Netzwerktechnologien
3 VO

Univ.-Prof. Dr. Helmut Hlavacs
helmut.hlavacs@univie.ac.at

Dr. Ivan Gojmerac
gojmerac@ftw.at

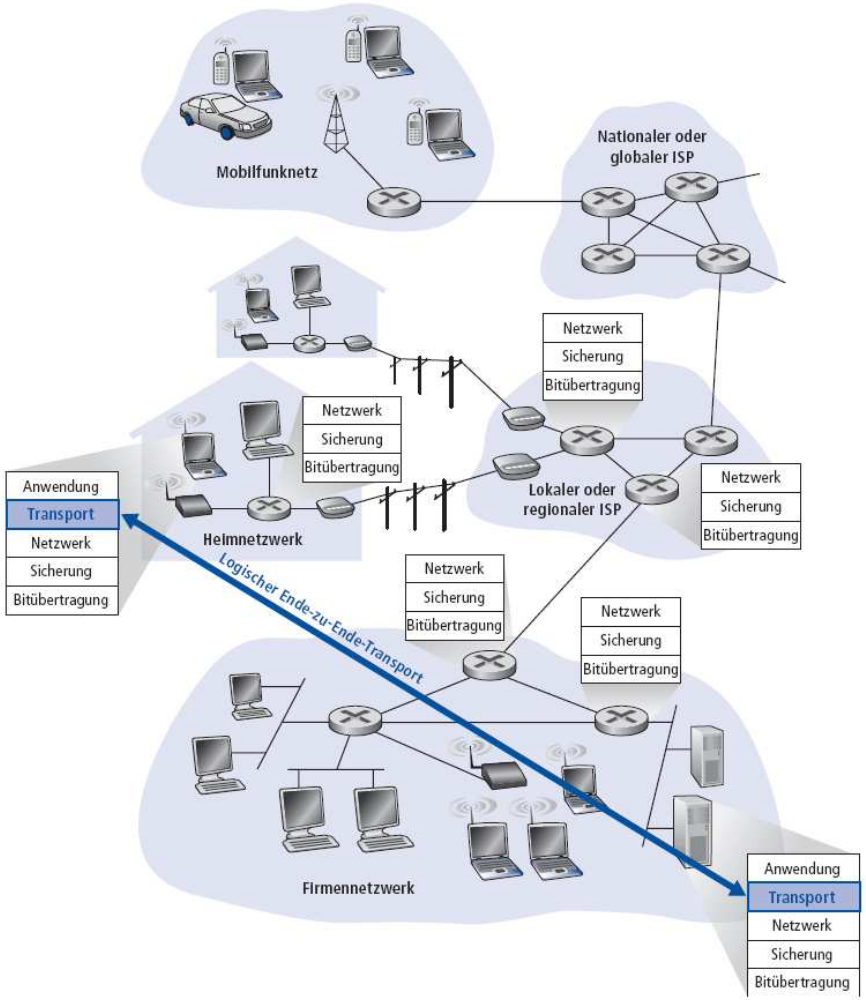
Bachelorstudium Medieninformatik
SS 2012

Kapitel 3 – Transportschicht

3.1 Dienste der Transportschicht
3.2 Multiplexing und Demultiplexing
3.3 Verbindungsloser Transport: UDP
3.4 Grundlagen der zuverlässigen Datenübertragung
3.5 Verbindungsorientierter Transport: TCP
3.6 Grundlagen der Überlastkontrolle
3.7 TCP-Überlastkontrolle

3.1 Dienste der Transportschicht

- Stellen **logische Kommunikation** zwischen Anwendungsprozessen auf verschiedenen Hosts zur Verfügung
- Transportprotokolle laufen auf Endsystemen
  - *Sender*: teilt Anwendungsnachrichten in **Segmente** auf und gibt diese an die Netzwerkschicht weiter
  - *Empfänger*: fügt Segmente wieder zu Anwendungsnachrichten zusammen, gibt diese an die Anwendungsschicht weiter
- Es existieren verschiedene Transportschichtprotokolle
  - Internet: TCP und UDP



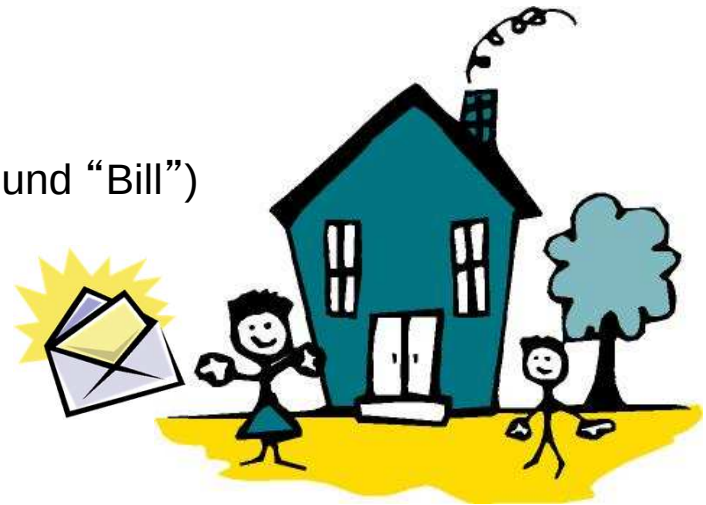
3.1 Unterschied: Transport- und Netzwerkschicht

- *Netzwerkschicht:* logische Kommunikation zwischen Hosts
- *Transportschicht:* logische Kommunikation zwischen Prozessen
  - verwendet und erweitert die Dienste der Netzwerkschicht

<u>Analogie: Briefzustellung an mehrere Bewohner des selben Hauses</u>

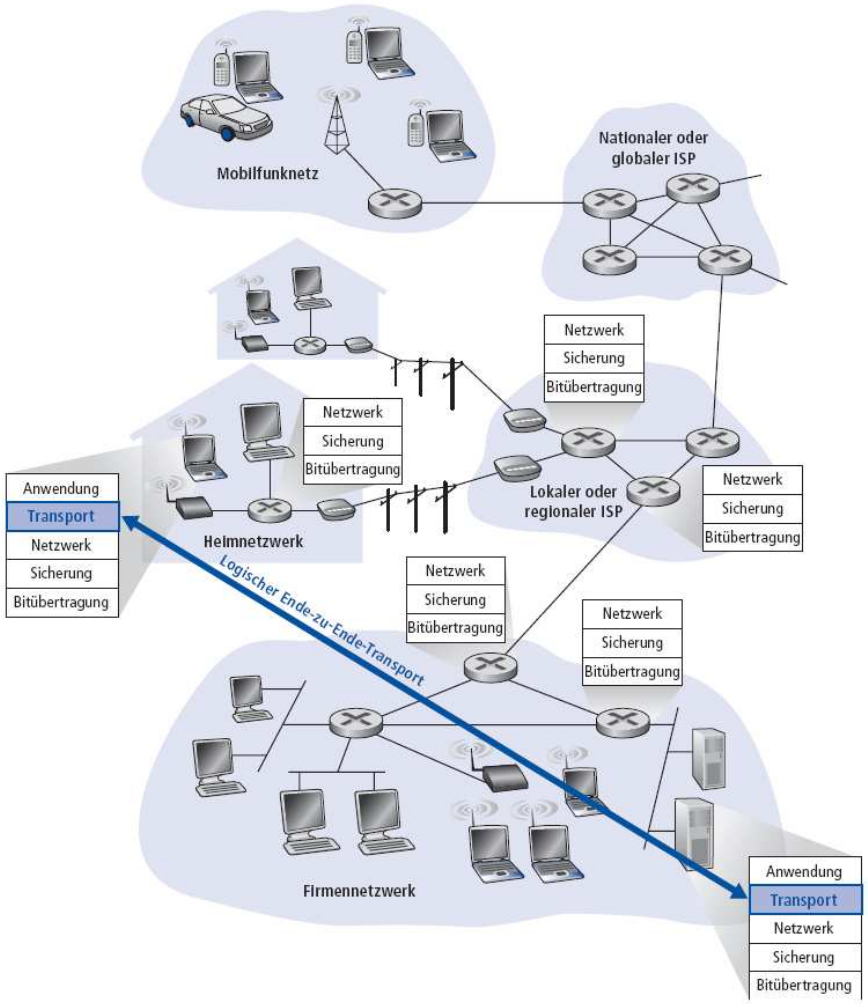
→ *12 Kinder senden Briefe an 12 andere Kinder.*

- Prozess = Kind
- Anwendungsnachricht = Brief in einem Umschlag
- Host = Haus
- Transportprotokolle = Namen der Kinder (z.B. “Anne” und “Bill”)
- Netzwerkprotokoll = Postdienst

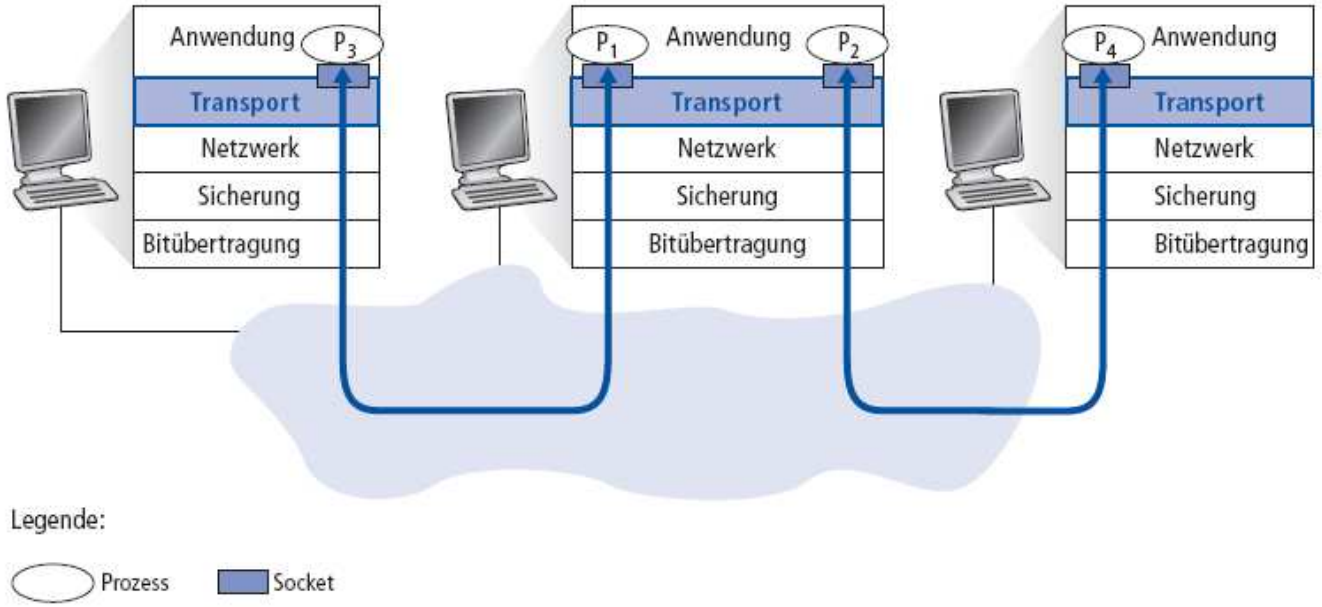


3.1 Transportprotokolle im Internet

- Zuverlässige, reihenfolgeerhaltende Auslieferung (TCP)
  - Überlastkontrolle
  - Flusskontrolle
  - Verbindungsmanagement
- Unzuverlässige Datenübertragung ohne Reihenfolgeerhaltung: UDP
  - Minimale Erweiterung der “Best-Effort”-Funktionalität von IP
- Dienste, die <u>nicht</u> zur Verfügung stehen:
  - × Garantien bezüglich Bandbreite oder Verzögerung



3.2 Multiplexing und Demultiplexing



- **Multiplexing** beim Sender:

Daten von mehreren Sockets einsammeln, Daten mit einem Header versehen (der später für das Demultiplexing verwendet wird).

- **Demultiplexing** beim Empfänger:

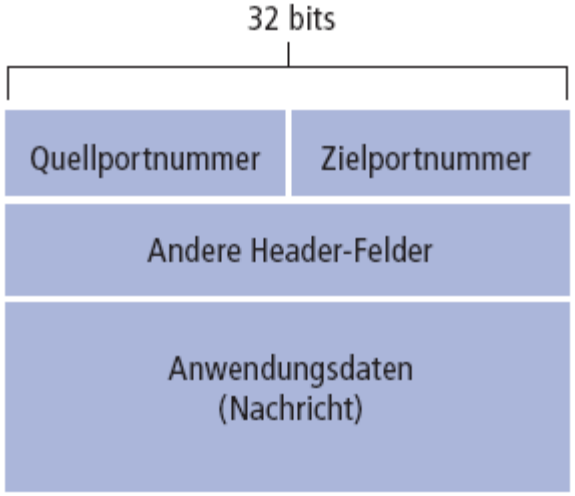
Empfangene Segmente am richtigen Socket abliefern.

3.2.1 Demultiplexing

Analogie: Briefzustellung an mehrere Bewohner des selben Hauses

Bill erhält einen Stapel Briefe vom Briefträger und gibt jeden davon dem Kind, dessen Name auf dem Brief steht.

- Host empfängt IP-Datagramme
  - Jedes Datagramm hat eine Absender-IP-Adresse und eine Empfänger-IP-Adresse
  - Jedes **Datagramm** beinhaltet ein **Transportschichtsegment**
  - Jedes **Segment** hat eine Absender- und eine Empfänger-Portnummer
- Hosts nutzen **IP-Adressen** und **Portnummern**, um Segmente an den richtigen Socket weiterzuleiten



TCP/UDP Segmentformat

3.2.1 Verbindungsloses Demultiplexing (UDP)

- Sockets mit Portnummer anlegen:

```
DatagramSocket mySocket1 = new DatagramSocket(12534);
DatagramSocket mySocket2 = new DatagramSocket(12535);
```

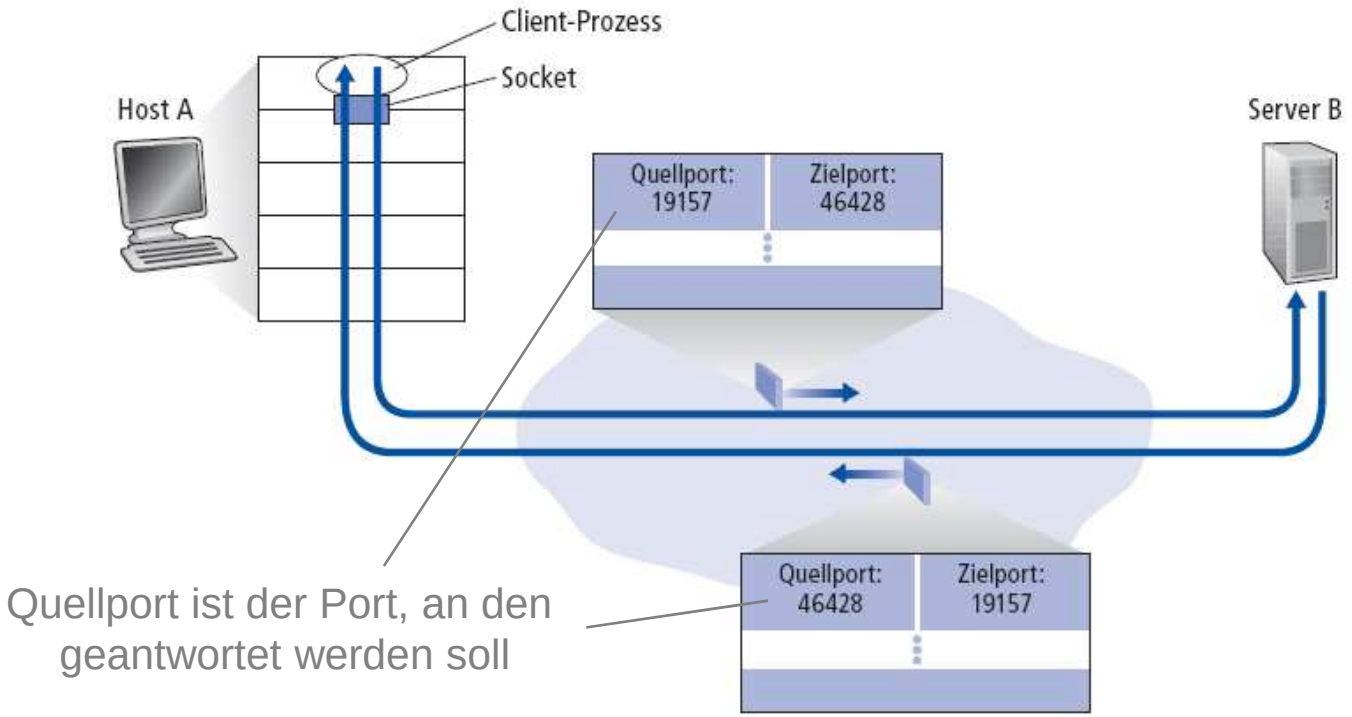
- **UDP**-Socket wird durch ein 2-Tupel identifiziert:

(Empfänger-IP-Adresse, Empfänger-Portnummer)

- Wenn ein Host ein UDP-Segment empfängt:
  1. Lese Empfänger-**Portnummer**
  2. Das UDP-Segment wird an den UDP-Socket mit dieser Portnummer weitergeleitet
- IP-Datagramme mit anderer Absender-IP-Adresse oder anderer Absender-Portnummer werden an **denselben Socket** ausgeliefert

3.2.1 Verbindungsloses Demultiplexing

```
DatagramSocket serverSocket = new DatagramSocket(46428);
```



3.2.2 Verbindungsorientiertes Demultiplexing (TCP)

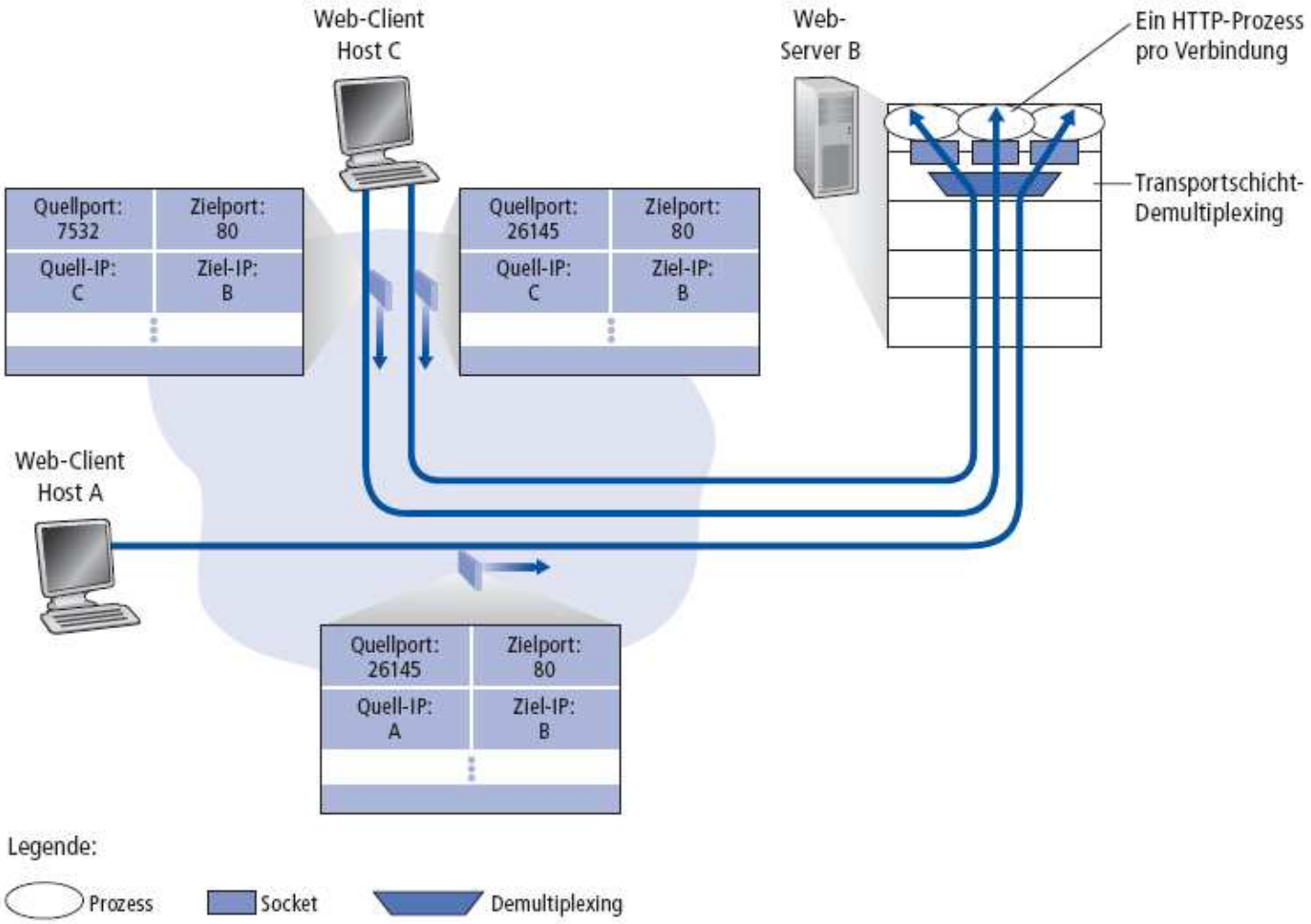
- **TCP**-Socket wird durch ein 4-Tupel identifiziert:
  - Absender-IP-Addresse
  - Absender-Portnummer
  - Empfänger-IP-Adresse
  - Empfänger-Portnummer

→ *Empfänger nutzt alle vier Werte, um den richtigen TCP-Socket zu identifizieren*

- Server kann viele TCP-Sockets gleichzeitig offen haben:
  - Jeder Socket wird durch sein eigenes 4-Tupel identifiziert

- Webserver haben verschiedene Sockets für jeden einzelnen Client
  - Bei nichtpersistentem HTTP wird jede Anfrage über einen eigenen Socket beantwortet (dieser wird nach jeder Anfrage wieder geschlossen)

3.2.2 Verbindungsorientiertes Demultiplexing (TCP)



3.3 Verbindungsloser Transport: UDP

- Minimales Internet-Transportprotokoll
- "Best-Effort"-Dienst, UDP-Segmente können:
  - verloren gehen
  - in der falschen Reihenfolge an die Anwendung ausgeliefert werden
- *Verbindungslos:*
  - kein Handshake zum Verbindungsaufbau
  - jedes UDP-Segment wird unabhängig von allen anderen behandelt

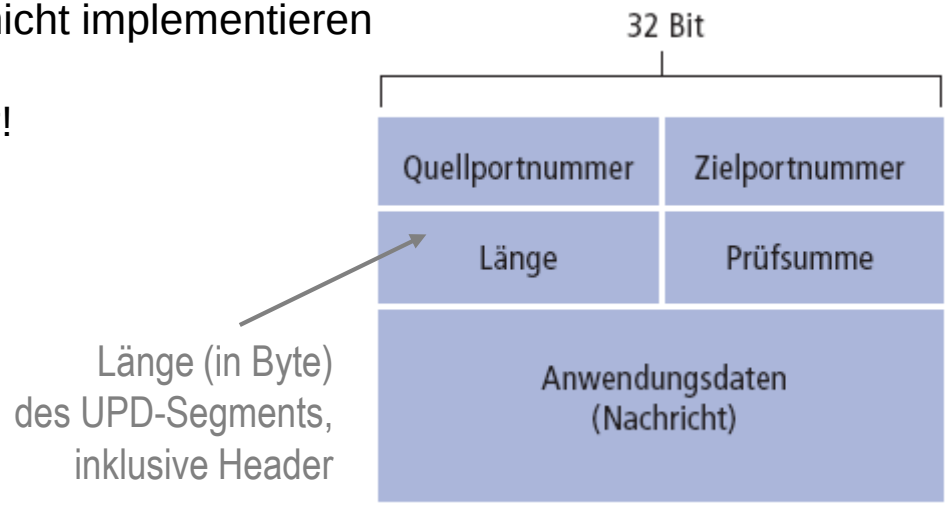
<u>Warum gibt es UDP?</u>

- Kein Verbindungsaufbau (der zu Verzögerungen führen kann)
- Einfach: kein Verbindungszustand im Sender oder Empfänger
- Kleiner Header
- Keine Überlastkontrolle: UDP kann so schnell wie von der Anwendung gewünscht senden

3.3 Verbindungsloser Transport: UDP

- Häufig für Anwendungen im Bereich **Multimedia-Streaming** eingesetzt
  - Verlusttolerant
  - Mindestrate
- Andere Einsatzgebiete
  - DNS
  - SNMP
- Zuverlässiger Datentransfer über UDP:
  Zuverlässigkeit auf der Anwendungsschicht implementieren

→ Anwendungsspezifische Fehlerkorrektur!



UDP Segmentformat

3.3.1 Fehlerkorrekturmechanismus: UDP Prüfsumme

Ziel: Fehler im übertragenen Segment erkennen (z.B. verfälschte Bits)

Sender:

- Betrachte Segment als Folge von 16-Bit-Integer-Werten
- Prüfsumme: Addition (1er-Komplement-Summe) dieser Werte
- Sender legt das *invertierte Resultat* im UDP-Prüfsummenfeld ab

Empfänger:

- Berechne die Prüfsumme des empfangenen Segments **inkl. des Prüfsummenfeldes**
- Sind im Resultat alle Bits 1?
  - → NEIN – Fehler erkannt
  - → JA – Kein Fehler erkannt

3.3.1 Fehlerkorrekturmechanismus: UDP Prüfsumme

Zahlen werden addiert und ein Übertrag aus der höchsten Stelle wird zum Resultat an der niedrigsten Stelle addiert.

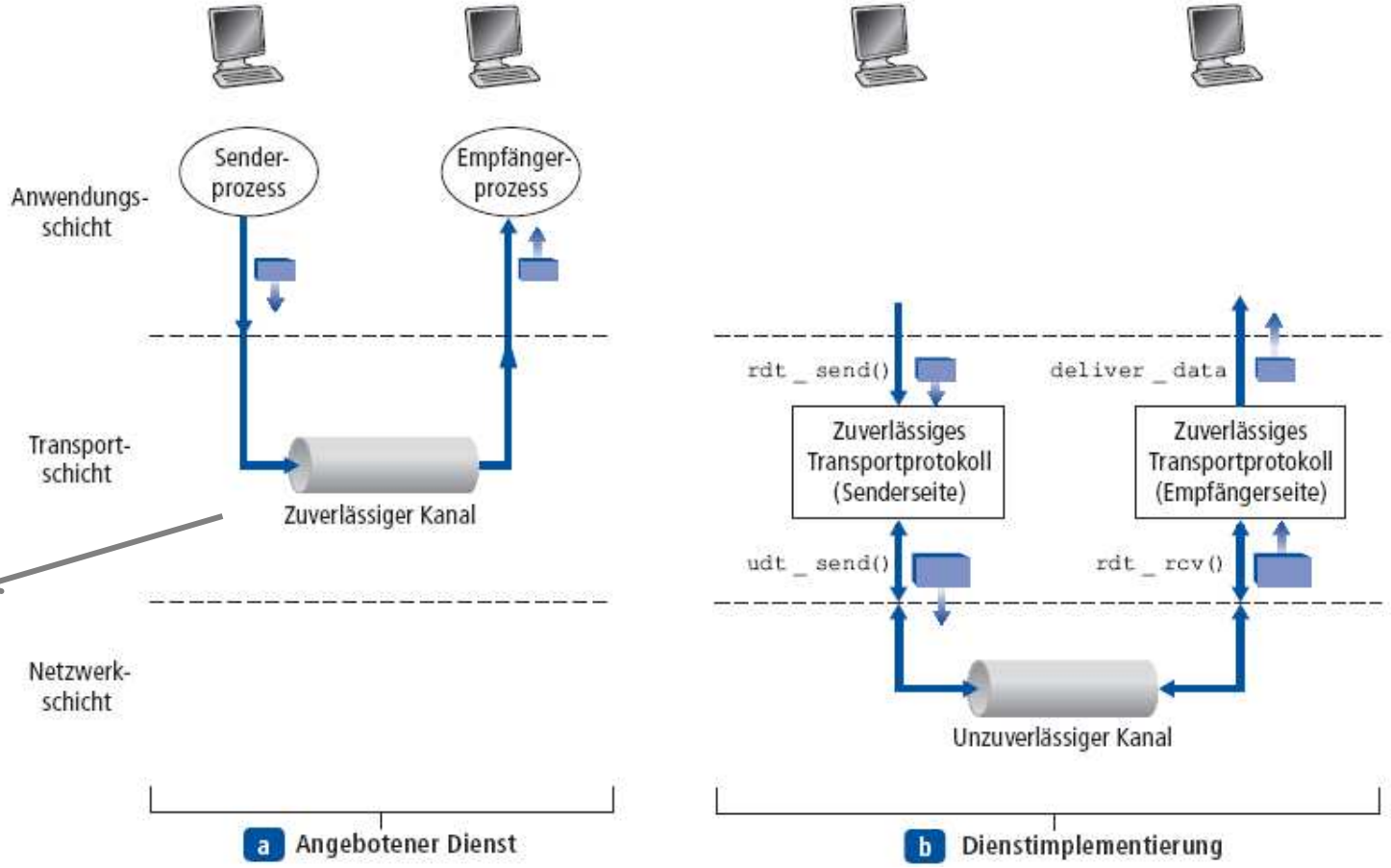
Beispiel:
Addiere zwei 16-Bit-Integer-Werte.

		1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
	+	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
Übertrag		1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
Summe (mit Übertrag)		1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
Prüfsumme		0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

3.4 Zuverlässigkeit der Datenübertragung

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

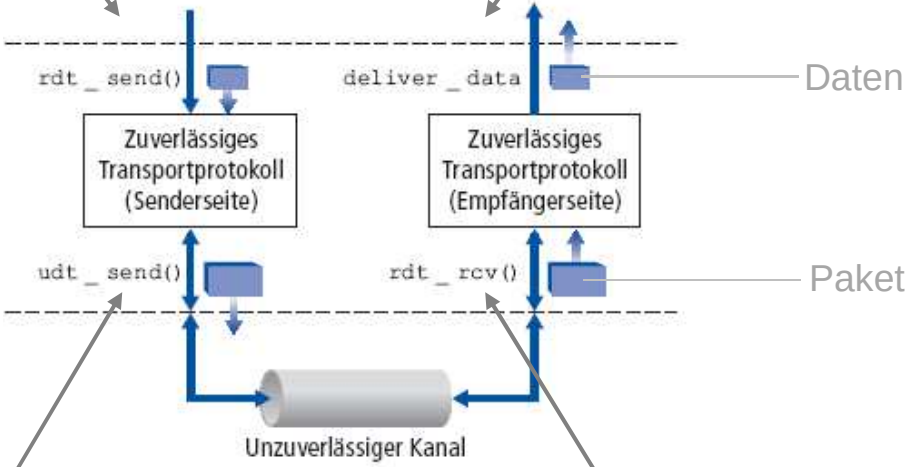
Eigenschaften des unzuverlässigen Kanals bestimmen die Komplexität des Protokolls zur zuverlässigen Datenübertragung (rdt)



3.4 Grundlagen der zuverlässigen Datenübertragung

`rdt_send():` von oben (z.B. der Anwendung) aufgerufen, übergebene Daten sollen beim Empfänger nach oben ausgeliefert werden

`deliver_data():` von rdt aufgerufen, um Daten nach oben weiterzugeben



`udt_send():` von rdt aufgerufen, um Paket über unzuverlässigen Kanal zum Empfänger zu übertragen

`rdt_rcv():` von unten aufgerufen, wenn ein Paket auf der Empfängerseite des unzuverlässigen Kanals ankommt

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

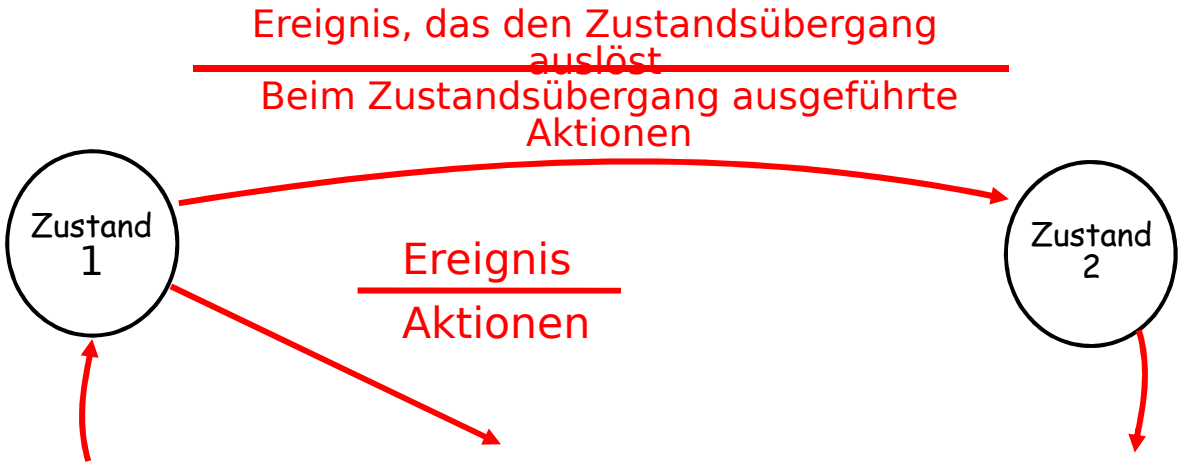
3.4.1 Aufbau eines zuverlässigen Datentransferprotokolls

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

3.4.1 Endliche Automaten

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

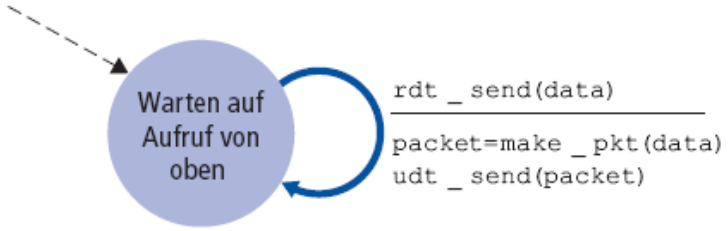
Zustand: Durch den Zustand und ein Ereignis ist der Nachfolgezustand bestimmt



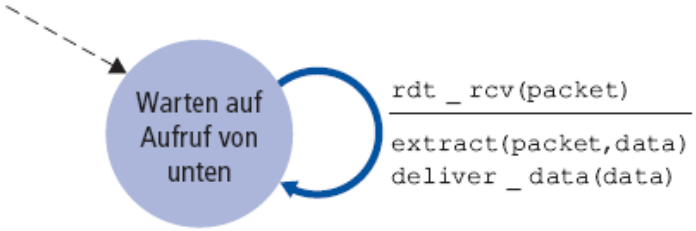
3.4.1 Aufbau eines zuverlässigen Datentransferprotokolls

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

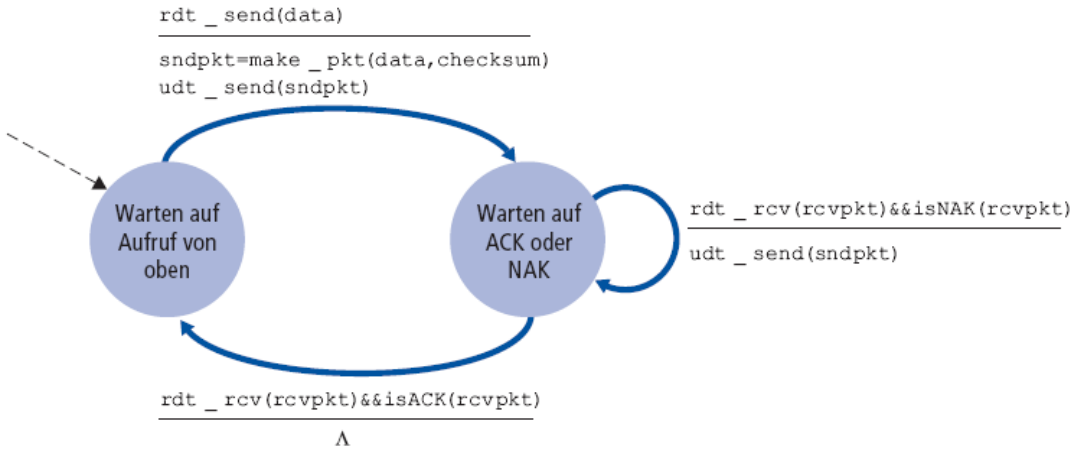
3.4.1 Aufbau eines zuverlässigen Datentransferprotokolls

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

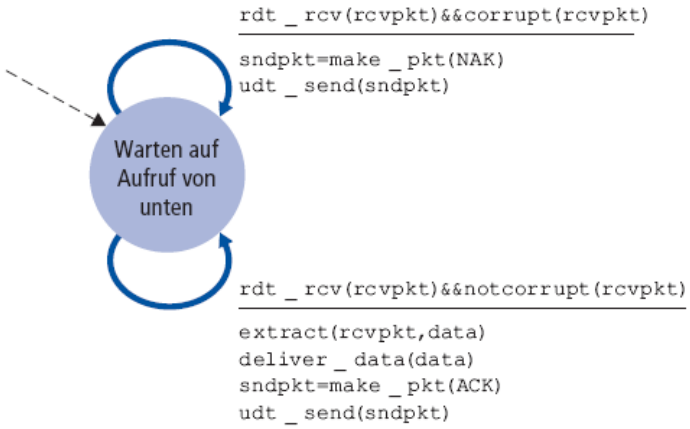
- Verfälschte Bits sind durch eine Prüfsumme erkennbar
- Mechanismen zur zuverlässigen Datenübertragung:
  - *Acknowledgements (ACKs):*

    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



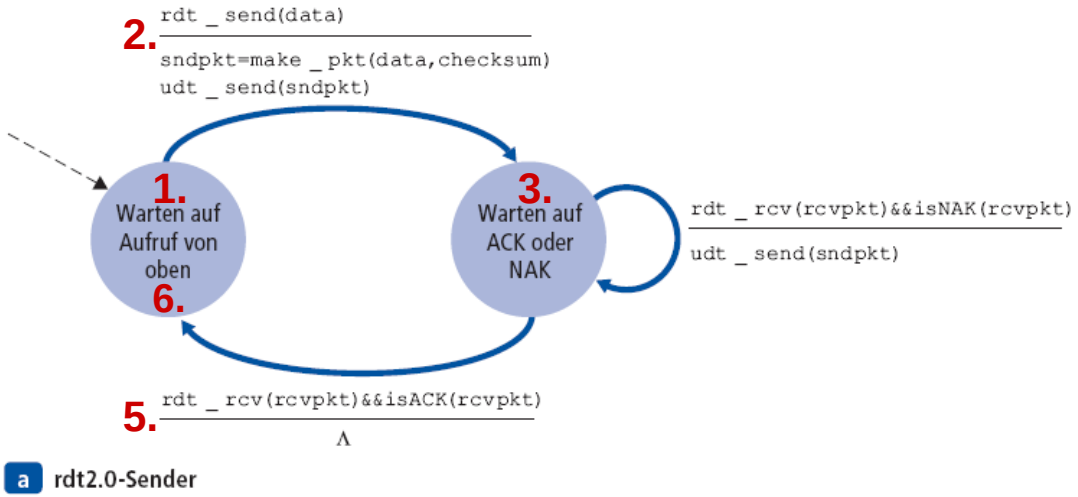
b rdt2.0-Empfänger

3.4.1 Aufbau eines zuverlässigen Datentransferprotokolls

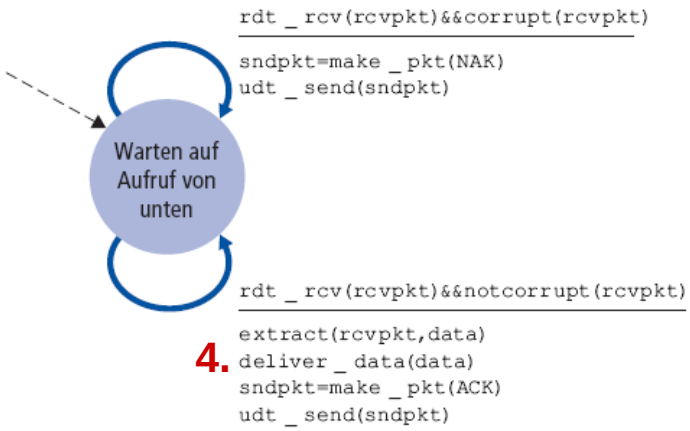
Zuverlässiger Datentransfer über einen Kanal mit Bitfehlern: **rdt2.0**

- 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



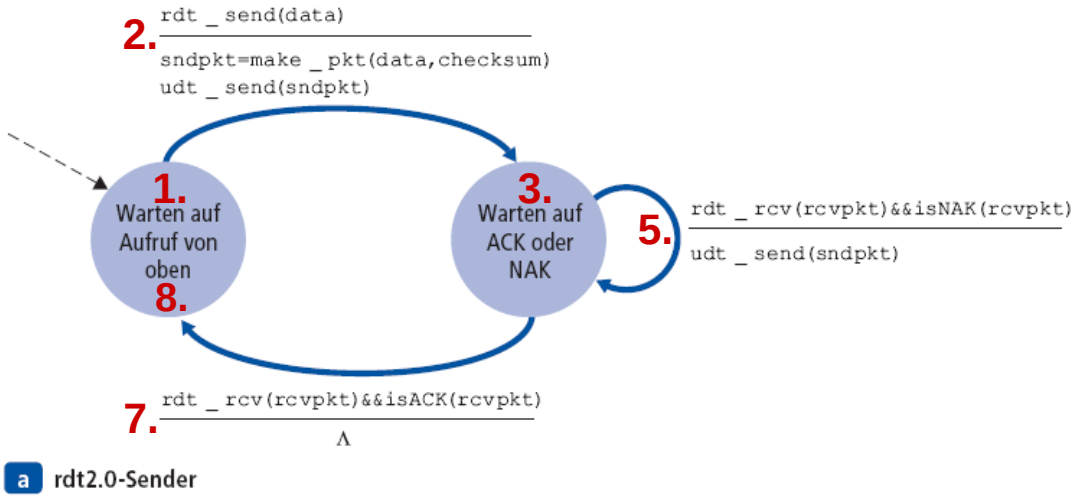
b rdt2.0-Empfänger

3.4.1 Aufbau eines zuverlässigen Datentransferprotokolls

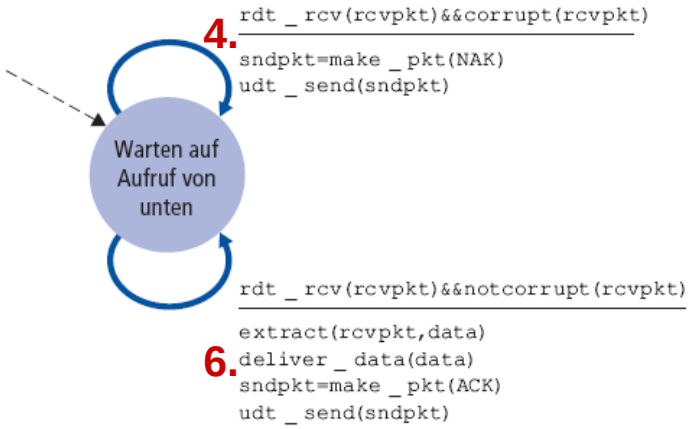
Zuverlässiger Datentransfer über einen Kanal mit Bitfehlern: **rdt2.0**

- 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:
    Fortlaufende Sequenznummer in Datenpakete einfügen.

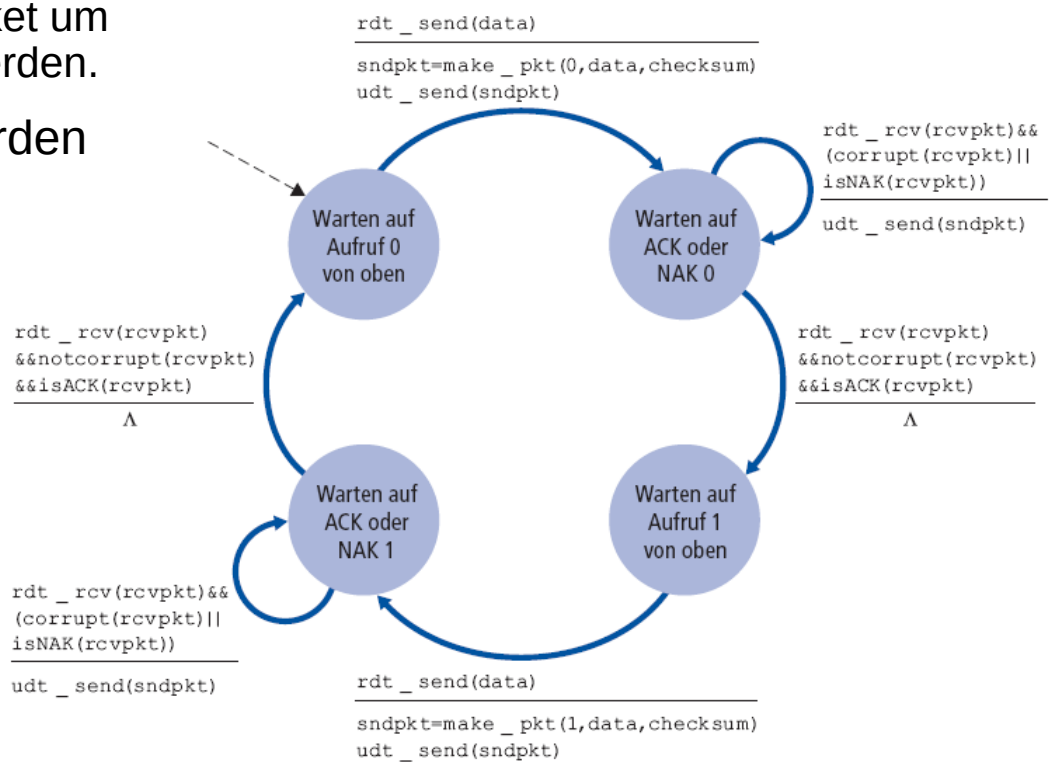
  → Bei Stop-and-Wait-Protokollen genügt eine 1-Bit-Zahl zum Vergleich der Sequenznummer mit der des zuletzt empfangenen Paketes.

3.4.1 Aufbau eines zuverlässigen Datentransferprotokolls

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

- Sequenznummern hinzugefügt
- Zwei Sequenznummern reichen (0,1).
  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

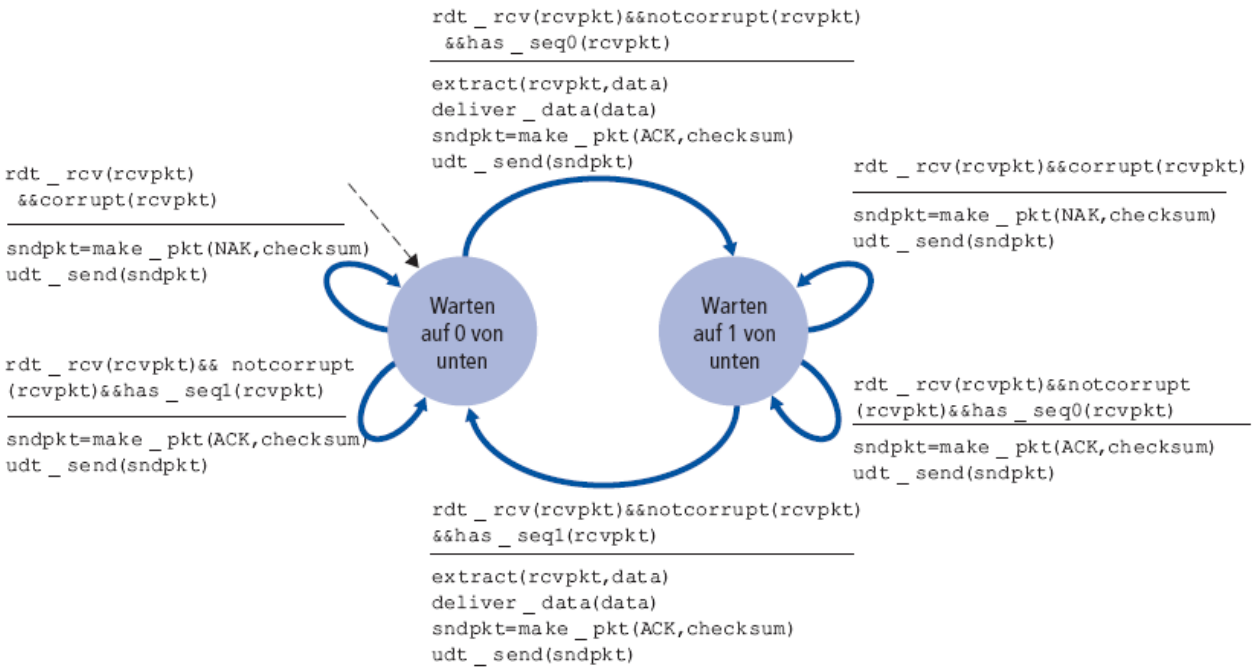


3.4.1 Aufbau eines zuverlässigen Datentransferprotokolls

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.



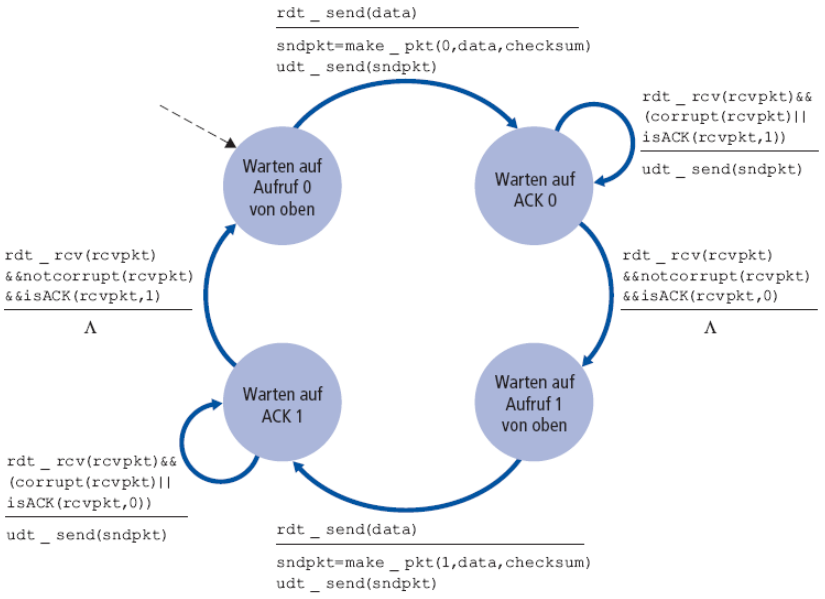
3.4.1 Aufbau eines zuverlässigen Datentransferprotokolls

rdt2.2: Elimination von NAKs

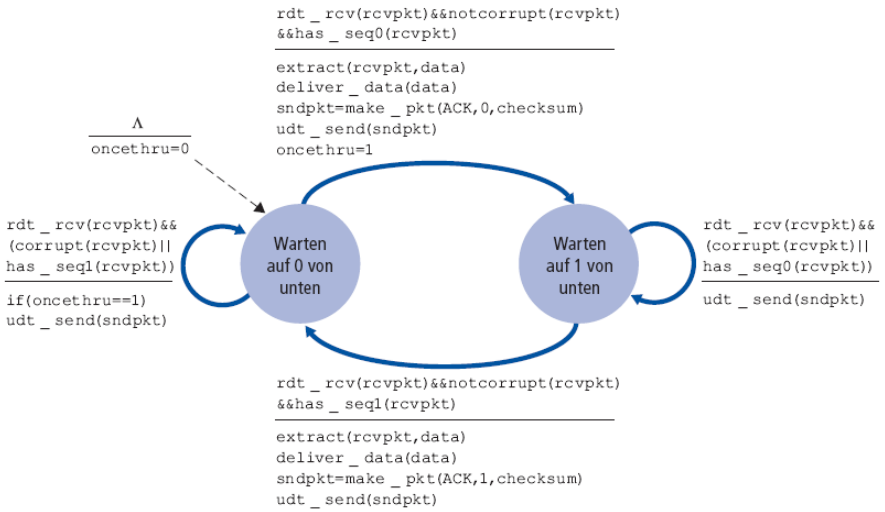
- Anstelle von NAKs: ACK für das letzte korrekt empfangene Paket

→ Empfänger muss die Sequenznummer des bestätigten Paketes im ACK mitschicken

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



a rdt2.2-Sender



b 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?

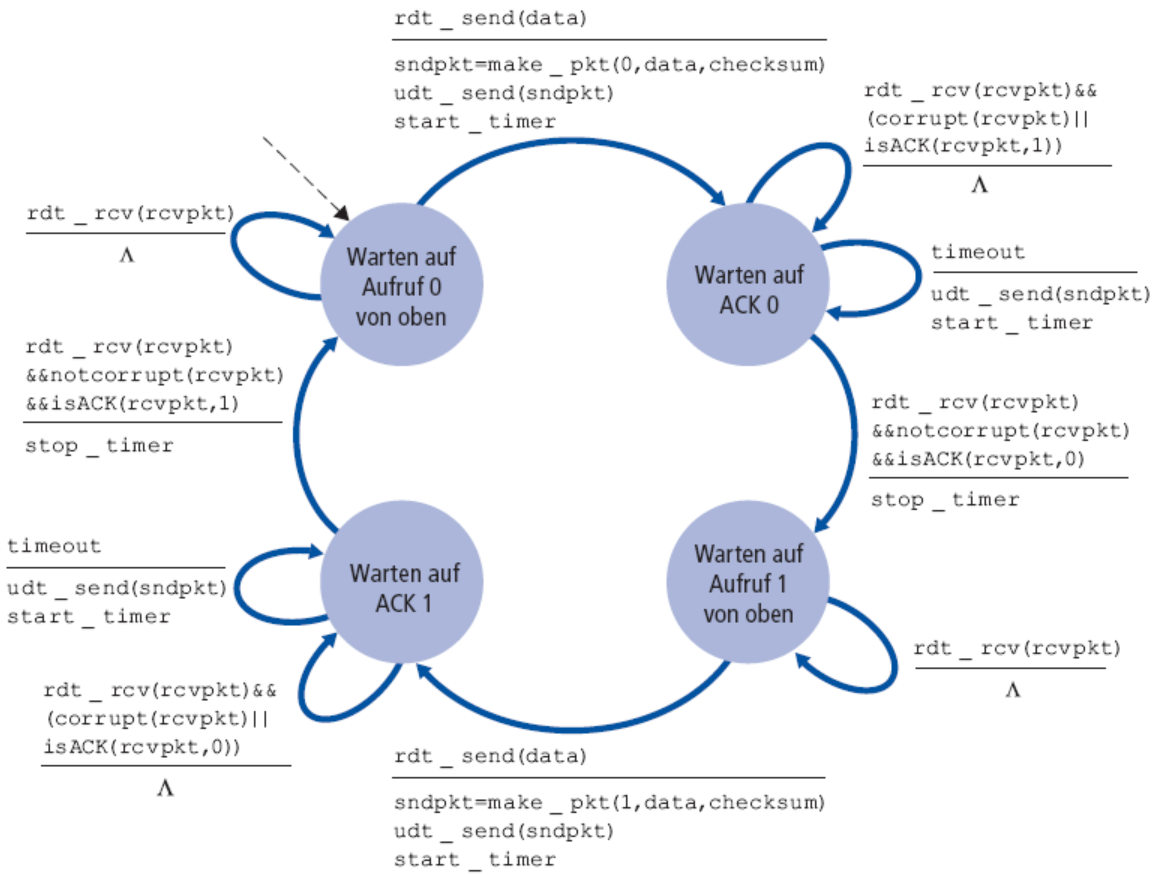
Lösung: Sender wartet eine „vernünftige Zeitspanne“ auf ein ACK

- 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**

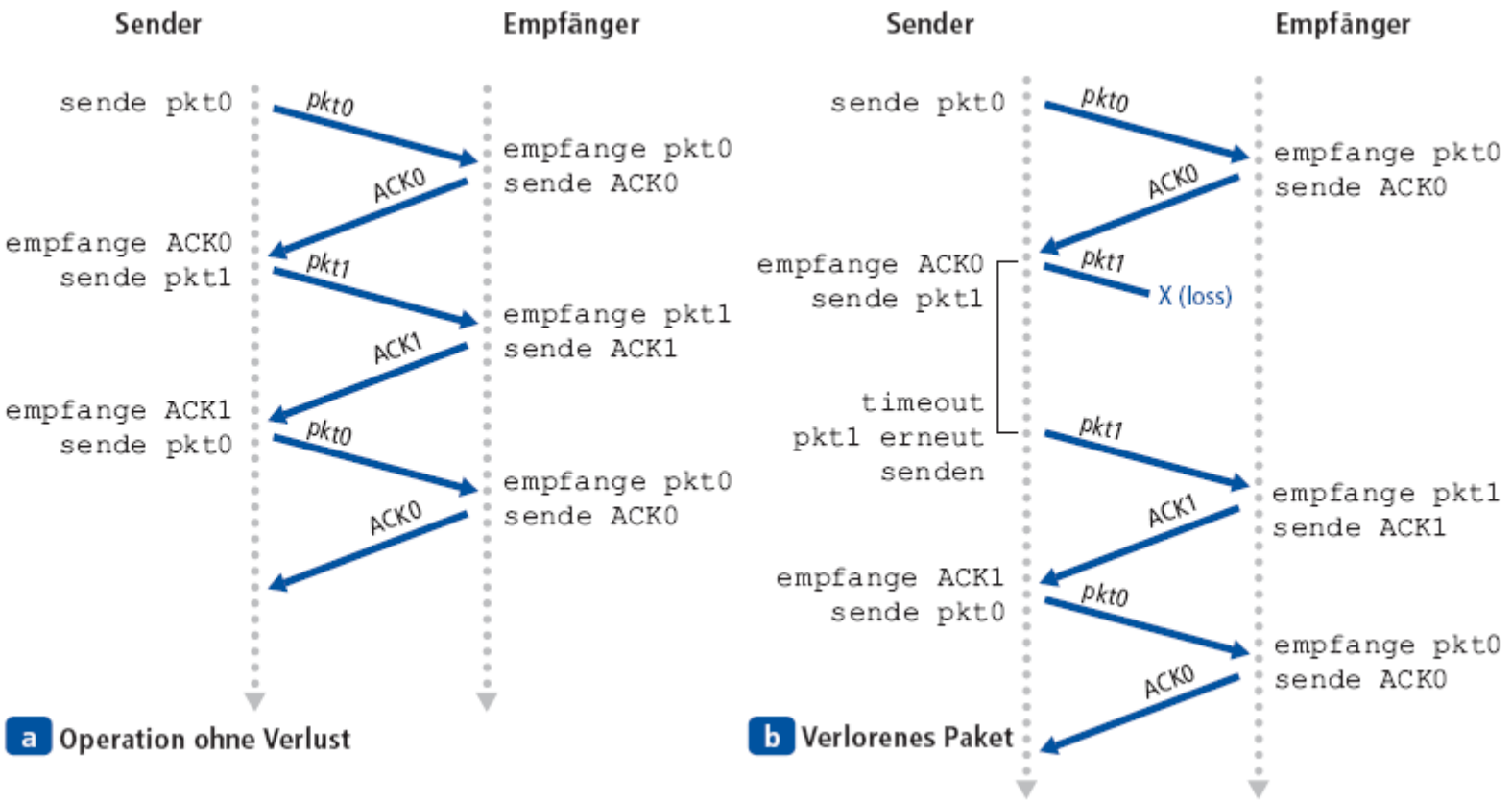
3.4.1 Aufbau eines zuverlässigen Datentransferprotokolls

rdt3.0: Sender mit Wartezeiten



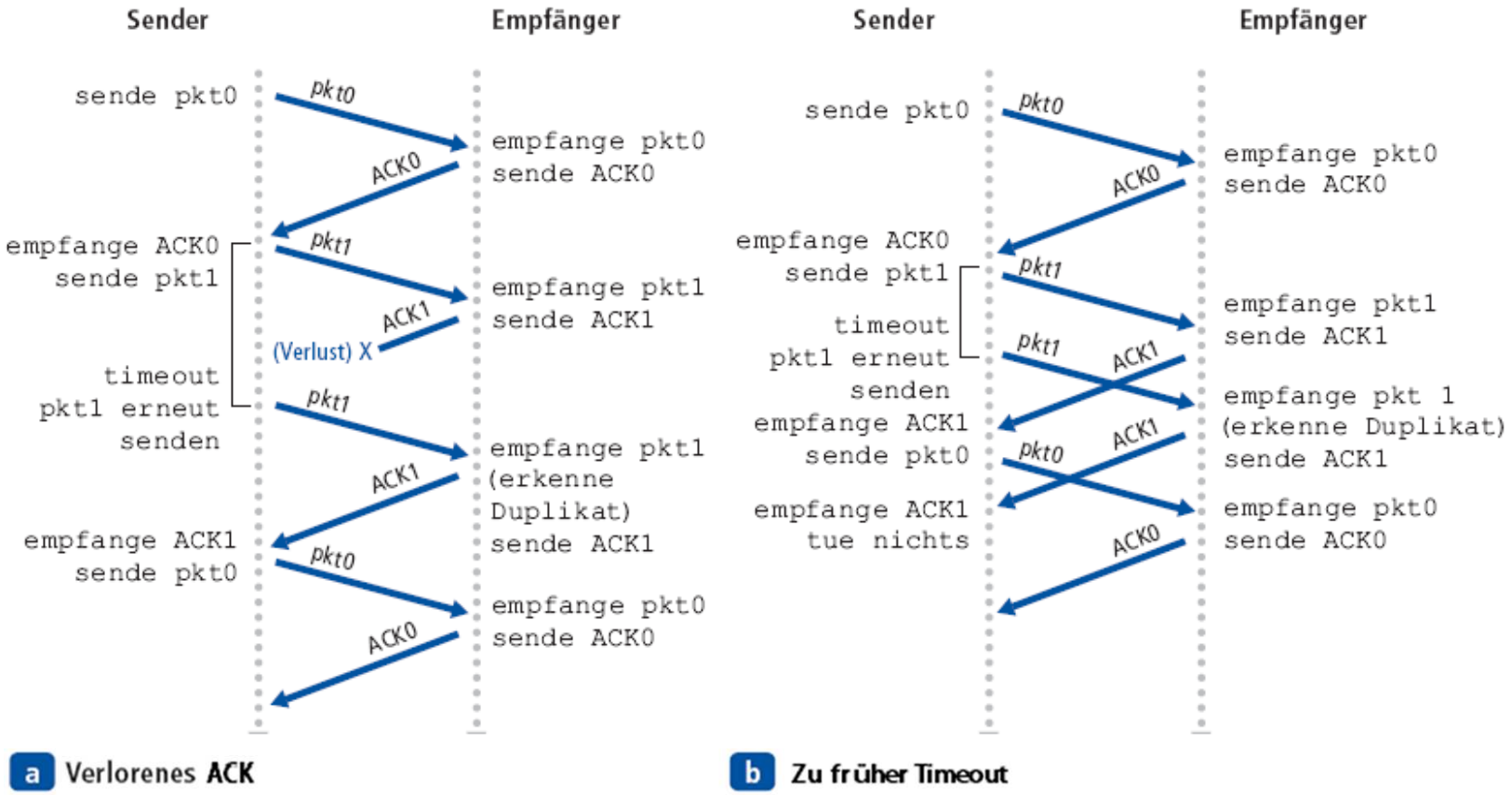
3.4.1 Aufbau eines zuverlässigen Datentransferprotokolls

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

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

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

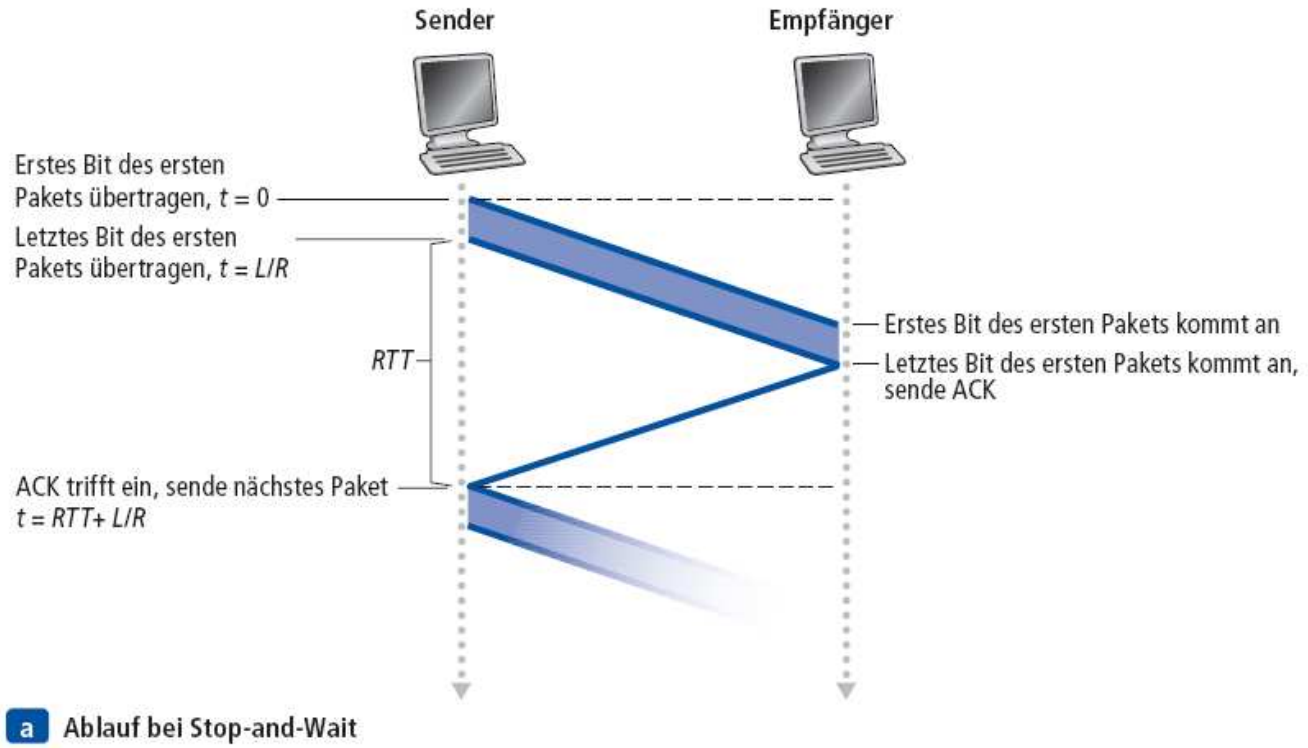
U_{sender}: Utilization (engl. für Auslastung) – Anteil der Zeit, in der tatsächlich gesendet wird

•Einmal 8000 Bit alle ~30 ms -> 33KBit/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

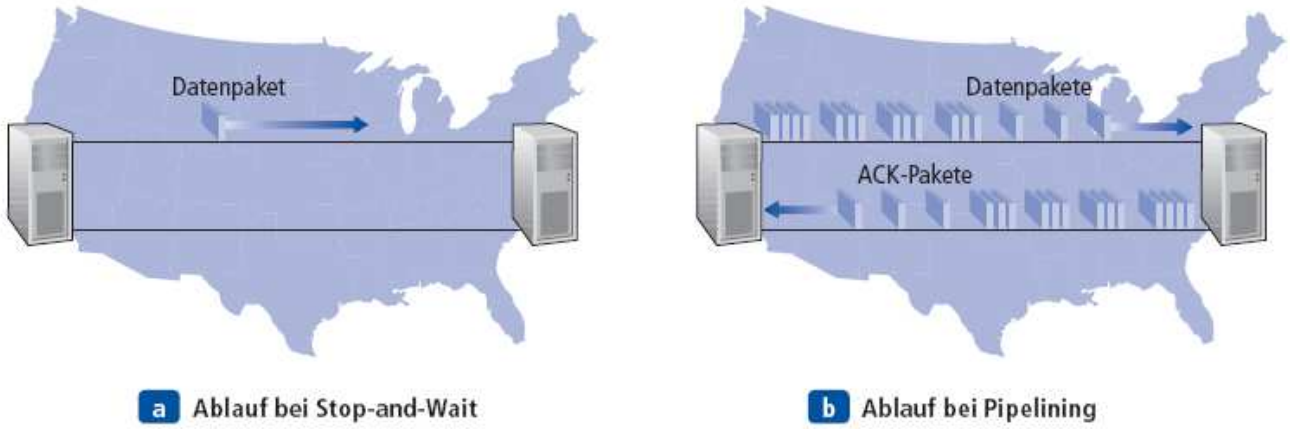


a Ablauf bei Stop-and-Wait

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

3.4.2 Zuverlässige Datentransferprotokolle mit Pipelining

Der Kern des Leistungsproblems mit rtd3.0 liegt darin, daß es ein Stop-and-Wait-Protokoll ist. Stop-and-Wait im Vergleich zu Pipelining:

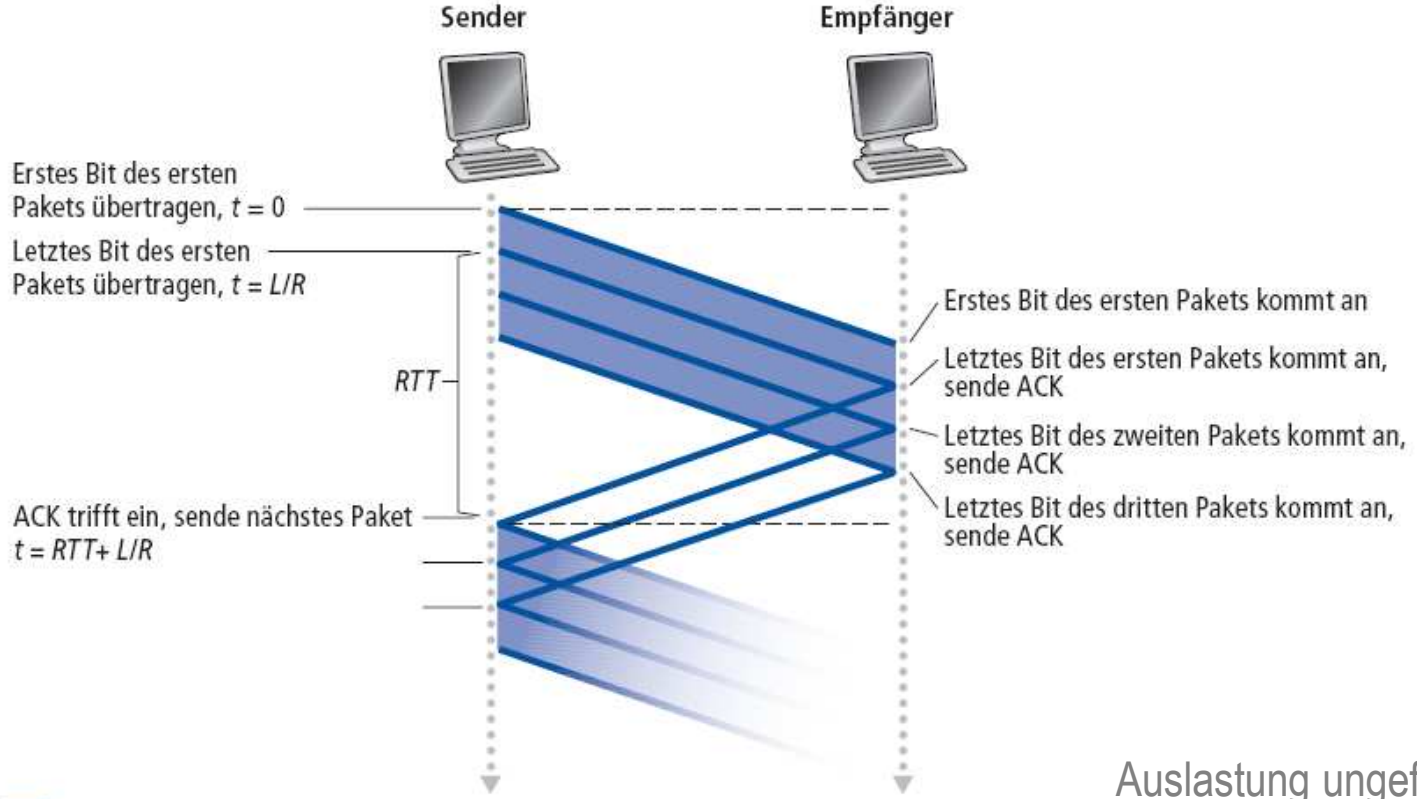


Pipelining: Sender lässt nicht nur eines, sondern mehrere unbestätigte Pakete zu

- ► Die Anzahl der Sequenznummern muss erhöht werden
- ► Pakete müssen beim Sender und/oder Empfänger gepuffert werden

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

3.4.2 Zuverlässige Datentransferprotokolle mit Pipelining



b Ablauf bei Pipelining

Auslastung ungefähr um Faktor 3 erhöht!

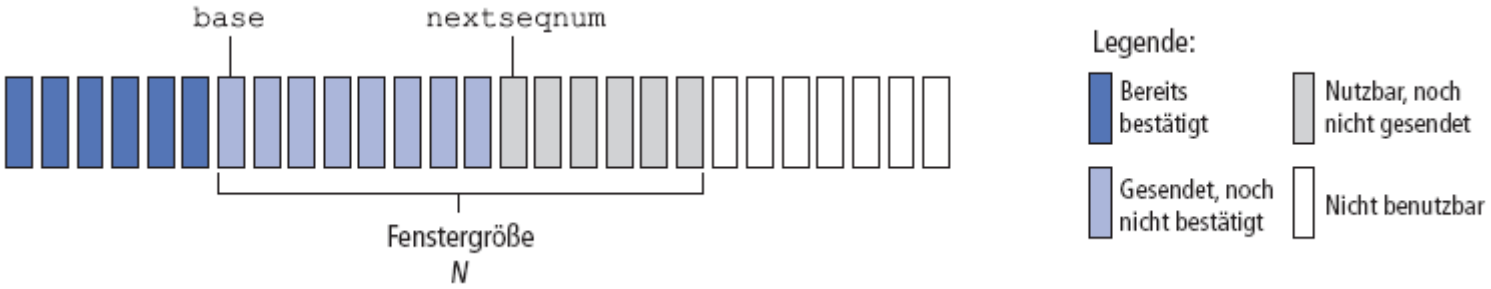
$$U_{sender} = \frac{3 * L / R}{RTT + L / R} = \frac{.024}{30.008} = 0.0008$$

3.4.2 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 Sender bei Timeout	Alle Pakete im Fenster erneut senden	Nur ein einzelnes Paket
Fenster notwendig	Nur bei Sender	Bei Sender und Empfänger
Puffer beim Empfänger	Nicht notwendig	Notwendig
Implementation	Einfach	Komplex
Statusvariablen	Weniger	Mehr
Effizienz	Schlechter	Besser

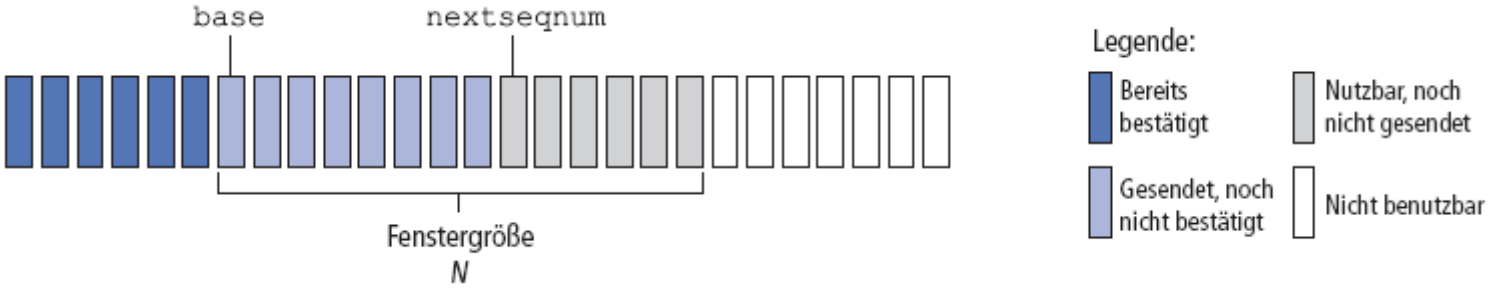
3.4.3 Go-Back-N (GBN)

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)
- Ein **Timer** für jedes 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.

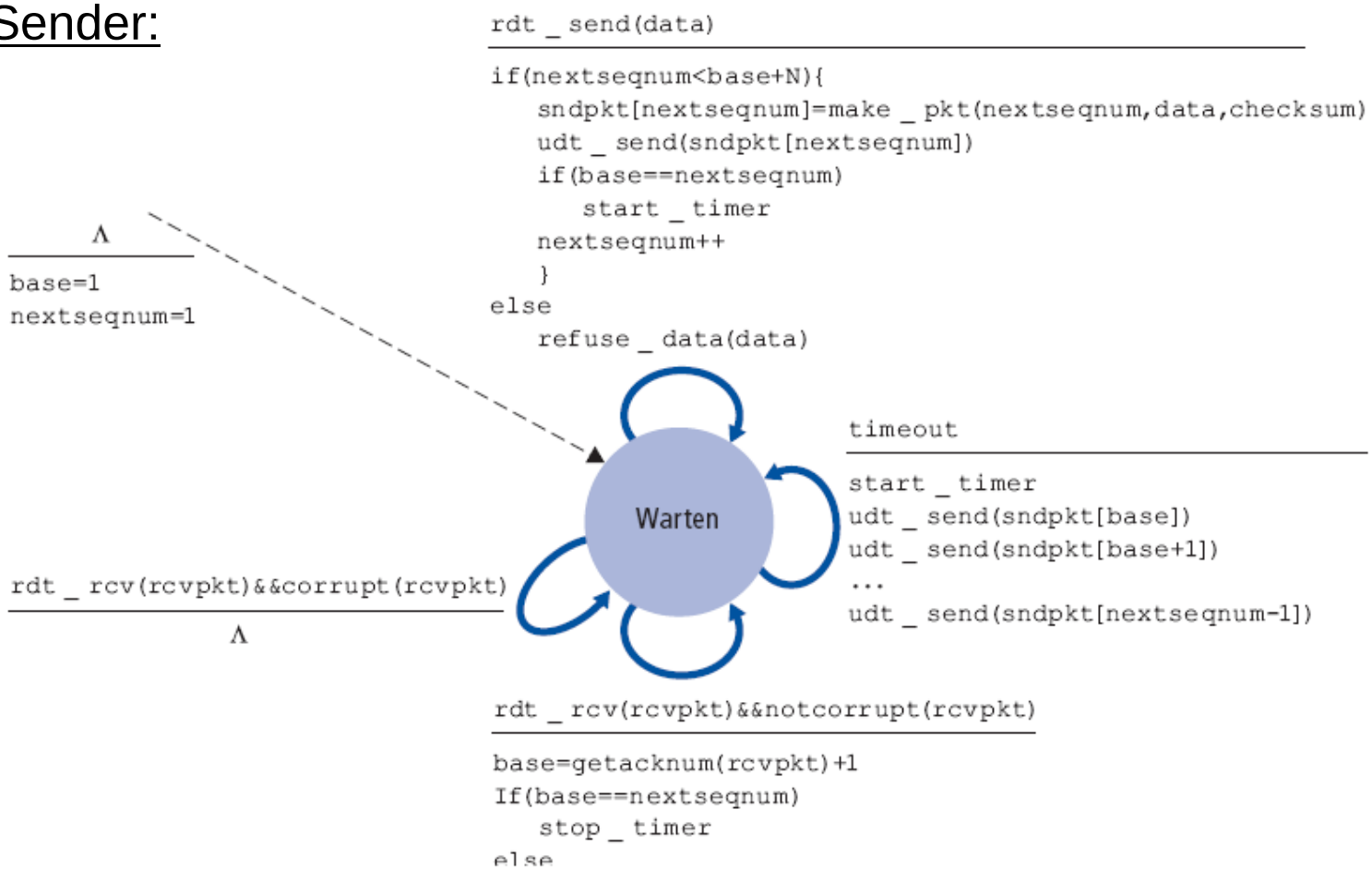
Gesendet, noch nicht bestätigt: **[`base, nextseqnum-1`]** umfasst Pakete, die zwar gesendet wurden, aber noch nicht bestätigt worden sind.

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)

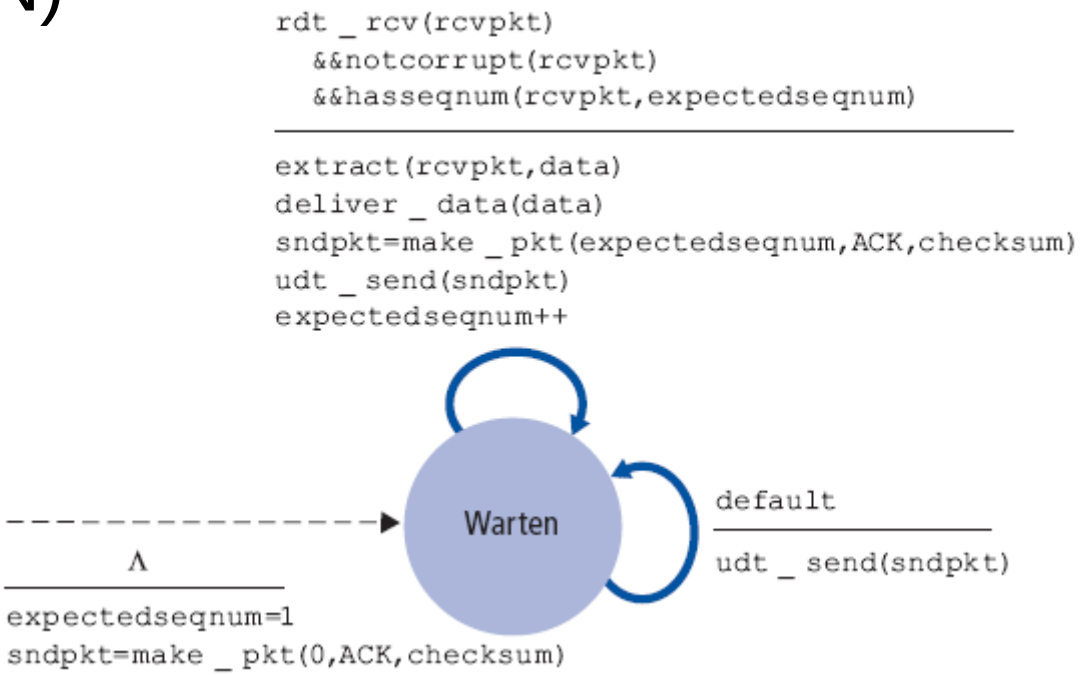
GBN Sender:



start_timer

3.4.3 Go-Back-N (GBN)

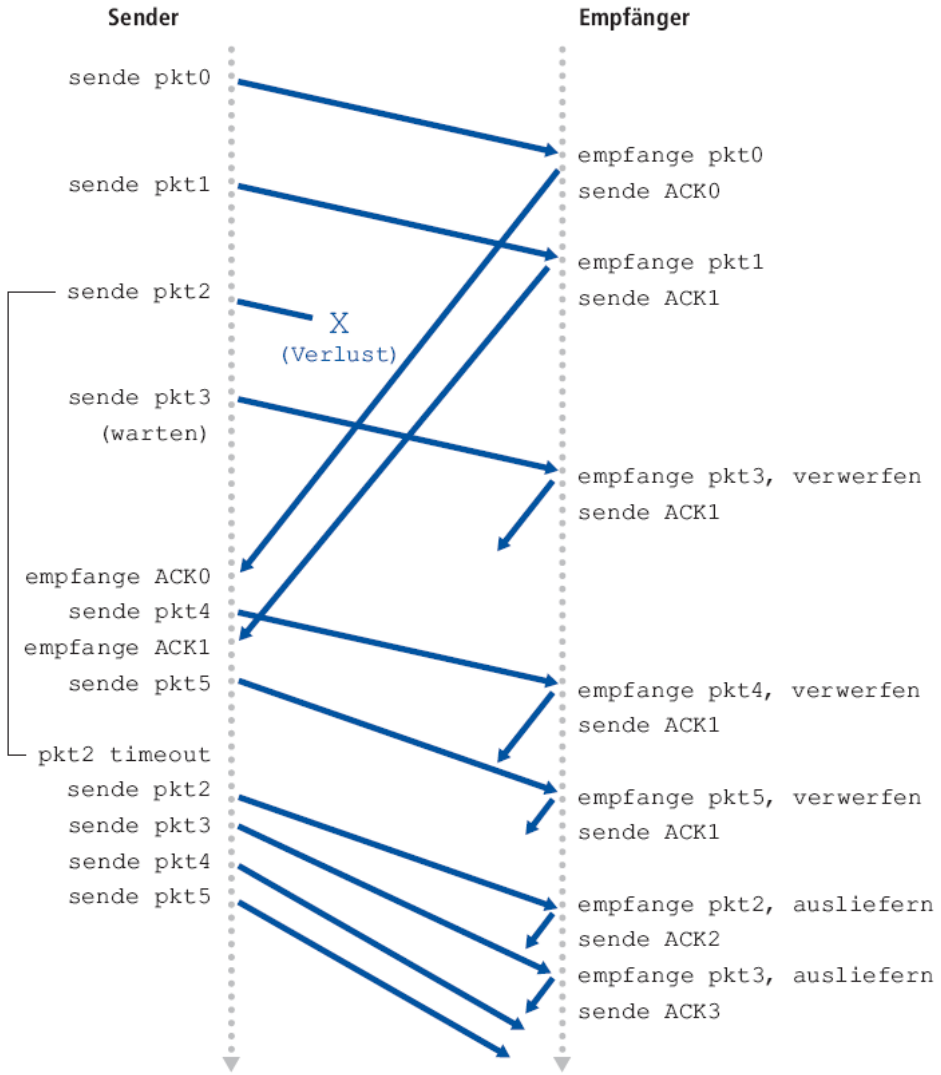
GBN Empfänger:



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

3.4.3 Go-Back-N (GBN)

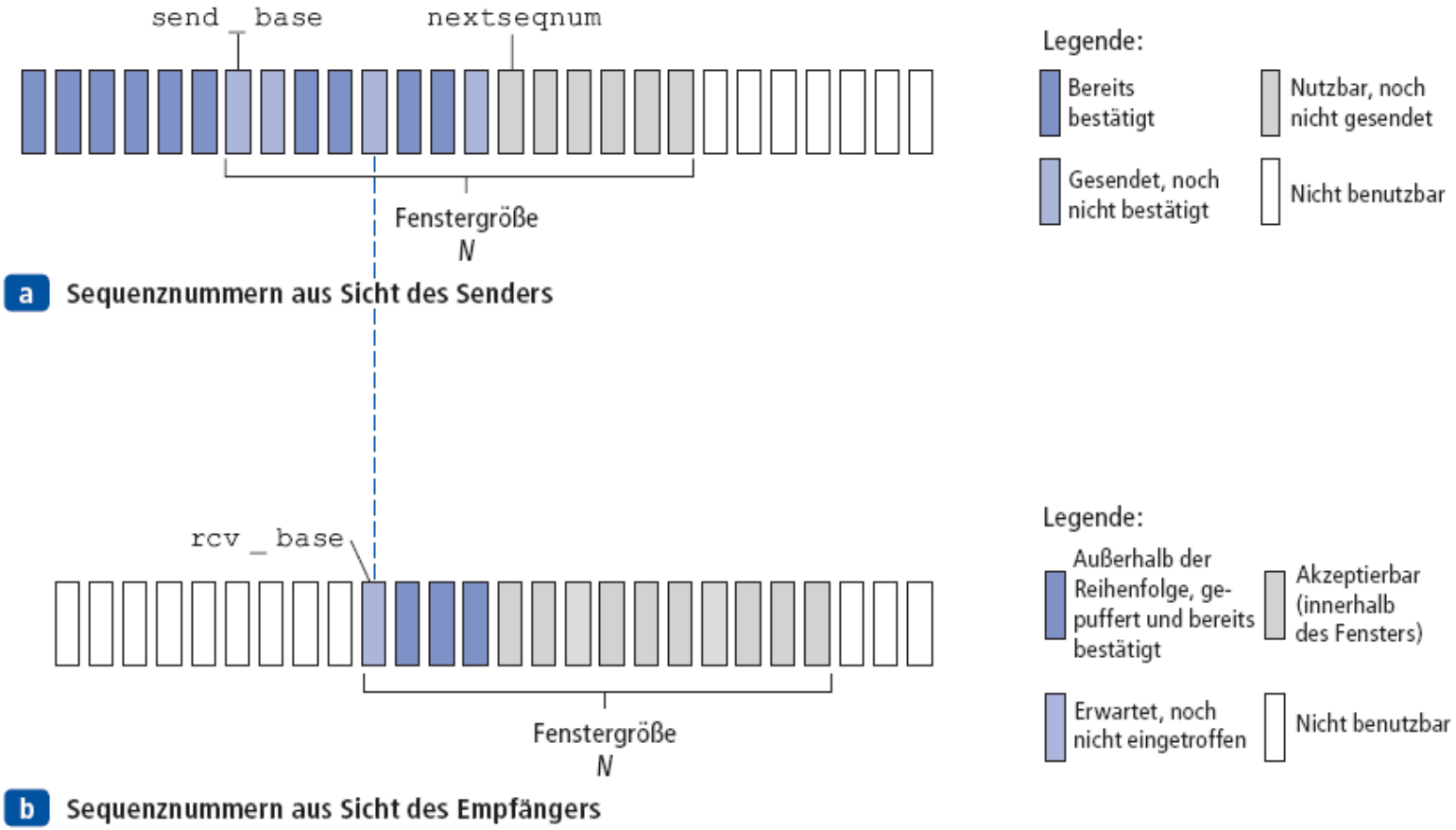
GBN Ablauf:



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)



3.4.4 Selective Repeat (SR)

Sender:

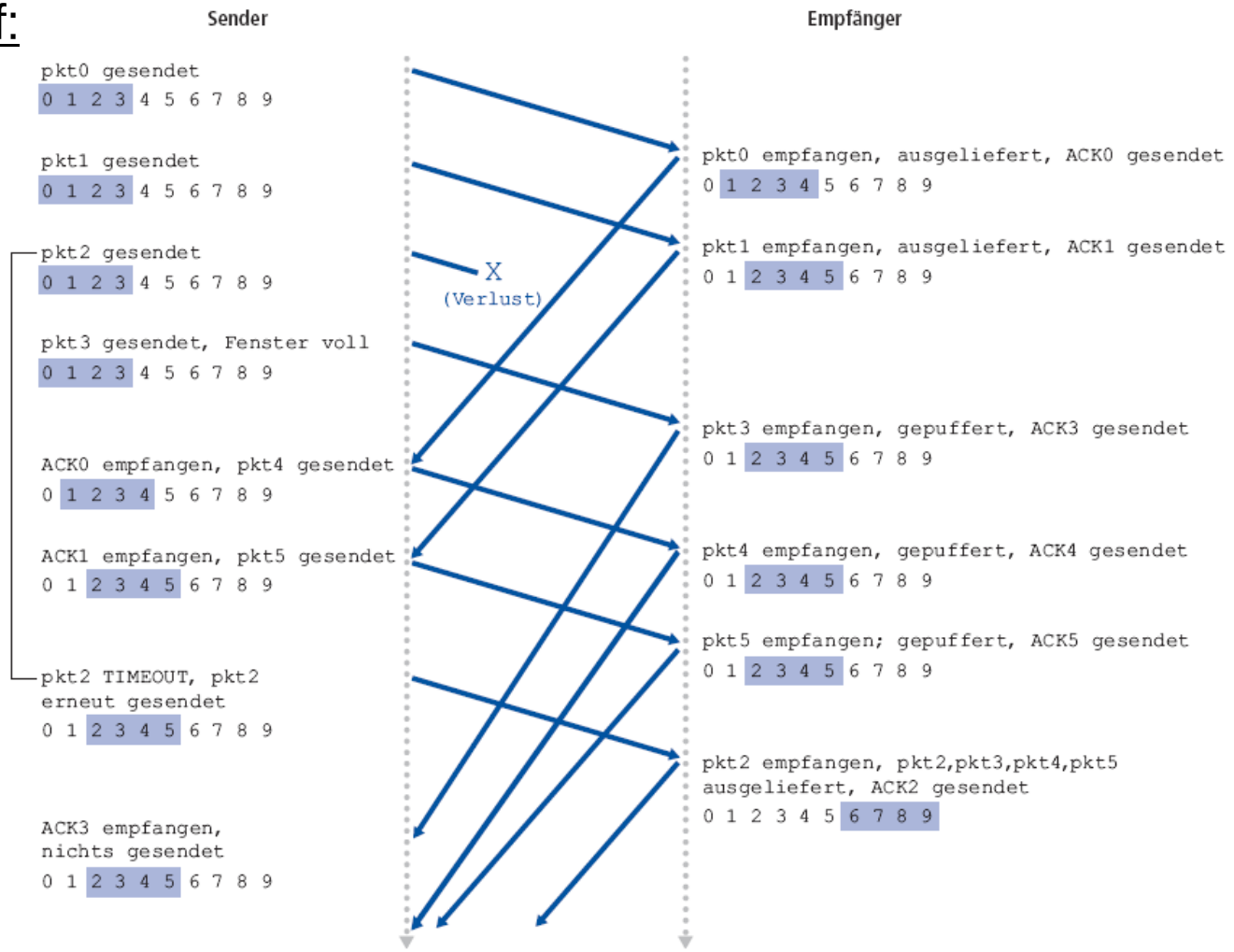
- **Daten von oben**:
  - Wenn nächste Sequenznummer im Fenster liegt: Paket senden, Timer(n) starten!
- **Timeout(n):**
  - Paket n erneut übertragen, Timer(n) starten
- **ACK(n) aus** [sendbase,sendbase+N]
  - Paket n als empfangen markieren
  - Wenn n die kleinste unbestätigte Sequenznummer ist, Fenster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

Empfänger:

- **Paket aus** [rcvbase, rcvbase+N-1]
  - Sende ACK(n)
  - Außer der Reihe: Puffern
  - In der Reihe: Ausliefern (auch alle gepufferten 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!