Betriebssysteme und Netzwerke Vorlesung N04

Artur Andrzejak

TCP Grundlagen

Transmission Control Protocol - Überblick

- **Point-to-Point**
  - 1 Sender, 1 Empfänger
- Zuverlässiger **Datenstrom**
  - Zustellung garantiert
  - Es gibt keine “Grenzen“ der Nachrichten
- **Pipelined**
  - Daten kommen in der Reihenfolge des Sendens an
- **Vollduplex**
  - Bi-direktionaler Datenfluss
- **Verbindungsorientiert**
  - Handshaking initialisiert den Zustand des Senders / Empfängers vor dem Datentransfer
- **Flusskontrolle**
  - Der Sender wird den Empfänger nicht überfluten
- **Überlaststeuerung/ Staukontrolle**
  - Sender passt sich an die (aktuelle) Bandbreite der Leitung an

Struktur eines TCP-Segmentes

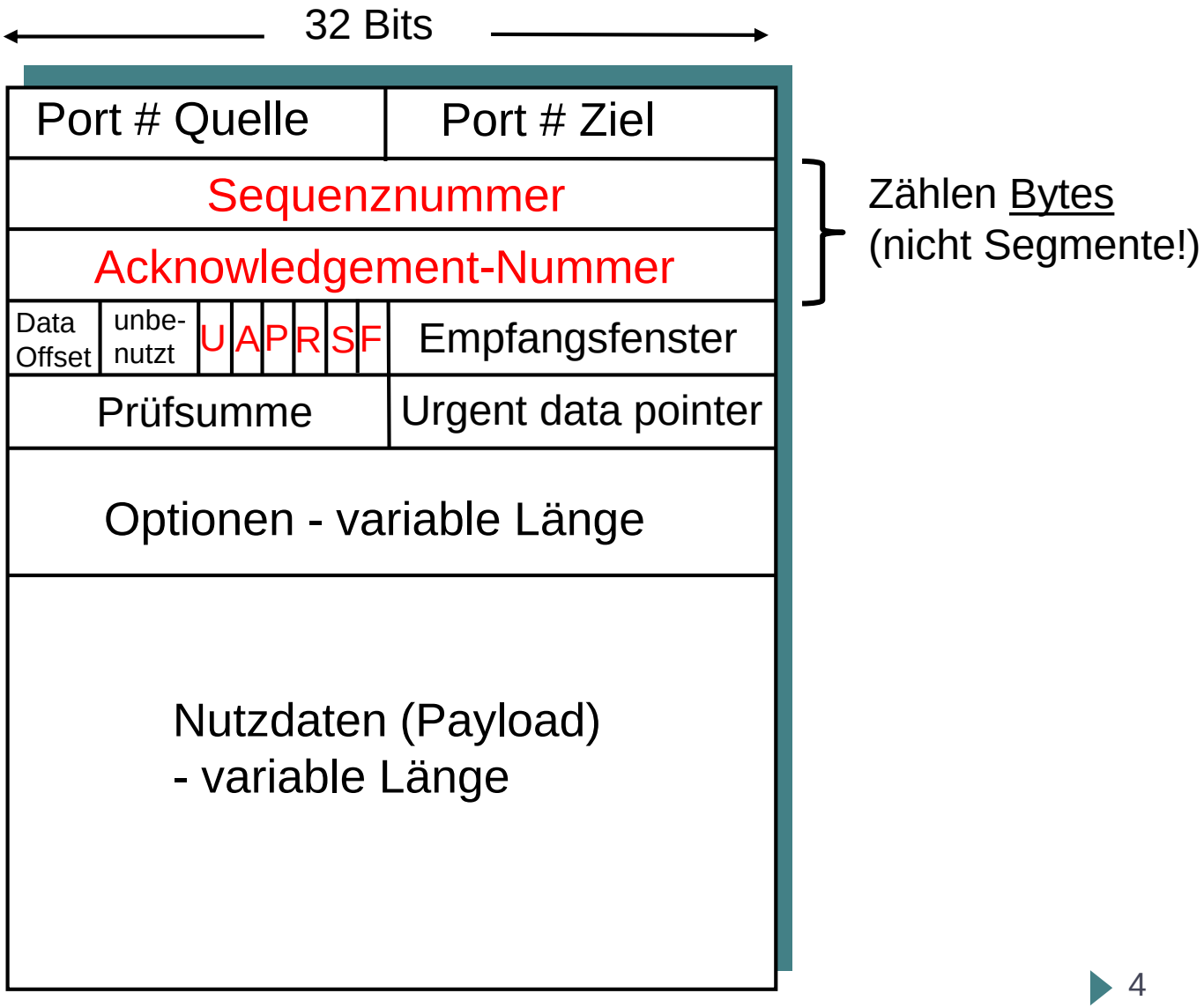
Flags:

U: Urgent Data

A: ist ACK# gültig?

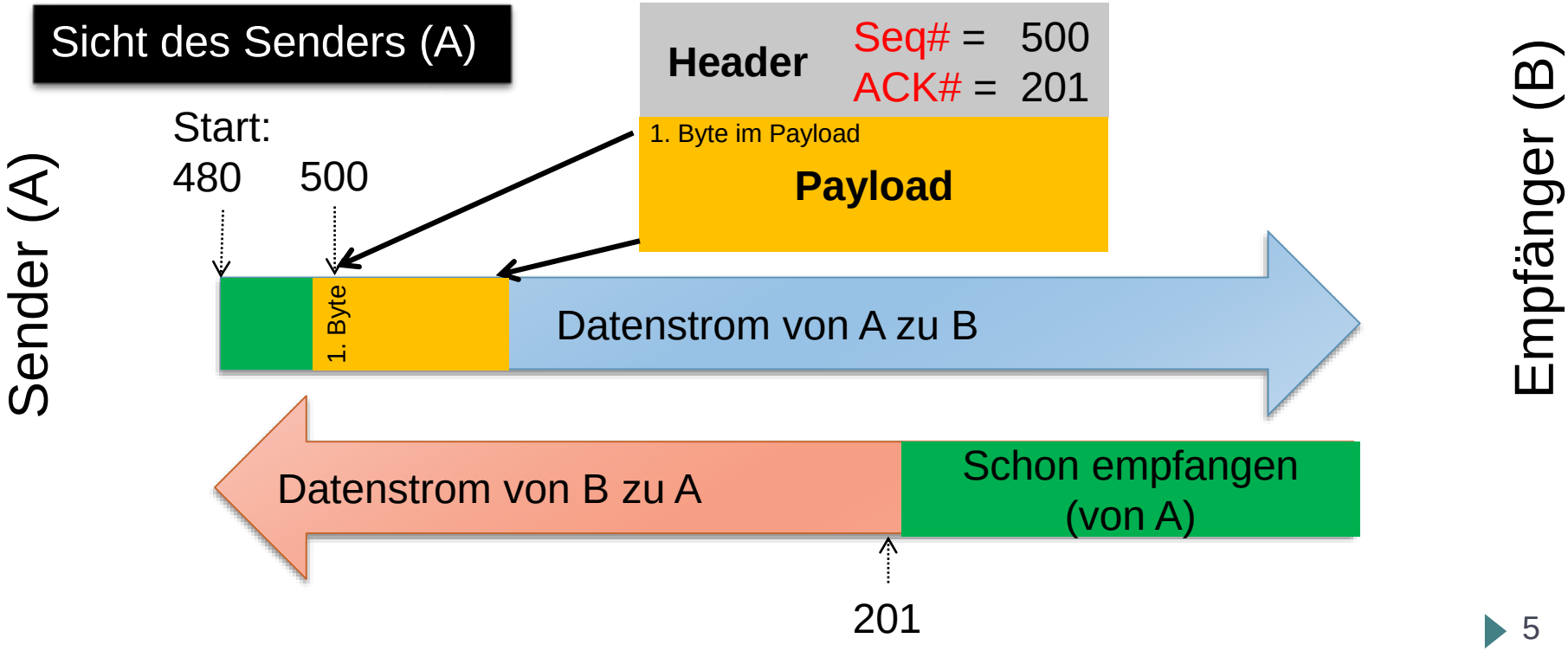
P(SH): „push data now“

R(ST), **S**(YN), **F**(IN): für Aufbau und Schließen der Verbindung



Sequenznummer und ACK-Nummer /1

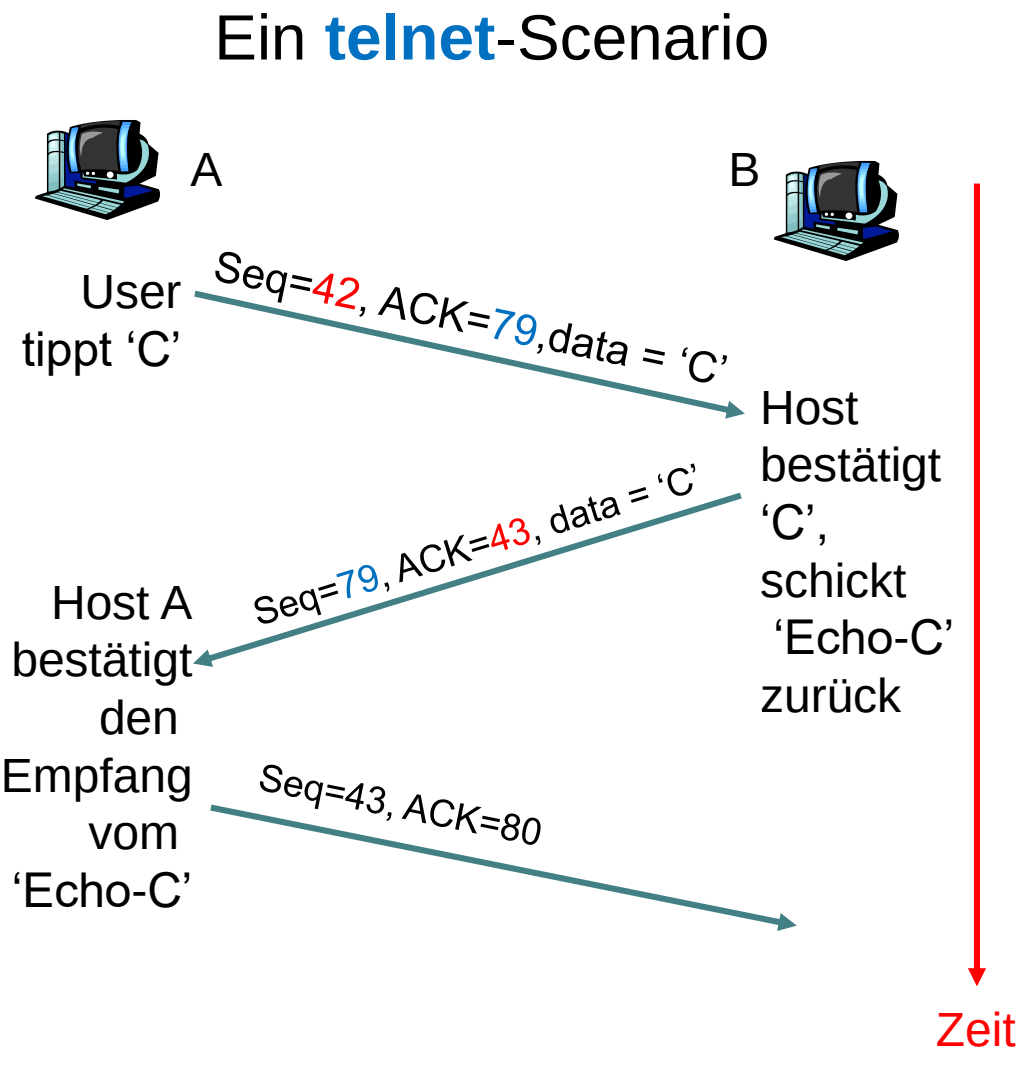
- Sequenznummer: Index in dem Datenstrom (vom Sender zum Empfänger) des 1. Bytes des Paketinhalts (Payload)
- Acknowledgement-Nummer (ACK-Nummer): Index des nächsten von A erwarteten Bytes (in dem ”Antwort”-Datenstrom, d.h. B zu A)



TCP-Sequenz- und ACK-Nummern /2

- **Sequenznummer**: „Daten-strom-Index“ des 1. Bytes des Payloads im Paket vom Sender A zum Empfänger B
- **ACK-Nummer**: „Daten-strom-Index“ des nächsten von A erwarteten Bytes
- Funtionen von Seq# / ACK#?

- Seq#: Notwendig, um die Pakete beim Empfänger in die richtige Reihenfolge zu bringen
- ACK#: Zeigen dem Sender, dass Daten angekommen sind und ggf. welche nochmals geschickt werden müssen (später mehr dazu)



TCP-Header: Ausgewählte Felder

- Data Offset: Länge des Headers in 32-Bit-Blöcken
- Empfangsfenster (bzw. Receive-Window)
  - Anzahl der Datenbytes, die der Sender dieses TCP-Paketes bereit ist (als Antwort) zu empfangen – für Flusskontrolle
- Prüfsumme über den Header, die Daten und einen Pseudo-Header
  - Der Pseudo-Header besteht aus der Ziel-IP, der Quell-IP, der TCP-Protokollkennung (6) und der Länge des TCP-Headers inkl. Nutzdaten (in Bytes)
- Urgent-Pointer
  - Benutzt, um „drigendende“ Daten (z.B. Ctrl-C) außerhalb des Datenstroms einzufügen (falls der URG-Flag gesetzt ist)
  - In diesem Fall fangen die eingeschobenen Daten direkt nach dem Header bis zur Position (im Payload), auf die der Urgent-Pointer zeigt (der Datenstrom fängt im Payload danach an)
  - Es wird kaum verwendet

TCP-Header: Flags

- URG: Zeigt vorhandensein von ”Urgend”-Daten an
- ACK: Gibt an, ob Acknowledgment-Nummer gültig ist
- PSH: „Push Data“ – Auch kleinere Datenmengen werden als eigene Pakete verschickt bzw. an Applikation sofort ausgeliefert (d.h. umgeht das Zusammenfassen von kleineren Übertragungen in einem Paket) – z.B. für telnet
- RST: Wird gesetzt, um die Verbindung abzubrechen oder abzuweisen
- SYN: Paket mit gesetztem SYN-Flag initiieren eine Verbindung; Antwort mit SYN+ACK oder RST
- FIN: „Finish“ – Zeigt an, dass keine Daten mehr vom Sender kommen

TCP – Protokoll: Verbindungsverwaltung

TCP Verbindungsaufbau

Merke: SEQ# und ACK# spielen anfangs eine andere Rolle als später!

- Es wird eine (virtuelle) Verbindung aufgebaut (nur an den Hosts, nicht an den Routern) durch Initialisierung der TCP-Variablen
  - Sequenznummer (Seq#)
  - Puffer und Variablen der Flusskontrolle
- Client ist die Initiator-Seite:

  Socket c=new Socket („host", port#);
- Server reagiert durch das Erzeugen eines neuen Sockets:

  Socket connectionSocket =
  welcomeSocket.accept();

Drei-Wege-Handshake:

1: Client sendet ein **SYN-Segment**
   - Seq# := zufällige Zahl (**client_isn**), SYN-Bit := 1
   - Keine Daten
2: Server antwortet mit **SYNACK**
   - SYN-Bit := 1; ACK# := **client_isn**+1; Seq# := zufällige Zahl (**server_isn**)
   - Server belegt Puffer
3: Client emfängt SYNACK und antwortet mit ACK-Segment
   - ACK := **server_isn**+1
   - Segment kann Daten haben

TCP Verbindungsabbau /1

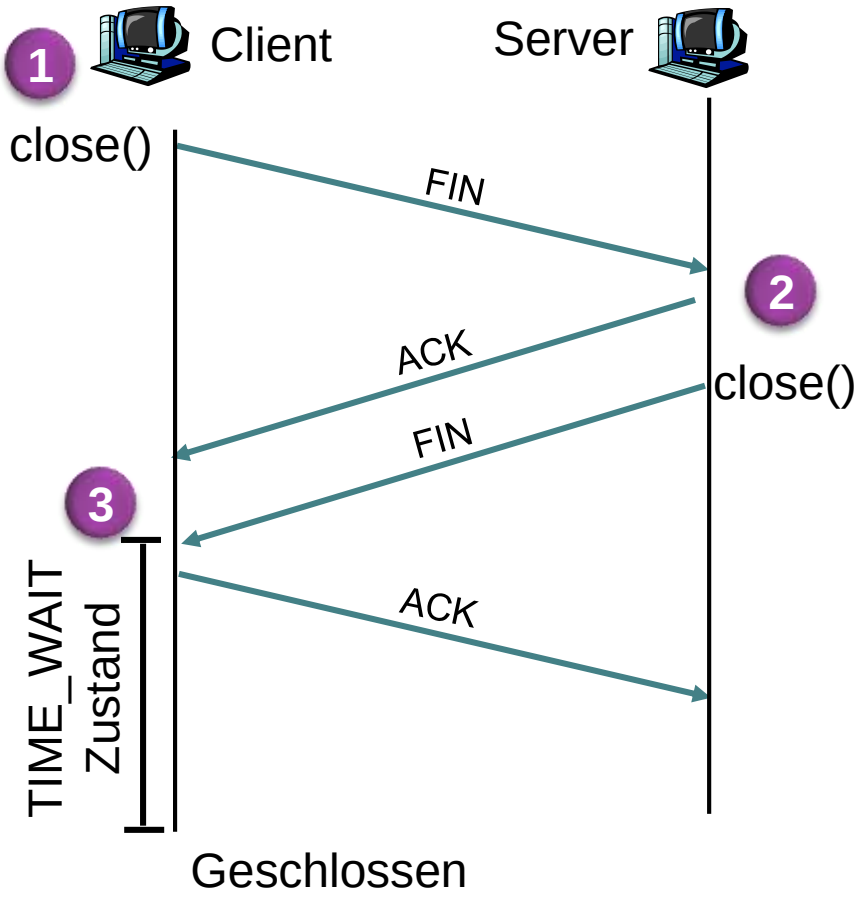
Anwendungsschicht:

- Client schließt den Socket via `clientSocket.close();`

Netzwerkschicht:

1: Client schickt FIN-Segment an den Server (FIN-Bit im Header gesetzt)

2: Server empfängt FIN, antwortet mit ACK; dann schließt er die Verbindung, sendet auch FIN



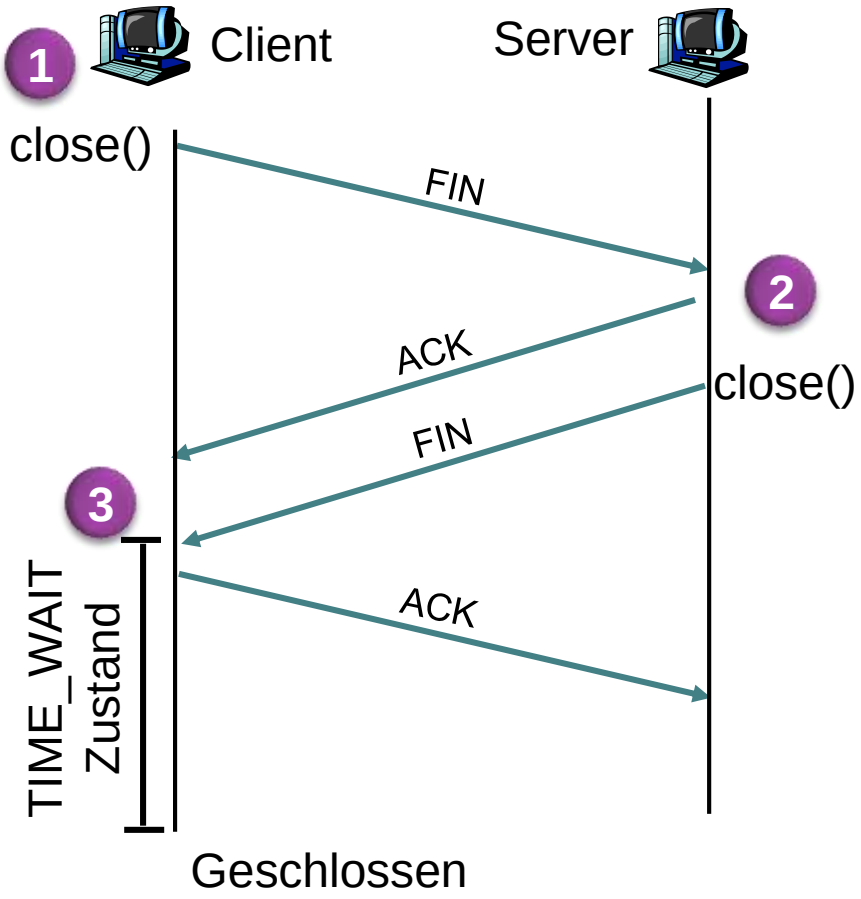
TCP Verbindungsabbau /2

Netzwerkschicht (Forts.):

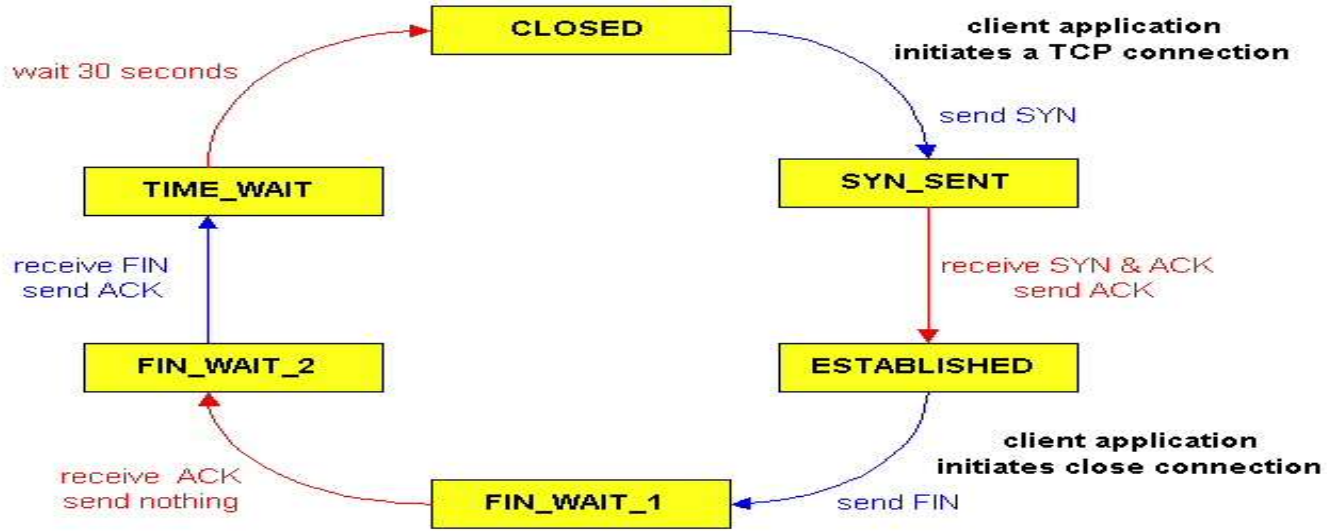
3: Client empfängt FIN-Segment, sendet ACK an den Server

- Geht in den Wartezustand TIME_WAIT über: 30 Sek – 2 Min

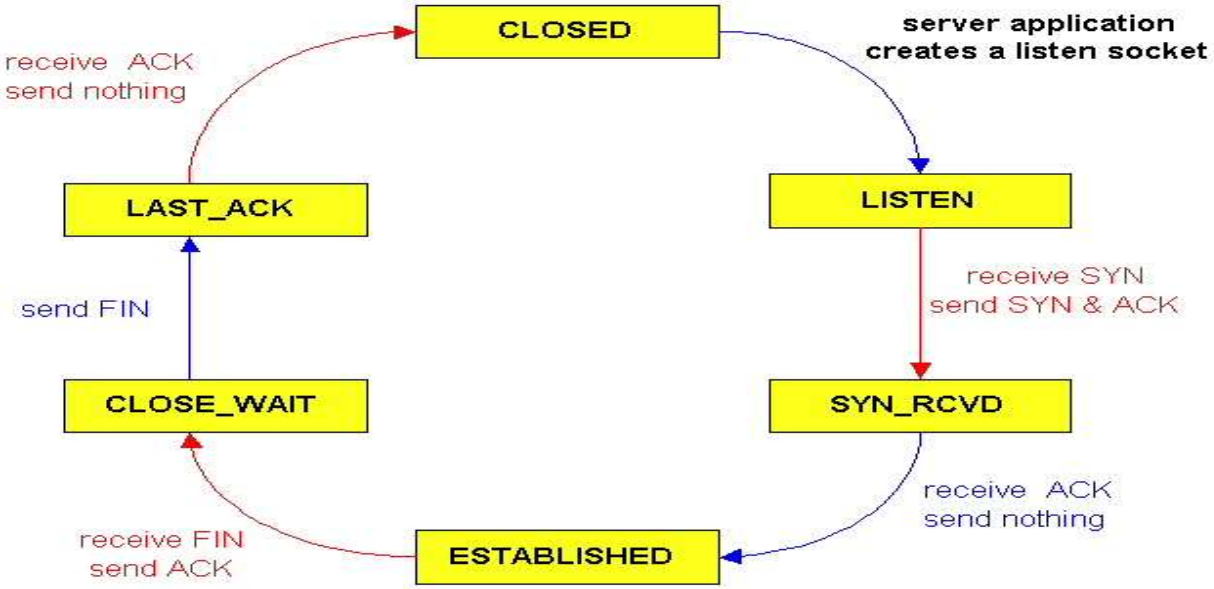
- Danach ist die Verbindung endgültig geschlossen
  - Alle Ressourcen (inkl. Portnummer) werden freigegeben



Zustände einer TCP-Verbindung



Client oder Server?



Client oder Server?

Videos

- TCP connection walkthrough Networking tutorial (13 of 13) [N04d]
  - https://www.youtube.com/watch?v=F27PLin3TV0
  - Ab 7:05 bis Ende (min:sec)
- The TCP Connection (siehe Abschnittsanfang)
  - https://www.youtube.com/watch?v=uBMb83rlJrQ&index=7
  - Verbindungsaufbau, Ansatz Verlässlichkeit, Verbindungsabbau
- internet tcp/ip in work, networking, data transfer
  - Allgemeines Video zu TCP/IP
  - Animation - witzig, aber wenig Details
    - https://www.youtube.com/watch?v=E-w4ybYrtTQ

TCP-Verbindungsverwaltung: Sonderfälle

- Was passiert, wenn der Server keine Verbindung aufbauen möchte / kann? (Ursachen?)
  - Server antwortet mit einem **RST-Segment**, d.h. Segment bei dem der RST-Bit gesetzt ist
- **Port-Scanning**: Angreifer senden ein SYN-Segment nacheinander an (fast) alle Ports x eines Zielhosts
  - SYNACK als Antwort: Port x geöffnet, hier weiter „bohren“
  - RST als Antwort: Port x geschlossen, aber keine Firewall
  - Keine Antwort: es gibt eine Firewall
- **SYN-Flood**-Angriff
  - Ein **Distributed Denial of Service** Angriff (**DDoS**-Angriff)
  - Server reserviert unnötig viele Ressourcen (infolge von empfangenen SYN-Segmenten) und kann bald keine legitimen Verbindungsversuche bedienen

Drei Große Herausforderungen bei TCP

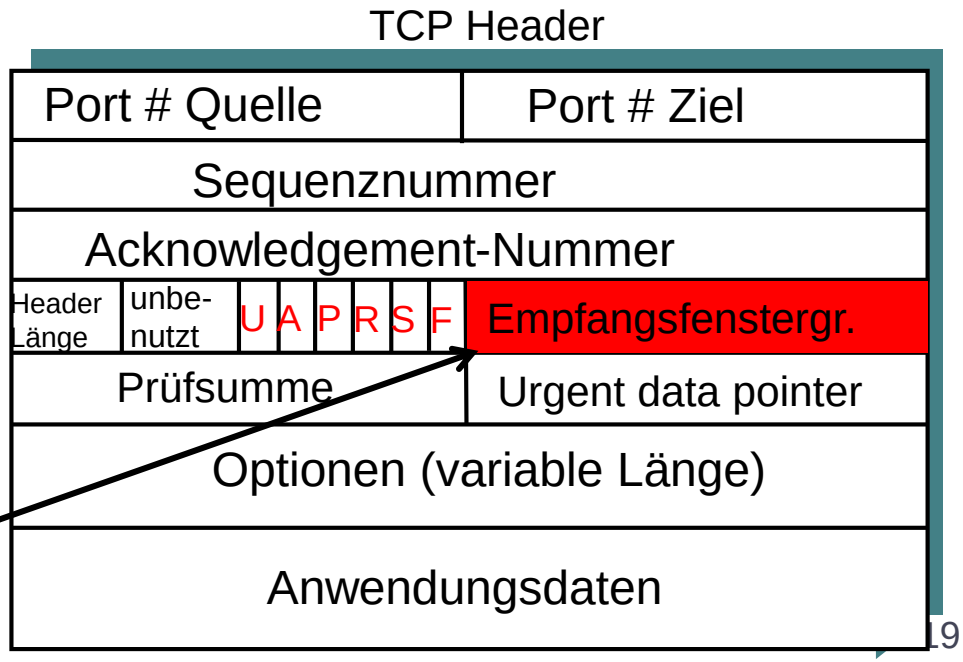
- Wie schafft man es, dass der Sender dem Empfänger nicht mehr Daten schickt, als dieser gerade verarbeiten kann? => Flusskontrolle
- Wie schafft man es, dass verlorene oder doppelte Pakete erkannt werden und die Daten in richtiger Reihenfolge an die Anwendung ausgeliefert werden?
  - => Verlässliche aber effiziente Nachrichtenzustellung, insbesondere Sliding-Window-Verfahren
- Wie erkennt man, wie viele Daten das Netzwerk gerade übermitteln kann und passt die Sendegeschwindigkeit an?
  - => Überlaststeuerung/ Staukontrolle

TCP – Protokoll: Flusssteuerung

Flusssteuerung:
Verhindert das Überfüllen der Puffer

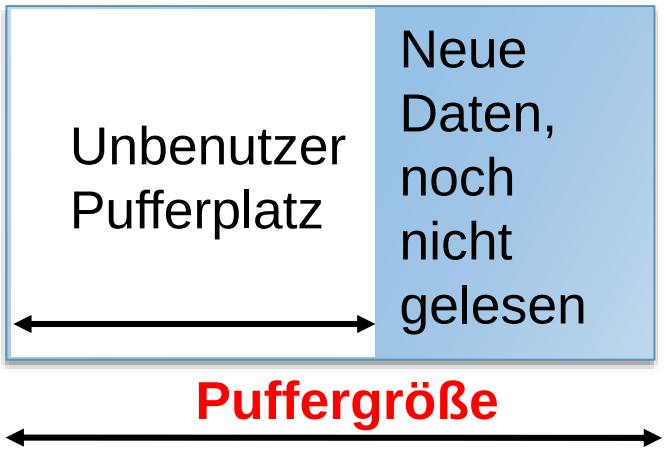
- Hosts auf jeder Seite der Verbindung reservieren einen Eingangspuffer
  - Die Anwendungsschicht holt Daten von diesem Puffer ab
- Dies kann aber unregelmäßig geschehen
  - Dann könnte der Pufferplatz ausgehen, und Pakete würden verlorengehen

- Lösung: Host A teilt dem anderen in Header der TCP-Segmente mit, wie viel Platz er (d.h. A) noch im Puffer hat
- Dazu dient die Variable **RcvWindow = Empfangsfenstergröße**



Flusssteuerung /2

- Angenommen: Host **A** schickt eine Nachricht an Host B und teilt B (im Header) den Wert seine Empfangsfenster-Größe RcvWindow (**A**) mit
- Der Host B sendet dann maximal RcvWindow (**A**) viele noch unbestätigte Bytes an den Host **A**
  - Dieses Verfahren ist symmetrisch (A und B vertauscht)
- Wie berechnet ein Host seinen RcvWindow-Wert?
- RcwWindow := (Puffergröße) – (Anzahl der Daten im Puffer, die noch nicht gelesen wurden)

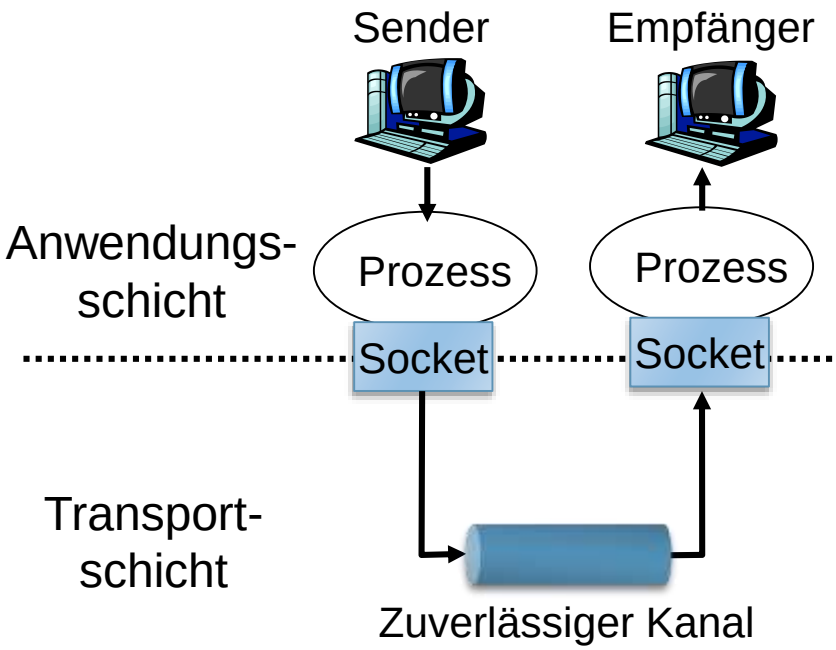


Mehr: http://de.wikipedia.org/wiki/Transmission_Control_Protocol

Verlässliche Nachrichtenzustellung: Naive Implementierung

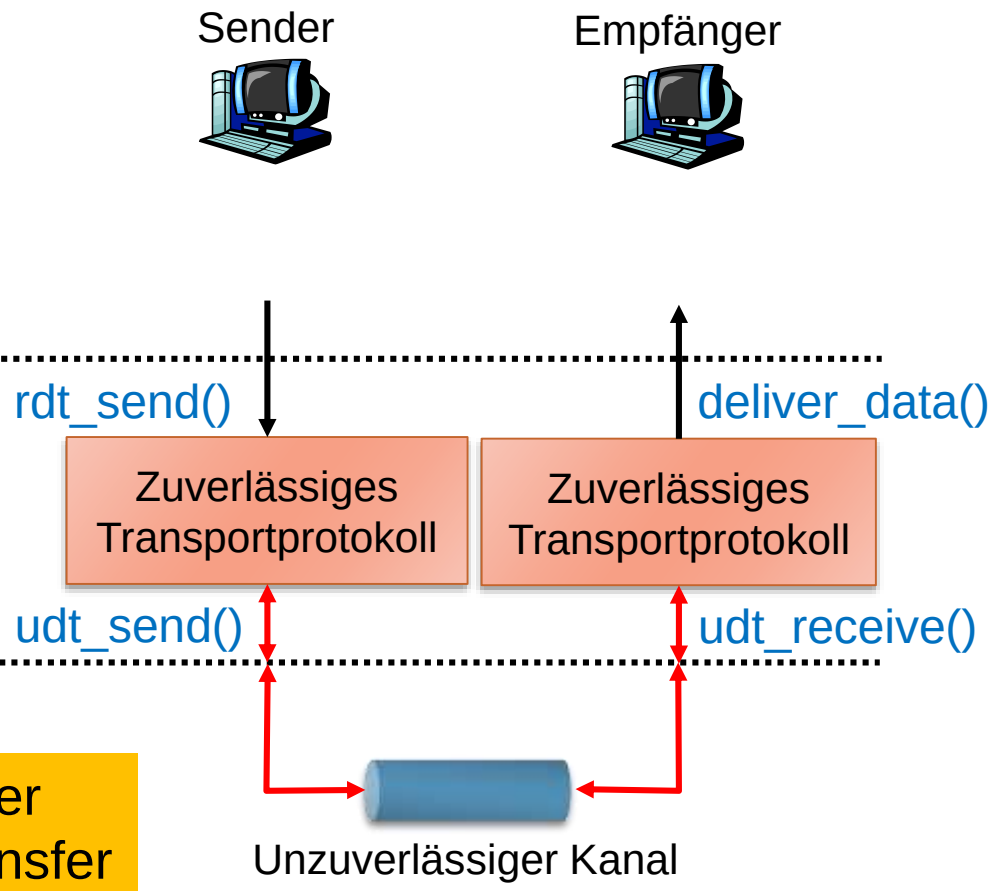
Dienstmodell und Diensimplementierung

Dienstmodell



rdt = reliable data transfer
udt = unreliable data transfer

Dienstimplementierung



Fehlermodell

- Wir betrachten weiterhin das Protokoll TCP
  - Insbesondere: Wir haben bei jedem Paket die Sequenznummer und ACK-Nummer
- Wir brauchen zunächst Annahmen, welche Fehler auftreten können - ein **Fehlermodell**
- Wir nehmen das worst case Szenario an
  1. Pakete können verlorengehen
  2. Pakete können verfälscht ankommen (Bitfehler)
  3. Pakete können doppelt ankommen
- Wie können wir 2 und 3 behandeln?

Teillösungen

- Unser Fehlermodell
  1. Pakete können verlorengehen
  2. Pakete können verfälscht ankommen (Bitfehler)
  3. Pakete können doppelt ankommen (Folge von 1)
- Lösungen für 2 & 3?
- Zu 2: Bitfehler auf Paketverluste zurückführen
  - Wenn die Prüfsumme nicht stimmt, wird ein Paketverlust angenommen!
- Zu 3: Doppelte Pakete
  - Hier wird jedem Paket eine fortlaufende **Sequenznummer** zugeordnet - der Empfänger kann so Duplikate identifizieren

Es bleibt also nur, Problemtyp 1 (Paketverlust) zu lösen

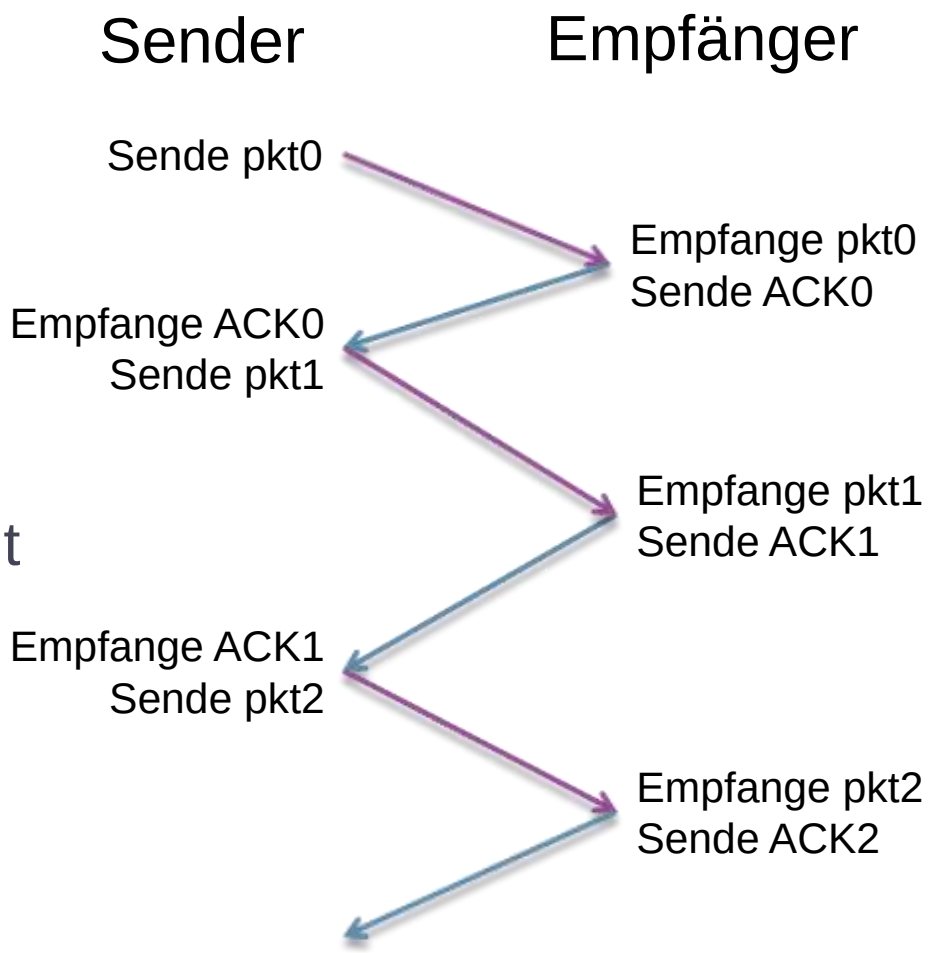
Implementierung – Version **S&W**

Auflösung später

Hauptidee: **Bestätigung** (Acknowledgement) **von korrekten Paketen**

Ablauf:

- Sender schickt ein Paket und startet einen Timer
  - **Sequenznummer x** kommt in den Header
- Empfänger antwortet mit einer **Bestätigung ACK_x** (die auf x verweist)
- Wenn der Sender ACK_x empfängt, schickt er Paket mit Sequenznummer x+1 ...

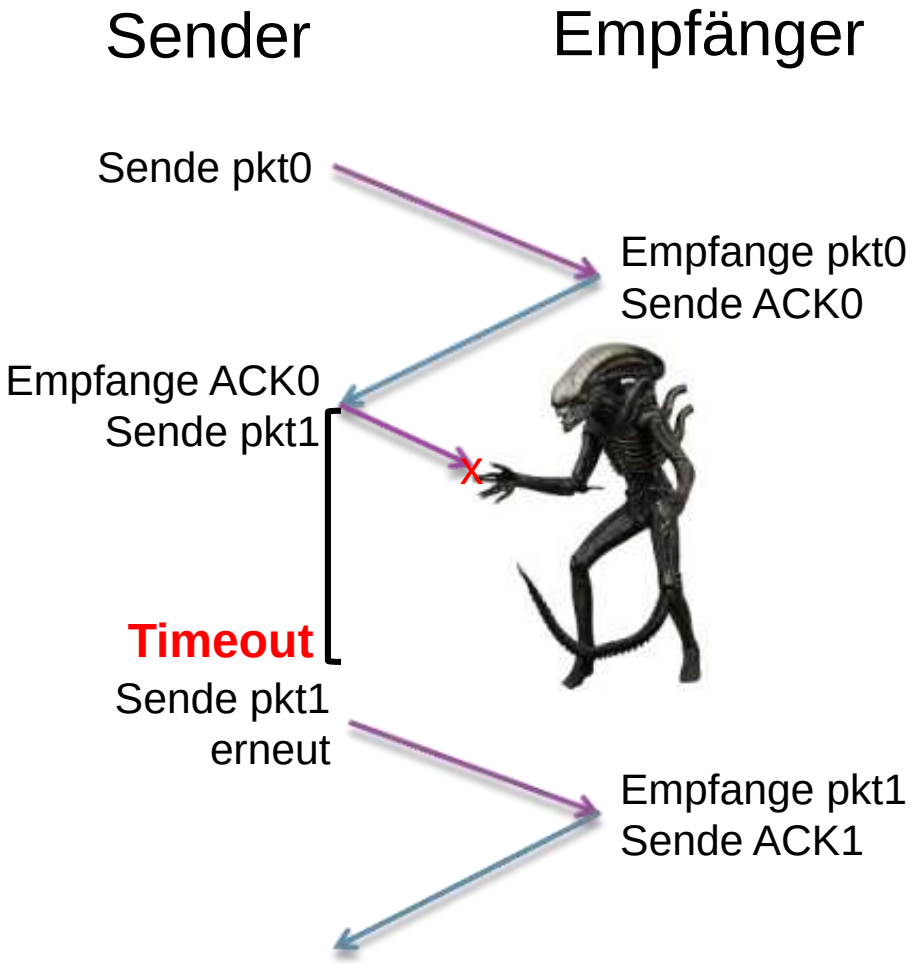


Operation ohne Fehler

Implementierung – Version S&W /2

Paketverlust

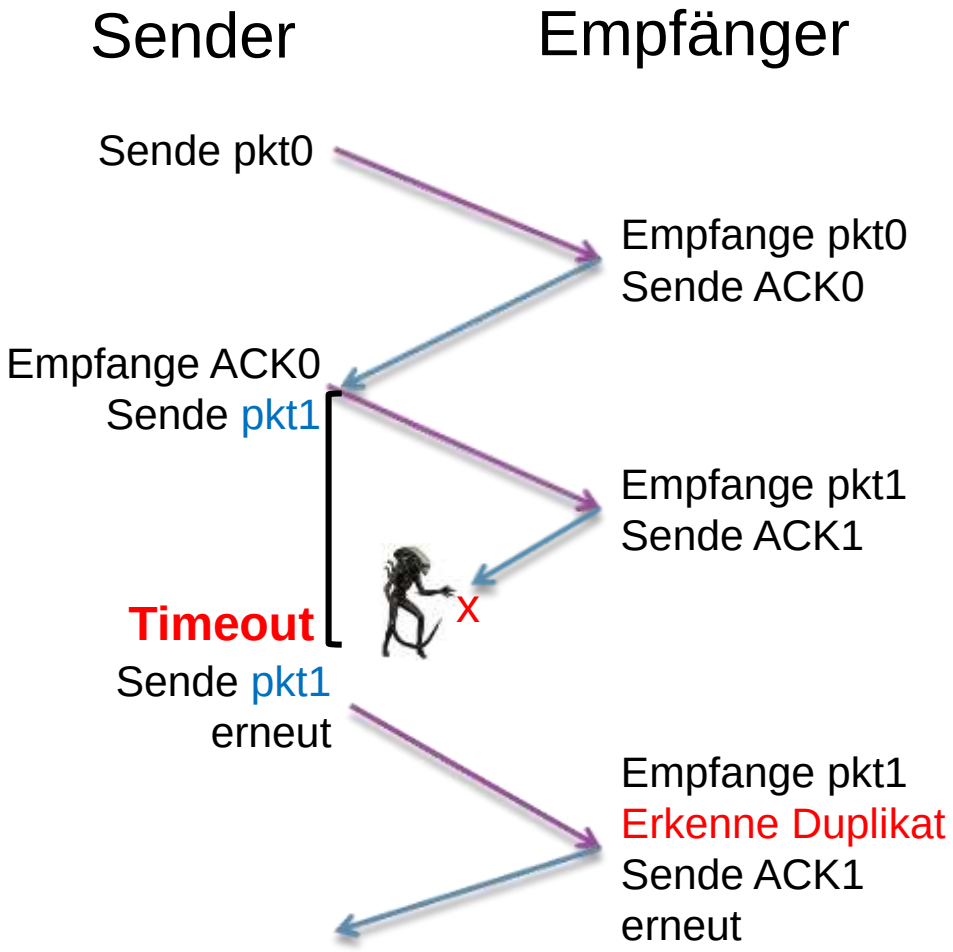
- Wenn ein Paket verlorengeht (oder die Prüfsumme falsch ist), gibt es kein ACK
- Irgendwann erzeugt der Timer des Senders einen Alarm
  - Sender nimmt an, dass das Paket nicht angekommen ist, und schickt es erneut



Implementierung – Version S&W /3

Verlust der ACK

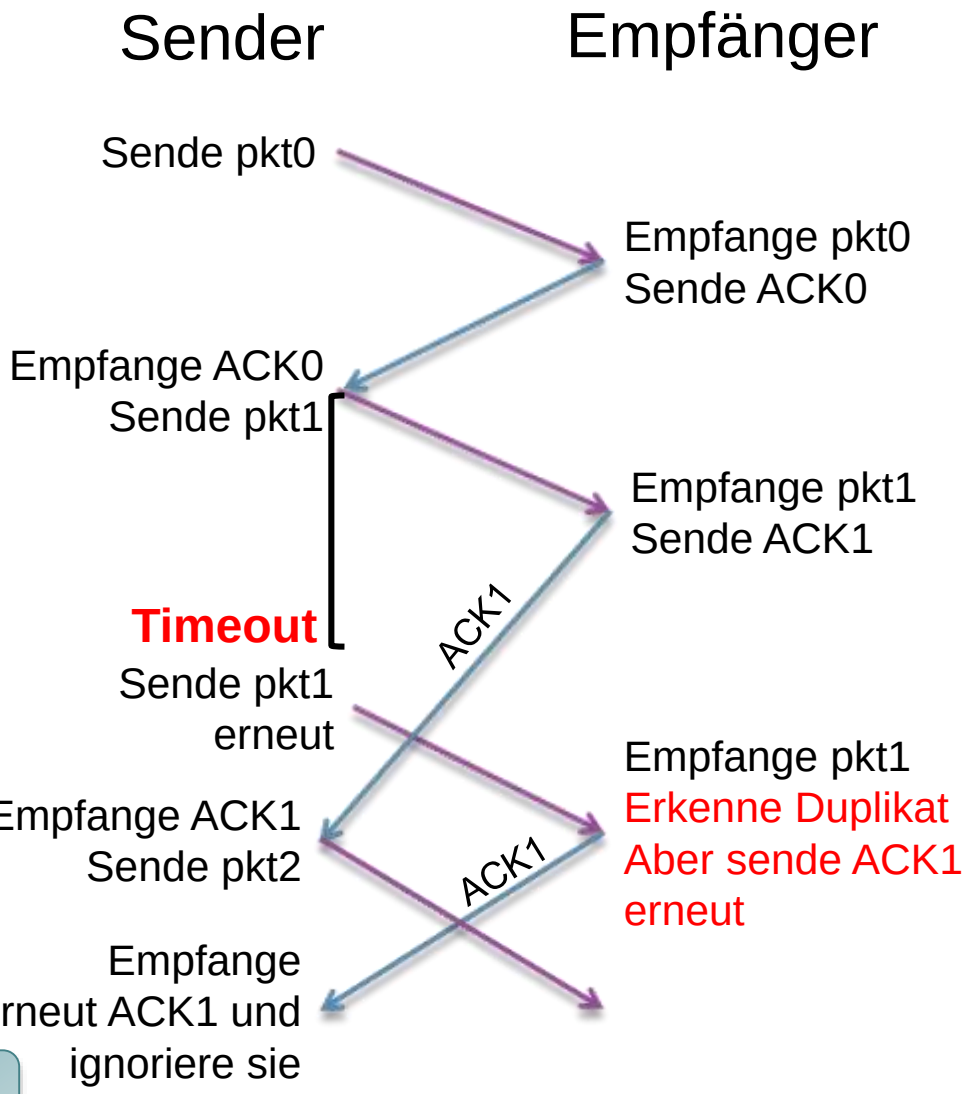
- Wenn die Bestätigung (ACK_x) verlorengeht, verhält sich Sender gleich, d.h. …
- Irgendwann erzeugt der Timer des Senders einen Alarm …
- Der Empfänger erkennt aber Duplikat von Paket pkt1 und schickt nochmals das ACK1



Implementierung – Version S&W /4

Verfrühtes Timeout / doppeltes Paket

- Wenn die Übertragung zu langsam ist, kann ein verfrühter Timeout auftreten
- Der Empfänger erkennt doppeltes Paket pkt1, schickt aber trotzdem die ACK1 nochmals
  - Warum?
- Wofür steht S&W?



Stop & Wait

Video – Stop & Wait / Pipelining

- Stop & Wait Protokoll:
  - Computer Networks 3-2: Retransmissions
  - https://www.youtube.com/watch?v=gokNkNMjkyI
  - Ab ca. 12:30 bis 17:05 (min:sec)

- Pipelining (genannt hier Sliding Window)
  - Computer Networks 3-2: Retransmissions
  - https://www.youtube.com/watch?v=gokNkNMjkyI
  - Ab 17:05 (min:sec)

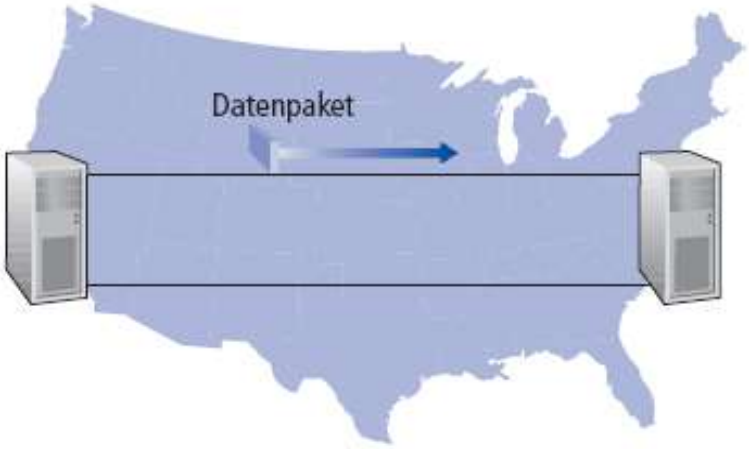
- Ist S&W ein gutes Protokoll? (Was ist „gut“?)

Stop-and-Wait-Protokoll - Beispiel

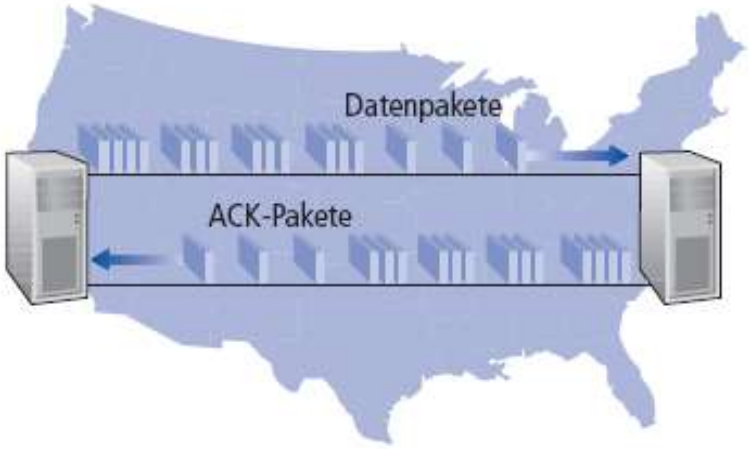
- Was ist mit der Leistung? Beispiel:
  - 1 Gbps Verbindung, Paket von 8000 Bits, Übertraguns-verzögerung ist dann 8000 Bits / 10^9 bps= 8 Mikrosek.
  - Sei die Ausbreitungsverzögerung 15 ms (L ca. 4500 km)
- Was ist dann die **Auslastung** des Absenders?
  - := Bruchteil der Zeit, in der der Sender tatsächlich sendet
- Auslastung = Übertragunsverzögerung /
  [RTT + (Übertragunsverzögerung Paket) + (Übertragunsverzögerung ACK (64 Bytes))]
  - Auslastung = 0.008 ms / (30.0085 ms) = 0.000266
- **Durchsatz?** – d.h. (Übetragene Datenmenge) / Zeit?
- 1 kByte-Paket alle 30.0085 ms => ca. 33 kByte/sek durch eine 1 Gbps Leitung => *Weak weak weak*!

Die Lösung: Pipelining

- Bei **Pipelining** sendet der Sender mehrere Pakete, ohne auf eine Bestätigung von jedem zu warten
- Die Pakete werden vom Empfänger “gruppenweise” bestätigt
- Das erfordert **Pufferung** der Pakete beim Sender und Empfänger

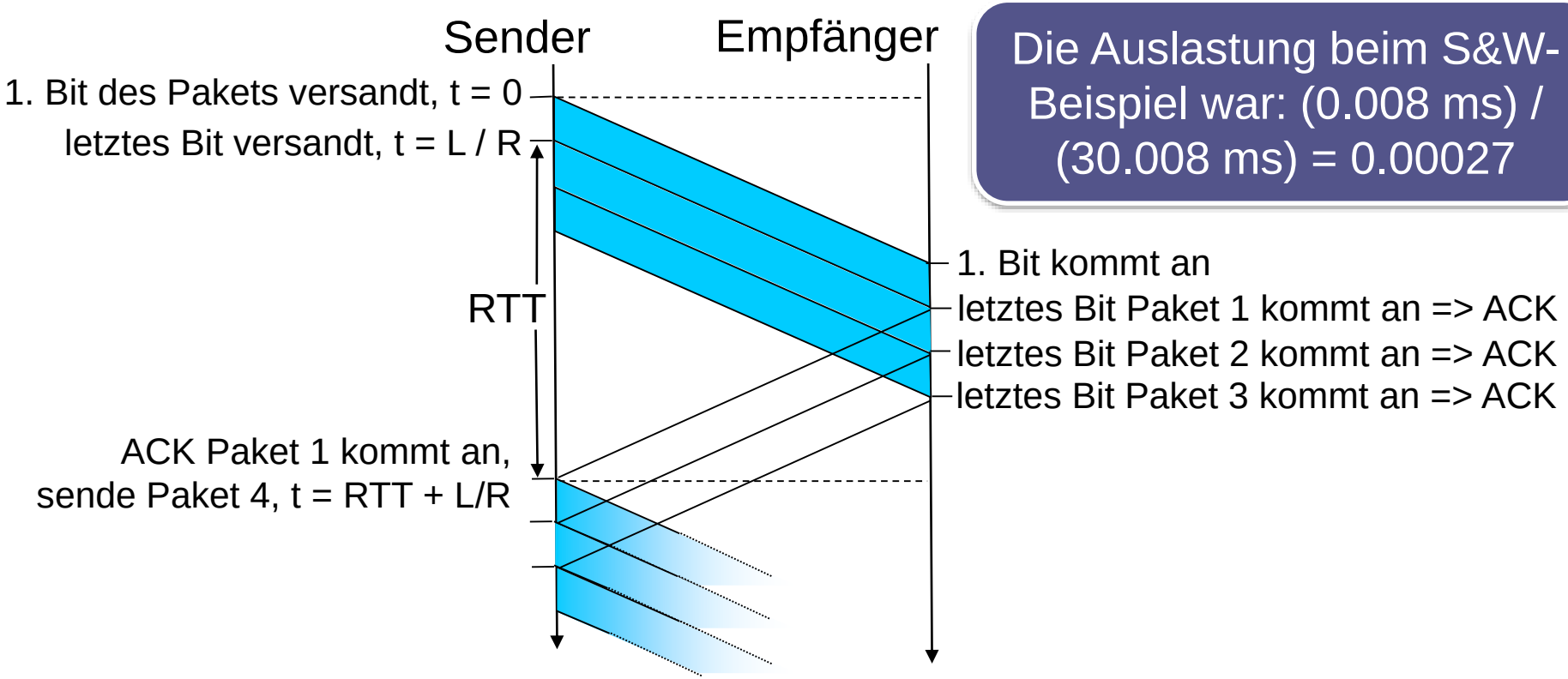


a Ablauf bei Stop-and-Wait



b Ablauf bei Pipelining

Pipelining erhöht die Auslastung



Die Auslastung beim S&W-Beispiel war: (0.008 ms) / (30.008 ms) = 0.00027

Was ist die Auslastung, wenn der Sender N (hier 3) Pakete schickt und erst danach auf Bestätigungen wartet? (Bedingungen wie beim S&W-Bsp.)

- Auslastung:
  - (3*0.008 ms) / (30.008 ms) = 3*0.00027

Auslastung erhöht sich um Faktor N

Verlässliche Nachrichtenzustellung: Strategien der Bestätigung

Video aus Coursera: http://1drv.ms/1FQTsFr

- Internet B - Technology/ B05 - Transport Layer
- Von 2:00 bis 5:30 (min:sec)

Pipelining v1 – **Selective Repeat**

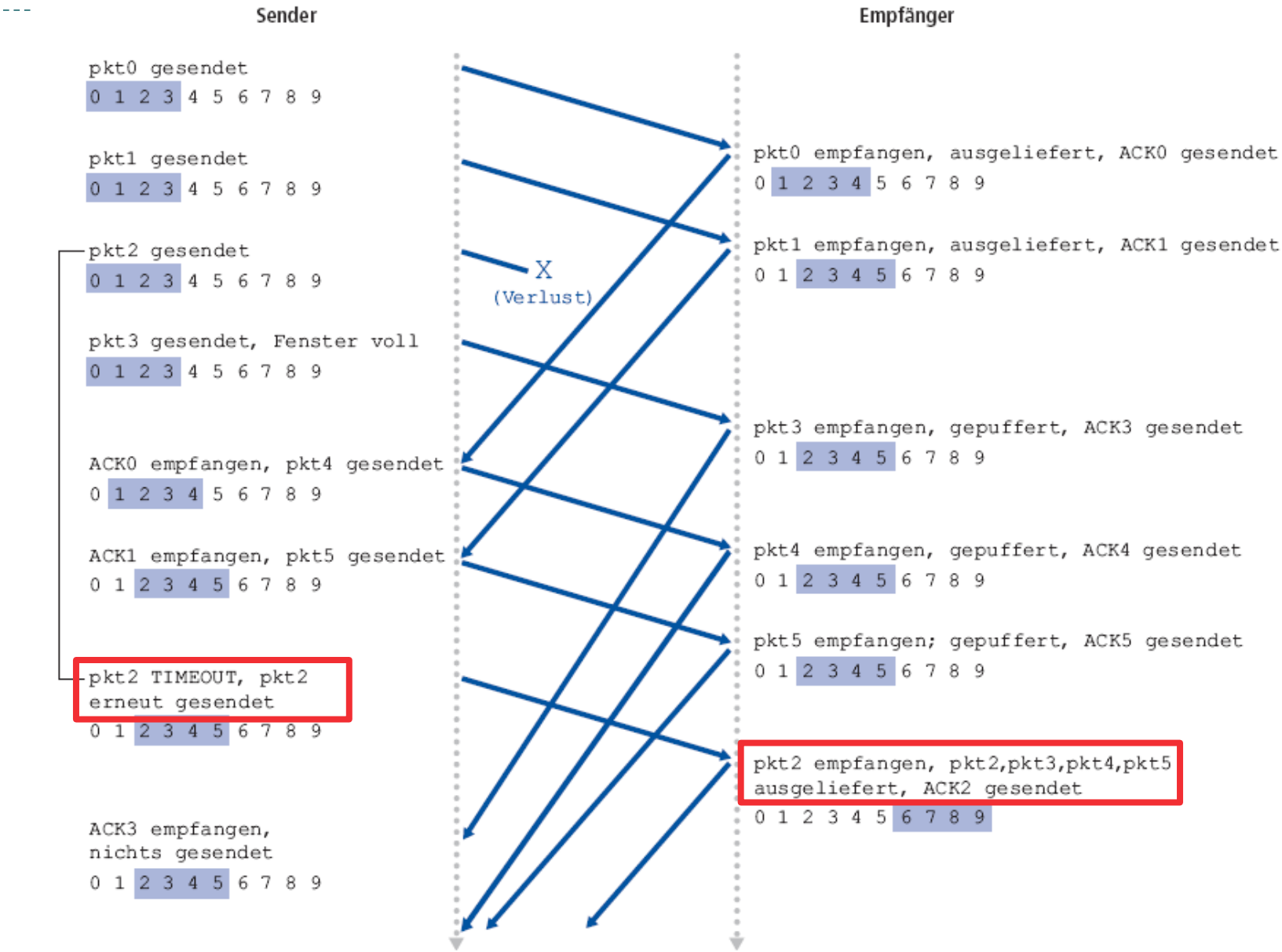
Empfänger

- Empfänger bestätigt er jedes korrekt empfangene Paket individuell
- Empfängt alle Pakete (im gewissen Bereich der Sequenznummer) und bringt sie in richtige Reihenfolge

Sender

- Sendet bis zu N Pakete ohne ACKs (Pipelining)
- Hat einen Timer für jedes nicht-bestätigte Paket
- Sender schickt erneut nur diese Pakete, für die die ACKs nicht empfangen wurden
  - Daher der Name „Selective Repeat“

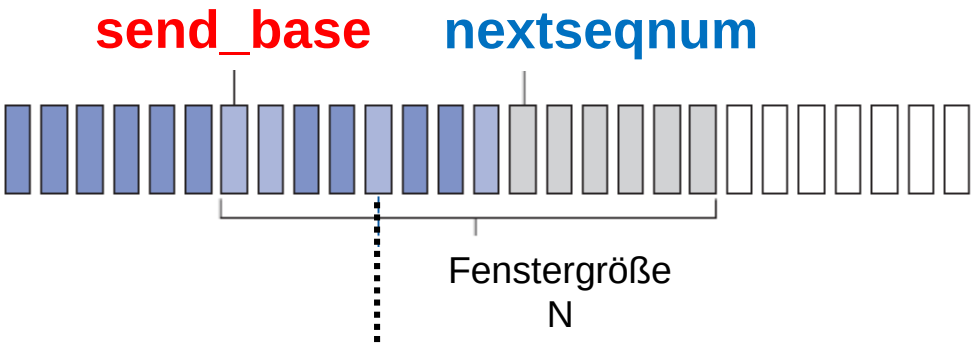
Selektive Repeat: Beispiel



Selective Repeat – Fenster /1

- Es gibt ein **Fenster der Größe N** beim Sender und beim Empfänger (**Sliding Window**)
- **Sender**: Das Fenster verschiebt sich nach rechts (zu höheren Sequenznummern), nur wenn das Paket im Fenster mit kleinster Sequenznr. bestätigt wurde
  - Kleinste Sequenznummer (linker Rand) im Fenster ist **send_base** (rechter Rand ist send_base+N-1)
- **Empfänger**: Das „älteste“ erwartete aber noch nicht empfangene Paket (d.h. mit kleinster Sequenznr.) bestimmt den „linken“ Fensteranfang
  - Kleinste Sequenznummer (linker Rand) im Fenster ist **rcv_base**

Selective Repeat – Fenster /2



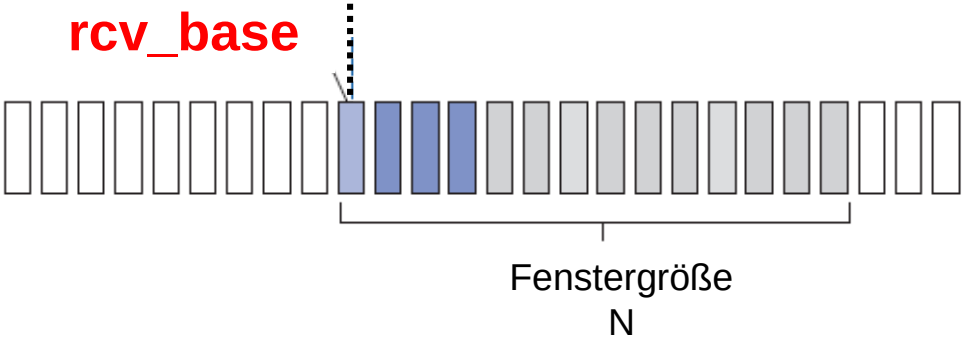
Bereits bestätigt

Nutzbar, noch nicht gesendet

Gesendet, noch nicht bestätigt

Nicht benutzbare Sequenznummern

Sequenznummer aus der Sicht des Senders



Außerhalb der Reihenfolge, bestätigt

Akzeptierbar (innerhalb des Fensters)

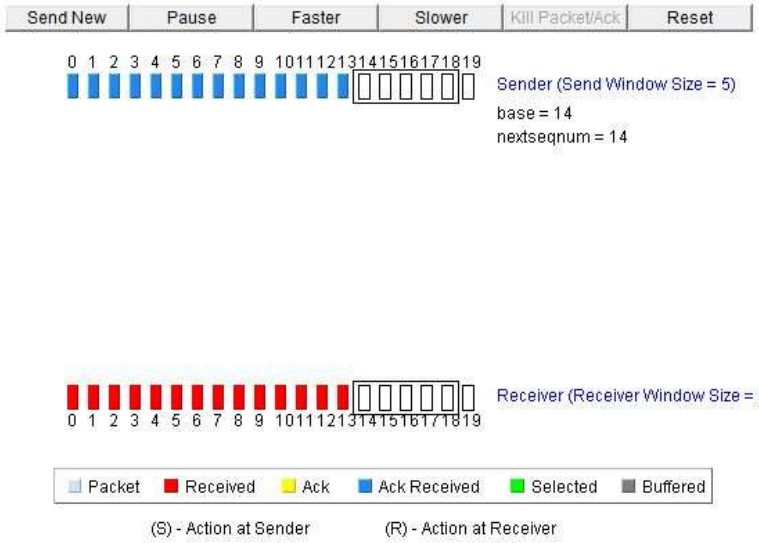
Erwartet, noch nicht eingetroffen

Nicht benutzbare Sequenznummern

Sequenznummer aus der Sicht des Empfängers

Selective Repeat Demo / Video Sliding Win.

- Java Demo: http://goo.gl/ZpoaD
- TCP sliding window with error animation
  - https://www.youtube.com/watch?v=lk27yiITOvU



- Problem bei Selective Repeat?

- Wir schicken sehr viele ACKs!
  - Je ein Bestätigungspaket pro ein empfangenes Paket - sehr ineffizient

Zusammenfassung

- TCP – Protokoll (Grundlagen)
- TCP – Protokoll: Verbindungsverwaltung
- TCP – Protokoll: Flusskontrolle
- Verlässliche Nachrichtenzustellung
  - Naive Umsetzung
  - Selective Repeat

- Quellen:
  - Kurose / Ross Kapitel 3, u.a. Abschnitte 3.4 - 3.5
  - Wikipedia

Danke.

Zusätzliche Folien

Zwei Grundmodelle des Pipelining

- **Go-Back-N** (GBN)
  - Etwas einfacher
  - Effizient, wenn die meisten Pakete ankommen
  - Weniger effizient, wenn viele Pakete verlorengehen
- **Selective Repeat**
  - Etwas komplizierter
  - Effizient, wenn viele Pakete verlorengehen
  - Weniger effizient, wenn die meisten ankommen
- Welches Modell benutzt TCP?
- Eine Mischung aus beiden (mit mehr Go-Back-N)

Selective Repeat: Ereignis-basierter Algorithmus

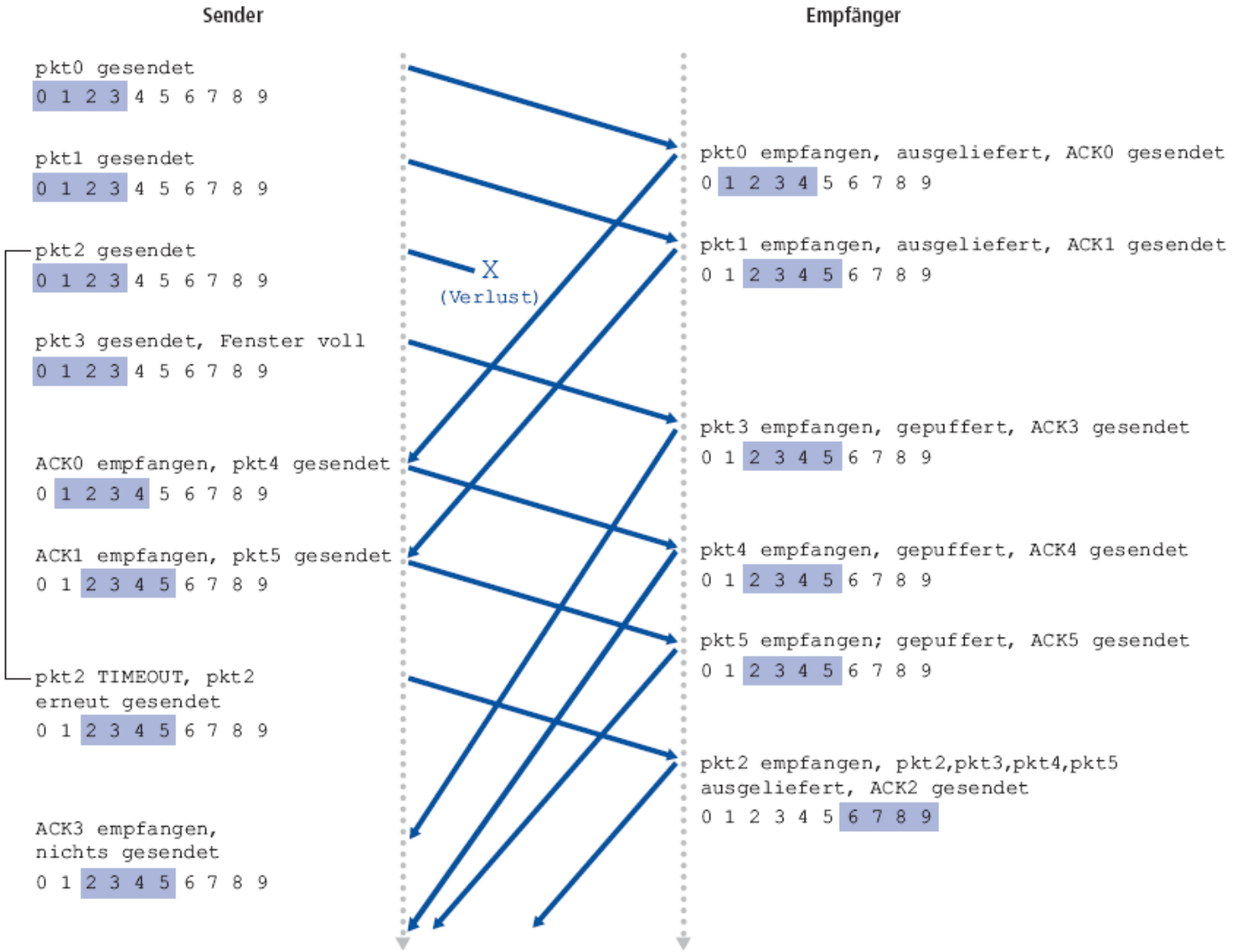
Sender

- Bei neuen Daten:
  - Sende, falls nächste ungenutzte Seq# im Fenster liegt; Timer(n) starten
- Timeout für Paket n:
  - Sende Paket n erneut und starte erneut Timer dafür
- ACK(n) in [send_base, send_base+N-1]:
  - Markiere n als bestätigt
  - War n die kleinste nicht-bestätigte Seq#, vergrößere send_base zur nächsten nicht-bestätigten Seq#

Empfänger

- Paket n in [rcv_base, rcv_base+N-1]:
  - sende ACK(n)
  - Puffere Paket n
  - Falls k Pakete ab rcv_base in Reihenfolge, liefere diese k Pakete aus und erhöhe rcv_base um k
- Paket n in [rcv_base-N, rcv_base-1]:
  - sende ACK(n) – warum?
- Sonst:
  - Ignoriere

Selektive Repeat: Beispiel



Problem

- Beispiel:
  - Sequenznummern: 0-3
  - Fenstergröße=3
- Empfänger kann beide Fälle (a), (b) nicht unterscheiden!
- Gibt "verdoppeltes" altes Paket als neue Daten nach oben!
- Die Anzahl der Sequenznummern muss viel höher als die Größe des Fensters sein