Rechnernetze
Kapitel 6: Transport Layer

Prof. Dr. Wolfgang Mühlbauer
Fakultät für Informatik
wolfgang.muehlbauer@th-rosenheim.de

Wintersemester 2019/2020

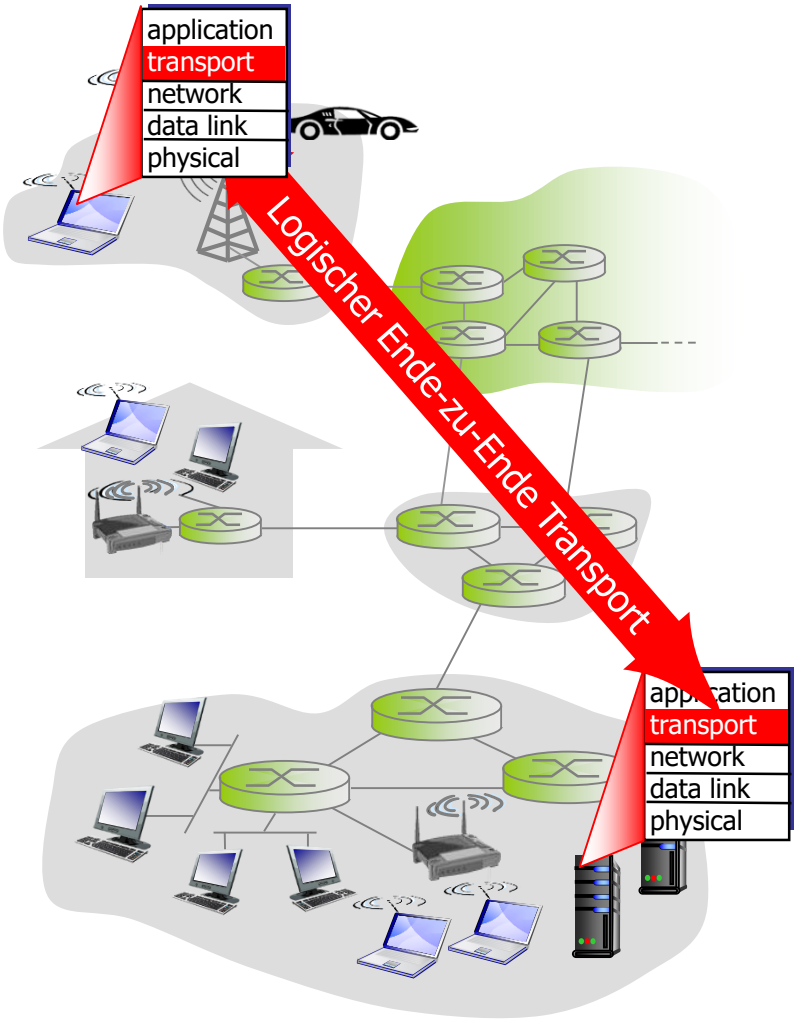
Slides are based on:
J. Kurose, K. Ross: Computer Networks – A Top-Down Approach
A. Tanenbaum, D. Wetherall: Computer Networks

Inhalt

- **Allgemeines, UDP**
- TCP: Allgemeine Prinzipien
- TCP: Konkrete Umsetzung
- Flow und Congestion Control bei TCP
- Network Address Translation (NAT)

Transport Layer

- Kommunikation zwischen ***Prozessen*** auf Sender Empfängerseite
  - Vergleiche: Network Layer bietet Kommunikation zwischen ***Hosts***.
- Implementiert bei Hosts, nicht in Routern
  - *Sender:* Verpackt Anwendungsdaten (z.B. HTTP) in *Segment,* Weitergabe an Network Layer
  - *Empfänger*: Baut aus *Segmenten* Nachrichten zusammen, Weitergabe an Anwendung
- Transport Layer ist Teil des Betriebssystems.

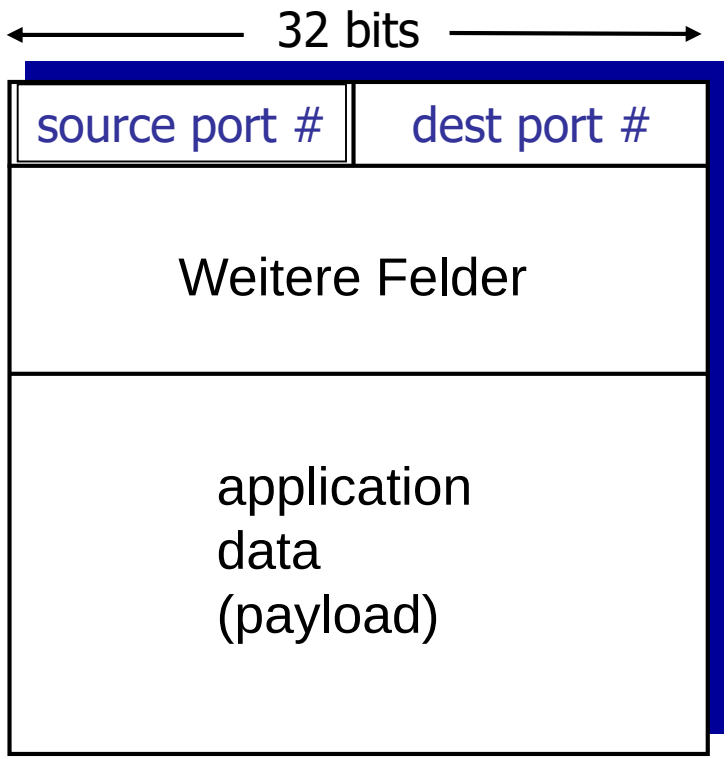


Transport Layer im Internet

- **User Datagram Protocol (UDP)**
  - ***Transport-Layer Multiplexing:*** Ordne IP Pakete den Prozessen des Betriebssystems zu.
  - Erkennung von *Übertragungsfehlern* durch hecksumme
- **Transmission Control Protocol (TCP)**
  - UDP + zusätzliche „Feature“!
  - ***Verbindungsorientiert:*** Baue Verbindung vor Datenübertragung auf.
  - ***Zuverlässig (engl. "reliable")***: Absicherung gegen
    - Übertragungsfehler
    - Paketverluste
    - Veränderung der Reihenfolge
  - ***Flow Control***: Vermeide Überlastung des Empfängers
  - ***Congestion Control***: Vermeide Überlastung des Netzwerks
- UDP und TCP: Keine Garantie bzgl. Delay und Bandbreite!

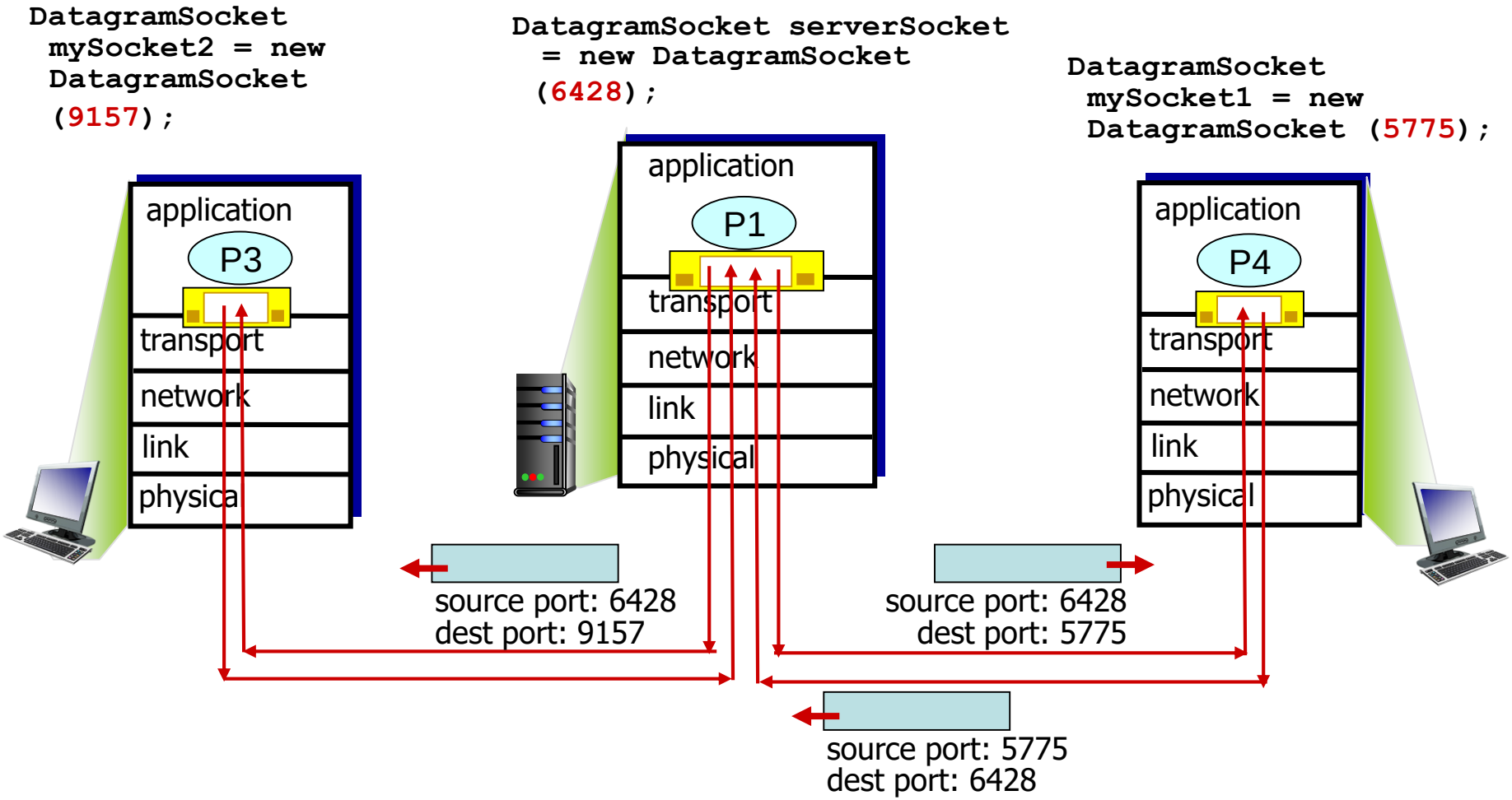
Transport-Layer Multiplexing

- **Multiplexing**
  - Zuordnung von Transport-Layer Verbindungen zu Server-/Client-Programm des Betriebssystems
- **Sender**
  - Socketadresse == ***Portnummer***
    - 16 Bit: 1..65535
  - *Destination Port* des Server muss bekannt sein
    - Beispiel HTTP(S): Port 80 bzw. 443
    - Well-Known Ports:
    - *Source Port* meist beliebig wählbar.
- **Empfänger**
  - Empfangene Pakete werden anhand des Destination Ports zu korrekten Prozess geleitet
  - HTTP, FTP, E-Mail Prozess



Format eines TCP/UDP Segments

UDP: Beispiel in Java



http://openbook.rheinwerk-verlag.de/java7/1507_11_011.html#dodtpce42b389-f532-4bfe-aff9-50e5e00d9bf9

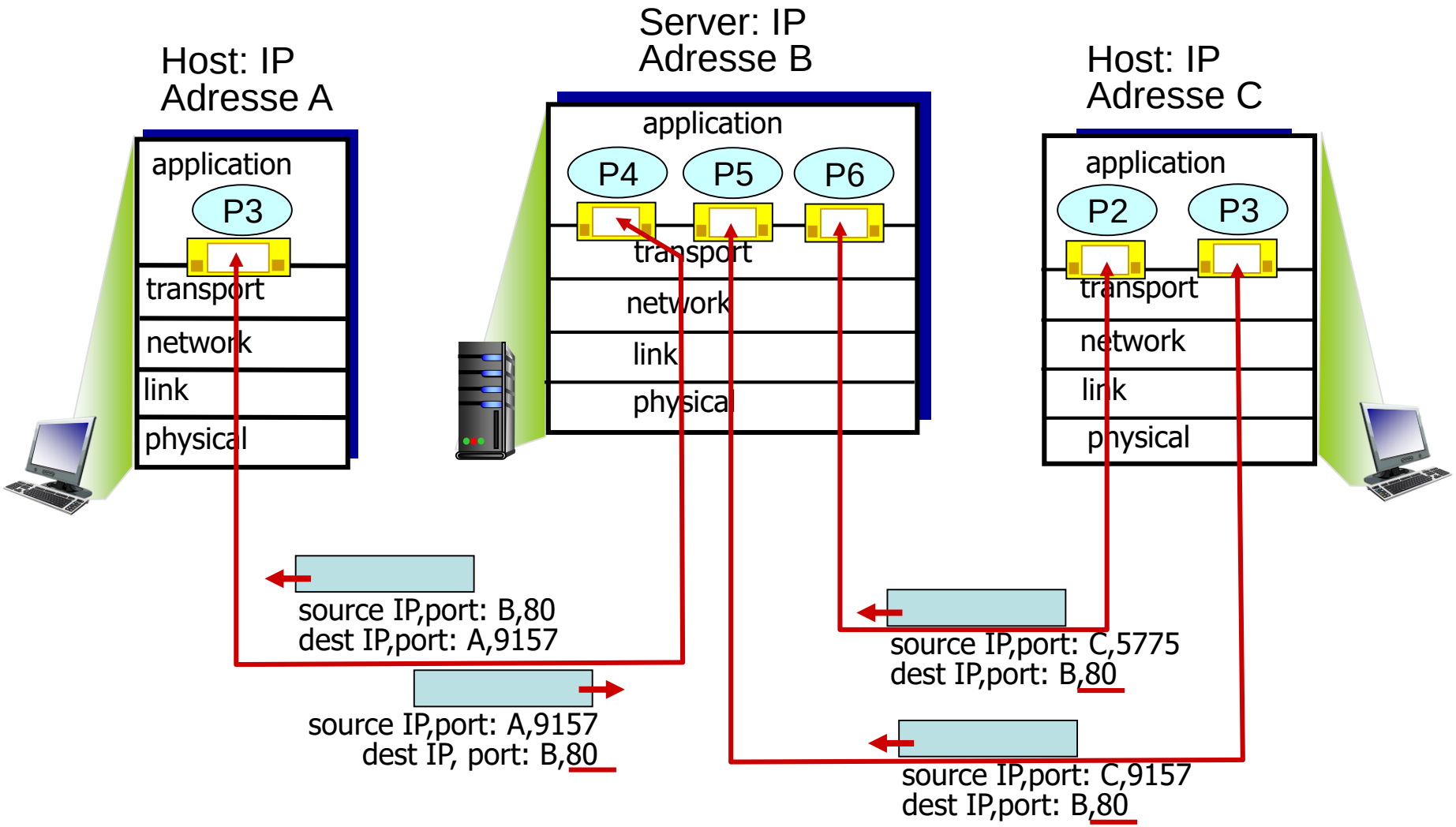
Multiplexing: TCP vs. UDP

- **UDP ist verbindungslos.**
  - UDP Socket festgelegt durch: Dst IP + Dst Port
  - Segmente mit gleichem Dst IP/ Dst Port werden an gleiches Socket weitergeleitet.
  - Empfänger schaut sich Src IP und Src Port gar nicht an.

- **TCP ist verbindungsorientiert.**
  - TCP Socket festgelegt durch: Src IP + Src Port + Dst IP + Dst Port
  - Nur wenn *alle* 4 Werte gleich → Weiterleitung an gleiches Socket

- **Beispiel:** Web Server wartet auf Anfragen an Port 80 → bind()
  - Hosts erzeugen parallele Web-Anfragen, aber mit verschiedenen Src Port und Src IP.
  - Server erzeugt erst bei jedem `accept()` das eigentliche Socket

Multiplexing bei TCP



Der Server “betreibt” 3 Sockets, alle mit der gleicher dst IP und dst/port.
Src IP/src Port unterscheiden sich jedoch.

Publikums-Joker: Ports

Welche der folgenden Aussagen ist ***falsch***?

A. Ein HTTP(S) Webserver läuft auf Port 80 bzw. 443.

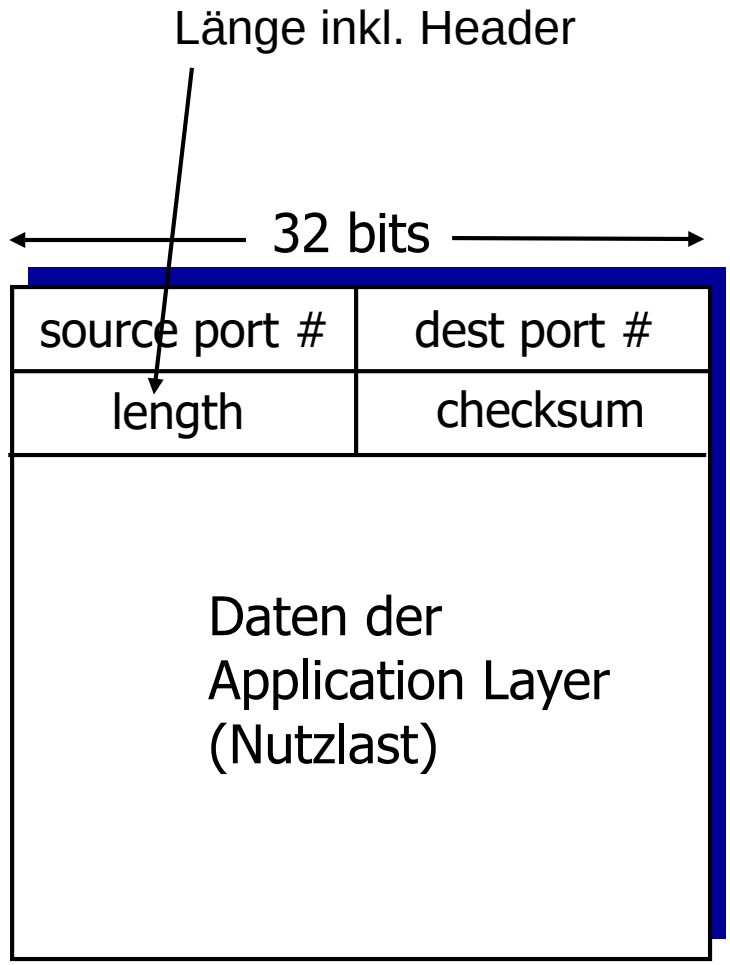
B. Es gibt Netzwerkpakete, die weder TCP noch UDP verwenden.

C. Dem Betriebssystem des UDP Empfängers ist die Portnummer des Senders egal.

D. Die folgende URL ist gültig: https://www.spiegel.de:443

User Datagram Protocol (UDP)

- ***„Best Effort“*** Service
  - Einziger Mehrwert: Port Multiplexing
  - Paketverluste möglich
  - Keine Einhaltung der Reihenfolge
- ***Verwendung***
  - Multimedia, Streaming
  - DNS
  - SNMP
- ***„Vorteil“: Einfach!***
  - Kein Verbindungsaufbau
  - Schlanke Implementierung
  - Wenig Overhead
  - …



Format des UDP Segments

Inhalt

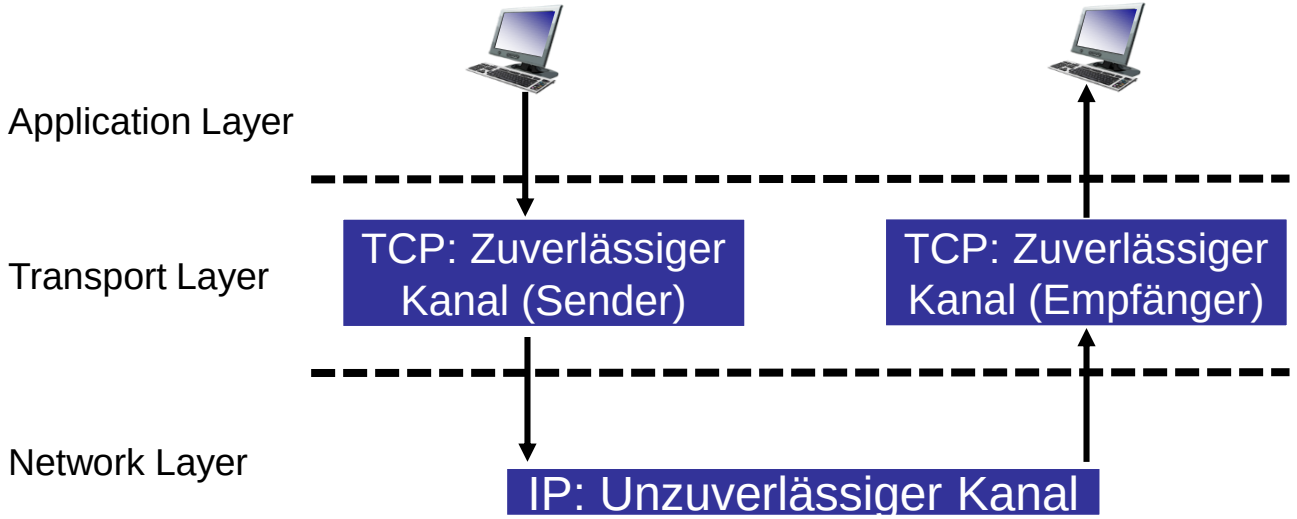
- Allgemeines, UDP
- **TCP: Allgemeine Prinzipien**
- TCP: Konkrete Umsetzung
- Flow und Congestion Control bei TCP
- Network Address Translation (NAT)

Zuverlässige Datenübertragung

- **Definition: Zuverlässig (engl. "reliable")**
  - Alle Daten werden korrekt ohne Bitfehler übertragen.
  - Keine Daten gehen verloren.
  - Alle Daten in korrekter Reihenfolge ausgeliefert.

- TCP gewährleistet eine *zuverlässige* Datenübertragung.

- **Herausforderung:** *Zuverlässige* Übertragung über einen ***unzuverlässige Network Layer (IP)!***



Zuverlässige Datenübertragung

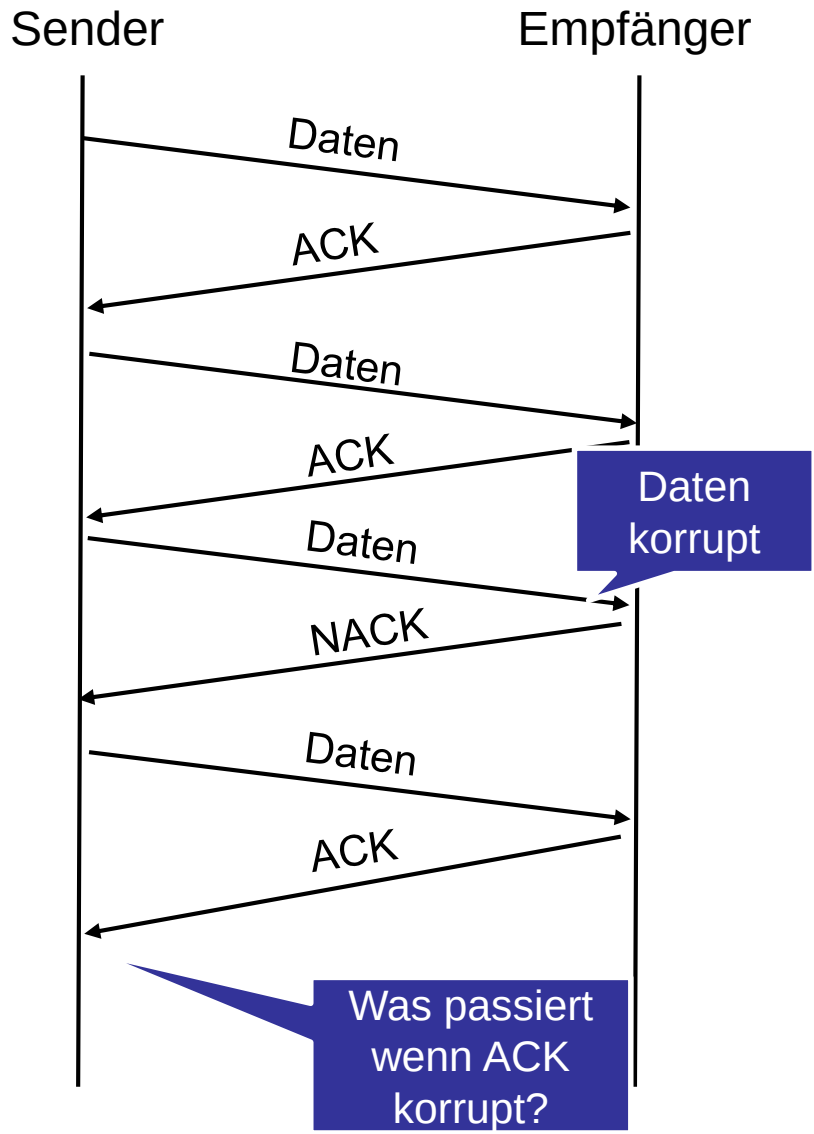
- **Ziel: Fiktives** Protokoll
  - Keine Bitfehler, kein Datenverlust, korrekte Reihenfolge
  - **Erst im nächsten Abschnitt: Implementierung bei TCP!**

- **Vorgehen:** Schrittweises Verfeinern des Protokolls

- **Vereinfachende Annahmen:**
  - *Unidirektionale* Datenübertragung von Sender zu Empfänger
    - Verallgemeinerung auf bidirektional folgt später.
  - Übertragungskanal lässt *Reihenfolge* der Pakete unverändert.
    - Bei Routing-Änderungen könnten sich Pakete überholen.
  - Bitfehler erkennbar durch Checksumme.

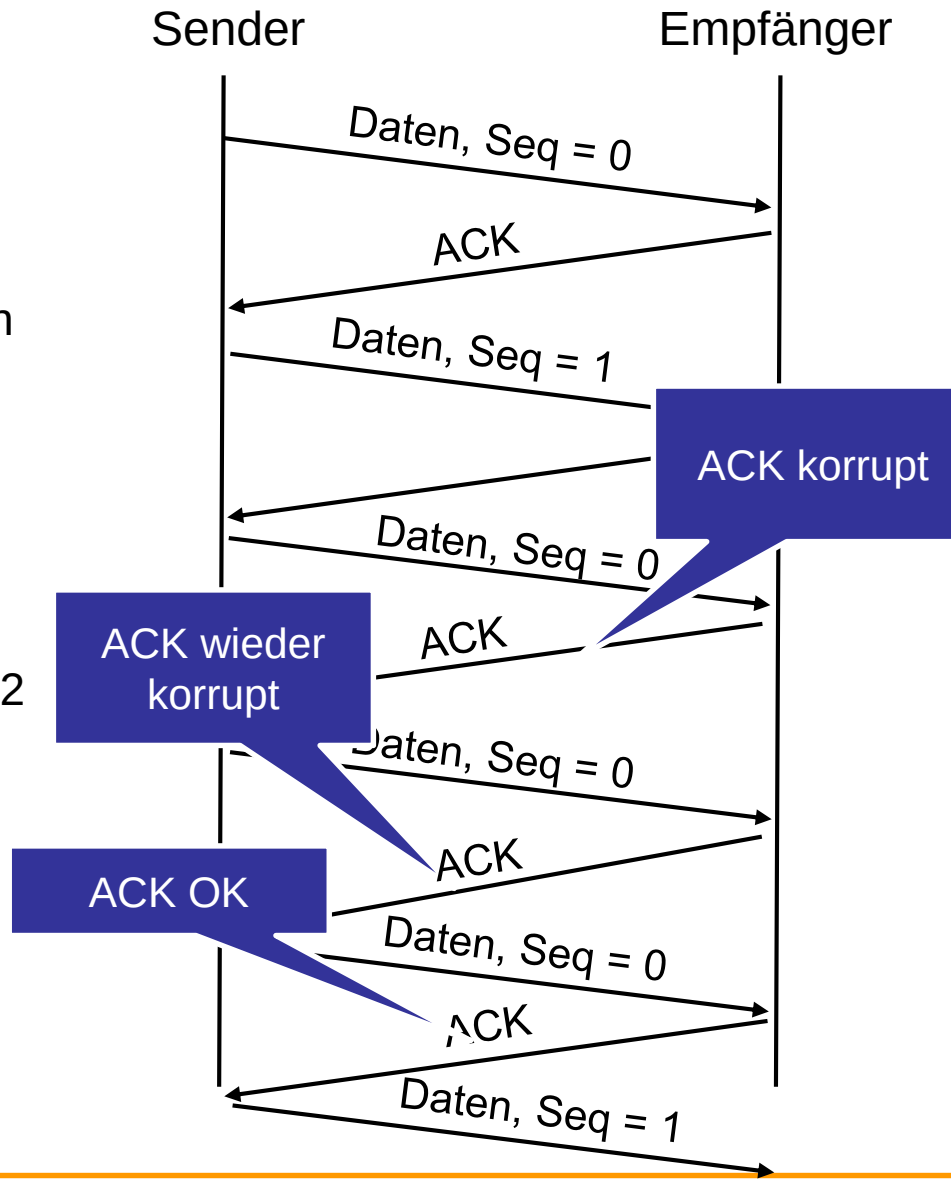
Version 1.0: Bitfehler im Datenpaket

- Problem:
  - Einzelne Bits in den Datenpaketen können während Übertragung umkippen
- Erkennung:
  - Checksum, CRC, Parität, etc.
- Reaktion: Explizites Feedback
  - Positives Acknowledgment (ACK)
    - „OK!“
  - Negatives Acknowledgment (NACK)
    - „Bitte nochmal senden!“ → *Retransmission*
    - ***Automatic Repeat reQuest (ARQ)***
- **Stop-and-Wait-Prinzip**
  - Nächstes Paket wird erst gesendet nach Erhalt von ACK des vorherigen Pakets



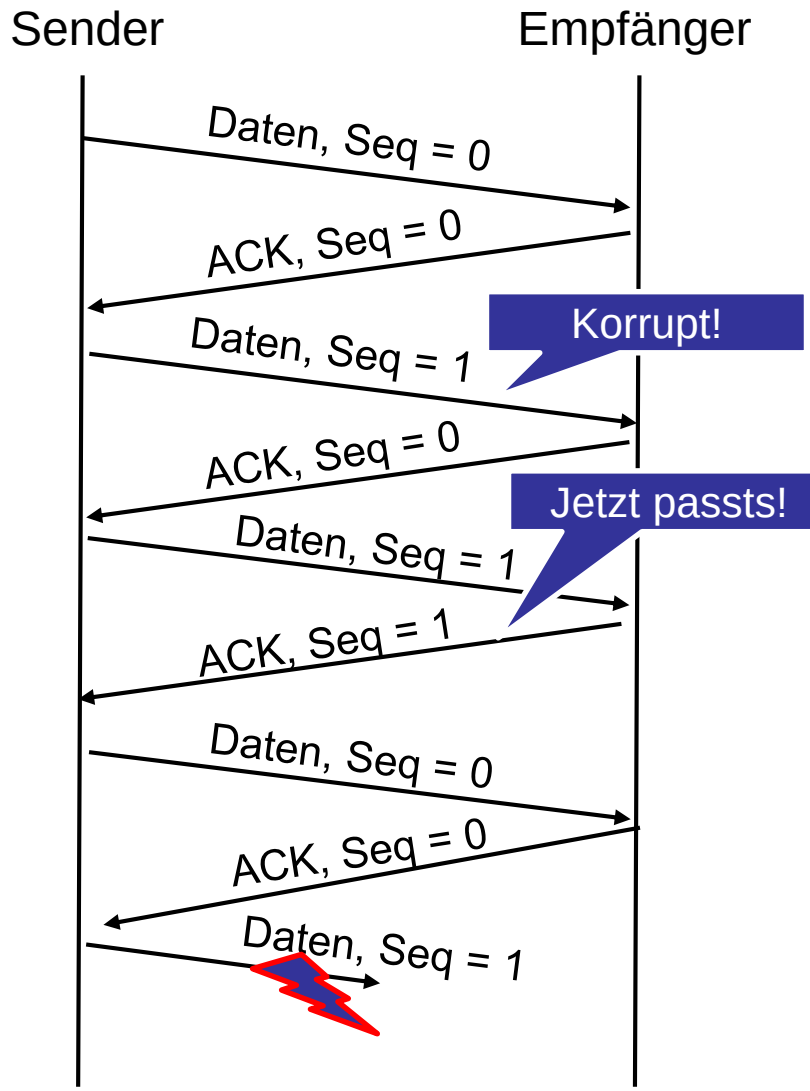
Version 1.1: Bitfehler im ACK/NACK

- Problem: ACK/NACK Paket korrupt
- Naheliegende Lösung
  - Checksumme auch für ACK/NACK Pakete
  - *Retransmission*: Einfach nochmal übertragen
  - Aber: Wie erkennt Empfänger Duplikate?
- Erkennen von Duplikaten
  - Sender fügt jedem Datenpaket **Sequenznummer** hinzu (1 Bit genügt hier)
  - Empfänger verwirft auftretende Duplikate (= 2 gleiche Sequenznummern hintereinander)
- Weiterhin Stop-and-Wait
- Braucht man NACKs überhaupt?



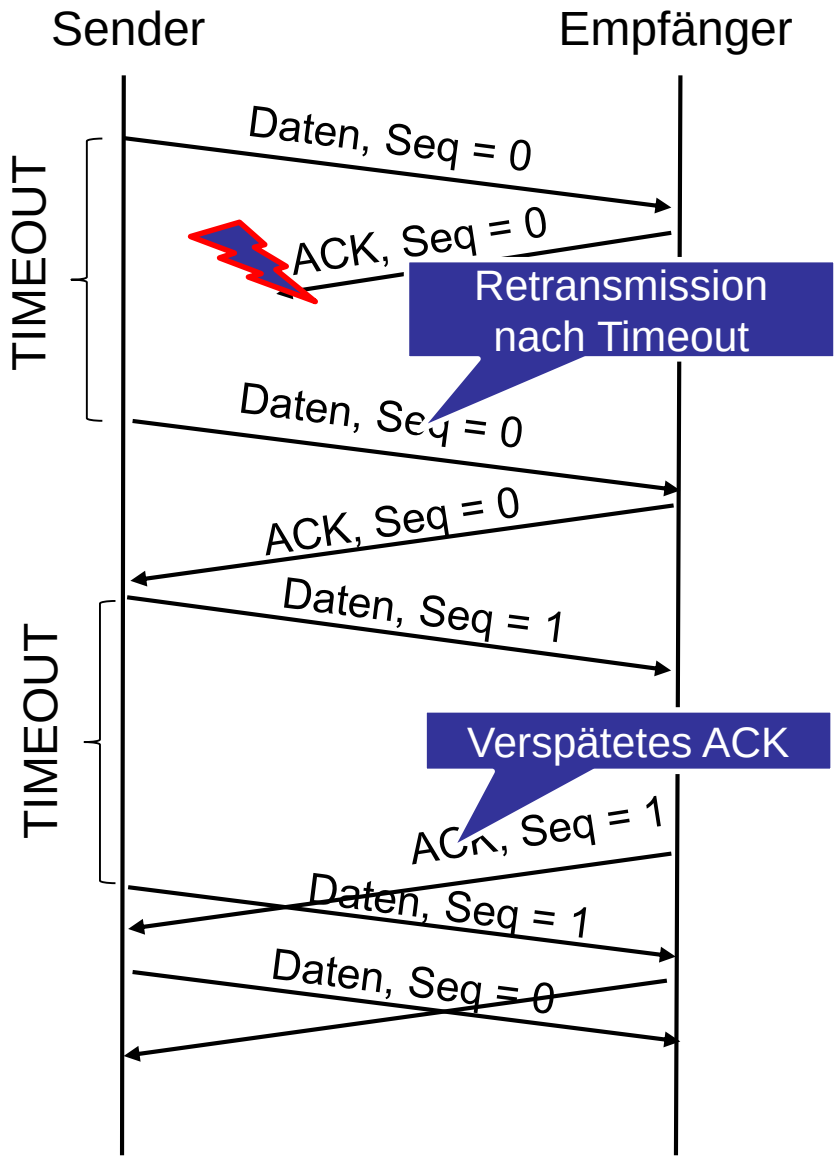
Version 1.2: NACK-freies Protokoll

- Auf Nachrichtentyp NACK kann "verzichtet" werden
- Lösung
  - Sequenznummern auch in ACK
  - Empfänger sendet ACK, selbst wenn gerade empfangenes Paket korrupt
  - Aber: Empfänger bestätigt immer das letzte, **korrekte** Paket (und nicht das letzte, **korrupte** Paket)
  - Empfängt Sender zweimal das gleiche ACK → Retransmission
- **Was passiert falls Paket komplett verloren geht?**



Version 2.0: Paketverluste

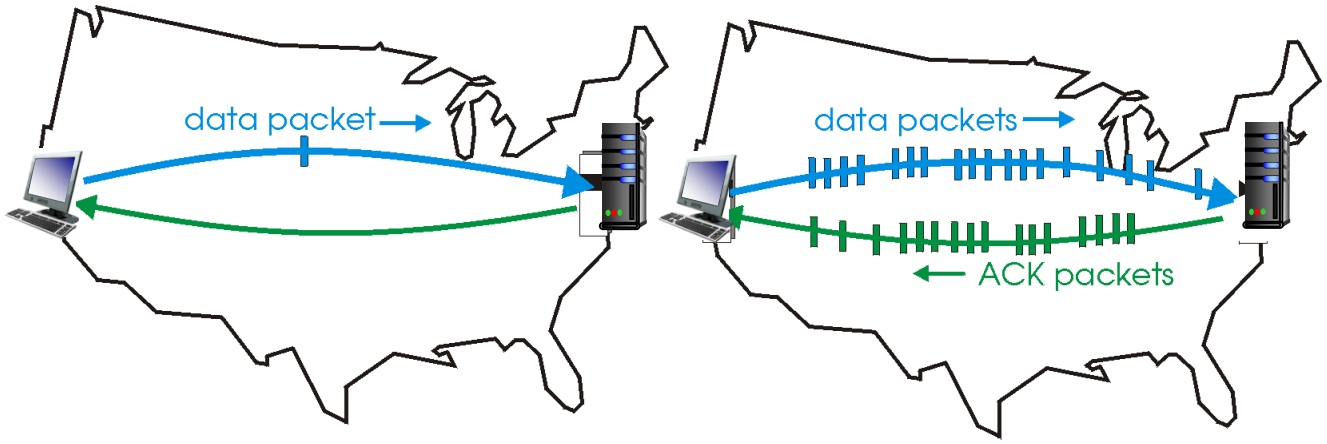
- Problem:
  - Daten und/oder ACK Paket geht komplett verloren
- Lösung: **Timeout**
  - Falls Sender kein ACK innerhalb einer „vernünftigen“ Zeit bekommt → Retransmission
  - Wie wählt man Timeout-Wert?
- Duplikate können entstehen, falls verspätetes ACK (nach Ablaufen des Timers) doch noch ankommt.
  - ACK als auch Daten können doppelt gesendet werden.
  - Erkennung durch Sequenznummern: Empfänger gibt immer an welches Paket er bereits empfangen hat.
  - 1-Bit Sequenznummer genügt!



Version 3.0: Pipelined Datenübertragung

- Problem von Version 2.0: Katastrophale Performance!
  - **Stop-and-Wait**: Sender muss mindestens die *Round Trip Time (RTT)* abwarten, bevor er das nächste Paket sendet,

- **Pipelining:** Sender darf **gleichzeitig** mehrere Pakete senden
  - Es dürfen aber nur eine begrenzte Anzahl von unbestätigten Paketen „unterwegs“ sein
  - Mehr als 2 Sequenznummern notwendig!



Aus Kurose&Ross

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

(b) a pipelined protocol in operation

Publikums-Joker: Zuverlässige Datenübertragung

Welche Aussage ist ***falsch***?

A. WLAN verwendet positive Acknowledgments.

B. Stop-and-Wait Protokolle sind vor allem bei großen Propagation Delays günstig.

C. Pipelining erfordert das Zwischenspeichern von Paketen auf der Senderseite.

D. Version 2.0 stellt sicher: Erkennung von Paketverlusten und Erkennung von Übertragungsfehlern

Pipelined Datenübertragung: 2 Ansätze

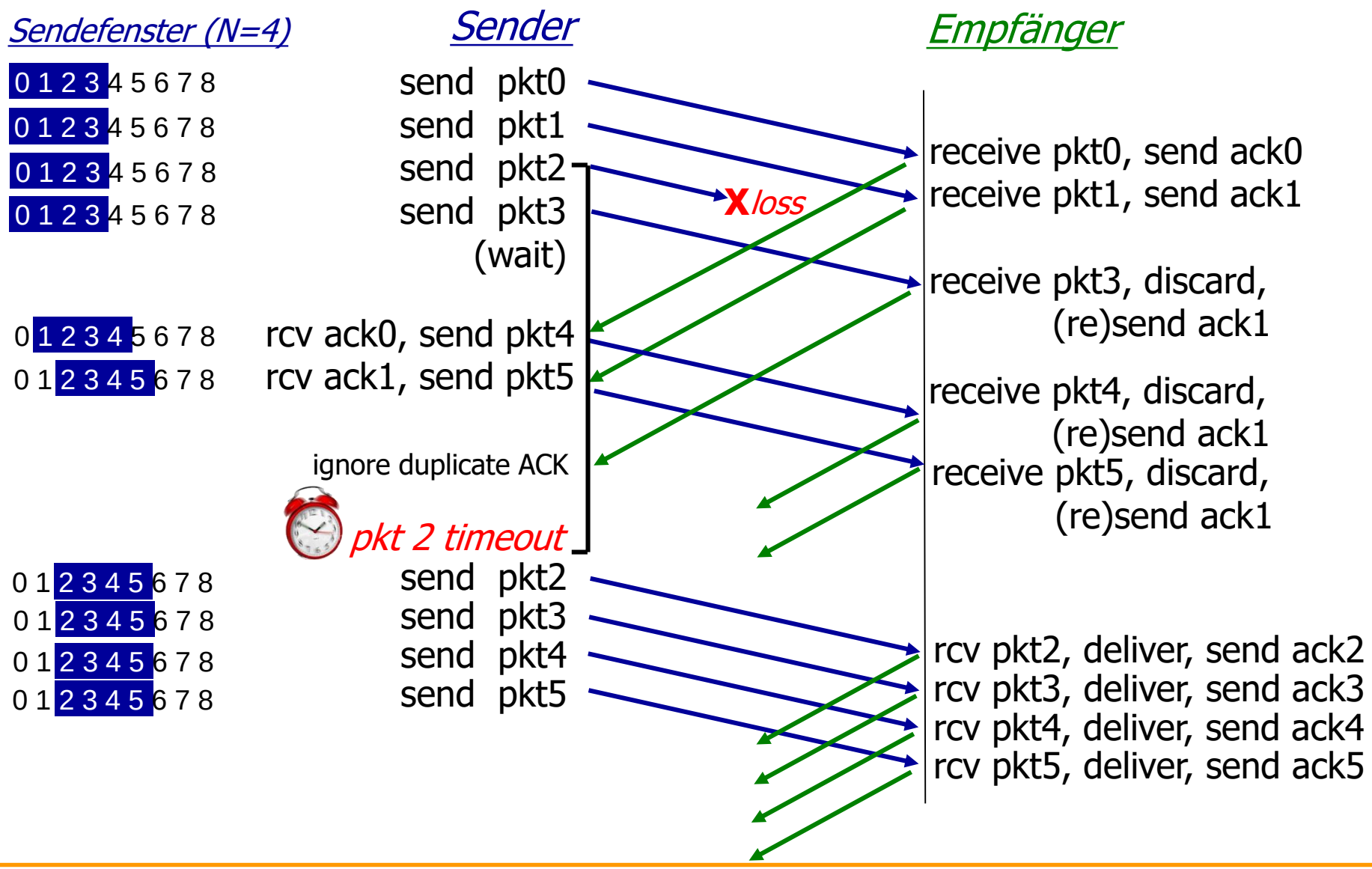
Go-Back-N

- Sender kann bis zu *N* unbestätigte Pakete in der Pipeline (***Sendefenster***) haben
- Empfänger sendet **kumulatives ACK**
  - Jedes ACK des Empfängers bestätigt mit *größter* Sequenznummer *S* für die gilt: **Alle** Pakete mit $s \leq S$ wurden bereits empfangen.
  - Sender weiß so, welche Sequenznummer er dem Sender dringend als nächstes senden muss.
- Meist nur 1 Timer für ***ältestes unbestätigtes*** Paket
- Retransmission **aller** aktuell unbestätigten Pakete

Selective Repeat

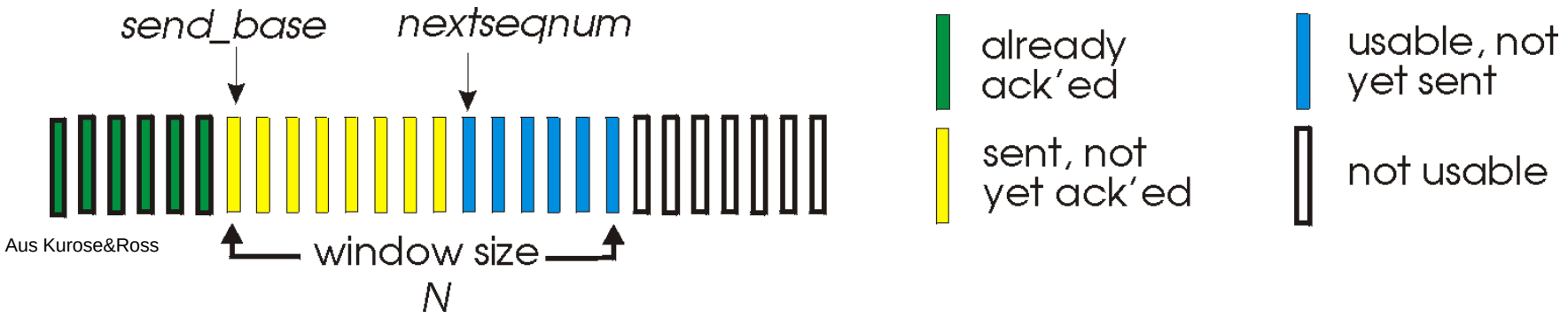
- Sender kann bis zu *N* unbestätigte Pakete in der Pipeline (***Sendefenster***) haben
- Empfänger sendet **individuelle ACKs** für jedes Paket
- Sender unterhält Timer für ***jedes*** unbestätigte Paket
- Timer läuft aus: Retransmission **nur des betreffenden** unbestätigten Pakets.

Go-Back-N: Kumulative ACKs



Go-Back-N: Kumulative ACKs

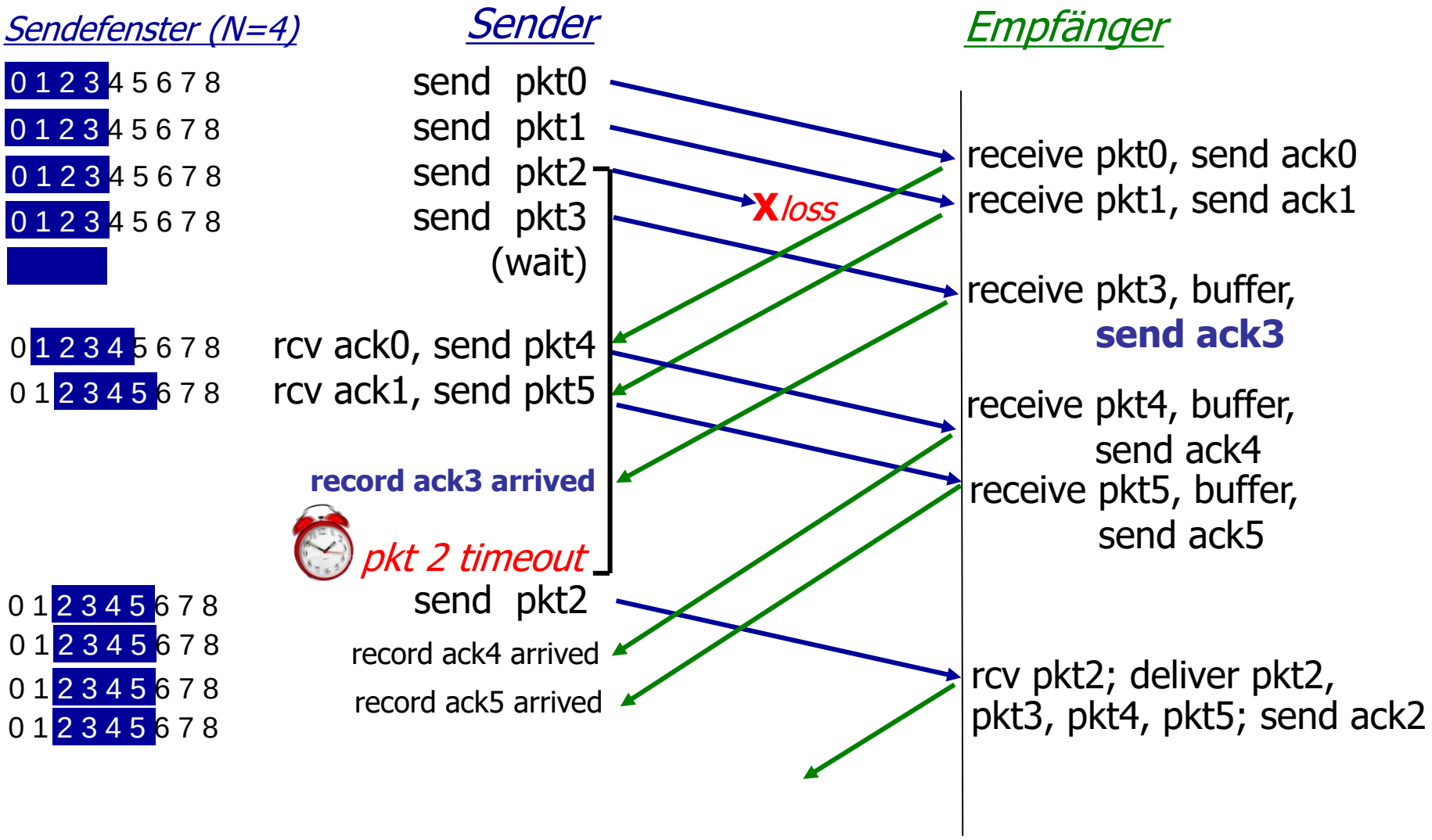
Sequenznummernraum aus Sicht des Senders



- Sender verwendet *k* Bit Sequenznummer im Paket-Header
- `Send_base`: "Älteste" noch unbestätigte Sequenznummer
- `nextseqnum`: Nächste zu verwendende Sequenznummer
- `Window Size N`: Sender sendet bis zu *N* ***aufeinanderfolgende*** Pakete gleichzeitig.

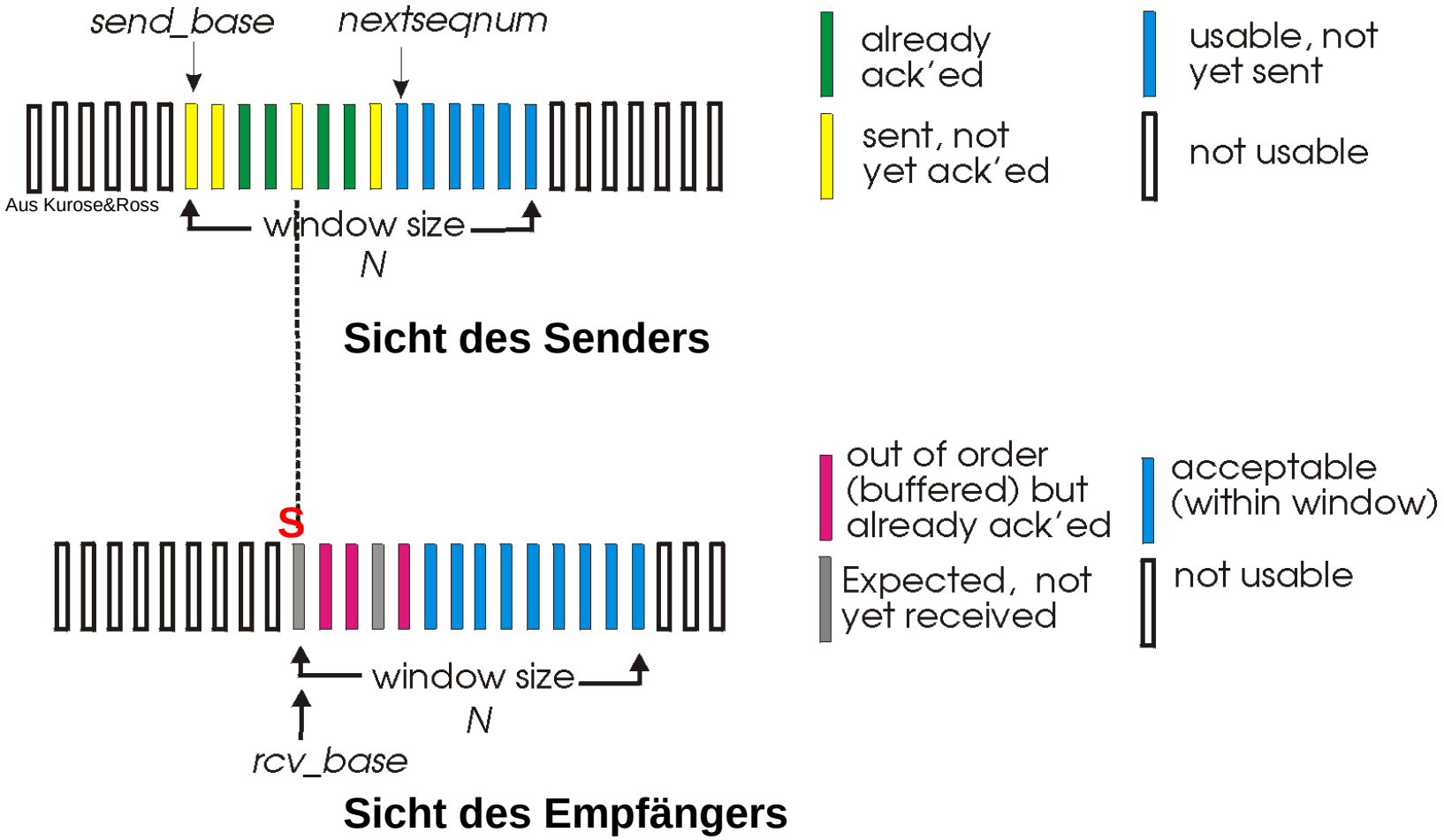
- Empfänger bestätigt mit *größter* Sequenznummer ***S*** für die gilt: **Alle** Pakete mit Sequenznummer $s \leq S$ wurden bereits empfangen.
  - Duplikat-ACKS sind möglich

Selective Repeat: Selective ACKs



Selective Repeat: Selective Acknowledgments

- Retransmission **nur** der verlorengegangenen Pakete
- **Individuelles** Acknowledgment durch Empfänger



Demo: Go-Back-N vs. Selective Repeat

- http://www.ccs-labs.org/teaching/rn/animations/gbn_sr/

Publikums-Joker: Pipelining

Welche Aussage ist ***falsch***?

A. *Go-Back-N* verwendet kumulative ACKs, Selective Repeat verwendet individuelle ACKs.

B. Sowohl *Go-Back-N* als auch *Selective Repeat* sind Stop-and-Wait Verfahren.

C. Sowohl bei *Go-Back-N* als auch bei *Selective Repeat ist* das Sendefenster größer als 1 Paket.

D. Bei Go-Back-N und bei Selective Repeat wird bei einem Timeout mind. 1 Paket erneut gesendet.

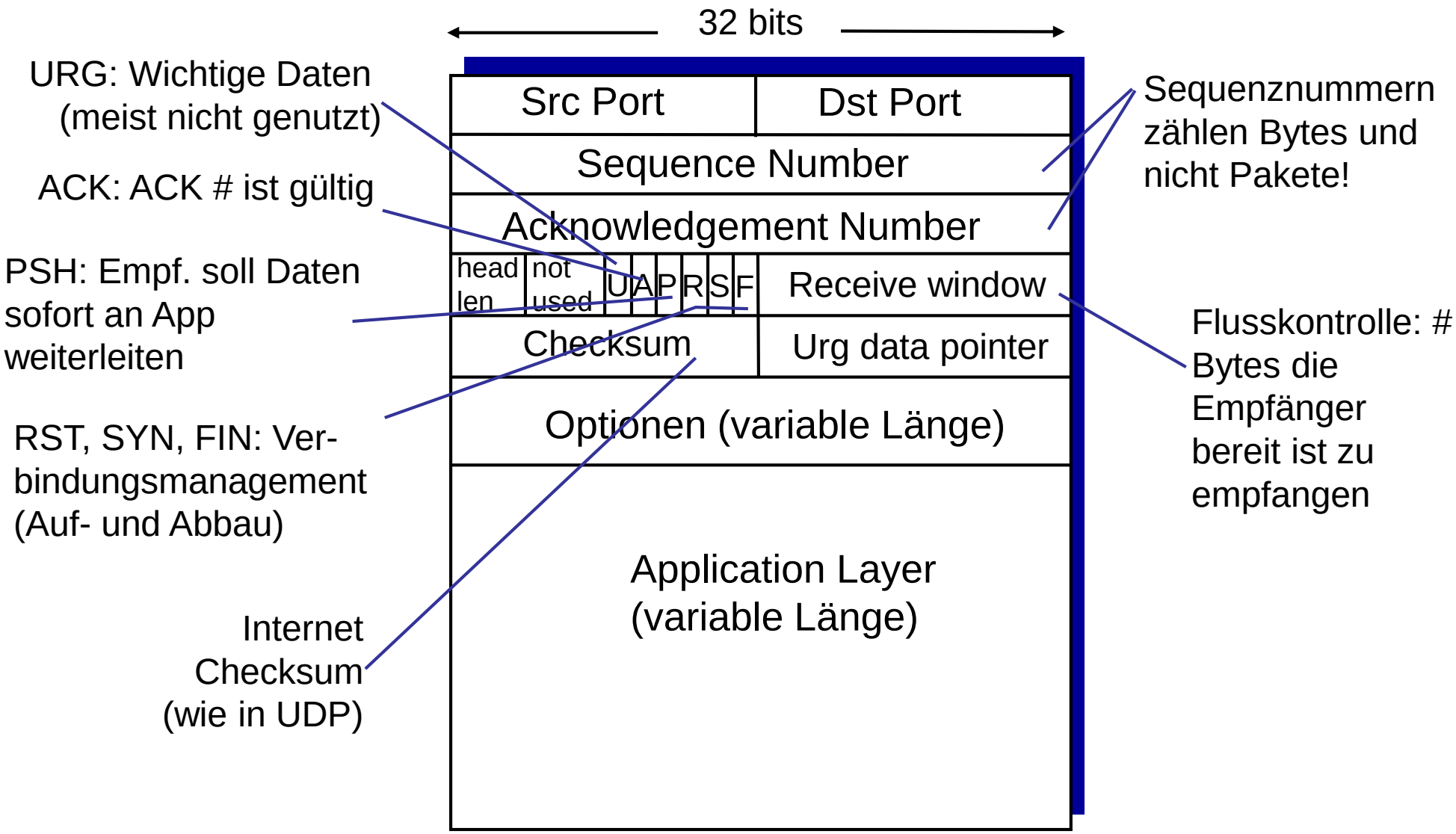
Inhalt

- Allgemeines, UDP
- TCP: Allgemeine Prinzipien
- **TCP: Konkrete Umsetzung**
- Flow und Congestion Control bei TCP
- Network Address Translation (NAT)

TCP: RFCs 793, 1122, 1323, 2018, 2581

- ***Zuverlässige* (engl. "reliable") Datenübertragung**
  - Bestätigt einzelne Bytes, keine Pakete!
- ***Pipelining***
  - Mischung aus Go-Back-N und Selective Repeat.
  - TCP Congestion und Flow Control bestimmen die Fenstergröße (beim Sender)
- ***Vollduplex***
  - Bidirektionale Übertragung innerhalb der gleichen Verbindung
  - MSS (=*Maximum Segment Size*) richtet sich nach MTU (= *Maximum Transmission Unit*) der Link Layer
- ***Verbindungsorientiert***
  - Verbindungsaufbau vor Übertragung.
  - Sender und Empfänger initialisieren ihre „State Machine“
- ***Flow Control*** und ***Congestion Control***

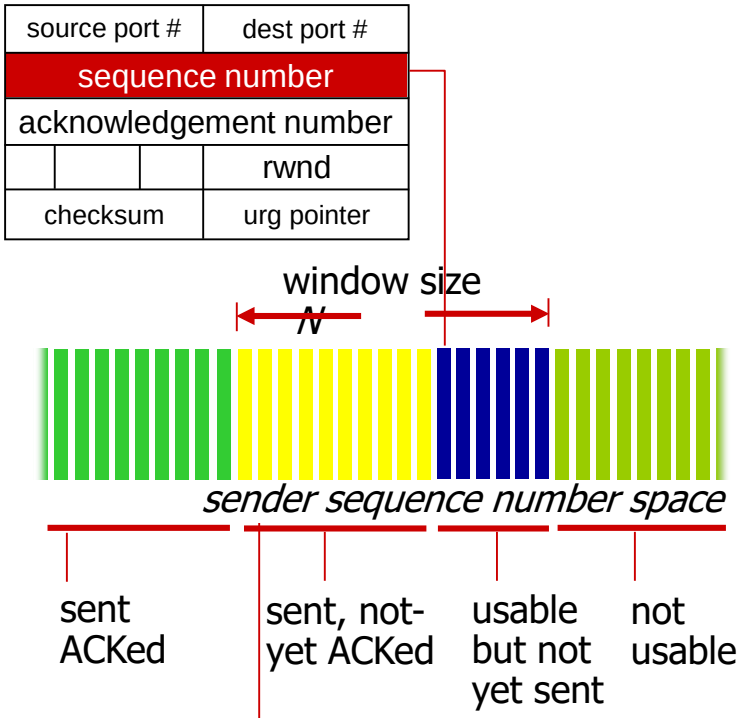
Aufbau eines TCP Segments



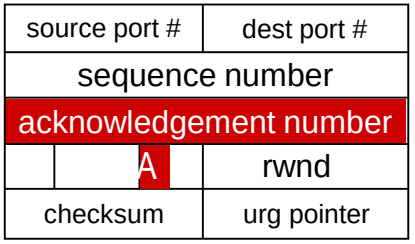
TCP: Sequenz- und Acknowledgmentnummern

- Daten werden als unendlicher Bytestrom aufgefasst.
- **Sequenznummer**
  - Nummer des Bytes im Bytestrom
    - **Nicht Paketnummer wie bisher!**
  - Bytenummer des *ersten* Bytes der Nutzlast.
- **Acknowledgment**
  - Sequenznummer des ***nächsten erwarteten*** Bytes von der Gegenseite
  - **Nicht größte / jüngste Sequenznummer die bereits empfangen wurde wie bisher**
- Bidirektional
  - Sequenznummern in der einen Richtung sind ACK-Nummern in der anderen Richtung.

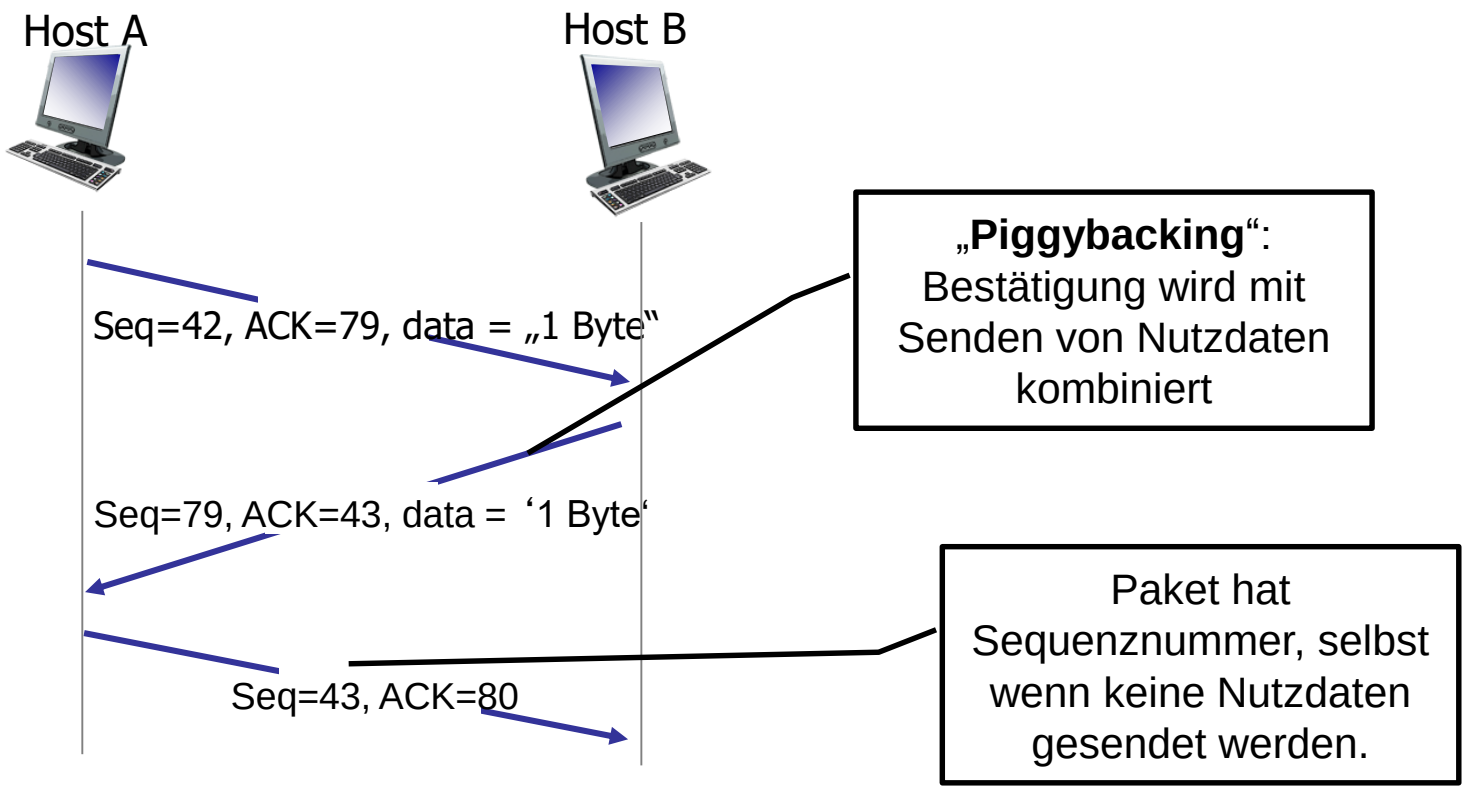
Segment vom Sender zum Empfänger



Antwort vom Empfänger zum Sender



Beispiel: TCP Sequenznummern



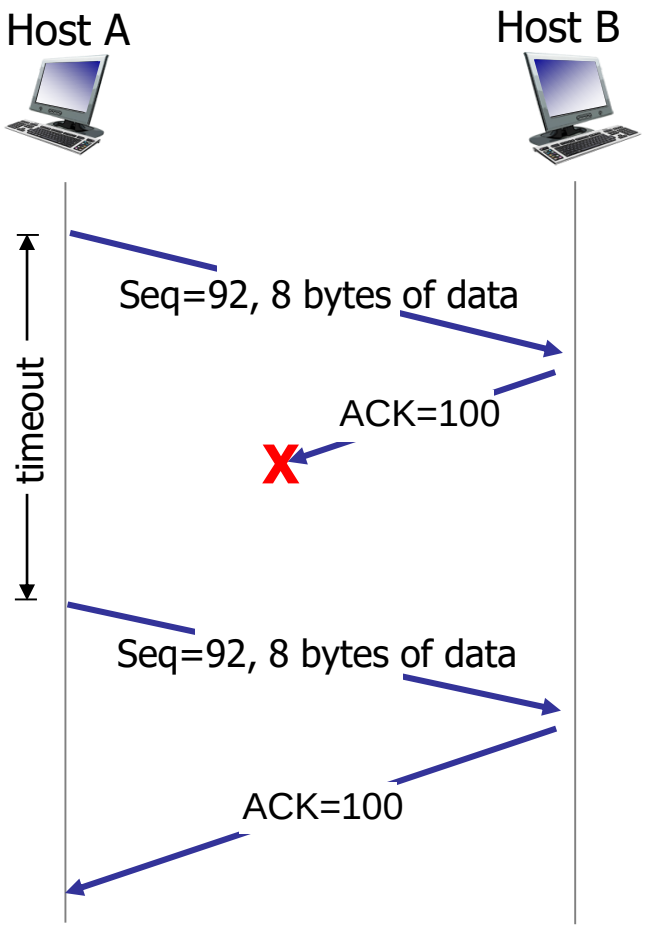
Erkennung von Paketverlusten bei TCP

- Timeout sollte etwas größer sein als die ***Round Trip Time*** (RTT)
  - Langsame Reaktion vs. unnötige Retransmissions

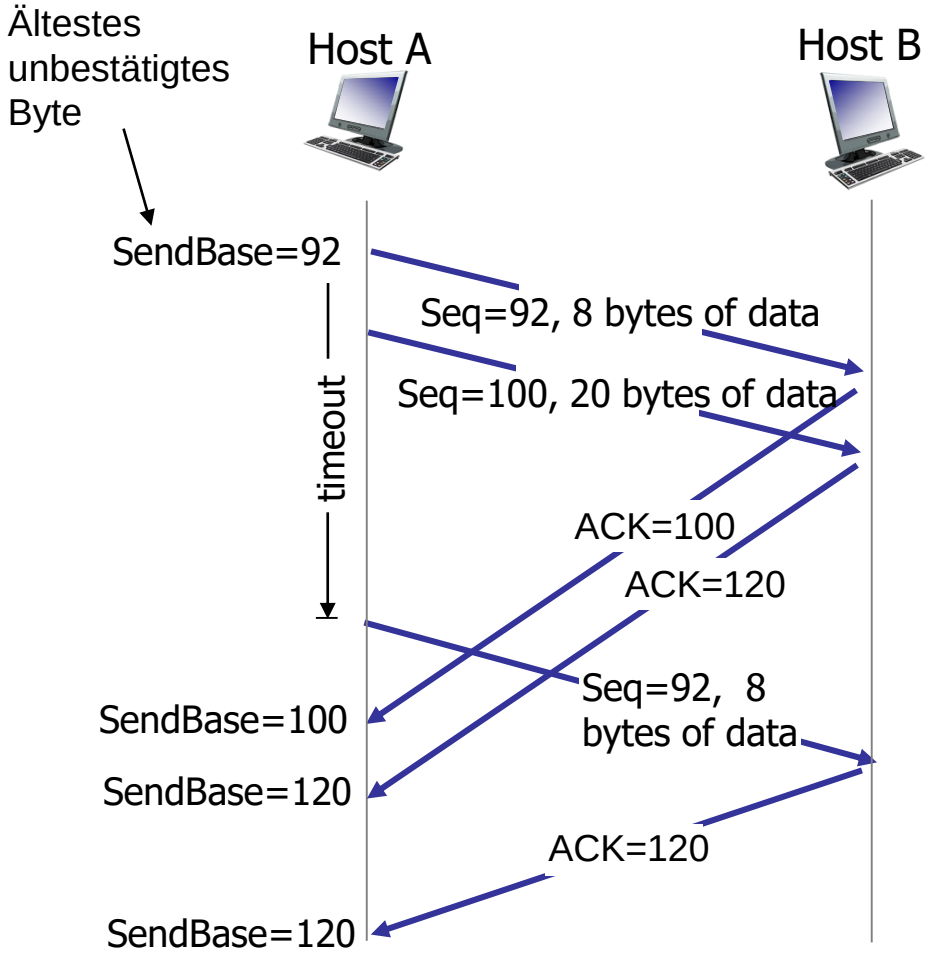
- Wie schätzt man die RTT ab?
  - TCP misst regelmäßig die Zeit zwischen Übertragung eines Segments und Erhalt des entsprechenden ACKs
  - Mittelwert über Messungen

- **TCP ist ein Hybrid zwischen Go-Back-N und Selective Repeat**
  - Von Go-Back-N
    - ACKs sind kumulativ. Jedes ACK bestätigt auch alle vorherigen ACKs.
    - Nur 1 Retransmission-Timer, der sich auf ältestes noch unbestätigtes Segment bezieht.
  - Von Selective-Repeat
    - TCP Empfänger puffern Pakete, selbst wenn ältere Pakete noch ausstehen.
    - Bei Timeout wird nur das verlorengegangene Paket erneut gesendet.

TCP in Aktion (1)



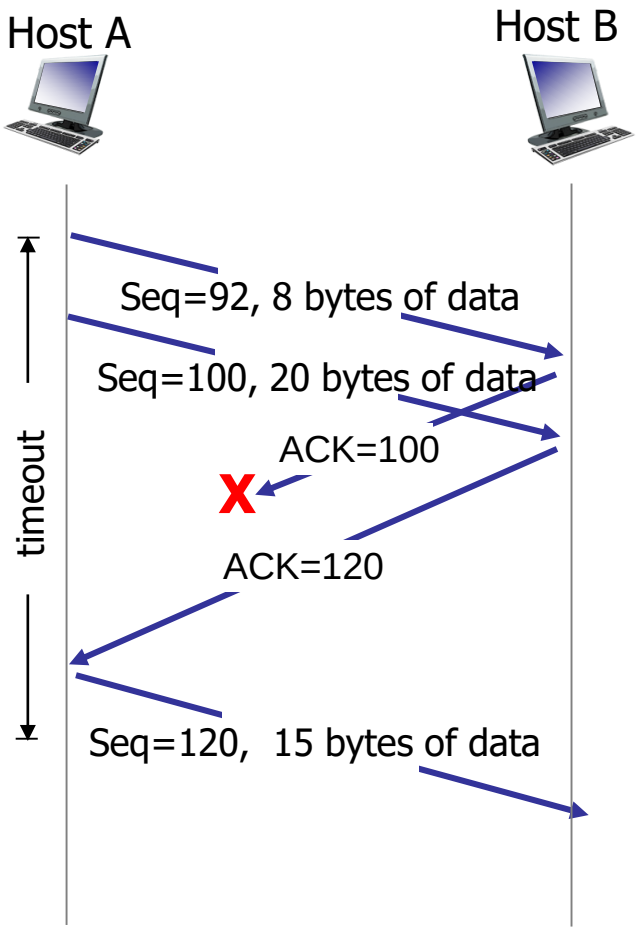
ACK geht verloren



ACKs kommen beide erste nach Timeout

Dennoch keine Retransmission für Seq=100 notwendig!

TCP in Aktion (2)

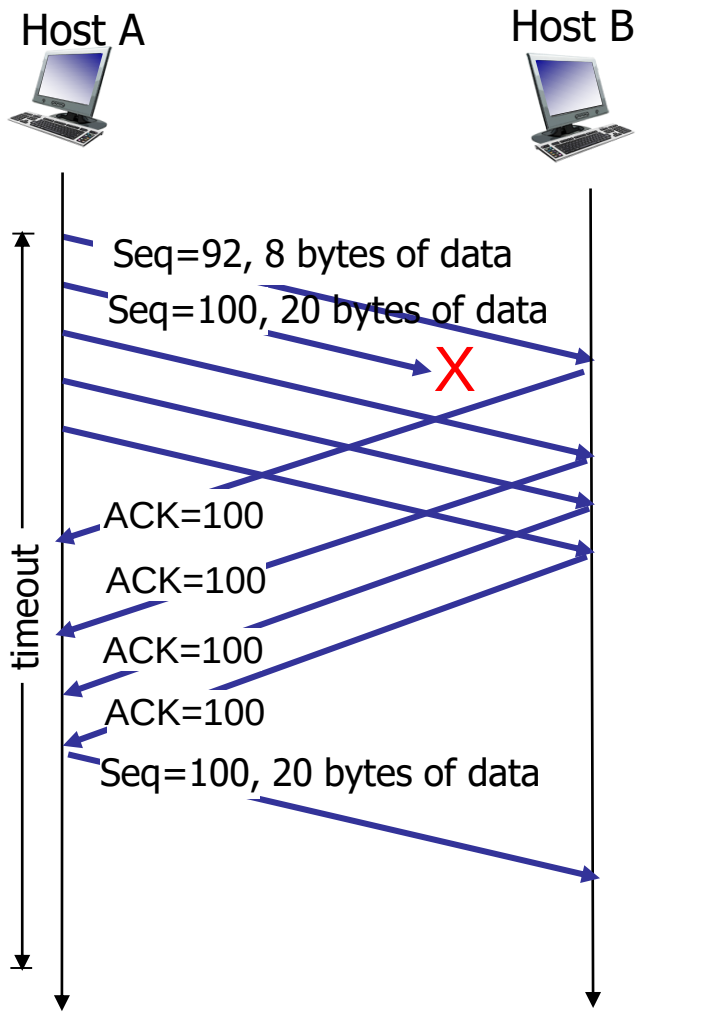


Kumulative ACKs

Für **beide** Segmente keine Retransmission notwendig, da ACK=120 innerhalb der Timeout Periode ankommt und auch Seq=92 mitbestätigt.

TCP Fast Retransmit

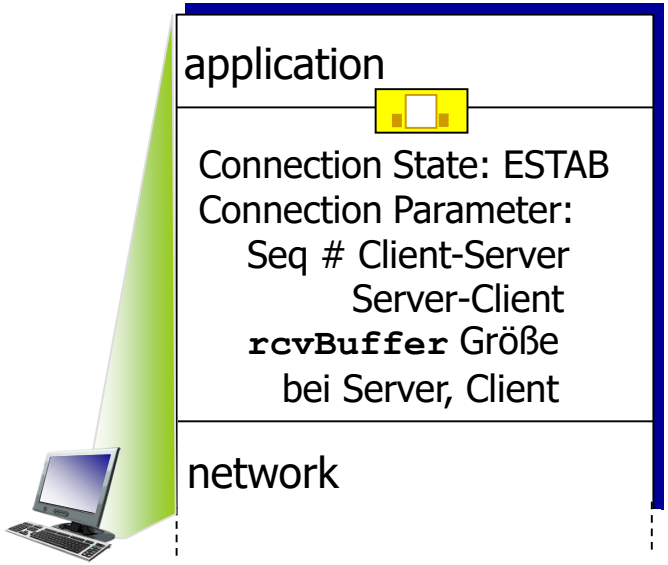
- Timeout Periode ist oft zu lang.
- Deshalb: Retransmission oft schon nach Erhalt von **3 Duplicate ACKs**
  - 3 Duplicate ACK == Indiz für Paketverlust.
- Auch dann: Retransmission des ältesten, unbestätigten Segments
  - Hier: Seq=100



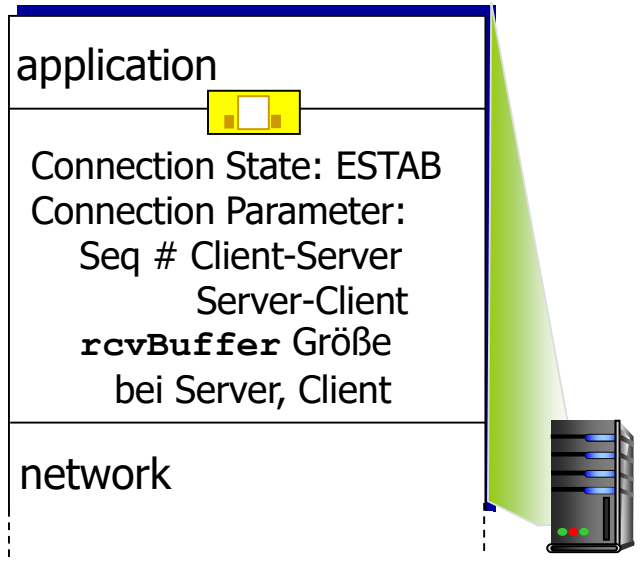
Fast Retransmit nach dem 3. Duplikat ACK

TCP: Aufbau einer Verbindung

- Verbindungsaufbau: Vor Datenaustausch schütteln sich Sender und Empfänger die Hand („**Handshake**“)
  - Einigung über Verbindungsparameter (v.a. initiale Sequenznummern)
  - Beide Seiten erzeugen Puffer für empfangene und zu sendende Daten.
  - Erst danach Datenaustausch möglich.



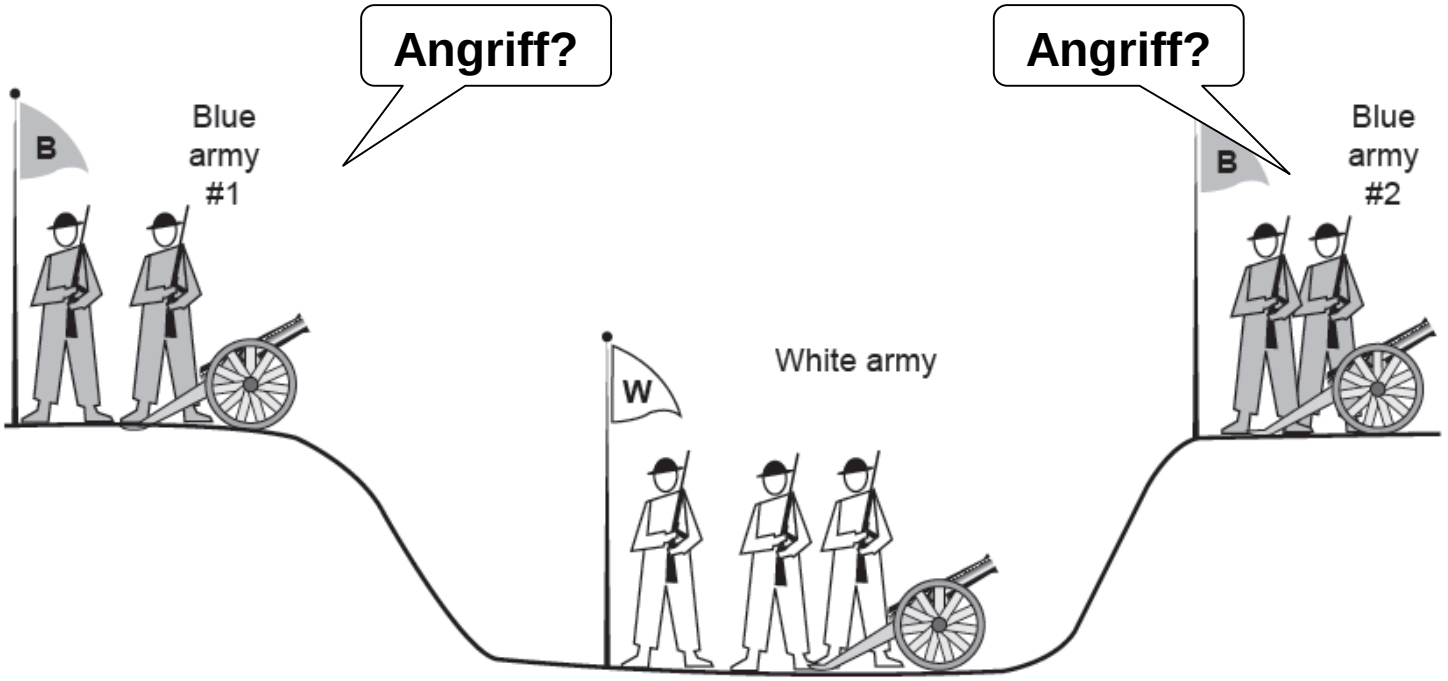

```
Socket clientSocket =
  newSocket("hostname","port
   number");
```




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

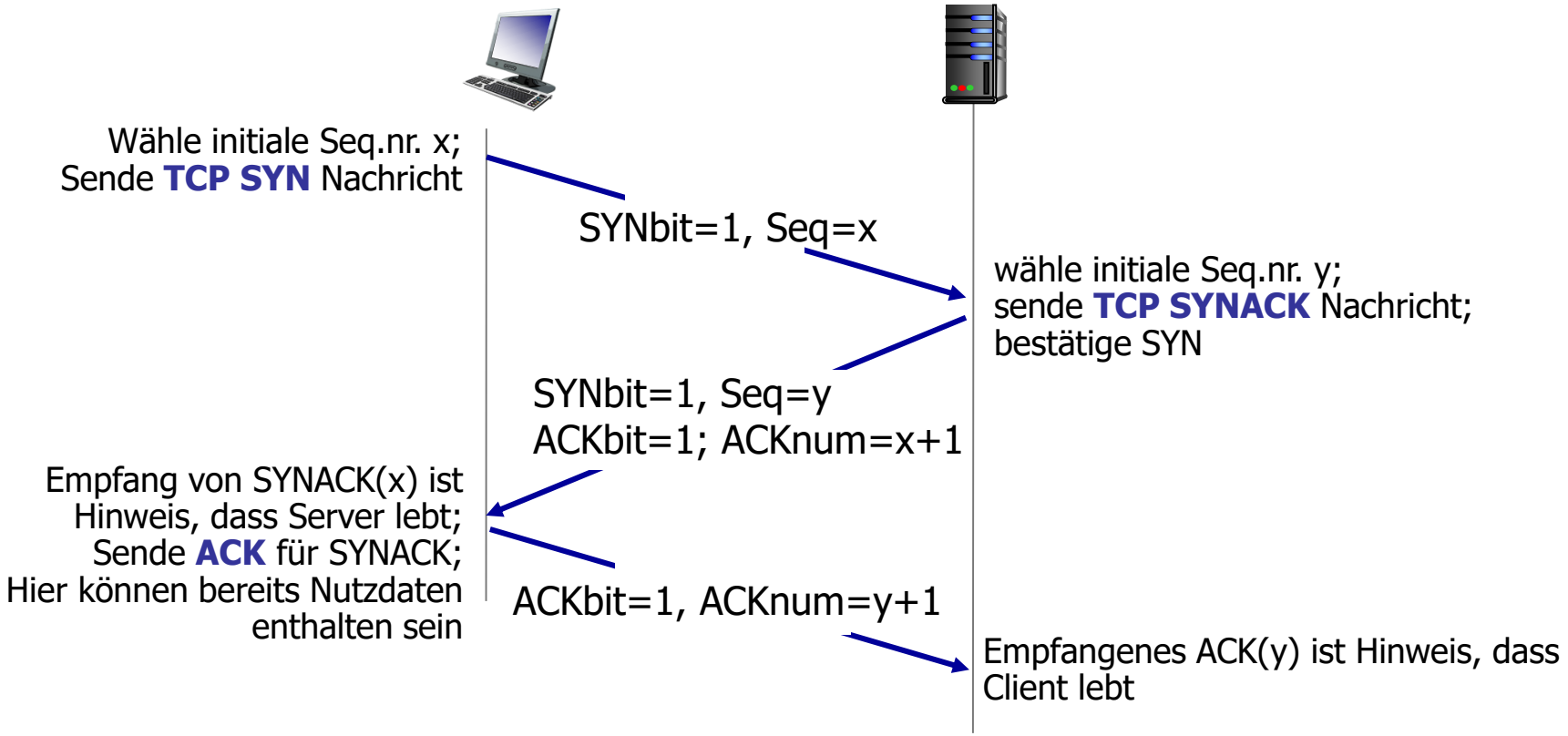
Analogie: 2-Armeen Problem

- Die 2 Teile der blauen Armee können sich nicht **sicher** abstimmen ob sie entweder beide angreifen oder beide ausharren.
- Kommunikation nur durch Boten möglich, der weiße Armee durchqueren muss.
- Angriff wäre nur erfolgreich, wenn beide blauen Armeen gleichzeitig angreifen.



TCP: 3-way Handshake

- Spezielle Steuernachrichten zum Aufbau einer TCP Verbindung
  - TCP Flags SYN: Kennzeichnet erstes Paket für jede Richtung
  - TCP Flag ACK: Falls gesetzt, zeigt an, dass ein vorheriges Paket bestätigt wird.
- Verbindungsaufbau == Overhead!



Publikums-Joker: Link Layer

Der Wert des Acknowledgement-Feldes in einem Segment bedeutet:

A. Anzahl der bereits empfangenen Bytes

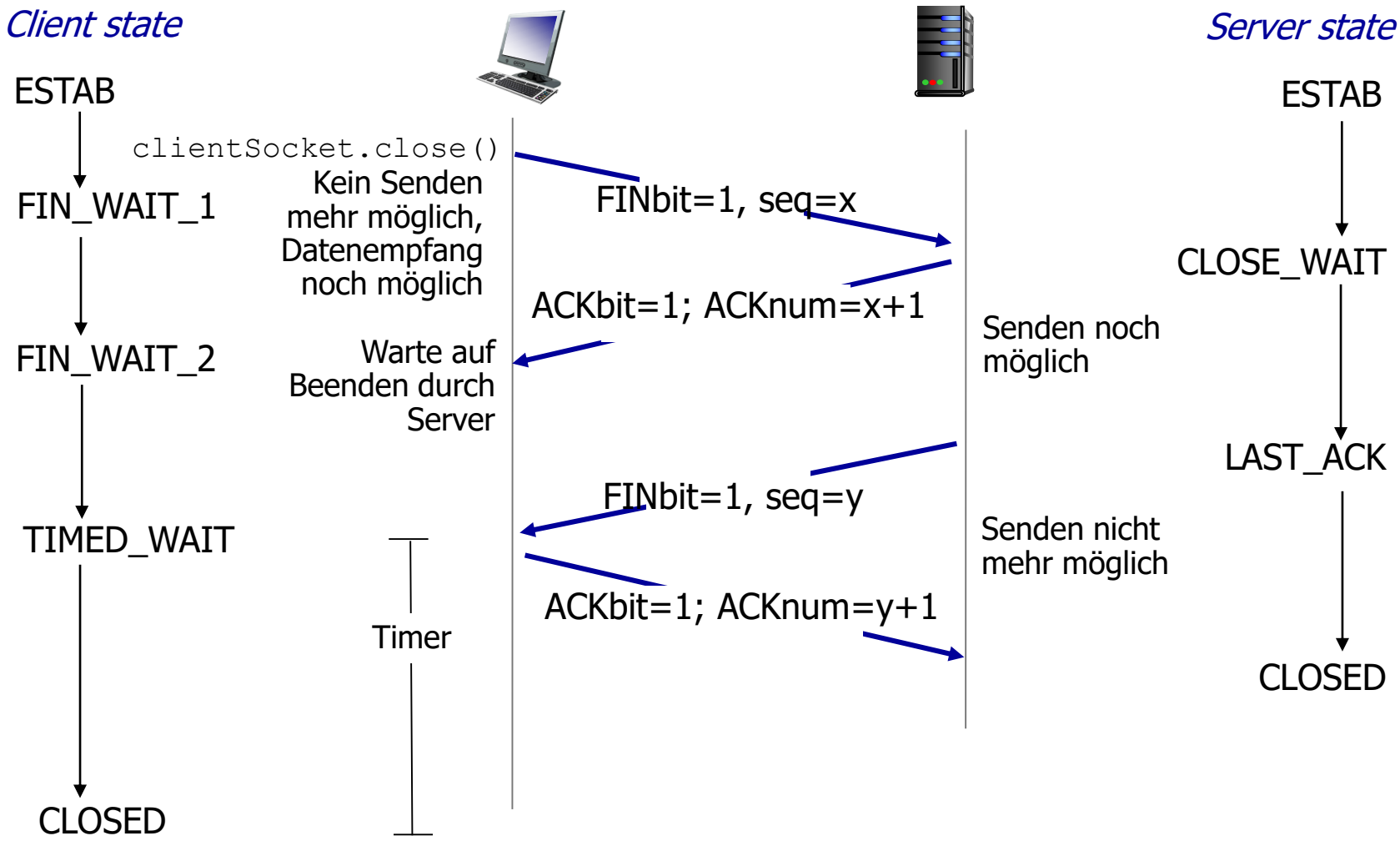
B. Gesamtzahl der zu empfangenden Bytes.

C. Nummer des nächsten zu empfangenden Bytes.

D. Sequenz von 0er und 1er.

TCP Verbindungsabbau

- Geordneter Abbau: FIN Flag

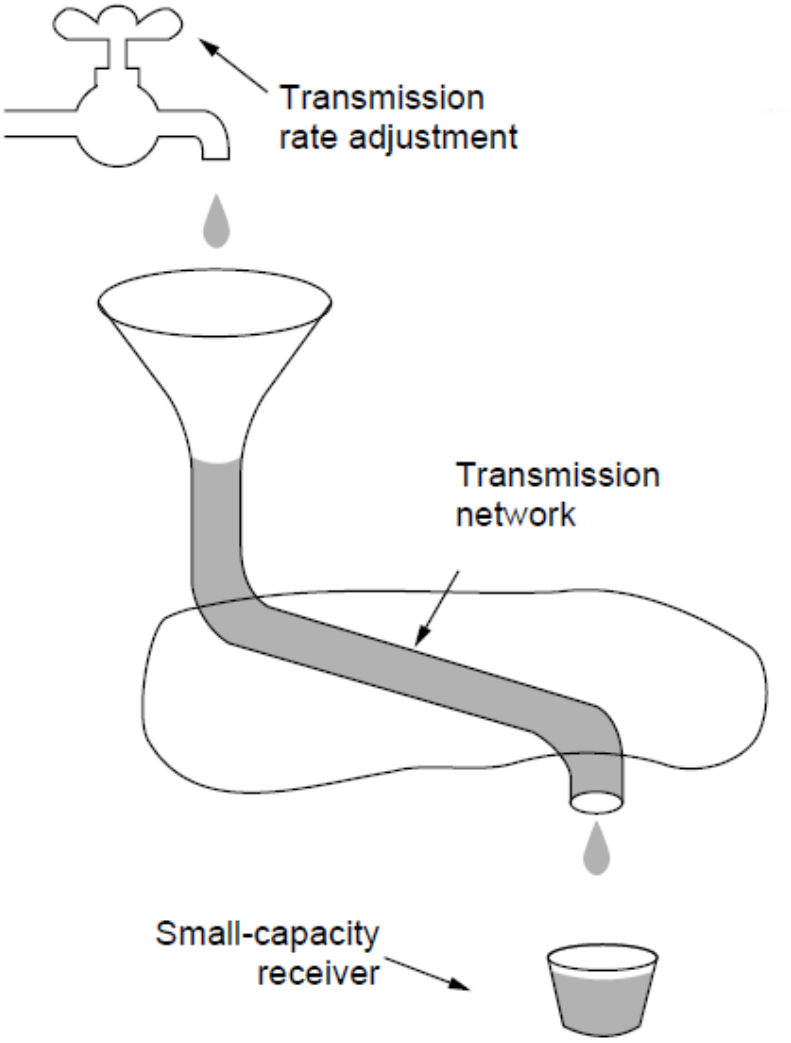


Inhalt

- Allgemeines, UDP
- TCP: Allgemeine Prinzipien
- TCP: Konkrete Umsetzung
- **Flow und Congestion Control bei TCP**
- Network Address Translation (NAT)

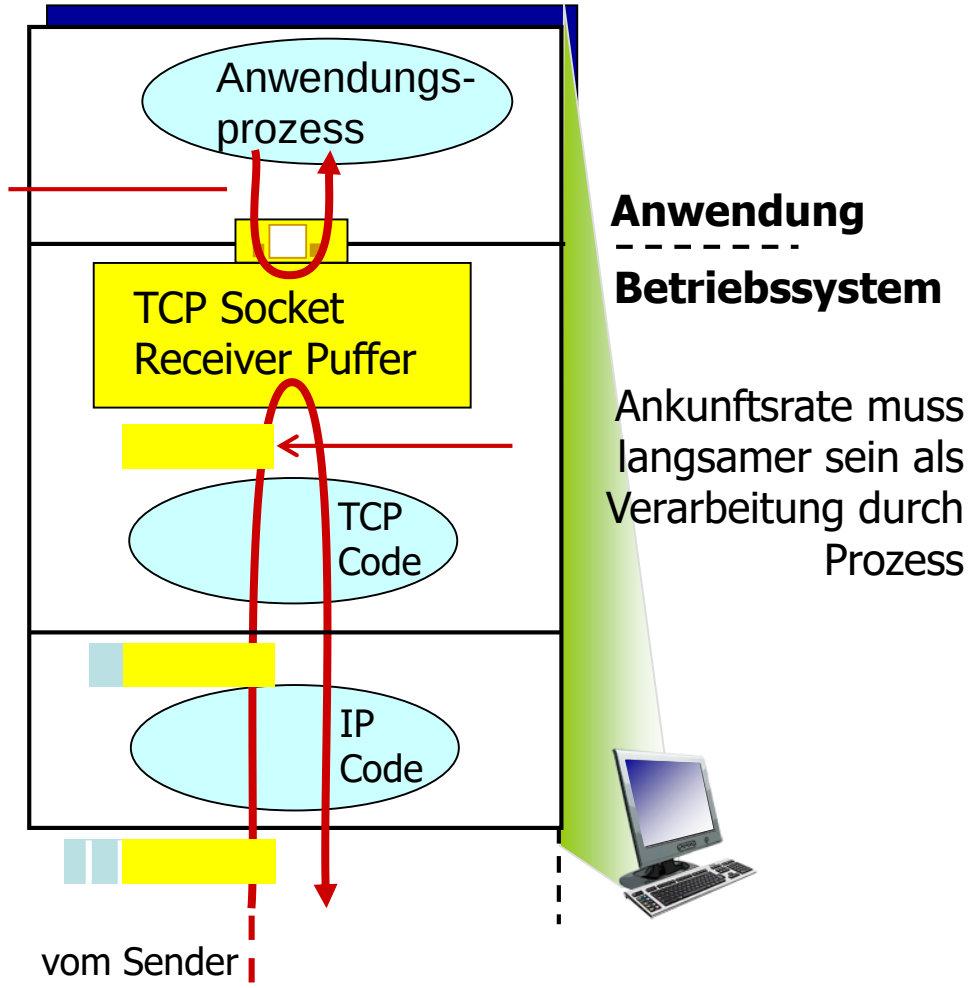
Anpassung der TCP Senderate

- Der Sender muss seine Geschwindigkeit verringern falls
  - ***Empfänger*** nicht schnell genug ist → **Flow Control**
  - ***Netzwerk*** nicht schnell genug ist → **Congestion Control**



Flow Control (dt. „Flußsteuerung)

- ***Problem***
  - Anwendungsprozess im Empfänger holt Daten zu langsam oder zu spät aus TCP Empfangspuffer ab.
- ***Reaktion des TCP Senders***
  - Verringern der Senderate durch Anpassen des Sendefensters.
- ***Ansatz***
  - Empfänger teilt in *jedem* Paket mit, wieviel Platz noch in seinem Puffer ist.
  - Sender verkleinert daraufhin Sendefenster.

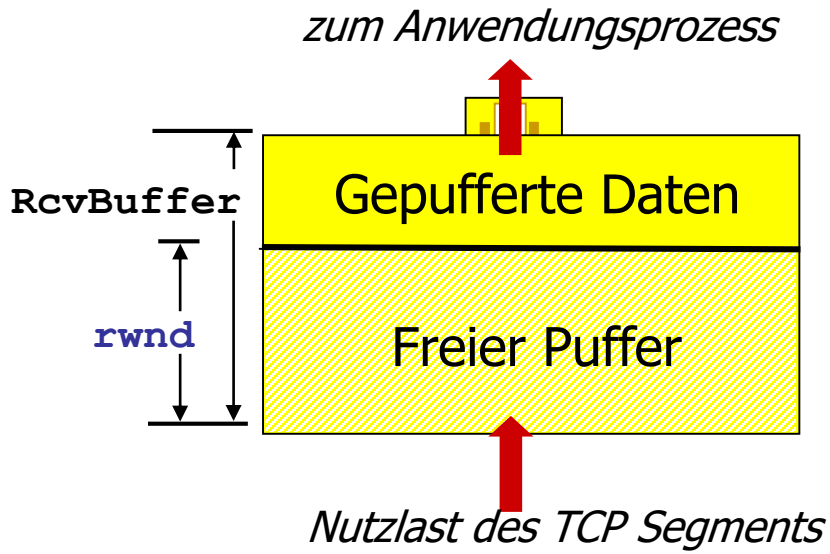


Protokollstack des Empfängers

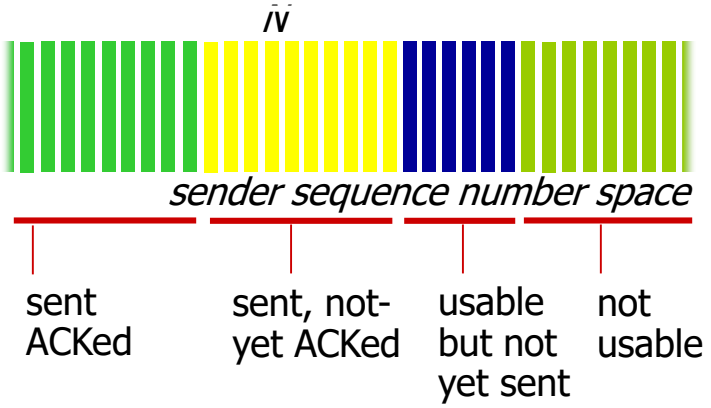
Flow Control

- **Größe des Puffers:** `RcvBuffer`
  - Meist über Socketoptionen konfigurierbar.
  - Default-Wert durch OS vorgegeben.
- **Freier Puffer:** **`rwnd`**
  - `RcvBuffer – [LastByteRcvd – LastByteRead]`
- Empfänger teilt `rwnd` dem Sender im TCP Header mit.
- Sender verwendet `rwnd` als ***obere Schranke*** für Sendefenster
  - Menge der aktuell unbestätigten Daten muss kleiner als `rwnd` sein

Puffer bei Empfänger



Window Size bei Sender



Congestion Control (dt. Staukontrolle)

- **Tradeoff**
  - Großes Sendefenster → mögliche Überlastung des Netzwerks!
  - Kleines Sendefenster → geringe Datenrate

- **2 Ansätze** für Congestion Control / Anpassung der Senderate
  - **Netzwerk-unterstützt:** *Explicit Congestion Notification (ECN).*
    - Router geben Rückmeldung an Hosts bei Überlastung
    - Immer noch selten eingesetzt.
  - **Ende-zu-Ende:** *"Implizit"*
    - Netzauslastung durch Beobachten der RTTs und Auftreten von Paketverlusten abgeschätzen.
    - TCP wählt diesen Ansatz!

- **Unzählige Varianten**
  - https://en.wikipedia.org/wiki/TCP_congestion_control
  - Im Folgenden wird exemplarisch **TCP Reno** betrachtet.

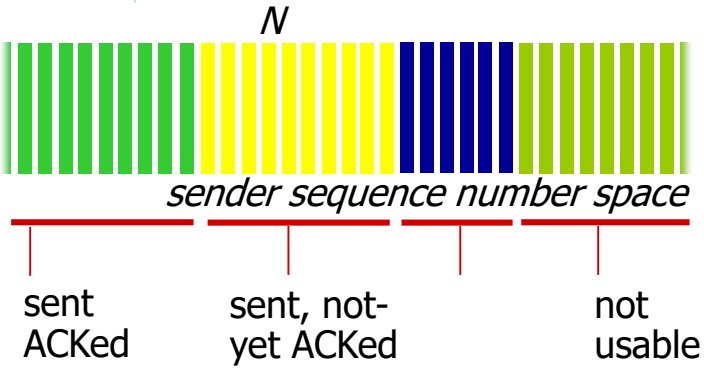
Congestion Control: Wie Senderate begrenzen?

- Idee: Begrenze erlaubte Anzahl der unbestätigten Pakete (=**Sendefenster**)
  - **Window Size** ≤ min{**cwnd**, rwnd}
  - Großes `cwnd` bedeutet hohe Datenrate wegen geringer Netzauslastung.

- **`cwnd`** wird als Funktion der Netzauslastung berechnet.
  - Je höher Netzauslastung durch Sender „geschätzt“ wird, desto kleiner `cwnd`

- Sowohl Flow als auch Congestion Control beeinflussen Sendefenster!

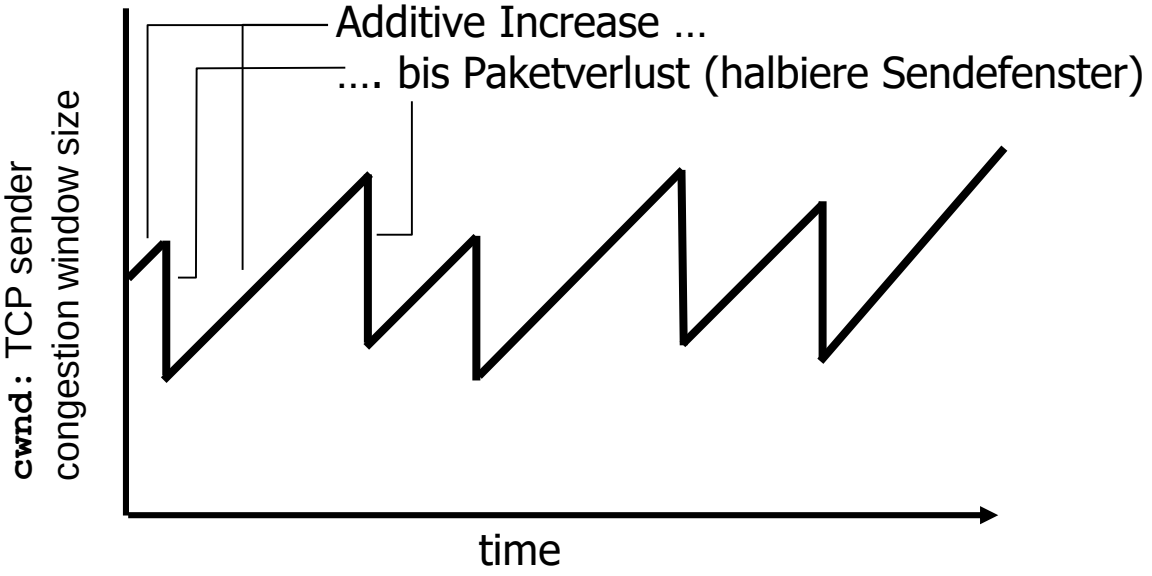
Window Size bei Sender



TCP Congestion Control

- **Ansatz:** Sender versucht verfügbare "Bandbreite" zu erkennen
  - Vergrößere Datenrate (=Größe Sendefensters) bis Paketverluste auftreten.
- **Additive Increase**
  - Vergrößere nach jeder *Round Trip Time* (RTT) Congestion Window (`cwnd`) um 1 *Maximum Segment Size* (MSS) bis Paketverlust erkannt wird.
- **Multiplicative Decrease**
  - Halbiere `cwnd` nach erkanntem Paketverlust.

"Sägezahn"



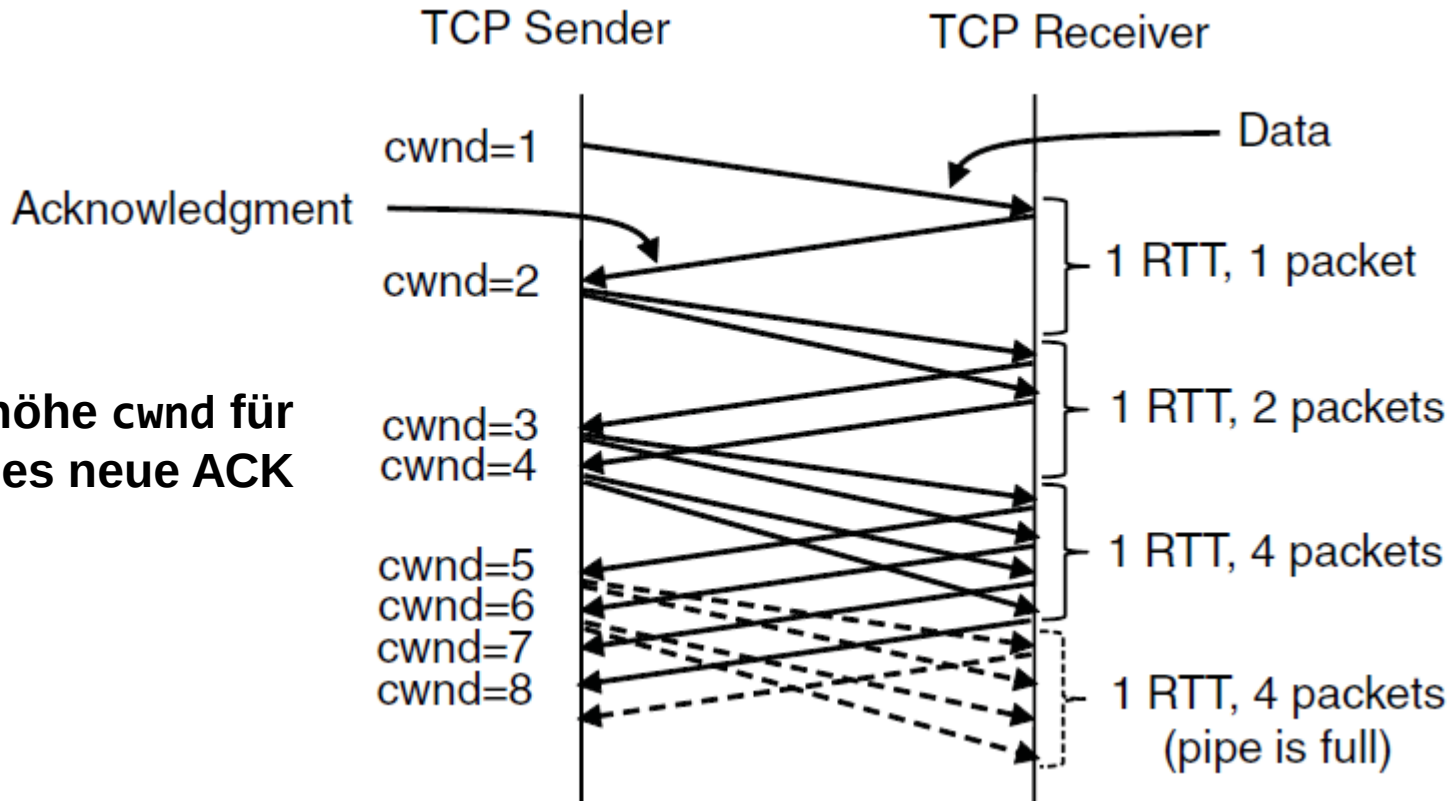
TCP Reno

- TCP Sender **erkennt** „Congestion“ *indirekt* über beobachtete Paketverluste
  - ***Timeout:*** Paket wird nicht rechtzeitig bestätigt
  - ***3 Duplicate ACKs,*** d.h. ACKs, die das gleiche Paket bestätigen

- **Verhalten: 3 Phasen**
  - ***Slow Start / exponentiell****:* Verdoppele `cwnd` nach jeder *Round Trip* Time bis zu einem gewissen Schwellwert (**`ssthresh`**)
  - ***Congestion Avoidance / linear****:* Dann vergrößere `cwnd` um 1 nach jeder *Round Trip Time.*
  - **Bei Paketverlust / Problem:** Halbiere `cwnd`.

TCP Reno: Slow Start

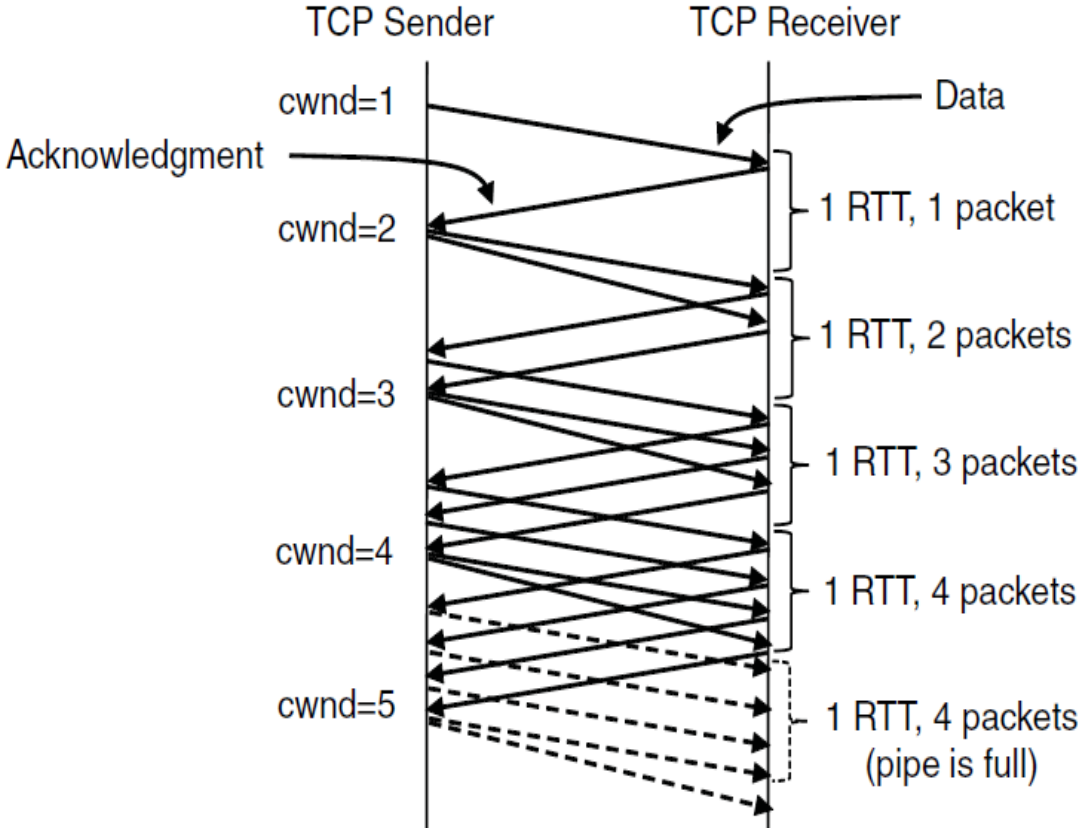
- Starte mit `cwnd`=1 MSS (Maximum Segment Size)
- Verdopple `cwnd` nach jeder RTT
  - Anders ausgedrückt: Vergrößere um 1 MSS nach ***jedem*** erhaltenen ACK.
- "Slow Start" vergrößert `cwnd` damit xponentiell



TCP Congestion Avoidance

- ***TCP Congestion Avoidance***
  - Tritt ein nach Erreichen eines Thresholds **`ssthresh`**
    - == halber Wert von `cwnd` während letzter Congestion.
  - nach 3 Duplicate ACKs.

- **Additive Increase**
  - `cwnd` wird jede RTT-Periode um 1 erhöht
    - Nicht bei jedem ACK!
  - Anders ausgedrückt: Jedes Paket erhöht `cwnd` um `1/cwnd`



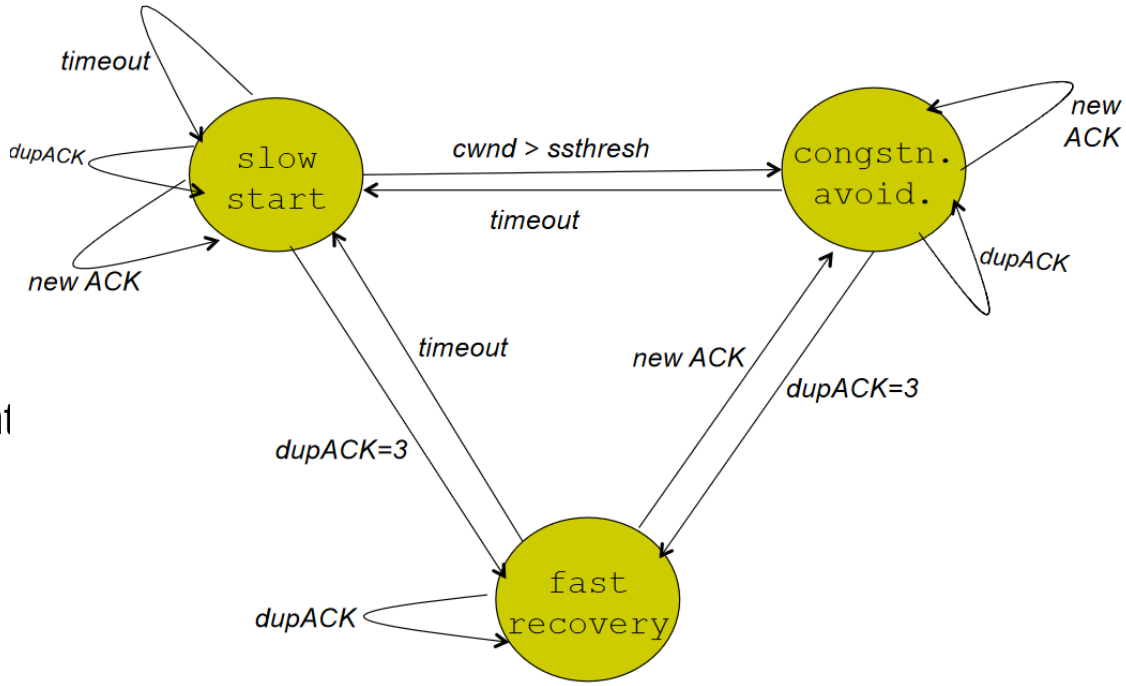
Normalerweise beginnt Congestion Avoidance nicht bei cwnd=1!

Verhalten bei Paketverlusten (TCP Reno)

- **Hinweis**
  - **`ssthresh:`** Grenze zwischen "Slow Start" und Congestion Avoidance"
  - Speichert halben Wert von `cwnd` während der letzten aufgetretenen Congestion (=Paketverlust / Duplicate ACK).

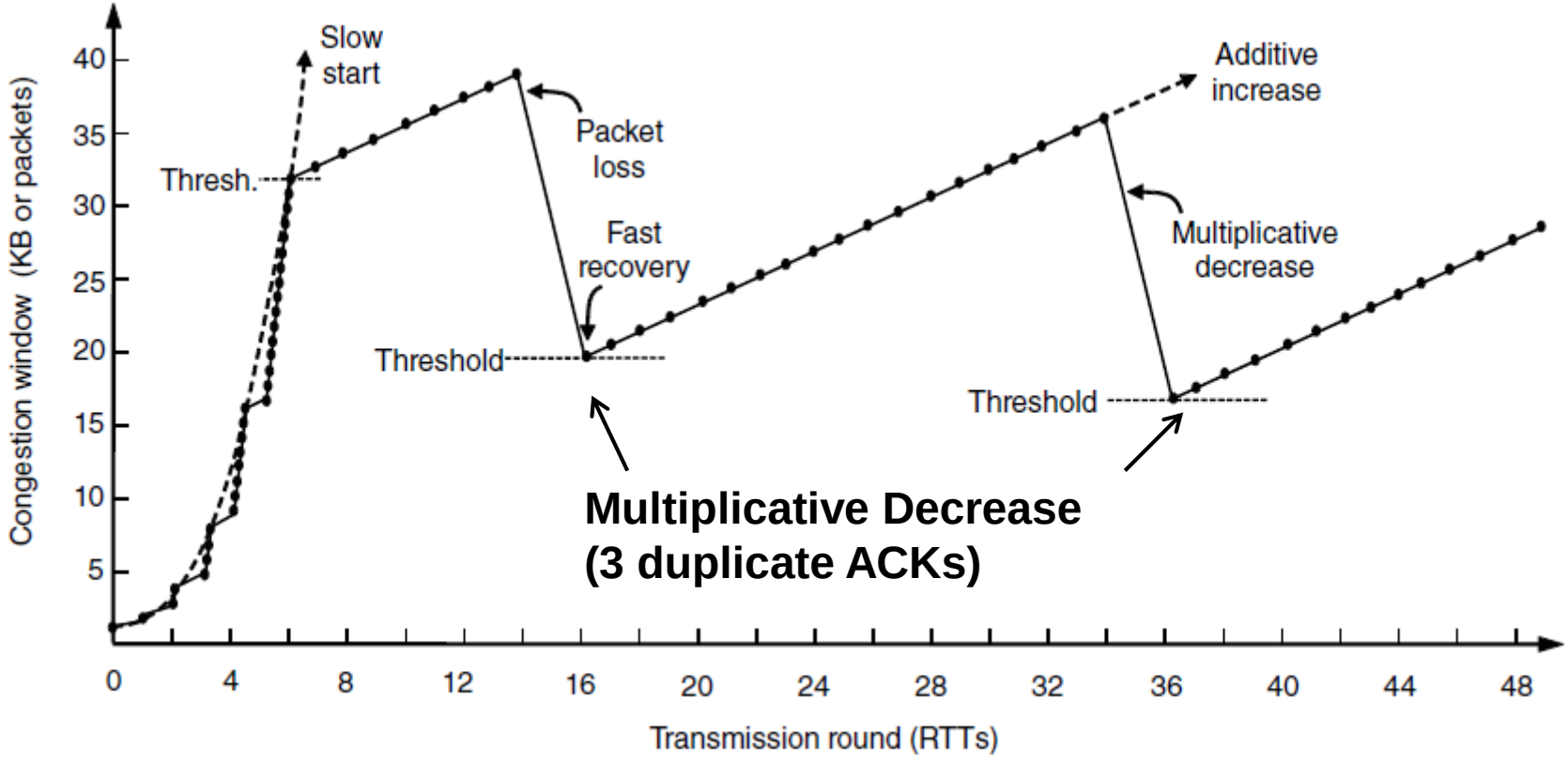
- **Nach Timeout**
  - TCP Slow Start
  - `ssthres = cwnd/2`
  - `cwnd = 1`

- **Nach 3 Duplicate ACKs**
  - Congestion Avoidance
  - Hinweis, dass Netz zumindest nicht vollständig überlastet ist.
  - `ssthres = cwnd/2`
  - `cwnd = cwnd/2 + 3`



TCP Reno: Fast Retransmit

- Nach 3 Duplicate ACKs
  - Setze `ssthresh` auf `cwnd`/2
  - Halbiere `cwnd` und wechsle in *Additive Increase* Modus.



Publikums-Joker: Congestion Control

Welche Aussage ist ***falsch***?

A. Falls zusätzlich Flow Control verwendet wird, dann bestimmt alleine die Flow Control die erreichte Datenrate.

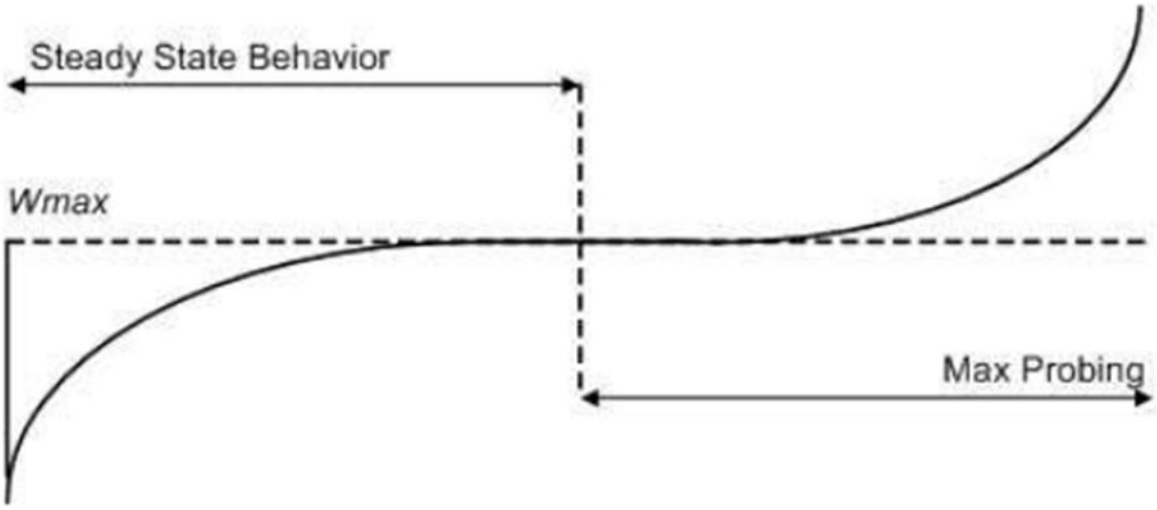
B. Der Sender erkennt eine mögliche Netzüberlastung nur durch TCP Timeouts.

C. TCP Teilnehmer, die unterschiedliche Algorithmen für Congestion Control verwenden, können dennoch miteinander sprechen.

D. Mit UDP lassen sich schneller höhere Datenraten erreichen.

Ausblick: TCP Cubic

- Ersetzt zunehmend TCP Reno
  - TCP Cubic verwendet häufig weiterhin klassischen "Slow Start"
  - Das "Additive Inrease" wird mehr oder weniger ersetzt.
- Optimiert für Netze mit hohen Bandbreiten und ***hoher*** Latenz
- CUBIC vergrößert Sendefenster abhängig von verstrichener Zeit seit letzter Congestion
  - Nicht bei jedem empfangenen ACK etc!
  - Bevorzugt im Gegensatz zu TCP Reno keine Flows mit kurzen RTTs.
- Sendefenster ist eine kubische Funktion.
  - Man verweilt länger bei optimaler Größe des Sendefensters.

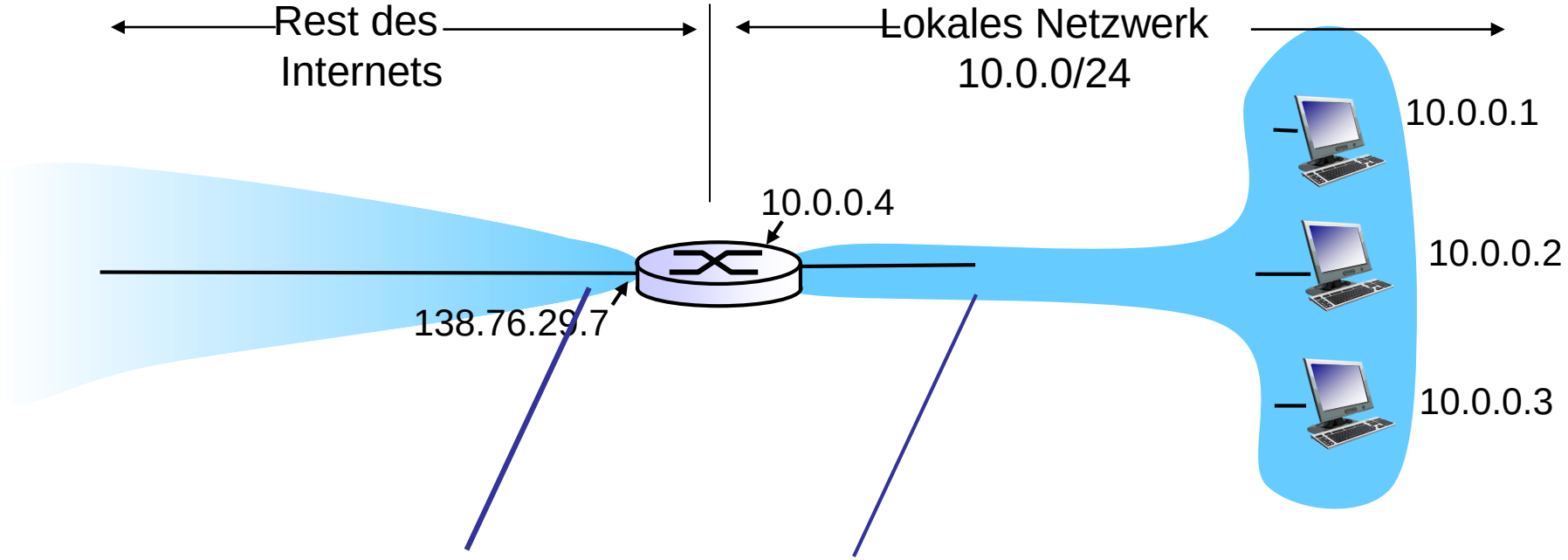


Inhalt

- Allgemeines, UDP
- TCP: Allgemeine Prinzipien
- TCP: Konkrete Umsetzung
- Flow und Congestion Control bei TCP
- **Network Address Translation (NAT)**

NAT: Network Address Translation

- Wie kann man sich öffentliche IP Adressen teilen?



Alle Datagramme, die das lokale Netz verlassen, haben die ***gleiche Src IP*** Adresse, aber ***verschiedene Src Ports***

Datagramme von/zu Hosts in diesem Netzwerk haben Quell- bzw. Ziel IP Adressen aus einem privaten Adressbereich

Motivation

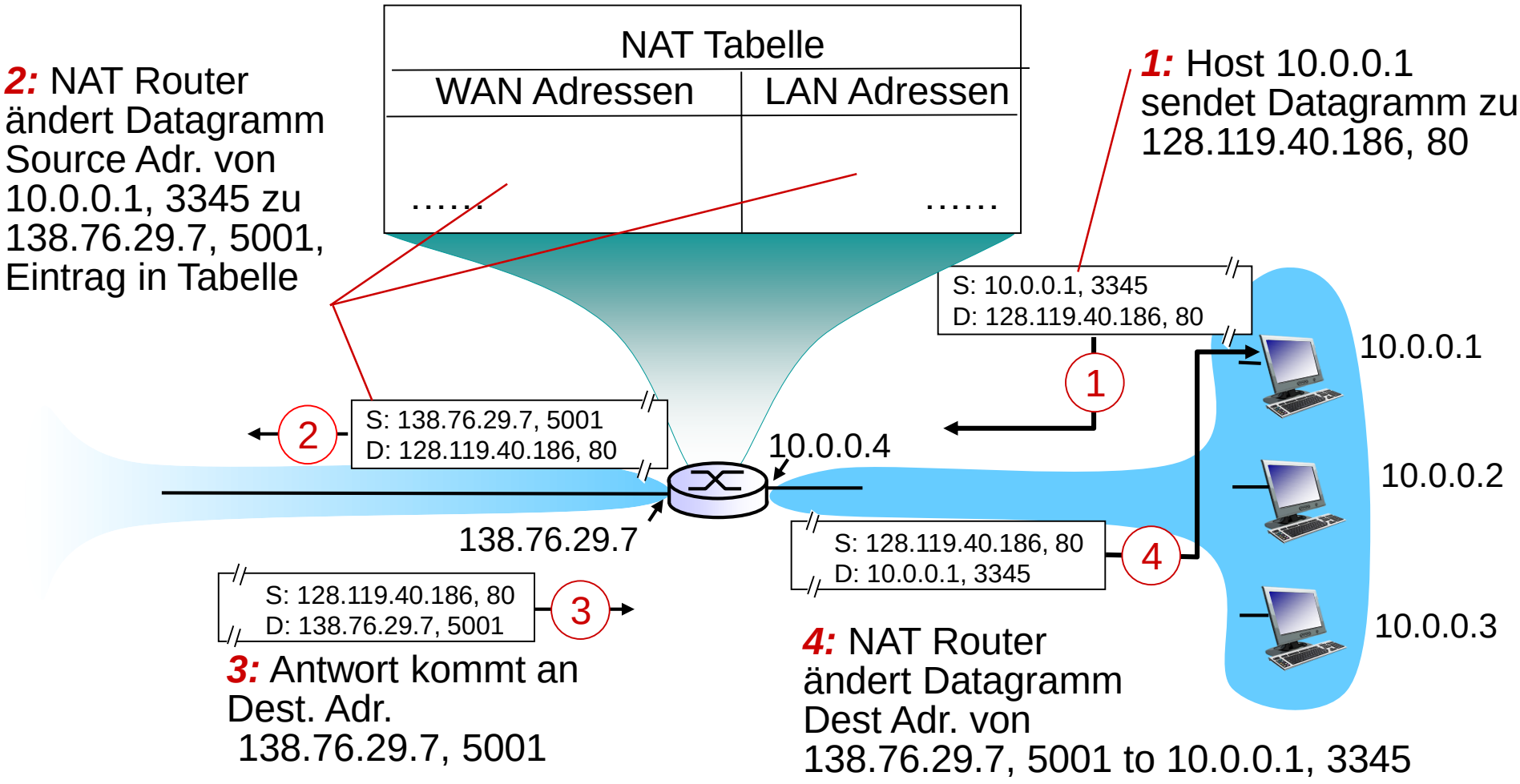
- Lokales Netzwerk benutzt nur 1 IP Adresse, um mit dem Rest des Internets zu kommunizieren.

- **Vorteile**:
  - Einsparen von IP Adressen: Es genügt wenn Internet Service Provider (ISP) nur 1 Adresse zuweist.
  - Man kann alle IP Adressen im lokalen Netzwerk ändern ohne den Rest der Welt zu informieren.
  - Man kann den ISP wechseln ohne seine IP Adressen im lokalen Netz zu ändern.
  - Geräte innerhalb des lokalen Netzwerks sind nicht direkt adressierbar aus dem Rest des Internets (Plus an Sicherheit).

Implementierung eines NAT Routers

- Ausgehende Datagramme (ins Internet)
  - Ersetze **(Source IP, Port #)** durch **(NAT IP, neue Port #)**
  - Entfernte Hosts antworten mit (NAT IP Adresse, neue Port #) als Zieladresse
- NAT Translation Table
  - Jeder NAT Router merkt sich folgende Zuordnung:
  - **(Source IP, Port #)** zu **(NAT IP, neue Port #)**
- Ankommende Datagramme (aus dem Internet)
  - Schlage in NAT Translation Table nach
  - Ersetze (NAT IP Adresse, neue Port #) im Zielfeld jedes ankommenden Datagramms mit (Source IP, Port #)

NAT: Network Address Translation



NAT: Network Adress Translation

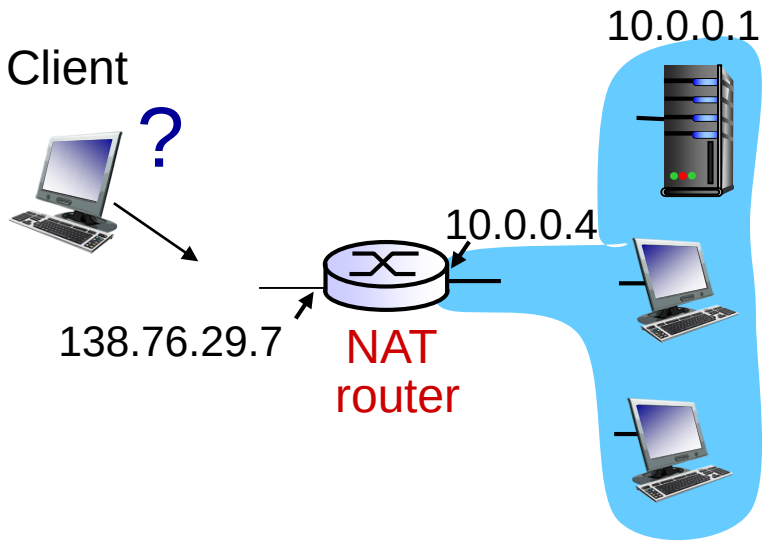
- 16-Bit Portnummer-Feld
  - > 60000 gleichzeitige Verbindungen mit 1 öffentlichen IP

- NAT ist umstritten
  - Routern sollten die Schicht 4 (Portnummer) nicht berücksichtigen!
  - Einfluss auf Ende-zu-Ende Beziehung: Hosts im lokalen Netz können von außen nicht adressiert werden
  - IPv4 Adressmangel sollte lieber durch IPv6 gelöst werden.

NAT Traversal (1)

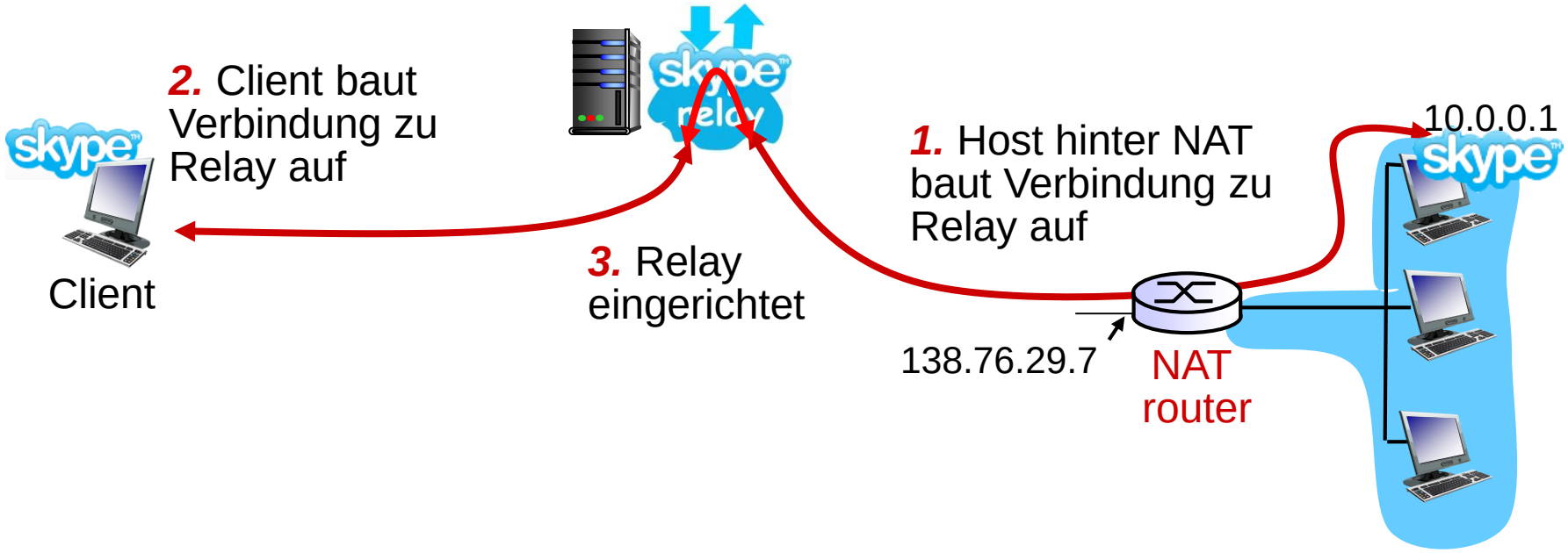
- Client möchte sich zu Server mit der Adresse 10.0.0.1 verbinden
  - Server Adresse 10.0.0.1 ist privat, nicht als Zieladresse verwendbar. werden
  - Nur 1 externe NAT Adresse sichtbar: 138.76.29.7

- **Port Forwarding**
  - Statisches Weiterleiten: Verbindungsanfragen an einen bestimmten Port werden fest zu bestimmten Server weitergeleitet.
  - Beispiel: (123.76.29.7, Port 2500) wird immer zu (10.0.0.1, Port 25000) weitergeleitet.
  - Manuelle Konfiguration notwendig



NAT Traversal (2)

- **Relaying**
  - Wird beispielsweise bei Skype verwendet
  - Client hinter NAT verbindet sich mit Relay
  - Externer Client verbindet sich mit Relay



Zusammenfassung

- Schicht 4 ist Prozess-zu-Prozess Kommunikation, Port Multiplexing
- Verbindungslose Kommunikation: UDP
- Prinzipien der verbindungsorientierten, zuverlässigen Kommunikation
  - Keine Bitfehler, Einhaltung der Reihenfolge, kein Datenverlust
  - Stop-and-Wait, Go-Back-N, Selective Repeat
- Transmission Control Protocol
  - Verbindungsauf- und abbau, Sequenznummern
- Flow und Congestion Control bei TCP
  - `rwnd` und `cwnd`
- Network Address Translation (NAT)