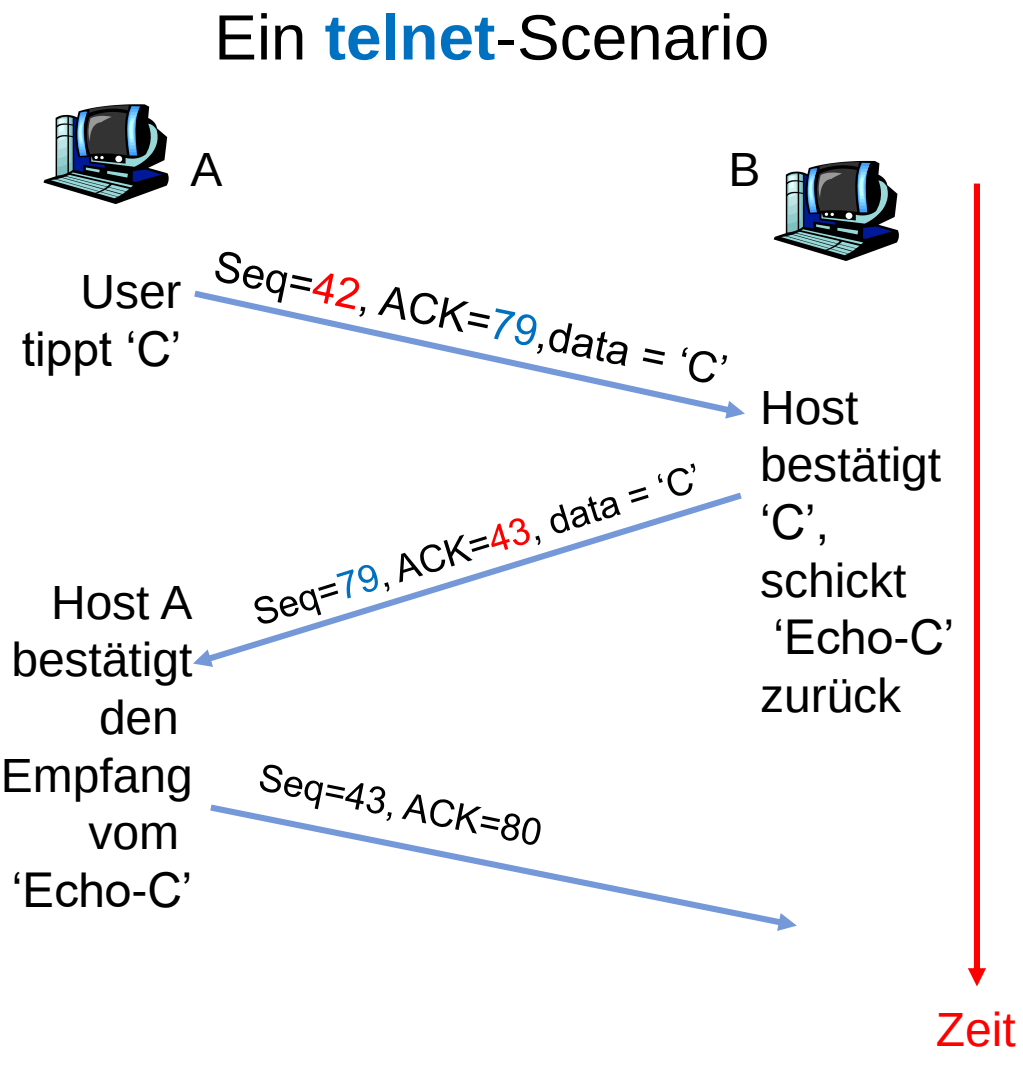
Betriebssysteme und Netzwerke Vorlesung N05

Artur Andrzejak

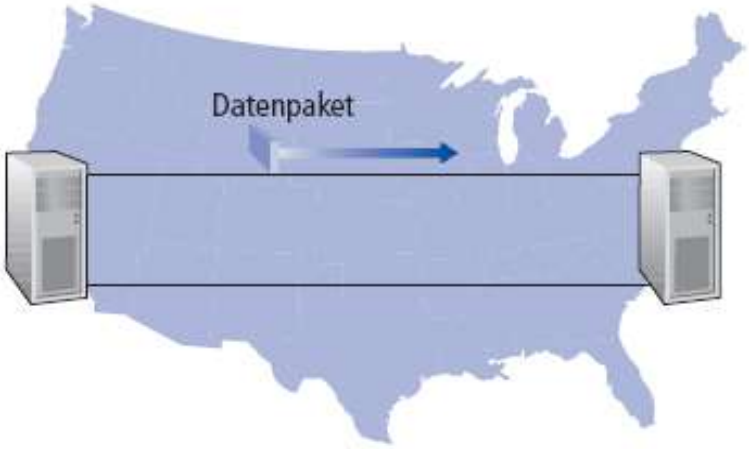
Wiederholung: TCP-Sequenz- und ACK-Nummern

- **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

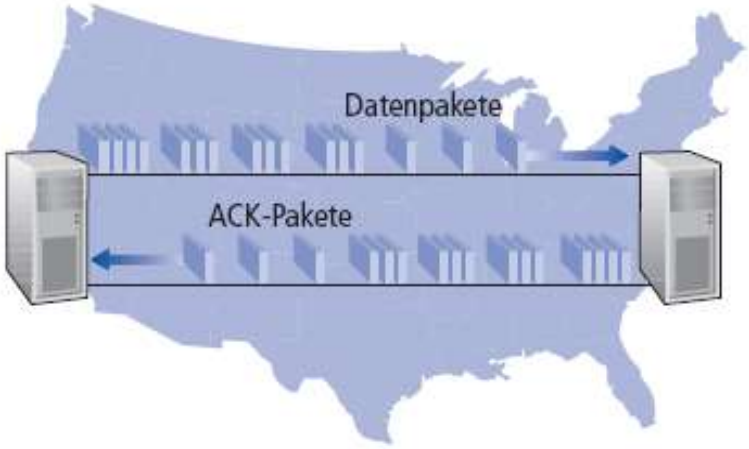


Wiederholung: Verlässliche Zustellung

- Verlässlichkeit: Empfänger bestätigt Pakete
- a. **Stop-and-Wait**: Sender wartet nach jedem Paket auf Bestätigung, erst dann sendet er weiter
- b. **Pipelining:** der Sender schickt mehrere Pakete, ohne auf eine Bestätigung von jedem zu warten
- Die Pakete werden vom Empfänger “gruppenweise”



a Ablauf bei Stop-and-Wait



b Ablauf bei Pipelining

Wiederholung: **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“

Verlässliche Nachrichtenzustellung: Strategien der Bestätigung (Fortsetzung)

Pipelining v2 – **Go-Back-N**

Empfänger

- Sendet beim Empfang eines Paketes nur die ACK für das letzte Paket, das korrekt und in richtiger Reihenfolge empfangen wurde
- Pakete ausserhalb der Reihenfolge (aus der „Zukunft“) werden verworfen

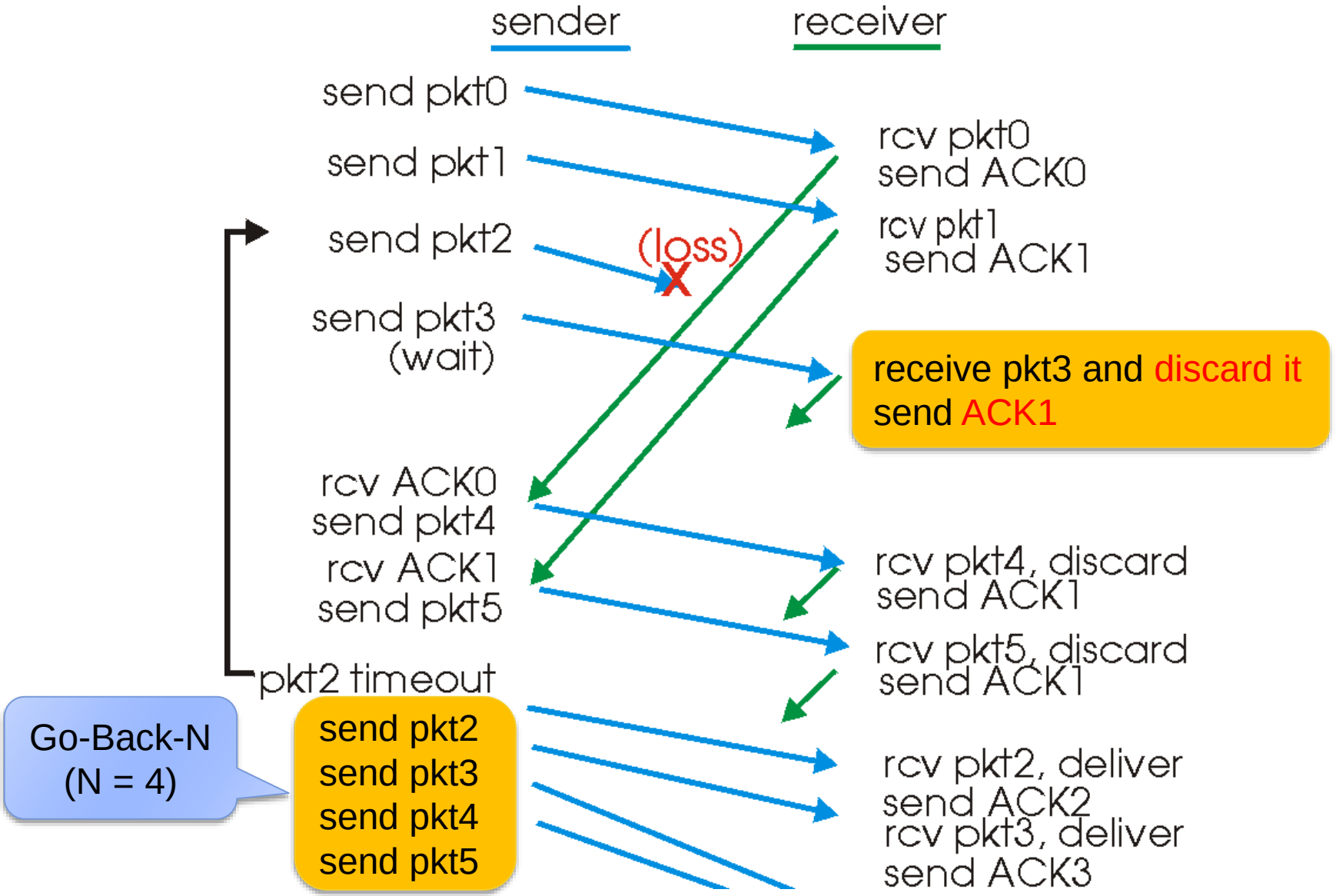
Sender

- Sendet bis zu N Pakete ohne ACKs
- Hat einen Timer für das letzte nicht bestätigte Paket
- Falls der Timer ausläuft: Sender schickt erneut alle noch nicht bestätigten Pakete (bis zu N Stück)
  - Daher “**Go-back-N**”

Es gilt die Konvention der **kumulativen Bestätigung**:

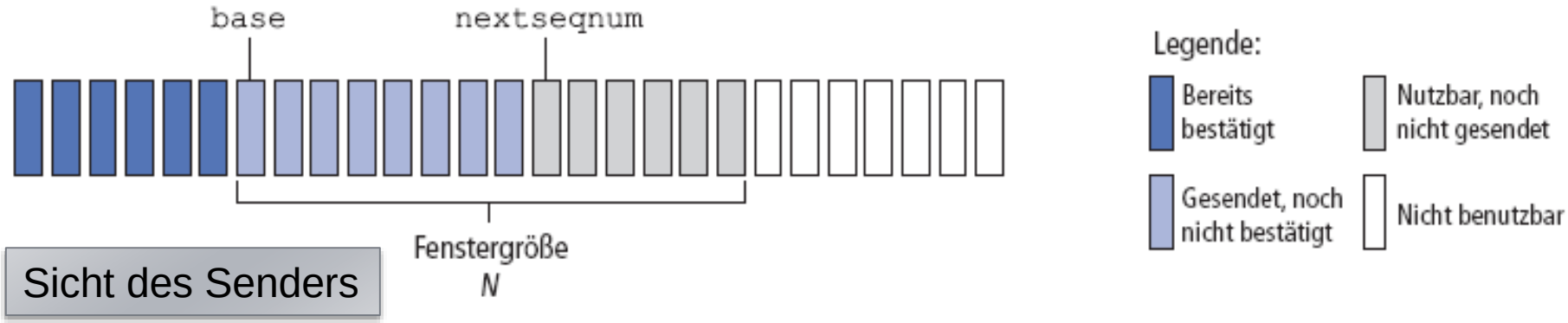
Die letzte ACK bestätigt ALLE vorherigen Segmente

Beispiel Go-Back-N (N=4)



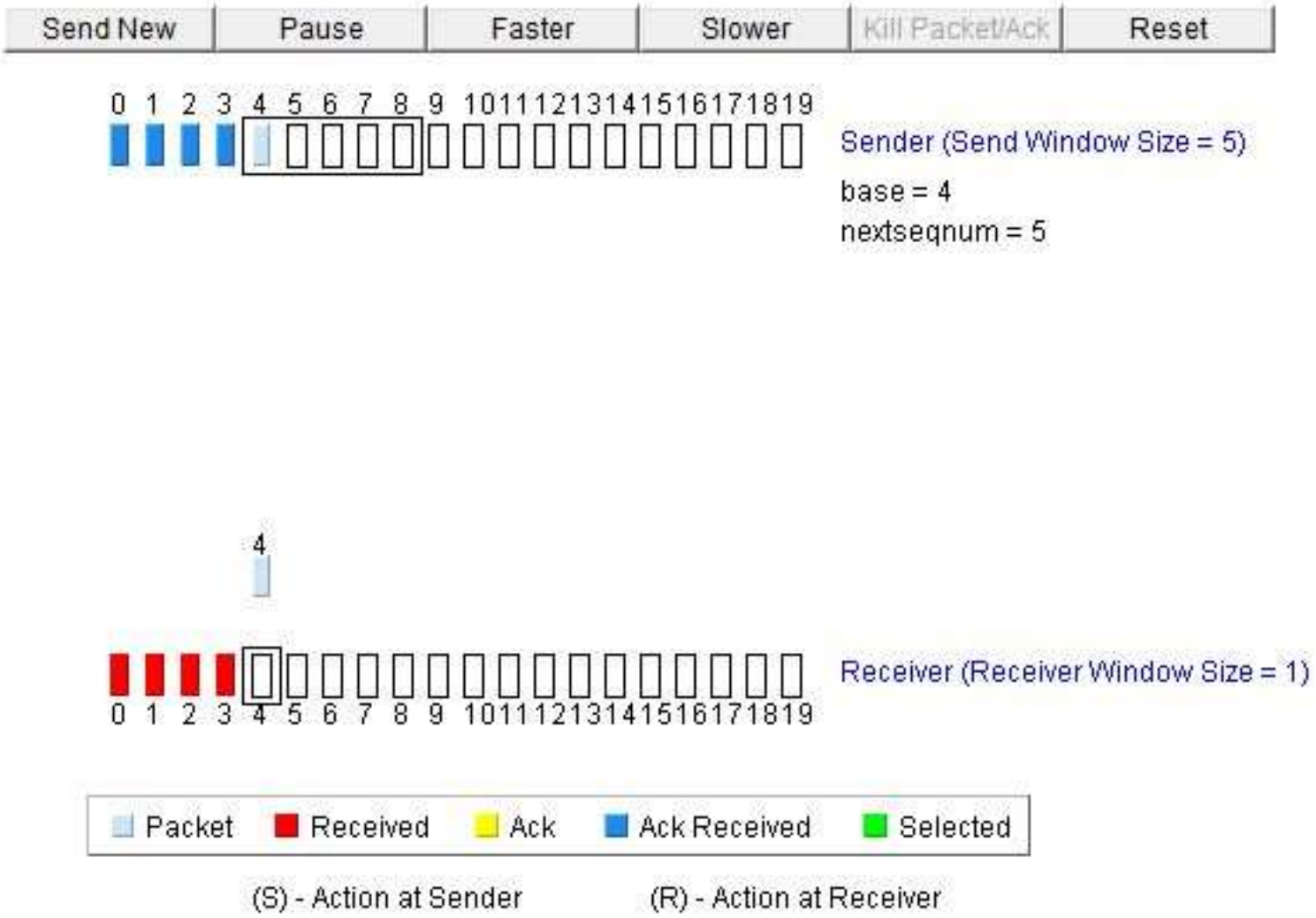
Go-Back-N und Puffer

- Sender: Falls der Timer ausläuft: Sender schickt erneut alle noch nicht bestätigten Pakete (bis zu N)
- Empfänger: Pakete ausserhalb der Reihenfolge (aus der „Zukunft“) werden verworfen
- Wer braucht Paket-Puffer und wie groß?
- Sender: Puffer für N Pakete; Empfänger: kein Puffer für Pakete außerhalb der Reihenfolge



Go-Back-N Demo

- Java Demo: http://goo.gl/rmLSf



Vergleich: Pipelining SR vs. Go-Back-N

- **Selective Repeat** (SR)
  - Effizient, wenn viele Pakete verlorengehen
  - Weniger effizient, wenn die meisten Pakete ankommen
- **Go-Back-N** (GBN)
  - Effizient, wenn die meisten Pakete ankommen
  - Weniger effizient, wenn viele Pakete verlorengehen
  - Etwas einfacher

- Welches Modell benutzt TCP?

- Primär Go-Back-N, aber *TCP verwirft nicht die Pakete aus der Zukunft* (Verhalten von Sel. Repeat)

Verlässliche Nachrichtenzustellung: TCP – Protokoll

TCP Empfänger /1

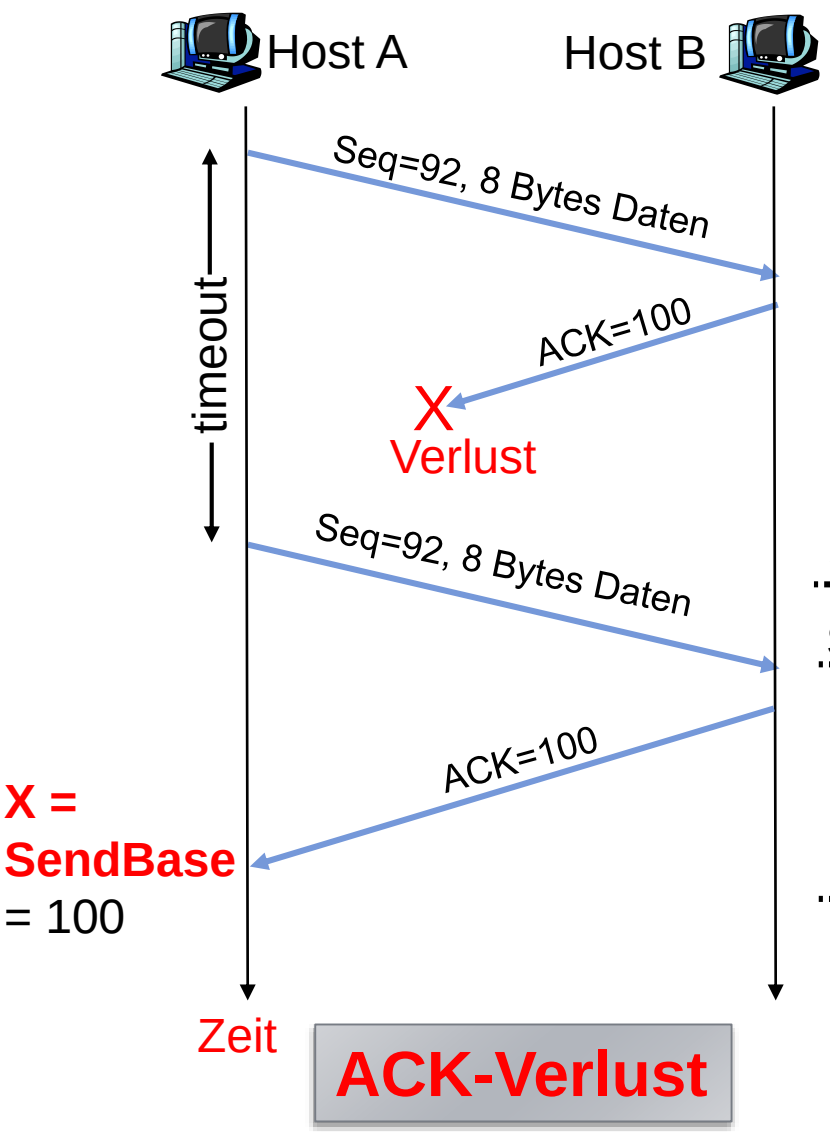
- Die wichtigste Verhaltensregel des Empfängers:
  - **Bei Erhalt eines Paketes sende eine ACK für das letzte „in-Reihenfolge“ erhaltene Paket** (Seq# X)
- Das bedeutet insbesondere:
  - Wenn Segmente außerhalb der Reihenfolge ankommen (d.h. X+2 oder höher), werden sie nicht bestätigt
- **Kumulative Bestätigung**:
  - Ein ACK für Paket X bestätigt auch alle früheren Pakete (X-1, X-2, usw.)

TCP Empfänger /2

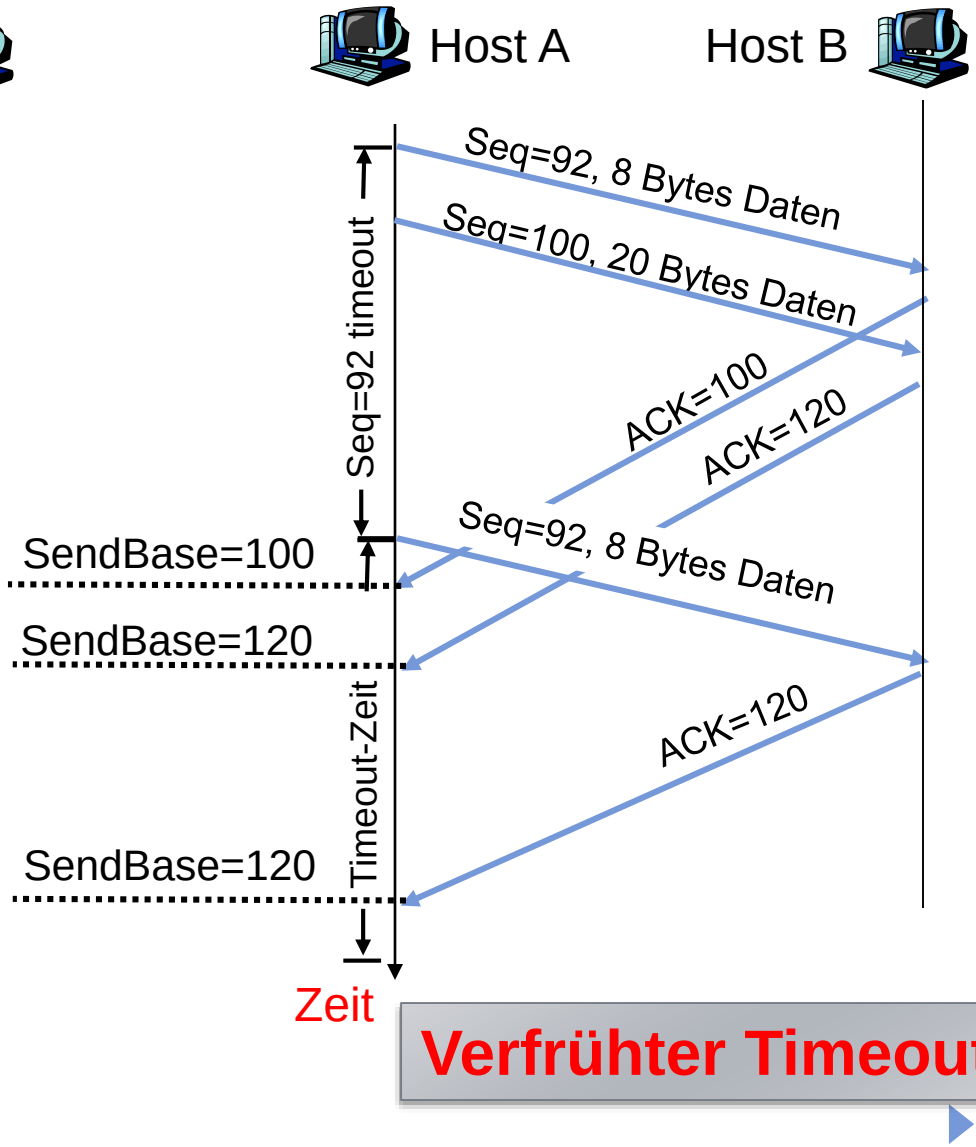
- Die wichtigste Verhaltensregel des Empfängers:
  - **Bei Erhalt eines Paketes sende ein ACK für das letzte „in-Reihenfolge“ erhaltene Paket** (Nummer X)
- Ist das Go-Back-N oder Selective Repeat?

- Im Prinzip ist das ein Go-Back-N-Protokoll
- Allerdings puffern die meisten Implementierungen korrekt empfangene Segmente, auch wenn sie nicht in der Reihenfolge eintreffen (d.h. aus der „Zukunft“)
  - Diese Eigenschaft ist von Selective Repeat und macht einen Puffer auch beim Empfänger nötig

TCP Szenarien: Erneutes Senden

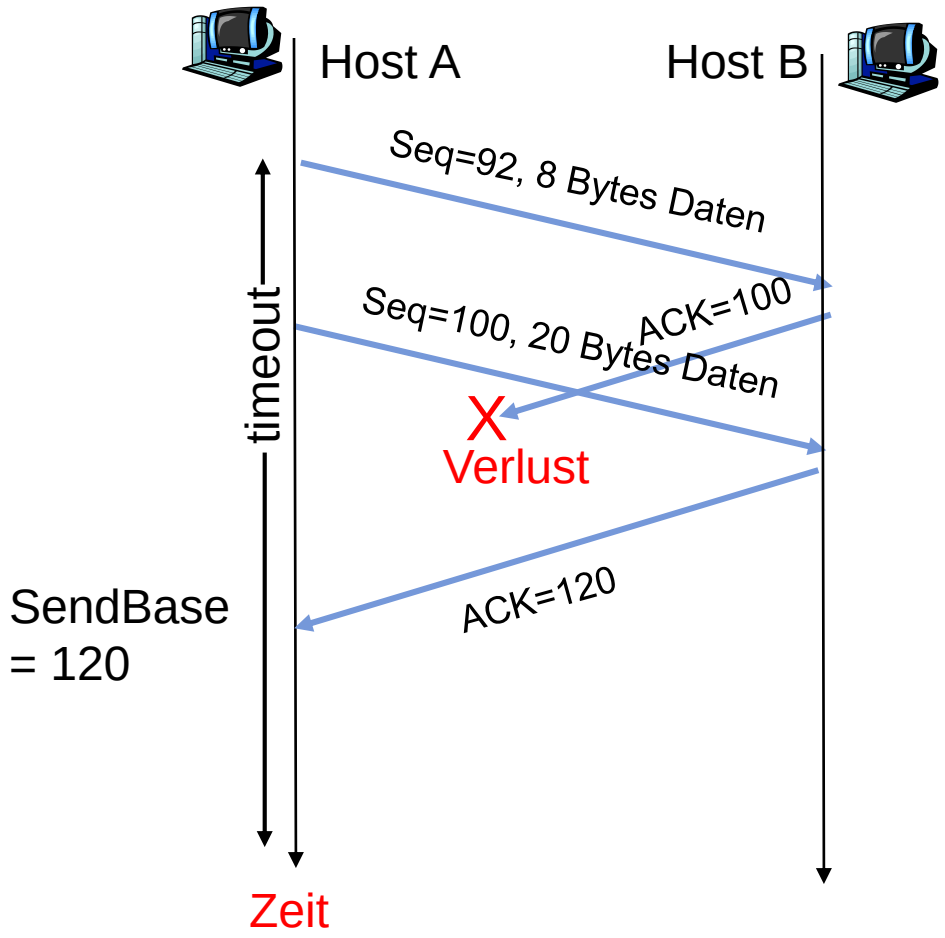


ACK-Verlust



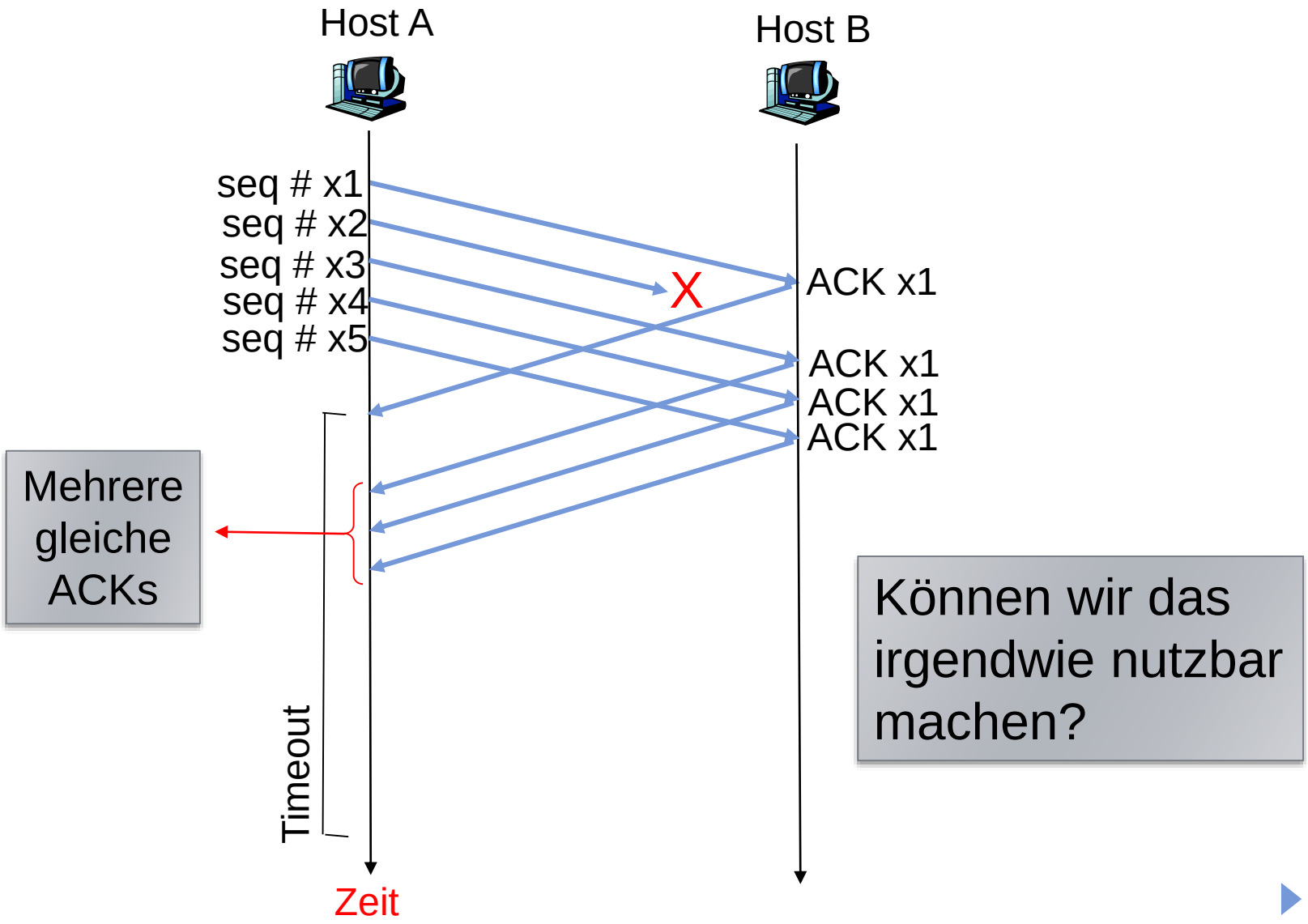
Verfrühter Timeout

TCP Szenarien: Erneutes Senden /2



Kumulatives ACK verhindert die erneute Übertragung des 1. Segmentes (mit Daten 92-99 und ACK 100)

Phänomen: mehrfache ACKs bei Verlust

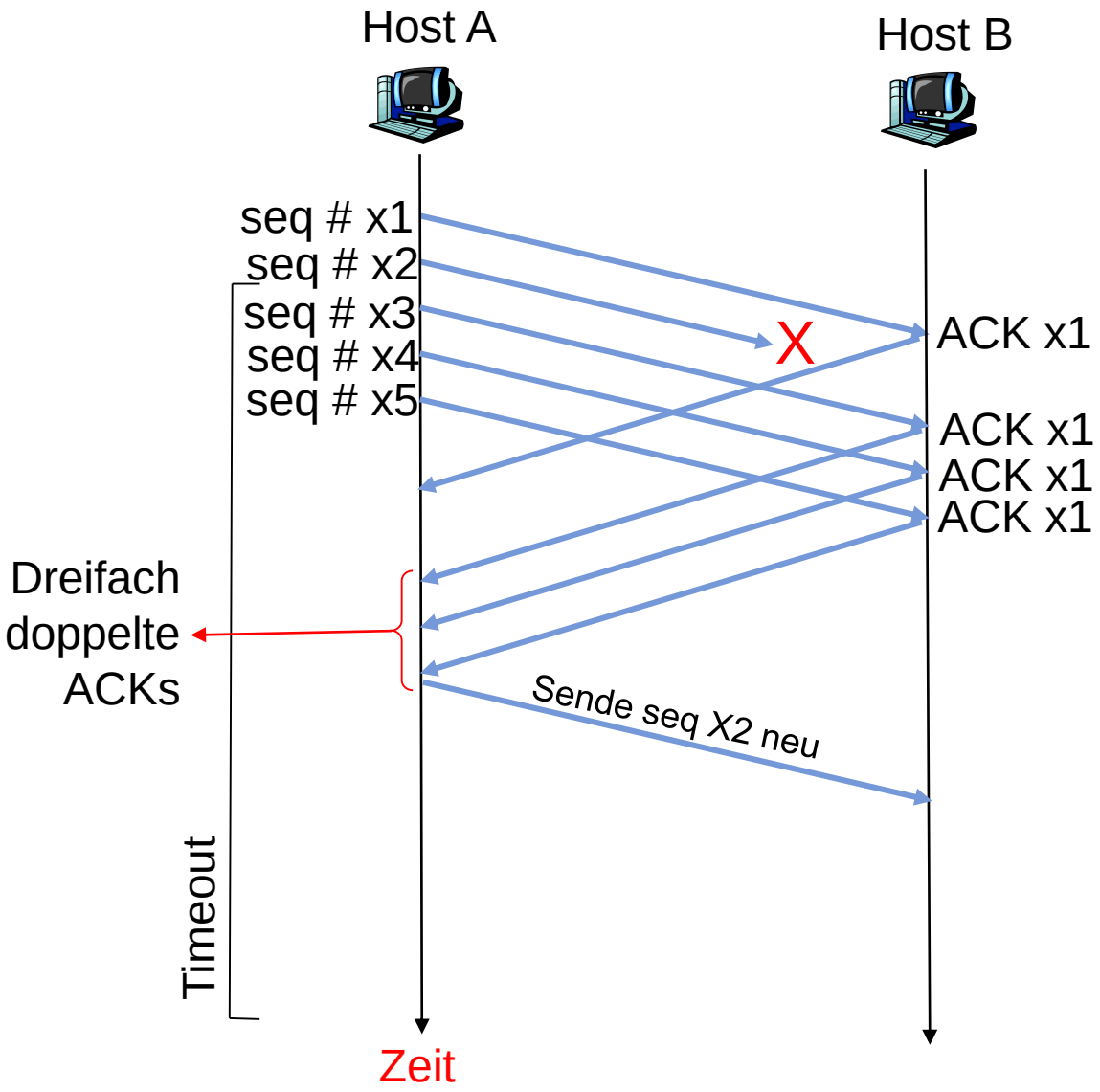


Schnelle Übertragungswiederholung (**Fast Retransmit**)

- Phänomen des Protokolls: Mehrfache ACKs bei Paketverlust
  - Empfänger bestätigt nach jedem Empfang das letzte in richtiger Reihenfolge eingegangene Datenbyte
  - Bei Paketverlust bewirkt das mehrfache, gleiche ACKs
- FR reagiert viel schneller als Timeout des Timers
  - Das verhindert zu große Verzögerung beim Neusenden eines Paketes

- Idee: nutze diesen Effekt, um verlorengegangene Pakete zu identifizieren
- Wenn der Sender 3 gleiche zusätzliche ACKs für gleiches Paket (seq# k) erhält, nimmt er an, dass das Segment nach den seq# der ACKs verlorenging
- Das ist **fast retransmit**: schicke das entsprechende Segment erneut, noch bevor der Timer abläuft

Schnelle Übertragungswiederholung /2



TCP- Ereignisse beim Sender (vereinfacht)

Bei **neuen Daten:**

- Erzeuge Segment mit Sequenznummer q
  - q ist die Datenstromnr. des ersten Datenbytes im Segment
- Starte den Timer, falls dieser noch nicht läuft
  - Es gibt nur einen Timer: für das älteste noch nicht bestätigte Segment (Timer-Ablaufzeit: TimeOutInterval)

Bei **Timeout:**

- Sende erneut ab dem „ältesten“ nicht bestätigten Segment
- Starte den Timer neu

Bei **ACK empfangen:**

- Galt ACK für noch unbestätigte Segmente?
  - Merke, welche Segmente damit bestätigt wurden
  - Starte den Timer neu, falls es noch unbestätigte Segmente gibt
- Ggf. sog. Fast Retransmit
  - Sende erneut ab dem „ältesten“ nicht bestätigten Segment

TCP-ACK-Erzeugung [RFC 1122, RFC 2581]

Event at Receiver	TCP Receiver action
Arrival of in-order segment with expected seq #. All data up to expected seq # already ACKed	Delayed ACK. Wait up to 500ms for next segment. If no next segment, send ACK
Arrival of in-order segment with expected seq #. One other segment has ACK pending	Immediately send single cumulative ACK, ACKing both in-order segments
Arrival of out-of-order segment higher-than-expect seq. # . Gap detected	Immediately send *duplicate ACK*, indicating seq. # of next expected byte
Arrival of segment that partially or completely fills gap	Immediate send ACK, provided that segment starts at lower end of gap

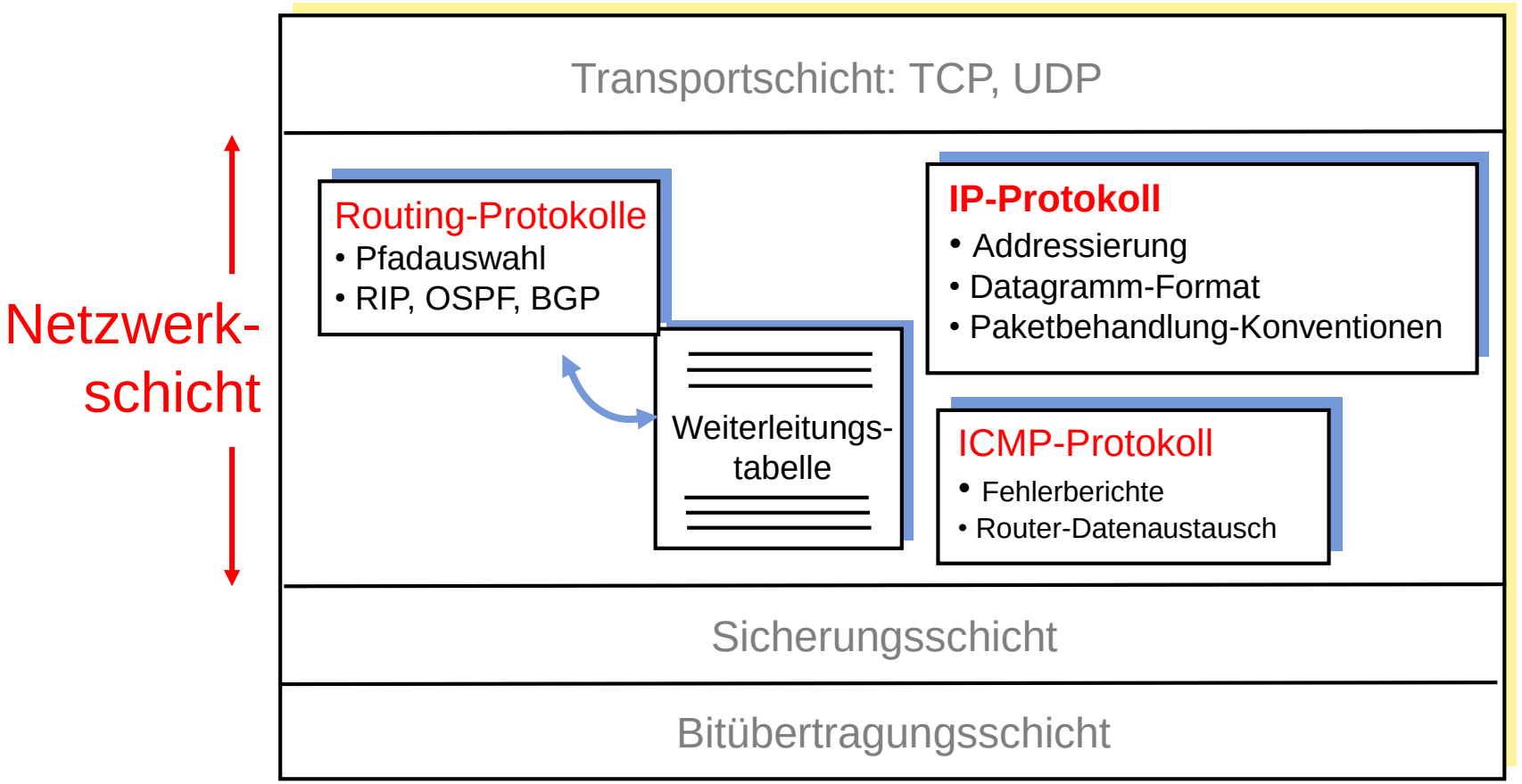
TCP: Zuverlässiger Datentransfer

- TCP stellt einen zuverlässigen Datentransfer über den unzuverlässigen Datentransfer von IP zur Verfügung
- Pipelining von Segmenten
- Kumulative ACKs
- TCP verwendet einen einzigen Timer für Übertragungswieder-holungen
- Übertragungswieder-holungen werden ausgelöst durch:
  - Timeout
  - Doppelte ACKs
- Video: 12 TCP – Reliable …
  - https://www.youtube.com/watch?v=6S2jOnMm5I0

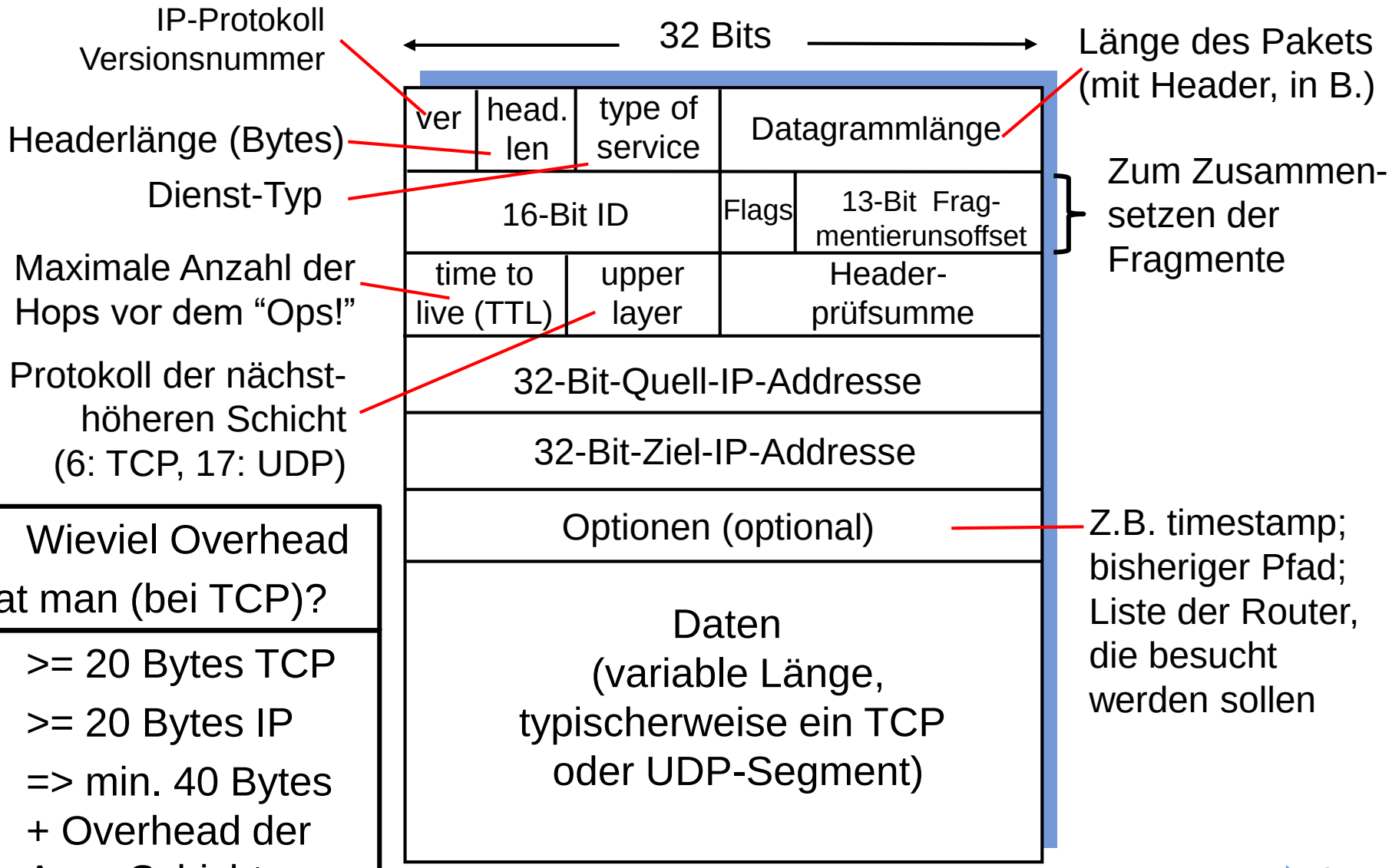
Netzwerkschicht:
Das Internetprotokoll (IP) – Grundlagen

Netzwerkschicht des Internet-Stacks

Drei zentrale Elemente der Netzwerkschicht



Format des IP-Datagramms

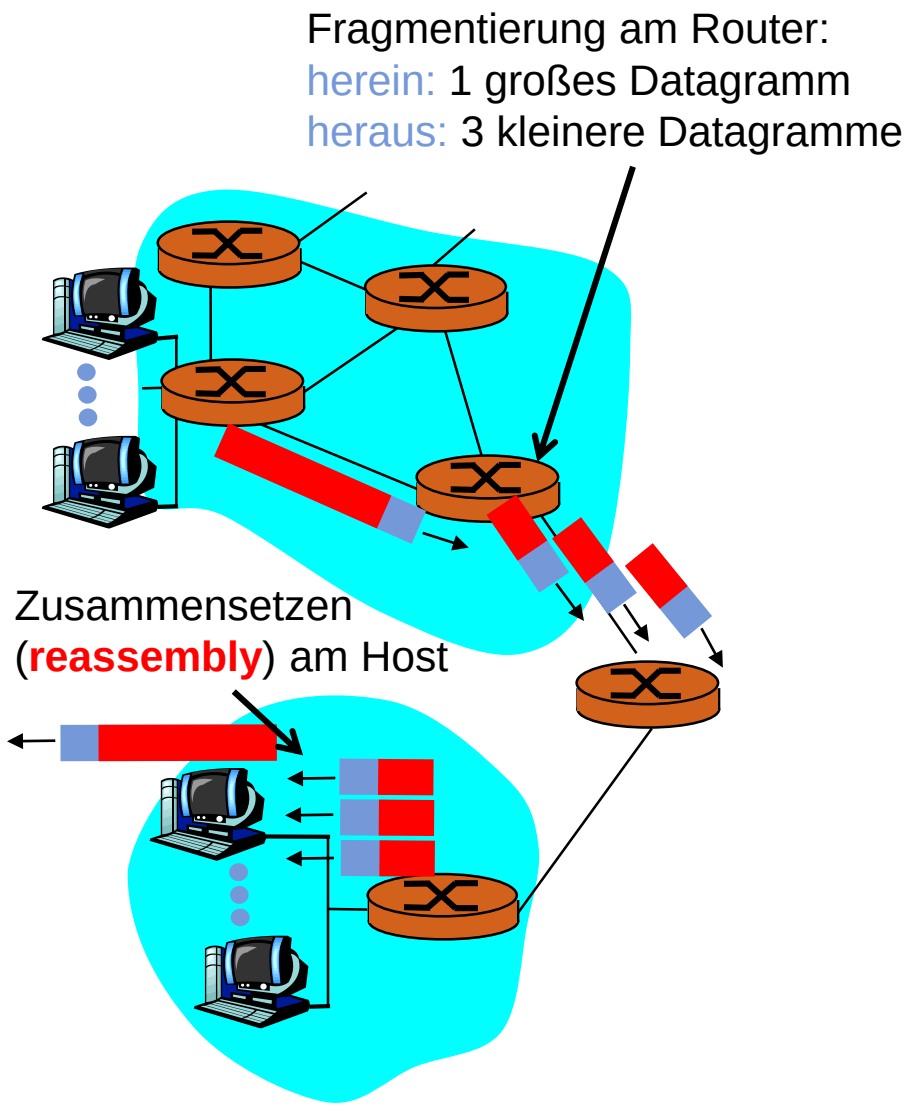


Wieviel Overhead hat man (bei TCP)?

- >= 20 Bytes TCP
- >= 20 Bytes IP
- => min. 40 Bytes + Overhead der Anw.-Schicht

Fragmentierung und Zusammensetzen

- Netzwerkleitungen erlauben je nach Typ die max. Datenmenge, die ein Rahmen (frame) der Sicherungsschicht tragen kann
  - Diese nennt man **MTU**, **maximum transmission unit**
- Große IP-Datagram werden somit auf manchen Leitungen in mehrere Rahmen aufgeteilt (**fragmentiert**)
  - Sie werden erst beim Ziel (Host) zusammengesetzt – warum?
  - IP-Header enthält die Informationen, welche Fragmente zusammengehören, und wie das Datagramm aufgeteilt wurde



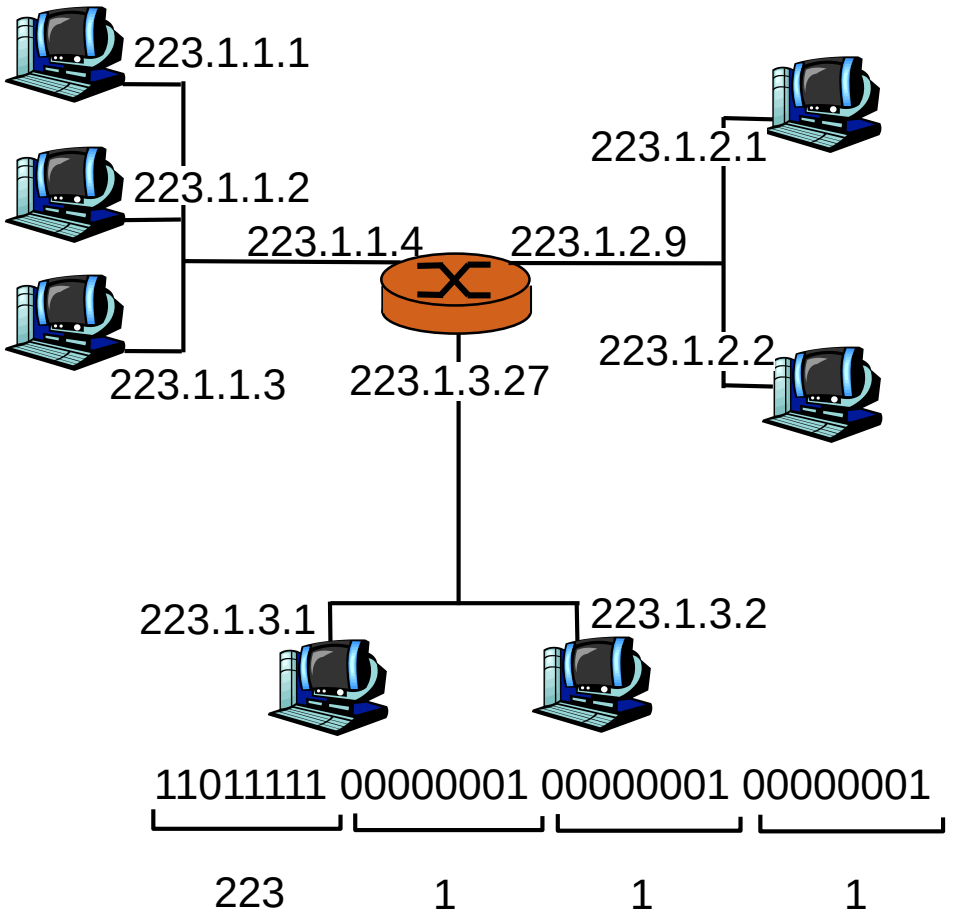
Fragmentierung von Datagrammen

- Drei Felder im Header
  - **Identifikation** (16 Bits) x: Alle IP-Pakete mit dem gleichem Wert von x sind Fragmente eines „großen“ Paketes
  - **Flags** (3 Bits): Bit 0: reserviert; Bit 1: darf (0) bzw. darf nicht (1) zerlegt werden; Bit 2 („more fragments“): letztes Fragment (0) bzw. weitere Fragmente folgen (1)
  - **Fragment Offset** (13 Bits): Besagt, ab welcher Position im großen Paket das Fragment anfängt, in „8 Bytes“-Einheit
- Z.B. Datagramm mit ca. 3 kBytes wird an einem Router mit MTU = 1200 Bytes zerlegt

Fragment#	Länge	ID	Flags-Bit 2	Offset
1	960 Bytes	z.B. 999	1	0
2	**960 Bytes**	**z.B. 999**	**1**	**120 (da 120*8 = 960)**
3	1020 Bytes	z.B. 999	0	240 (da 240*8 = 2*960)

Interfaces und IP-Addressen

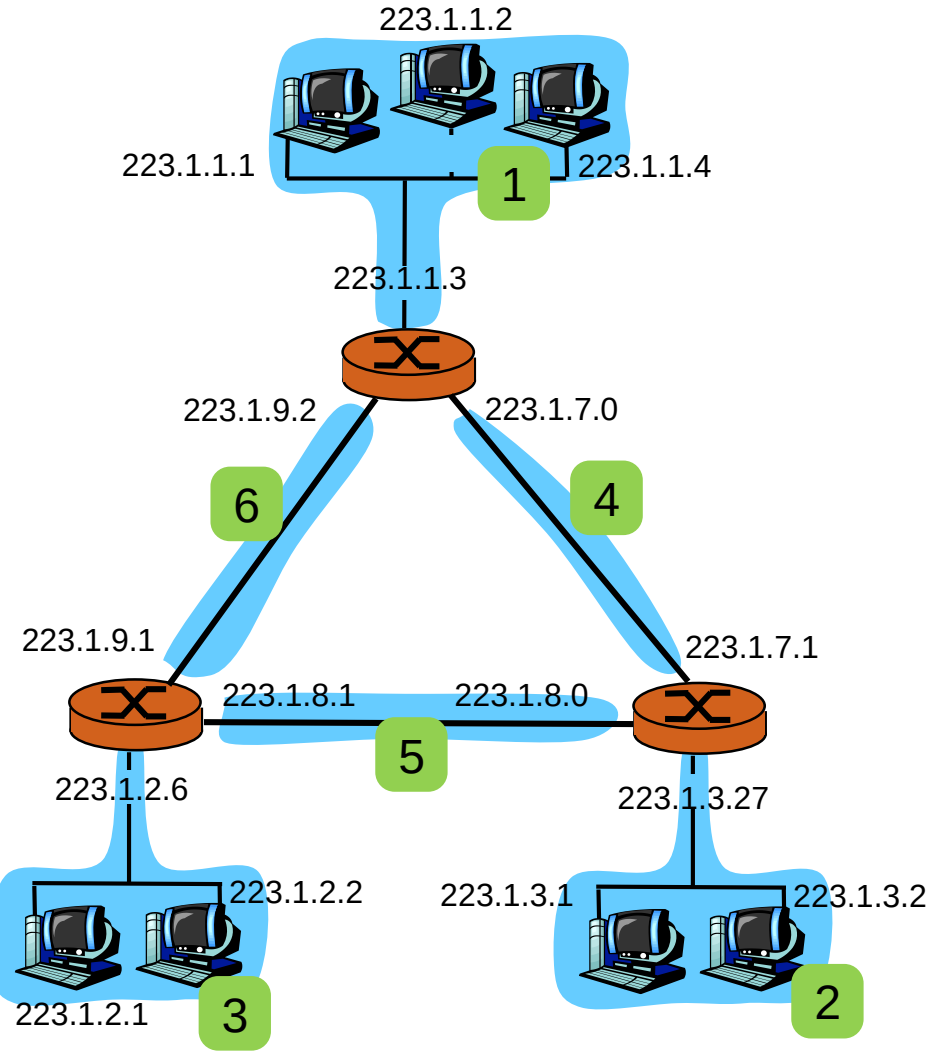
- **Interface** (**Schnittstelle**): Verbindung zwischen Host / Router und einer physischen Leitung
  - Router haben i.A. mehrere Interfaces
  - Hosts haben i.A. nur ein Interface: Netzwerkkarte
- **IP-Adresse**: eine 32-bit Identifikation für das Interface eines Hosts oder Routers
- Jedes Interface hat eine oder mehrere sog. IP-Adressen



Schreibweise als 4 durch „.“ getrennte Bytes, hier: 223.1.1.1

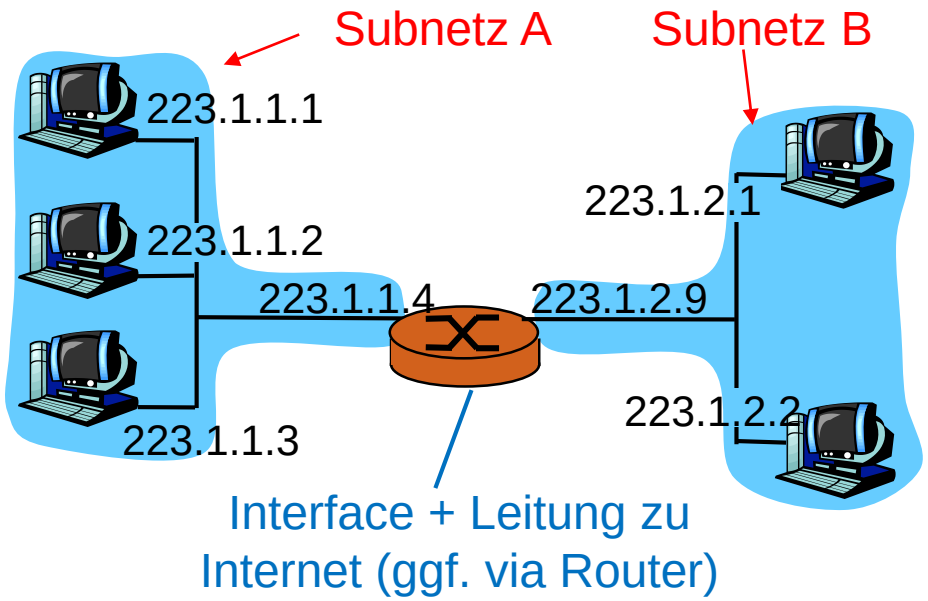
(Sub)**Netzwerk** – „Hardware“-Definition (Kurose)

- „Netzwerk“: Eine zusammenhängende Netzkomponente an einem einzigen Routerausgang
- *Um die (Sub)Netzwerke zu bestimmen, trennen Sie jede Schnittstelle von ihrem Router. Dadurch entstehen „Inseln“ (aus verbundenen Leitungen + Interfaces), in denen sich separate Netzwerke (…) befinden. Jedes dieser einzelnen Netzwerke wird als (**Sub)Netzwerk** bezeichnet.*
- Hier: wie viele (Sub)Netzwerke?



Adressen in (Sub)Netzwerken (= Subnetzen)

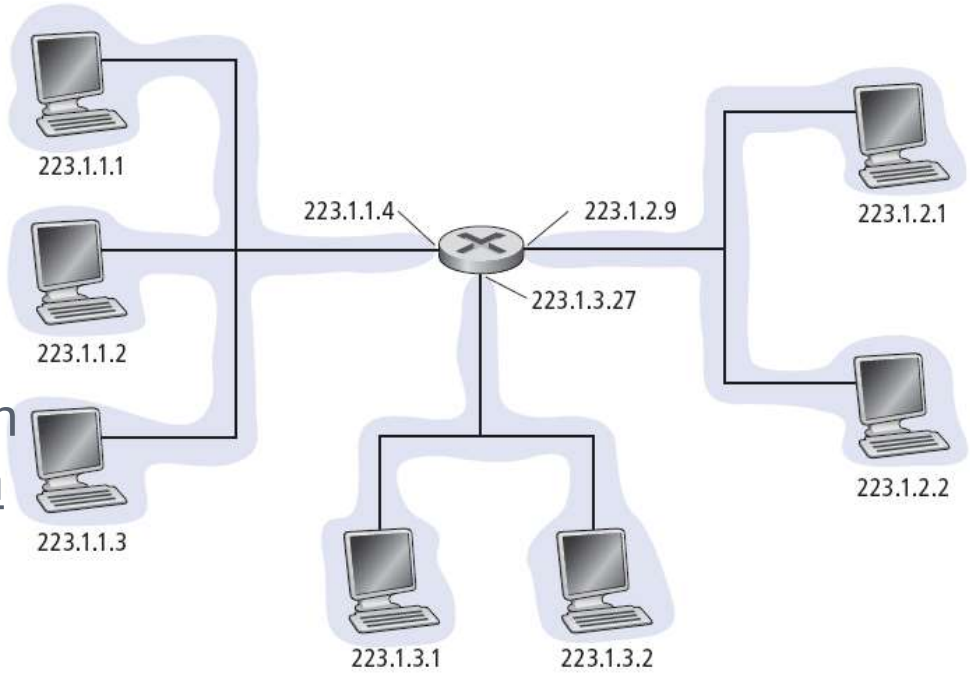
- In einem Subnetz haben i.A. alle Interfaces die IP-Adressen aus einem zusammenhängenden Bereich
  - Auch die IP-Adresse des Router-Interfaces ist dabei
- Was wären die minimalen Adressenbereiche in Subnetzen A? Und in B?
  - Aus diversen Gründen ist die 0-te und die letzte Adresse im Subnetz ggf. nicht verwendbar
    - Bei B: 223.1.2.0 und 223.1.2.15



- Für Subnetz A:
  - 223.1.1.0 bis 223.1.1.7 (7 wegen Adresse mit *.4)
- Für Subnetz B:
  - 223.1.2.0 bis 223.1.2.15 (15 wegen Adresse mit *.9)

IP-Adressierung – Netzwerke

- IP-Adressen haben zwei Bestandteile:
  - **Netid: Netzwerkteil**: Die oberen Bits der Adresse, identifizieren ein Netzwerk
  - **Hostid: - Hostteil**: Die unteren Bits der Adresse, identifizieren ein Interface innerhalb des Netzwerks
- Wenn der Adressenbereich eines Netzwerks 2^k Adressen umfasst, dann
  - Hostid hat die unteren k Bits
  - Netid hat die oberen 32-k Bits



- Z.B. das untere Netzwerk hat 256 IP-Adressen:
  - 8 (untere) Bits als Hostteil
  - 24 obere Bits (223.1.3.*) als Netzwerkteil

Wozu überhaupt Netid und Hostid?

- Da alle Interfaces in einer Firma / Uni / Organization X (i.A.) im gleichen Netzwerk sind, haben sie gleiche Netid (d.h. gleiche obere Bits in Adressen)
  - Die Router außerhalb von X müssen nur die Netid (von X) betrachten, um die Pakete korrekt weiterzuleiten
  - Ähnlich wie Vorwahlnummern bei Telefonen
- Das hat einige Vorteile – welche?

1. Die Daten in den Routertabellen sind kleiner, da es viel weniger Netids als IP-Adressen gibt
2. Jede Organization kann intern beliebig die Hostids zu Interfaces zuordnen und braucht das nach „draußen“ nicht mitzuteilen

Zusammenfassung

– Transportschicht (Ende)–

- Verlässliche Zustellung - Go-Back-N
- TCP – Protokoll: Verlässliche Nachrichtenzustellung (effizient)
- Quellen:
  - Kurose / Ross Kapitel 3
  - Wikipedia

– Netzwerkschicht –

- Das Internetprotokoll (IP) – Grundlagen, Adressierung
- Quellen:
  - Kurose / Ross Kapitel 4, Wikipedia

Danke.

Zusätzliche Folien

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

TCP Sender (vereinfacht)

```
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

loop (forever) {
  switch(event)

  event: Neue Daten aus Anw.-Schicht
      Erzeuge Segment mit Seq# NextSeqNum
      if (timer läuft z.Z. nicht)
          Starte timer
      reiche Segment an die IP-Schicht weiter
      NextSeqNum = NextSeqNum + length(data)

  event: timer Timeout
      sende erneut das noch-nicht-bestätigte Segment
                  mit der kleinsten Seq#
      Starte timer

  event: ACK empfangen, Wert des ACK-Felds ist y
      if (y > SendBase) {
          SendBase = y
          if (es gibt noch nicht bestätigte Segmente)
                  Starte timer
          }
} /* Ende Endlosschleife*/
```

Bemerkung:

- SendBase-1: zuletzt kumulativ bestätigtes Byte
- Beispiel:
  - SendBase-1 = 71; y= 73, also Empfänger will Bytes 73+ als nächstes haben
  - y > SendBase => wir können das „linke Ende“ des Sende-fensters nach rechts schieben

Fast Retransmit Algorithmus:

```
event: ACK empfangen mit Wert y des ACK-Feldes
        if (y > SendBase) {
            SendBase = y
            if (es gibt noch nicht bestätigte Segmente)
                Starte timer
        }
        else {
            erhöhe die Anzahl der empfangenen ACKs für y
            if (die Anzahl der empfangenen ACKs für y == 3) {
                Sende erneut Segment mit Sequenznummer y
            }
```

wir haben ein mehrfaches ACK für das Segment y

fast retransmit passiert hier

Intervall-Länge des Timers

- Bei neuen Daten oder Empfangen eines ACKs
  - Schätze die Intervall-Länge anhand der RTT, ähnlich wie beim Exponential Moving Average-Ansatz
- Bei Paketverlust:
  - Verdopple den Wert der Intervall-Länge
  - Warum?