

4.1 Multiplexing

4.2 UDP

4.3 TCP

4.3.1 Übersicht

4.3.2 Kernfunktionalität

4.3.3 Datenübertragung

4.4 Zusammenfassung

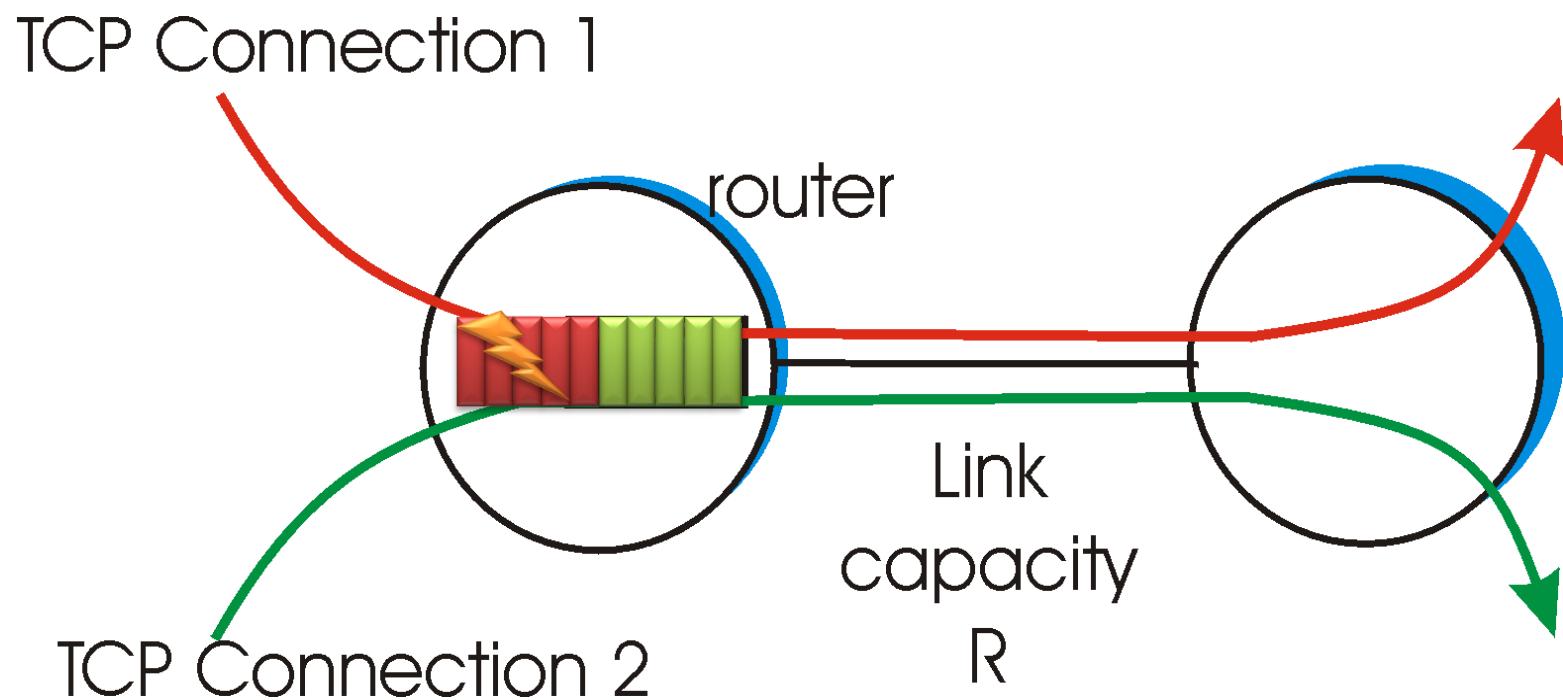
TCP – Transmission Control Protocol

- TCP bildet zusammen mit IP, dem Internet Protokoll, den Kern des Internets
- TCP ist verantwortlich für eine gesicherte Datenübertragung
 - Paketverluste müssen erkannt und durch erneute Übertragung behoben werden
 - die empfangenen Daten müssen in die richtige Reihenfolge gebracht werden
- TCP ist verantwortlich für die „faire“ Verteilung von Übertragungskapazitäten im Internet
 - wenn alle Sender ihre Daten so schnell wie möglich - also so schnell wie es die eigene Netzwerkkarte erlaubt – übertragen, kommt es zur Überlastungen von Links im Internet und Pakete gehen dort verloren
 - die wichtigste Funktion von TCP ist das Anpassen der Senderate auf die „im Internet momentan verfügbare“ Datenrate, d.h. die Datenrate pro Fluss auf dem Bottleneck-Link
 - TCP versucht die Datenrate zu erreichen, die das Ergebnis der theoretischen Berechnung mit dem Max-Min-Fair-Share-Algorithmus sind

TCP – Transmission Control Protocol

- TCP gibt es seit den Anfängen des Internets
 - RFC 675: SPECIFICATION OF INTERNET TRANSMISSION CONTROL PROGRAM, December 1974 (Vince Cerf et al)
- Die Kernfunktionalität wurde 1981 in RFC 793 festgelegt
 - RFC 793: TRANSMISSION CONTROL PROTOCOL, September 1981
- Seitdem gibt es zahlreiche Ergänzungen und unterschiedliche Versionen insbesondere der Congestion Control, um TCP zu verbessern. Diese werden in RFC 7414 zusammengefasst und beschrieben
 - RFC 7414: A Roadmap for Transmission Control Protocol (TCP) Specification Documents, February 2015
- Es gibt keine einheitliche Version von TCP, die im Internet auf allen Rechnern läuft. Alle TCP Versionen müssen die Kernfunktionalität implementieren, können sich aber speziell hinsichtlich der Congestion Control unterscheiden.

Bottleneck und Paketverlust durch Buffer-Overflow



TCP Congestion Control

- Die TCP Congestion Control (Überlaststeuerung) bestimmt, wie schnell ein Sender Daten übertragen darf.
- Die unterschiedlichen Versionen wurden zeitweise nach Orten in Nevada bzw. den entsprechenden BSD Unix Releases (TCP Tahoe, TCP Reno, TCP Vegas, TCP New Reno) benannt
- RFCs 2001 und 2581 spezifizieren die TCP Congestion Control New Reno.
 - RFC 2581: TCP Congestion Control, April 1999 (Allman, Paxson, Stevens)
- Die aktuelle Version der von der IETF empfohlenen TCP Congestion Control ist in RFC 5681 beschrieben. Die Vorlesung beschreibt weitgehend diese Version.
 - RFC 5681: TCP Congestion Control, September 2009 (Allman, Paxson, Blanton)
- Die eigentliche TCP Version ist abhängig vom Betriebssystem
 - unter Windows wird derzeit Compound TCP verwendet
 - unter Linux kann die TCP Version ausgewählt werden, Default ist häufig die Version TCP Cubic

4.1 Multiplexing

4.2 UDP

4.3 TCP

4.3.1 Übersicht

4.3.2 Kernfunktionalität

4.3.2.1 Paketformat

4.3.2.2 Verbindungsaufbau

4.3.2.3 Fehlerbehandlung

4.3.2.4 Verbindungsabbau

4.3.3 Datenübertragung

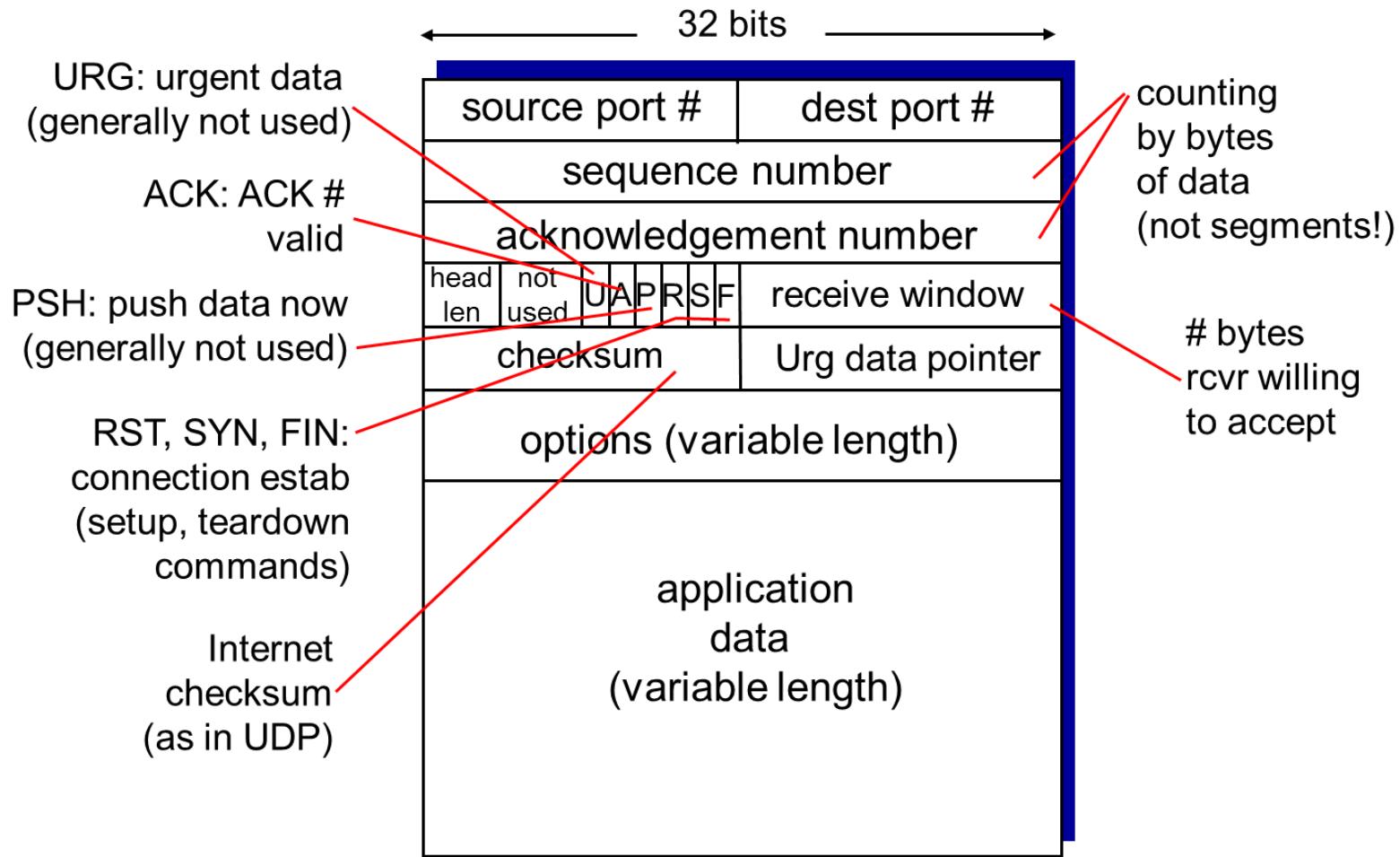
4.4 Zusammenfassung

TCP - Kernfunktionalität

- Verbindungsaufbau und –abbau
- Paketformat
- Erkennung von Paketverlusten über Acknowledgements und Timer
- Datenflusssteuerung und einen fenster-basierter Sendemechanismus
 - Flow Control with a Sliding Window Mechanism

TCP - Paketformat

- TCP -Header enthalten wesentlich mehr Informationen als UDP-Header
- Gemeinsamkeit: Source /Destination-Port, Checksum



4.1 Multiplexing

4.2 UDP

4.3 TCP

4.3.1 Übersicht

4.3.2 Kernfunktionalität

4.3.2.1 Paketformat

4.3.2.2 Verbindungsaufbau

4.3.2.3 Fehlerbehandlung

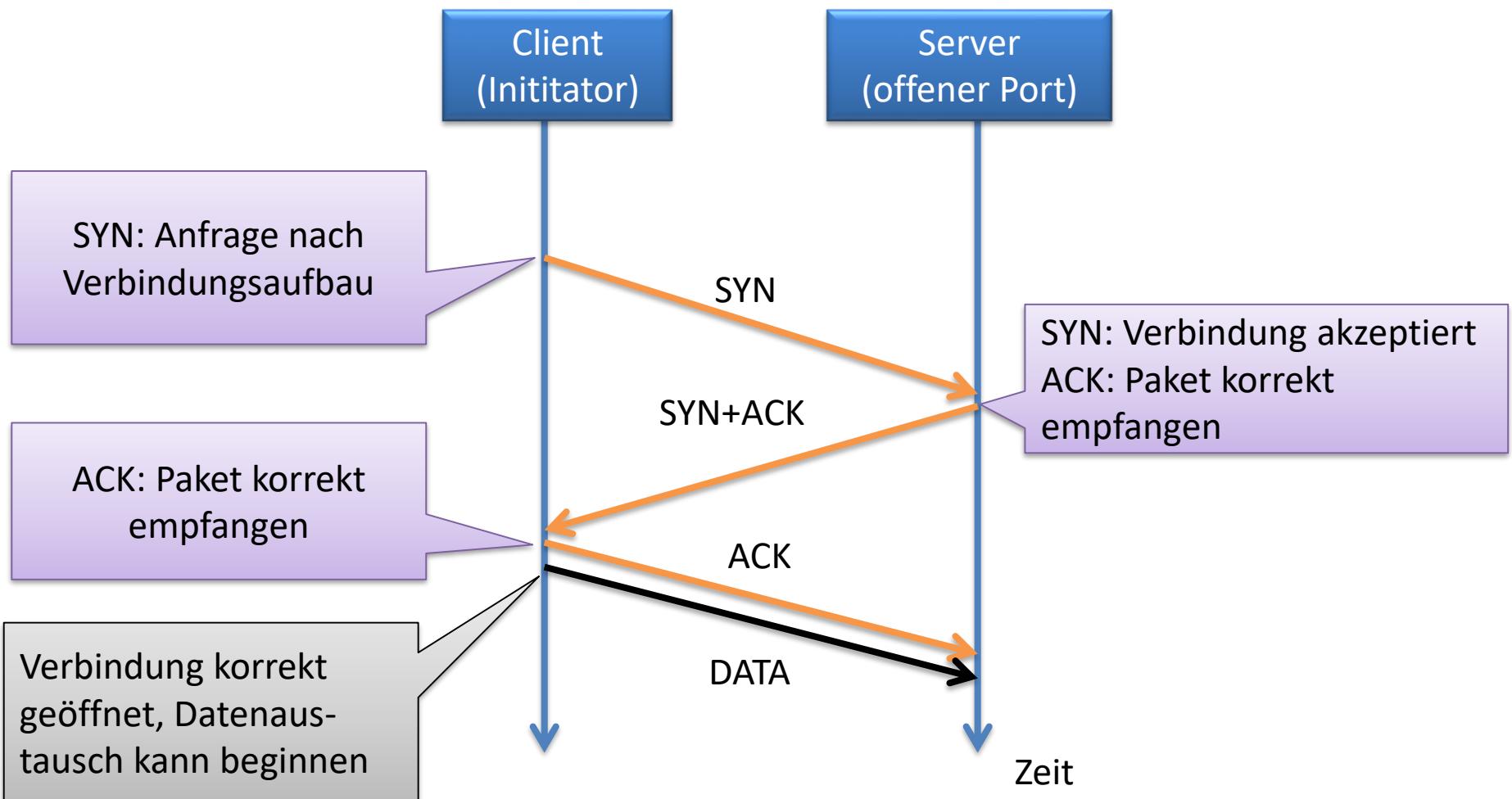
4.3.2.4 Verbindungsabbau

4.3.3 Datenübertragung

4.4 Zusammenfassung

TCP Verbindungsauftbau

- TCP Verbindungsauftbau über Three-Way-Handshake
- Verbindung ist nach dem Austausch von drei Paketen (SYN, SYN+ACK, ACK) geöffnet
- ACK (Acknowledgment) bestätigt den korrekten Empfang
- Letztes ACK ist notwendig, um korrekten Empfang des SYN+ACK Pakets unmittelbar zu bestätigen. DATA wird erst nach Interaktion mit Anwendung (erfolgreicher Abschluss der connect-Anweisung und darauffolgende send-Anweisung) gesendet.

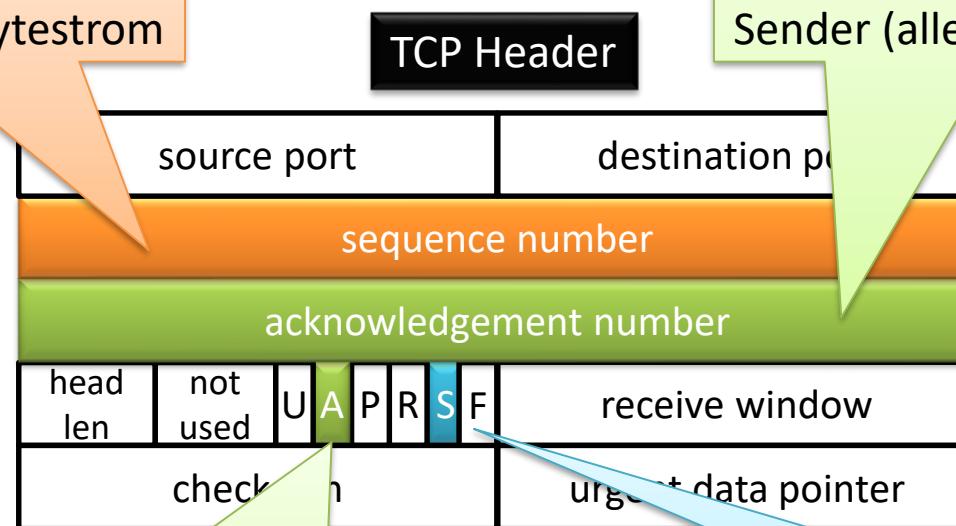


TCP Verbindungsaufbau - Header

Sequenz-Nummer:

Nummer des ersten Bytes dieses Segments im Bytestrom

ACK-Nummer: Nummer des nächsten erwarteten Bytes vom Sender (alles vorher korrekt)



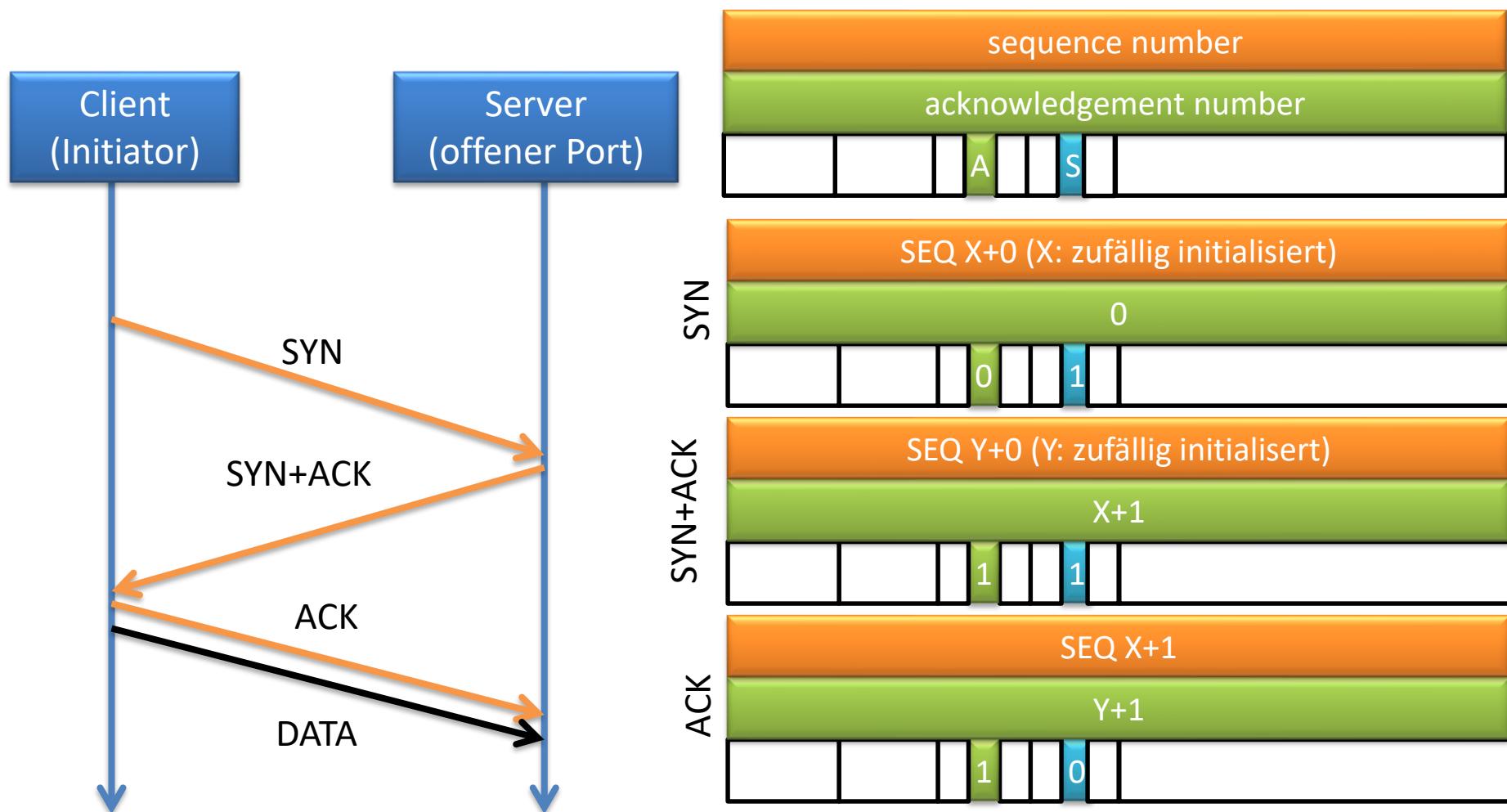
ACK-Flag (Acknowledgment):

wird gesetzt, wenn Paket Empfang neuer Daten quittiert

SYN-Flag (Synchronisierung): wird gesetzt um neue Verbindungen anzufragen und zu akzeptieren

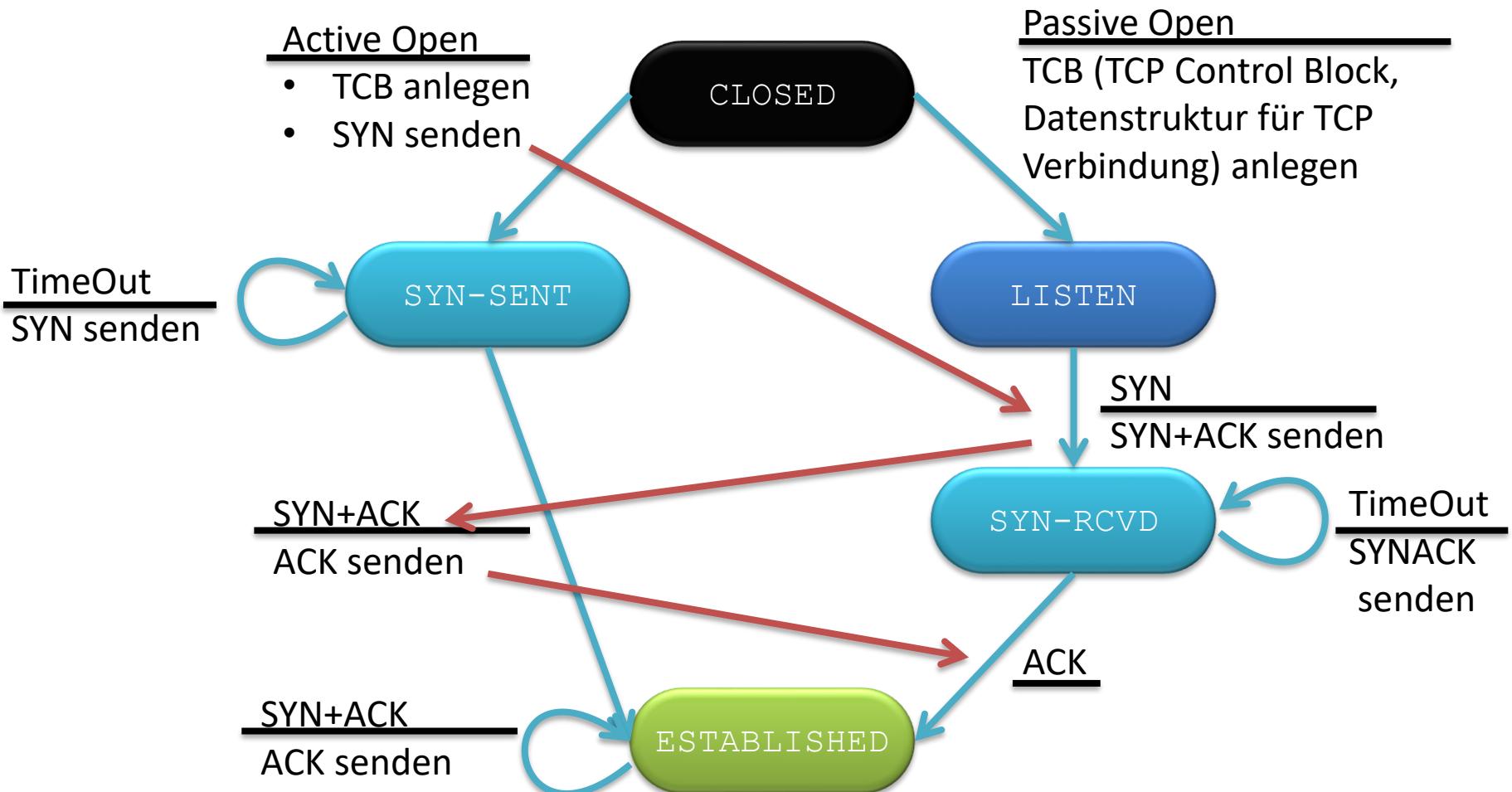
Header beim Three-Way-Handshake

- Start der Sequenz-Nummern einer TCP-Verbindung werden zufällig gewählt und fangen nicht bei 0 an, um Überschneidungen mit verspäteten Paketen alter Verbindungen zu vermeiden.
- Acknowledgement-Nummer für ein SYN-Paket (enthält ein Byte Daten) ist jeweils die initiale Sequenznummer +1, (SYN ist Byte X, als nächstes erwartet wird Byte X+1, X wurde korrekt empfangen)



TCP Zustandsdiagramm (Verbindungsauftbau)

- TCP ist ein zustandsbehaftetes (stateful) Protokoll
- Client und Server merken sich den Zustand (CLOSED, LISTEN, SYN-RCVD, SYN-SENT, ESTABLISHED) der TCP Verbindung. Übergänge zwischen Zuständen ausgelöst durch Empfang von Nachrichten (extern: Pakete, intern: Timeout); können das Senden von Nachrichten beinhalten.
- Client und Server legen für jede TCP-Verbindungen einen TCB (TCP Control Block, Datenstruktur für TCP Verbindungen) an, in dem der detaillierte Zustand der TCP Verbindung gespeichert wird.



4.1 Multiplexing

4.2 UDP

4.3 TCP

4.3.1 Übersicht

4.3.2 Kernfunktionalität

4.3.2.1 Paketformat

4.3.2.2 Verbindungsaufbau

4.3.2.3 Fehlerbehandlung

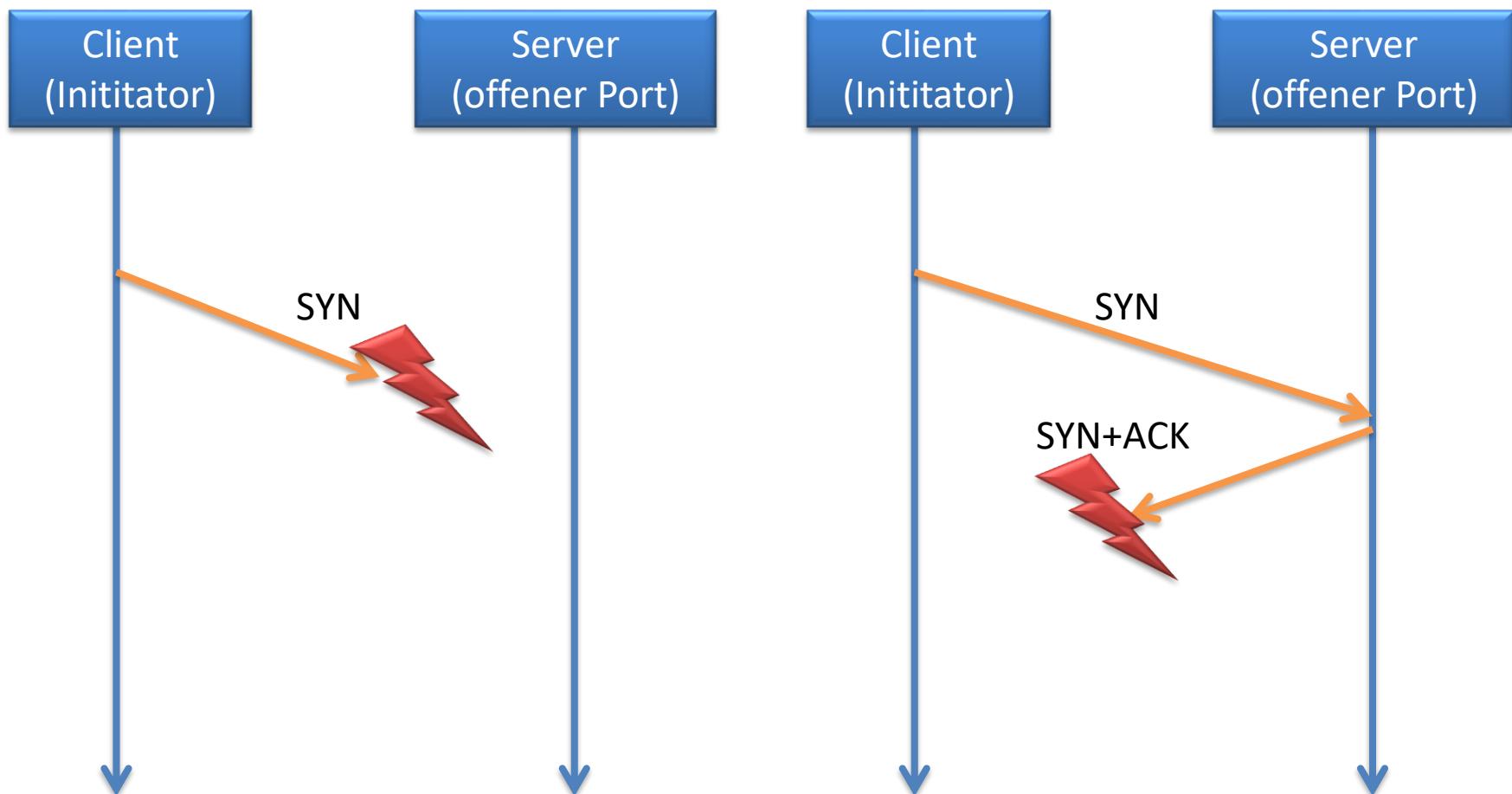
4.3.2.4 Verbindungsabbau

4.3.3 Datenübertragung

4.4 Zusammenfassung

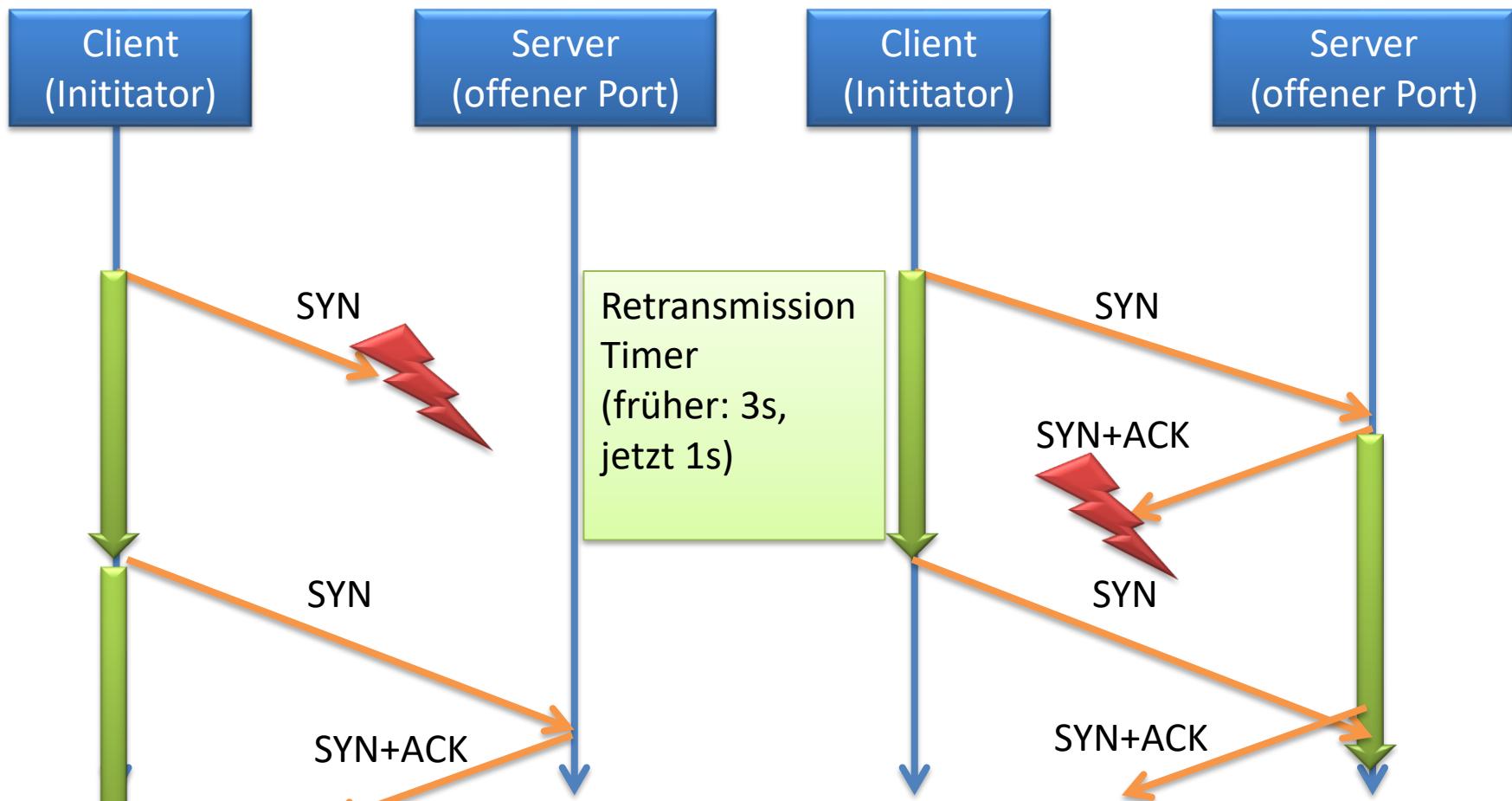
Paketverluste

- Pakete können auf dem Weg zum Internet verloren gehen
- Was passiert?

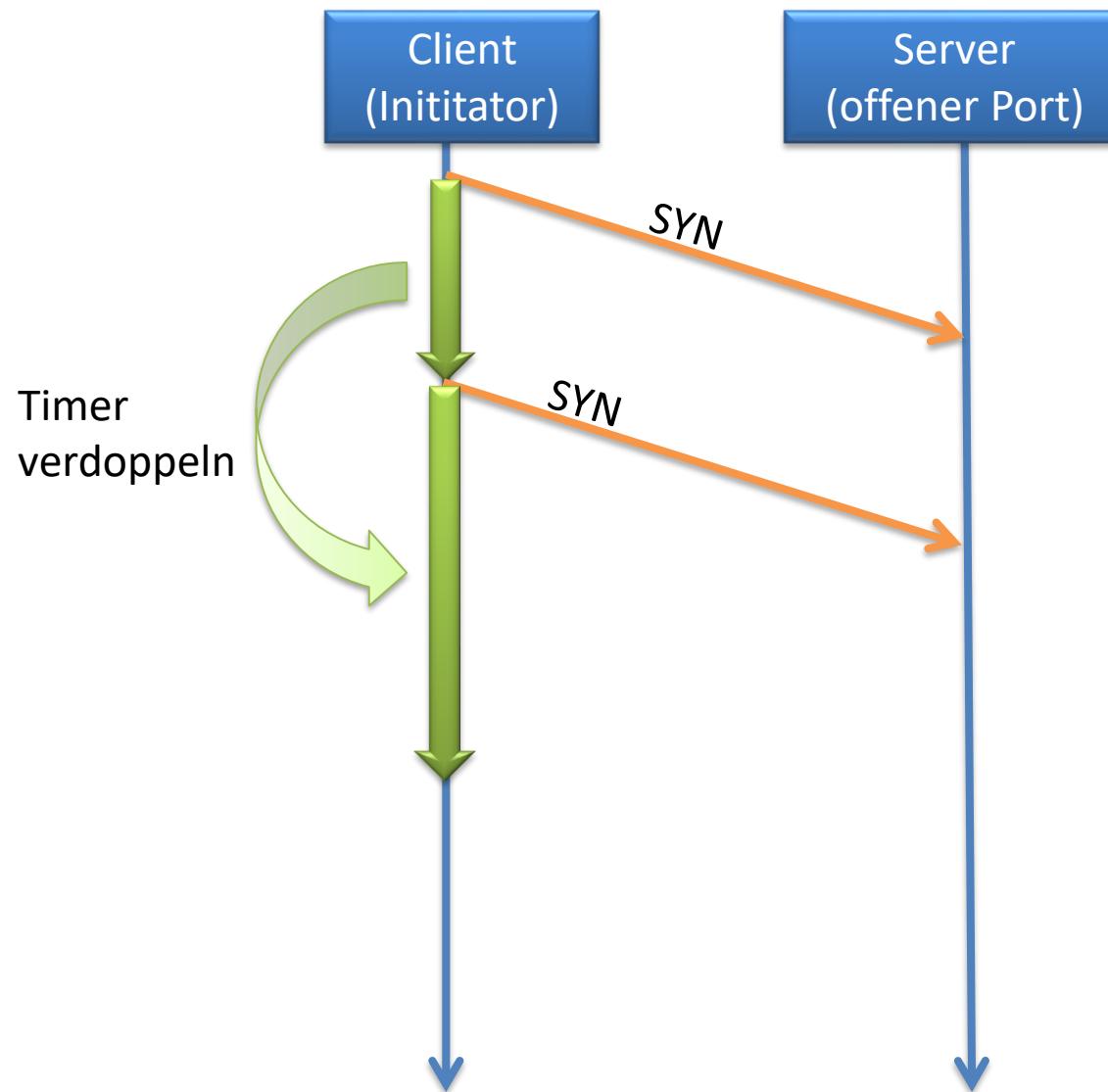


Paketverluste

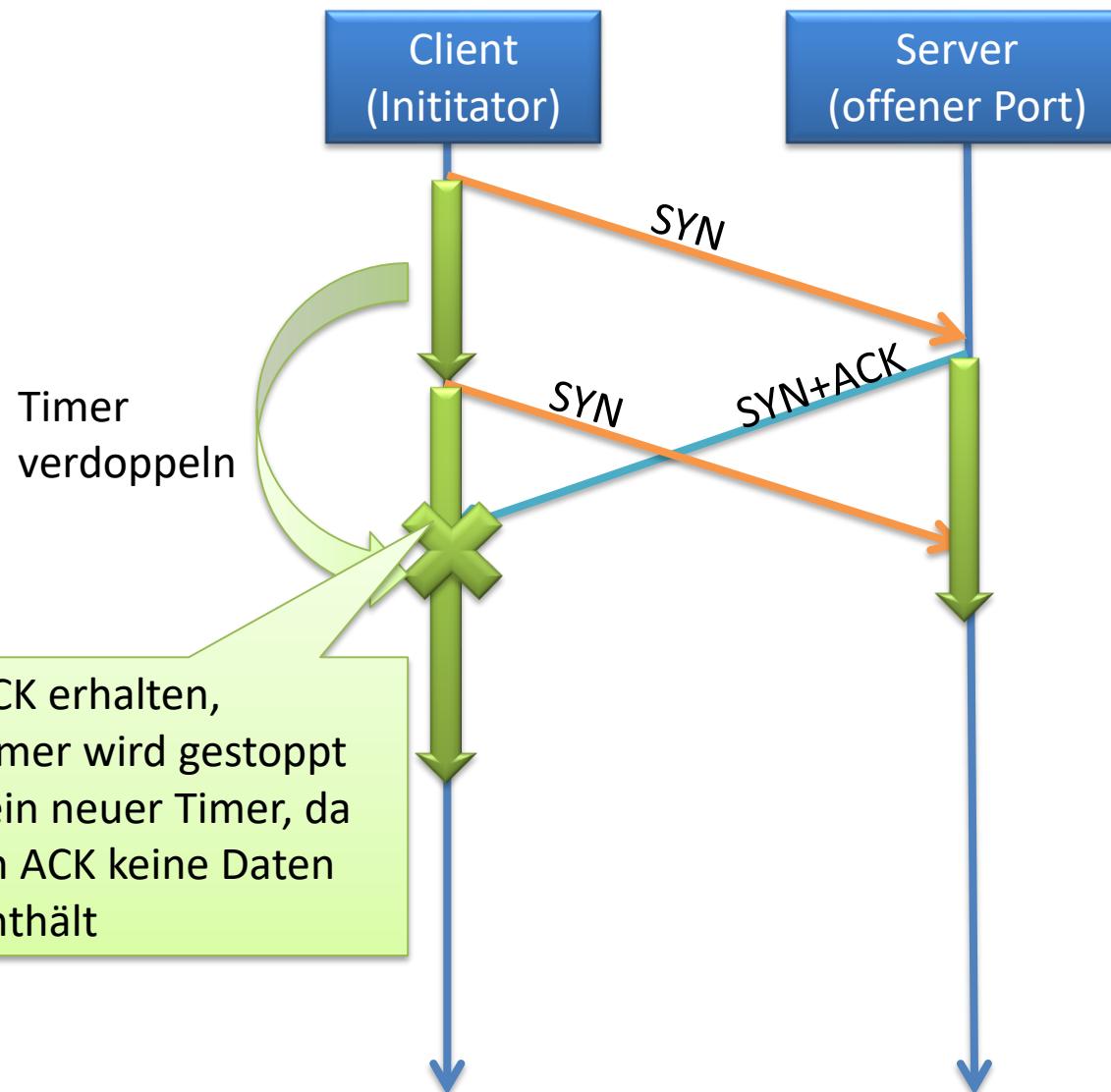
Beim Versenden eines Pakets wird ein Timer, der Retransmission Timer, gestartet. Beim Ablauf dieses Timers, einem Retransmission Timeout, wird das Paket noch einmal gesendet. Trifft ein Acknowledgement für das Paket vor Ablauf des Timers ein, so wird der Timer gestoppt und es kommt zu keiner wiederholten Übertragung. Ein typischer initialer Wert für den Retransmission Timer ist eine Sekunde (RFC 6298, konfigurierbar). Geht ein Paket mehrfach verloren, so wird der Retransmission Timer bei jeder Übertragungswiederholung verdoppelt. Dies wird wiederholt, bis die maximale Anzahl von Wiederholungsübertragungen erreicht ist (hängt vom Betriebssystem ab, konfigurierbar).



Timeout zu klein



Timeout zu klein



4.1 Multiplexing

4.2 UDP

4.3 TCP

4.3.1 Übersicht

4.3.2 Kernfunktionalität

4.3.2.1 Paketformat

4.3.2.2 Verbindungsaufbau

4.3.2.3 Fehlerbehandlung

4.3.2.4 Verbindungsabbau

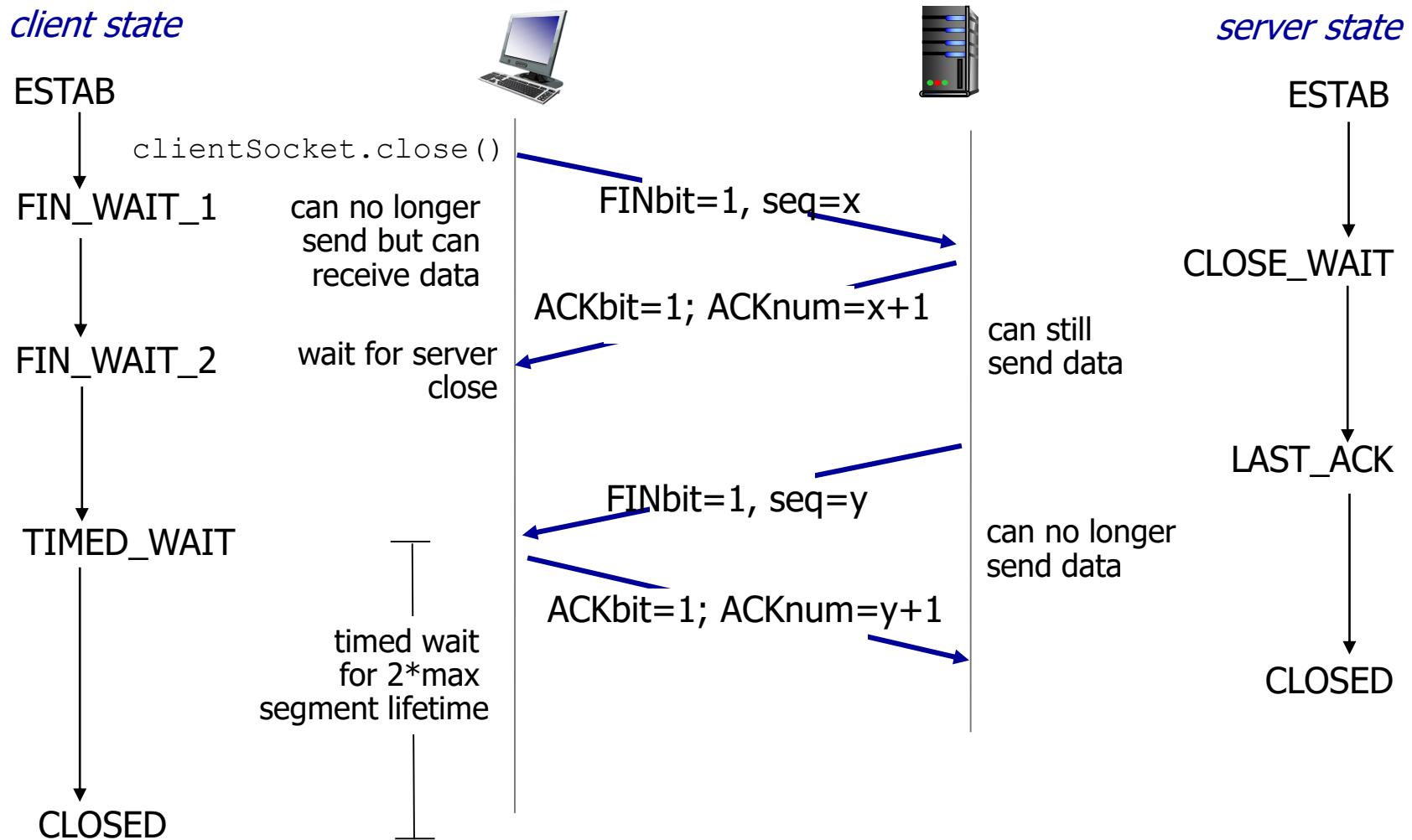
4.3.3 Datenübertragung

4.4 Zusammenfassung

Beenden einer TCP Verbindung

- Initiieren der Close-Prozedur einer TCP-Verbindung
 - Client
 - Server
 - beide gleichzeitig
- Close-Prozedur
 - Client und Server müssen beiden FINs senden
 - Erhalt beider FINs muss bestätigt bleiben
 - Timeout nach Senden des ACKs für den Fall, dass das ACK verloren geht und das FIN wiederholt übertragen wird

Verbindungsabbau



4.1 Multiplexing

4.2 UDP

4.3 TCP

4.3.1 Übersicht

4.3.2 Kernfunktionalität

4.3.3 Datenübertragung

4.3.3.1 Paketgröße – Maximum Segment Size

4.3.3.2 Datenflussteuerung – Flow Control

4.3.3.3 Überlaststeuerung – Congestion Control nach RFC 5681

4.3.3.4 Fairness

4.3.3.5 Installierte TCP Varianten

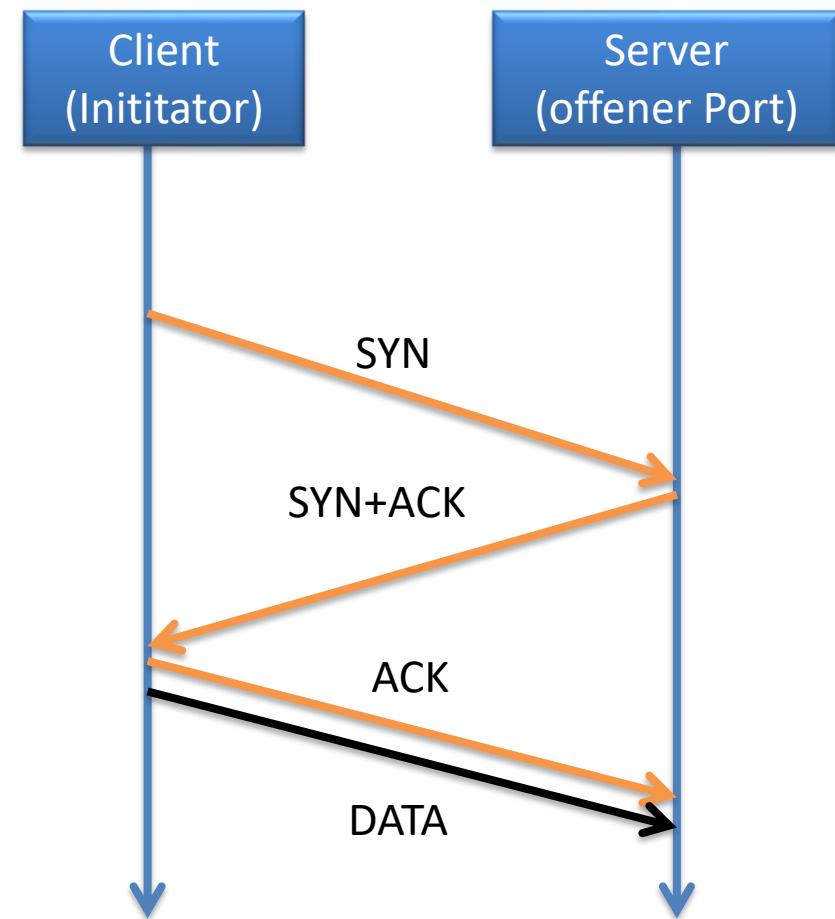
4.4 Zusammenfassung

Datenübertragung

- Die TCP-Verbindung ist aufgebaut. Die Datenübertragung kann beginnen.
 - Wie groß sind die Segmente?



- Wie viele Daten darf ich senden?
 - immer nur ein Paket?
 - alle auf einmal?



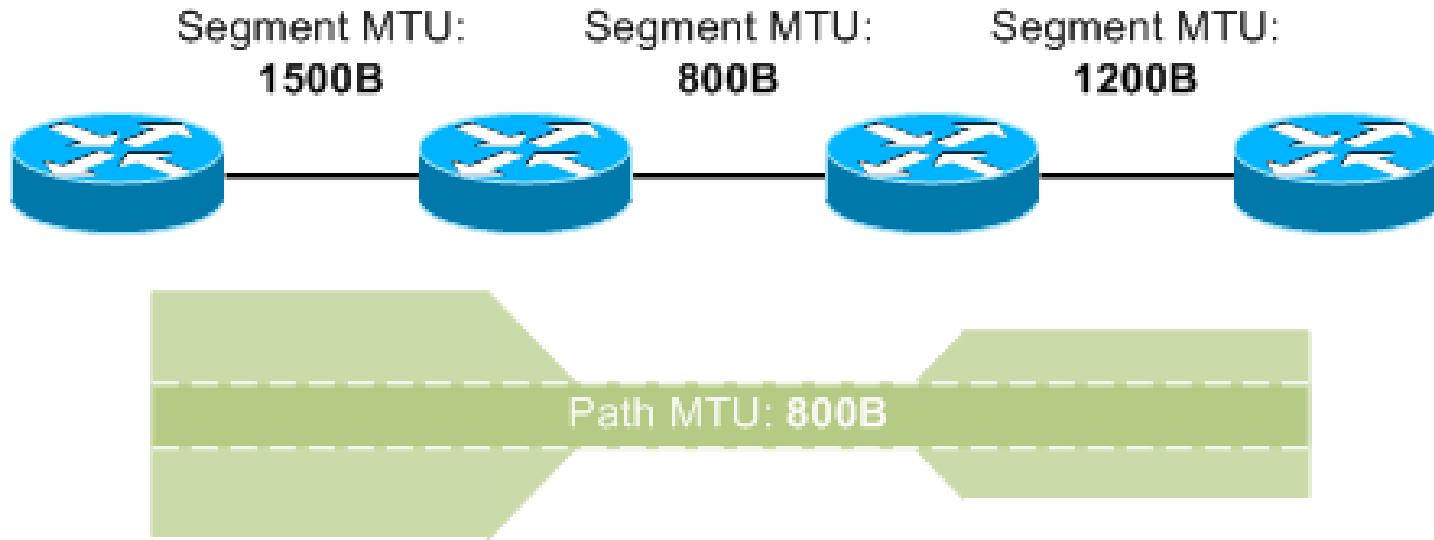
Wie groß darf ein Paket sein?

- Payload eines TCP Segments ist limitiert durch die Maximum Segment Size
 - MTU (Maximum Transfer Unit) ist die maximale Payload (Größe eines IP-Pakets), das nicht fragmentiert werden muss um von Protokollen unterhalb von IP übertragen zu werden
 - Ethernet hat eine MTU von 1500 Bytes, was in einer MSS von 1460 Bytes plus 20 Bytes TCP Header plus 20 Bytes IP Header resultiert
- TCP versucht die größte MSS zu wählen, die
 - vom Empfänger der Nachrichten unterstützt wird
 - unterstützte MSS kann in Header-Option in SYN Paket mitgeteilt werden
 - von allen Links auf dem Pfad unterstützt wird
 - MTU Path Discovery (verschiedene Varianten)
 - Probleme: 8 Byte DSL Header, zusätzlicher Tunnel-Header
 - Vermeidung der Fragmentierung von IP-Paketen

MTU Path Discovery

Verschiedene Varianten

- über ICMP (Internet Control Message Protokoll): Links informieren Sender über kleinere MTU
- über MSS Clamping: Router reduzieren MSS in TCP Header
- Packetization Layer Path MTU Discovery: TCP "versucht" verschiedene Paketgrößen



MTU Path Discovery –Praxis

- TCP Three-Way-Handshake
 - in SYN und SYN-ACK Paketen wird über eine TCP Header Option eine MSS ausgetauscht
 - der Client trägt die MSS entsprechend seines lokalen Interfaces ein
 - Windows: netsh interface ipv4 show subinterface
 - alle Router auf dem Weg und der Server können die MSS entsprechend ihrer MTU reduzieren
 - der Server nutzt die empfangene MSS
 - das Verfahren für den Pfad von Server zu Client erfolgt analog
- Path MTU Discovery (für IPv4)
 - wird im Betriebssystem als Default-Einstellung bzw. individuell für jeden Socket aktiviert oder deaktiviert
 - mit Path MTU Discovery setzt IP bei allen Paketen das DF (Dont' Fragment) Bit
 - Router, die das IP Paket nicht ohne Fragmentierung weiterleiten können, weil das Paket größer als die MTU ist, verwerfen das Paket und informieren den Sender darüber via ICMP
 - der Sender (TCP) verringert seine MSS nach einem bestimmten Algorithmus. Das ICMP Paket selbst enthält keine Information über die unterstützte MTU
 - die Path MTU Discovery sucht nicht aktiv nach der perfekten MTU. Die MSS wird reaktiv bei Problemen verringert.
 - Path MTU Discovery funktioniert nicht, wenn Router keine ICMP-Nachrichten senden oder diese blockieren

4.1 Multiplexing

4.2 UDP

4.3 TCP

4.3.1 Übersicht

4.3.2 Kernfunktionalität

4.3.3 Datenübertragung

4.3.3.1 Paketgröße – Maximum Segment Size

4.3.3.2 Datenflusssteuerung – Flow Control

4.3.3.3 Überlaststeuerung – Congestion Control nach RFC 5681

4.3.3.4 Fairness

4.3.3.5 Installierte TCP Varianten

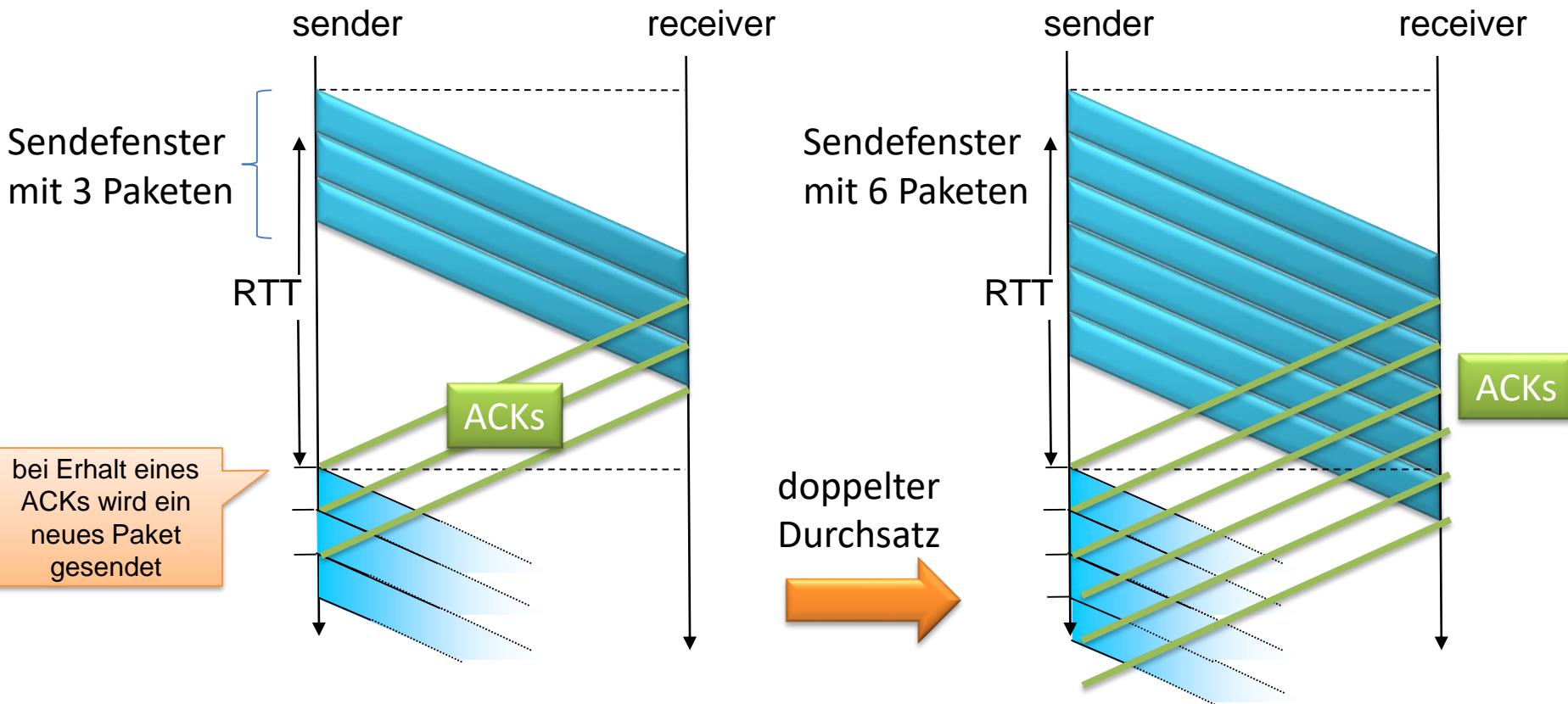
4.4 Zusammenfassung

Datenflusssteuerung – Flow Control

- Die Datenflusssteuerung legt fest
 - wie schnell ein Sender Daten an den Empfänger senden darf
 - wie der Empfänger den Empfang von Daten bestätigt
 - wie mit Paketverlusten umgegangen wird
- Grundlegende Verfahren
 - Send-and-Wait
 - Go-Back-N
 - Selective Repeat
- Go-Back-N und Selective Repeat sind sendefenster-basierte Verfahren, d.h. die Geschwindigkeit, mit der Daten übertragen werden können, wird über ein **Sendefenster** reguliert.

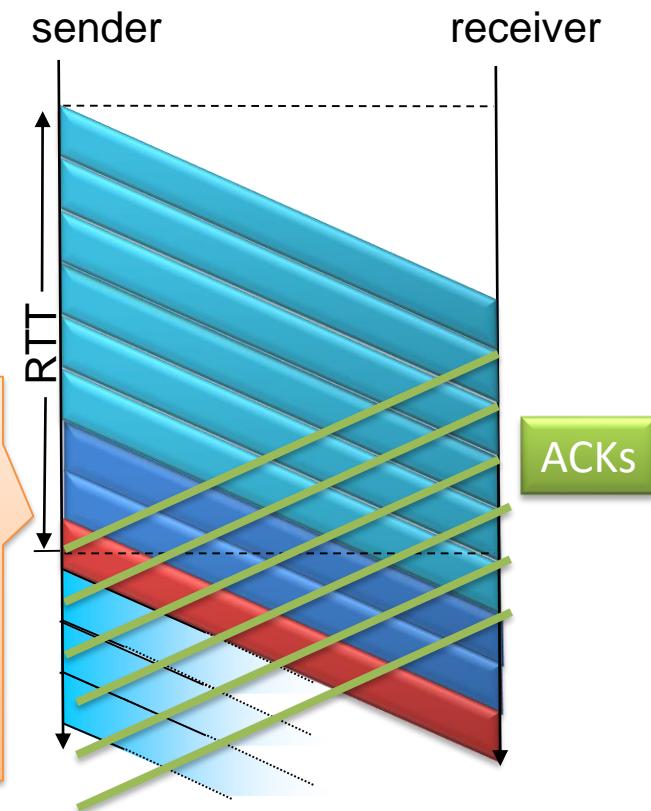
Sendfenster

- Das Sendfenster bezeichnet die Anzahl von Daten oder Paketen, die ein Sender an den Empfänger abschicken darf ohne ein Bestätigung (ACK) erhalten zu haben.
- Die Anzahl versendeter und unbestätigter Daten wird als Flightsize bezeichnet. Die Flightsize muss immer kleiner als das Sendfenster sein.
- Erhält der Sender eine Empfangsbestätigung (ACK) darf er neue Daten schicken, da im Sendfenster "Platz" geworden ist bzw. da sich die Flightsize verringert hat.



Bandwidth-Delay-Product

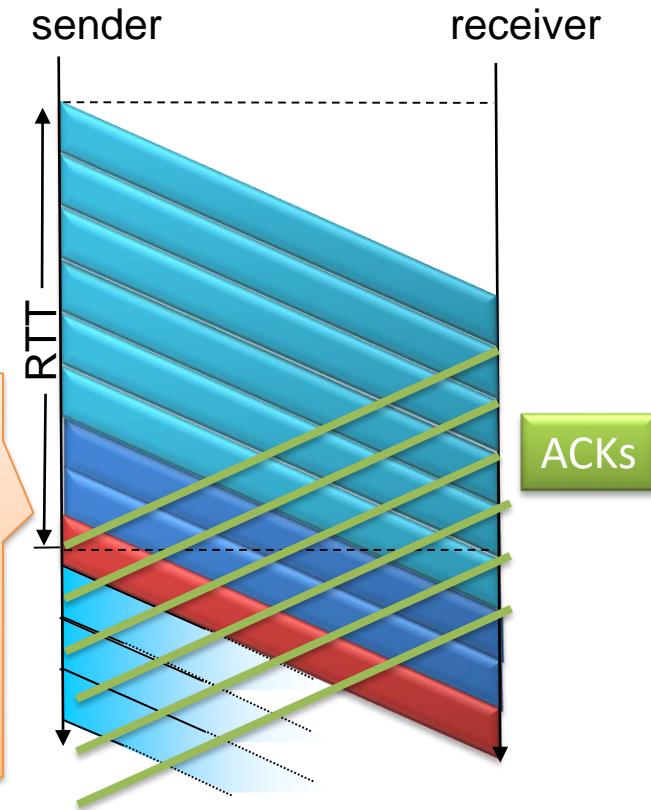
- Vergleich: Sendefenster von 6 Paketen verdoppelt den Durchsatz gegenüber Sendefenster von 3 Paketen
- Wo ist die Grenze?



während einer RTT können 8 volle und der Teil eines Paketes gesendet werden. Ab einer Fenstergröße von 9 ist der Link voll ausgelastet.

- Anzahl Pakete bis Eintreffen des ersten ACK bei Kapazität C und Paketgröße L?
 - Zeit pro Paket: L/C
 - Anzahl Pakete: $RTT/(L/C) = RTT \cdot C/L$
 - **Sendefenster in Bit: $RTT \cdot C$**
- Das Bandwidth-Delay-Product (Delay=Round-Trip-Delay) bezeichnet das Sendefenster, das notwendig ist, um die Bandbreite einer Übertragungsstrecke vollständig auszunutzen.

Bandwidth-Delay-Product (Beispiel)



- Beispiel: 16 Mbps DSL mit 1500 Byte Paketen und 30 ms RTT
 - $L/C: 12e3 \text{ b} / 16e6 \text{ bps} = 0,75 \text{ ms}$
 - $RTT/(L/C): 30e-3\text{s} / 0,75 \text{ ms} = 40 \text{ Pakete}$
 - $RTT*C: 16e6 \text{ bps} * 30e-3\text{s} = 60 \text{ kB}$
- Bandwidth-Delay-Product: 60kB
- ein Sendefenster von 60 kB wird benötigt, um den DSL-Link vollständig auszunutzen.

Paketverluste: Go-Back-N und Selective Repeat

Go-Back-N und Selective Repeat sind zwei generelle Verfahren zur Datenflusssteuerung, auf deren Grundideen die Datenflusssteuerung von TCP aufgebaut ist.

Go-Back-N

- Sendefenster von N Paketen
- Empfänger sendet kumulative ACKs
 - kumulativ: ACK für alle bisher korrekt empfangenen Pakete
 - in TCP nächste erwartete Sequenznummer
- Sender hält **einen** Retransmission Timer
 - Retransmission Timer wird bei jedem Senden eines Pakets neu initialisiert
 - bei einem Timeouts wird vollständiges Fenster noch einmal übertragen

Selective Repeat

- Sendefenster von N Paketen
- Empfänger sendet individuelle ACKs
- Sender hält Retransmission Timer pro Paket
 - bei Timeout wird das zugehörige Paket noch einmal übertragen

Anmerkung

Bei Go-Back-N kann der Timeout auch für das letzte "offene" Paket laufen (z.B: Kurose). Der Sender muss dann die Sendezeitpunkte aller Pakete speichern, um den Timer neu zu initialisieren.

Go-Back-N

sender window (N=4)

0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8

sender

send pkt0
send pkt1
send pkt2
send pkt3
(wait)

0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8

rcv ack0, send pkt4
rcv ack1, send pkt5

*pkt 2 timeout*

0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8

send pkt2
send pkt3
send pkt4
send pkt5

receiver

receive pkt0, send ack0
receive pkt1, send ack1

receive pkt3, discard,
(re)send ack1

receive pkt4, discard,
(re)send ack1

receive pkt5, discard,
(re)send ack1

rcv pkt2, deliver, send ack2
rcv pkt3, deliver, send ack3
rcv pkt4, deliver, send ack4
rcv pkt5, deliver, send ack5

Selective Repeat

sender window (N=4)

0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8

sender

send pkt0
send pkt1
send pkt2
send pkt3
(wait)

receiver

receive pkt0, send ack0
receive pkt1, send ack1

receive pkt3, buffer,
send ack3

receive pkt4, buffer,
send ack4
receive pkt5, buffer,
send ack5

0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8

rcv ack0, send pkt4
rcv ack1, send pkt5

record ack3 arrived



pkt 2 timeout

send pkt2

record ack4 arrived
record ack4 arrived

0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8
0	1	2	3	4	5	6	7	8

rcv pkt2; deliver pkt2,
pkt3, pkt4, pkt5; send ack2

Q: what happens when ack2 arrives?

4.1 Multiplexing

4.2 UDP

4.3 TCP

4.3.1 Übersicht

4.3.2 Kernfunktionalität

4.3.3 Datenübertragung

4.3.3.1 Paketgröße – Maximum Segment Size

4.3.3.2 Datenflussteuerung – Flow Control

4.3.3.3 Überlaststeuerung – Congestion Control nach RFC 5681

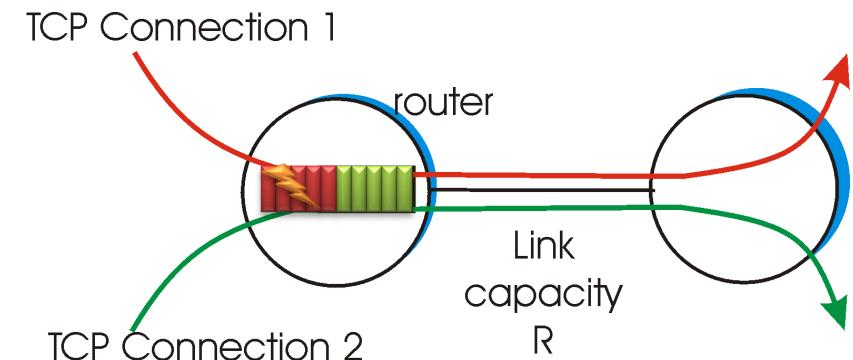
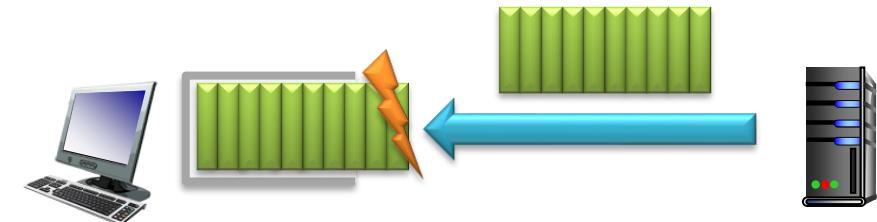
4.3.3.4 Fairness

4.3.3.5 Installierte TCP Varianten

4.4 Zusammenfassung

Wie viele Daten dürfen am Anfang gesendet werden?

- Ziel:
 - Sender möchte Daten möglichst schnell übertragen
- Einschränkungen:
 - Sender darf Empfänger nicht überfrachten
 - Sender darf Netzwerk-Bottleneck nicht überlasten, sonst gibt es Paketverluste
 - Sender soll sich fair gegenüber anderen TCP-Verbindungen verhalten
 - Bottleneck-Bandbreite soll fair zwischen allen TCP-Verbindungen aufgeteilt werden



TCP – Congestion Control (Überlaststeuerung)

- Die Datenflusssteuerung bei TCP beruht auf einem fensterbasierten Mechanismus. Durch die Überlaststeuerung (congestion control) wird die Größe des **Sendefenster dynamisch angepasst**. Generell ist bei TCP die klare Trennung zwischen Datenflusssteuerung und Überlaststeuerung schwierig.
- Die Elemente der Congestion Control sind
 - Slow Start
 - Fast Retransmit
 - Fast Recovery
 - Congestion Avoidance

Sendefenster

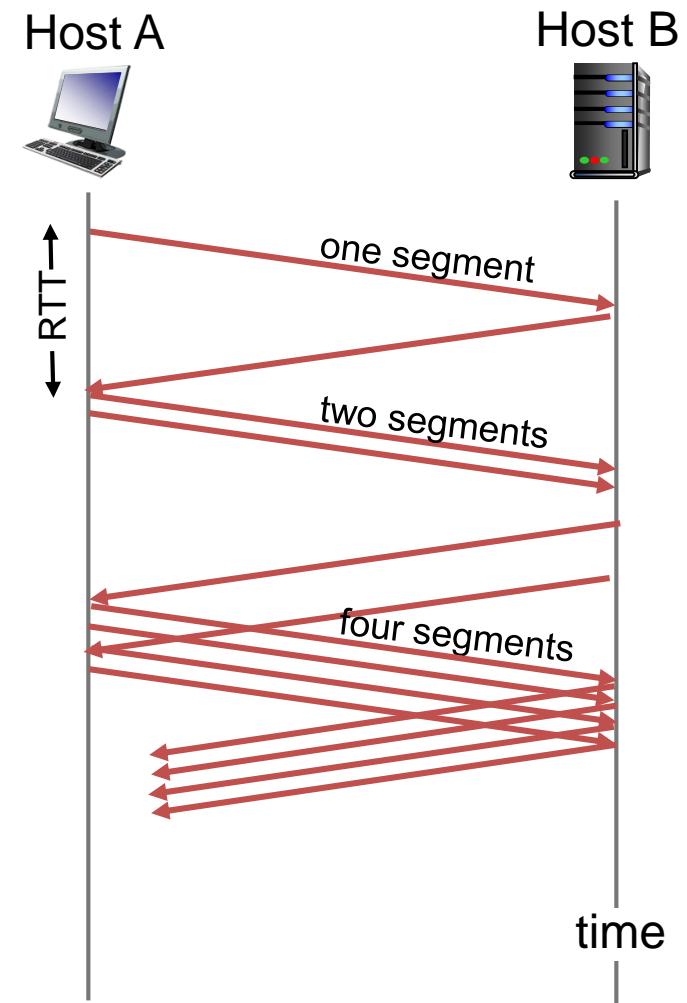
- Das Sendefenster wird bei TCP als **Congestion Window (cwnd)** bezeichnet und gibt die Anzahl der Bytes an, die ohne Acknowledgement unterwegs sein dürfen. Das cwnd wird von der Congestion Control dynamisch angepasst
 - in der Vorlesung und Übung wird zur Vereinfachung das cwnd oftmals in Segmenten und nicht in Bytes angeben
- Das Sendefenster wird nach Aufbau der TCP Verbindung mit dem **Initial Window (IW)** initialisiert. Die Größe des IWs ist ein Betriebssystemparameter. Ein großes IW bewirkt einen schnellen Beginn der Datenübertragung, kann bei einem initialen Paketverlust aber auch zu einem Timeout führen, der mehrere Sekunden dauern kann. Empfehlungen für das IW sind
 - früher: $IW=1$ MSS
 - zwischenzeitlich: $IW=3$ MSS (RFC 3390 2002)
 - heute: bis zu 10 MSS (RFC 6928 in 2013)
- Das **Receive Window (rwnd)** ist eine weitere Obergrenze für die Anzahl der unbestätigten Bytes und wird dem Sender vom Empfänger im TCP Header mitgeteilt. Das rwdn dient dazu, den Sender zu bremsen, falls der Empfänger die Daten nicht schnell genug verarbeiten kann bzw. nicht mehr genug Speicher im Socket zur Verfügung steht. Ein typischer Wert für das rwdn sind 64kBytes bei Windows. Mittlerweile gehen mit der Window Scale Option auch bis zu 16 Megabyte.
- Die **Flightsize** bezeichnet die Anzahl Bytes, die gesendet und noch nicht bestätigt wurden.

Slow Start

- Beim Eintreffen eines ACKs wird das cwnd um die Anzahl der bestätigten Bytes bzw. maximal um MSS erhöht
- Dies bewirkt eine Verdopplung des cwnds pro RTT (Round Trip Time)
 - die RTT ist hier die Zeit vom Versenden des ersten Bytes(Segments) eines Fensters bis zum Empfang des ACKs für das letzte Byte (Segment) eines Sendefensters
 - die RTT ist hier also von der Größe des Sendefensters abhängig



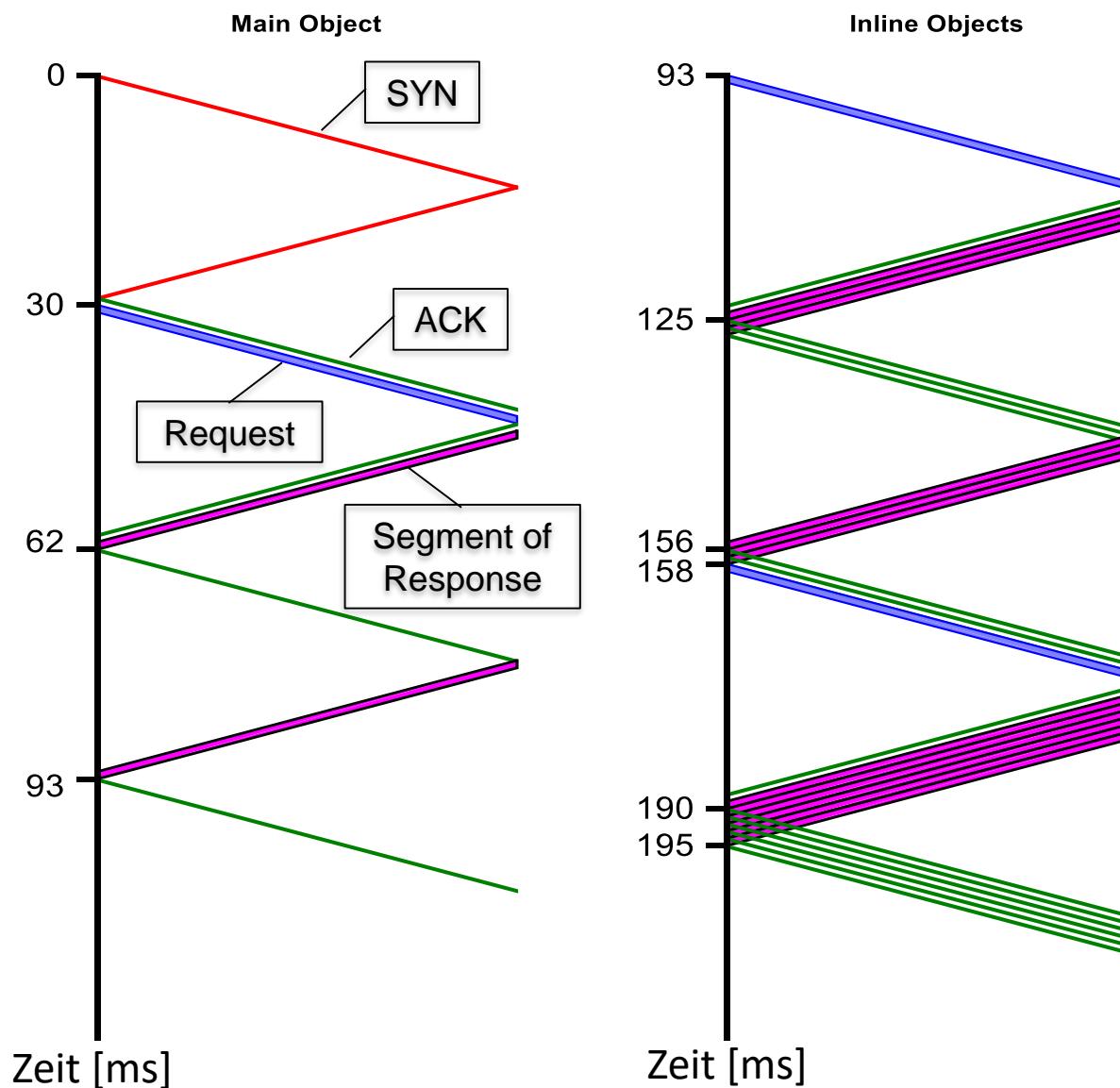
Beispiel für IW=1 MSS



Beispiel für die Übertragung einer Web-Seite

- Web-Seite:
 - HTML Code (Main Object, MO): 3kByte
 - 5 Bilder (Inline Objects 1-5, IO1-IO5): je 9kByte
 - Request für HTML Code oder Inline Objects: 155 Bytes
- TCP:
 - MSS=1500 Byte
 - MO: 2 Segmente
 - IO: 6 Segmente
 - Request: 1 Segment
 - IW=MSS
- Netz:
 - RTT=30ms
 - $T_{tx}(\text{Segment})=1ms$
 - $T_{tx}(\text{ACK})=0ms$
 - $T_{tx}(\text{SYN})=0ms$
 - $T_{tx}(\text{SYN-ACK})=0ms$

Persistent HTTP

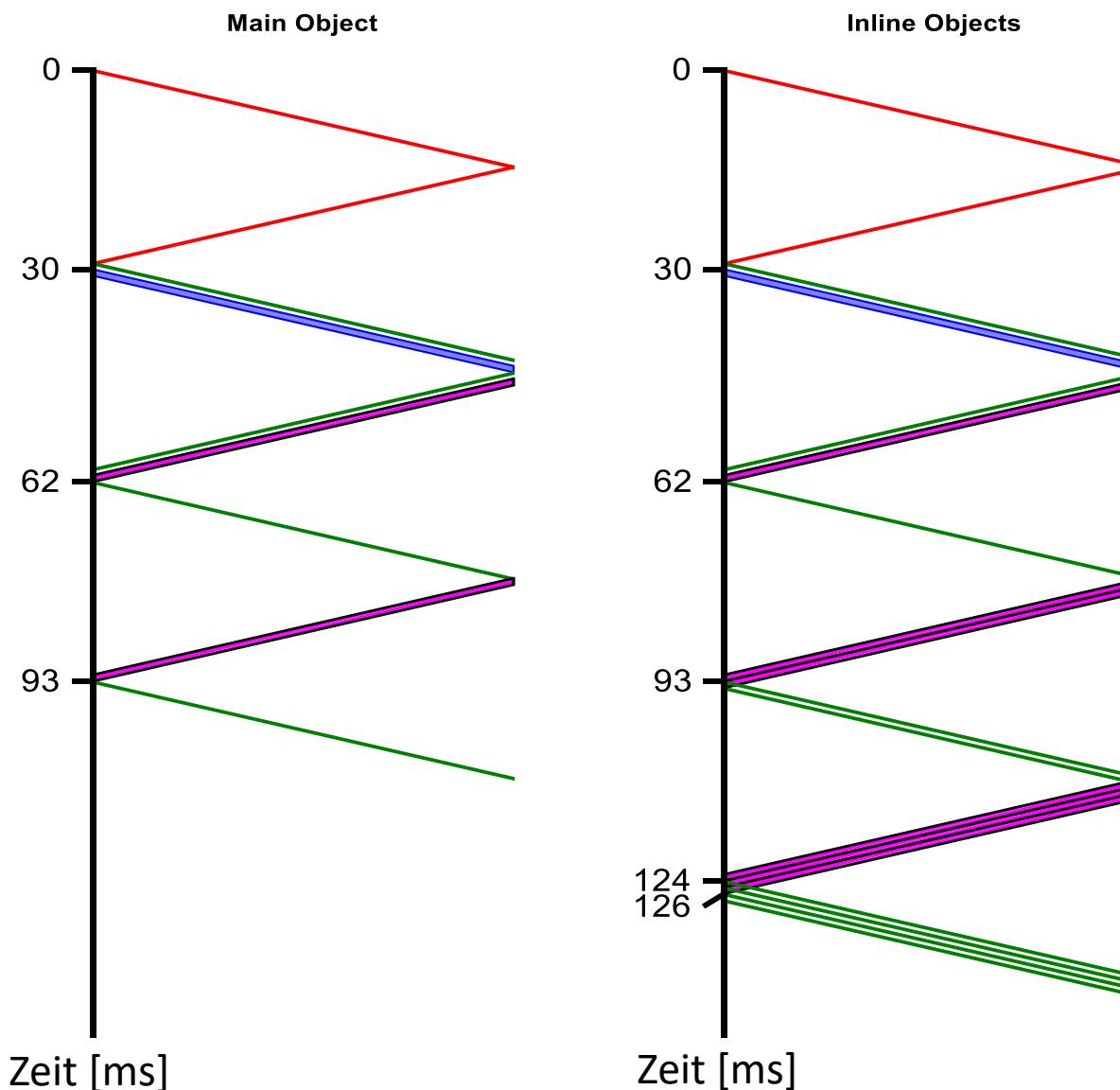


Persistent HTTP

30ms: TCP Verbindungsaufbau (SYN-ACK bei Client) : 15ms+15ms=30ms, → gesendet: Request MO
45ms: TCP Verbindungsaufbau (ACK bei Server) → cwnd=1W=1
46ms: Ankunft Request(MO): 30ms+1ms (T_{tx}) +15ms=46ms → gesendet: MO Segment 1
62ms: Ankunft MO (1. Segment): 46ms+1ms(T_{tx})+15ms=62ms
77ms: Ankunft ACK: 62ms+0ms (T_{tx})+15ms=77ms → cwnd=2 → gesendet: MO Segment 2
93ms: Ankunft MO (2. Segment): 77ms+16ms=93ms
108ms: Ankunft ACK → cwnd=3
109ms: Ankunft Request(IO1) → gesendet: IO1 Segmente 1-3
125ms, 126ms, 127ms: Ankunft IO1 Segmente 1-3
140ms: Ankunft ACK für IO1 Segment 1 → cwnd=4 → gesendet: IO1 Segmente 4 und 5
141ms: Ankunft ACK für IO1 Segment 2 → cwnd=5 → gesendet: IO1 Segment 6
142ms: Ankunft ACK für IO1 Segment 3 → cwnd=6
156ms, 157ms, 158ms: Ankunft IO1 Segmente 4-6 → gesendet: Request IO2
171ms, 172ms, 173ms: Ankunft ACK für IO1 (4-6) → cwnd: 9
174ms: Ankunft Request IO2 → gesendet: IO2 Segmente 1-6
190ms-195ms: Ankunft IO2 Segmente 1-6 → gesendet: Request IO3
200ms-210ms: Ankunft ACK für IO2 Segmente 1-6 → cwnd: 15
211ms: Ankunft Request IO3 → gesendet: IO3 Segmente 1-6
227ms-232ms: Ankunft IO3 Segmente 1-6 → gesendet: Request IO4
...
→ ab IO2: Zeit von Request bis Erhalt Response: 37ms (IO2: 158ms-195ms, IO3: 195ms-232ms, ...)

Gesamt: 30ms (TCP Handshake)+63ms (MO)+65ms (IO1)+4*37ms(IO2-IO5)=306ms

Non-Persistent HTTP



Non-Persistent HTTP

Main Object:

30ms: TCP Verbindungsaufbau (SYN-ACK bei Client) : 15ms+15ms=30ms, → gesendet: Request MO
45ms: TCP Verbindungsaufbau (ACK bei Server) → cwnd=lW=1
46ms: Ankunft Request(MO): 30ms+1ms (T_{tx}) +15ms=46ms → gesendet: MO Segment 1
62ms: Ankunft MO (1. Segment): 46ms+1ms(T_{tx})+15ms=62ms
77ms: Ankunft ACK: 62ms+0ms (T_{tx})+15ms=77ms → cwnd=2 → gesendet: MO Segment 2
93ms: Ankunft MO (2. Segment): 77ms+16ms=93ms

Inline Object:

30ms: TCP Verbindungsaufbau (SYN-ACK bei Client) : → gesendet: Request IO
45ms: TCP Verbindungsaufbau (ACK bei Server) → cwnd=lW=1
46ms: Ankunft Request IO → gesendet: IO Segment 1
62ms: Ankunft MO Segment 1
77ms: Ankunft ACK für IO Segment 1 → cwnd=2 → gesendet: IO Segmente 2 und 3
93ms, 94ms: Ankunft IO Segment 2 und 3
108ms: Ankunft ACK für IO Segment 2 → cwnd=3 → gesendet: IO Segmente 4 und 5
109ms: Ankunft ACK für IO Segment 3 → cwnd=4 → gesendet: IO Segment 6
124ms, 125ms, 126ms: Ankunft IO Segmente 4-6

Gesamt: 93ms (MO)+5*126ms (IO1-5)=723ms

Congestion Avoidance

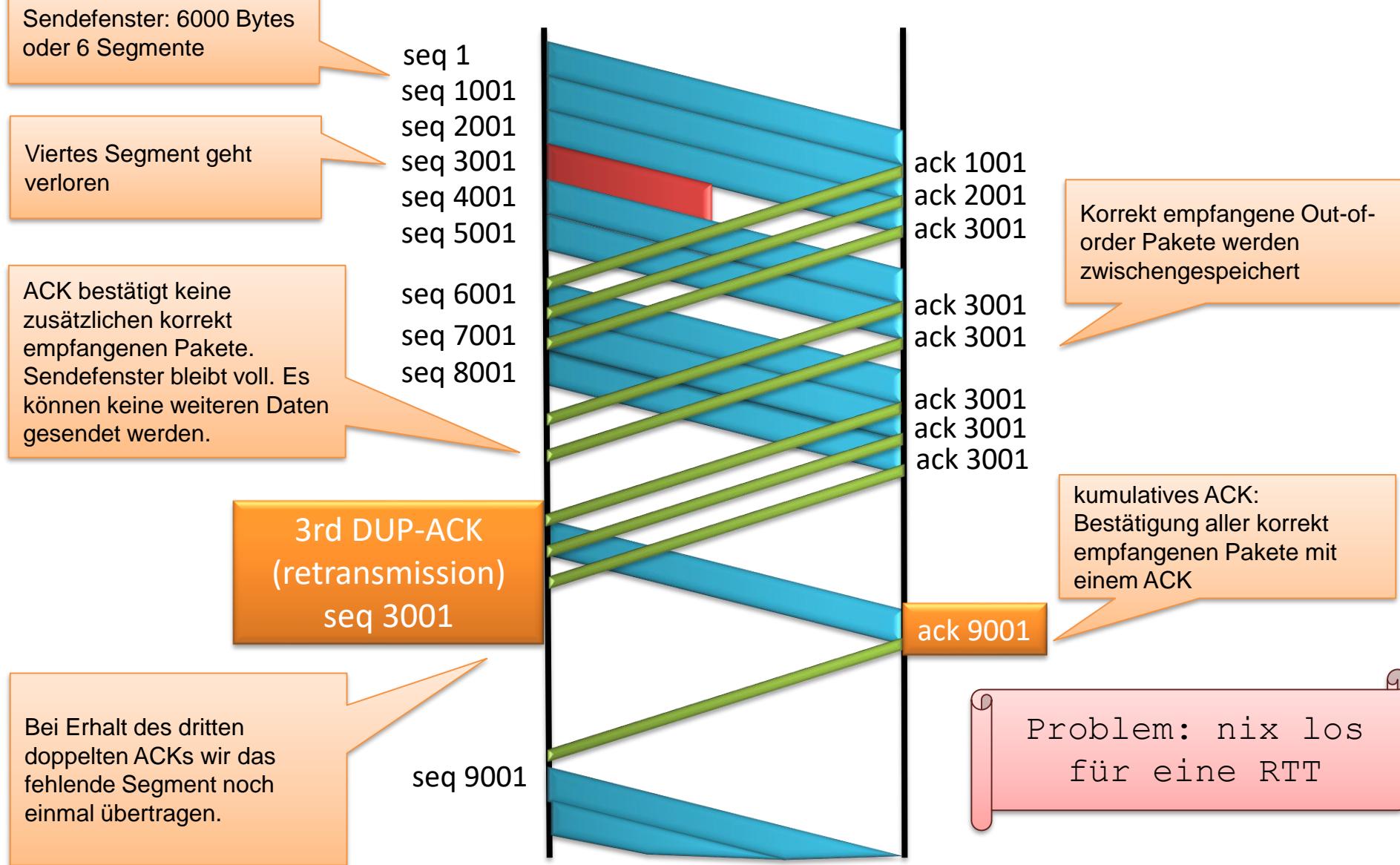
- In der Congestion Avoidance Phase wächst das cwnd langsamer als in der Slow Start Phase. Das cwnd wird pro RTT um die MSS erhöht.
- Dies kann realisiert werden, indem der Sender sich jeweils das erste Byte (Segment) eines neuen Sendefensters merkt und das cwnd bei Eintreffen der Bestätigung für dieses Byte (Segment) erhöht. Der Sender merkt sich danach wieder das nächste gesendete Byte (Segment). In den Beispielen und der Übung wird dieses Segment als CA-Segment bezeichnet,
- Der Übergang von Slow Start nach Congestion Avoidance wird über den Parameter ssthresh (Slow Start Threshold) kontrolliert
 - $cwnd < ssthresh$: slow start
 - $cwnd \geq ssthresh$: congestion avoidance
- Der Parameter ssthresh wird typischerweise mit einem sehr großen Wert initialisiert z.B. dem maximal möglichen Wert für rwnd und bei Eintreten eines Paketverlusts reduziert.

Reaktion auf Paketverluste

Bei TCP gibt es zwei Möglichkeiten, den Verlust eines Pakets zu erkennen:

- **Retransmission Timeout**: bei jedem Versenden eines Paket wird der Retransmission Timer neu gesetzt. Sind beim Auslaufen des Timers nicht alle gesendeten Pakete bestätigt, so tritt ein Retransmission Timeout ein und das erste unbestätigte Segment wird erneut übertragen, d.h. es wird eine Retransmission durchgeführt. Zudem beginnt nach einem Retransmission Timeout eine **neue Slow Start Phase** mit
 - $cwnd=1 \text{ MSS}$ und $ssthresh=\max(\text{FlightSize}/2, 2 \text{ MSS})$
- **3 Dup-ACKs** (duplicate ACKs, doppelte ACKs): der Empfänger bestätigt immer das nächste benötigte Byte. Tritt ein Paketverlust auf, aber es kommen aus der Reihe weitere Segmente an, so wird für jedes dieser Segmente ein ACK gesendet. Da das nächste benötigte Byte unverändert ist, werden also „doppelte ACKs“ gesendet. Das Eintreffen von 3 DupACKs wird als Anzeichen für einen Paketverlust gewertet und das fehlende Paket wird erneut übertragen. Das wird als **Fast Retransmit** bezeichnet. Nach einem Fast Retransmit startet erst eine **Fast Recovery Phase** und darauf eine **Congestion Avoidance Phase** mit halbem $cwnd$.

Fast Retransmit (Beispiel mit Sendefenstergröße 6)



Fast Retransmit und Fast Recovery

- Die Fast Recovery Phase startet nach einem Fast Retransmit oder genauer **beim Eintreffen des ersten Dup-ACKs**. Die Fast Recovery Phase dient dazu, dass nach einem Fast Retransmit weiter Pakete übertragen werden können und nicht auf das Eintreffen des ACKs für das erneut übertragene Paket gewartet werden muss.
- Die Idee von Fast Recovery ist, dass das Eintreffen von DupACKs zeigt, dass die Verbindung noch funktioniert und weitere Segmente gesendet werden können. Pro eingetroffenem DupACK dürfen weitere Daten gesendet werden. Dies wird als Aufblasen des Sendefensters (**congestion window inflation**) bezeichnet.
 - 1. DupACK: 1 neues Segment übertragen, cwnd bleibt gleich
 - 2. DupACK: 1 neues Segment übertragen, cwnd bleibt gleich
 - 3. DupACK: $ssthresh = \max(\text{FlightSize}/2, 2 \text{ MSS})$; $cwnd = ssthresh + 3 \text{ MSS}$; Retransmission; weitere Übertragungen nach cwnd
 - jedes weitere DupACK: $cwnd = cwnd + \text{MSS}$; Übertragungen nach cwnd
 - neues ACK: $cwnd = ssthresh$, weiter mit Congestion Avoidance

Fast Retransmit und Fast Recovery

(in der Vorlesung animiert)

1. DupAck: Segment gesendet, cwnd bleibt gleich

2. DupAck: Segment gesendet, cwnd bleibt gleich

3. DupAck: Retransmission,
ssthresh=4000 (FlightSize/2), cwnd=7000
(ssthresh+3 MSS)

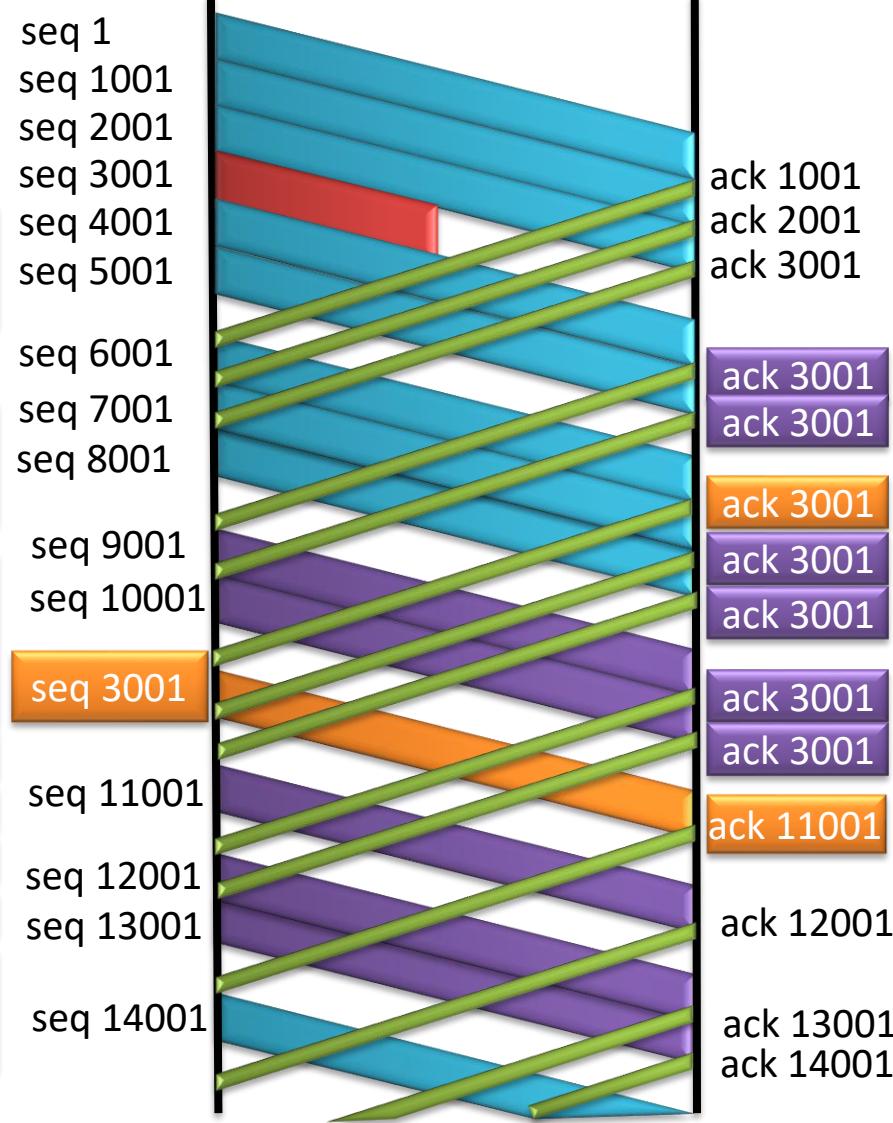
4. DupAck: cwnd=8000 → bis 11000

5. DupAck: cwnd=9000 → bis 12000

6. DupAck: cwnd=10000 → bis 13000

7. DupAck: cwnd=11000 → bis 14000

New Ack: cwnd=4000 (cwnd=ssthresh)
→ Bis 15000



Fast Retransmit und Fast Recovery (mit Übersicht des Sendefensters)

Skript

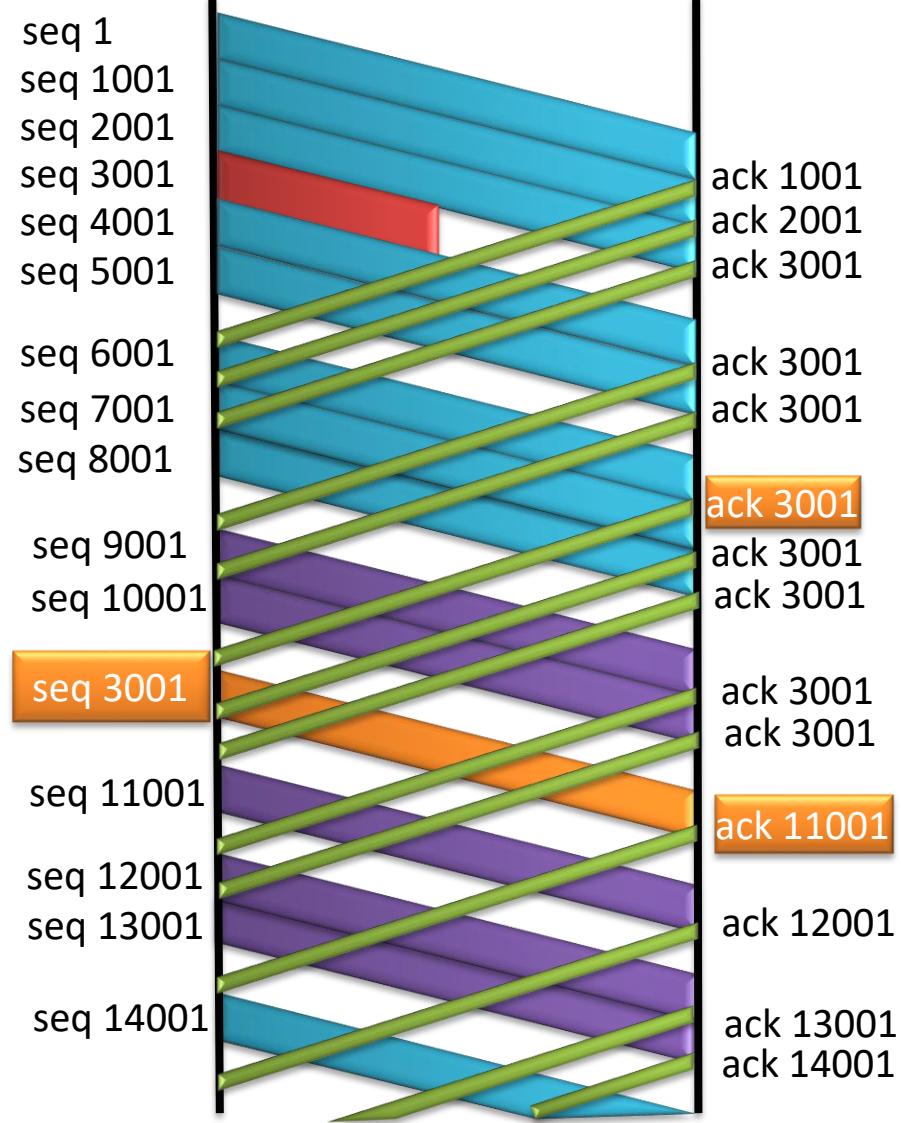
Sendefenster

1. und 2. DupAcks:
Segment gesendet,
cwnd bleibt gleich.

3. DupAck:
Retransmission,
 $ssthresh=4000$
(FlightSize/2),
 $cwnd=7000$
($ssthresh+3$ MSS)

New Ack: $cwnd=4000$
($cwnd=ssthresh$)

Größe	Start	Ende
6000	1	6000
6000	1001	7000
6000	2001	8000
6000	3001	9000
6000	3001	9000
7000	3001	10000
8000	3001	11000
9000	3001	12000
10000	3001	13000
11000	3001	14000
4000	11001	15000



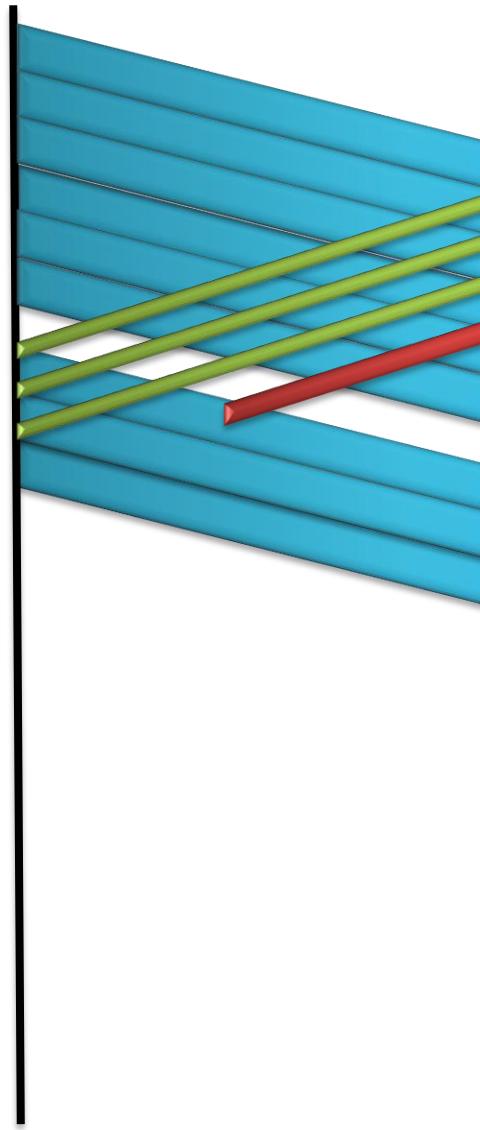
Auch ACKs können verloren gehen

(Beispiel ohne Veränderung des Sendefensters von 6 Segmenten)

Sendefenster: 6000 Bytes
oder 6 Segmente

seq 1
seq 1001
seq 2001
seq 3001
seq 4001
seq 5001

seq 6001
seq 7001
seq 8001



ack 1001
ack 2001
ack 3001
ack 4001

ack 5001

ACK für Bytes bis
einschließlich Byte 4000
geht verloren

Was passiert?

Auch ACKs können verloren gehen

(Beispiel ohne Veränderung des Sendefensters von 6 Segmenten)

Sendefenster: 6000 Bytes
oder 6 Segmente

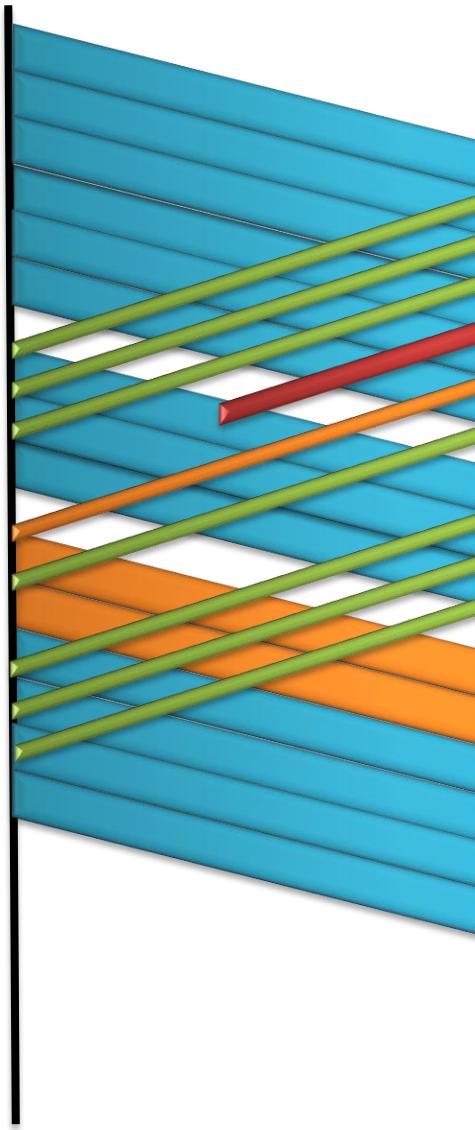
Sendefenster ausgeschöpft:
• ACK: 3000
• Sendefenster: 6000
• Letztes Segment: 8001-9000

Können 2 Segmente senden:
• ACK: 5001
• Sendefenster: 6000
• Letztes Paket: 8001-9000
• Pakete: 9001-10000 und
10001-11000

seq 1
seq 1001
seq 2001
seq 3001
seq 4001
seq 5001

seq 6001
seq 7001
seq 8001

seq 9001
seq 10001
seq 11001
seq 12001
seq 13001
seq 14001



ack 1001
ack 2001
ack 3001
ack 4001

ack 5001
ack 6001

ack 7001
ack 8001
ack 9001

Anmerkung:

In der Darstellung wird das nächste Paket erst gesendet, nachdem das vorige Paket übertragen wurde (siehe Übertragungsverzögerung). Tatsächlich sieht TCP keine Übertragungsverzögerung, da die Segmente direkt in einen Socket zwischen TCP und IP geschrieben und dann von IP übertragen werden. Wenn das Sendefenster TCP erlaubt zwei Segmente zu schreiben, so werden diese beiden Segmente unmittelbar nacheinander in den Socket geschrieben.

Zusammenfassung (in Segmenten)

Slow Start und Congestion Avoidance (CA):

- ssthresh (slow start threshold)
- ab ssthresh nur noch lineares Wachstum (congestion avoidance)
- nach Verbindungsaufbau:
 - cwnd=IW
- bei Erhalt eines ACKs
 - cwnd<ssthresh (Slow Start)
 $cwnd=cwnd+1$
 - cwnd>=ssthresh (CA)
 $cwnd=cwnd+1$ (wenn ACK für CA Segment)
- nach Übergang in Congestion Avoidance Phase aus Slow Start oder Fast Recovery:
 - senden nach cwnd
 - nächstes gesendetes Segment als CA Segment merken

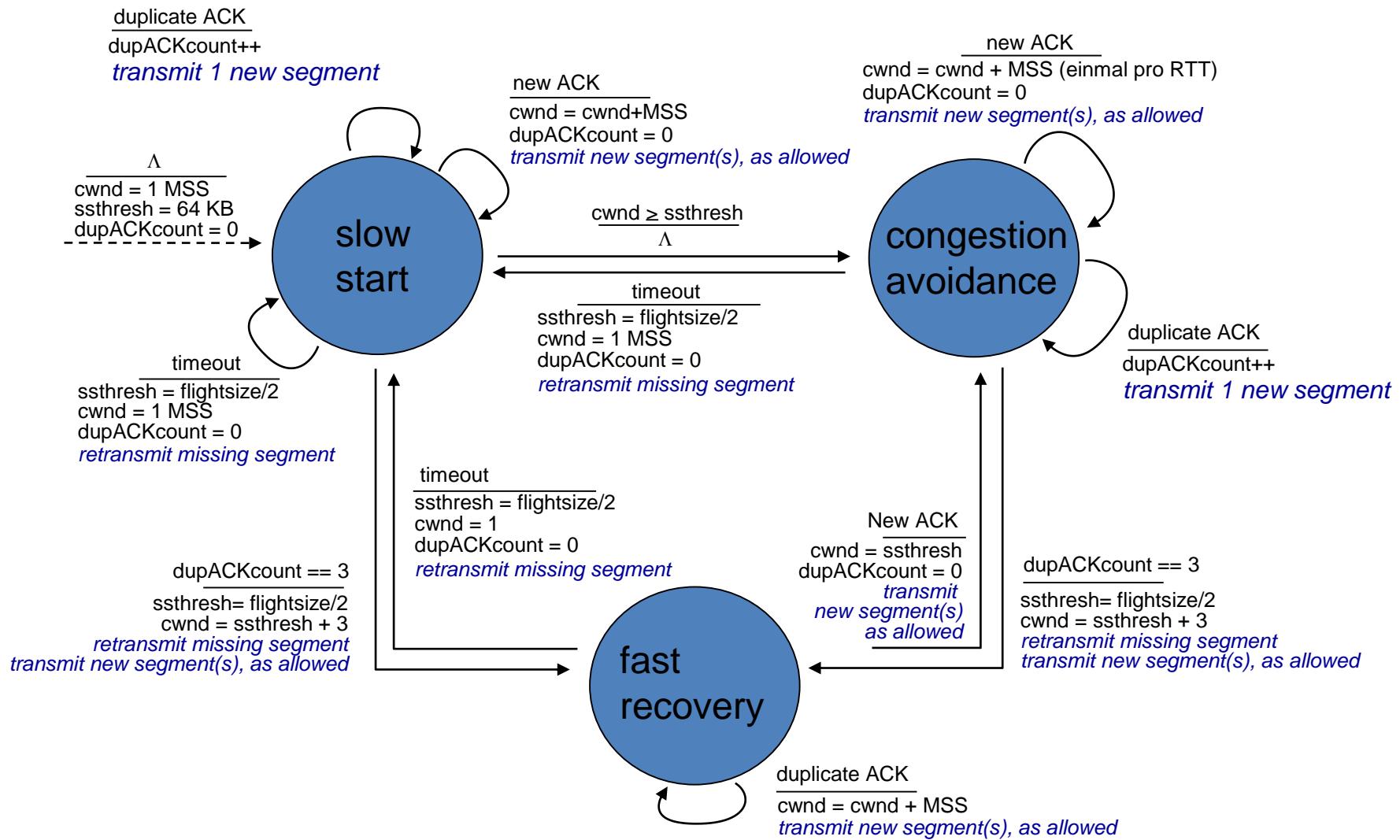
Retransmission Timeout:

- cwnd=1
- ssthresh= $\max(2, \text{flightsize}/2)$
- flightsize=0 (es wird angenommen, das alle Segmente verloren gegangen sind. Diese können aber dennoch bestätigt werden)
- Retransmission

Fast Retransmit mit Fast Recovery

- 1. DupACK: 1 neues Segment übertragen, cwnd bleibt gleich
- 2. DupACK: 1 neues Segment übertragen, cwnd bleibt gleich
- 3. DupACK:
 $ssthresh=\max(2, \text{flightsize}/2);$
 $cwnd=ssthresh+3$; Retransmission
- jedes weitere DupACK: $cwnd=cwnd+1$
- neues ACK: $cwnd=ssthresh$, Congestion Avoidance

Übersicht TCP Congestion Control



Ablauf einer TCP Verbindung

Betrachtet wird die Übertragung von 30 Segmenten über eine TCP Verbindung, wobei

- das Initial Window (IW) ein Segment beträgt
- die Segmente 3, 14, 15 und 17 verloren gehen
- das Congestion Window (cwnd) in Segmenten betrachtet wird
- nur ganze Segmente gesendet werden, auch bei einem nicht-ganzzahligen cwnd
- der Retransmission Timeout (RTO) 5 RTTs beträgt

Notation:

- FlightSize1: Anzahl unbestätigter übertragener Segmente nach Betrachtung der angekommenen ACKs
- FlightSize2: Anzahl unbestätigter übertragener Segmente inklusive der neu gesendeten Segmente

Tabelle

RTT	ACKs An	flightsize 1	DupACK	ssthresh	cwnd	Segmente Ab	flightsize 2	CA Segment	Segmente AN	ACKs Ab
0	x	0	0	64	1	1	1	x	1	1
1	1	0	0	64	2	2-3	2	x	2	2
2	2	0	0	64	3	4-5	3	x	4-5	2x2
3	2x2	3	2	64	3	6-7	5	x	6-7	2x2
4	2	5	3	2,5	5,5	3	5	x	3	7
	2	5	4	2,5	6,5	8	6	x	8	8
5	7	1	0	2,5	2,5	9	2	9	9	9
	8	1	0	2,5	2,5	10	2	9	10	10
6	9	1	0	2,5	3,5	11-12	3	11	11-12	11-12
	10	2	0	2,5	3,5	13	3	11	13	13
7	11-13	0	0	2,5	4,5	14-17	4	14	16	13
8	13	4	1	2,5	4,5	18	5	14	18	13
9	13	5	2	2,5	4,5	19	6	14	19	13
10	13	6	3	3	6	14	6	x	14	14
11	14	5	0	3	3	x				
15	TIMEOUT	0	0	2,5	1	15	1	x	15	16
16	16	0	0	2,5	2	17-18	2	x	17-18	2x19
17	19	0	0	2,5	3	20-22	3	20	20-22	20-22
	19	0	1	2,5	3	23	4	20	23	23
18	20-23	0	0	2,5	4	24-27	4	24	24-27	24-27
19	24-27	0	0	2,5	5	28-30	3	28	28-30	28-30
20	28-30	0	0	2,5	6					

In Moodle finden sie auch eine Excel-Tabelle mit Erklärungen

4.1 Multiplexing

4.2 UDP

4.3 TCP

4.3.1 Übersicht

4.3.2 Kernfunktionalität

4.3.3 Datenübertragung

4.3.3.1 Paketgröße – Maximum Segment Size

4.3.3.2 Datenflussteuerung – Flow Control

4.3.3.3 Überlaststeuerung – Congestion Control nach RFC 5681

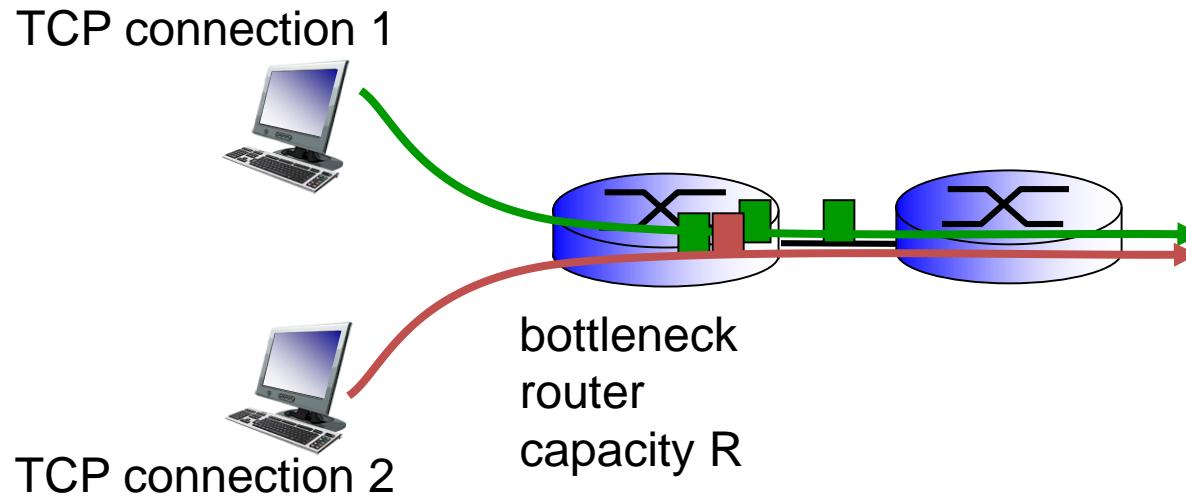
4.3.3.4 Fairness

4.3.3.5 Installierte TCP Varianten

4.4 Zusammenfassung

TCP Fairness

Ideale Fairness: wenn sich K TCP Verbindungen einen Bottleneck-Link mit Bandbreite R teilen, dann sollte jede Verbindung eine Rate von R/K erhalten.

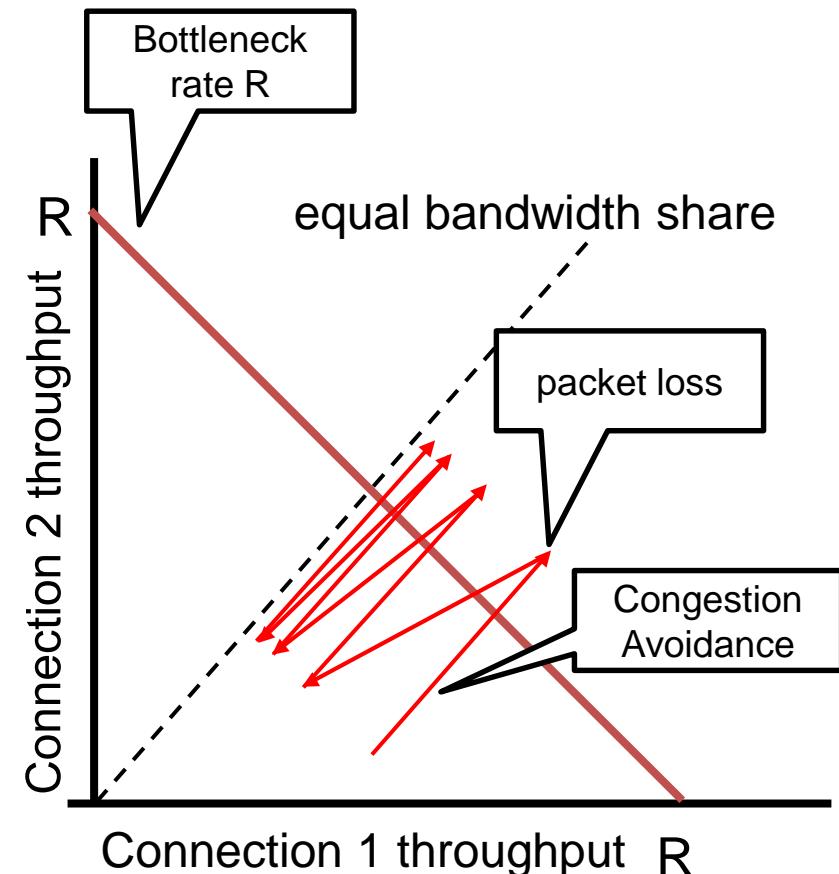


- Idealerweise und auf lange Sicht teilen sich homogene TCP-Verbindungen die Bandbreite fair auf
- Aber nicht auf kurze Sicht und bei heterogenen TCP-Verbindungen

Zwei konkurrierende Verbindungen während Congestion Avoidance:

- Rate wächst mit einer Steigung von 1 pro RTT (**additive increase**)
- proportionale Verringerung durch **multiplicative decrease**

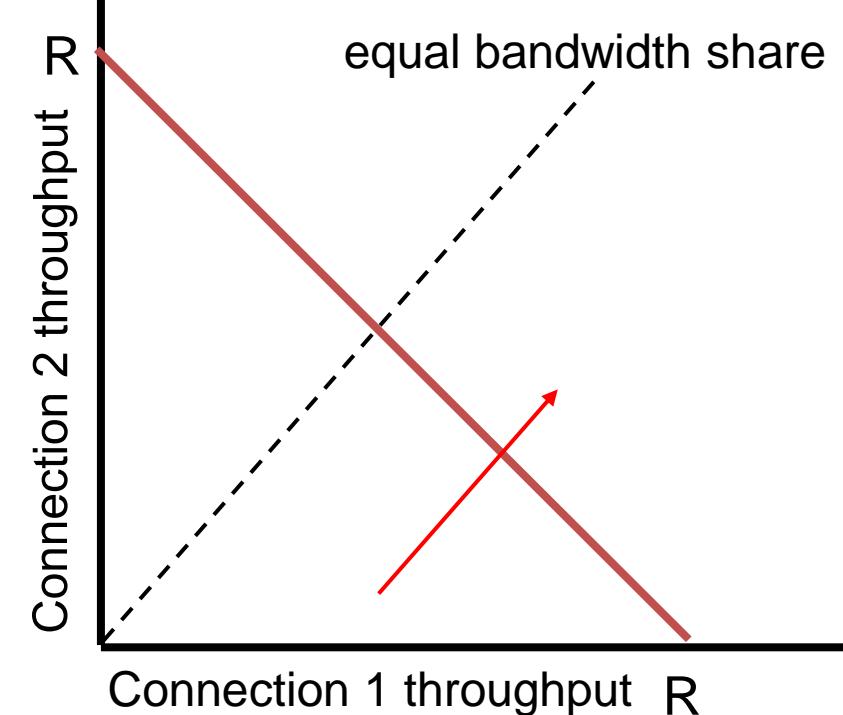
loss: decrease window by factor of 2
congestion avoidance: additive increase



Das Sendefenster beider Verbindungen wächst linear und gleich schnell unabhängig von der absoluten Größe des Sendefensters:

$$\text{cwnd} = \text{cwnd} + \text{MSS} \text{ pro RTT}$$

congestion avoidance: additive increase

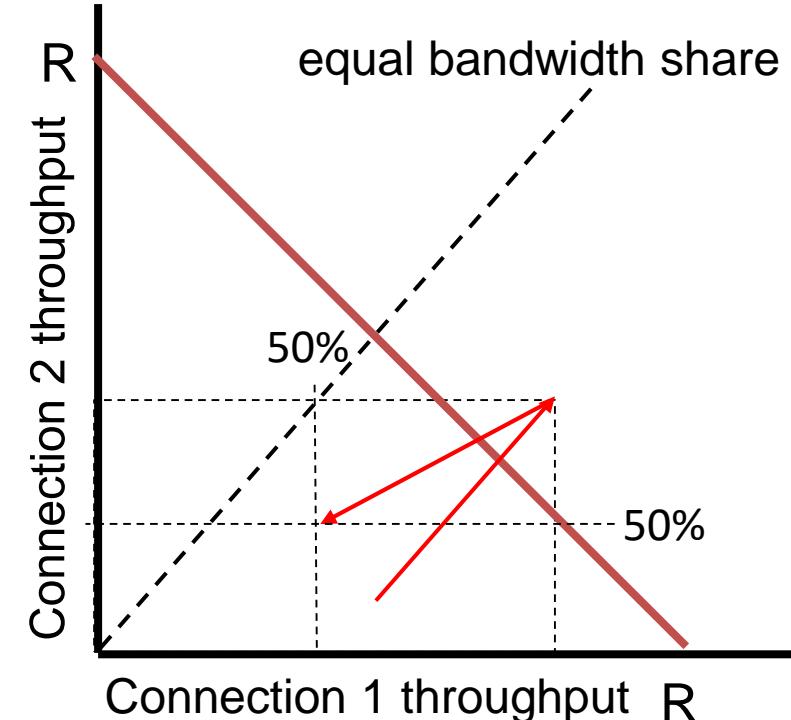


Beide Verbindungen erleiden einen Paketverlust.
Das Sendefenster wird halbiert:

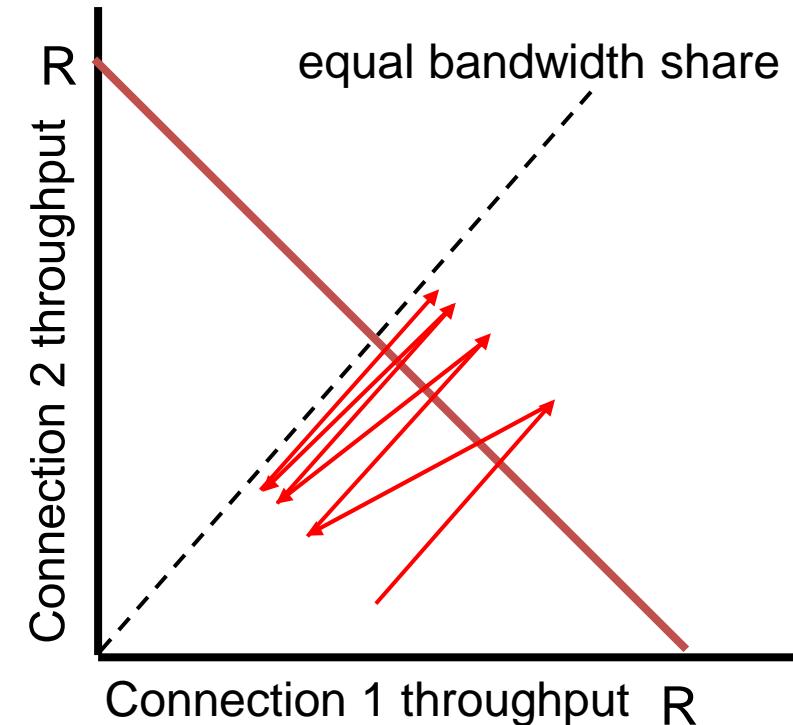
$$\text{cwnd} = \text{cwnd}/2$$

Die Senderate der Verbindung mit größerem Sendefenster wird - absolut gesehen - stärker reduziert als die Verbindung mit kleinerem Sendefenster. Dadurch nähern sich die Sendefenster der beiden Verbindungen an.

loss: decrease window by factor of 2



- Senderate der Verbindung mit größerem Sendefenster wird in absoluten Zahlen stärker reduziert
- Senderaten gleichen sich an



TCP Fairness und Probleme

Fairness und parallele TCP-Verbindungen:

- Eine Anwendung kann zwei oder mehr parallele TCP-Verbindungen öffnen
 - Beispiel: Engpass hat eine Rate von R , bisher existieren neun Verbindungen
 - Neue Anwendung legt eine neue TCP-Verbindung an und erhält die Rate $R/10$
 - Neue Anwendung legt elf neue TCP-Verbindungen an und erhält mehr als $R/2$!
- Faire Aufteilung auf TCP-Verbindungen führt zu Unfairness auf Applikationsebene
- Web-Browser öffnen mehrere Verbindungen, noch extremer sind P2P Anwendungen, die von vielen Peers parallel laden (BitTorrent)

Heterogene TCP-Verbindungen

- Verbindungen mit kurzer RTT gewinnen gegen Verbindungen mit langer RTT
 - Durchsatz: Sendefenster pro RTT
 - TCP Fairness: gleiche Sendefenster
 - zusätzlich: bei kurzer RTT schnelleres Wachstum des Sendefensters
- Mice and Elephant
 - kleine neue TCP Verbindungen (Mice) können sich kaum gegen große existierende TCP Verbindungen (Elephant) durchsetzen
 - Beispiel: Laden einer Web-Seite bei andauerndem Server-Back-Up

LFN (Long Fat Networks):

- haben ein sehr großes Bandwidth-Delay Product
- TCP braucht sehr lange, um ein volles Sendefenster zu erreichen

4.1 Multiplexing

4.2 UDP

4.3 TCP

4.3.1 Übersicht

4.3.2 Kernfunktionalität

4.3.3 Datenübertragung

4.3.3.1 Paketgröße – Maximum Segment Size

4.3.3.2 Datenflusssteuerung – Flow Control

4.3.3.3 Überlaststeuerung – Congestion Control nach RFC 5681

4.3.3.4 Fairness

4.3.3.5 Installierte TCP Varianten

4.4 Zusammenfassung

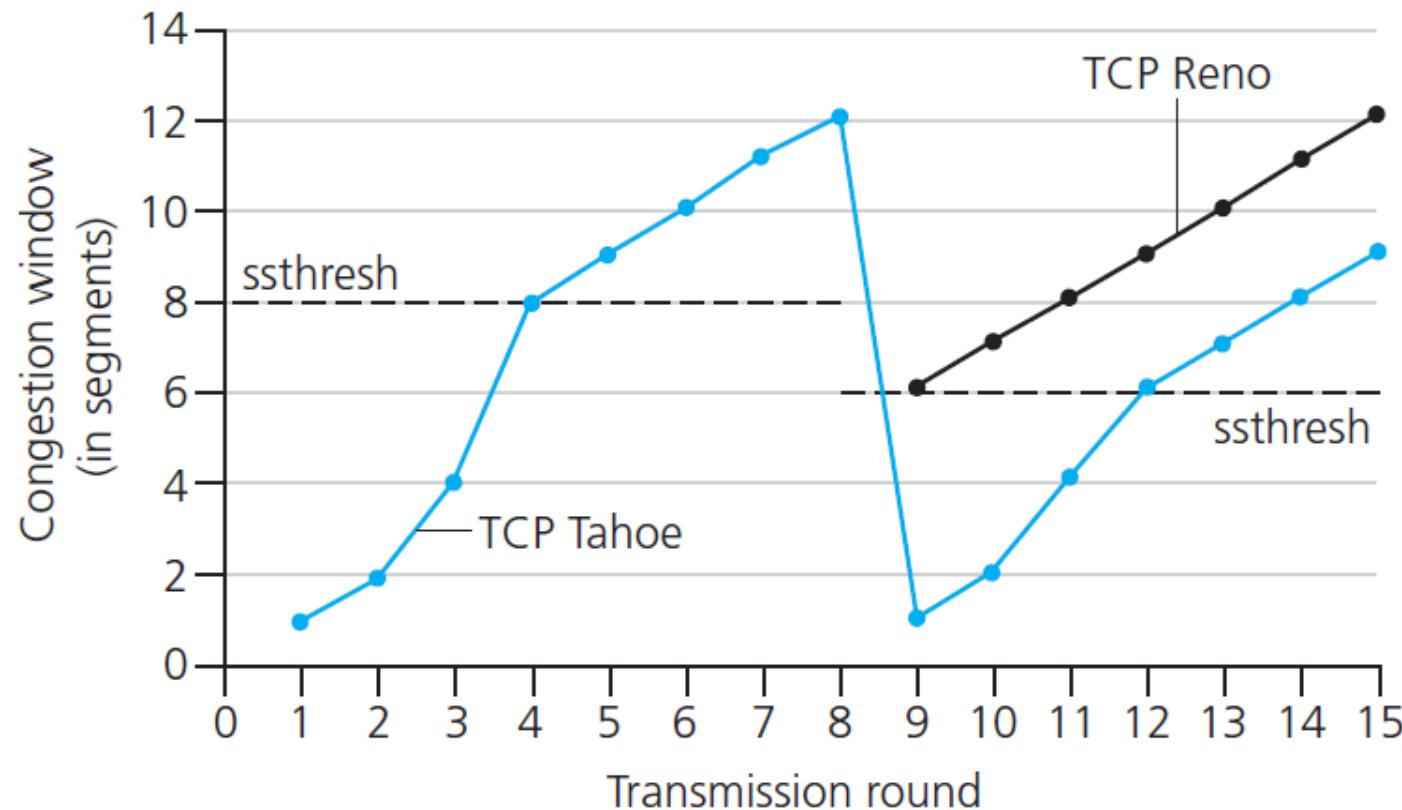
TCP in the Real World

- Populärste TCP Varianten?

- Kernproblem: TCP funktioniert schlecht bei Netzen mit großem Delay-Bandwidth-Produkt (generell: heutige und zukünftige Internetzugänge)
- Compound Congestion Control (Windows)
 - basiert auf Reno , berücksichtigt RTT bei Anpassung von cwnd
- CUBIC (Linux default)
 - momentan populärste Variante
- TCP BBR (Bottleneck Bandwidth and Round-trip propagation time)
 - 2016 von Google vorgestellt
 - in der Entwicklung, große Forschungsaktivität
- QUIC (Quick UDP Internet Connections):
 - von Google getriebene Alternative zu TCP
 - HTTP/2 über TCP → HTTP/3 über QUIC

TCP Tahoe und Reno (ältere TCP Versionen)

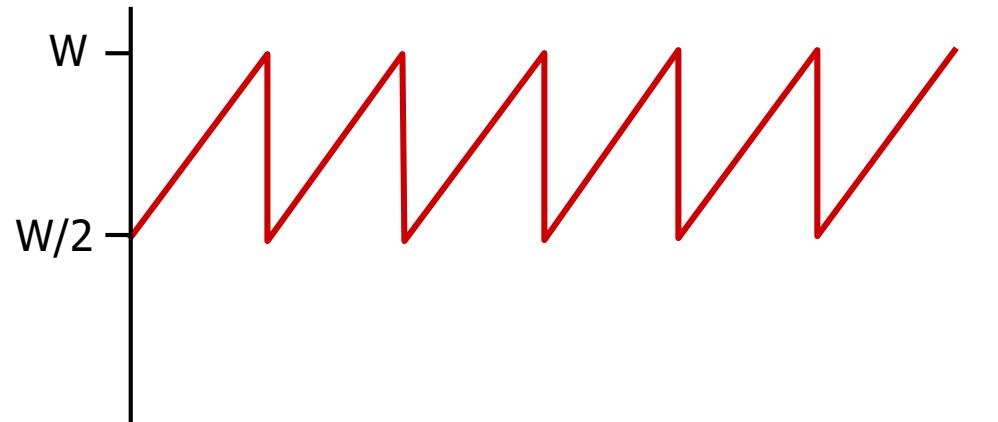
Typischer Verlauf einer TCP Verbindung für die Varianten Tahoe und Reno. Nach einer Slow Start Phase folgt eine Congestion Avoidance Phase. Nach einem Paketverlust bricht das Sendefenster ein. Hier unterscheiden sich Tahoe und Reno. Tahoe beginnt immer mit einem Sendefenster von 1, Reno halbiert das Sendefenster nach einem Fast Retransmit, wie hier dargestellt.



TCP Durchsatz

TCP versucht, sich immer der maximal möglichen Datenrate zu nähern, bis Paketverlust entsteht. Dann wird die Datenrate verringert, indem das Sendefenster verkleinert wird. Generell ist die Datenrate ein Sendefenster pro RTT. Die Kapazität eines Bottlenecks kann nur zu etwa 75% ausgenutzt werden.

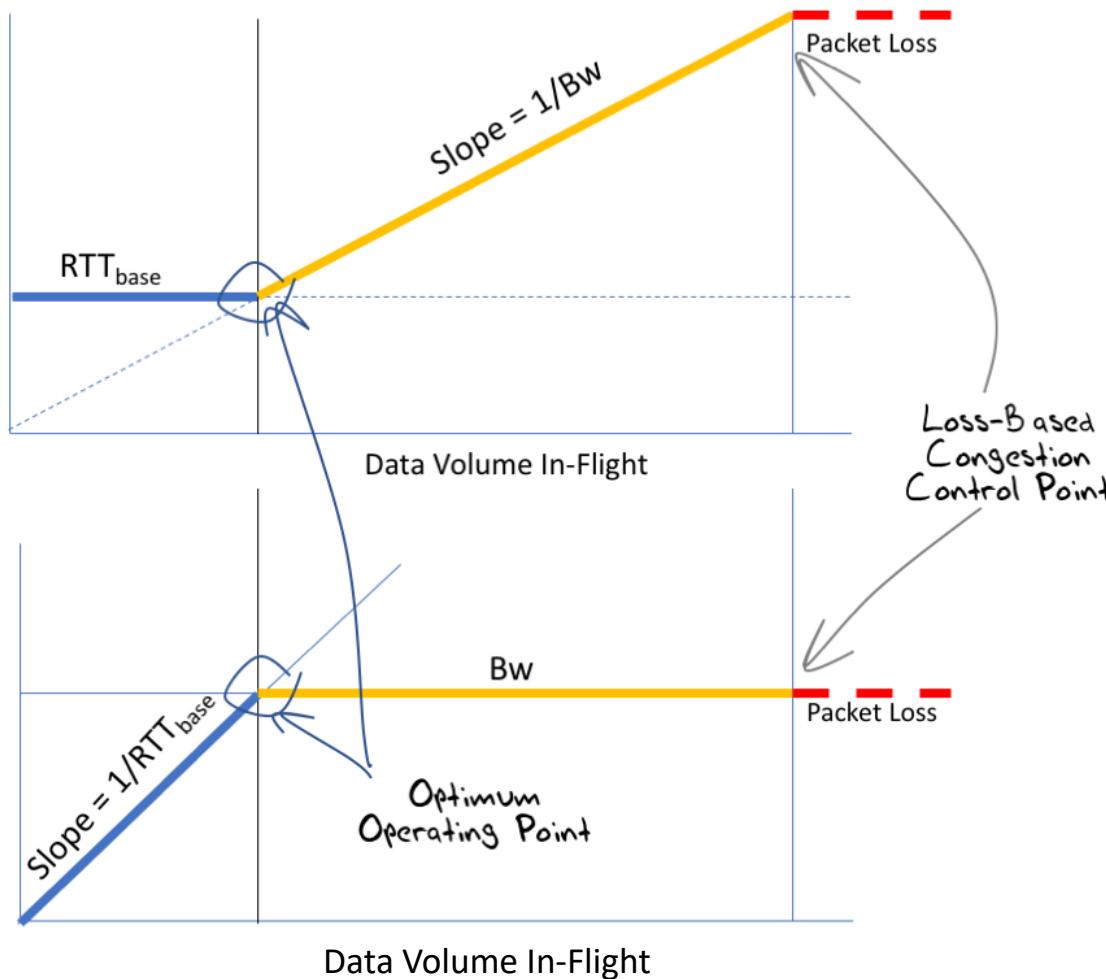
- TCP "Sawtooth" Curve – AIMD
 - additive increase (congestion avoidance): $cwnd = cwnd + MSS * (MSS/cwnd)$
 - multiplicative decrease (packet loss): $cwnd = cwnd / 2$



- TCP Durchsatz: $cwnd/RTT \approx 3/4 * W / RTT$
 - W: Sendefenster, bei dem Paketverlust entsteht

Ziel der Congestion Control Varianten

Round Trip Time



Ziel

- blaue Phase so schnell wie möglich verlassen
- möglichst lang in gelber Phase bleiben
- rote Phase vermeiden

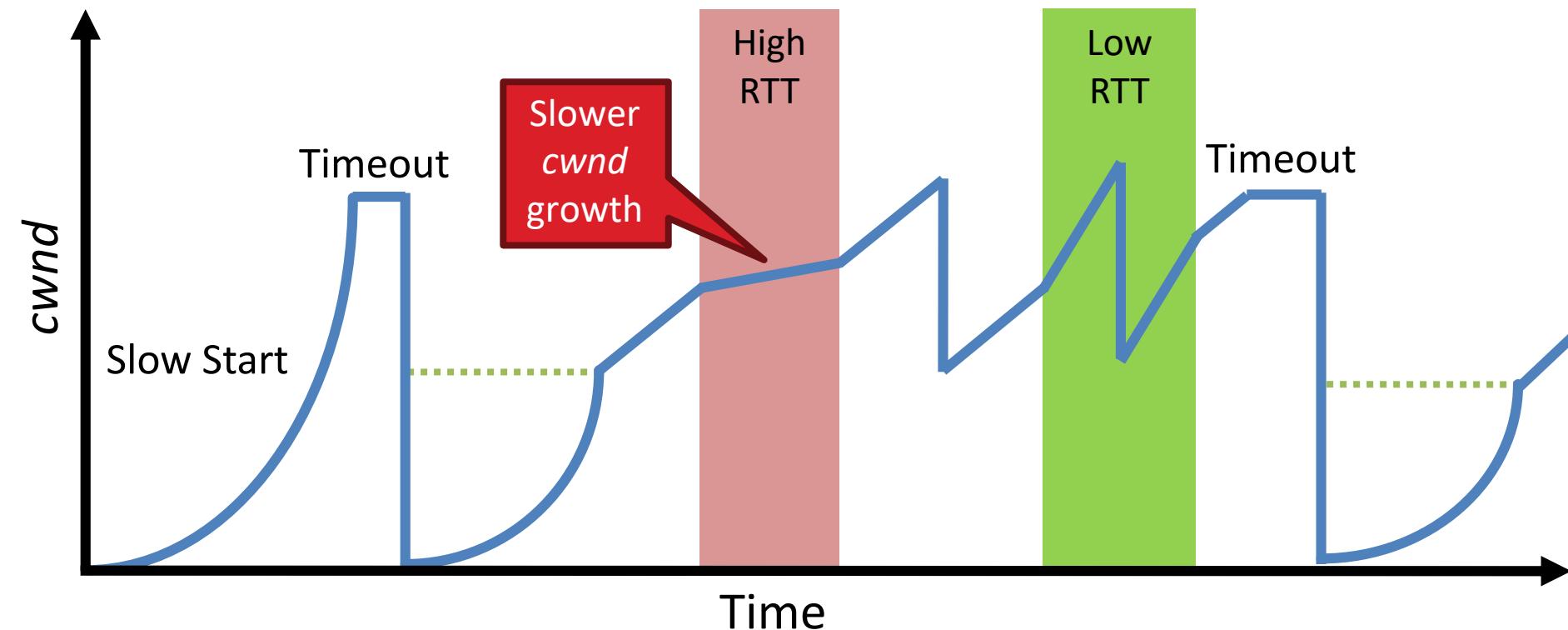
Problem:

- verfügbare Bandbreite verändert sich
- größere Bandbreite: in blauer statt in gelber Phase
- niedrigere Bandbreite: rote Phase statt gelber Phase

Idee:

- Signale (RTT) aus dem Netz interpretieren, um Veränderungen zu erkennen und länger in gelber Phase zu bleiben

Windows: Compound Congestion Control



- Aggressivität durch Veränderungen in der RTT gesteuert
- Vorteil: schneller Anstieg, fairer bei unterschiedlichen RTTs
- Nachteil: genaue RTT Schätzung ist schwierig

- Default TCP Implementierung unter Windows
- Hauptidee: aggressiveres Wachstum als TCP Reno durch zusätzliches Sendefenster (*dwnd*), das über die RTT kontrolliert wird. Es gibt also zwei Sendefenster:
 - *cwnd*: traditionell, basierend auf Paketverlust
 - *dwnd*: zusätzlich, basierend auf gemessener RTT
 - Das GesamtSendefenster (*wnd*) ist
 - $wnd = \min(cwnd + \text{dwnd}, \text{rwnd})$
 - *cwnd* wird über AIMD-Mechanismus kontrolliert
 - *dwnd* ist das Delay-Fenster und erlaubt eine aggressiveres (zusätzliches) Wachstum als traditionelles TCP (Reno)

- Idee:
 - Eine sinkende RTT bedeutet eine kürzere Wartezeit im Buffer und damit, dass der Buffer sich leert. Die Rate des gesamten ankommenden Verkehrs ist also geringer als die Bandbreite des Bottlenecks. Die TCP Verbindung kann das Sendefenster und somit die Datenrate schnell erhöhen. Da der Buffer des Bottlenecks nicht voll ist, drohen auch keine unmittelbaren Paketverluste.
 - Eine steigende RTT bedeutet dagegen, dass der Buffer des Bottleneck-Links voller wird und die Rate des ankommenden Verkehrs die Bottleneck-Kapazität übersteigt. Die TCP-Verbindung erhöht das Sendefenster langsamer, mindestens aber so schnell wie TCP Reno, wenn $dwnd=0$.
- Anpassen des dwnd:
 - Vergrößerung RTT → verkleinere dwnd ($dwnd \geq 0$)
 - Verringerung RTT → vergrößere dwnd
 - Veränderung dwnd ist proportional zur Veränderung der RTT

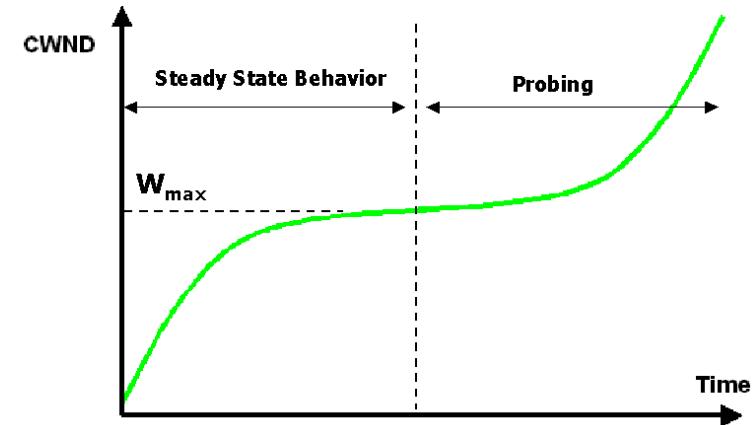
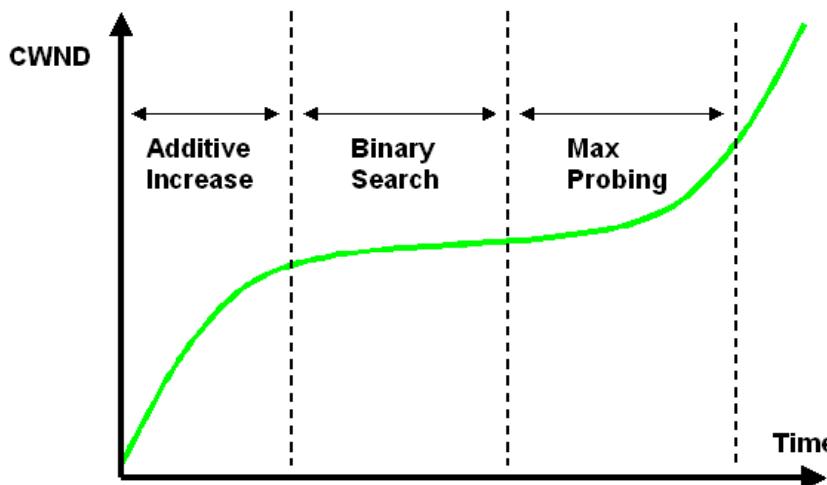
- BIC (Binary Increase Congestion Control) und Cubic (xxx)
- Highspeed-TCP-Varianten: Fairness unabhängig von RTT, gut für LFN
- Cubic ist Standard in Linux

BIC:

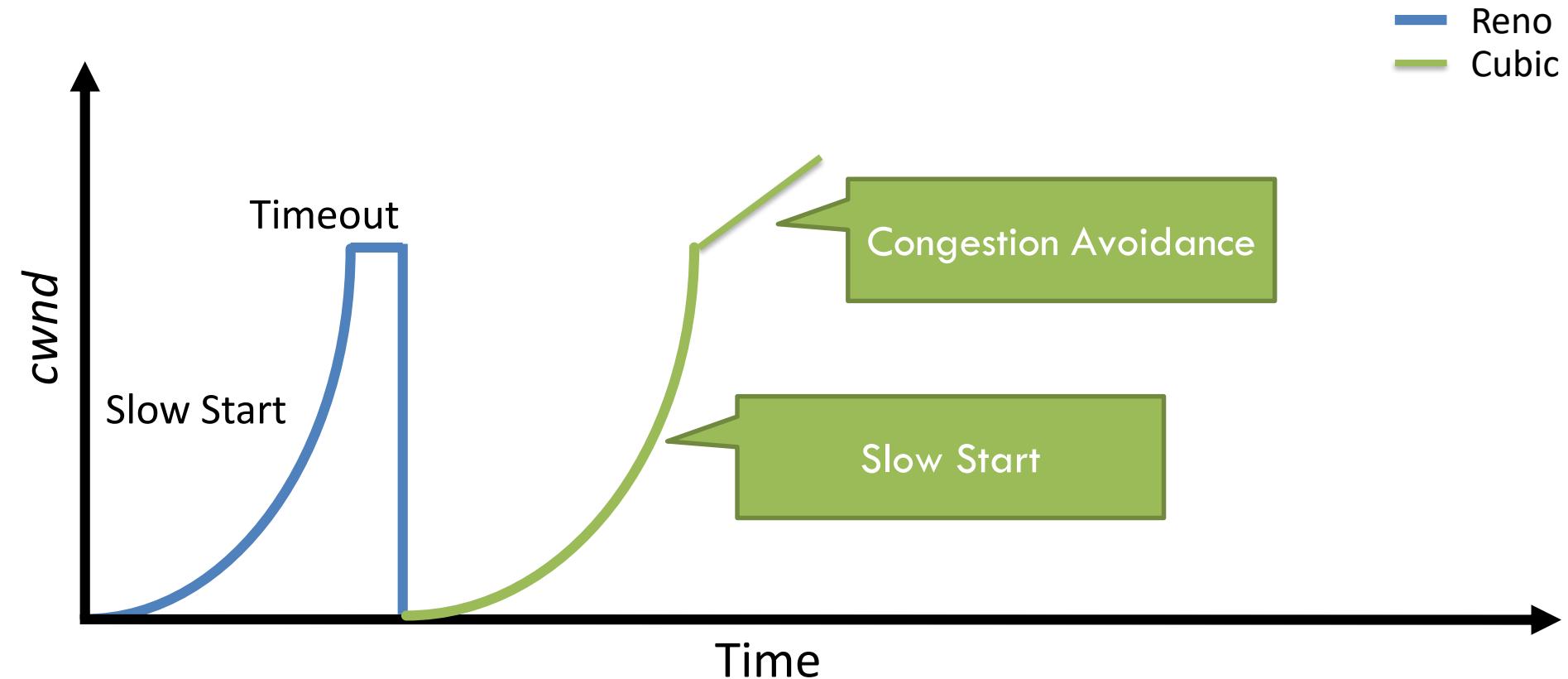
- schnell von reduziertem zu altem Sendefenster aufschließen (binary search)
- zunehmend aggressives probing, um zusätzliche Bandbreite auszuloten

Cubic:

- weniger aggressiv als BIC
- nach Paketverlust schnell auf stabiles Plateau
- langes Steady-state-Behavior
- aggressives Probing

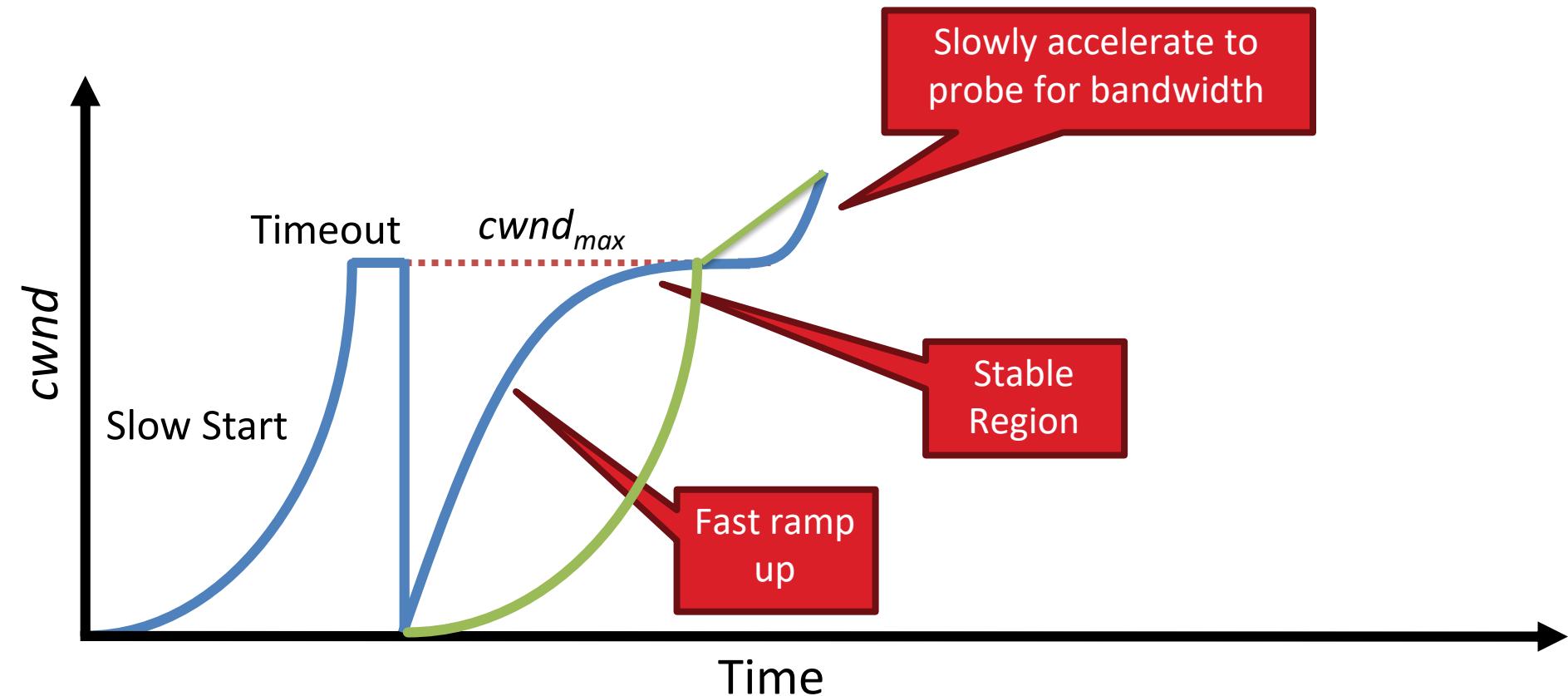


Linux default: Cubic

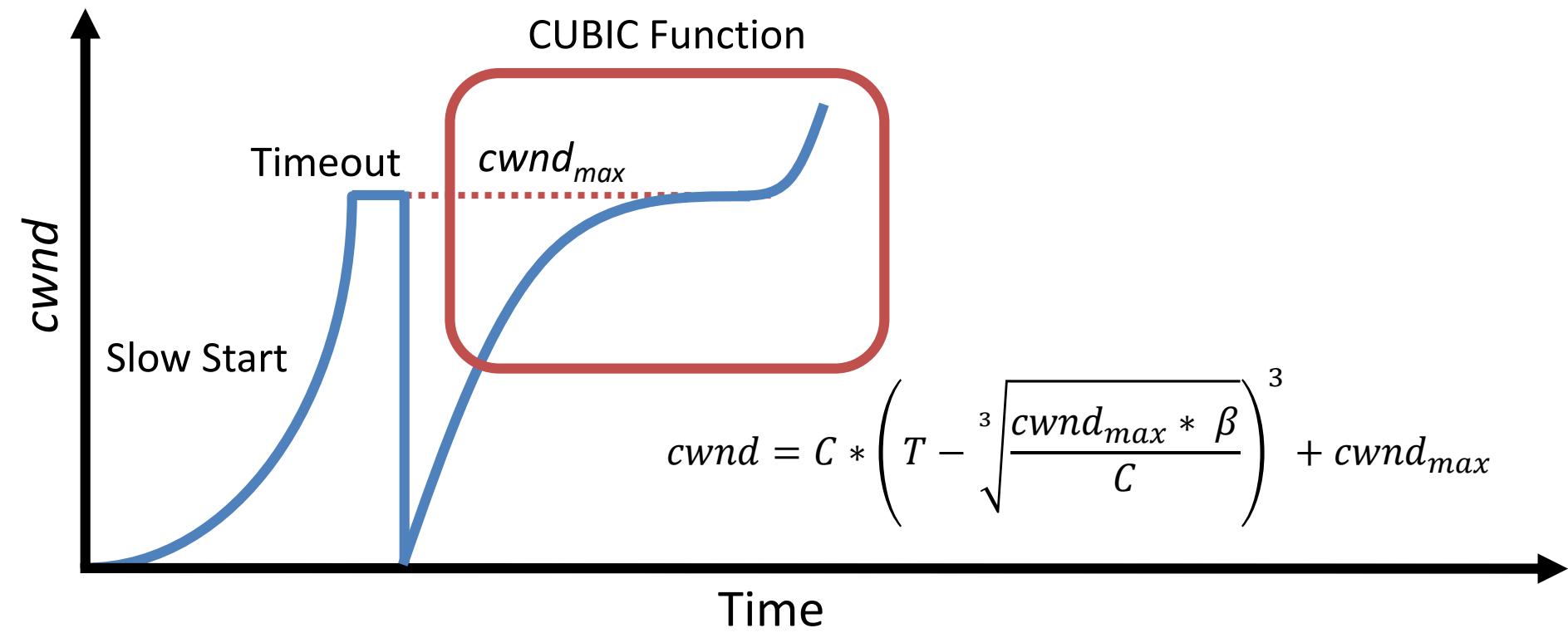


- Idee von Cubic
 - nach Paketverlust Fenster zuerst schnell erhöhen
 - auf Niveau von letztem Paketverlust vorsichtig erhöhen (slow probing)
 - nach Überschreiten des Niveaus wieder aggressiv erhöhen (fast probing)

Linux default: Cubic



- Idee von Cubic
 - nach Paketverlust Fenster zuerst schnell erhöhen
 - auf Niveau von letztem Paketverlust vorsichtig erhöhen (slow probing)
 - nach Überschreiten des Niveaus wieder aggressiv erhöhen (fast probing)



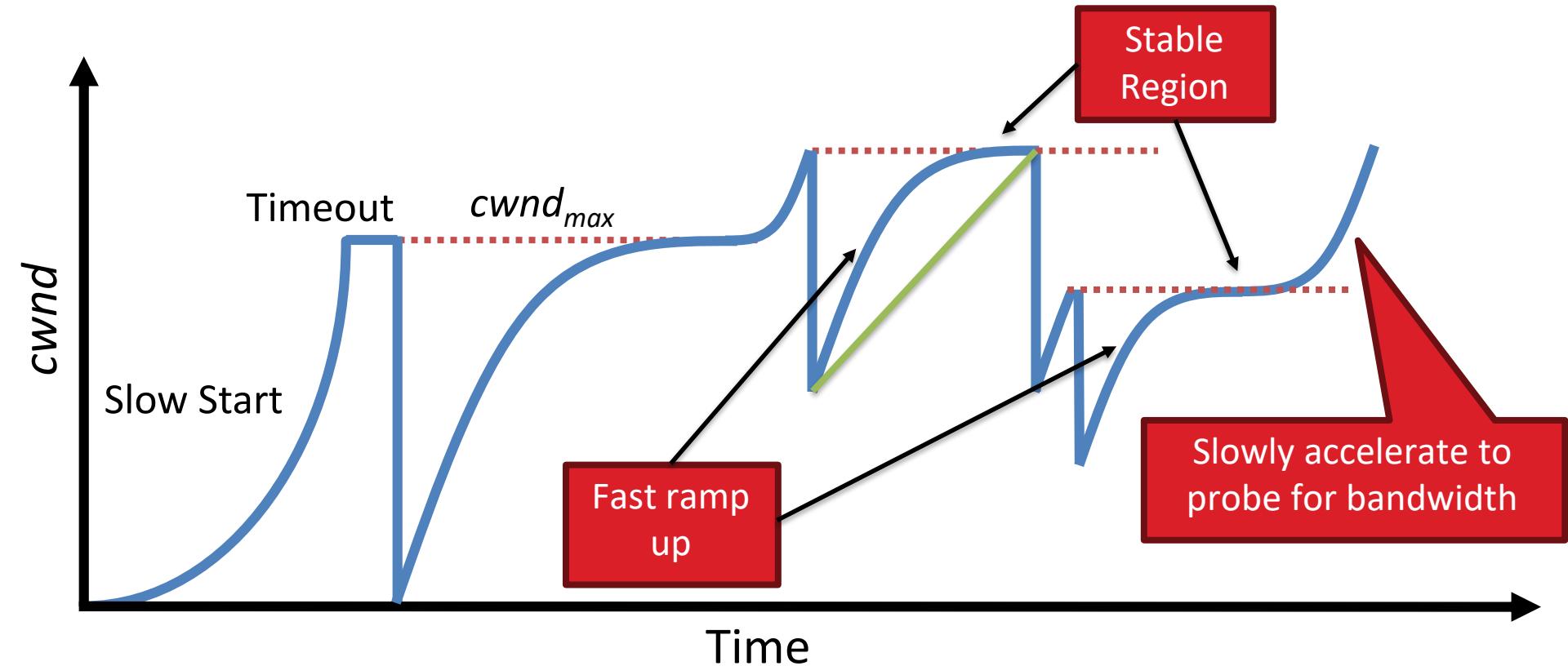
C → constant scaling factor

β → constant fraction for multiplicative decrease

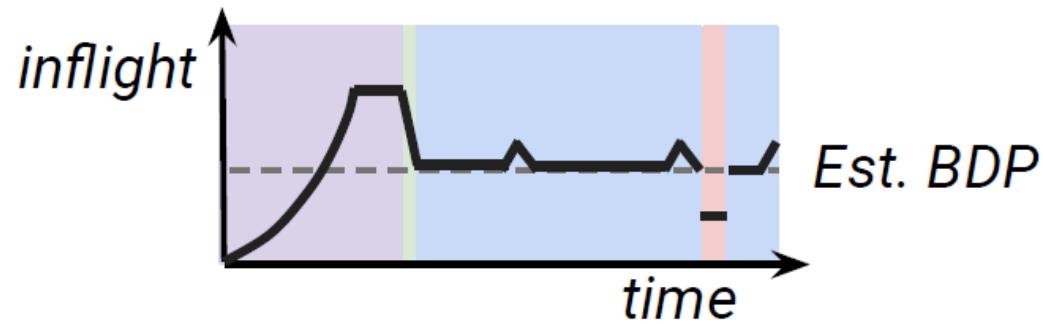
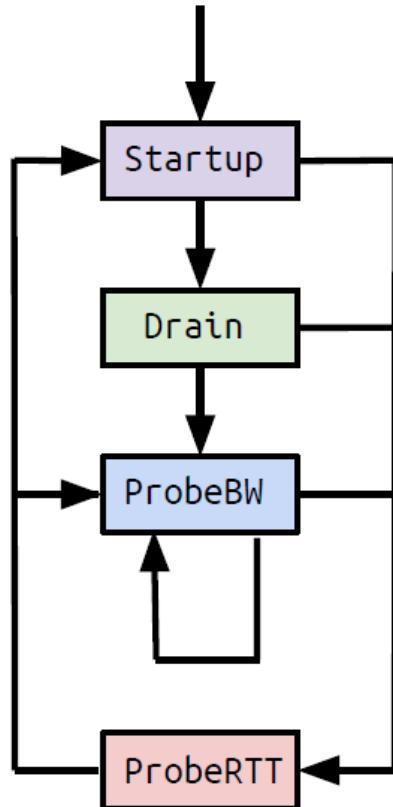
T → time since last packet drop

cwnd_{max} → cwnd when last packet dropped

Linux default: Cubic

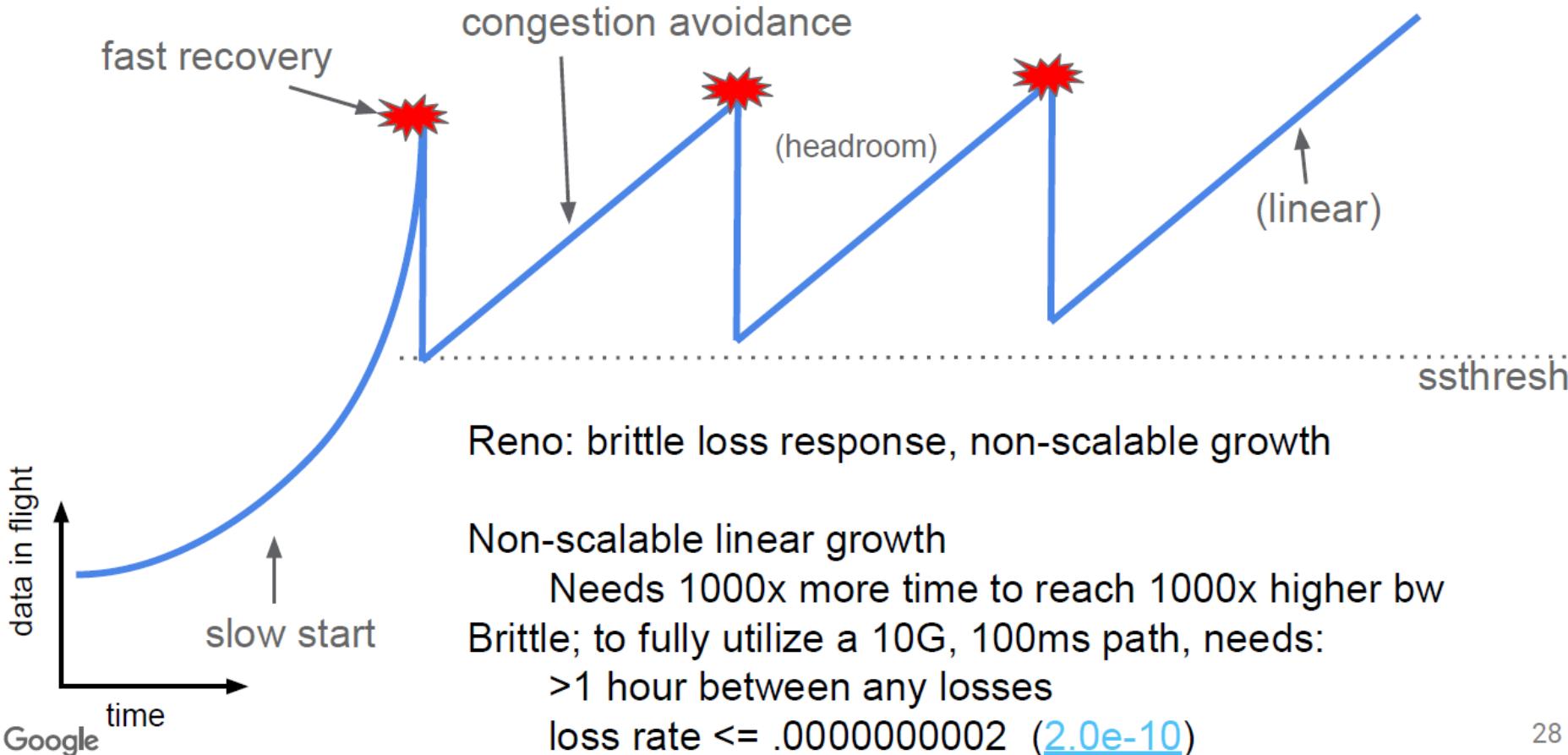


- Idee von Cubic
 - nach Paketverlust Fenster zuerst schnell erhöhen
 - auf Niveau von letztem Paketverlust vorsichtig erhöhen (slow probing)
 - nach Überschreiten des Niveaus wieder aggressiv erhöhen (fast probing)



- Congestion Control Algorithmus bildet ein Modell des Pfades ab
 - Probing Prinzip ähnlich wie Cubic aber komplexer
- Congestion Control über Sendefenster und **Senderate**
 - Sendefenster kann nicht mehr auf einmal übertragen werden
- Experimente zeigen, dass BBRv1 unfair gegenüber Cubic ist
 - BBRv2 ist besser, kann im Einzelfall aber unfair sein
- BBR zeigt bessere Ausnutzung von Long-Fat-Pipes

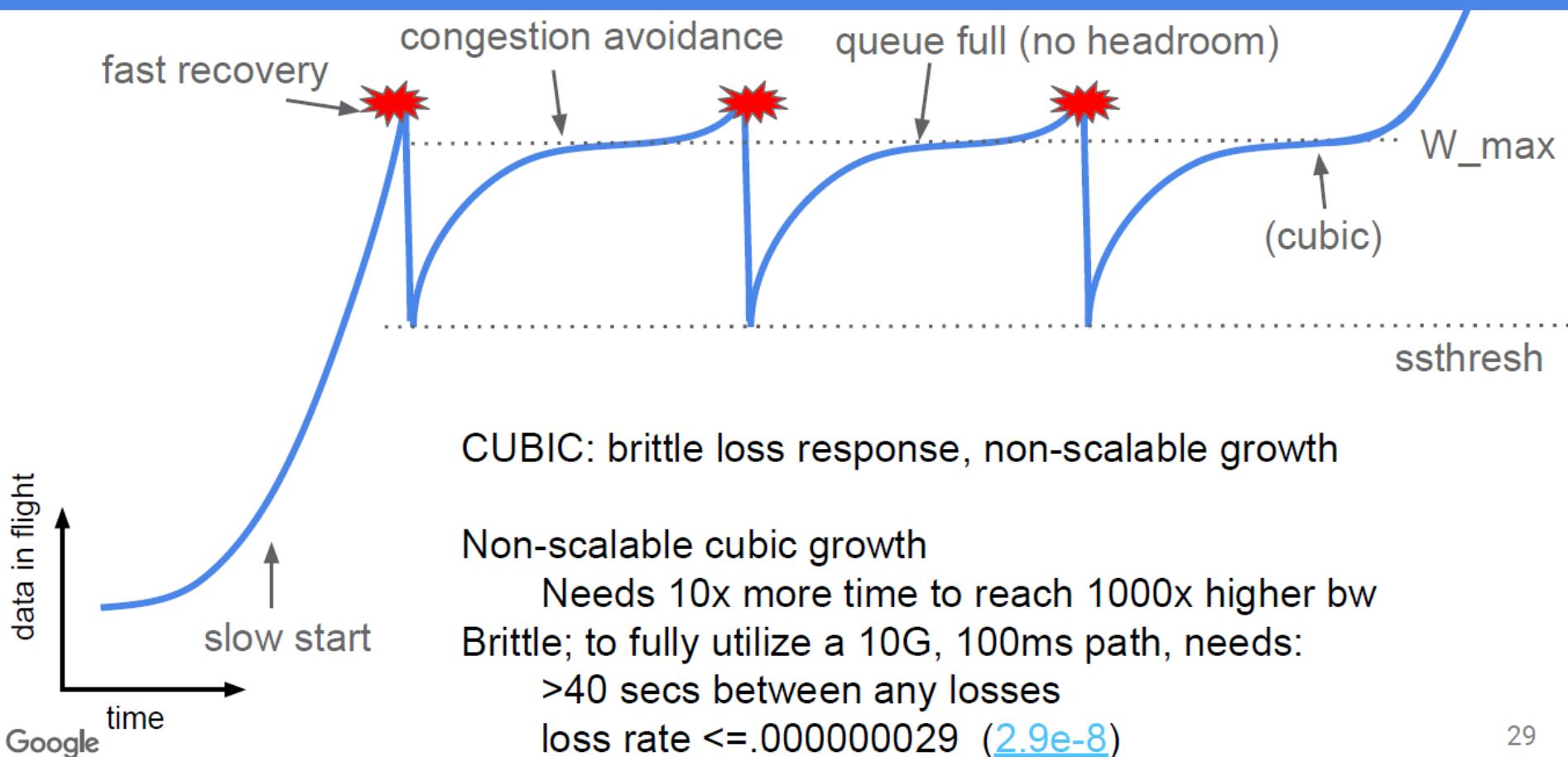
Reno



<https://datatracker.ietf.org/meeting/104/materials/slides-104-icrcg-an-update-on-bbr-00>

Cubic

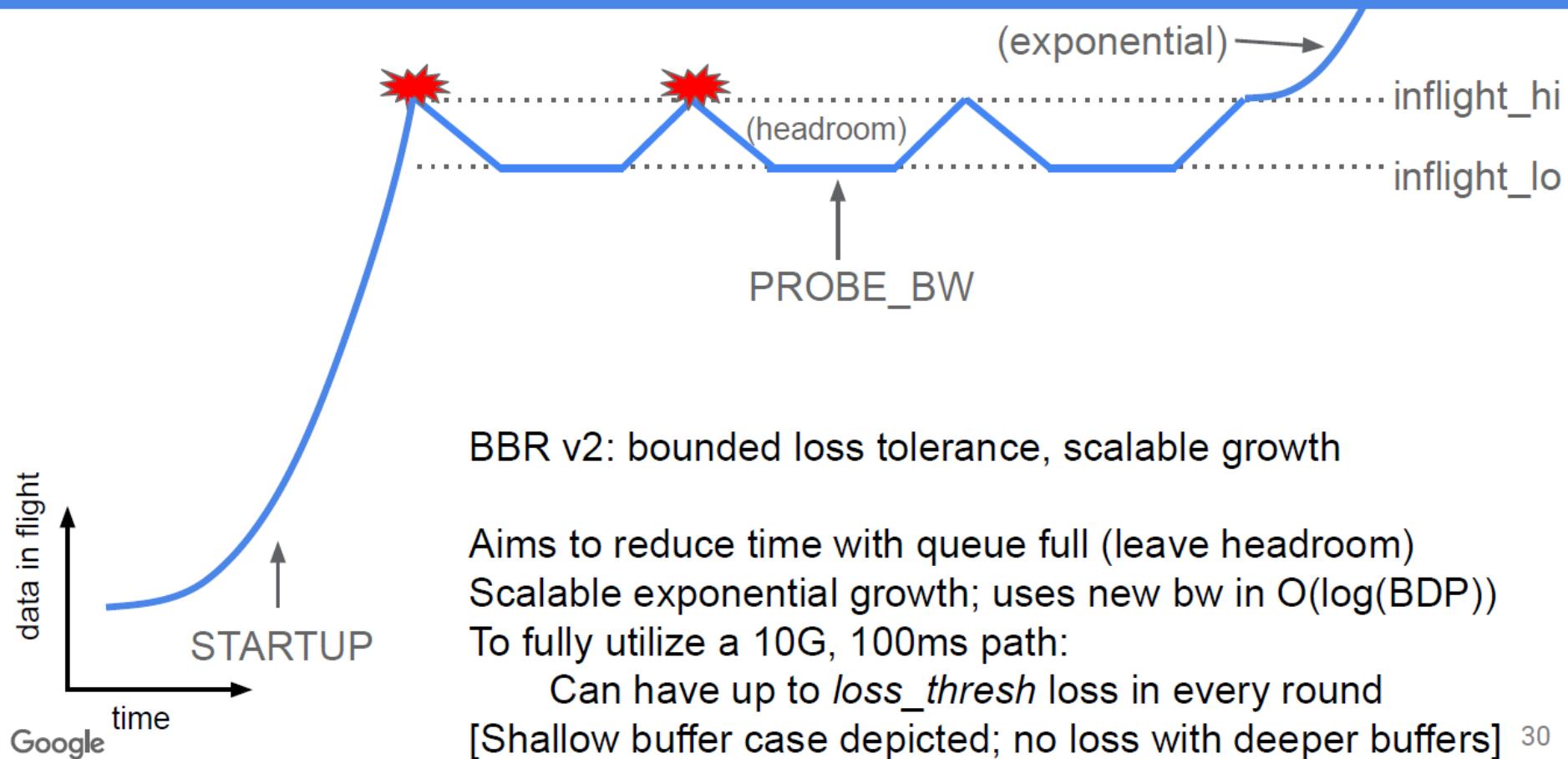
CUBIC



29

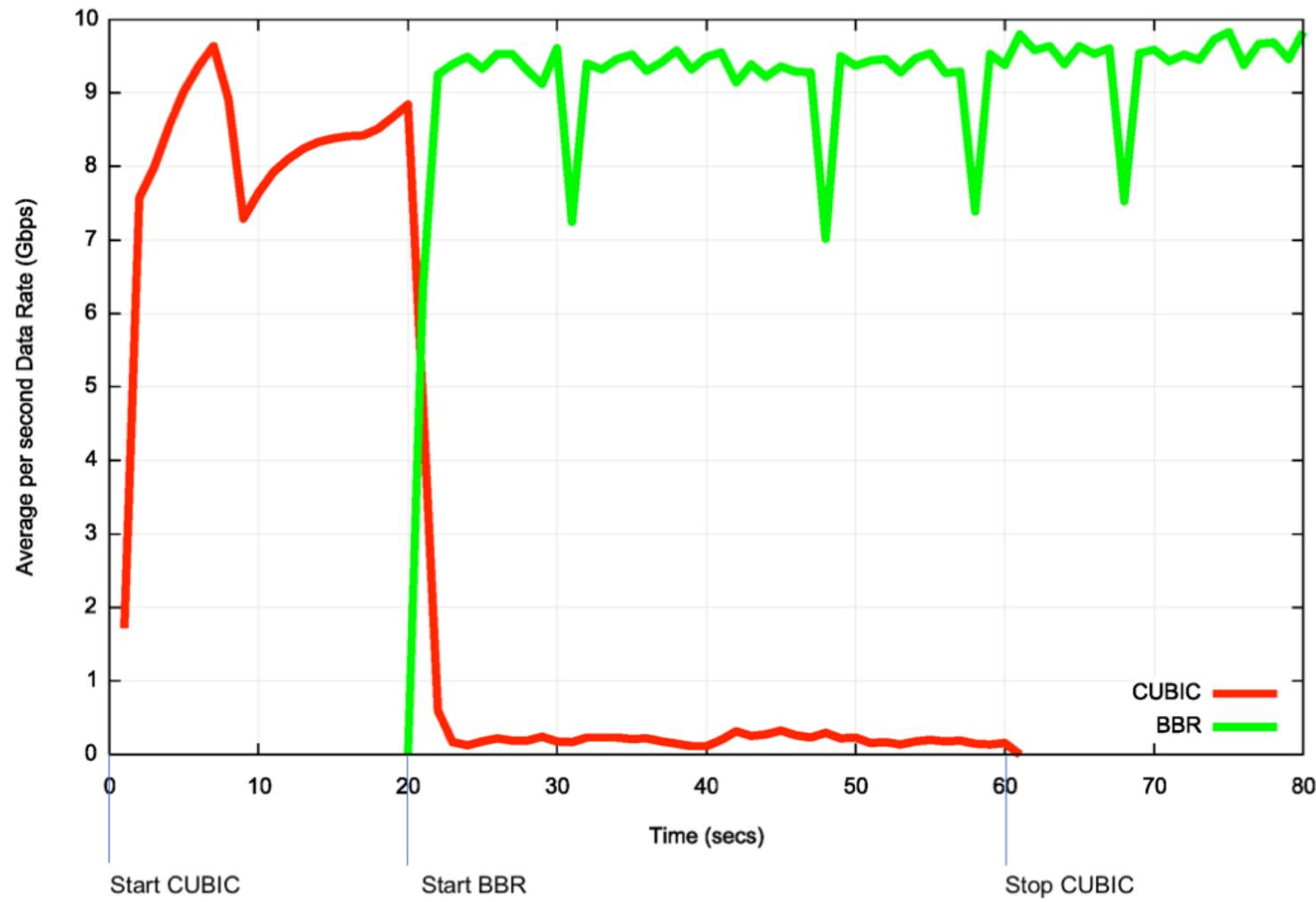
<https://datatracker.ietf.org/meeting/104/materials/slides-104-icrcg-an-update-on-bbr-00>

BBR v2



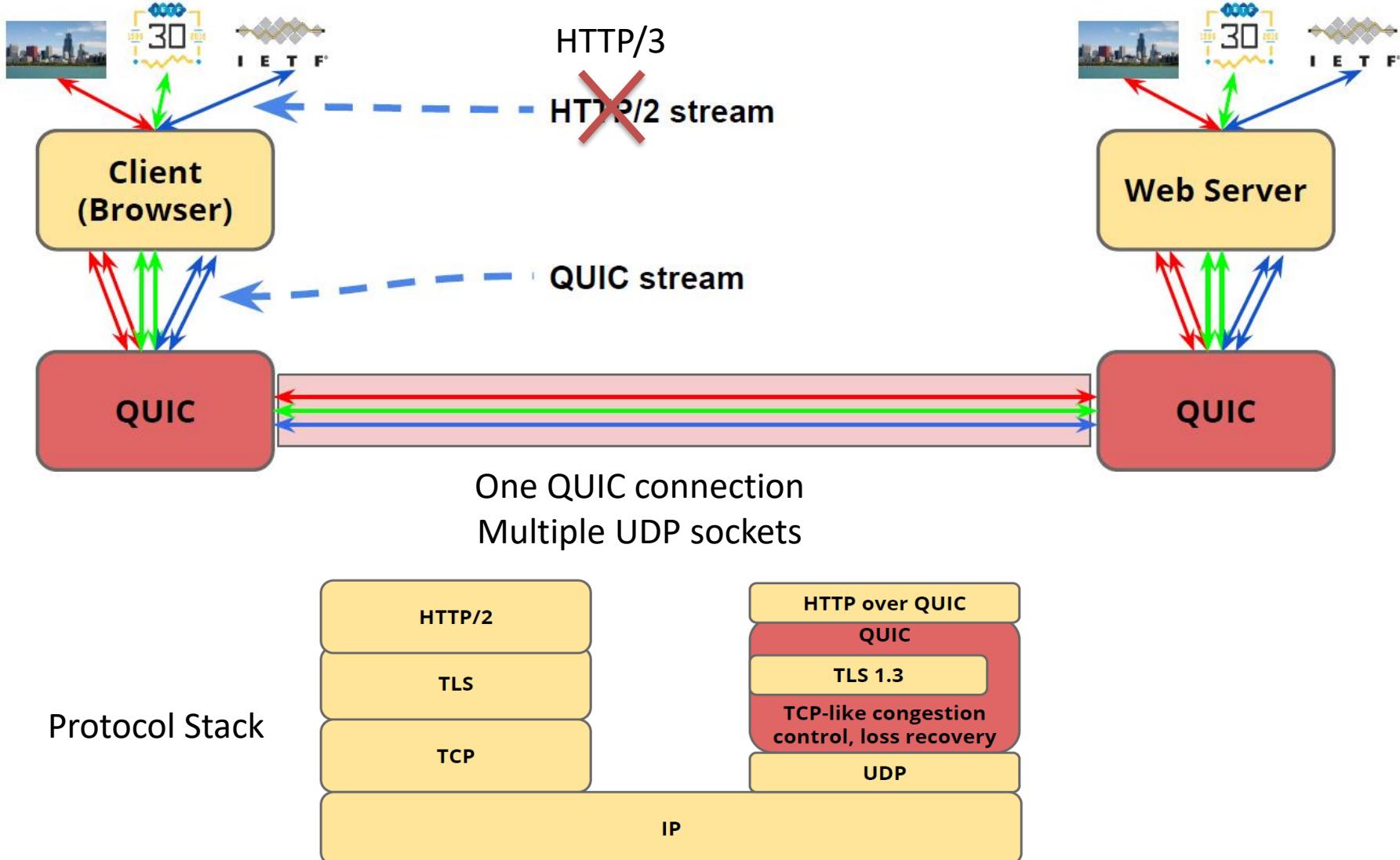
<https://datatracker.ietf.org/meeting/104/materials/slides-104-icrcg-an-update-on-bbr-00>

Experiment BBRv1 vs Cubic



<https://labs.apnic.net/presentations/store/2018-05-15-bbr.pdf>

QUIC



<https://www.ietf.org/proceedings/98/slides/slides-98-edu-sessf-quic-tutorial-00.pdf>

QUIC Features

- User-Space statt Kernel-Space
 - einfacheres Deployment und Updates
- Verschlüsselung, Authentisierung, etc. mit TLS
 - im Gegensatz zu TCP minimaler sichtbarer Header
 - port, connection_id, flags
 - Congestion Control Parameter verschlüsselt
 - (beabsichtigte) Probleme bei Middleboxes
- Verbindungsaufbau
 - keine Handshake
 - Wiederwendung von Schlüsseln bei bekanntem Server
 - 0-RTT oder 1-RTT Verbindungsaufbau
- Congestion Control
 - ähnlich wie bei TCP, verschiedene Varianten z.B. BBR
 - Verbesserungen angedacht
- Seamless Handover
 - Connection nicht über IP-Adresse identifiziert sondern über Connection ID
 - WLAN nach Mobilfunk ohne Verbindungsabbruch
- Tiefergehendes Video über QUIC
 - https://www.youtube.com/watch?v=pq_xk_Pecu4&feature=youtu.be

4.1 Multiplexing

4.2 UDP

4.3 TCP

4.3.1 Übersicht

4.3.2 Kernfunktionalität

4.3.3 Datenübertragung

4.3.3.1 Paketgröße – Maximum Segment Size

4.3.3.2 Datenflussteuerung – Flow Control

4.3.3.3 Überlaststeuerung – Congestion Control nach RFC 5681

4.3.3.4 Fairness

4.3.3.5 Installierte TCP Varianten

4.4 Zusammenfassung

Zusammenfassung

- UDP: User Datagram Protocol
 - ungesicherte Übertragung von Datagrammen
 - verbindungslos, zustandslos
 - Header: Source/Destination Port, Länge, Checksum
- TCP: Transmission Control Protocol
 - gesicherte Datenübertragung
 - Verbindungsaufbau und -abbau (3-Way-Handshake)
 - Header enthält Numbers der gesendeten und bestätigten Bytes
 - Paketverlust durch Timeout und 3 Dup-Acks
 - Datenflusssteuerung (Flow Control) durch Sendfenstermechanismus
 - Congestion Control: dynamische Anpassung der Fenstergröße
 - Prinzipien:
 - Slow Start, Congestion Avoidance, Fast Retransmit und Fast Recovery
 - Ziel: Balance zwischen Effizienz und Fairness
 - Zahlreiche Varianten: Reno, Compound, Cubic
- QUIC (Quick UDP Internet Connections):
 - von Google getriebene Alternative zu TCP
 - Transport über mehrere UDP Verbindungen
 - Congestion Control auf Anwendungsschicht
 - Vorteile: niedrige Latenz, schnellere Entwicklung der Congestion Control im User Space, Unterstützung der Streams von HTTP/2.0