Netzwerktechnologien
3 VO

Dr. Ivan Gojmerac
ivan.gojmerac@univie.ac.at

6. Vorlesungseinheit, 24. April 2013

Bachelorstudium Medieninformatik
SS 2013

3.5.3 TCP Rundlaufzeit und Timeout

Wie bestimmt TCP den Wert für den Timeout?

- Muss größer als die Rundlaufzeit (Round Trip Time, RTT) sein
  Aber RTT ist nicht konstant!
- Wird der Wert zu klein gewählt (zu kurz): → unnötige Timeouts
  Führt zu unnötigen Übertragungswiederholungen!
- Wird der Wert zu groß gewählt (zu lang): → langsame Reaktion auf den Verlust von Segmenten

Wie kann man die RTT schätzen?

- **`SampleRTT`**: gemessene Zeit vom Absenden eines Segments bis zum Empfang des dazugehörenden ACKs
  - Segmente mit Übertragungswiederholungen werden ignoriert
- **`SampleRTT`** ist nicht konstant, daher wird der Durchschnitt über mehrere Messungen verwendet

3.5.3 TCP Rundlaufzeit und Timeout

Der Durchschnitt der **`SampleRTT`** Messungen wird **`EstimatedRTT`** genannt.

Wenn eine neue **`SampleRTT`** gemessen wird, aktualisiert TCP den Wert **`EstimatedRTT`** entsprechend der folgenden Formel:

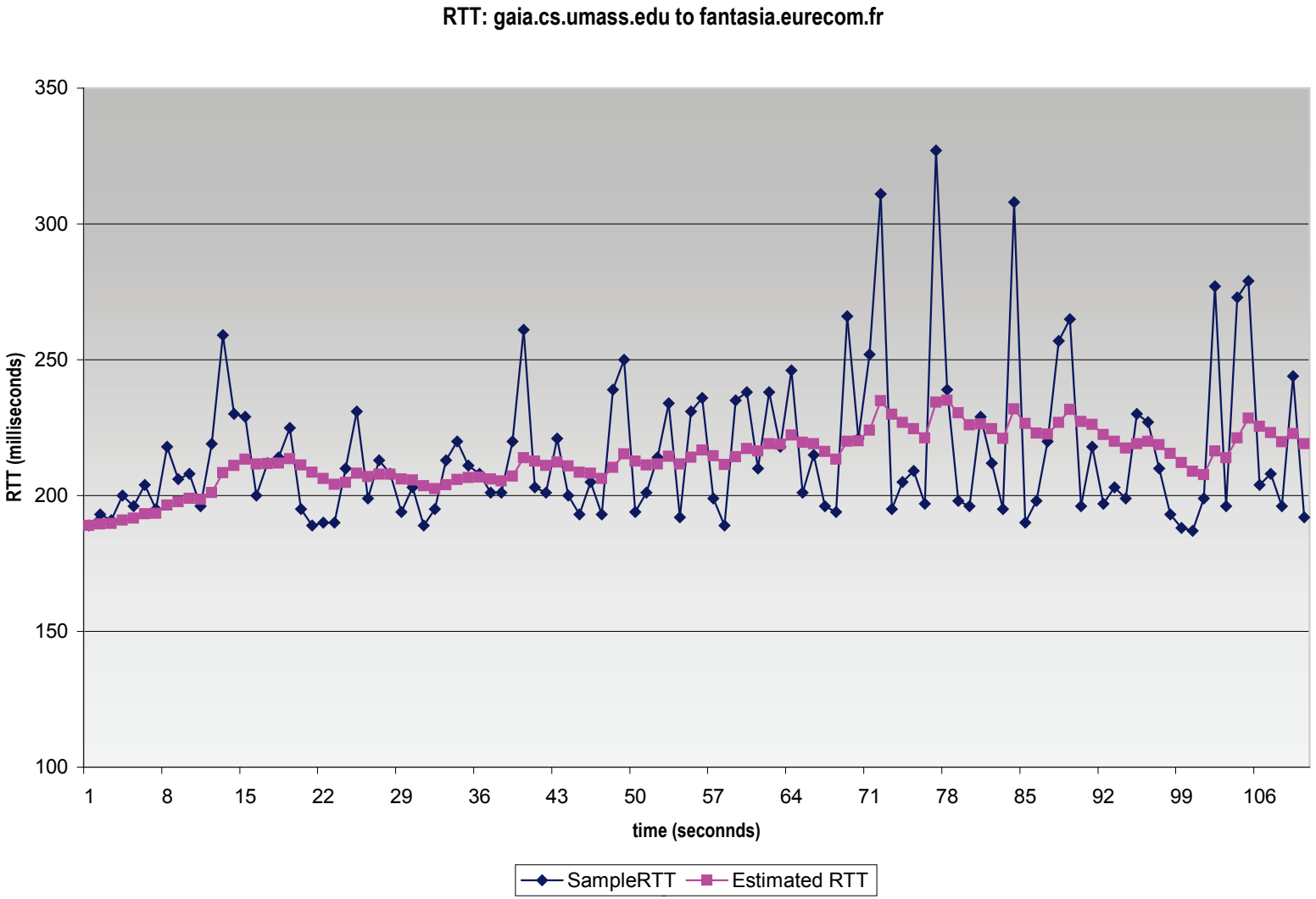
$$\texttt{EstimatedRTT} = (1-\alpha)*\texttt{EstimatedRTT} + \alpha*\texttt{SampleRTT}$$

Der neue Wert von **`EstimatedRTT`** ist eine gewichtete Kombination des vorherigen Wertes von **`EstimatedRTT`** und der neuen **`SampleRTT`.**

→ Einfluss vergangener Messungen verringert sich exponentiell schnell

→ Üblicher Wert: $\alpha = 0.125$ (RFC 2988)

3.5.3 Beispiel für die Rundlaufzeit Bestimmung



3.5.3 TCP Rundlaufzeit und Timeout

Wie bestimmt TCP den Wert für den Timeout?

- **`EstimatedRTT`** + "Sicherheitsabstand"
  Größere Schwankungen von **`EstimatedRTT`** → größerer Sicherheitsabstand
- Formel zur Bestimmung des Maßes der Variabilität der **`RTT`**:
  (Abschätzung wie sehr die **`SampleRTT`** typischerweise von der **`EstimatedRTT`** abweicht)

$$\mathbf{DevRTT = (1-\beta)*DevRTT + \beta*|SampleRTT-EstimatedRTT|}$$

(üblicherweise: $\beta = 0.25$)

Daraus wird der Timeout folgendermaßen bestimmt:

$$\mathbf{TimeoutInterval = EstimatedRTT + \underbrace{4*DevRTT}_{\text{Sicherheitsabstand}}}$$

3.5.4 Zuverlässiger Datentransfer mit TCP

- TCP stellt einen zuverlässigen Datentransfer über den unzuverlässigen Datentransfer von IP zur Verfügung
- Pipelining von Segmenten
- Kumulative ACKs
- TCP verwendet einen einzigen Timer für Übertragungswiederholungen
- Übertragungswiederholungen werden ausgelöst durch:
  - Timeout
  - Doppelte ACKs

Zu Beginn betrachten wir einen vereinfachten TCP-Sender:

- Ignorieren von doppelten ACKs
- Ignorieren von Fluss- und Überlastkontrolle

3.5.4 Zuverlässiger Datentransfer mit TCP

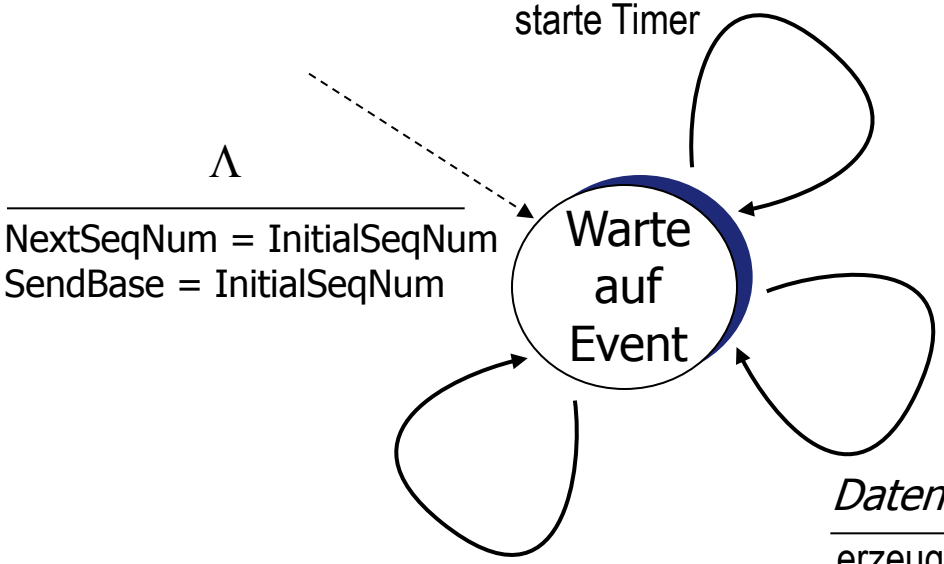
TCP Ereignisse im Sender:

- Daten von Anwendung erhalten:
  - Erzeuge Segment mit geeigneter Sequenznummer
    → Nummer des ersten Byte im Datenteil
  - Timer starten, wenn er noch nicht läuft
    → Timer für das älteste unbestätigte Segment
  - Laufzeit des Timers: `TimeOutInterval`
- Timeout:
  - Erneute Übertragung des Segments, für das der Timeout aufgetreten ist
  - Starte Timer neu
- ACK empfangen:
  - Wenn damit bisher unbestätigte Daten bestätigt werden:
    → Aktualisiere die Informationen über bestätigte Segmente
    → Starte Timer neu, wenn noch unbestätigte Segmente vorhanden sind

3.5.4 TCP-Sender – stark (!) vereinfacht

Timeout

übertrage das unbestätigte Segment mit der kleinsten Sequenznummer erneut
starte Timer



ACK empfangen für Sequenznummer y

```
if (y > SendBase) {
    SendBase = y
    if (es gibt noch unbestätigte Segmente)
        starte Timer
}
```

Daten von der Anwendung erhalten

```
erzeuge TCP Segment mit Sequenznummer NextSeqNum
if (Timer läuft nicht)
    starte Timer
gib Segment an IP weiter
NextSeqNum = NextSeqNum + length(data)
```

3.5.4 TCP-Sender – stark (!) vereinfacht

```
NextSeqNum = InitialSeqNum
SendBase = InitialSeqNum

loop (forever) {
  switch(event)

  event: Daten von der Anwendung
      erzeuge TCP Segment mit…
            …Sequenznummer NextSeqNum
      if (Timer läuft nicht)
          starte Timer
      gib Segment an IP weiter
      NextSeqNum = NextSeqNum + length(data)

  event: Timeout
      übertrage das unbestätigte Segment mit der kleinsten Sequenznummer erneut
      starte Timer

  event: ACK empfangen für Sequenznummer y
      if (y > SendBase) {
          SendBase = y
          if (es gibt noch unbestätigte Segmente)
              starte Timer
      }
} /* end of loop forever */
```

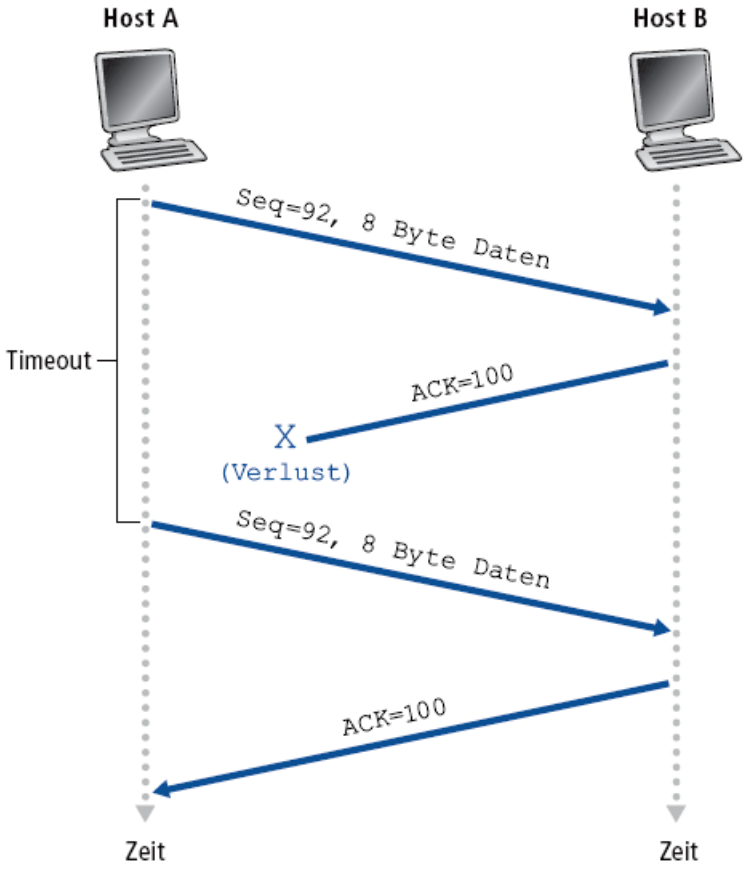
Anmerkung:
SendBase - 1 ist das letzte erfolgreich bestätigte Byte.

Beispiel:
- SendBase - 1 = 71
- ACK Wert y = 73 kommt an

Ist y > SendBase, dann bestätigt dieser ACK Wert y ein oder mehrere zuvor unbestätigte Segmente. (*Kumulative ACKs! → ACK Wert y bestätigt auch Segment 72 und 73*)

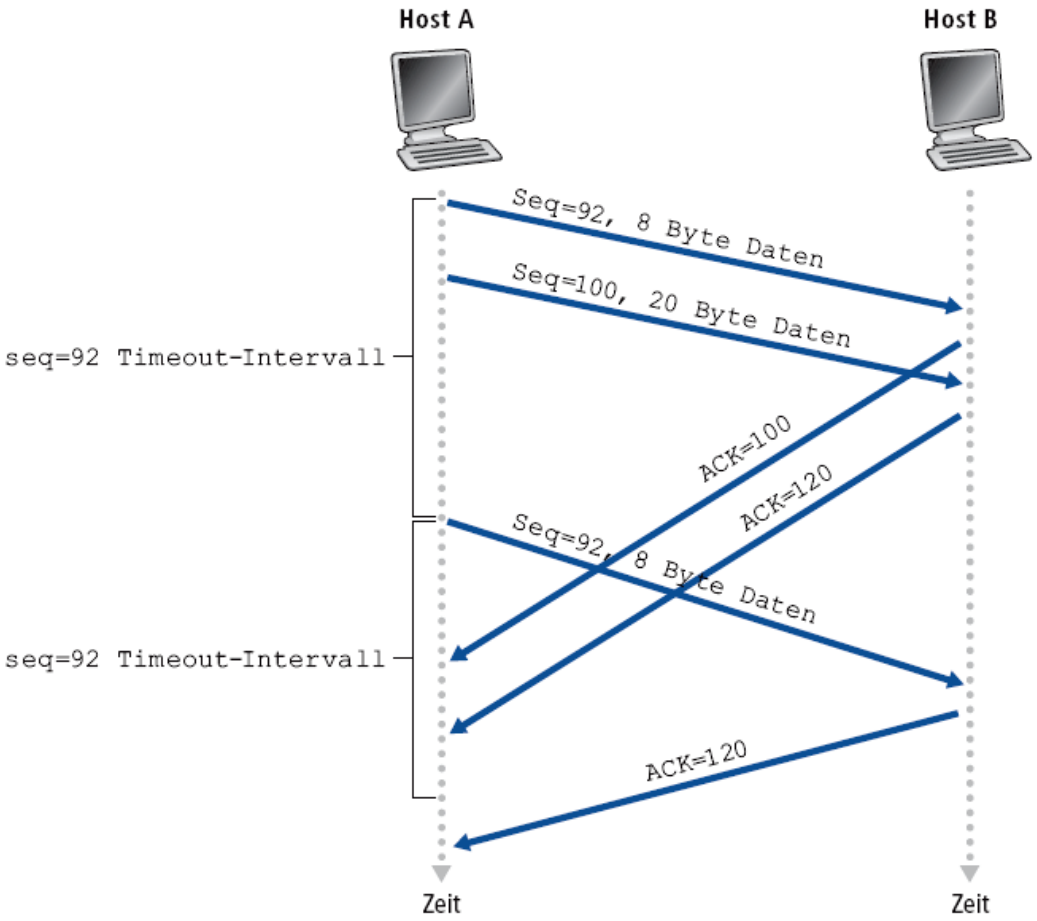
3.5.4 Beispiele für Übertragungswiederholungen: Paketverlust

Erneute Übertragung aufgrund eines verloren gegangenen ACKs:



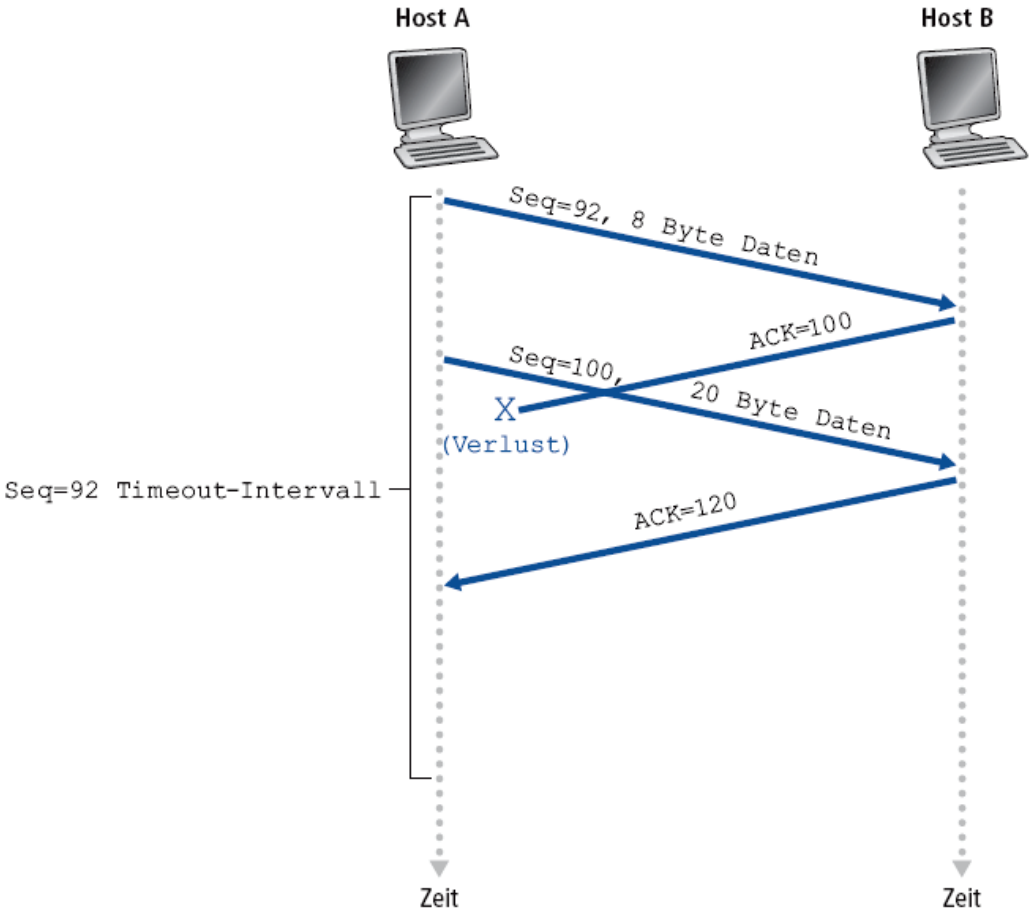
3.5.4 Beispiele für Übertragungswiederholungen: Verfrühte Timeouts

Segment 100 wird nicht erneut übertragen:



3.5.4 Beispiele für Übertragungswiederholungen: Kumulative ACKs

Das kumulative ACK verhindert die erneute Übertragung des ersten Segments:



3.5.4 TCP ACK-Erzeugung

Ereignis	Aktion des TCP-Empfängers
Ankunft des Segmentes in der richtigen Reihenfolge mit der erwarteten Sequenznummer. Alle Daten bis zur erwarteten Sequenznummer sind bereits bestätigt.	Verzögertes ACK. Wartet bis zu 500 ms auf die Ankunft eines anderen Segmentes in richtiger Reihenfolge. Wenn das nächste Segment nicht in diesem Zeitintervall eintrifft, wird ein ACK gesendet.
Ankunft eines Segmentes in der richtigen Reihenfolge mit erwarteter Sequenznummer. Ein anderes Segment in der korrekten Reihenfolge wartet auf die ACK-Übertragung.	Sendet sofort ein einzelnes kumulatives ACK, bestätigt beide in richtiger Reihenfolge eingetroffene Segmente.
Ankunft eines Segmentes außerhalb der Reihenfolge mit einer Sequenznummer, die größer ist als erwartet. Lücke im Bytestrom aufgetreten.	Sendet sofort ein doppeltes ACK, in dem er die Sequenznummer des nächsten erwarteten Bytes angibt.
Ankunft eines Segmentes, das die Lücke in den erhaltenen Daten ganz oder teilweise ausfüllt.	Sendet sofort ein ACK, vorausgesetzt, das Segment beginnt mit der Sequenznummer des nächsten erwarteten Bytes. Bestätigt alle nun lückenlos vorliegenden Bytes.

(Definiert in RFC 1122 und RFC 2581)

3.5.4 Fast Retransmit

- Zeit für Timeout ist häufig sehr lang:
  → Große Verzögerung vor einer Neuübertragung
- Erkennen von Paketverlusten durch doppelte ACKs:
  – Sender schickt häufig viele Segmente direkt hintereinander
  → Wenn ein Segment verloren geht, führt dies zu vielen doppelten ACKs

- Wenn der Sender 3 Duplikate eines ACK erhält (also insgesamt 4 Stück!), dann nimmt er an, dass das Segment verloren gegangen ist:
  → **Fast Retransmit** (schnelle Sendewiederholung):
  Segment erneut schicken, bevor der Timer ausläuft

3.5.4 Fast Retransmit Algorithmus

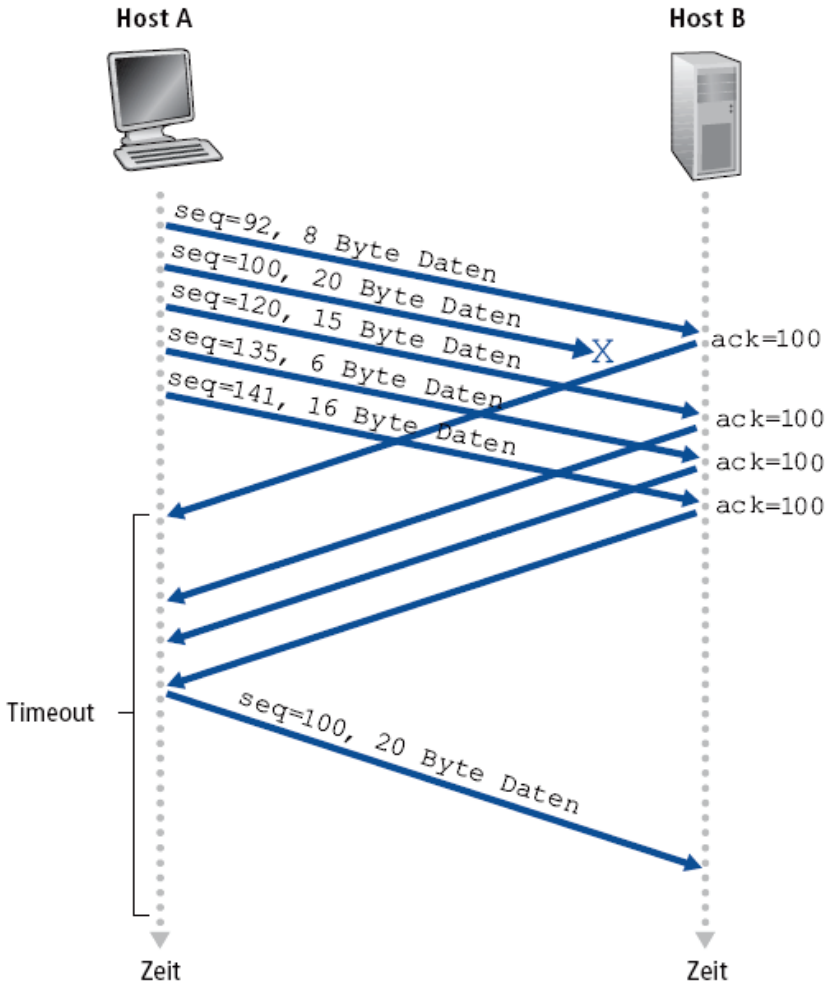
```
event: ACK empfangen, ACK-Nummer ist y
        if (y > SendBase) {
            SendBase = y
            if (wenn es noch andere unbestätigte Segmente gibt)
                starte Timer
        }
        else {
            erhöhe den Zähler für doppelte ACKs für y um eins
            if (Zähler für doppete ACKs für y = 3) {
                Neuübertragung des Segments mit Sequenznummer y
            }
        }
```

Ein doppeltes ACK für ein bereits bestätigtes Segment

Fast Retransmit

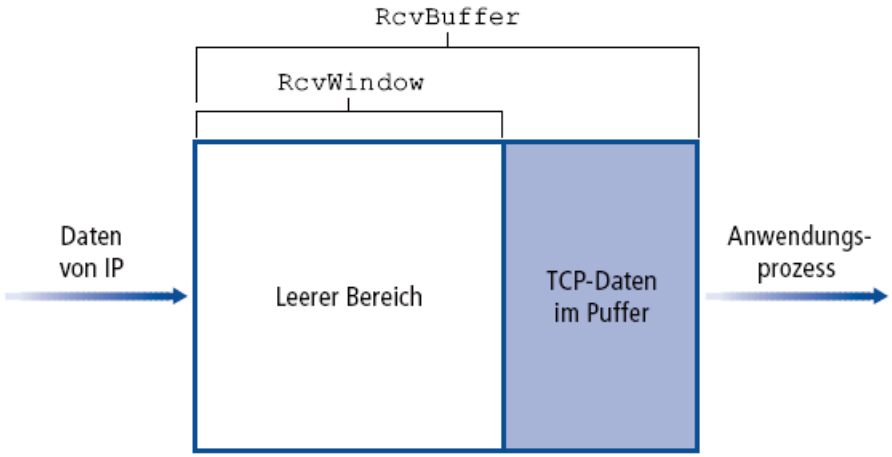
3.5.4 Fast Retransmit

Erneute Übertragung des fehlenden Segmentes, bevor der Timer des Segmentes abläuft:



3.5.4 TCP Flusskontrolle

Empfängerseite von TCP hat einen Empfängerpuffer:



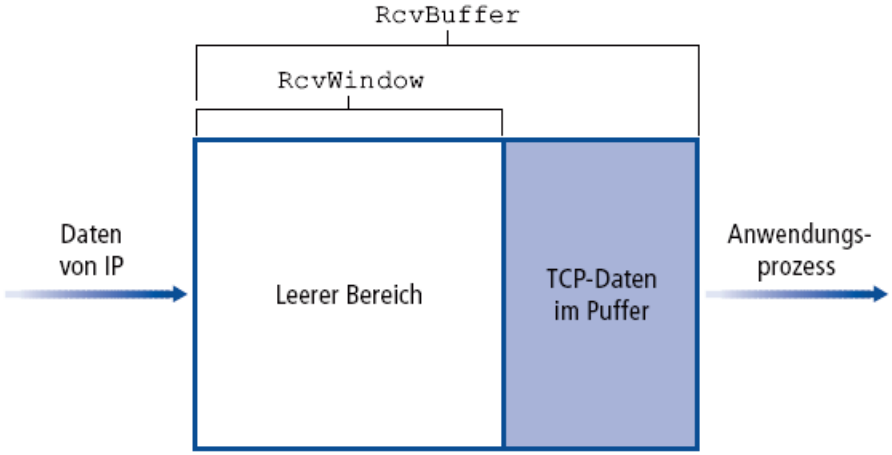
→ Die Anwendung kommt unter Umständen nicht mit dem Lesen hinterher.

TCP Flusskontrolle bedeutet, dass der Sender nicht mehr Daten schickt, als der Empfänger in seinem Puffer speichern kann.

Die Flusskontrolle ist ein Dienst zur Angleichung von Geschwindigkeiten: Senderate wird an die Verarbeitungsrate der Anwendung auf der Empfängerseite angepasst.

3.5.4 TCP Flusskontrolle: Funktionsweise

Annahme: Empfänger verwirft Segmente, die außer der Reihe ankommen.



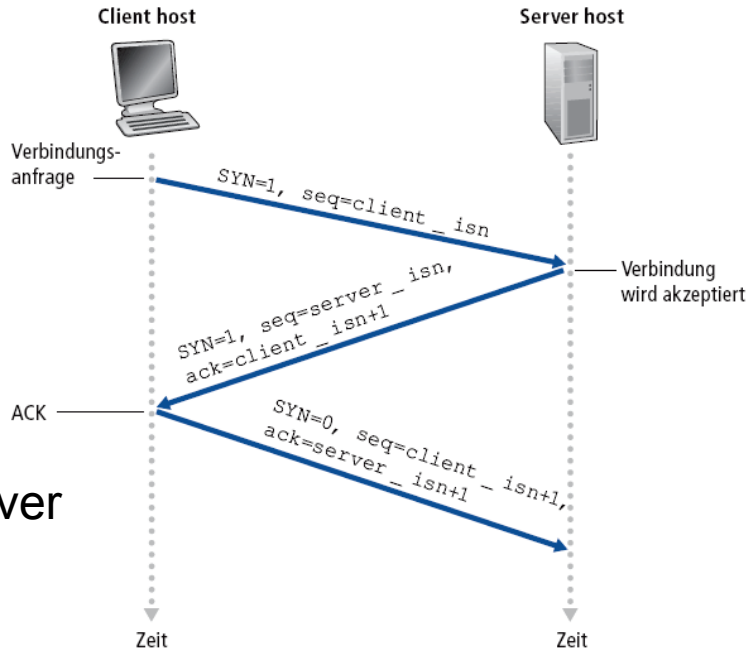
- Platz im Puffer ausgedrückt durch Variable **`RcvWindow`**
  `RcvWindow = RcvBuffer - [LastByteRcvd - LastByteRead]`
- Empfänger kündigt den Platz durch **`RcvWindow`** im TCP-Header an
- Sender begrenzt seine unbestätigt gesendeten Daten auf **`RcvWindow`**
  → Dann ist garantiert, dass der Puffer im Empfänger nicht überläuft

3.5.4 TCP Verbindungsmanagement

Erinnerung: TCP-Sender und Empfänger bauen eine Verbindung auf (Handshake), bevor sie Daten austauschen.

- Initialisieren der TCP-Variablen:
  Sequenznummern, Informationen für Flusskontrolle (z.B. **`RcvWindow`**)
- *Client:* (Initiator)
  ```
  Socket clientSocket = newSocket("hostname","port number");
  ```
- *Server:* (vom Client kontaktiert)
  ```
  Socket connectionSocket = welcomeSocket.accept();
  ```

3.5.4 TCP Verbindungsmanagement



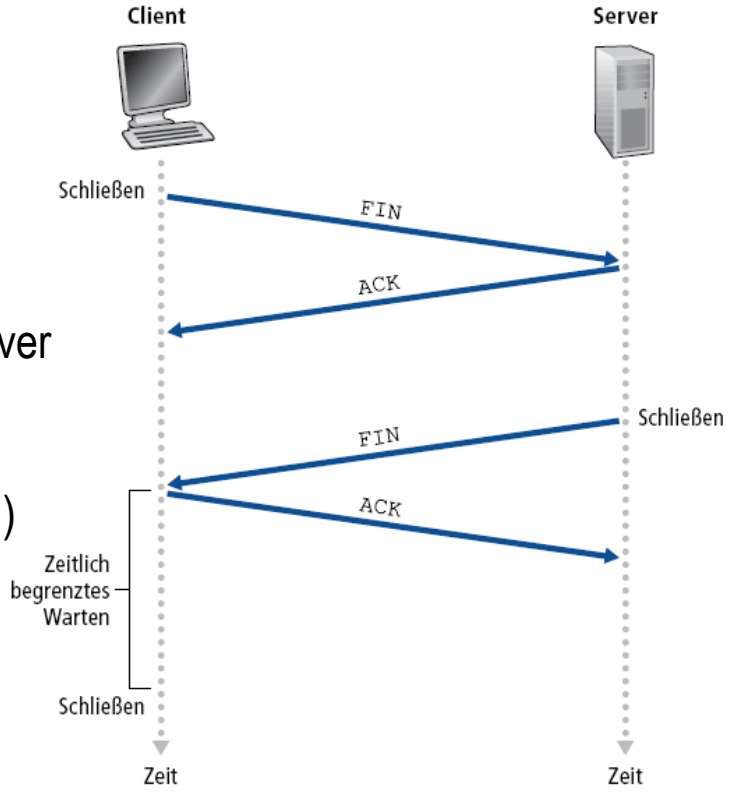
Drei-Wege-Handshake:

Schritt 1: Client sendet TCP-SYN-Segment an Server
- Initiale Sequenznummer (Client → Server)
- Keine Daten

Schritt 2: Server empfängt SYN und antwortet mit SYNACK
- Server legt Puffer an
- Initiale Sequenznummer (Server → Client)

Schritt 3: Client empfängt SYNACK und antwortet mit einem ACK
- Dieses Segment darf bereits Daten beinhalten

3.5.4 TCP Verbindungsmanagement



Schließen einer Verbindung:

Client schließt Socket: `clientSocket.close();`

Schritt 1: Client sendet ein TCP-FIN-Segment an den Server

Schritt 2: Server empfängt FIN, antwortet mit ACK;
dann sendet er ein FIN (kann im gleichen Segment erfolgen)

Schritt 3: Client empfängt FIN und antwortet mit ACK

- Beginnt einen "Timed- Wait"-Zustand: er antwortet auf Sende-wiederholungen des Servers mit ACK

Schritt 4: Server empfängt ACK und schließt die Verbindung

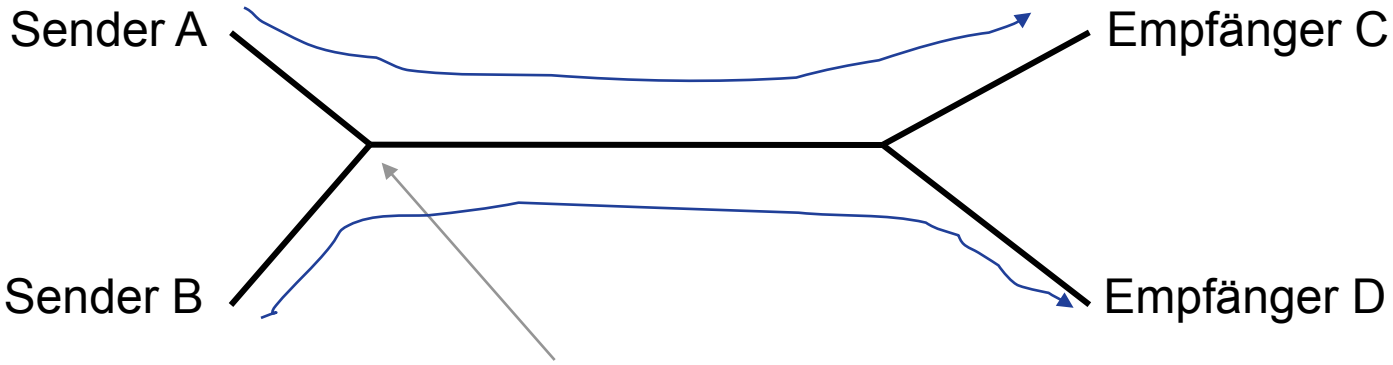
→ Mit kleinen Änderungen können so auch gleichzeitig abgeschickte FINs behandelt werden!

3.6 Grundlagen der Überlastkontrolle

3.6 Grundlagen der Überlastkontrolle

Problem:

Wenn alle Sender im Netz immer so viele Pakete losschicken, wie bei den Empfängern in den Puffer passen, dann kann es zu Überlast (*congestion*) im Netz kommen, da nicht auf die momentane Situation im Netz Rücksicht genommen wird.



Wenn alle Verbindungen die gleiche Bandbreite haben und sowohl A als auch B mit der Bandbreite der Verbindung senden, dann kommt es **hier** zu Netzwerküberlast (congestion)!

3.6 Grundlagen der Überlastkontrolle

Überlast:

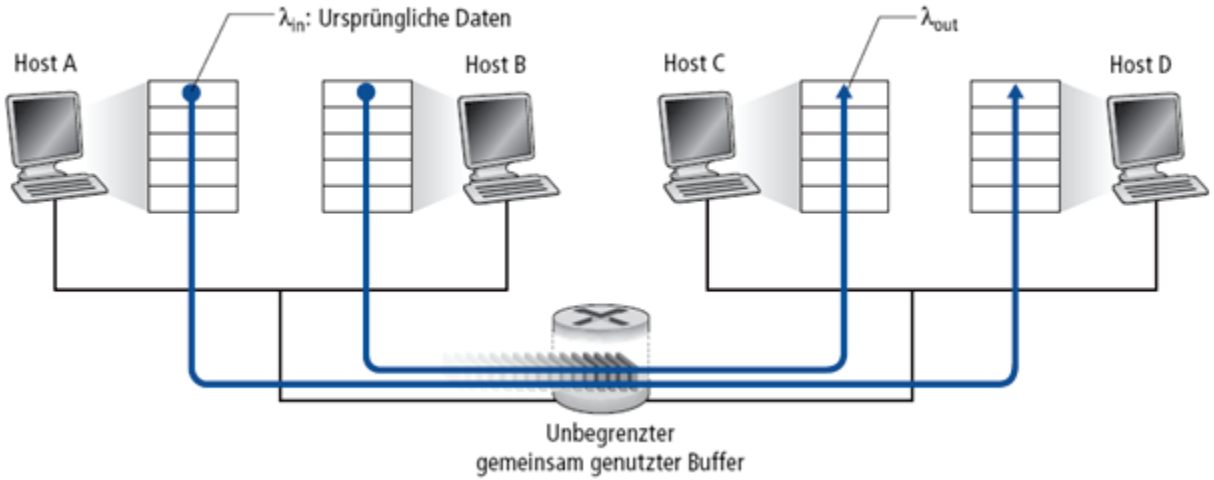
- Informell: „Zu viele Systeme senden zu viele Daten, das Netzwerk kann nicht mehr alles transportieren.“
- Nicht zu verwechseln mit Flusskontrolle!
- Erkennungsmerkmale:
  - Paketverluste (Pufferüberlauf in den Routern)
  - Lange Verzögerungen (Warten in den Routern)

→ Eines der zentralen Probleme in Computernetzwerken!

3.6.1 Ursachen und Kosten von Überlast

Betrachtung von 3 zunehmend komplexer werdenden Szenarien um die Ursache und die Auswirkungen von Überlast zu verdeutlichen:

Szenario 1: Zwei Quellen, ein Router mit unendlichem Puffer



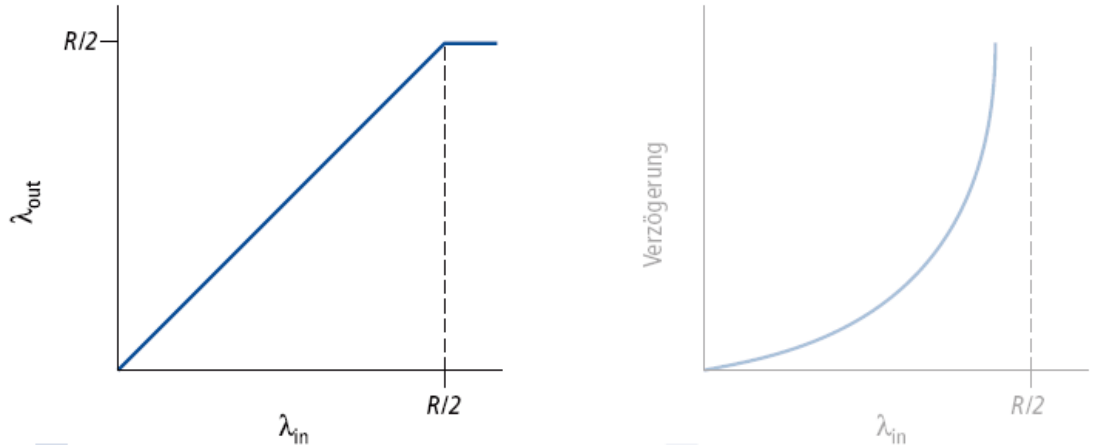
- Anwendungen auf Hosts A und B senden Daten mit der durchschnittlichen **Rate λ_{in} Byte/Sek**
- Keine Fehlerkorrektur (z.B. Übertragungswiederholungen), Flusskontrolle oder Überlastkontrolle
- Pakete der Hosts A und B gehen durch einen gemeinsamen Router und verlassen diesen über dieselbe Verbindung der **Kapazität R**

3.6.1 Ursachen und Kosten von Überlast

Szenario 1: Zwei Quellen, ein Router mit unendlichem Puffer

- Linker Graph: Durchsatz pro Verbindung (die Anzahl der Bytes pro Sekunde beim Empfänger) als Funktion der Senderate
  - Für Senderaten zwischen 0 und R/2 ist der Durchsatz beim Empfänger gleich der Senderate der Quelle = alles, was vom Sender verschickt wird, wird vom Empfänger mit begrenzter Verzögerung empfangen.
  - Bei Senderaten > R/2: der Durchsatz bleibt konstant bei R/2

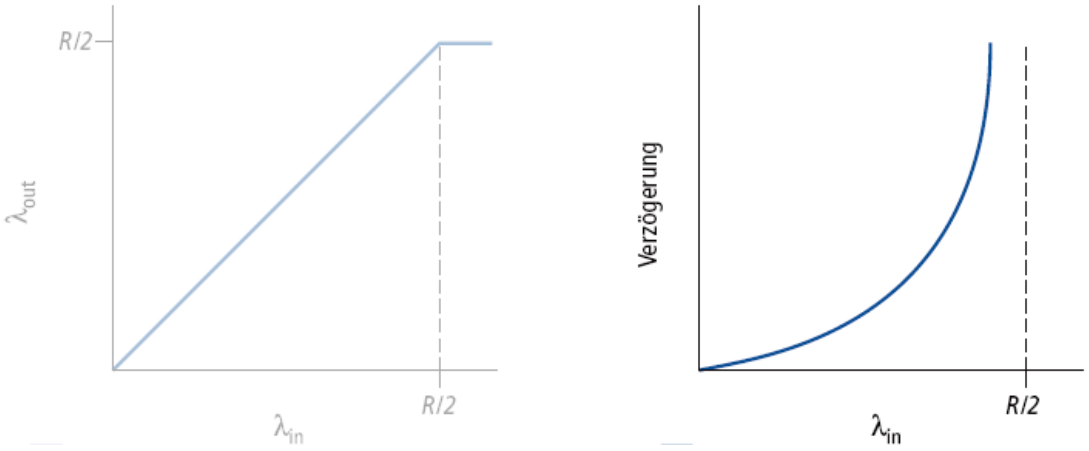
  → *Diese Obergrenze für den Durchsatz ist eine Folge des Teilens der Verbindungskapazität zwischen beiden Verbindungen!*



3.6.1 Ursachen und Kosten von Überlast

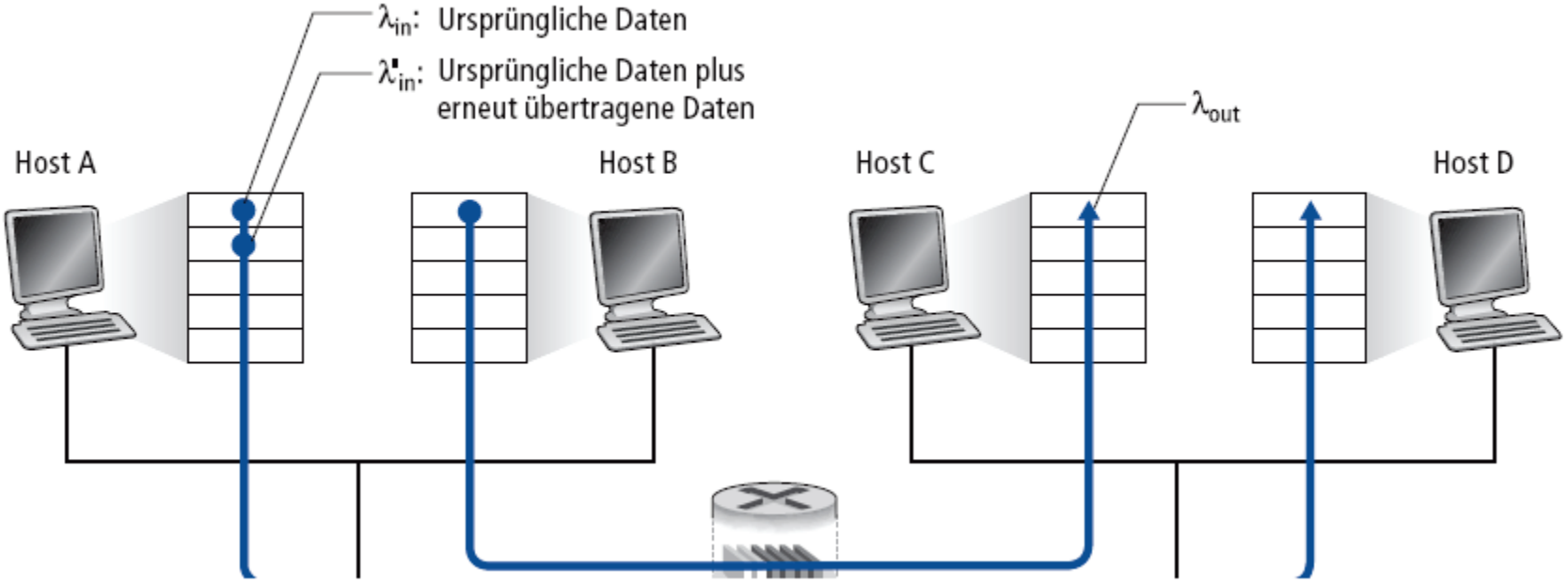
Szenario 1: Zwei Quellen, ein Router mit unendlichem Puffer

- Rechter Graph: Verzögerung als Funktion der Senderate
  - Sobald sich die Senderate R/2 annähert wird die Verzögerung immer größer
  - Bei Senderaten > R/2: durchschnittliche Anzahl der Pakete in der Warteschlange des Routers ist unbegrenzt und die durchschnittliche Verzögerung zwischen Quelle und Ziel wird unendlich



3.6.1 Ursachen und Kosten von Überlast

Szenario 2: Zwei Quellen und ein Router mit beschränktem Puffer



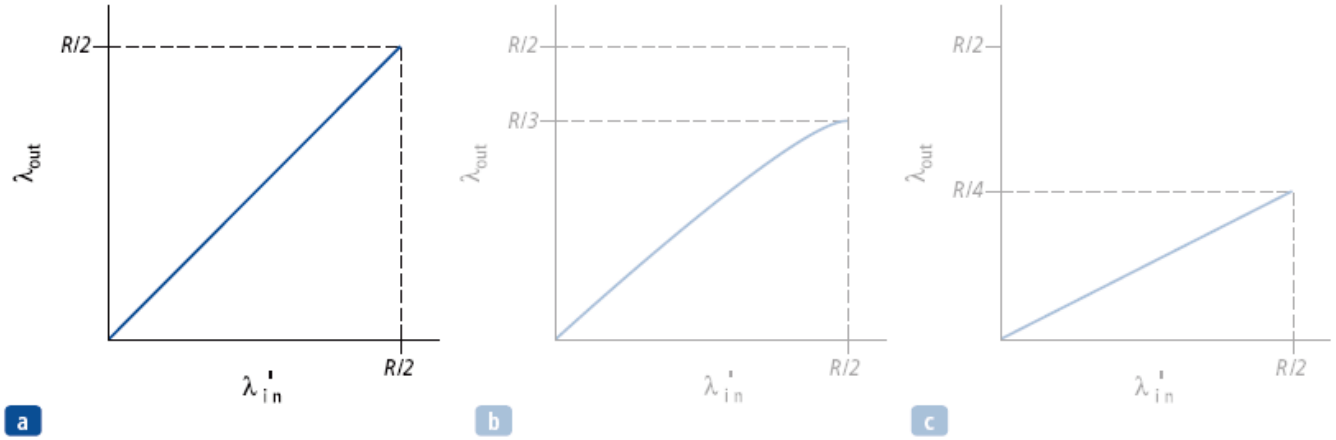
- Routerpuffer begrenzt, d.h. bei vollem Puffer werden weitere Pakete verworfen
- Übertragungswiederholung des Senders, wenn vom Router ein Paket verworfen wurde, das ein Transportschichtsegment enthielt
- **λ_{in} Byte/Sek** ist die Rate mit der eine Anwendung Daten in den Socket sendet. Die Rate mit der die Transportschicht Segmente ins Netz sendet (Originaldaten + erneut übertragene Daten) beträgt **λ'_{in} Byte/Sek** (= *offered load*).

3.6.1 Ursachen und Kosten von Überlast

Szenario 2: Zwei Quellen und ein Router mit beschränktem Puffer

- Graph **a**: Stellt den unrealistischen Fall dar, dass der Host A nur dann Pakete sendet, wenn Puffer im Router frei ist.
  - Kein Verlust tritt auf: $\lambda_{in} = \lambda'_{in}$ und der Durchsatz der Verbindung ist gleich λ_{in}.
  - Die Senderate des durchschnittlichen Hosts kann R/2 nicht überschreiten, da vorausgesetzt wird, dass Paketverlust nie stattfindet.

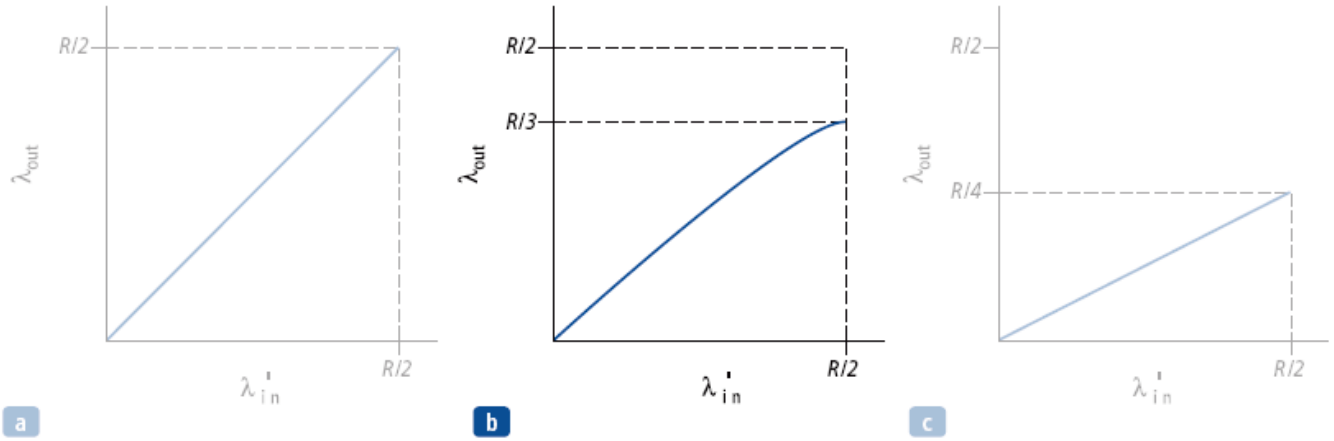
  → *Vom Standpunkt des Durchsatzes ist die Leistung ideal: Alles was verschickt wird, wird auch empfangen.*



3.6.1 Ursachen und Kosten von Überlast

Szenario 2: Zwei Quellen und ein Router mit beschränktem Puffer

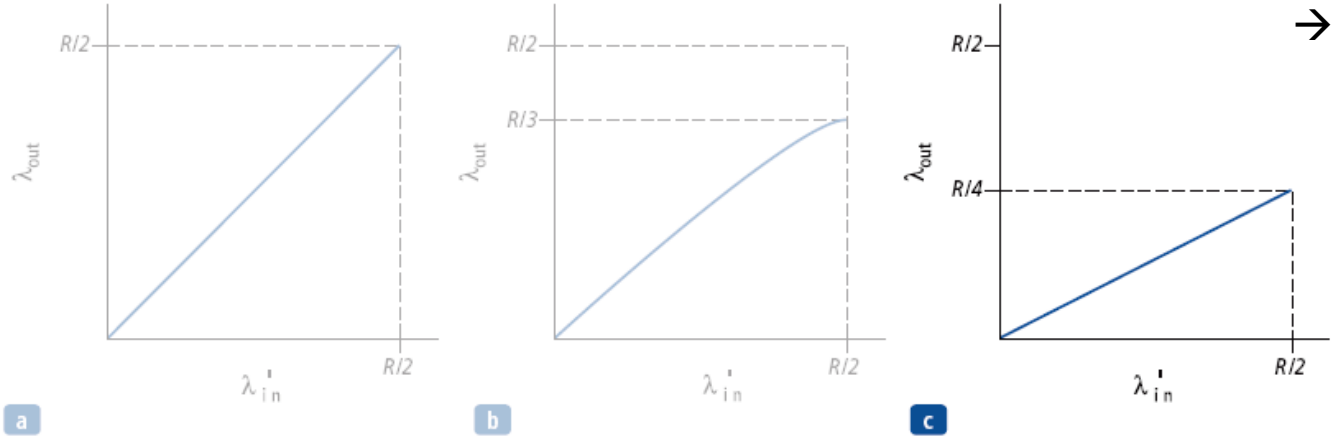
- Graph **b**: Stellt den etwas realistischeren Fall dar, dass der Sender nur dann die Übertragung wiederholt, wenn er sicher weiß, dass ein Paket verloren gegangen ist.
  - Die angebotene Last beträgt λ'_{in}= R/2, und die Rate λ_{out} , mit der die Daten an die empfangende Anwendung übermittelt werden, beträgt R/3.
    - ➔ Daher enthalten die 0,5*R gesendeten Daten durchschnittlich: 0,333*R Byte/Sek an Originaldaten und 0,166*R Byte/Sek an wiederholt übertragenen Daten.
  - → *Kostenfaktor eines überlasteten Netzwerkes: Wiederholtes Übertragen wegen Pufferüberläufen / zu hohen Latenzzeiten*



3.6.1 Ursachen und Kosten von Überlast

Szenario 2: Zwei Quellen und ein Router mit beschränktem Puffer

- Graph **c**: Stellt den realistischen Fall dar, dass beim Sender vorzeitig Timer auslaufen und so Übertragungswiederholungen für Pakete auftreten können die zwar in der Warteschlange verzögert wurden aber nicht wirklich verloren gegangen sind.
  - Wenn sowohl Originaldatenpaket als auch Übertragungswiederholung den Empfänger erreichen, verwirft dieser die Übertragungswiederholung. Der Router hätte die Übertragungskapazität der Verbindung besser verwendet, um stattdessen ein anderes Paket weiterzuleiten.
  - Wenn von jedem Paket angenommen wird, dass es im Durchschnitt zwei Mal vom Router weitergeleitet wird, hat der Durchsatz einen asymptotischen Wert von R/4, während die angebotene Last gegen R/2 geht.

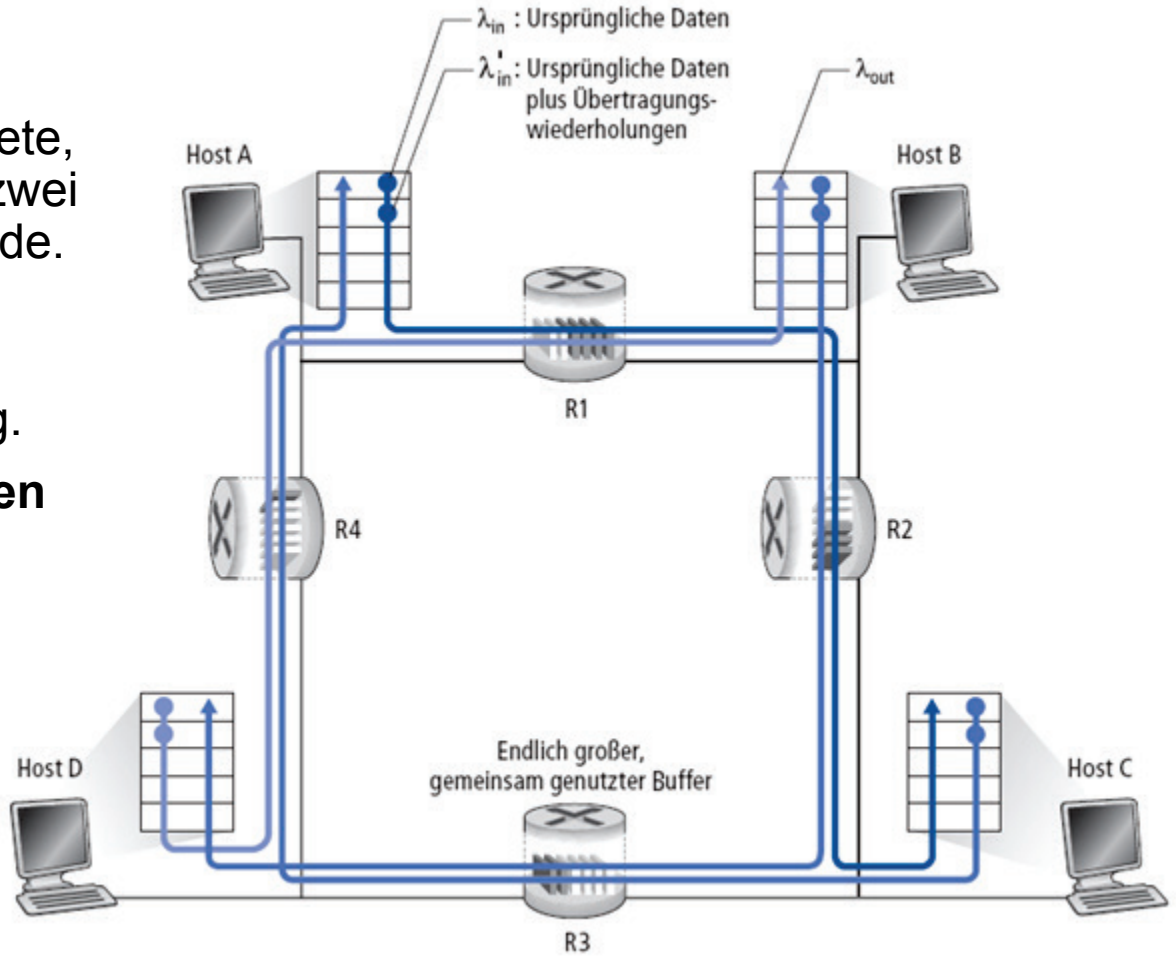


→ **Kostenfaktor:**
Unnötige Übertragungs-wiederholungen bei großen Verzögerungen bewirken, dass ein Router seine Verbindungsbandbreite zum Weiterleiten unnötiger Kopien eines Paketes verschwendet.

3.6.1 Ursachen und Kosten von Überlast

Szenario 3: Vier Quellen, Router mit beschränktem Puffer und Multihop-Pfade

- Vier Hosts übertragen Pakete, jeder über überlappende, zwei Router durchquerende Pfade.
- Zuverlässiger Datentransferdienst durch Übertragungswiederholung.
- Alle Hosts haben **denselben Wert für λ_{in}**.
- Alle Leitungen haben die **Kapazität R Byte/Sek**.

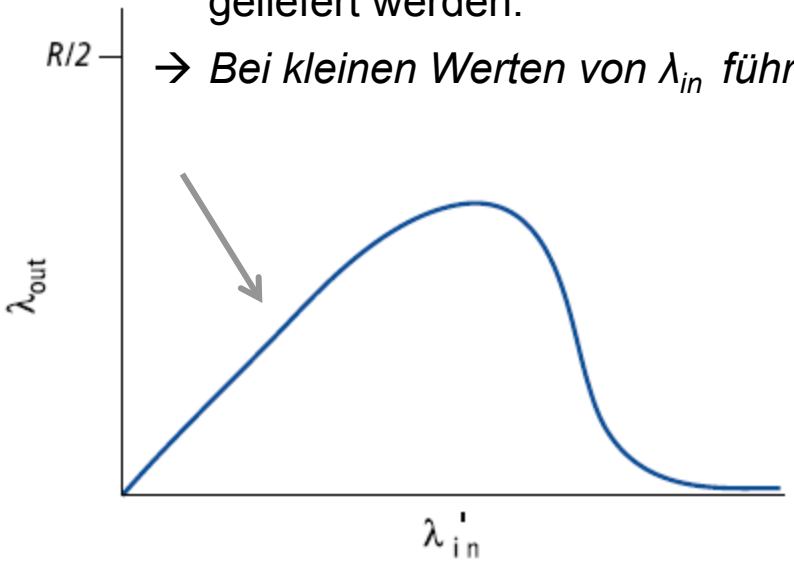


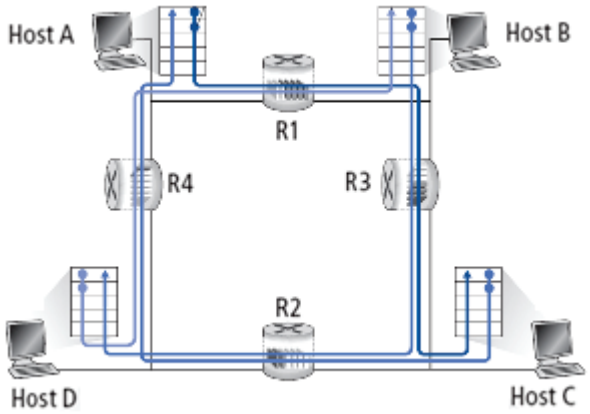
3.6.1 Ursachen und Kosten von Überlast

Szenario 3: Vier Quellen, Router mit beschränktem Puffer und Multihop-Pfade

- In unserem Beispiel teilt sich die Verbindung zwischen den Hosts A und C den verfügbaren Pufferplatz bei Router R1 mit der Verbindung D-B und bei Router R2 mit der Verbindung B-D.
- Im Fall äußerst geringen Verkehrsaufkommens:
  - Für äußerst kleine Werte λ_{in}: Pufferüberläufe sind wieder selten und der Durchsatz ist gleich der angebotenen Last.
  - Für geringfügig größere Werte λ_{in} : Pufferüberläufe immer noch selten, aber der Durchsatz ist größer, da weitere Originaldaten ins Netz gesendet und zum Zielort geliefert werden.

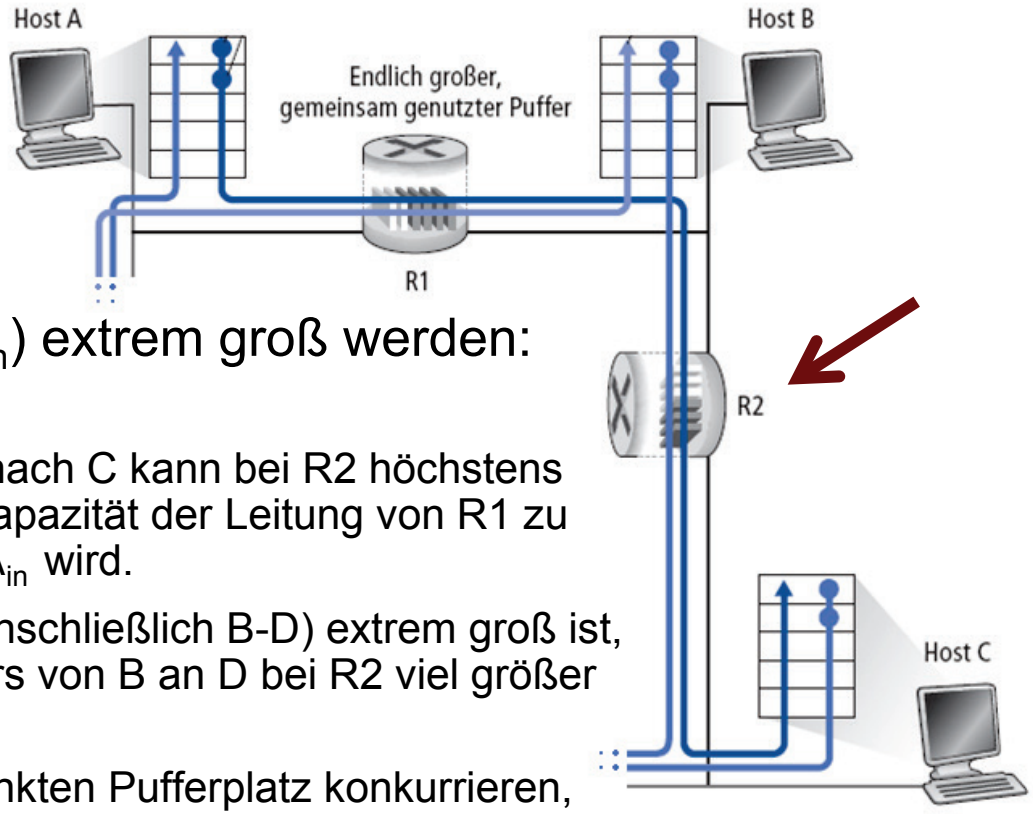
→ *Bei kleinen Werten von λ_{in} führt eine Zunahme von λ_{in} zu einer Zunahme von λ_{out}.*





3.6.1 Ursachen und Kosten von Überlast

Szenario 3: Vier Quellen, Router mit beschränktem Puffer und Multihop-Pfade



Fall in dem λ_{in} (und daher auch λ'_{in}) extrem groß werden:

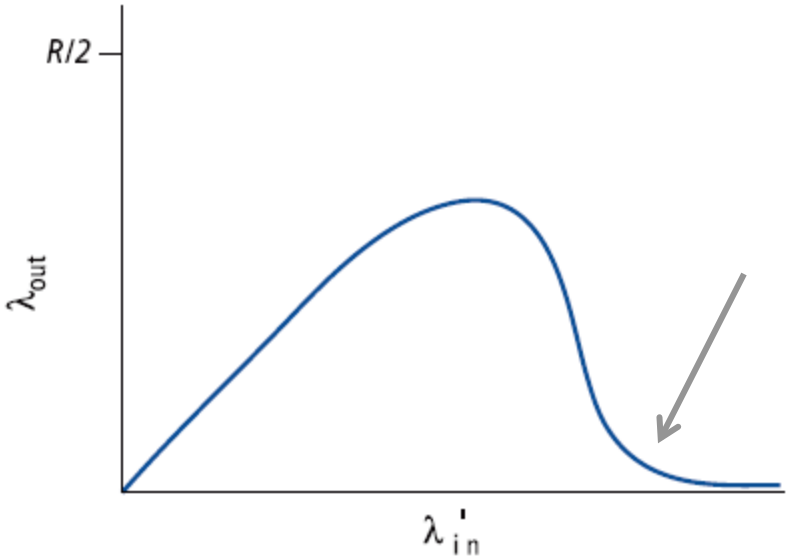
- Bsp. Router R2:
  - Der ankommende Verkehr von A nach C kann bei R2 höchstens eine Ankunftsrate von R haben (Kapazität der Leitung von R1 zu R2), unabhängig davon wie hoch λ_{in} wird.
  - Wenn λ_{in} für alle Verbindungen (einschließlich B-D) extrem groß ist, kann die Ankunftsrate des Verkehrs von B an D bei R2 viel größer sein als diejenige von A zu C.
  - Weil beide an R2 um den beschränkten Pufferplatz konkurrieren, verringert sich der erfolgreich weitergeleitete A-C-Verkehr (der also nicht durch Pufferüberlauf verloren geht) in dem Maße, in dem λ'_{in} von B-D immer größer wird.
  - Wenn λ'_{in} gegen unendlich geht: Ein leerer Puffer an R2 wird nun sofort von einem B-D Paket gefüllt und der Durchsatz der Verbindung A-C an R2 geht gegen Null.

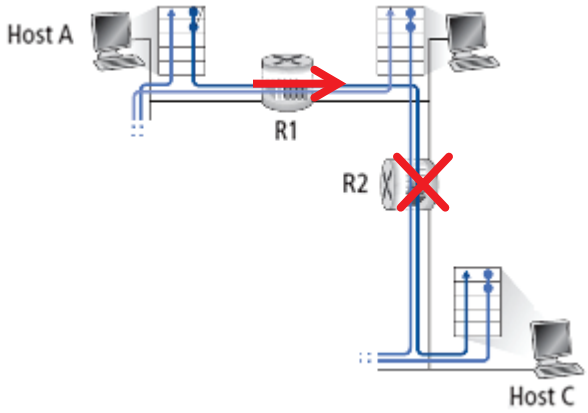
3.6.1 Ursachen und Kosten von Überlast

Szenario 3: Vier Quellen, Router mit beschränktem Puffer und Multihop-Pfade

- Bei hohem Verkehrsaufkommen kann also der Fall auftreten, dass ein Paket, das von R1 weitergeleitet wurde, von R2 verworfen wird. In dieser Situation ist die von R1 beim Weiterleiten geleistete Arbeit vergeudet.
- Wenn R1 die Übertragungskapazität, die für diese Pakete aufgewendet wurde, statt dessen für ein anderes Paket genutzt hätte wäre dies profitabler gewesen!

→ **Kostenfaktor beim Verwerfen eines Paketes aufgrund von Überlast: Wird ein Paket verworfen, ist die Übertragungskapazität verschwendet, die auf früheren Leitungen dafür benötigt wurde.**

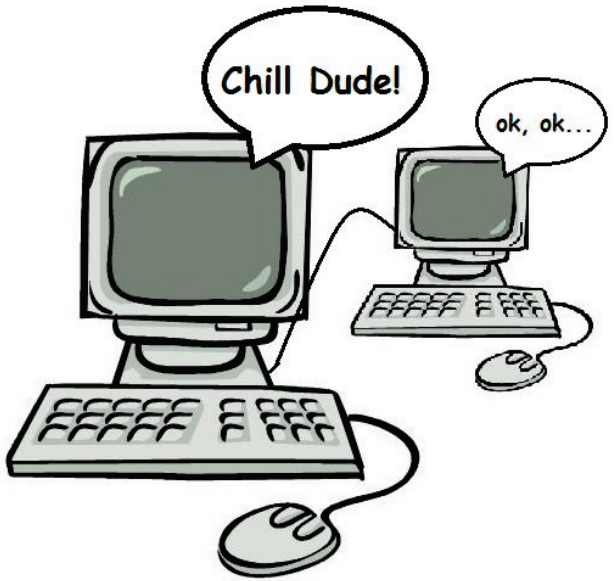




3.6.2 Ansätze zur Überlastkontrolle

Ende-zu-Ende-Überlastkontrolle:

- Keine explizite Unterstützung durch das Netzwerk
- Überlast wird von den Endsystemen durch Paketverlust und erhöhte Verzögerung festgestellt
- Dies ist das Vorgehen von TCP im Internet



3.6.2 Ansätze zur Überlastkontrolle

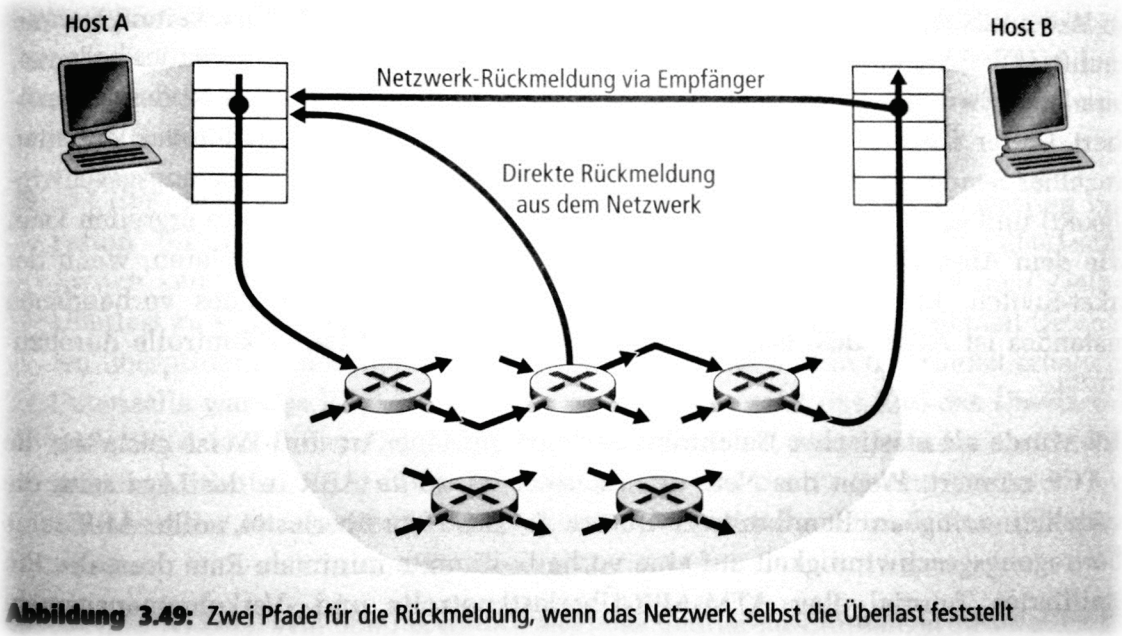


Abbildung 3.49: Zwei Pfade für die Rückmeldung, wenn das Netzwerk selbst die Überlast feststellt

Netzwerkunterstützte Überlastkontrolle:

- Router geben den Endsystemen Hinweise. Zwei Möglichkeiten:
  - Direkte Rückmeldung vom Router an den Sender (z.B. in Form eines Choke-Pakets)
  - Durch ein einzelnes Bit, welches im Paketheader von den Routern gesetzt werden kann (SNA, DECbit, TCP/IP ECN, ATM). Beim Empfang eines derart markierten Pakets informiert der Empfänger den Sender über die Überlast.

3.7 TCP-Überlastkontrolle

3.7 TCP-Überlastkontrolle

TCP muss statt netzwerkunterstützter Überlastkontrolle eine Ende-zu-Ende Überlastkontrolle verwenden, da die IP-Schicht den Endsystemen hinsichtlich der aktuellen Netzlast keine explizite Rückmeldung liefert!

→ Also stellt jeder Sender bei TCP seine Senderate als Funktion der wahrgenommenen Überlast selbst ein.

Das wirft folgende Fragen auf:

1. *Wie begrenzt ein TCP Sender die Rate, mit der er Daten über seine Verbindung sendet?*
2. *Wie erkennt ein TCP-Sender, dass es Überlast auf dem Pfad zwischen sich und dem Sender gibt?*
3. *Welchen Algorithmus sollte der Absender verwenden um sein Sendetempo als Funktion der erkannten Überlast zwischen den Endpunkten zu ändern?*

3.7 TCP-Überlastkontrolle

1. *Wie begrenzt ein TCP Sender die Rate, mit der er Daten über seine Verbindung sendet?*
   - Jede Seite einer TCP-Verbindung besteht aus einem Eingangspuffer, einem Sendepuffer, mehreren Variablen (LastByteRead, RcvWindow usw.) und der Variable **CongWin** (Congestion Window → Überlastfenster).

2. *Wie erkennt ein TCP-Sender, dass es Überlast auf dem Pfad zwischen sich und dem Sender gibt?*
   - Überlast wird am TCP-Sender auf zwei mögliche Arten wahrgenommen: durch das Auftreten eines Timeouts oder durch den Erhalt von drei doppelten ACKs.
   - Tritt kein Verlustereignis auf, empfängt der TCP-Sender ACKs für seine zuvor unbestätigten Segmente und vergrößert sein CongWin (und damit die Übertragungsgeschwindigkeit).

3.7 TCP-Überlastkontrolle

3. *Welchen Algorithmus sollte der Absender verwenden um sein Sendetempo als Funktion der erkannten Überlast zwischen den Endpunkten zu ändern?*

→ Der **TCP-Überlastkontroll-Algorithmus** besteht aus drei Hauptbestandteilen:

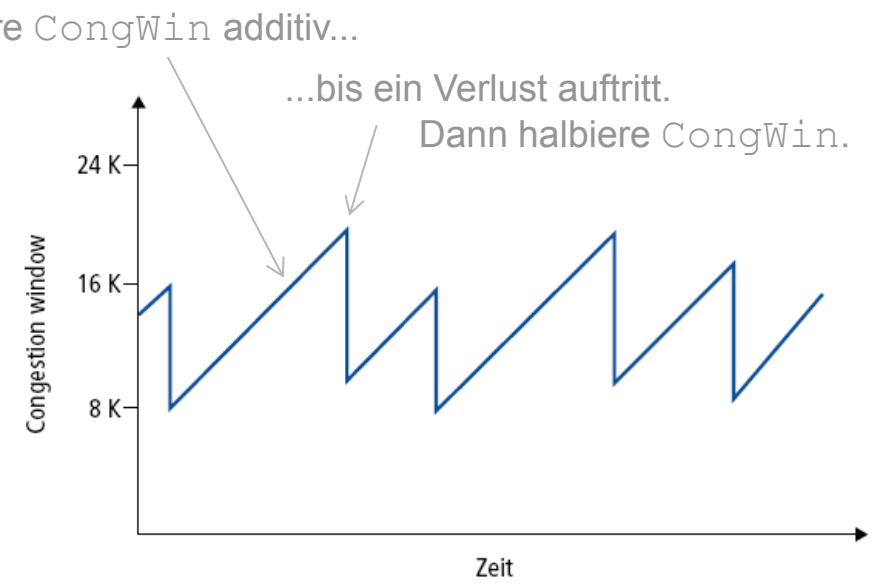
- Additive-Increase, Multiplicative-Decrease
- Slow Start und
- Reaktion auf Timeout-Ereignisse

3.7 TCP-Überlastkontrolle

Additive-Increase, Multiplicative-Decrease:

Ansatz: Erhöhe die Übertragungsrate (Fenstergröße), um nach überschüssiger Bandbreite zu suchen, bis ein Verlust eintritt

- **Additive Increase**: Erhöhe `CongWin` um eine MSS pro RTT, bis Verlust erkannt wird
- **Multiplicative Decrease**: Halbiere `CongWin`, wenn ein Verlust erkannt wird



3.7 TCP-Überlastkontrolle

Slow Start (ein **wenig** "langsamer" Anfang)

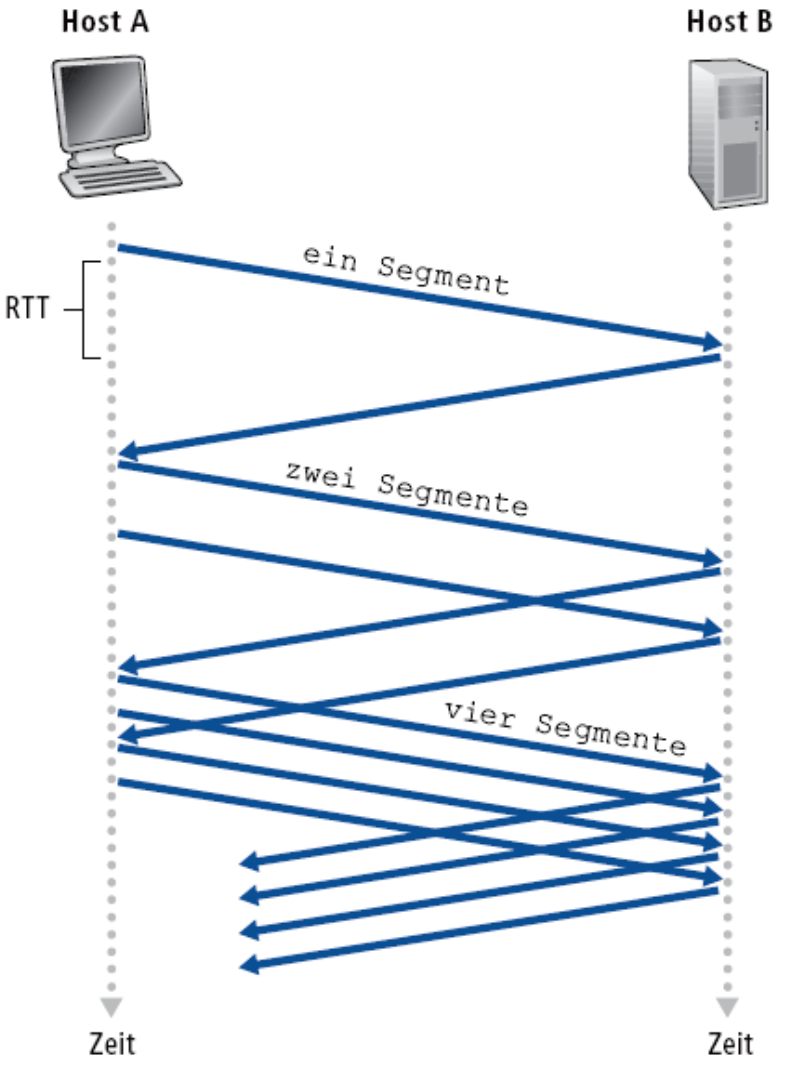
- Bei Verbindungsbeginn:
  `CongWin` = 1 MSS (Maximum Segment Size)

  Beispiel: MSS = 500 Byte & RTT = 200 ms
  - Initiale Rate = 20 kBit/s

  → Verfügbare Bandbreite kann aber viel größer als MSS/RTT sein!

- Die Rate sollte sich schnell der verfügbaren Rate anpassen, deshalb bei Verbindungsbeginn: Erhöhe die Rate exponentiell schnell bis zum ersten Verlustereignis
  - Angestrebt: Verdoppeln von **`CongWin`** in jeder RTT
  - Realisiert durch: **`CongWin`** um 1 für jedes erhaltene ACK erhöhen

- Ergebnis: *Initiale Rate ist gering, wächst aber exponentiell schnell!*



3.7 TCP-Überlastkontrolle

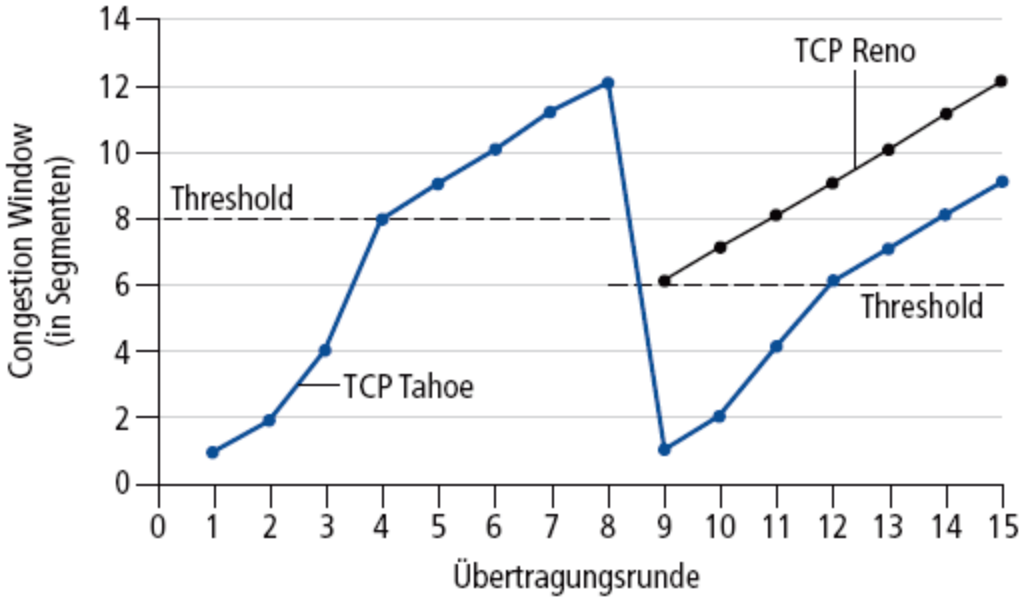
Slow Start

Wann soll vom exponentiellen zum linearen Wachstum übergegangen werden?

Wenn **`CongWin`** die Hälfte des Wertes erreicht, den es vor dem Verlustereignis hatte.

Implementierung:

- Variabler `Threshold`
- Bei einem Verlustereignis wird `Threshold` auf die Hälfte des Wertes gesetzt, den es vor dem Verlustereignis hatte.



3.7 TCP-Überlastkontrolle

Reaktion auf Timeouts

TCP-Überlastkontrolle reagiert unterschiedlich auf die zwei Arten der Verlustereignisse:

- Nach drei doppelten ACKs:
  - **`CongWin`** halbieren ("Fast Recovery"), `Threshold` auf diesen Wert setzen
  - Fenster wächst danach linear

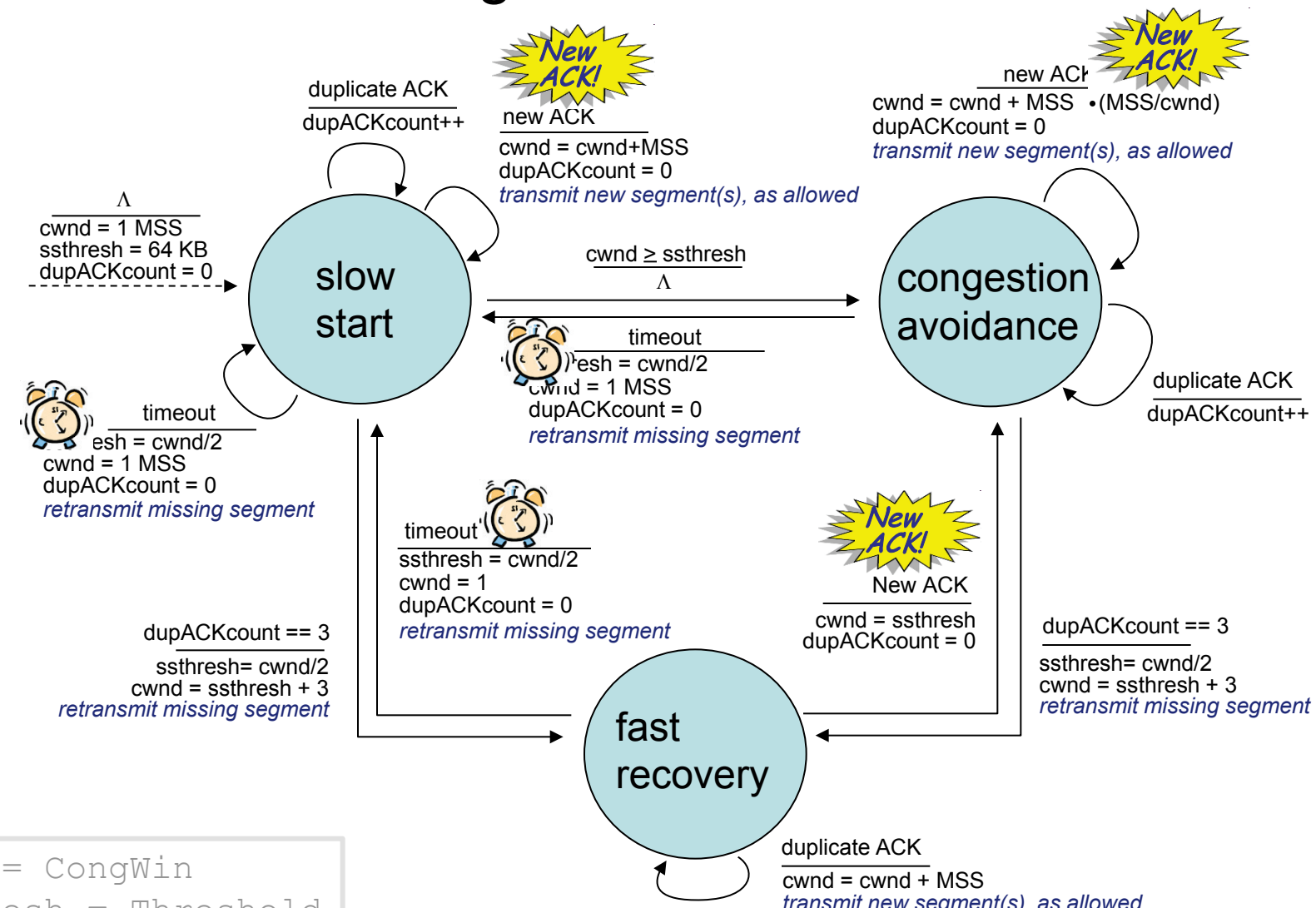
- Nach Timeout:
  - **`CongWin`** auf 1 MSS setzen, `Threshold` auf die Hälfte des alten Werts von **`CongWin`** setzen
  - Fenster wächst dann exponentiell, bis `Threshold` erreicht ist
  - Fenster wächst danach linear

→ Grund für unterschiedliche Reaktion: Drei doppelte ACKs zeigen an, dass das Netzwerk noch in der Lage ist, Pakete auszuliefern. Bei Timeout erfolgt gar keine Rückmeldung mehr.

3.7 Zusammenfassung TCP-Überlastkontrolle

Slow-Start-Phase: **`CongWin`** kleiner als `Threshold`	Das Fenster wächst exponentiell.
Congestion-Avoidance-Phase: **`CongWin`** größer als `Threshold`	Das Fenster wächst linear.
Bei **drei doppelten ACKs**:	`Threshold` wird auf **`CongWin/2`** gesetzt und daraufhin **`CongWin`** auf `Threshold`.
Bei einem **Timeout**:	`Threshold` wird auf **`CongWin/2`** gesetzt und daraufhin **`CongWin`** auf 1 MSS.

3.7 Zusammenfassung TCP-Überlastkontrolle




```
cwnd = CongWin
ssthresh = Threshold
```

3.7 Zusammenfassung TCP-Überlastkontrolle

Zustand	Ereignis	Reaktion der TCP-Überlastkontrolle	Kommentar
Slow Start (SS)	ACK für zuvor unbestätigte Daten empfangen	*CongWin* = *CongWin* + *MSS*, Wenn (*CongWin* > Threshold), setze Zustand auf „Congestion Avoidance"	Führt zu einer Verdopplung von *CongWin* in jeder *RTT*.
Congestion Avoidance (CA)	ACK für zuvor unbestätigte Daten empfangen	*CongWin* = *CongWin* + *MSS* · (*MSS*/*CongWin*)	Additive Increase, resultiert in einer Zunahme von *CongWin* um 1 *MSS* jede *RTT*.
SS oder CA	Verlustereignis entdeckt durch drei doppelte ACKs	Threshold = *CongWin*/2, *CongWin* = Threshold, setze Zustand auf „Congestion Avoidance"	Fast Recovery, implementiert Multiplicative Decrease. *CongWin* kann nicht unter 1 *MSS* fallen.
SS oder CA	Timeout	Threshold = *CongWin*/2, *CongWin* = 1 *MSS*, setze Zustand auf „Slow Start"	Erneute Slow-Start-Phase
SS oder CA	Doppeltes ACK empfangen	Erhöhe den Zähler für doppelte ACKs für das bestätigte Segment	*CongWin* und Threshold werden nicht verändert.

Die dargestellten Zustandswerte sind die Zustände des TCP-Senders unmittelbar bevor die beschriebenen Ereignisse eintreten.

3.7 TCP Durchsatzanalyse

Frage nach dem durchschnittlichen TCP-Durchsatz durch Sägezahnverhalten der Senderate schwierig.

→ Wiederholte Slow-Start-Phasen nach Timeouts werden im Folgenden vernachlässigt, da normalerweise sehr kurz.

- Sei W die Fenstergröße, wenn das Verlustereignis eintritt
- Wenn das Fenster W groß ist, dann ist der Durchsatz $\frac{W}{RTT}$
- Direkt nach dem Verlust verringert sich das Fenster auf $\frac{W}{2}$ und damit der Durchsatz auf $\frac{1}{2}\frac{W}{RTT}$

→ Durchschnittlicher Durchsatz ist also: $\frac{3}{4}\frac{W}{RTT}$

3.7 Die Zukunft von TCP

Die TCP-Überlastkontrolle hat sich im Laufe vieler Jahre entwickelt. Die Anforderungen ändern sich, was am folgenden Beispiel verdeutlicht werden soll:

Beispiel:

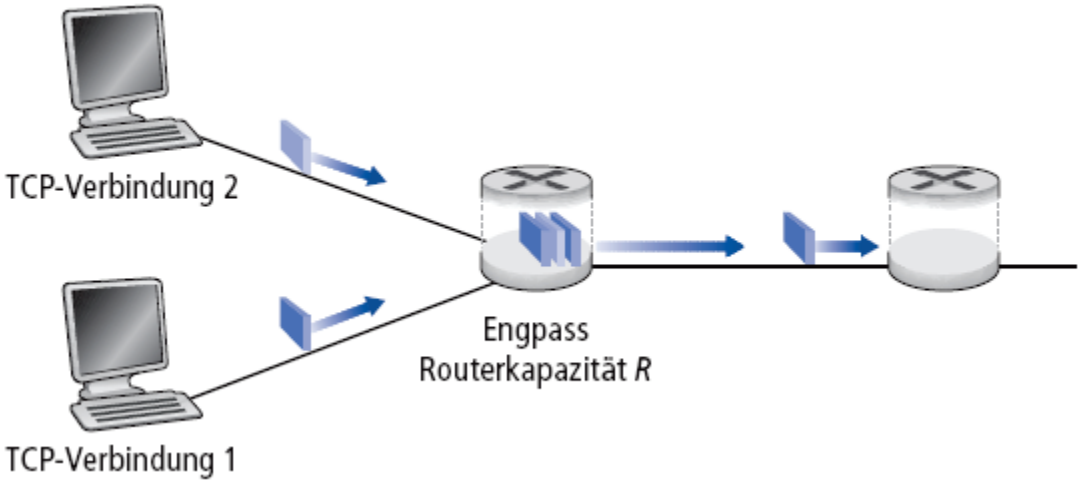
- 1500-Byte-Segmente, 100ms RTT, wir wollen 10 GBit/s als Durchsatz
- Erfordert Fenstergröße W = 83,333
- Formel für den Durchsatz in Abhängigkeit von der Verlustrate:

$$\frac{1.22 \cdot MSS}{RTT\sqrt{p}}$$

→ $p = 2 \cdot 10^{-10}$ ***= eine sehr kleine Verlustrate!***

→ Erfordert neue Versionen von TCP für Hochgeschwindigkeitsnetze!

3.7.1 Fairness



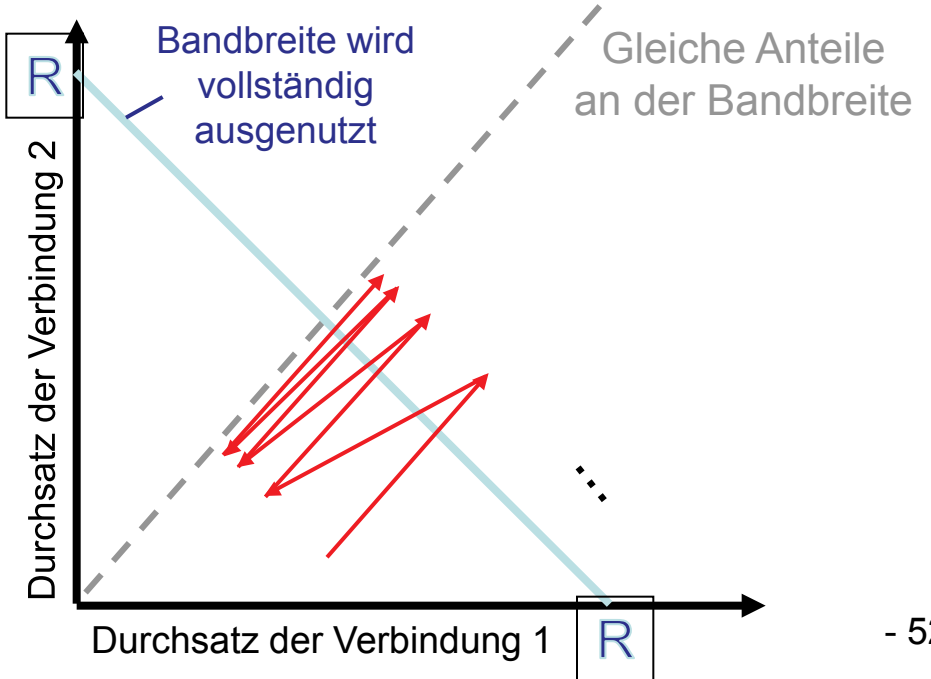
Ziel: Wenn K TCP-Sitzungen sich denselben Engpass mit Bandbreite R teilen, dann sollte jede Sitzung eine durchschnittliche Rate von R/K erhalten (die RTT-Abhängigkeit wird hierbei einfachheitshalber nicht beachtet)

3.7.1 Fairness von TCP

Warum ist TCP fair?

<u>Beispiel:</u> Zwei Verbindungen im Wettbewerb

- Additive Increase führt zu einer Steigung von 1, wenn der Durchsatz wächst
- Multiplicative Decrease reduziert den Durchsatz proportional



3.7.1 Fairness

Fairness und UDP

- Viele Multimediaanwendungen verwenden TCP nicht, um zu vermeiden, dass die Rate durch Überlastkontrolle reduziert wird
- Stattdessen Einsatz von UDP
  → z.B. Audio-/Videodaten mit konstanter Rate ins Netz leiten, Verlust hinnehmen
- TCP vermindert seine Übertragungsrate angesichts wachsender Überlast, UDP nimmt darauf keine Rücksicht
  → UDP-Quellen können den TCP-Verkehr möglicherweise verdrängen

- Forschungsgebiet: Überlastkontrollmechanismen für das Internet, die den UDP-Verkehr daran hindern den TCP-Verkehr zu verdrängen.

3.7.1 Fairness

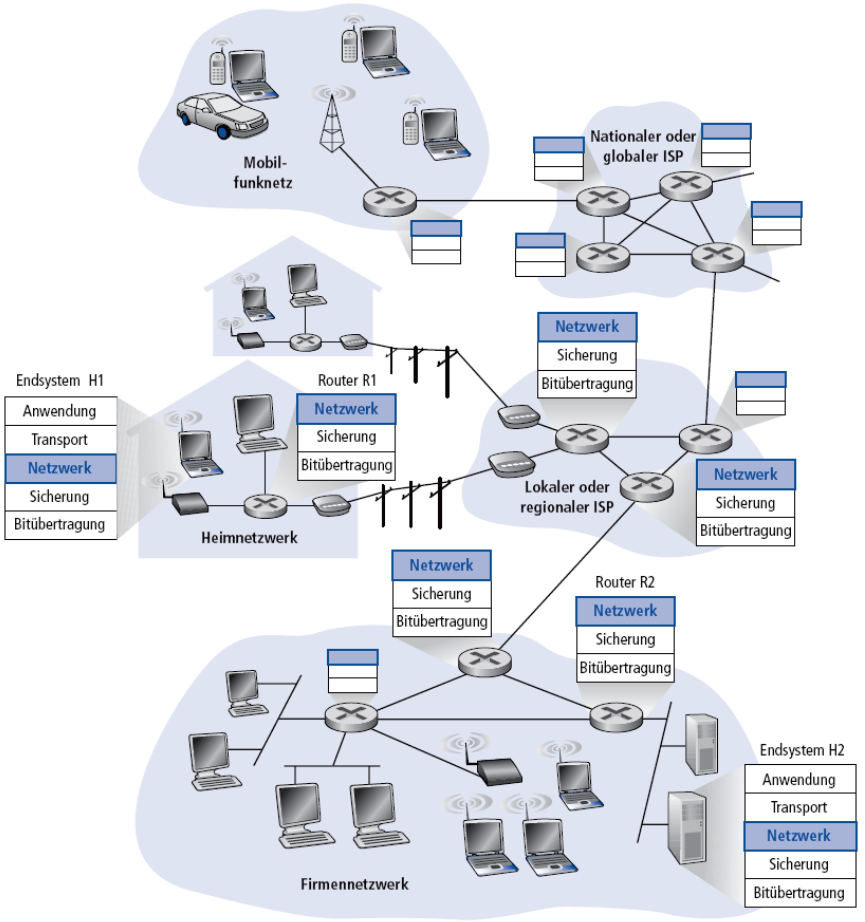
Fairness und parallele TCP-Verbindungen

- Eine Anwendung kann zwei oder mehr parallele TCP-Verbindungen öffnen
- Webbrowser machen dies häufig
- Beispiel
  Eine Leitung hat eine **Rate von R**, über die **neun Anwendungen** je eine TCP-Verbindung unterhalten.
  - Kommt eine neue Anwendung und legt **eine neue TCP-Verbindung** an, dann bekommt jede der insgesamt zehn Verbindungen ungefähr dieselbe Übertragungsgeschwindigkeit **R/10**
  - Wenn die neue Anwendung aber **elf neue TCP-Verbindungen** anlegt, dann erhält sie **mehr als R/2**!

Kapitel 4 – Netzwerkschicht

4.1 Die Netzwerkschicht

- Auch: **Vermittlungsschicht** oder **Network Layer**
- Daten von der nächsthöheren Schicht (Transportschicht) des Senders entgegennehmen
- In Datagramme verpacken
- Durch das Netzwerk leiten
- Auspacken des Vermittlungspakets beim Empfänger
- Ausliefern der Daten an die nächsthöhere Schicht (Transportschicht) des Empfängers
- Netzwerkschicht existiert in jedem Host und Router!



4.1.1 Funktionen der Netzwerkschicht

- Weiterleiten von Paketen (Forwarding):
  - Router nimmt Paket auf einer Eingangsleitung entgegen
  - Router bestimmt die Ausgangsleitung anhand lokaler Informationen (z.B. Routing-Tabelle)
  - Router legt das Paket auf die Ausgangsleitung
- Wegewahl (Routing):
  - Router kommunizieren miteinander, um geeignete Wege durch das Netzwerk zu bestimmen
  - Als Ergebnis erhalten sie Informationen, wie welches System im Netzwerk zu erreichen ist (z.B. wird eine Routingtabelle mit Einträgen gefüllt)
- Metapher:
  - Routing = Planen einer Strecke für eine Autofahrt
  - Weiterleitung = Verhalten an einer Autobahnkreuzung

4.1.1 Zusammenspiel von Routing und Forwarding