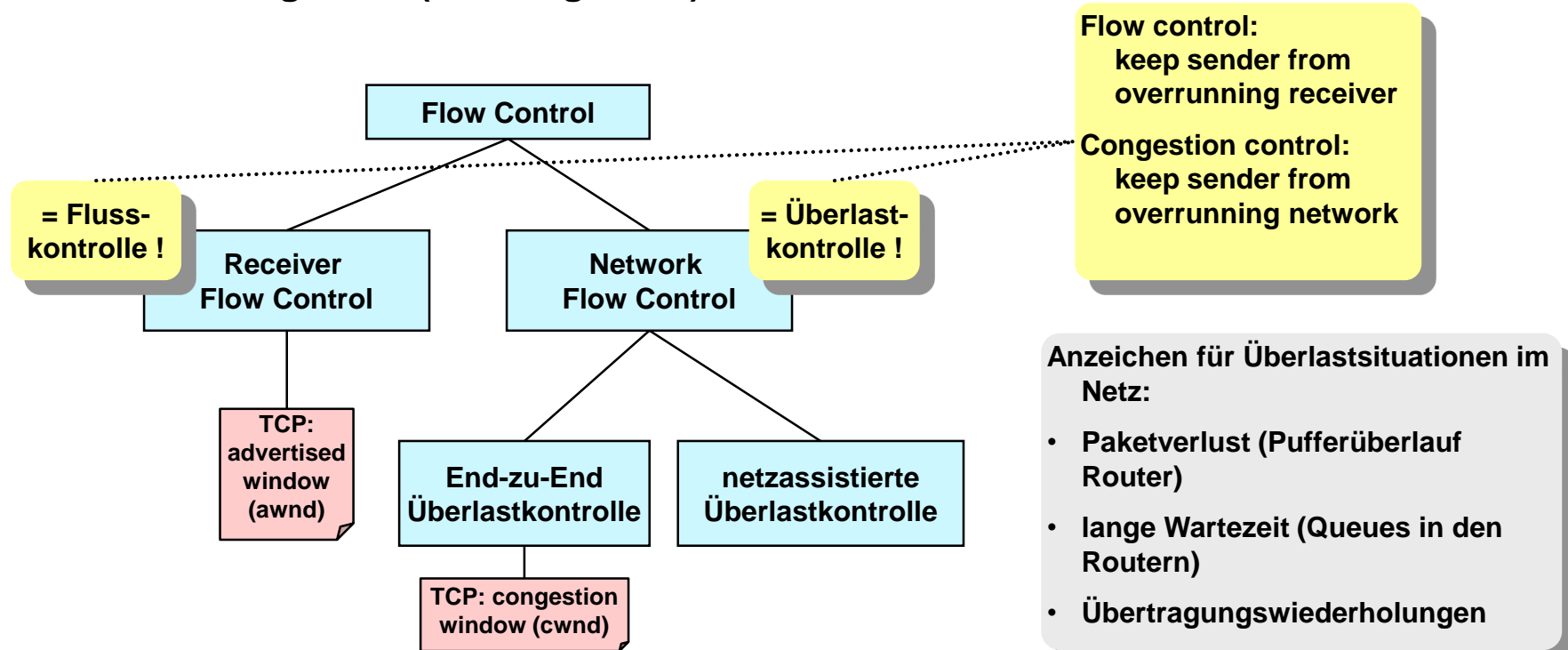


Begriffsklärung: Überlast- und Flusskontrolle

Definition (vgl. Steven Low)

Mit **Flusskontrolle** bezeichnen wir alle Methoden zur effizienten Nutzung der Bandbreite (eines verbindungslosen Netzes)

Mit **Fenster-Flusskontrolle** bezeichnen wir Flusskontrollmechanismen, die auf Fenstergrößen (=Puffergrößen) beruhen.

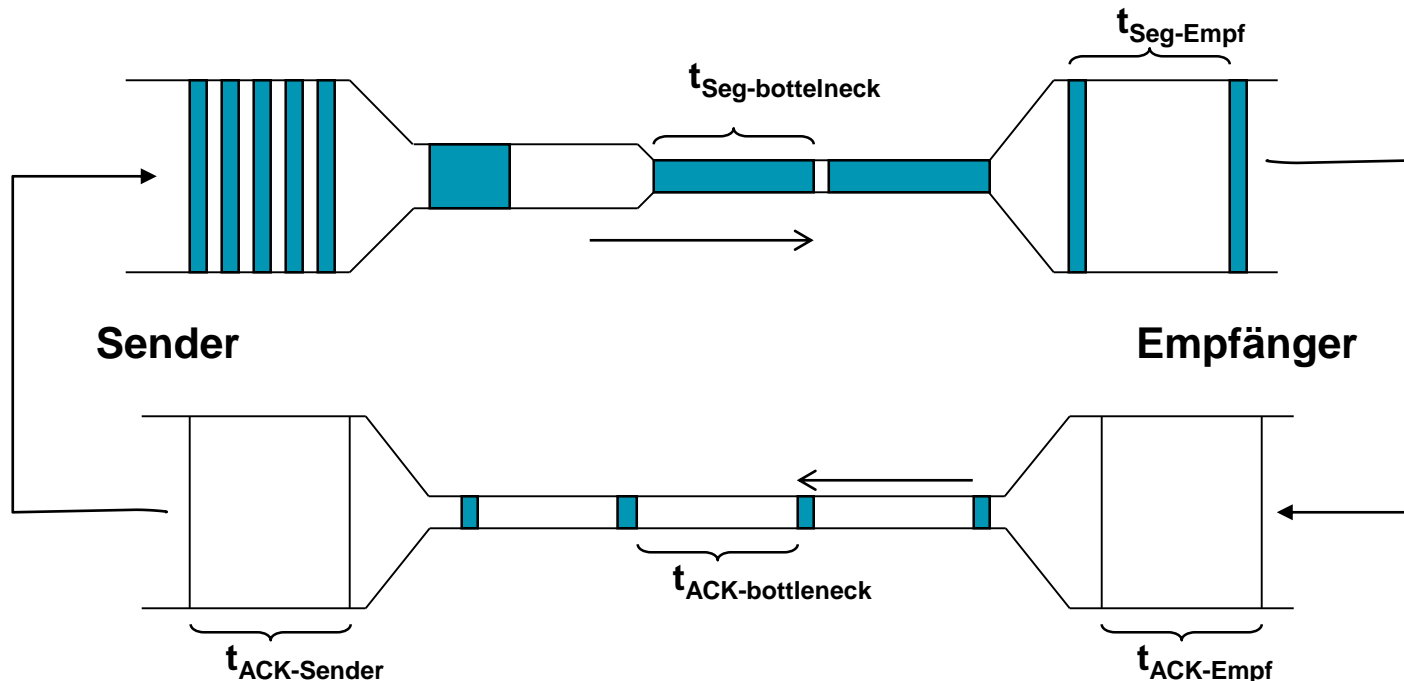


Prinzip: Self-Clocking

Equilibrium durch "Conservation of Packets" (robust gegen congestion collapse)

- Problem 1: Erreichen eines Equilibriums
- Problem 2: Schneller Übergang zu neuem Equilibrium bei Änderungen der Netzauslastung („Konvergenz“)
- Problem 3: Stabilität bei Netzstörungen und Netzverzögerungen

dynamische
Eigenschaften



**TCP
Selfclocking**
→ Anpassung
der Ü-Rate auf
langsamsten
Verbindungsteil

**Kernfrage
Congestion
Avoidance:**

**Wie wird
Equilibrium
sichergestellt?**



TCP-Überlastkontrolle

TCP versucht durch Überlastkontrolle zu erreichen:

- (1) eine hohe Nutzung des Netzes (→ hohes Equilibrium)
- (2) möglichst keine Überlast (→ stabiles Equilibrium)
- (3) gerechte Aufteilen der verfügbaren Bandbreite (→ gerechtes Equilibrium)

TCP verwendet hierfür zwei Flusskontrollmechanismen:

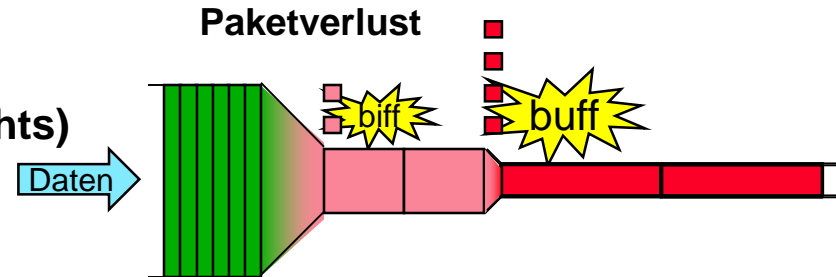
- **Receiver Flow Control**
 - Vermeidung der Überlast beim Empfänger
 - Gesteuert durch den Empfänger
 - Verwendete Fenstergröße: AdvertisedWindow (**awnd**)
- **Network Flow Control**
 - Gewinne Information über Belastbarkeit des Netzes (Verlust, Verzögerung, Hinweise) → (1)
 - Vermeide Überlastung des Netzes →(2)
 - Gesteuert durch den Sender
 - Verwendete Fenstergröße: CongestionWindow (**cwnd**)

Effektiv verwendetes Fenster: **wnd** = $\min(\text{awnd}, \text{cwnd})$

Kritische Phasen, Equilibrium und Algorithmen

Start der TCP-Übertragung

- **Slow Start** (Aufbau eines Gleichgewichts)



Erhalt des Gleichgewichts

- Einstellung des RTOs (**Jacobson/Karel Algorithmus**)
- Maximierung der Auslastung (→ **Additive Increase**)

Reaktion auf Überlastungsanzeichen

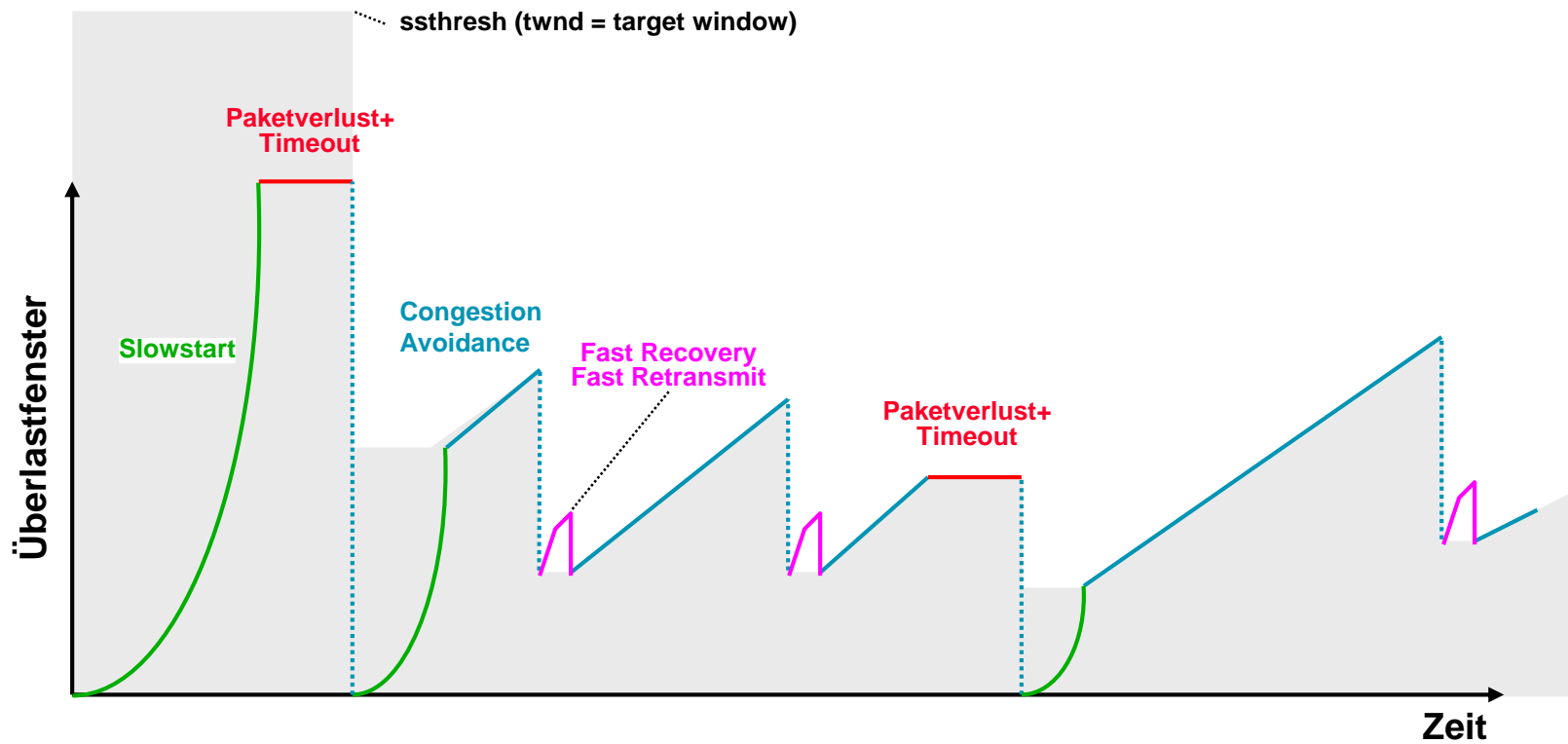
- Stabilisierung der Situation (→ **Multiplicative Decrease, Karn-Algo.**)
- Sicherstellung einer kontinuierlichen, unterbrechungsfreien Datenflusses (→ **Fast Retransmit, →Fast Recovery,**)

Fairness der Ressourcenverteilung

- automatisch (?) durch **AIMD** (= **Additive Increase / Multiplicative Decrease**)



TCP Sägezahnverhalten





Slow Start

Ziel von Slow Start:

- **Rasches Heranführen der TCP-Verbindung an das Equilibrium**

Strategie:

- **Progressives Testen der Belastbarkeit der Strecke Sender-Empfänger (bis in die Überlast hinein)**
- **Paketverluste (→ Retransmission Time Out) werden als Kennzeichen für eine Überlastsituation gewertet.
(Dies ist kritisch für mobile TCP → niedrige Übertragungsrate)**
- **Beruhigung des Netzes nach dem Eintreten einer Überlastsituation (Leeren der Router-Queues am Engpass)**

Einsatz:

- **Beim Start einer Verbindung**
- **nach Ablauf des Retransmission-Timers**
- **nach langer passiver TCP-Phase**



Slow Start-Algorithmus

Slow Start

cwnd = SMSS

FOR EACH received(ACK)

cwnd := cwnd + SMSS

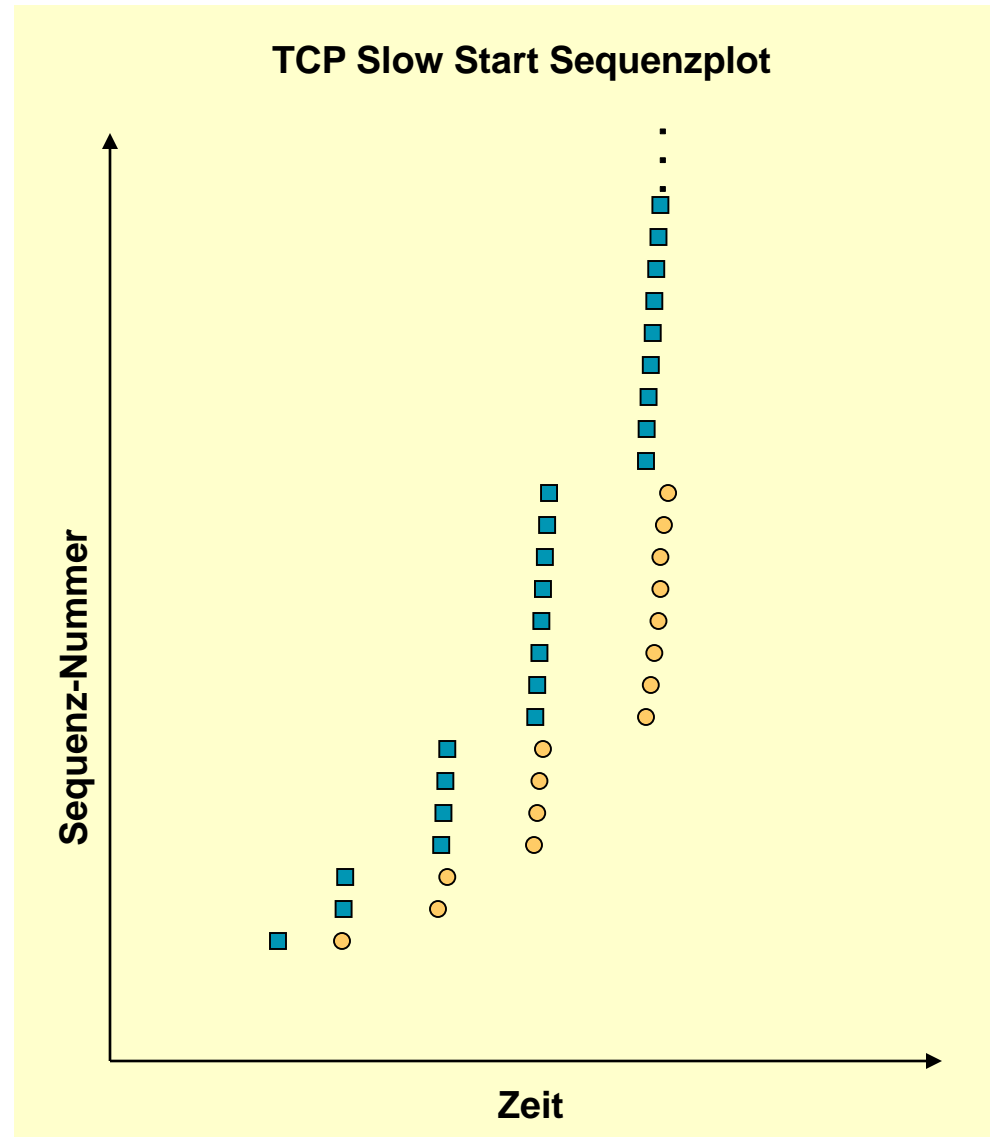
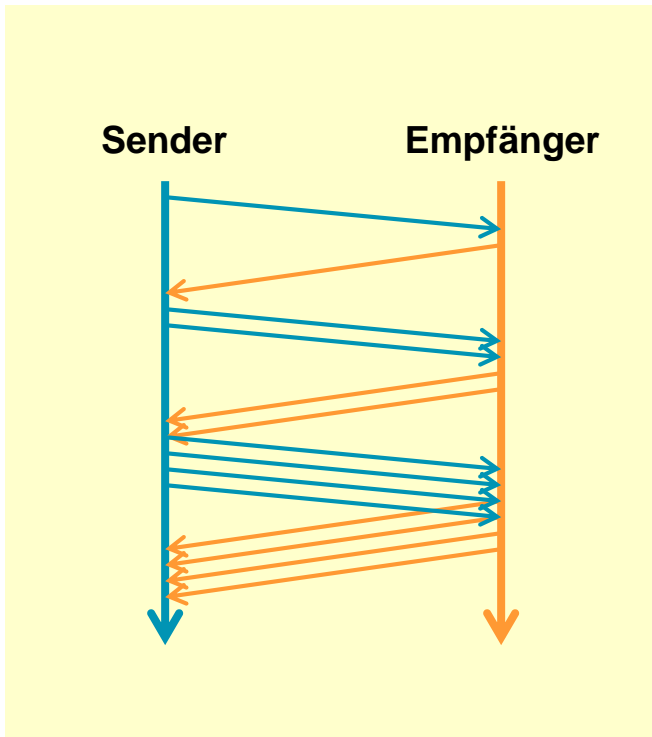
UNTIL cwnd >= twnd OR RTO

- **SMSS Sender Maximum Segment Size**
größtes Segment, das ein Sender senden kann
(536 Bytes nach RFC 1122, 1024 Byte bei den meisten Implementierungen).
- **Der Startwert für cwnd kann auf bis zu 2 SMSS gesetzt werden (nach RFC 2581)**
- **Für den ersten Slow Start kann das Zielfenster (twnd = ssthresh = slow start threshold) beliebig hoch gesetzt werden (z.B. auf awnd)**
- **(näherungsweise) Wachstum von cwnd als Funktion der Zeit gemessen in RTT:**

$$cwnd(t) = \sum_{i=0}^t 2^i = 2^{t+1} - 1$$



Veranschaulichung des Slow Start Algorithmus





Congestion Avoidance

Ziel

- **möglichst gute Ausnutzung der verfügbaren Bandbreite (greedy, gefräßig), aber Überlast möglichst vermeiden.**

Strategie

- **Kontinuierliches zeitlich lineares Vergrößern des cwnd , solange keine Überlastsituation eintritt (Addieren von 1 SMSS pro RTT)
→ Additive Increase**
- **Halbieren des cwnd, falls Überlastsituation eintritt.
Eine Überlastsituation wird z.B. nach Ablauf des Retransmissiontimers angenommen.
→ Multiplicative Decrease**
- **AIMD = Additive Increase / Multiplicative Decrease**



Congestion Avoidance-Algorithmus

Congestion Avoidance

additive increase

UNTIL LossEvent

FOR EACH received(ACK)

$wnd := wnd + SMSS * (SMSS/wnd)$

ENDFOR

ENDUNTIL

$twnd := wnd/2$

multiplicative decrease

IF intelligent

Perform **Fast Retransmit** or **Fast Recovery**

ELSE

$wnd := 1 * SMSS$

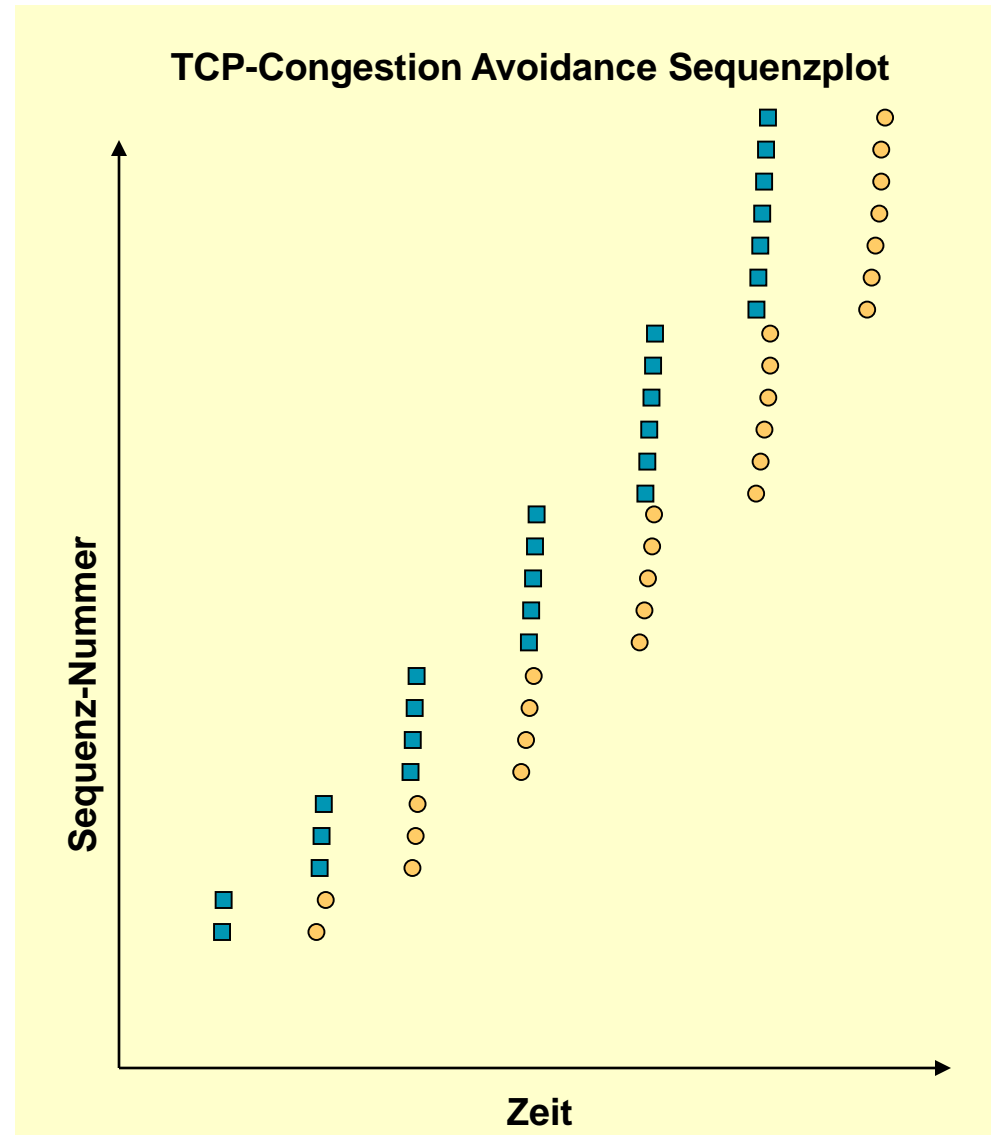
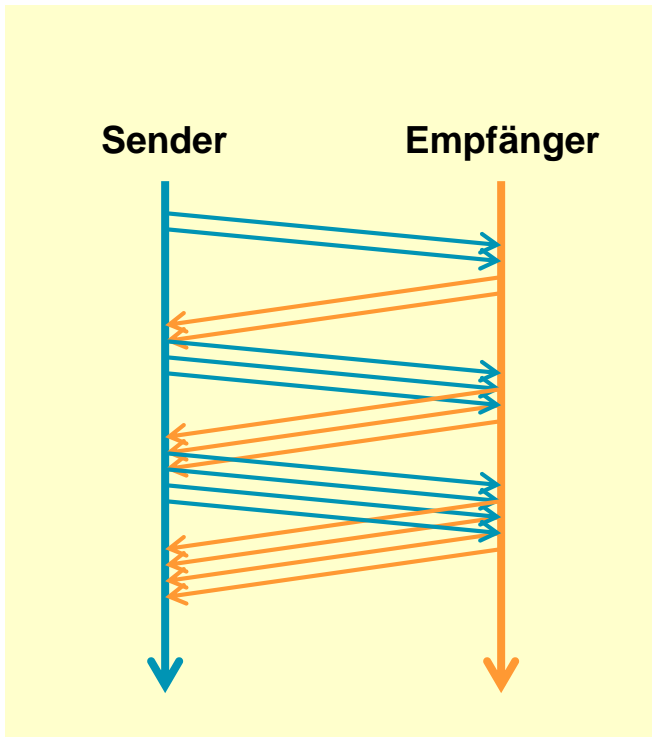
Perform SlowStart

ENDIF

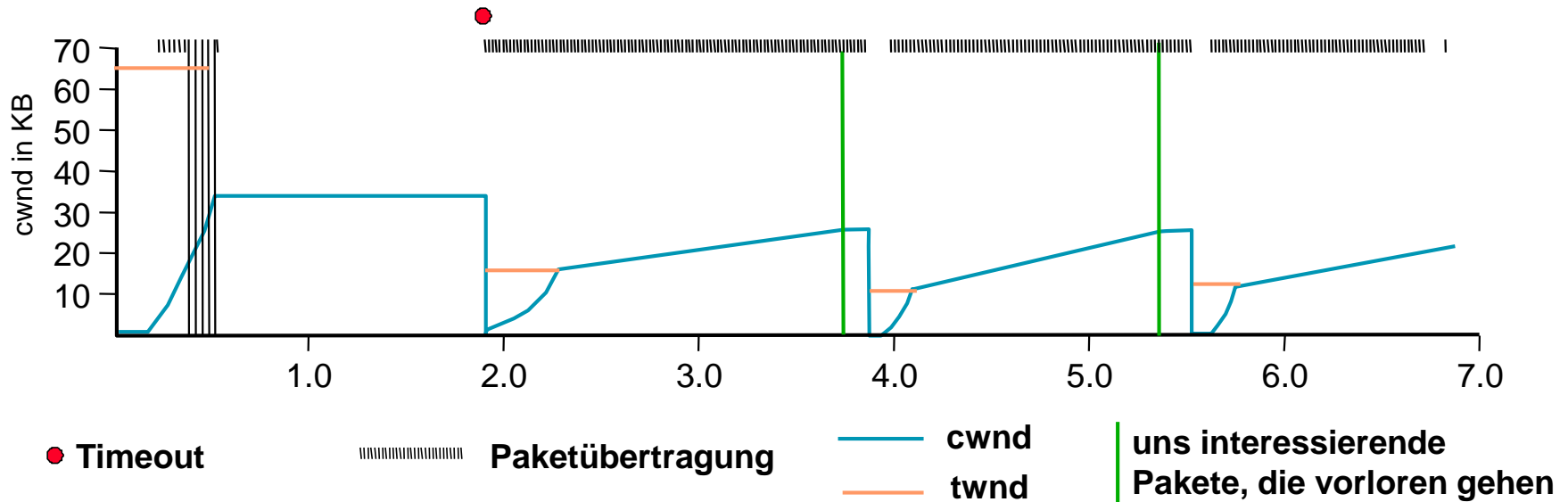
LossEvent kann durch drei
dupACKs oder RTO
angezeigt werden



Veranschaulichung des Congestion Avoidance Algorithmus



Beispiel (mit Fast Retransmit)



- **Verbesserung:** jetzt keine Wartephase bedingt durch RTO
- **noch Problem:** Leerlaufen der Pipeline, Neustart des Selfclockings durch Slow Start.
- **Folge:** kurzzeitiger Einbruch der Übertragungsleistung.
- **Lösung:** direkte Reduzierung des twnds unter gleichzeitiger gezielter Steuerung des cwnd. (Ziel: kontinuierliches unterbrechungsfreies Herunterfahren der Pipe auf den neuen twnd-Wert.)

→ Fast Recovery



Fast Recovery (nach RFC 2581)

Fast Recovery

AS-SOON-AS #dACK=3

$\text{twnd} := 0.5 * \text{cwnd}$ -- multiplik. decrease

$\text{cwnd} := \text{twnd} + 3 * \text{SMSS}$ -- inflating

send(LostSegment)

END AS-SOON-AS

FOR-EACH-ADDITIONAL dACK

$\text{cwnd} := \text{cwnd} + \text{SMSS}$

END-FOR-EACH ADDITIONAL

AS-SOON-AS regular-ACK

$\text{cwnd} := \text{twnd}$ -- deflating

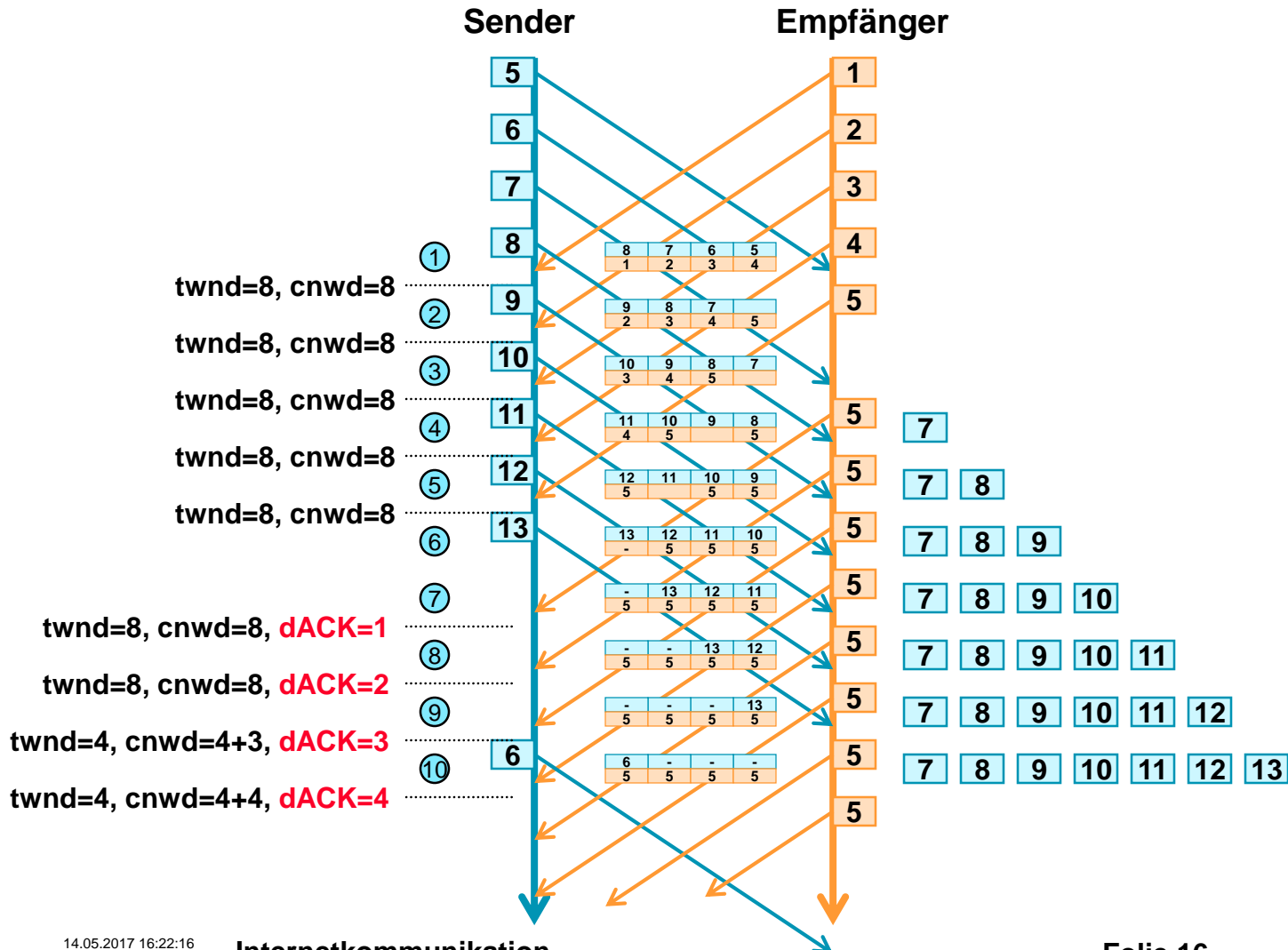
END AS-SOON-AS

Strategie:

Zusätzliches Öffnen des cwnd gemäß "Equilibrium-Konzept" soll kontinuierliches unterbrechungsfreies Herunterfahren der Pipe auf den neuen Soll-Wert des twnd steuern.

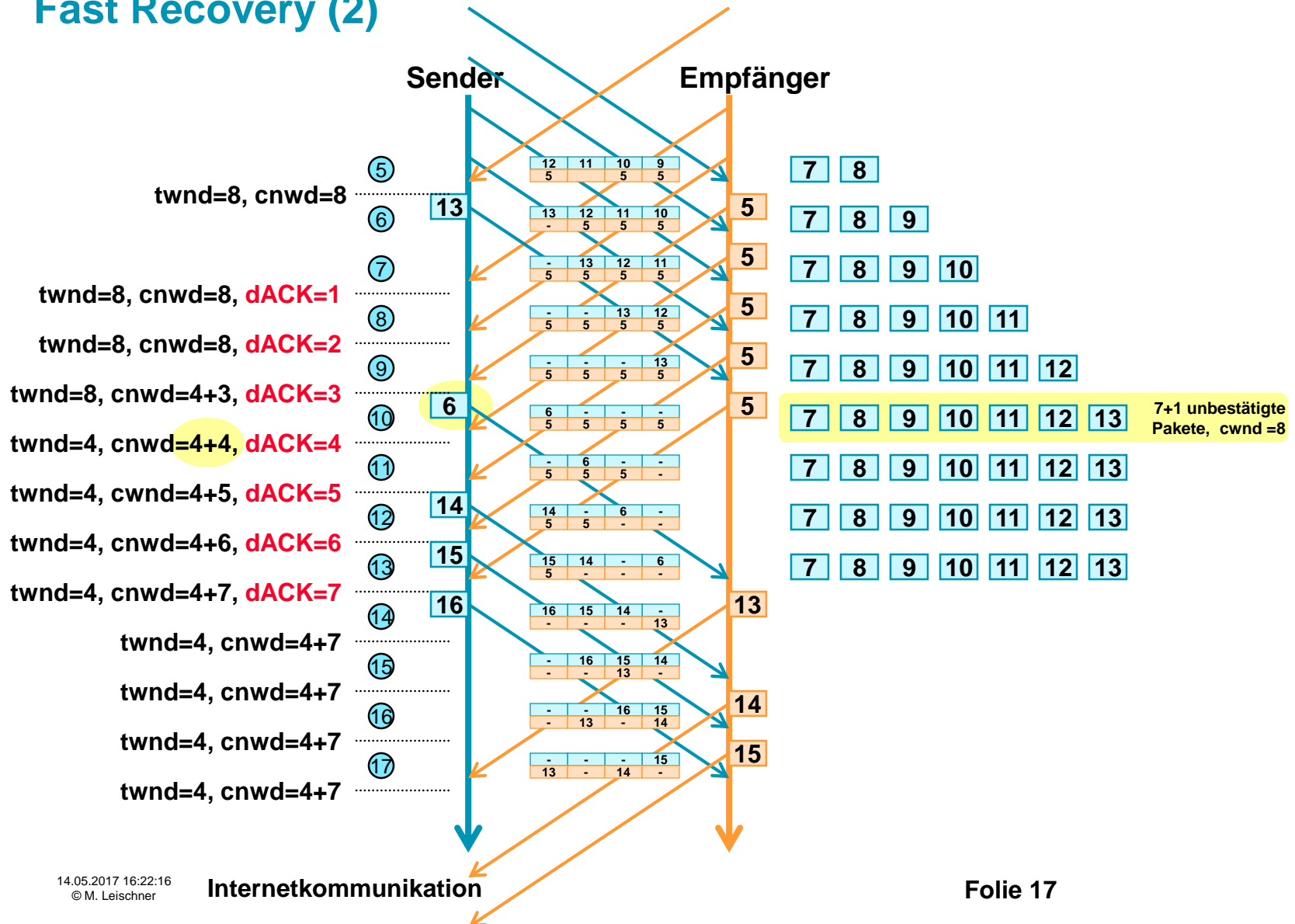


Fast Recovery (1)





Fast Recovery (2)





Fast Recovery (3)

twnd=8, cwnd=4+3, dACK=3

twnd=4, cwnd=4+4, dACK=4

twnd=4, cwnd=4+5, dACK=5

twnd=4, cwnd=4+6, dACK=6

twnd=4, cwnd=4+7, dACK=7

twnd=4, cwnd=4+7

twnd=4, cwnd=4+7

twnd=4, cwnd=4+7

twnd=4, cwnd=4

twnd=4, cwnd=4

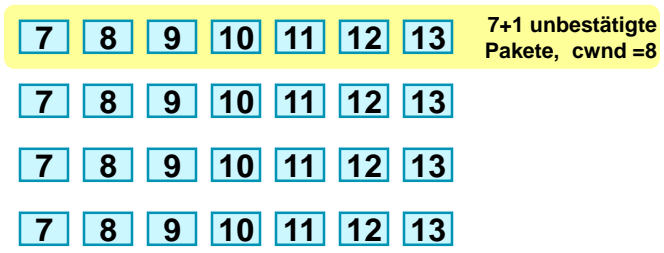
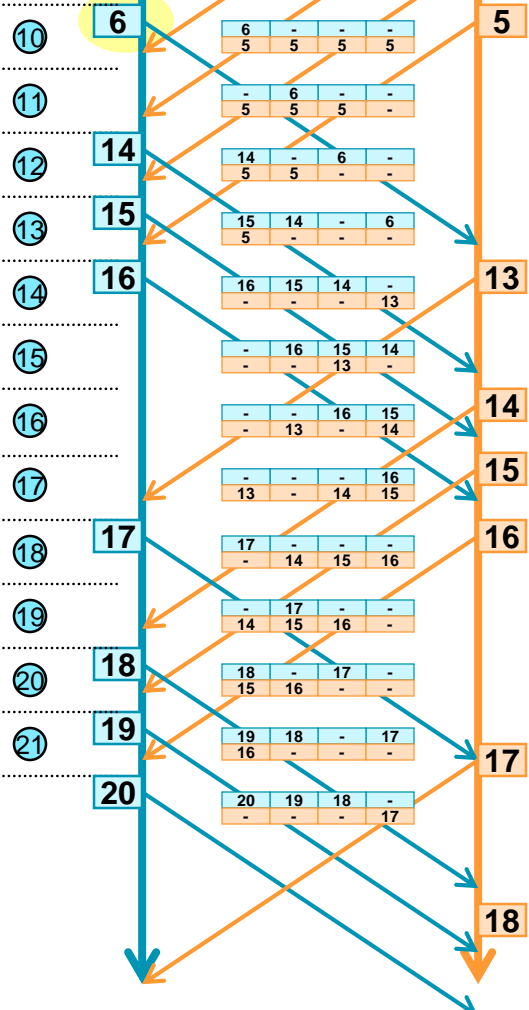
twnd=4, cwnd=4

twnd=4, cwnd=4

twnd=4, cwnd=4

Sender

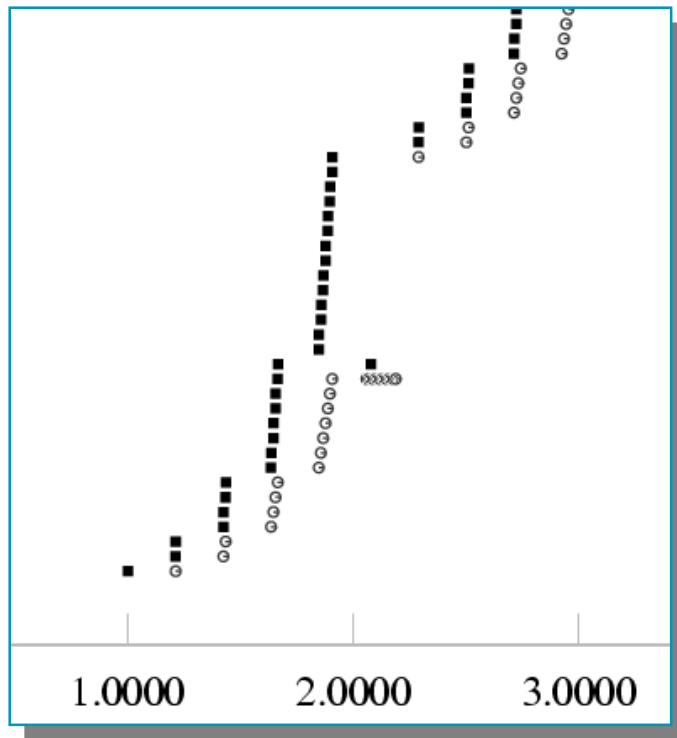
Empfänger





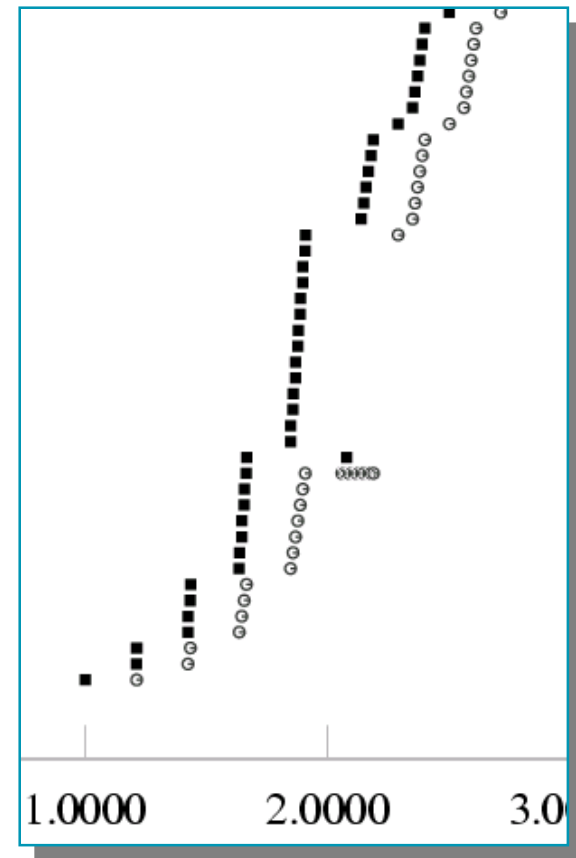
Sequenz-Plots für Fast Retransmit und Fast Recovery

Fast Retransmit



due to: Ramesh Govindan
University of Southern California
<http://netweb.usc.edu>

Fast Recovery

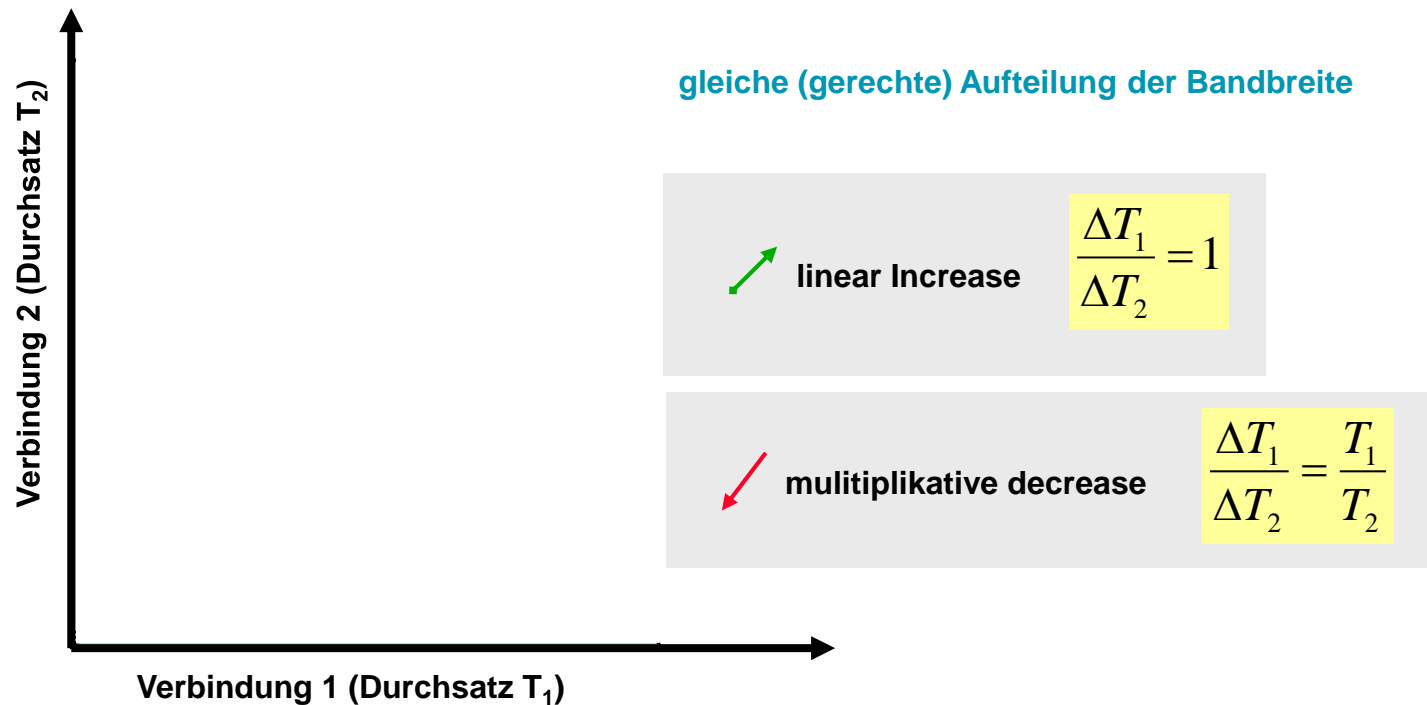




Warum ist TCP fair?

Zwei gleichzeitige TCP-Sessions

- Additive Increase --> Neigung 1
- Multiplicative Decrease --> proportionale Neigung



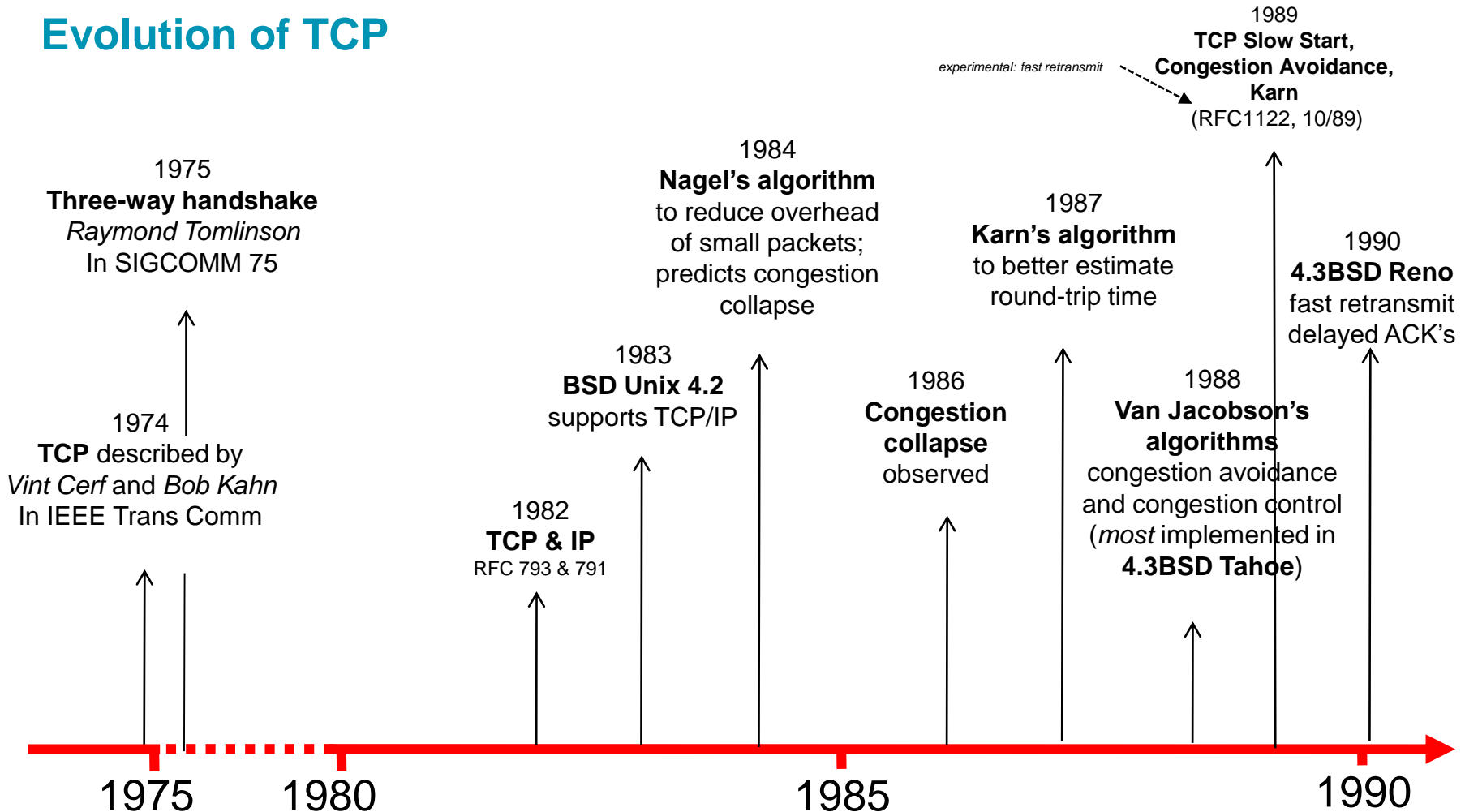
Verschiedene Implementierungen von TCP

- **TCP Tahoe (1988)**
 - BSD (Berkeley Software Distribution) Network Release 1.0
 - enthält AIMD, fast retransmit, Slow Start, RTT nach Jacobson)
 - aber noch kein Fast Recovery.
- **TCP Reno (1990)**
 - BSD Network Release 2.0
 - mit Fast Recovery
 - Implementiert in den meisten Betriebssystemen
- **TCP Vegas (1993)**
 - BSD Network Release 4.3
 - Senderbasierte Überlastvermeidung durch Monitoring des erwarteten und des gemessenen Durchsatzes.
 - Verwendung von Zeitstempeln bei den Paketen.



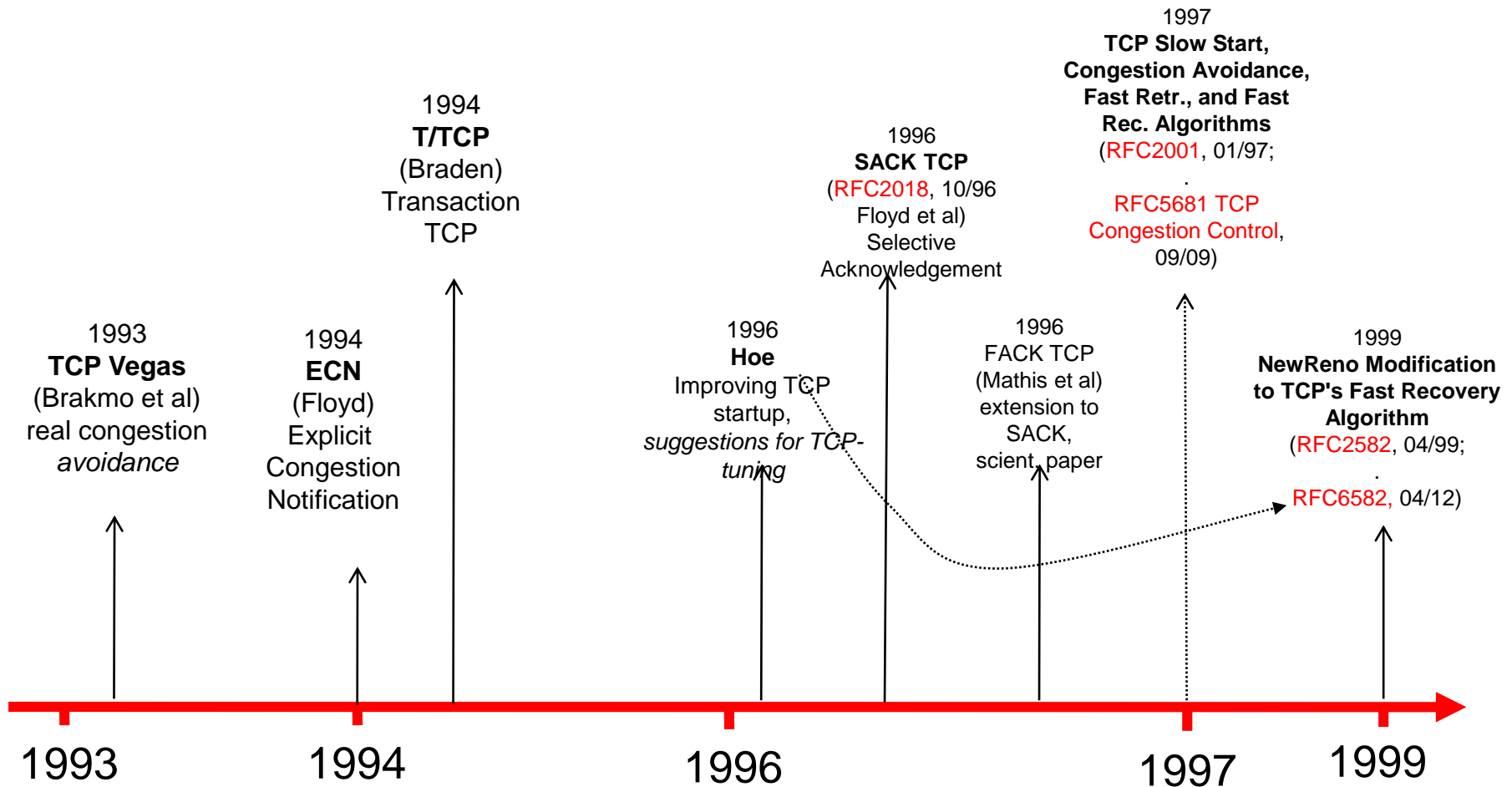


Evolution of TCP



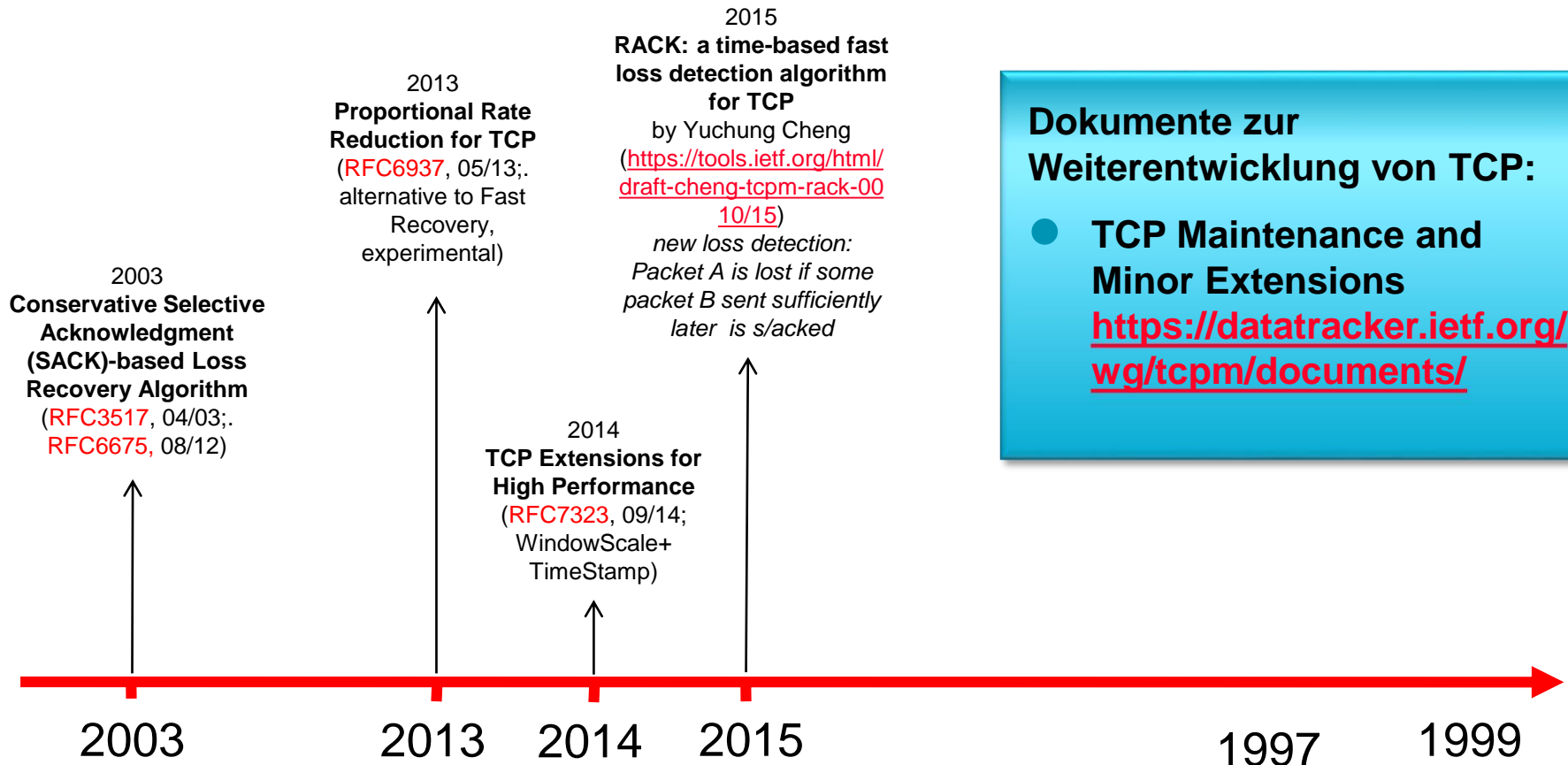


TCP Through the 1990s





TCP Through the 2000s





A Deterministic TCP Bandwidth Sharing Model

W. Lautenschlaeger, A Deterministic TCP
Bandwidth Sharing Model, 2014, online at
<http://arxiv.org/pdf/1404.4173v1>

Traditionally TCP bandwidth sharing has been investigated mainly by stochastic approaches due to its seemingly chaotic nature. Even though of great generality, the theories deal mainly with expectation values, which is prone to misinterpretation with respect to the Quality-of-Experience (QoE). We disassemble TCP operating conditions into dominating scenarios and show that bandwidth sharing alone follows mostly deterministic rules. From the analysis we derive significant root causes of well-known TCP aspects like unequal sharing, burstiness of losses, global synchronization, and on buffer sizing. We base our model on a detailed analysis of bandwidth sharing experiments with subsequent mathematical reproduction.

Problem
Lösung
Methodik



https://www.bell-labs.com/usr/wolfram.lautenschlaeger

self-clocking creates a smooth end-to-end packet flow at the bottleneck's capacity limit.

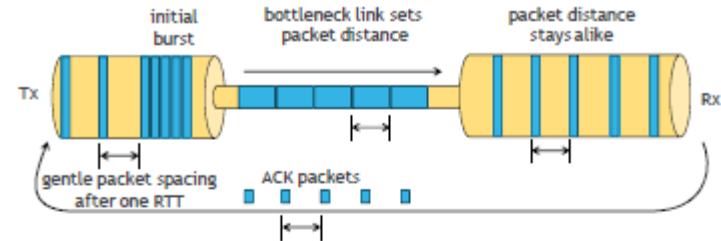


Fig. 1. Adaptation to bottleneck capacity by self-clocking

In case of bandwidth sharing the link capacity is not a bottleneck for a single flow. Other flows might constrict the available bandwidth, so that a single flow experiences macroscopically the equivalent of a bottleneck. On smaller time scales, however, there exists a remarkable difference: Self clocking, as in a hard bottleneck, does not necessarily occur, as illustrated in Fig. 2. Bursts of packets from different flows traverse sequentially the shared link. They are preserved on the reverse acknowledgement chain, and finally reproduce themselves in the next round.

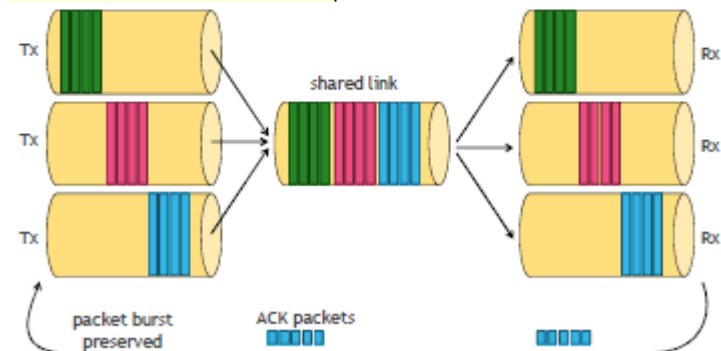


Fig. 2. Burst conservation in case of bandwidth sharing (no self-clocking)

The effect is well known [5], [6]. It can be easily