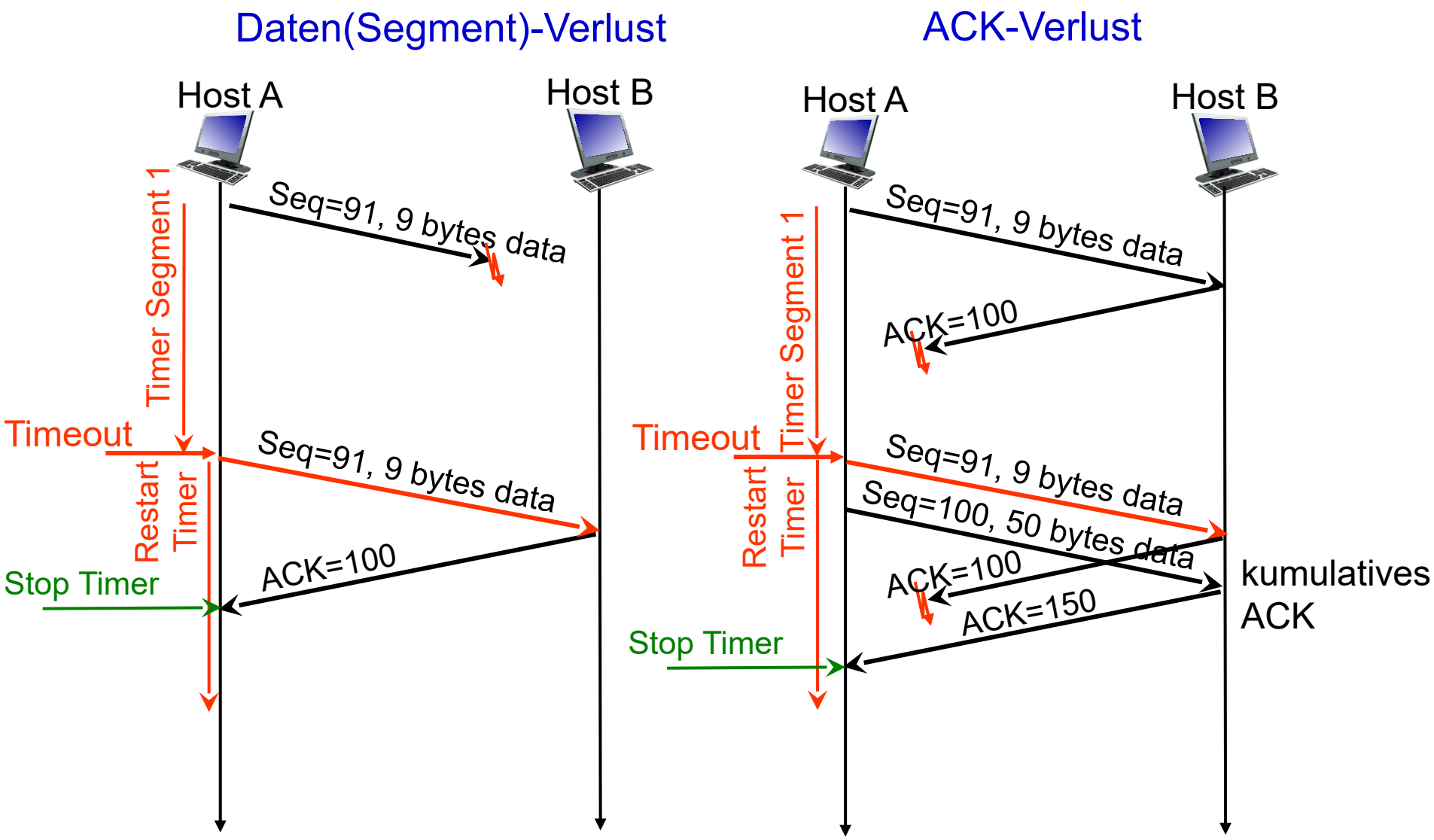
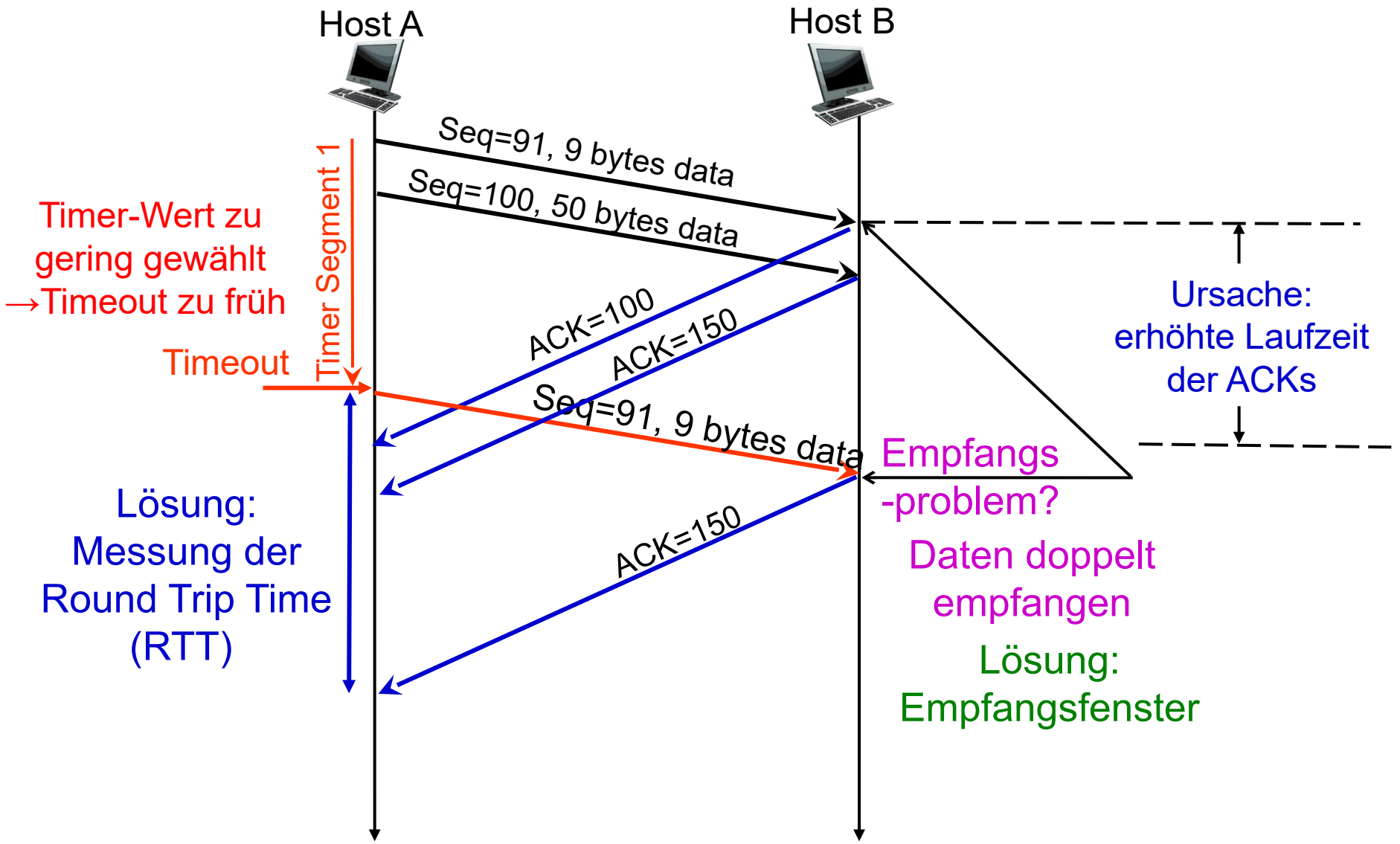
Einführung und Übersicht

Beispiel: Übertragungswiederholung



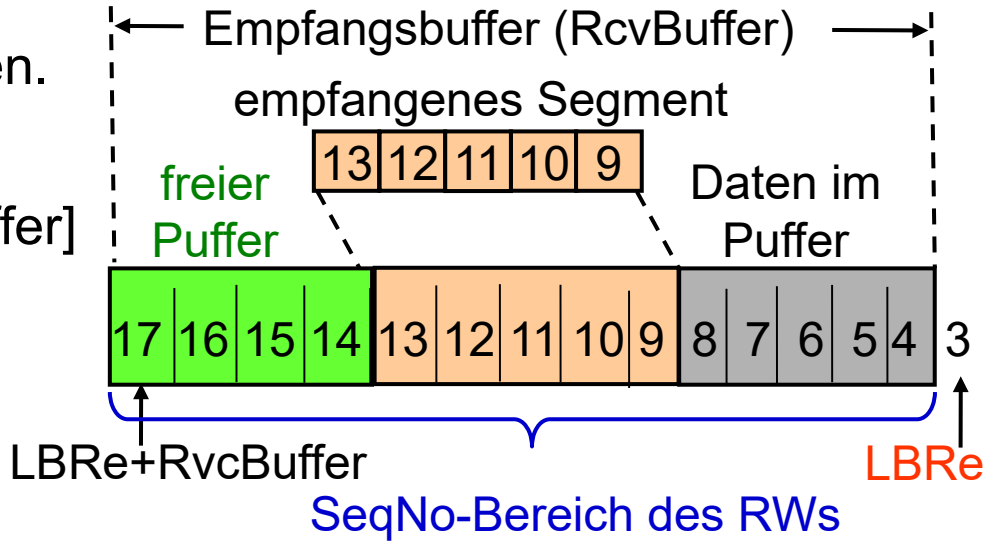
Beispiel: Vorzeitiges Timeout



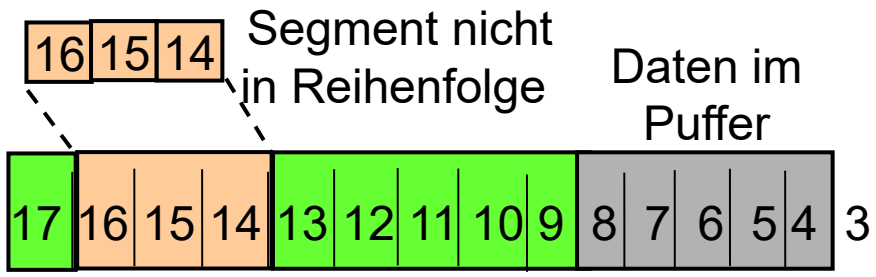
TCP Empfänger - Receive Window

Empfänger akzeptiert nur Segmente deren SeqNo im Bereich des Empfangsfensters (RW) liegen und noch nicht empfangen wurden.

- SeqNo∈[LBRe+1, LBRe+RvcBuffer]
  - LBRe (LastByteRead): Letzte SeqNo, die von der Anwendung aus dem RcvBuffer gelesen wurde



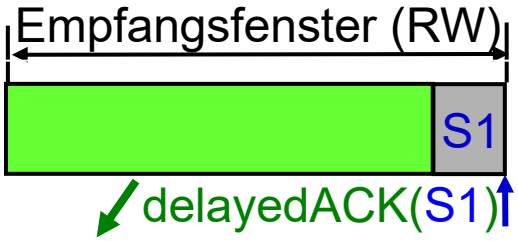
- Ankunft eines Segmentes nicht in Reihenfolge, aber mit SeqNo im Empfangsfenster
  - Daten im RcvBuffer speichern
  - Daten im RcvBuffer weisen eine Lücke auf

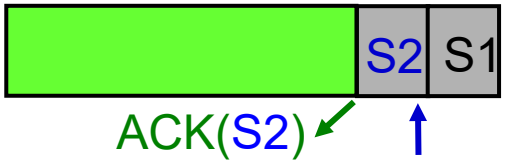


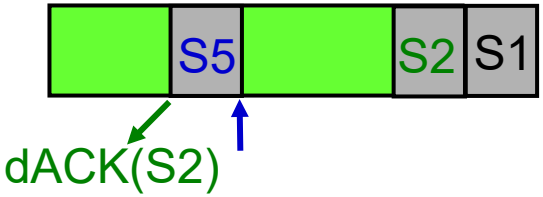
Acknowledgement im Empfänger

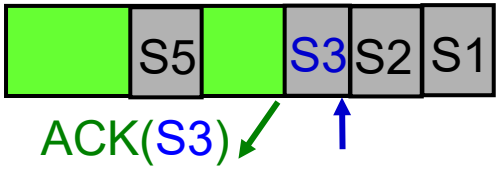
Alle Daten im Empfangsfenster (RW) quittiert

- Segment S1 mit SeqNo in RW empfangen
  - Delayed ACK: typisch bis zu 200 ms auf eigene Daten warten, danach ACK senden
- Ankunft eines Segmentes (S2) in Reihenfolge
  - sofort ein kumulatives ACK für (S2) senden
  - beide Segmente (S1 & S2) werden bestätigt
- Ankunft Segmentes (S5) nicht in Reihenfolge
  - Duplicate ACK (dACK):
  - das letzte in Reihenfolge richtig empfangene Segment wird nochmals quittiert
- Ankunft eines Segmentes (S3), das die Lücke teilweise oder vollständig schließt
  - kumulatives ACK für S3
- Bei duplicate/kumulativem ACK kein Delayed Ack









Acknowledgement im Empfänger

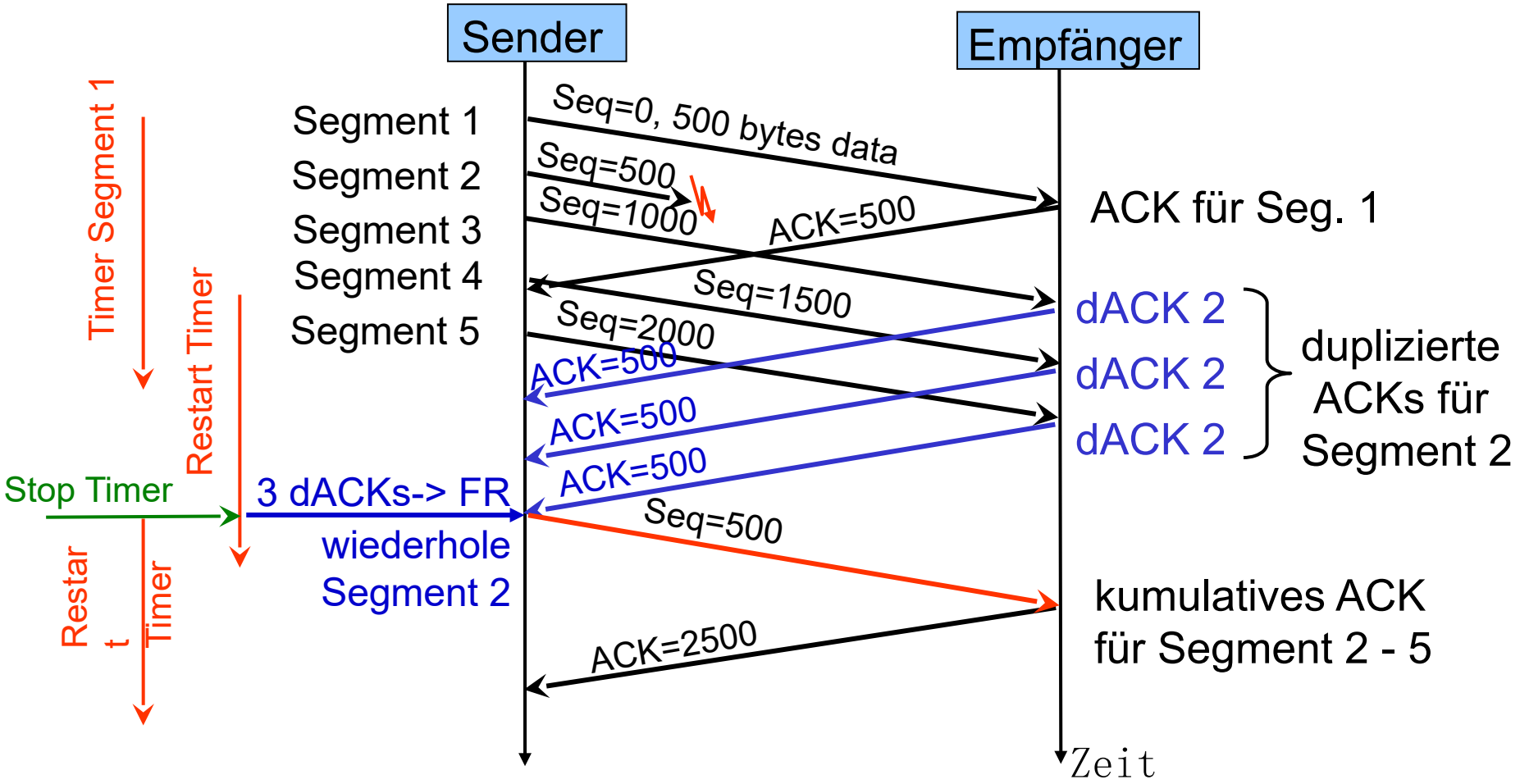
- Time-Out-Dauer oft relativ lang:
  - Lange Verzögerung bevor verlorene Pakete erneut gesendet werden

- Verlorene Segmente im Sender durch duplicate ACKs (dACKs) erkennen
  - Der Sender sendet oft mehrere Segmente hintereinander
  - Wenn ein Segment verloren geht, wird der Sender wahrscheinlich viele duplicate ACKs empfangen

- Fast-Retransmit
  - Selektive Übertragungswiederholung im Sender
  - Empfängt der Sender vor dem Ablauf des Retransmission Timers 3 dACK‘s kann er von einem Segmentverlust ausgehen
  - Das noch nicht bestätigte Segment mit „kleinster“ Sequenznummer wird vom Sender wiederholt

Acknowledgement im Empfänger

- Nur ein Timer im TCP Sender
- Sender kann 5 Segmente senden, je Segment MSS = 500 Bytes Daten



Three-Way-Handshake

Active Open: Client sendet SYN-Segment an Server

- SYN-Flag auf 1 gesetzt ist
- Die Initial Sequence Number (Client_ISN) wird zufällig gewählt
- keine Nutzdaten

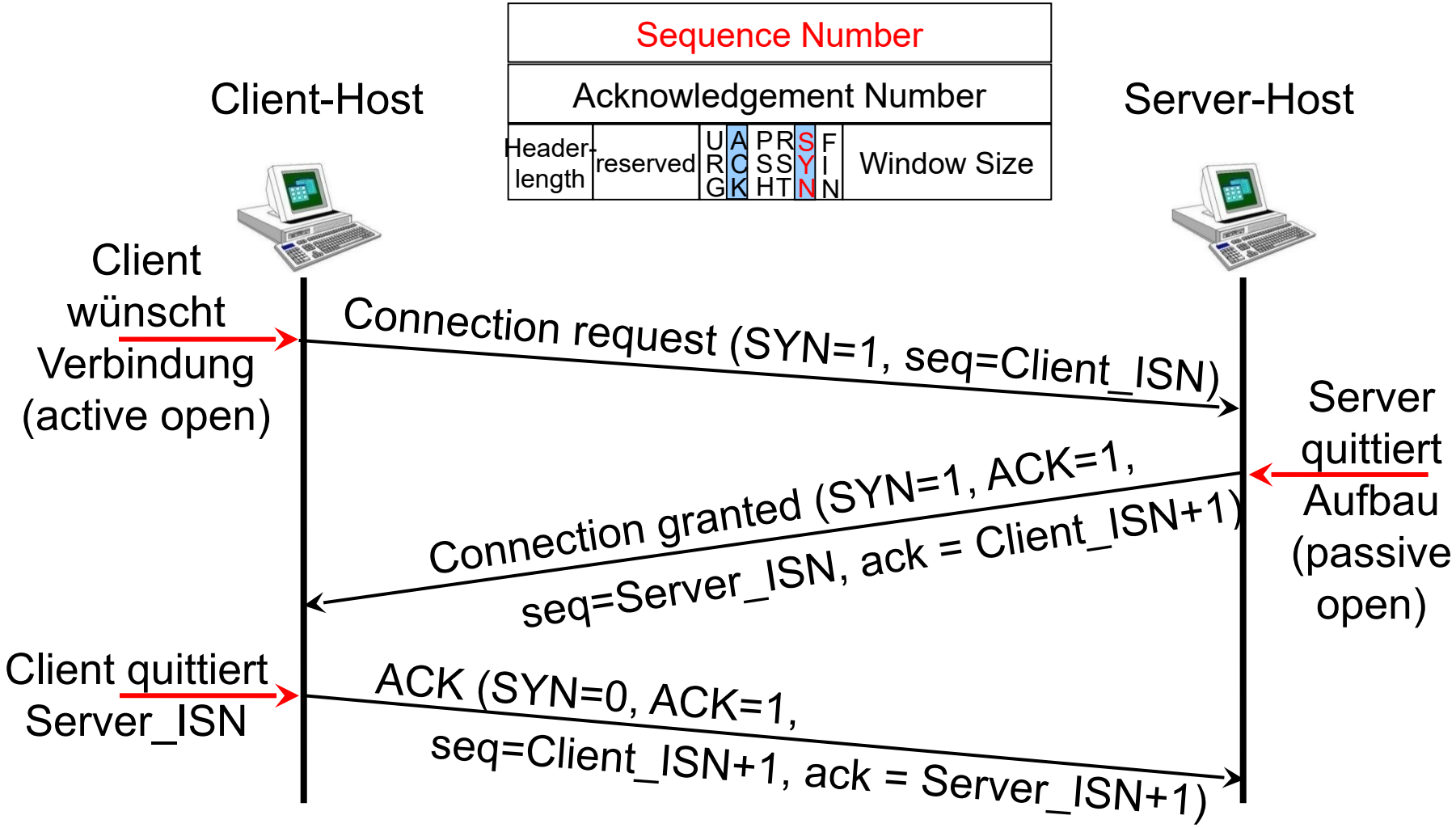
Passive Open: Server sendet SYN_ACK-Segment

- SYN- und ACK- Flag wird auf 1 gesetzt sind
- Acknowledgement Number wird auf Client_ISN+1 gesetzt
- Server wählt zufällige Initial Sequence Number (Server_ISN) aus

Client sendet ACK-Segment

- Client quittiert Server_ISN, wobei das ACK-Flag auf 1 gesetzt ist
- Sequenznummer wird auf Client_ISN+1 gesetzt, aber nicht verbraucht
- die Acknowledgement Number wird auf Server_ISN+1 gesetzt

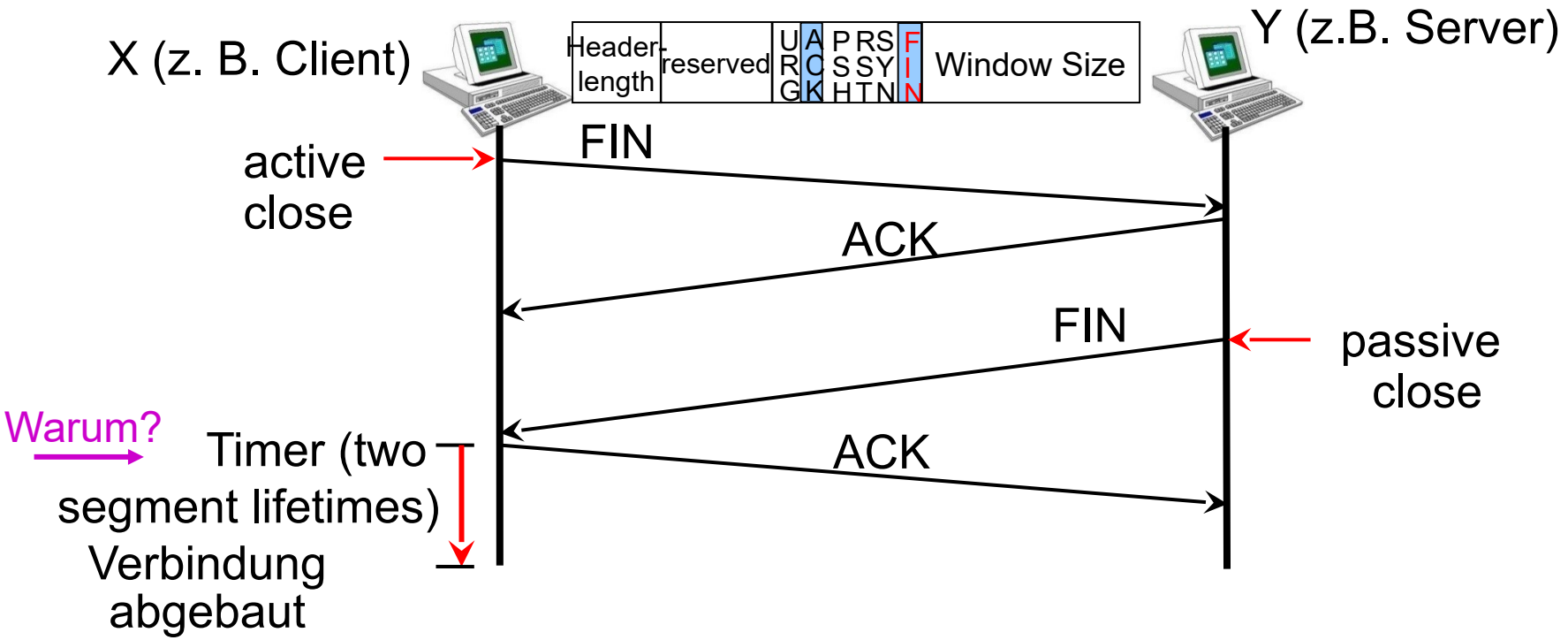
Beispiel: Three-Way-Handshake



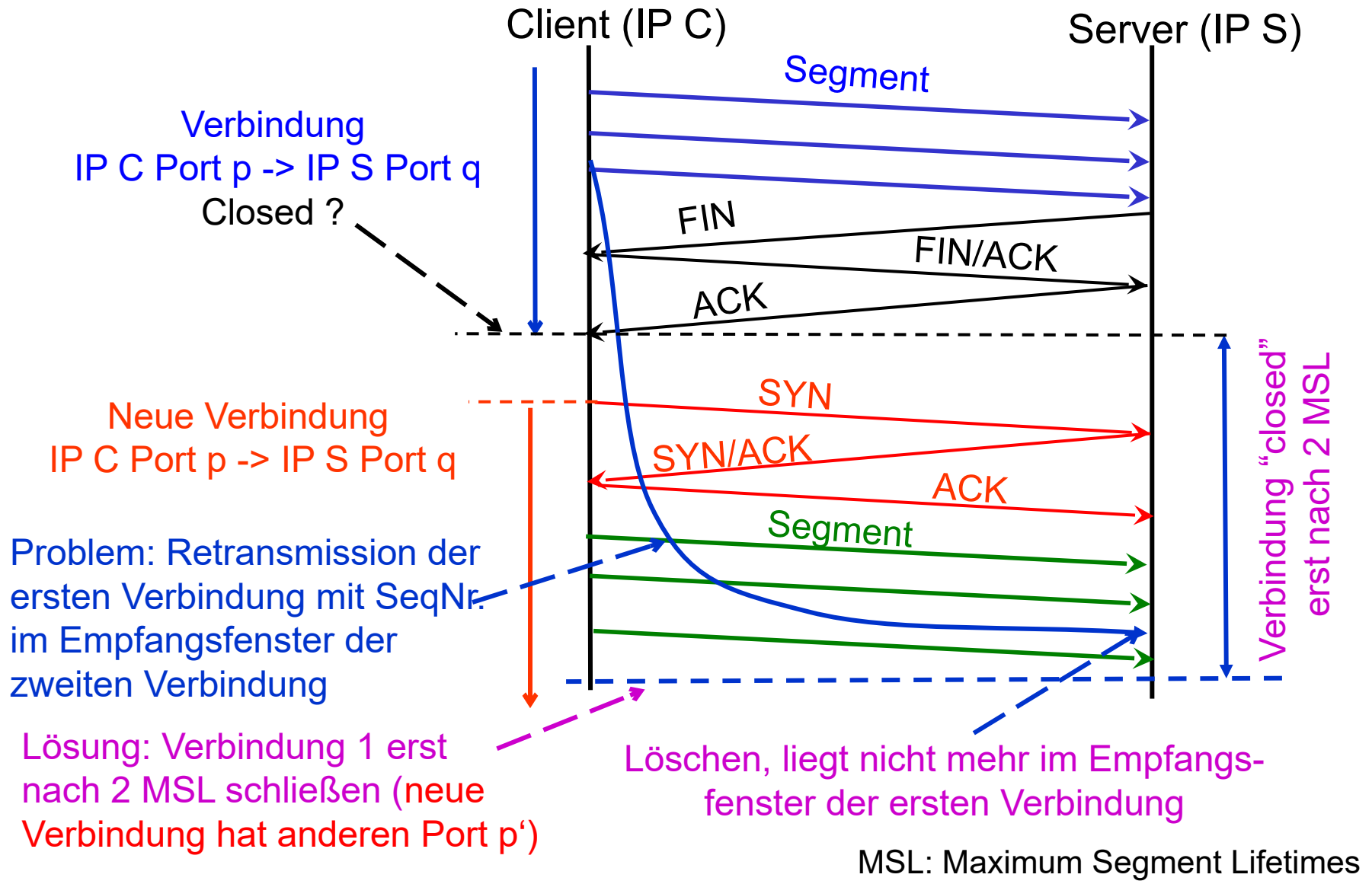
ISN: Initial Sequence Number

TCP-Verbindungsabbau

- Server- und Client bauen unabhängig voneinander die Verbindung ab
- **Active Close**: X sendet FIN-Segment (FIN-Flag gesetzt) an Y
  - Y sendet ACK-Segment als Bestätigung an X
- **Passive Close**: Y sendet FIN-Segment (FIN-Flag gesetzt) an X
  - X sendet ACK-Segment als Bestätigung an Y



Problem: Verbindungen und Sequenznummern



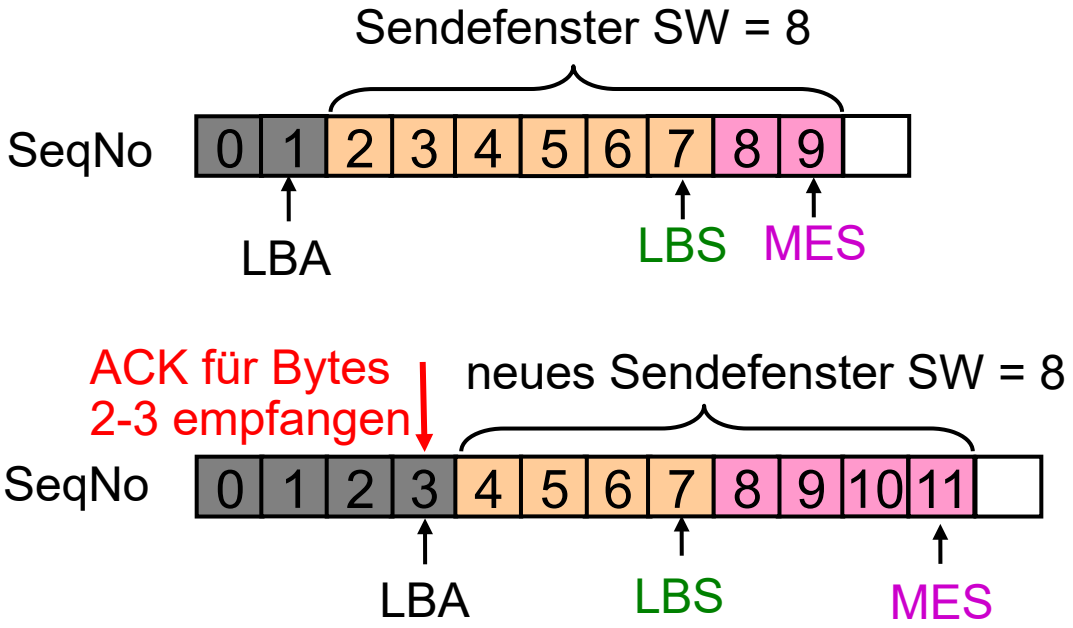
MSL: Maximum Segment Lifetimes

Flow Control (Advertised Window)

Prinzip des Sliding Windows

Betriebsweise im Sender

- Sendefenster (SW): Sender kann Bytes nur innerhalb des Sendefensters mit den Sendefolgenummern **SeqNo ∈ [LBA+1, LBA+SW]** senden
- Sliding Window: Sender verschiebt sein Sendefenster bei Empfang eines neuen ACKs um die Zahl der quittierten Bytes
- Beispiel: je Segment MSS = 1Byte, kumulatives ACK für die Bytes 2 – 3



gesendet und bestätigt
LBA (LastByteAcked): als letzte quittierte SeqNo

gesendet und nicht bestätigt
LBS (LastByteSent): letzte gesendete SeqNo

verwendbar, noch nicht gesendet . MES: Maximal erlaubte Sendenummer

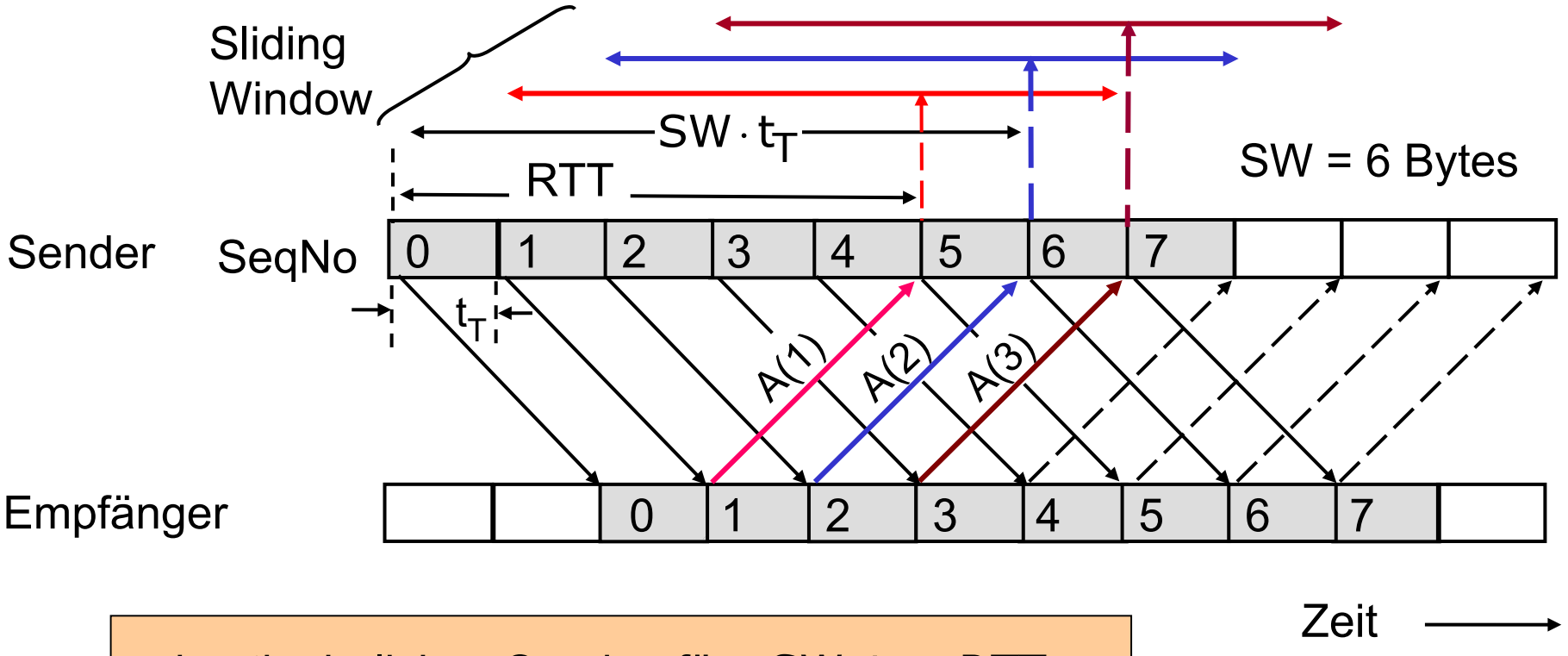
nicht verwendbar

Prinzip des Sliding Windows

Betriebsweise im Empfänger

- Quittierung (ACK) erfolgt sofort für jedes Segment

Beispiel: MSS = 1Byte; Sendefenster SW = 6 Bytes; A(i): TCP AckNo



kontinuierliches Senden für: $SW \cdot t_T \geq RTT$

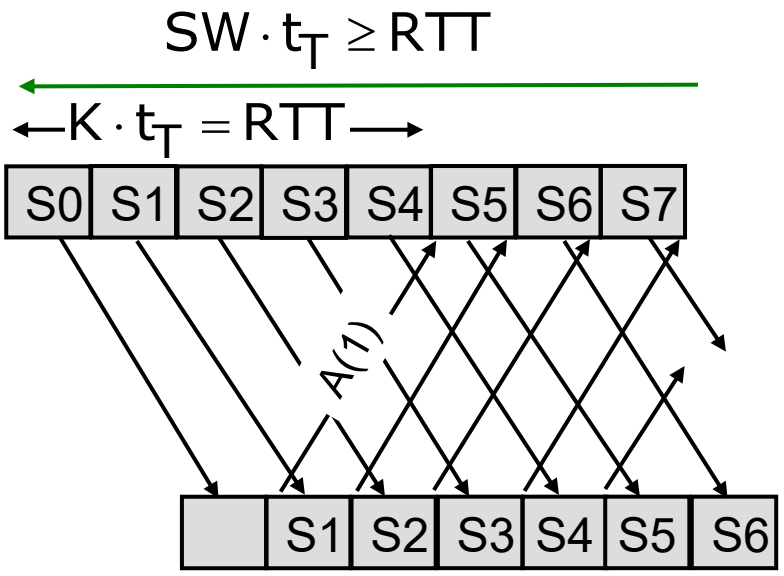
Anmerkung: Empfangsfenster von TCP detaillierter auf Slide 7

Durchsatz bei fehlerfreier Übertragung

Sender kann kontinuierlich senden falls:

$$SW \cdot t_T \geq RTT \quad \Rightarrow \quad SW \geq \frac{RTT}{t_T}$$

Für $SW \cdot t_T = K \cdot t_T = RTT$ ist der Durchsatz bereits maximal



- Zeitdauer bis zum ACK-Empfang wird durch andere Segmente ausgefüllt

Erfolgreicher Durchsatz

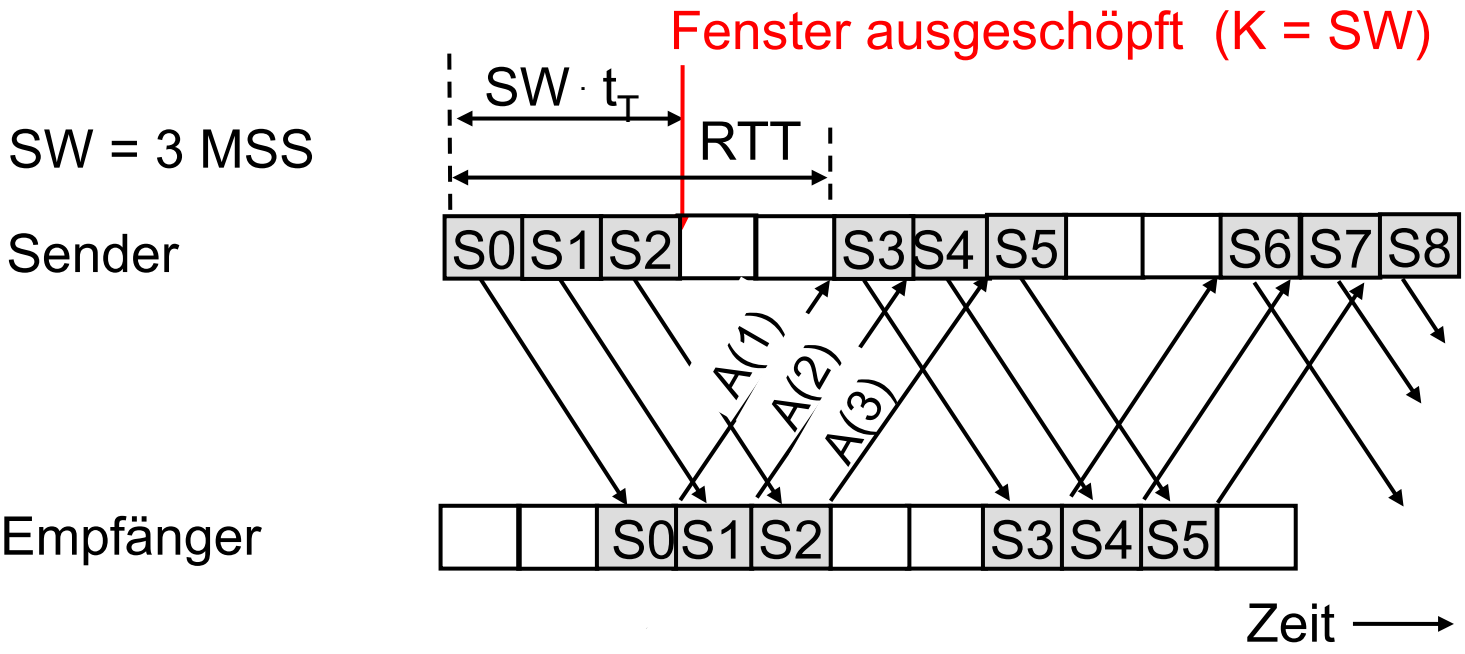
$$\gamma_E = \frac{\text{Anzahl der Bits, welche der Empfänger akzeptiert}}{\text{Zeitdauer bis diese Bits am Sender quittiert sind}}$$

- Während der RTT werden **K Segmente** (K · MSS Bytes) gesendet

$$\gamma_E = \frac{K \cdot MSS}{RTT} = \frac{MSS}{t_T} = v_B$$

Durchsatz bei fehlerfreier Übertragung (2)

Kein kontinuierliches Senden im Fall: $SW \cdot t_T < RTT$



- Durchsatz ohne Fehler für $SW \cdot t_T < RTT$:
  - Während der RTT werden SW Segmente gesendet

$$\gamma_E = \frac{SW \cdot MSS}{RTT} = \frac{MSS}{t_T} \frac{SW \cdot t_T}{RTT} = v_B \frac{SW \cdot t_T}{RTT}$$

Bandwidth Delay Product

- Bandbreite-Verzögerungsprodukt
  - Sendedauer eines Segments mit MSS Bytes: t_T
  - Effektive Übertragungsrate für ein Segment mit MSS Bytes:

$$v_B = \frac{MSS}{t_T}$$

  - kontinuierliches Senden wenn Buffer gefüllt ist bis das ACK für das erste Segment eintrifft, also für $SW \cdot t_T \geq RTT$

$$SW \cdot t_T \cdot \frac{MSS}{t_T} \geq RTT \cdot v_B \Rightarrow SW \cdot MSS = SW\_b \geq RTT \cdot v_B$$

Sendefenster in Bit

Bandbreite x Verzögerungsprodukt

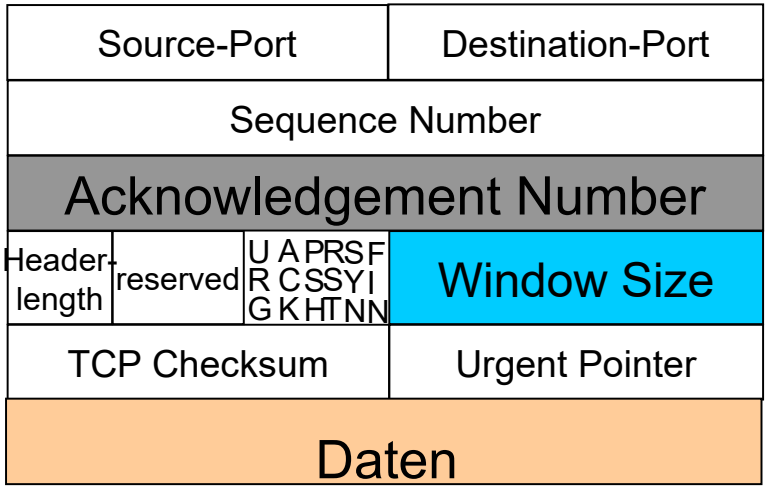
Window-based Flow Control

- TCP - Flusskontrolle arbeitet nach dem Prinzip des Sliding-Window
- Kreditbasierter Mechanismus: Empfänger teilt Sender den Sendekredit mit

Window Size (Fenstergröße)

- wird als Advertised Window bezeichnet
- wird in Bytes angegeben
- vom Empfänger an den Sender gesendeter Sendekredit

Empfänger → Sender

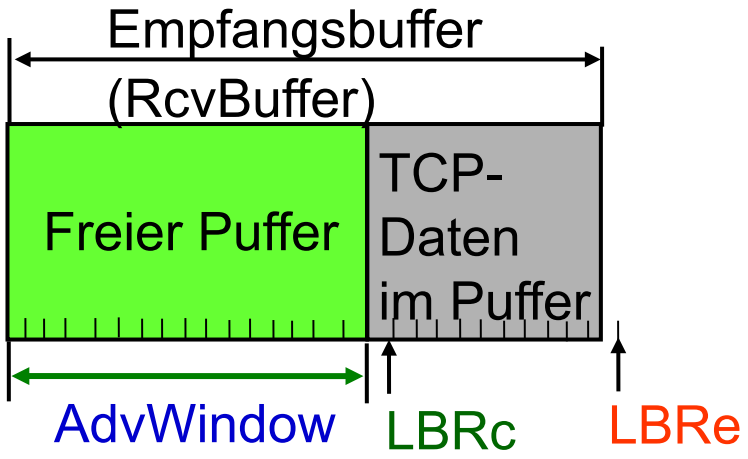


Acknowledgement

- Der Empfänger kann dem Sender unabhängig von der Flusskontrolle ACKs senden (Erweiterung des Sliding-Window Verfahrens)
  → Vorteil: Ein ACK erhöht nicht automatisch die Fenstergröße

AdvertisedWindow (AdvWindow, ReceiveWindow, Empfangsfenster)

- Freier Pufferplatz (in Bytes) für diese Verbindung im Empfänger



LBRe (LastByteRead): Letzte SeqNo, die von der Anwendung aus dem RcvBuffer gelesen wurde

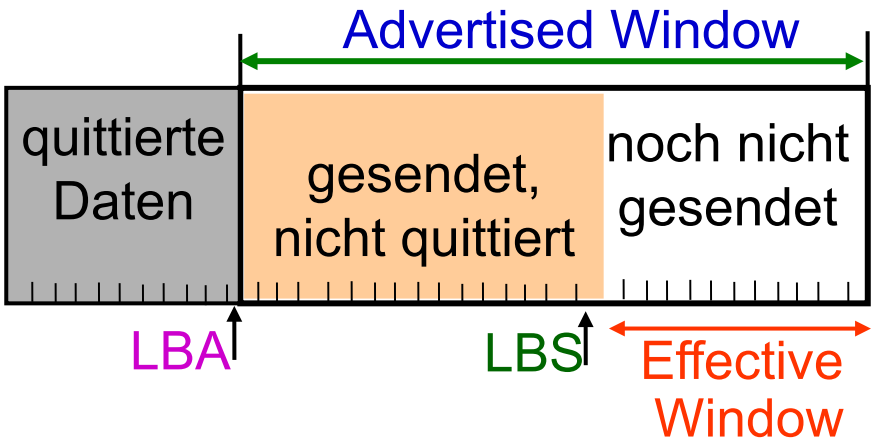
LBRc (LastByteRcvd): Letzte SeqNo, die in den RcvBuffer geschrieben wurde

AdvWindow = RcvBuffer – (LastByteRcvd – LastByteRead)

- Das Advertised Window wird im Empfänger neu bestimmt
  - bei jedem empfangenen Segment
  - falls die Anwendung Bytes ausgelesen hat

TCP – Flow Control im Sender

- Dem Sender wird der Sendekredit (AdvWindow) explizit mitgeteilt
  - wird dynamisch durch Empfänger verändert
  - bei Initialisierung: AdvWindow = RcvBuffer



LBS (LastByteSent): letzte SeqNo, die gesendet wurde

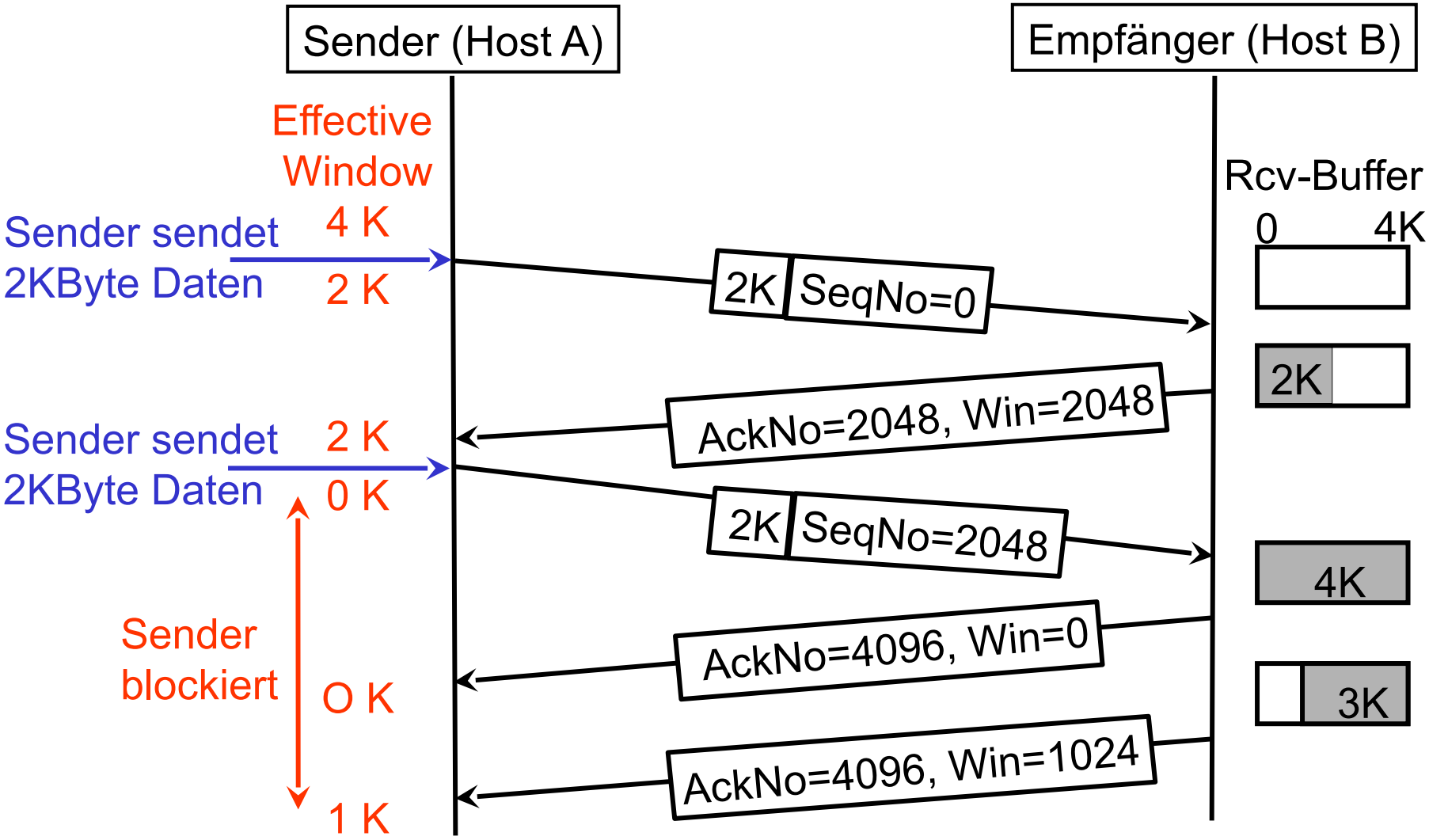
LBA (LastByteAcked): letzte SeqNo, die quittiert wurde

- Effective Window: Im Sender nutzbares Sendefenster

EffectiveWindow = AdvWindow – (LastByteSent – LastByteAcked)

noch nicht quittierte Anzahl von Bytes

Beispiel: TCP Flow Control



Zero Window Probing

Problem

- Host B hat dem Host A mitgeteilt, dass sein Empfangsbuffer voll ist (AdvWindow = 0)

  → *Host A kann keine weitere Daten an Host B senden*

- Host B hat keine weiteren Daten mehr an Host A zu senden

  → *Host B kann Host A nicht mehr darüber informieren, dass wieder freier Pufferplatz zur Verfügung steht*

Lösung

- Zero Window Probing
  - Host A darf auch bei AdvWindow = 0 Dateneinheiten mit 1 Byte senden, die von Host B quittiert werden
  - Persistent-Timer steuert das Zero Window Probing

Sequenznummernüberlauf

Problem

- Die Sequenznummer ist eine 32 Bit Binärzahl
  - Es können maximal 2^{32} Bytes = 4 GByte an Daten nummeriert werden
- Bei mehr als 4 GByte an Daten beginnen die Seq.-nummern von vorne
  → zwei Segmente mit der gleichen SeqNo sind im Internet vorhanden

Beispiel

- Die Zeitdauer T_{SO} bis zum Sequenznummernüberlauf ist von der Sendegeschwindigkeit abhängig

$$T_{SO} = 8 \cdot 2^{32} \text{Bit} / v_b$$

- Annahme: TCP sendet mit der Bitrate v_b:

Bitrate v_b	Zeitdauer T_{SO}
64 kbit/s	6,6 Tage
10 Mbit/s	57 Minuten
155 Mbit/s	3,7 Minuten
1 Gbit/s	34,4 sec

- maximale Lebenszeit eines Segments: 120s
  → Sequenznummernüberlauf möglich

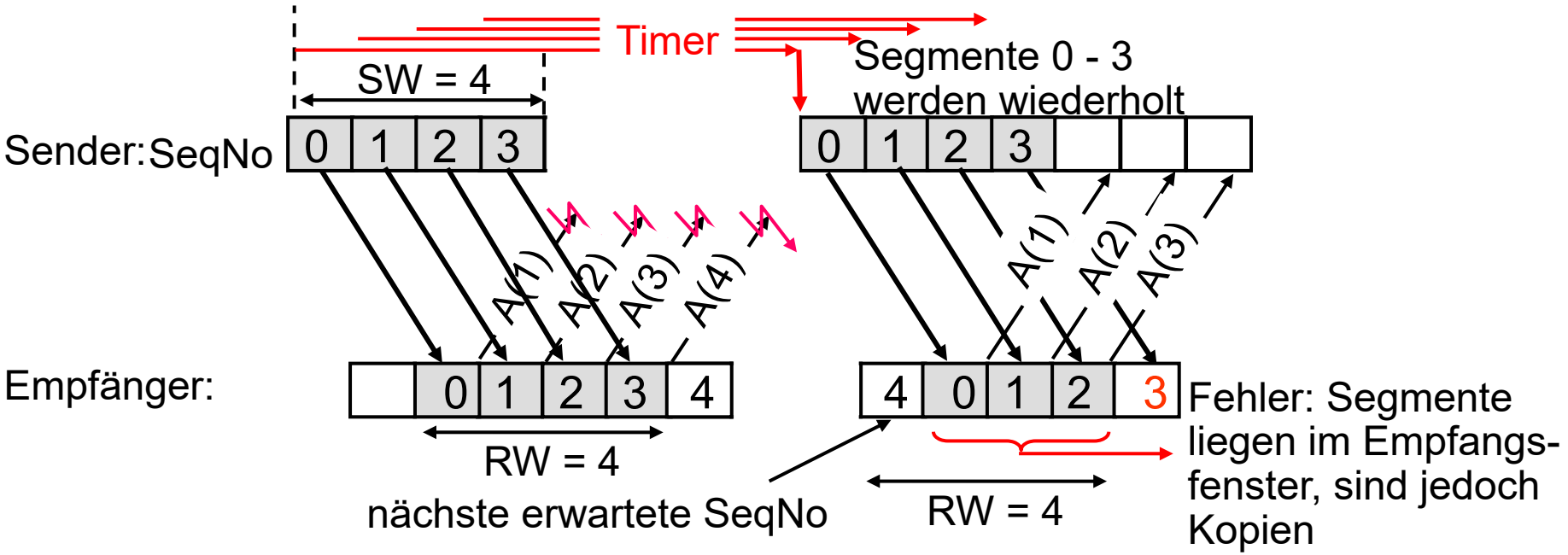
Sequenznummernüberlauf

„TCP Variante“: Sliding Window mit Timer je gesendetem Segment

- Empfangsfenster: RW = 4
- Sendefenster (AdvWindow): SW = 4
- Sequenznummern (je Byte): $N_{max} = 5$
- Je Segment: MSS = 1Byte
- Beispiel: Alle ACKs eines Fensters gehen verloren

→ Bedingung für RW=SW:
$N_{max} > 2 \cdot RW - 1$

$SeqNo \in \{0,1,2,3,4\}$



Congestion Control (Congestion Window)

Congestion (Überlast)

- Anschaulich:

  „Zu viele Quellen senden zu schnell zu viele Daten, sodass das Netzwerk überlastet wird“

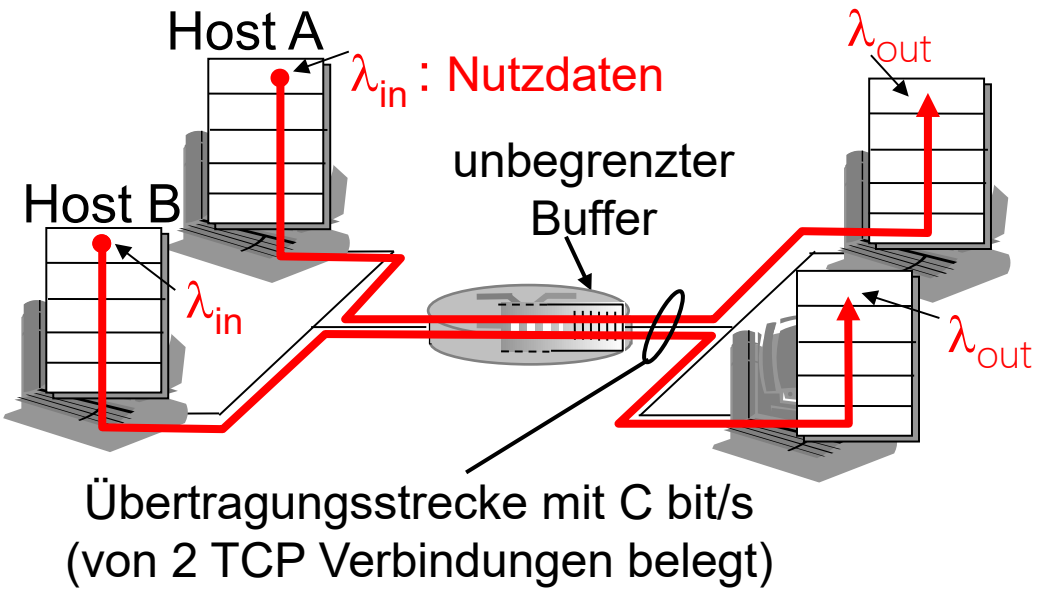
- Auswirkungen der Überlast:
  - Paketverluste (Bufferüberlauf im Router)
  - große Paketverzögerung (lange Warteschlangen in den Routern)

- Eines der wichtigsten Designprobleme im Internet

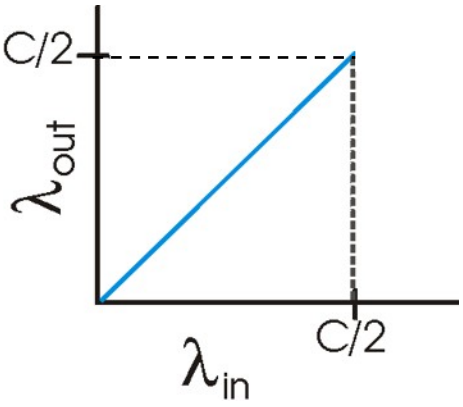
Ursachen einer Überlast

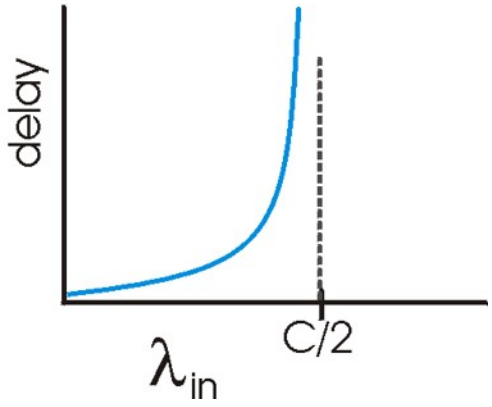
Beispiel 1:

- 2 Sender und 2 Empfänger
- Anwendung sendet Daten mit der Rate λ_{in}
- Empfänger liefert Daten mit der Rate λ_{out}
- Ein Router, unendlich großer Buffer
- keine Paketverluste



- Maximal erreichbarer fairer Durchsatz je Verbindung:
  $\lambda_{in} = \lambda_{out} < C / 2$
- Überlast bei hoher Auslastung
  $\lambda_{in}^{Host\ A} + \lambda_{in}^{Host\ B} \rightarrow C$
  → große Verzögerungszeiten

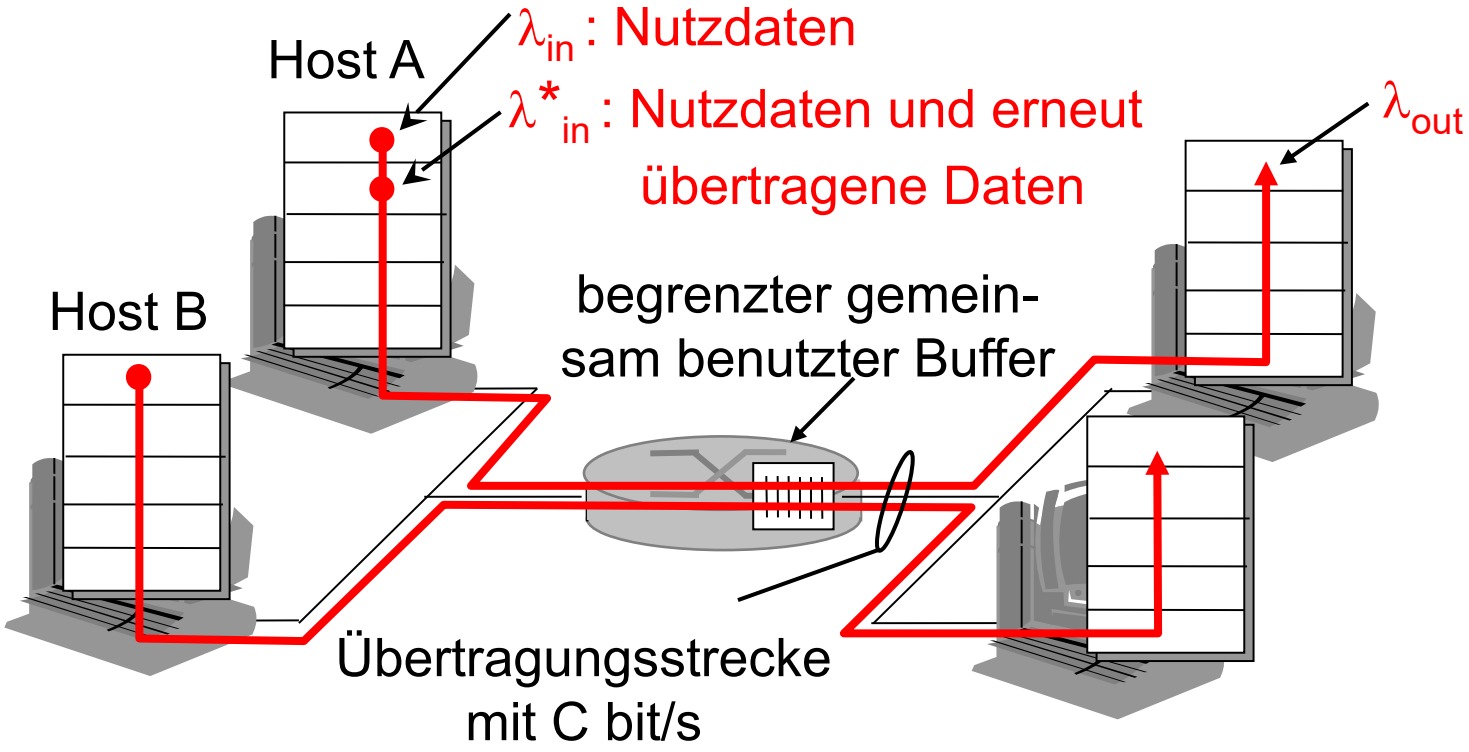




Ursachen einer Überlast - Beispiel 2

Beispiel 2:

- ein Router, **endlicher** Buffer
- Sender wiederholt verloren gegangene Segmente
  - Nutzdaten und Übertragungswiederholungen ergeben die Rate $\lambda^*_{in} > \lambda_{in}$

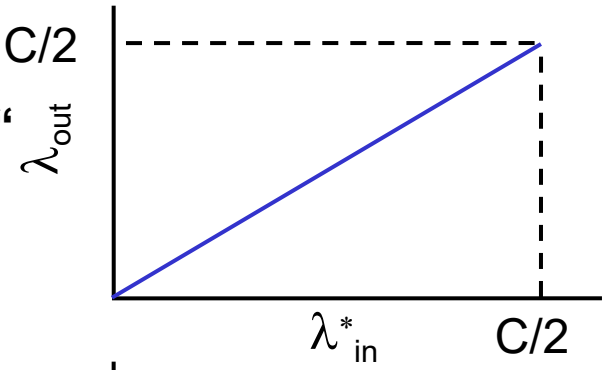


Ursachen einer Überlast - Beispiel 2

Ideales Verhalten:

siehe „Flow-Control“

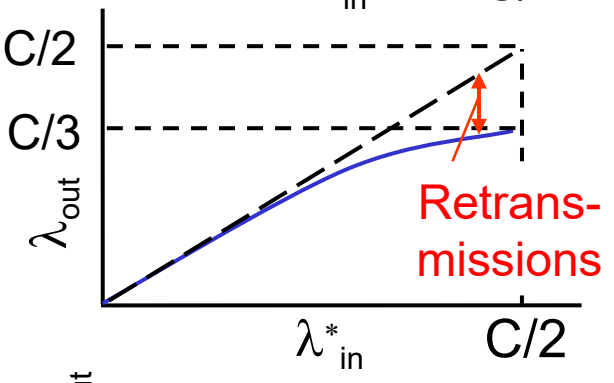
- Quelle sendet nur, falls Buffer frei
- keine Paketverluste: $\lambda_{out} = \lambda_{in} = \lambda^*_{in} \leq C/2$



Reales Verhalten:

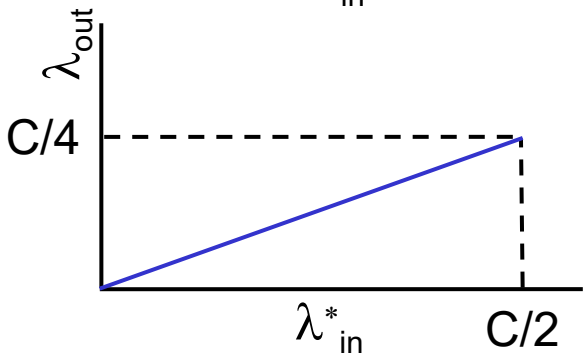
- I) Fehlendes ACK zeigt Paketverluste an
- fehlende Pakete werden wiederholt

$$\lambda_{out} < \lambda^*_{in} \leq C/2$$



- II) Timer läuft ab, aber kein Paketverlust
- alle Daten werden z. B. doppelt übertragen

$$\lambda_{out} = \lambda^*_{in}/2 \leq C/4$$



“Kosten” der Überlast:

- Das Netzwerk muss mehr Daten für den
- gleichen Durchsatz transportieren

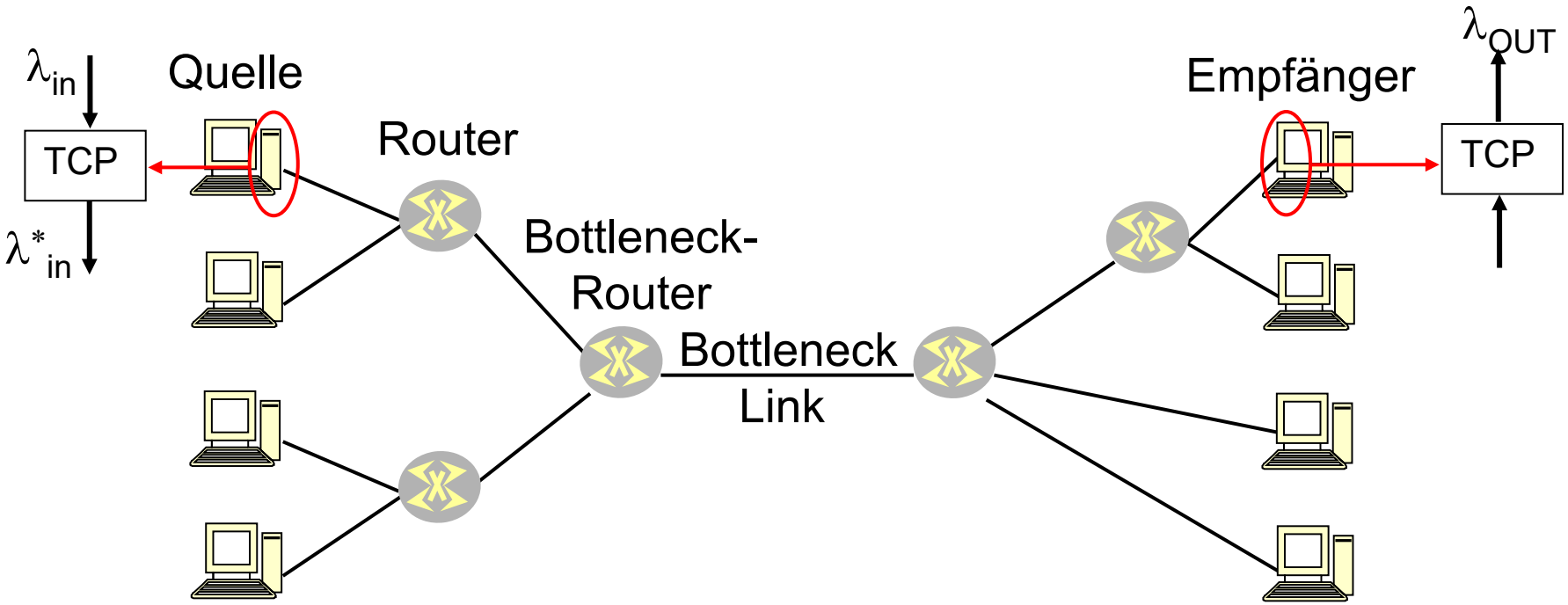
Ursachen einer Überlast - Beispiel 3

Beispielnetz mit Bottleneck - Link

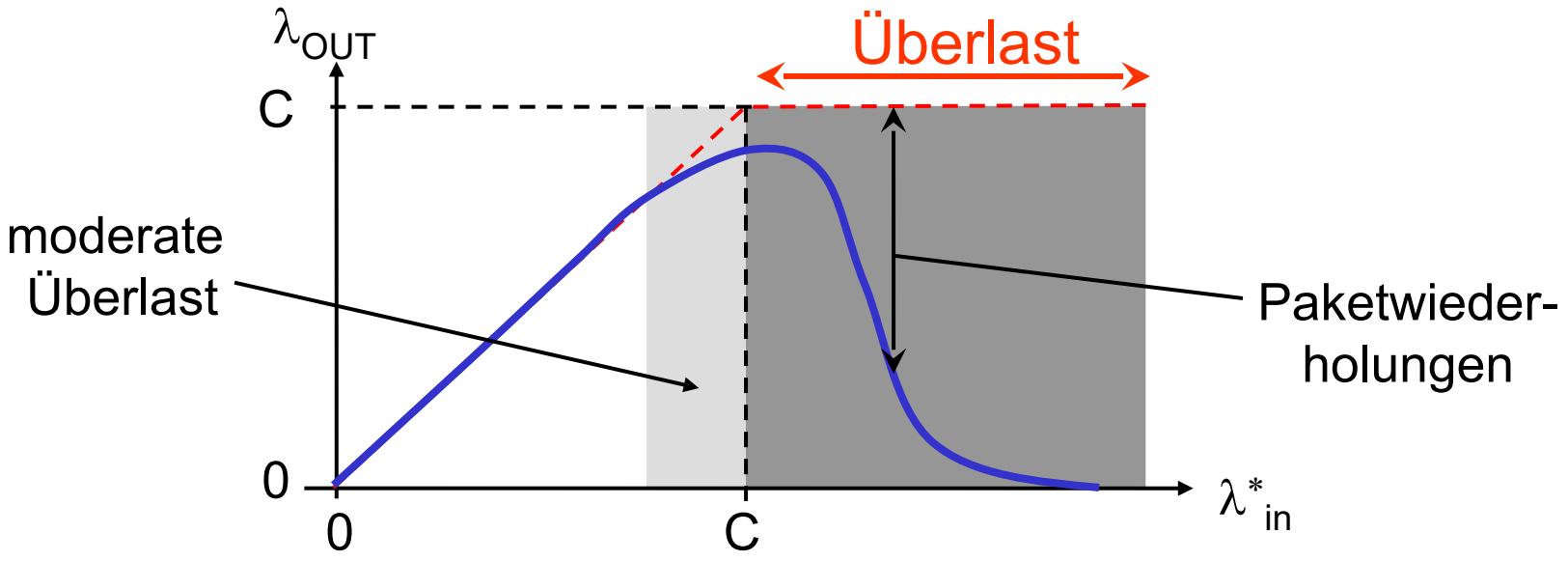
- Alle Links 10 Mbit/s außer Bottleneck-Link (1 Mbit/s)
- Jeweils eine Verbindung vom Sender zum Empfänger

Frage:

- Was passiert, wenn die Quellen ihre Nutzdatenrate λ_{in} erhöhen?



Ursachen einer Überlast - Beispiel 3



- Aufgrund einer stark ansteigenden Zahl von Paketwiederholungen kann der erfolgreiche Durchsatz im Überlastbereich zusammenbrechen
- Geht ein Paket verloren, wird die Kapazität der Links durch die Übertragungswiederholungen verschwendet

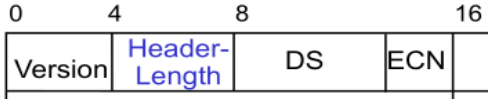
Schlussfolgerung: Überlastbereich muss vermieden werden

Ende-zu-Ende Congestion Control

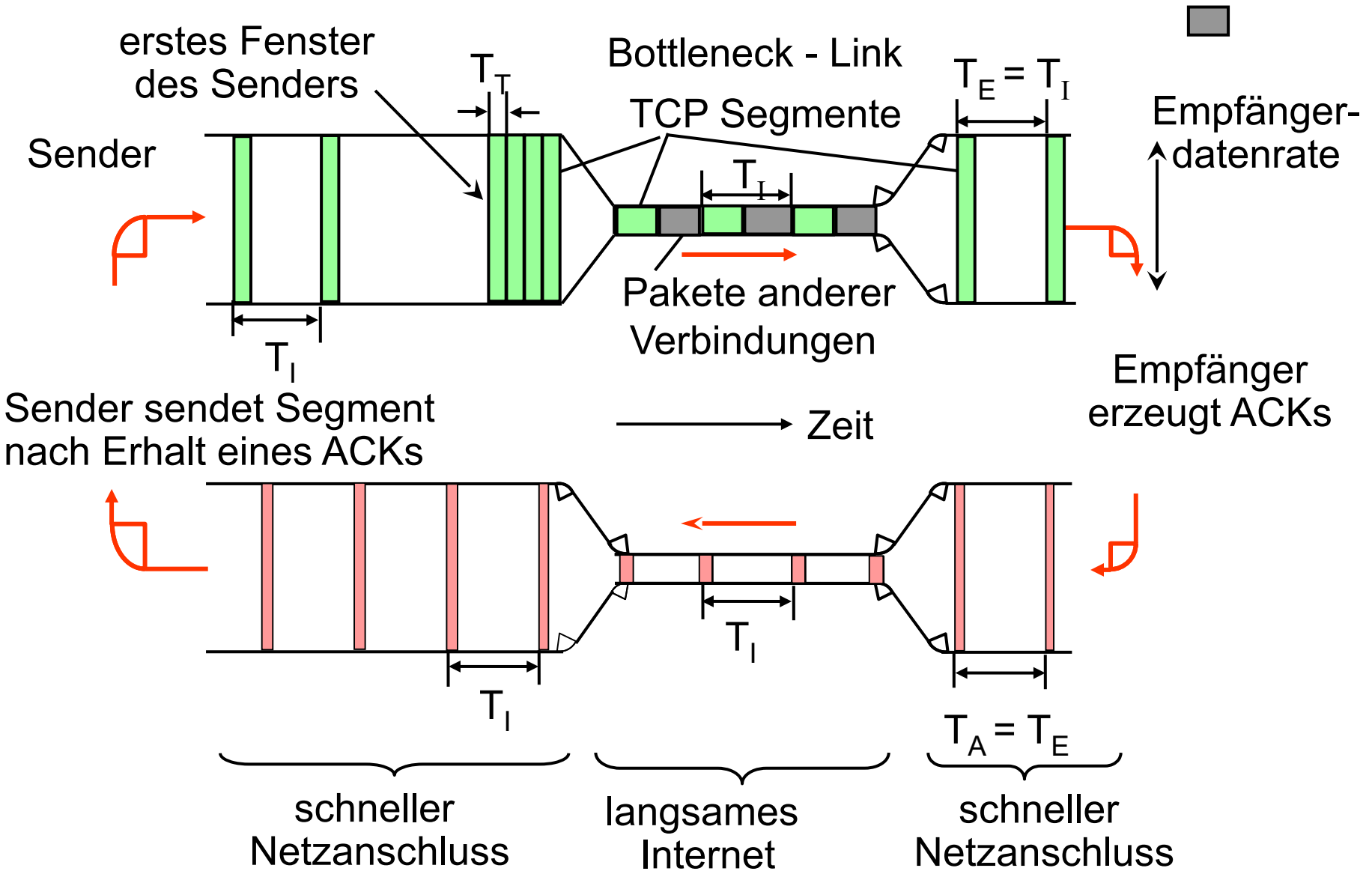
- Die IP-Vermittlungsschicht bietet der Transportschicht keine Unterstützung für die Überlastkontrolle
- Die Endsysteme müssen das Netzwerkverhalten beobachten, um auf eine Überlast zu schließen
  - Beobachtung der Datenverluste und Verzögerungsdauer
  - Beispiel: **TCP Congestion Control**

Netzwerkunterstützte Überlastkontrolle

- Netzknoten (Router) melden dem Sender explizit den Überlastzustand
  - durch gezieltes frühzeitiges Verwerfen von Daten
    - Beispiel: **Random Early Detection** in IP-Netzen
  - durch spezielle Meldungen, Beispiele sind:
    - IP-Erweiterung **Explicit Congestion Notification**
    - ABR-Überlastkontrolle bei ATM (Asynchroner Transfer Mode)



Gleichgewicht bei TCP durch Self-Clocking



Gleichgewicht bei TCP durch Self-Clocking

Probleme:

- Verbindung erreicht nicht das Gleichgewicht

  → Kapazität des Netzes bleibt unbenutzt

- Bottleneck-Link ist überlastet
  - volle Warteschlange des Bottleneck-Links führt zu hohen RTTs
  - Retransmissions nach Timeouts erhöhen die Überlastung
  - sobald ein Segment aus der überlasteten Warteschlange ausgeliefert wird, wird ein neues gesendet

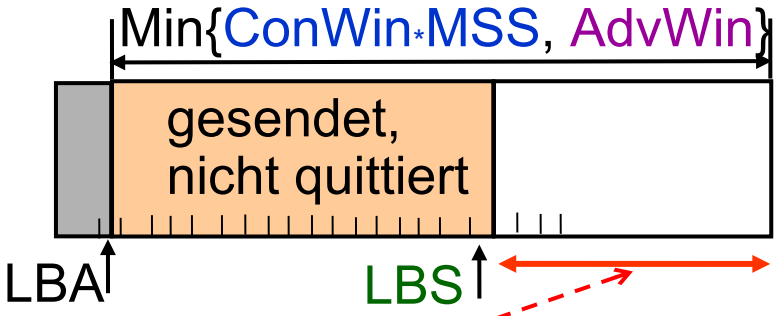
  → **Überlast kann sich allein durch die Self-Clocking-Eigenschaft nicht abbauen**

Lösungsansatze für TCP - Congestion Control

- **Congestion Detection**
  - Ablauf des Retransmission Timers: Verlust von vielen Segmenten
  - Duplicate ACKs: Verlustes von einzelnen Segmenten
- **Congestion Window**
  - Steuerung der Datenrate des Senders
- **Slow Start**
- **Congestion Avoidance (CA) mit Fast-Recovery**

Congestion Window

- Der Sender verwendet ein **Congestion Window (ConWin)**
- Der Sender darf nie mehr als **ConWin * MSS Bytes** ohne Quittierung senden
- ConWin wird in der Einheit Maximum Segment Size (MSS) angegeben
  - Default nach RFC 1122: MSS = 536 Byte
- Congestion Control und Flow Control sind im Sender gekoppelt
  - Maximal sendbar ist das **Effective Window = Minimum aus ConWin * MSS und AdvWindow**



Congestion Control, „Feedback des Netzwerks“

Flow Control, durch Empfänger gesteuert

Effective Window = Min{ConWin*MSS, AdvWindow} – (LBS – LBA)

noch nicht quittierte Anzahl von Bytes

Congestion Control - Phasen

Congestion Control funktioniert heutzutage in zwei Phasen:

- **Slow Start**
  - Von einem geringen Anfangswert aus wird die Datenrate der Verbindung gesteigert, bis erste Verluste entstehen
- **Congestion Avoidance with Fast Recovery** (TCP RENO)
  - Verlustabhängige Steuerung des Congestion Windows

Slow Start

Slow Start

Prinzip

- Mit einem kleinen Wert des Congestion Windows beginnend, wird dieses exponentiell erhöht, solange keine Überlast auftritt
- Slow Start wird bei Überschreitung eines Grenzwertes beendet
  - **Threshold (Thres)**: Grenzwert in Byte (default: 64 kByte)

Verfahren

- **Start:** Beim Verbindungsstart oder nach einem Segmentverlust

  ConWin = 1 MSS

- **Eingehendes ACK:**

  ConWin = ConWin + 1 MSS

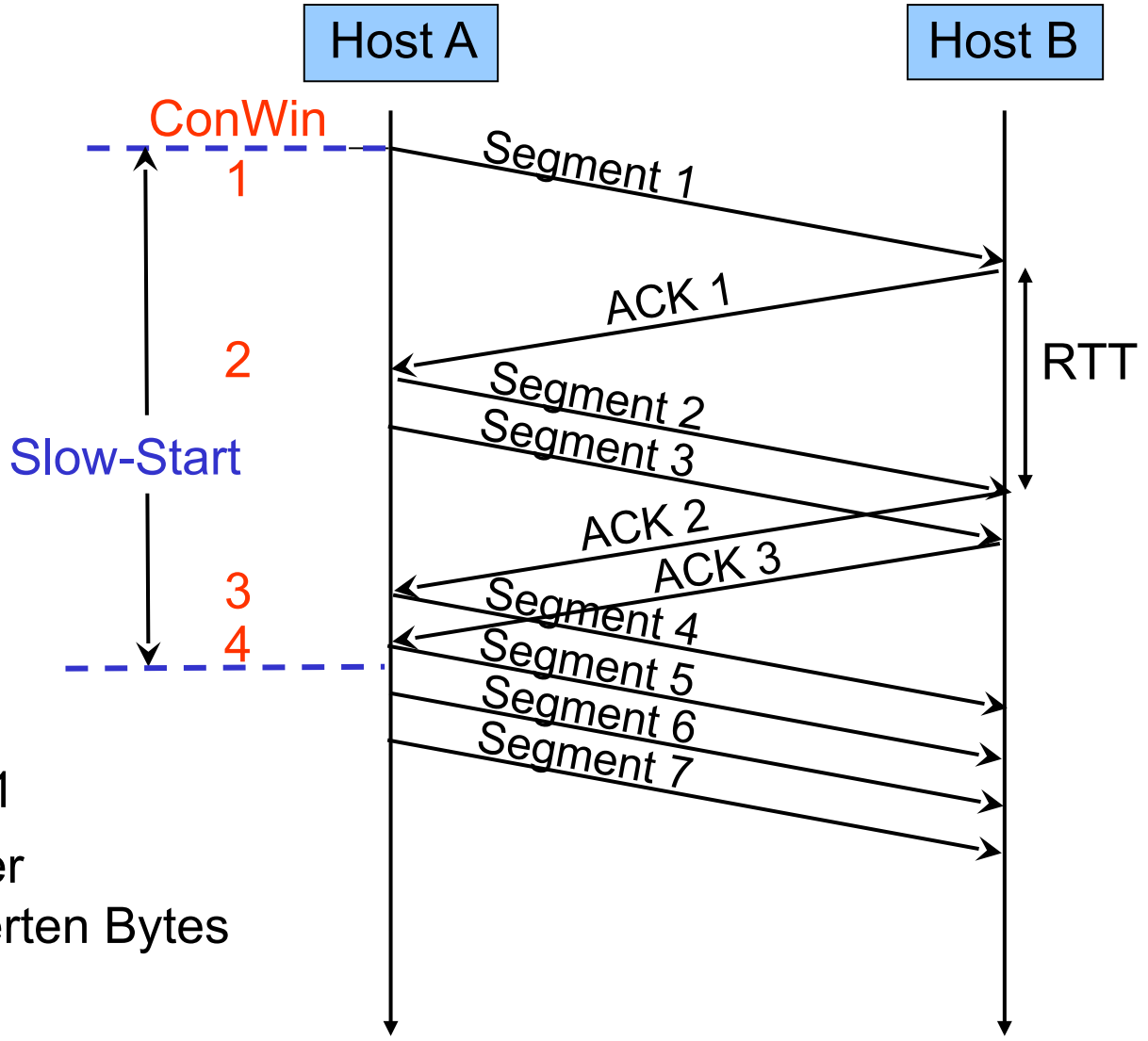
- **Stop:** Slow Start endet, falls ConWindow den Grenzwert überschreitet

  ConWin > Thres → stop Slow Start

Besipiel: Slow Start

Initialisierung:

- AdvWin > 50 MSS
- Tresh = 3 MSS
- kein delayed Ack



- Jedes ACK erhöht das ConWin um 1
- unabhängig von der Anzahl der quittierten Bytes