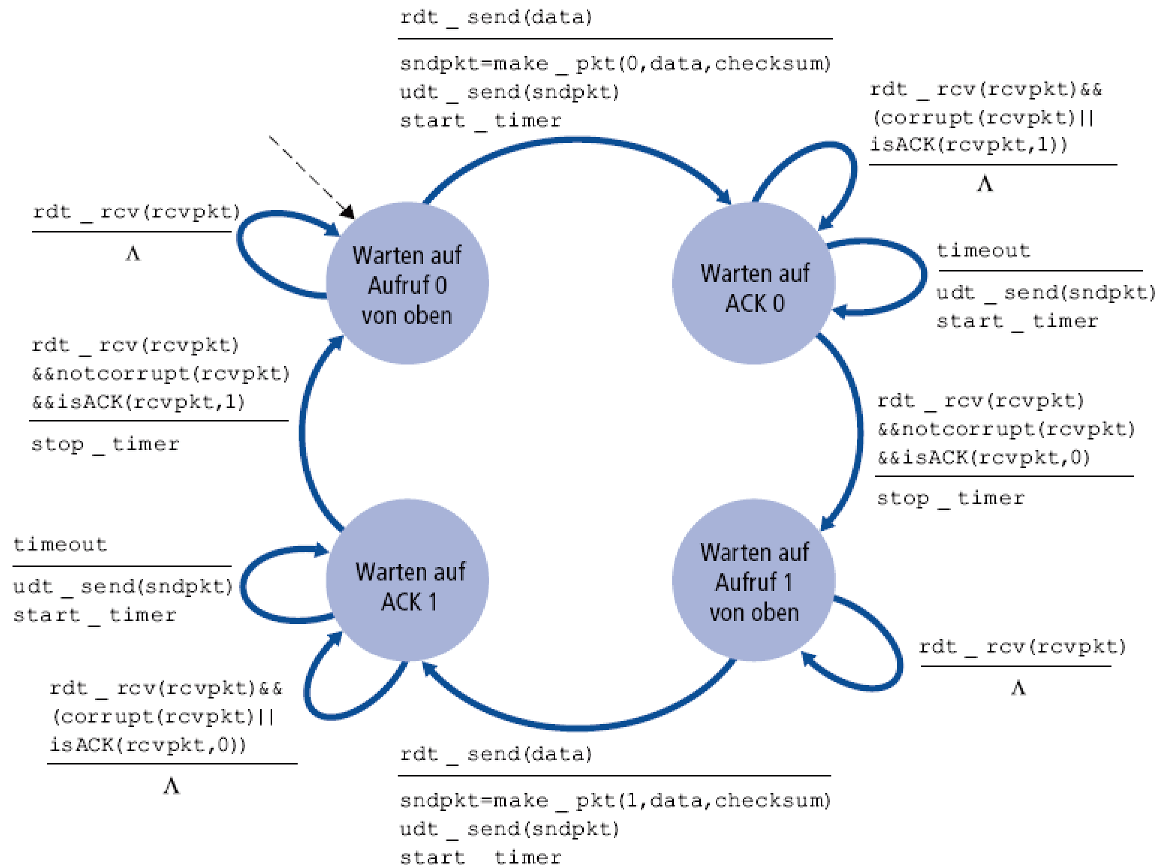
Lösung: **Sender wartet eine „vernünftige Zeitspanne“ auf ein ACK**

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

→ Erfordert **Timer zum Stoppen der Zeit**

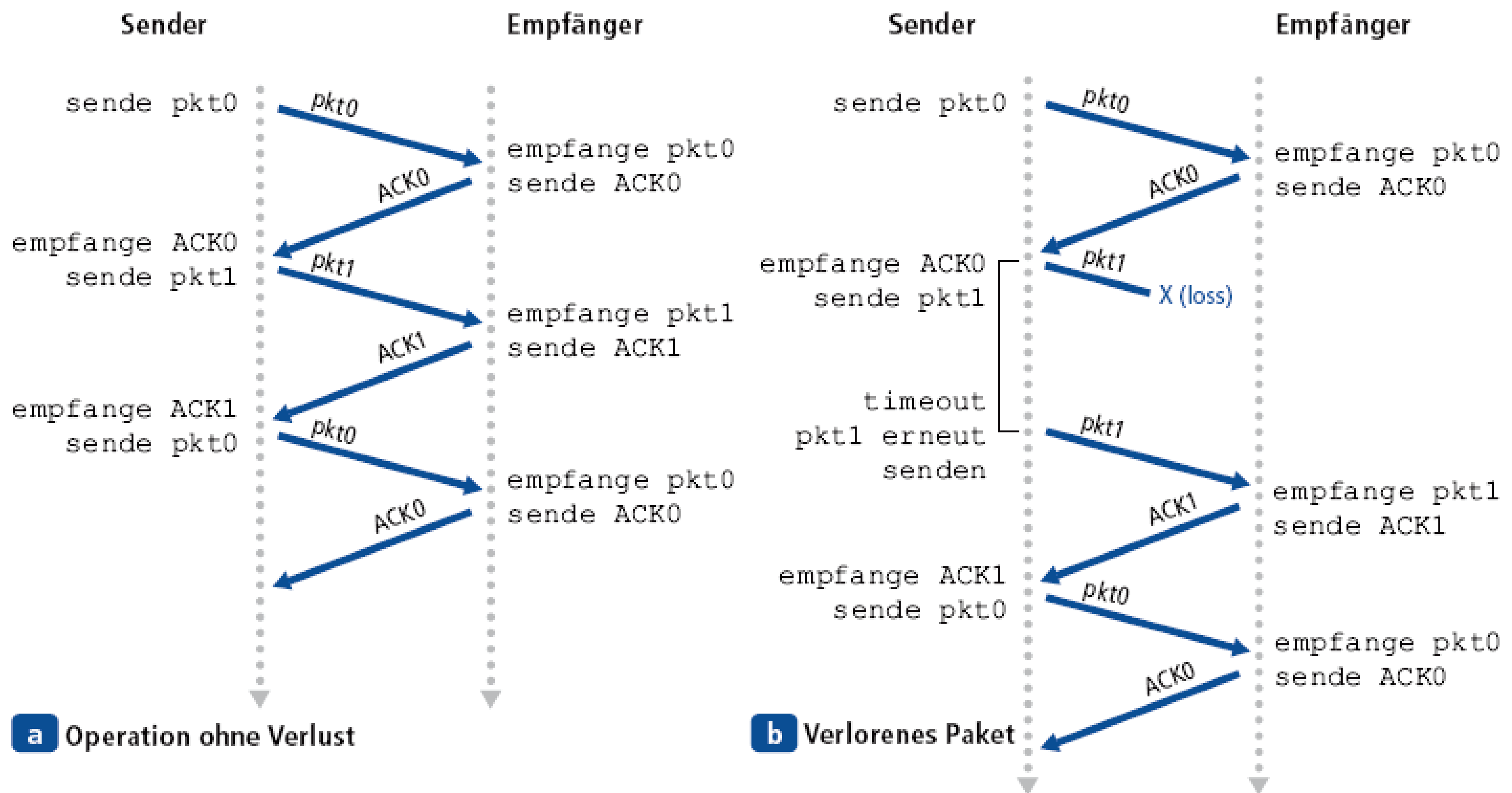
3.4.1 Aufbau eines zuverlässigen Datentransferprotokolls

rdt3.0: Sender mit Wartezeiten



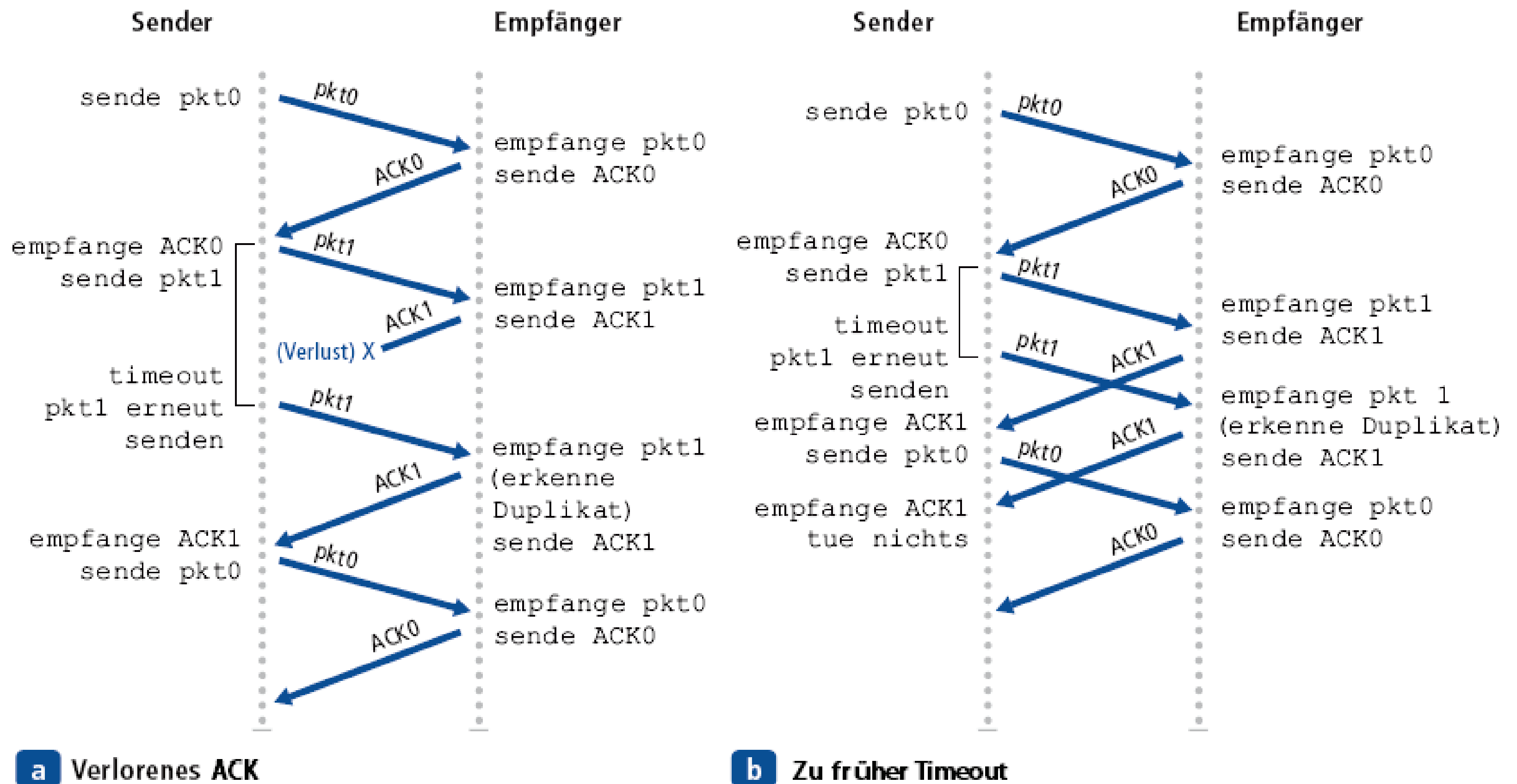
3.4.1 Aufbau eines zuverlässigen Datentransferprotokolls

rdt3.0: Arbeitsweise



3.4.1 Aufbau eines zuverlässigen Datentransferprotokolls

rdt3.0: Arbeitsweise



3.4.1 Aufbau eines zuverlässigen Datentransferprotokolls

rdt3.0: Performance

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

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

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

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

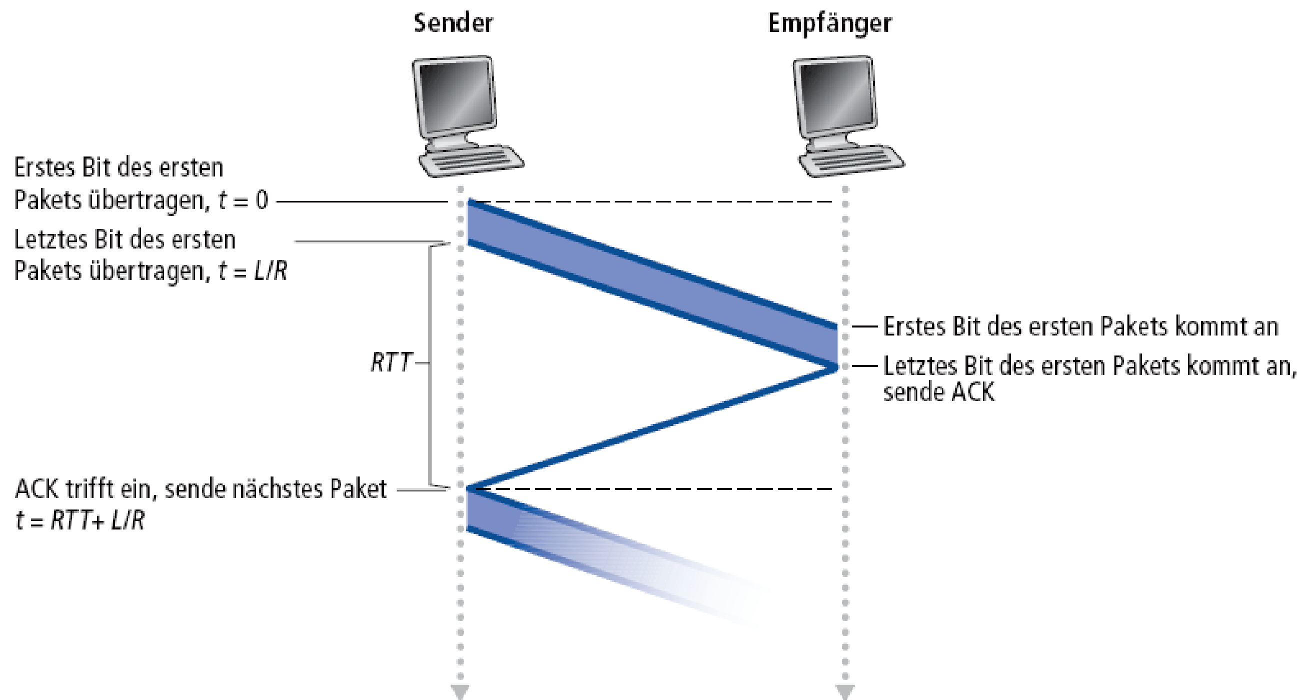
U_{sender}: Utilization (engl. für Auslastung) – Anteil der Zeit, in der tatsächlich gesendet wird

Einmal 8000 Bit alle ~30 ms → ~ 270 kbit/s Durchsatz über einen Link mit 1 Gbit/s

→ Das Protokoll beschränkt die Ausnutzung physikalischer Ressourcen!

3.4.1 Aufbau eines zuverlässigen Datentransferprotokolls

rdt3.0: Stop-and-Wait-Ablauf

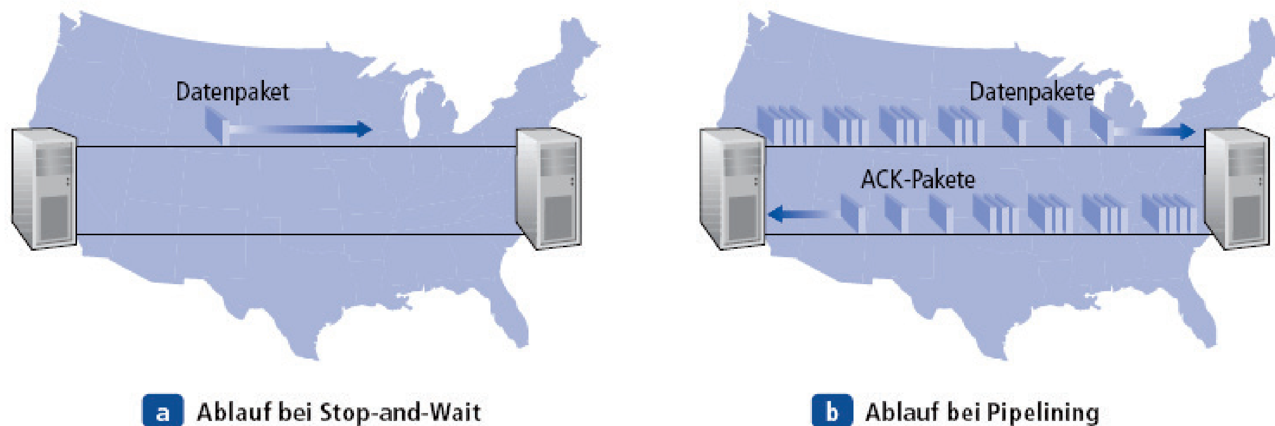


a Ablauf bei Stop-and-Wait

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

3.4.2 Zuverlässige Datentransferprotokolle mit Pipelining

Der Kern des Leistungsproblems mit rdt3.0 liegt darin, daß es ein Stop-and-Wait-Protokoll ist. Stop-and-Wait im Vergleich zu Pipelining:



a Ablauf bei Stop-and-Wait

b Ablauf bei Pipelining

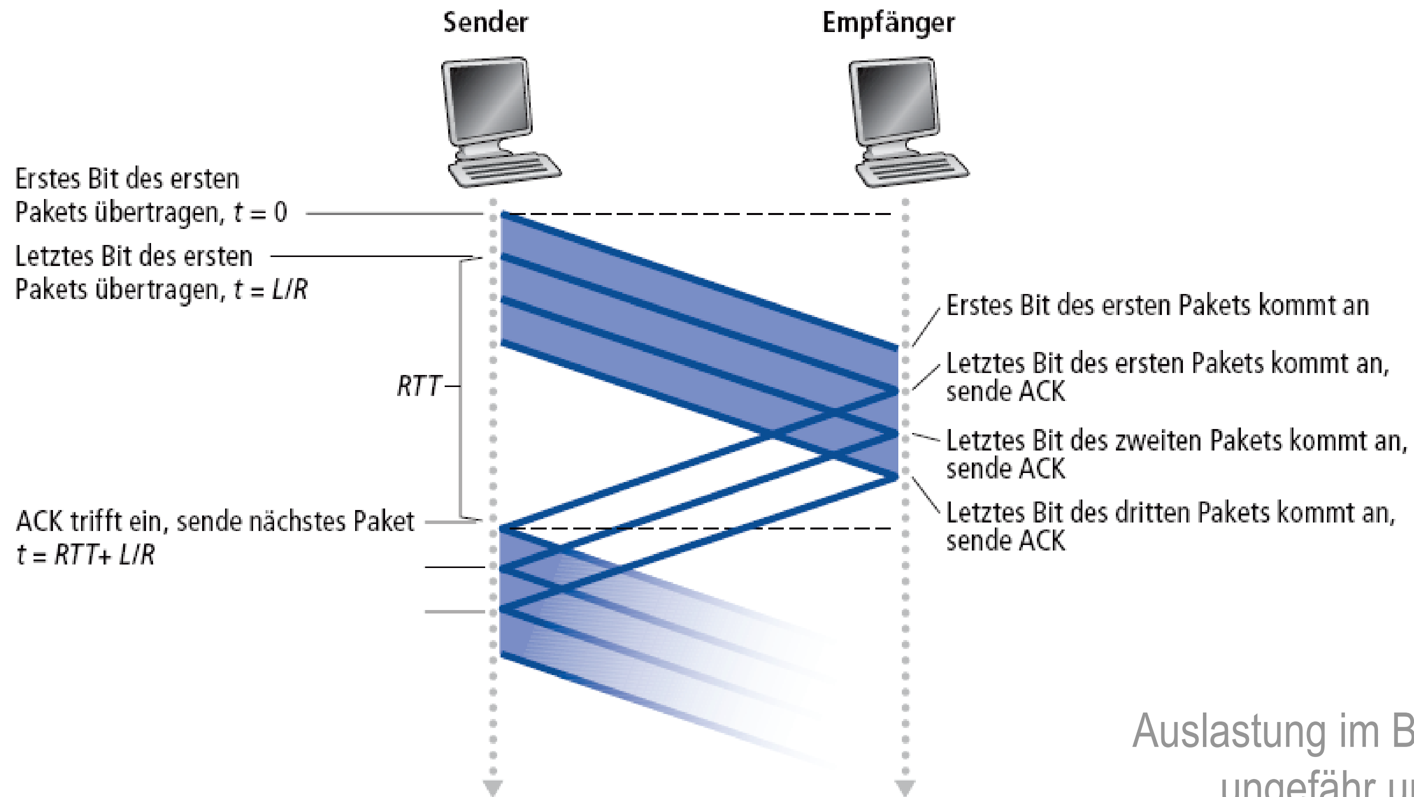
Pipelining: Sender lässt nicht nur eines, sondern mehrere unbestätigte Pakete zu

→ Die Anzahl der Sequenznummern muss erhöht werden

→ Pakete müssen beim Sender und/oder Empfänger gebuffert werden

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

3.4.2 Zuverlässige Datentransferprotokolle mit Pipelining



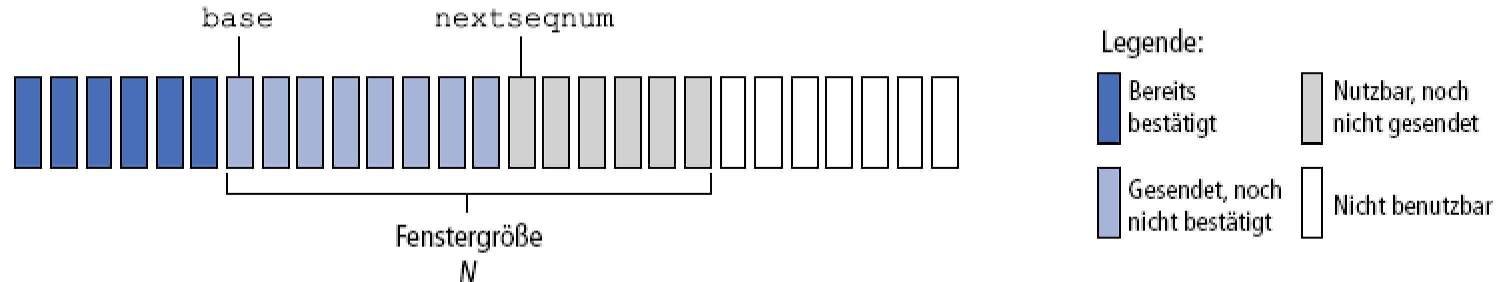
b Ablauf bei Pipelining

Auslastung im Beispiel ungefähr um Faktor 3 erhöht!

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

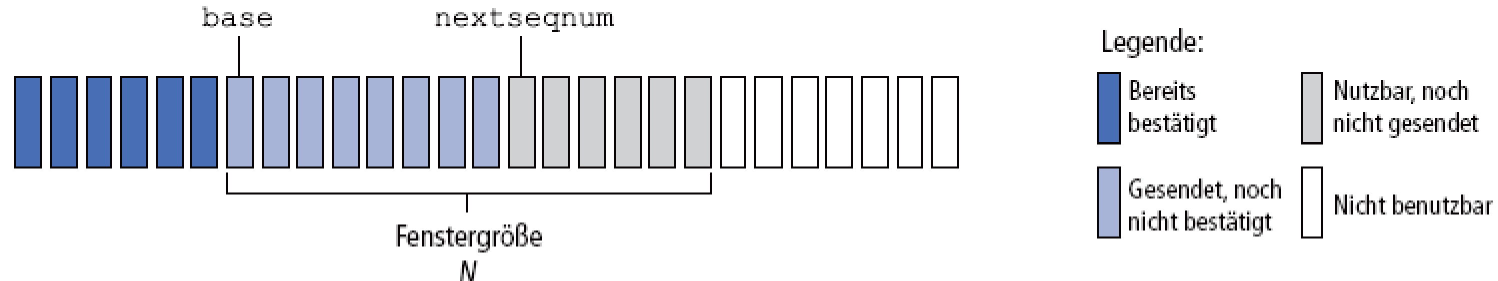
3.4.3 Go-Back-N (GBN)

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



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

3.4.3 Go-Back-N (GBN)



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

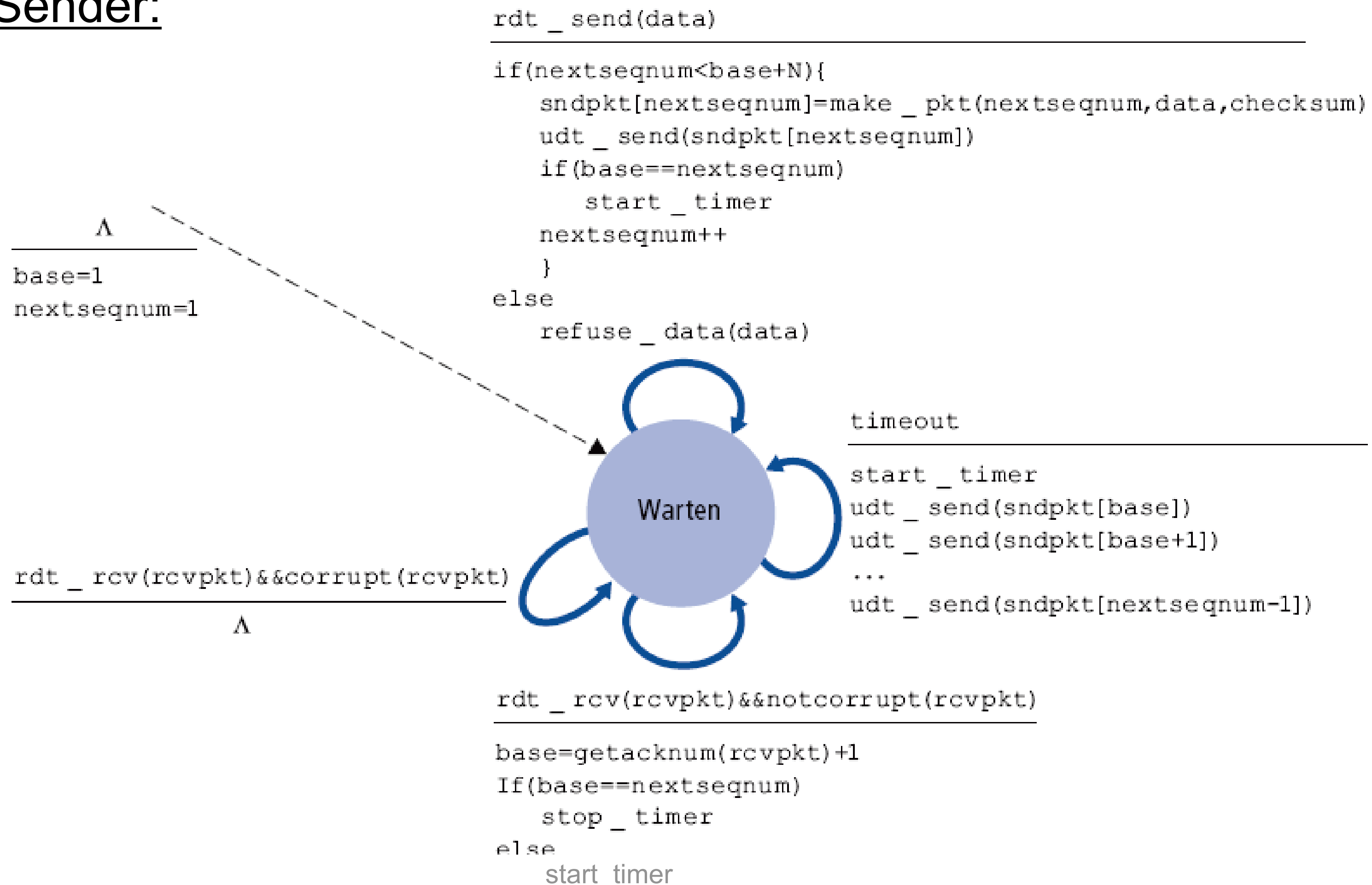
Gesendet, noch nicht bestätigt:
[base, nextseqnum-1] umfasst Pakete, die zwar gesendet wurden, aber noch nicht bestätigt worden sind.

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

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

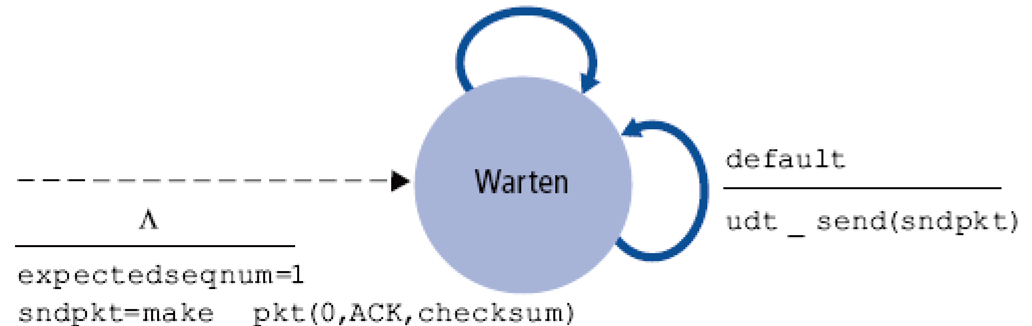
3.4.3 Go-Back-N (GBN)

GBN Sender:



3.4.3 Go-Back-N (GBN)

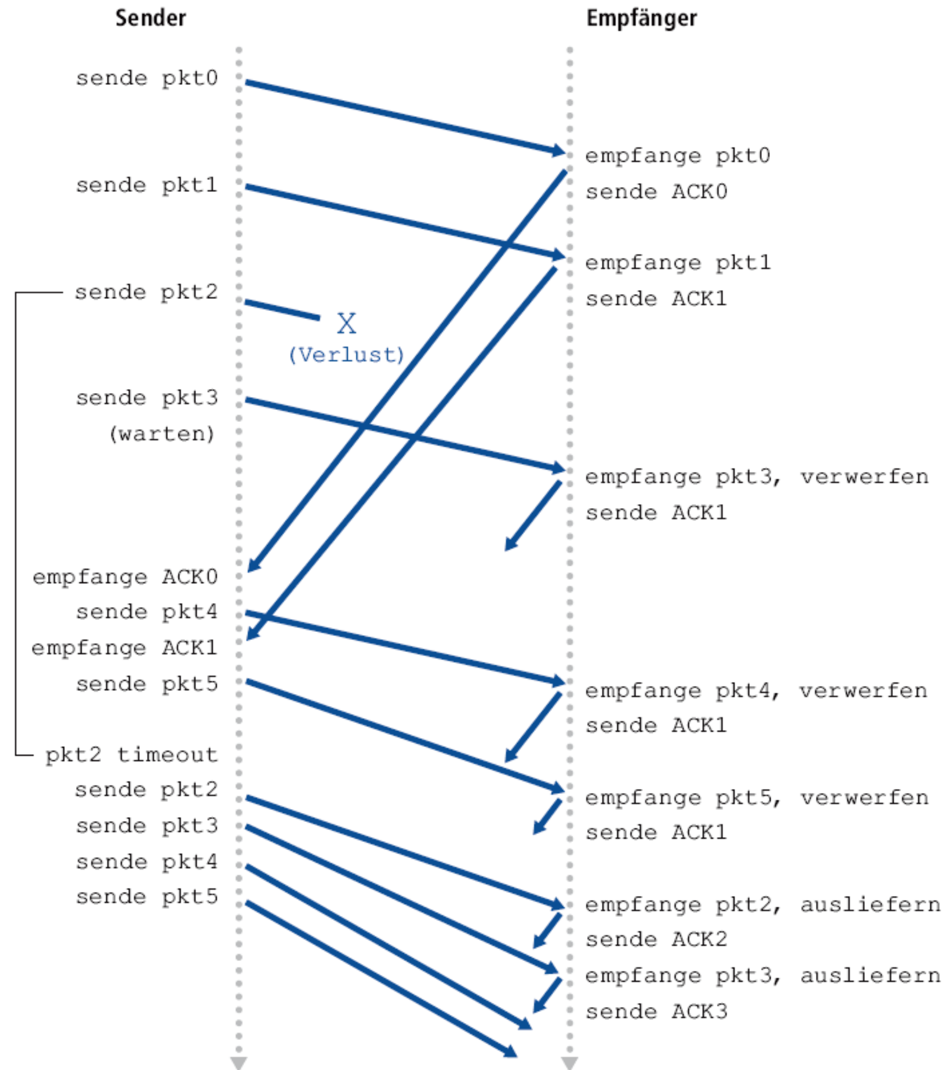
GBN Empfänger:



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

3.4.3 Go-Back-N (GBN)

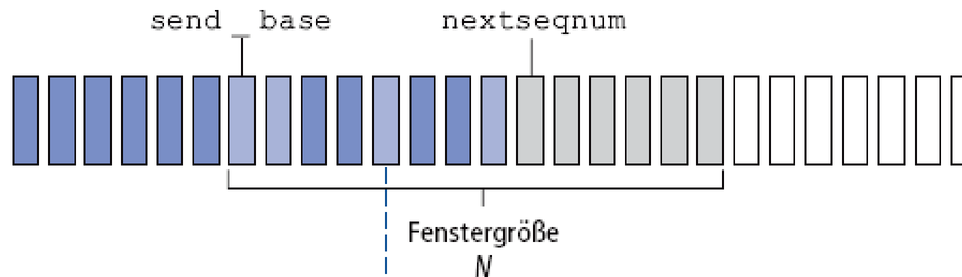
GBN Ablauf:



3.4.4 Selective Repeat (SR)

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

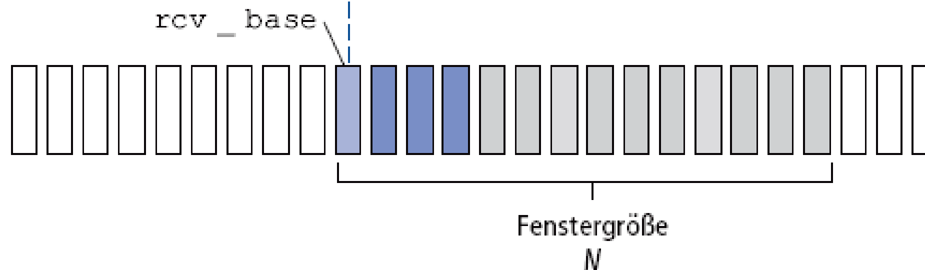
3.4.4 Selective Repeat (SR)



Legende:

- Bereits bestätigt
- Nutzbar, noch nicht gesendet
- Gesendet, noch nicht bestätigt
- Nicht benutzbar

a **Sequenznummern aus Sicht des Senders**



Legende:

- Außerhalb der Reihenfolge, gepuffert und bereits bestätigt
- Akzeptierbar (innerhalb des Fensters)
- Erwartet, noch nicht eingetroffen
- Nicht benutzbar

b **Sequenznummern aus Sicht des Empfängers**

3.4.4 Selective Repeat (SR)

Sender:

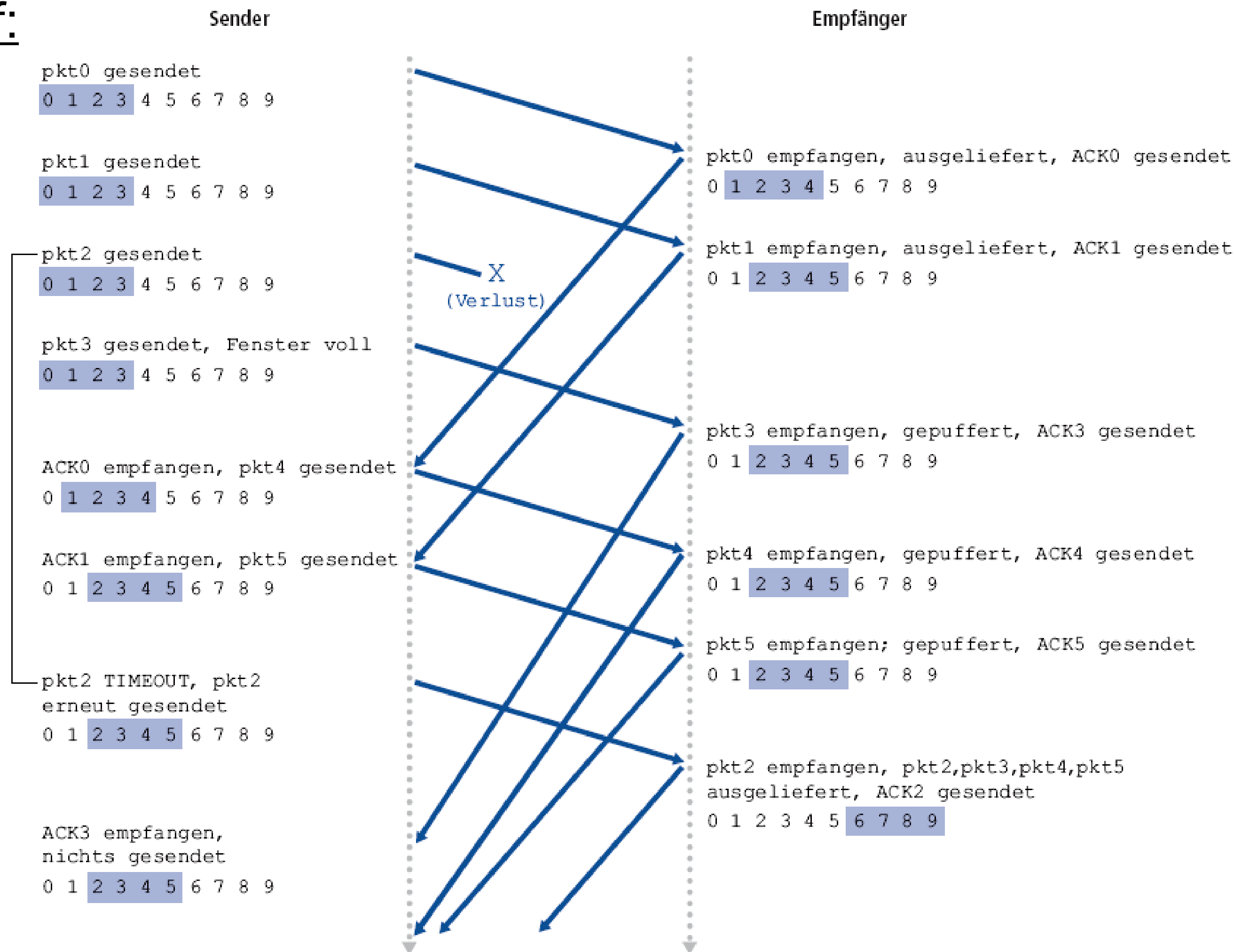
- **Daten von oben**:
  - Wenn nächste Sequenznummer im Fenster liegt: Paket senden, Timer(n) starten!
- **Timeout(n):**
  - Paket n erneut übertragen, Timer(n) starten
- **ACK(n) aus** [sendbase,sendbase+N]
  - Paket n als empfangen markieren
  - Wenn n die kleinste unbestätigte Sequenznummer ist, Fenster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

Empfänger:

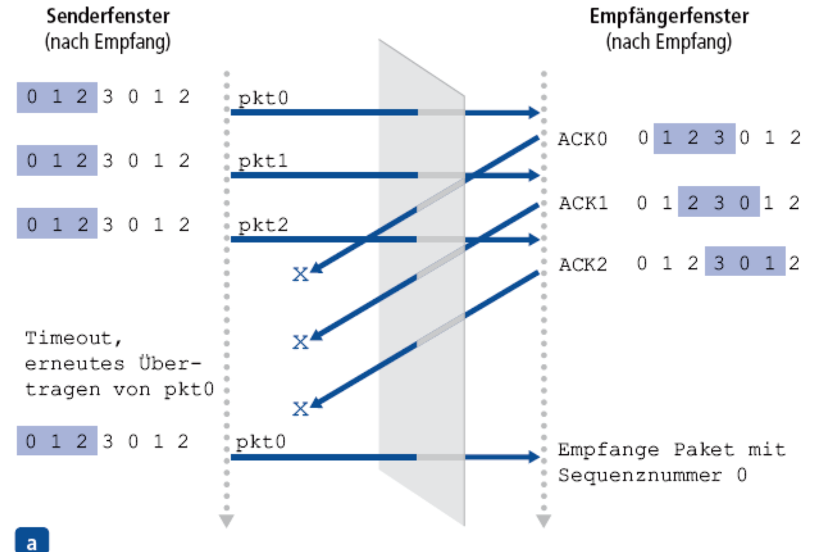
- **Paket aus** [rcvbase, rcvbase+N-1]
  - Sende ACK(n)
  - Außer der Reihe: Buffern
  - In der Reihe: Ausliefern (auch alle gebufferten Pakete ausliefern, die jetzt in der Reihe sind), Fenster zum nächsten erwarteten Paket verschieben
- **Paket aus** [rcvbase-N,rcvbase-1]
  - ACK(n)
- **Sonst:**
  -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

3.4.4 Selective Repeat (SR)

SR Ablauf:



3.4.4 Selective Repeat (SR)



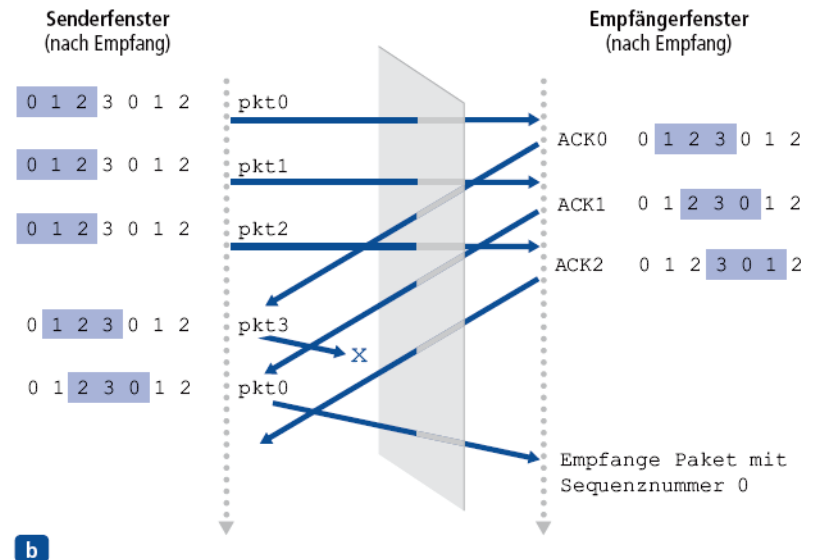
SR Problemfall:

Beispiel:

- Sequenznummern: 0-3
- Fenstergröße=3

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

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



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

3.4.4 Pipelining Techniken: GBN vs. SR

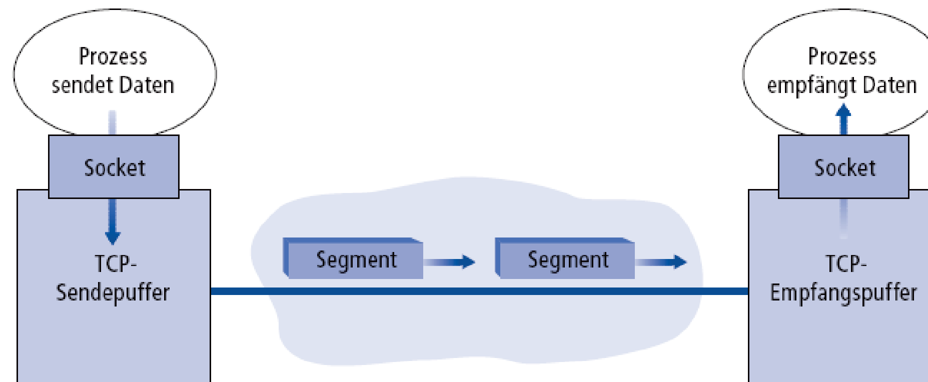
	Go-Back-N (GBN)	**Selective Repeat (SR)**
Empfänger sendet ACKs	Kumulativ	Für jedes Paket einzeln
Timer	Nur **einer** für das älteste unbestätigte Paket	Jedes Paket hat einen eigenen Timer
Verhalten des Senders bei Timeout	Alle Pakete im Fenster erneut senden	Nur ein einzelnes Paket
Fenster notwendig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

3.5 Verbindungsorientierter Transport: TCP

3.5 Verbindungsorientierter Transport: TCP

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

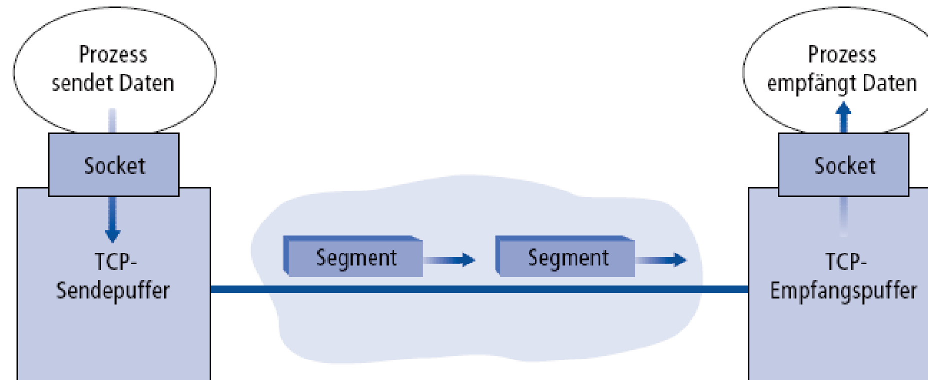
3.5 Verbindungsorientierter Transport: TCP



“**Drei-Wege-Handshake**” (three-way handshake) zum Verbindungsaufbau:

1. Client-Prozess überträgt einen Datenstrom durch den Socket
2. TCP packt diese Daten in den Sendebuffer der Verbindung
3. Diese Daten werden in TCP-Segmenten zur Netzwerkschicht heruntergereicht
   - **MSS** (*maximum segment size* = maximale Segmentgröße)
   - **MTU** (*maximum transmission unit* = maximale Übertragungseinheit)
4. Die Netzwerkschicht verkapselt die Segmente in IP-Datagramme und sendet sie ins Netz

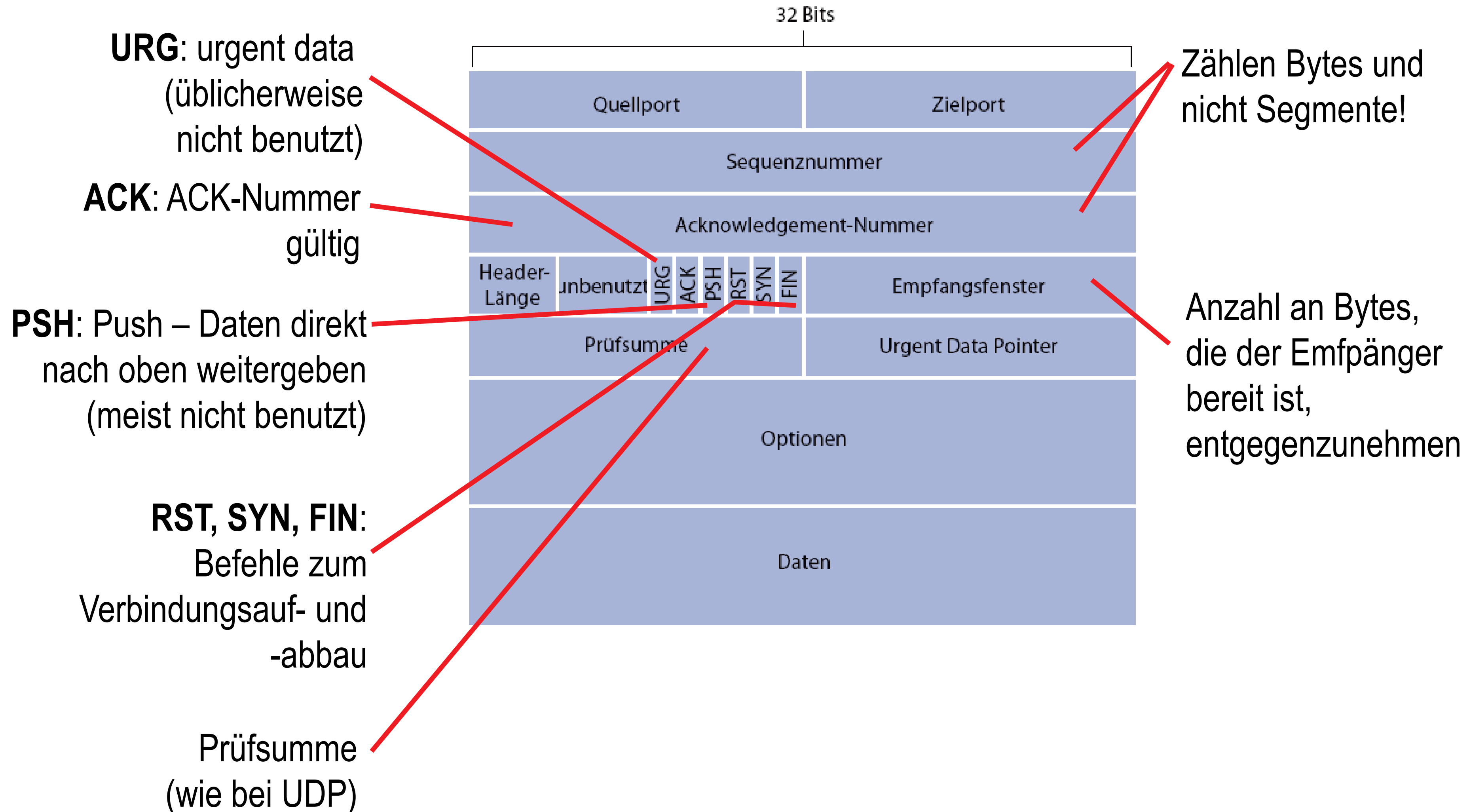
3.5 Verbindungsorientierter Transport: TCP



5. Sobald TCP auf dem Server ein Segment erhält, werden die Daten in den Eingangsbuffer der zugehörigen TCP-Verbindung eingefügt
6. Die Anwendung liest den Datenstrom aus diesem Buffer

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

3.5.2 TCP-Segmentstruktur



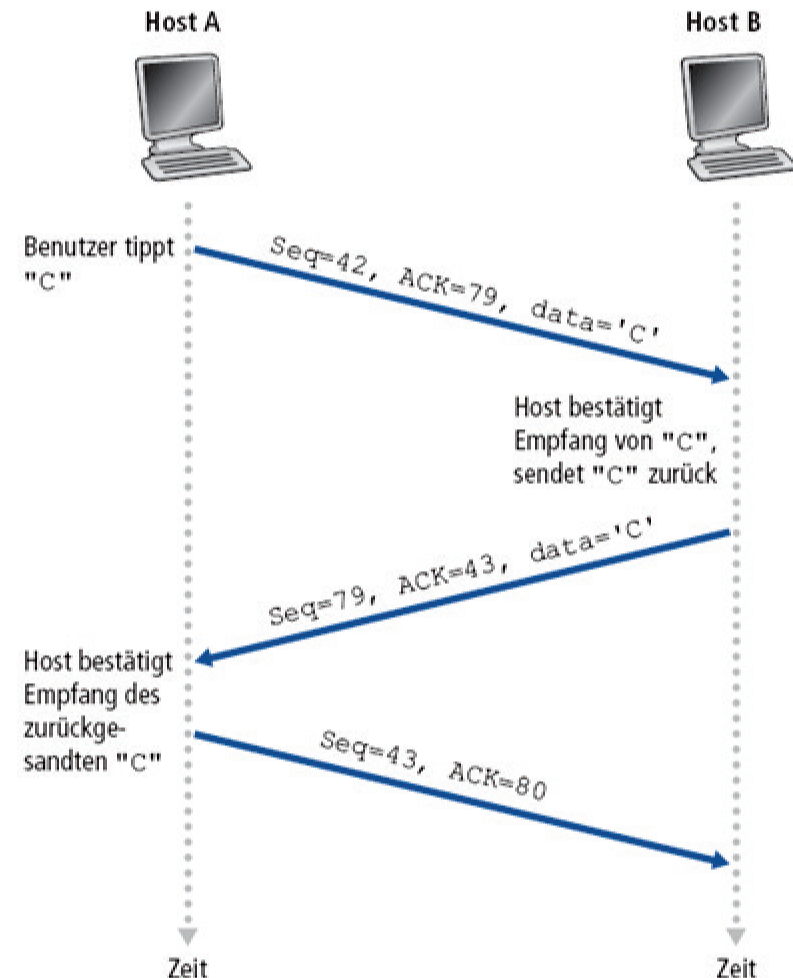
3.5.2 TCP-Sequenznummern und -ACKs

Zwei der wichtigsten Felder im TCP-Segment-Header:

<u>Sequenznummern</u>
= Nummer des ersten Byte im Datenteil.
Die Anfangssequenznummern werden von beiden Seiten zufällig gewählt.

<u>Acknowledgementnummern (ACKs)</u>
= Sequenznummer des nächsten Byte, das von der Gegenseite erwartet wird.

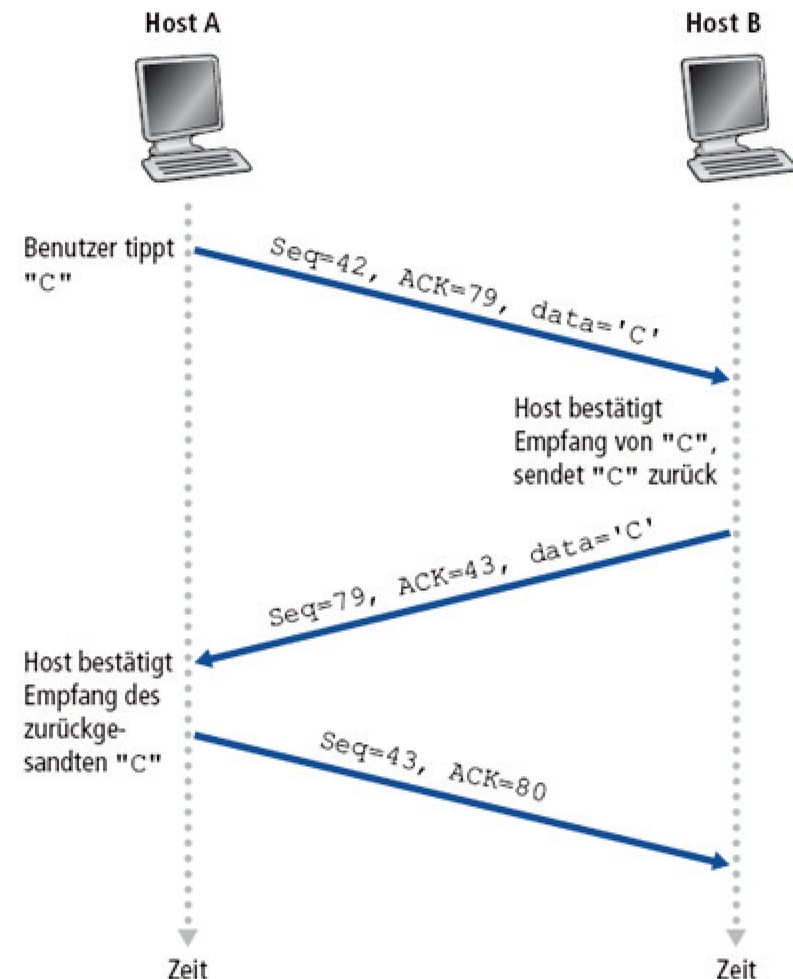
- Kumulative ACKs
  TCP bestätigt nur Bytes bis zum ersten fehlenden Byte im Strom
- “piggybacked” Acknowledgement
  Jedes ACK wird als Teil eines Segments transportiert das tatsächliche Daten enthalten kann



3.5.2 TCP-Sequenznummern und -ACKs

→ Was passiert, wenn Segmente außer der Reihe ankommen?

Ist in den RFCs für TCP **nicht vorgegeben**, wird der Implementation überlassen.



Zwei Möglichkeiten:

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

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