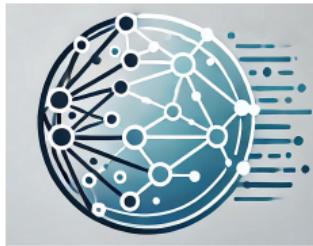

Netze

Modul 9: TCP

👤 Prof. Dr. Michael Rademacher



27. November 2025

Semesterplanung — Vorlesungen

Modul	Dozent	Datum	Thema
1	Rademacher	2. Oktober 2025	Einführung, OSI-Referenzmodell und Topologien
2	Rademacher	9. Oktober 2025	Übertragungsmedien und Verkabelung
3	Rademacher	16. Oktober 2025	Ethernet und WLAN
4	Tschofenig	23. Oktober 2025	IPv4, Subnetze, ARP, ICMP
5	Tschofenig	30. Oktober 2025	IPv6 und Autokonfiguration
6	Tschofenig	6. November 2025	Netzwerksegmentierung
7	Tschofenig	13. November 2025	Routing
8	Rademacher	20. November 2025	Transportschicht und UDP
9	Rademacher	27. November 2025	TCP
10	Rademacher	4. Dezember 2025	DNS und HTTP 1
11	Tschofenig	11. Dezember 2025	HTTP 2 und QUIC
12	Tschofenig	18. Dezember 2025	TLS und VPN
/	/	8. Januar 2026	Bei Bedarf / TBA
13	Tschofenig	15. Januar 2026	Messaging
14	Rademacher	22. Januar 2026	Moderne Netzstrukturen

Semesterplanung — Übungen und Praktika

ID	KW	Art	Thema
	40	/	/
UE-1	41	Übung	Topologien und OSI
UE-2	42	Übung	Übertragungen bspw. Kabel
P-1	43	Praktikum	Laboreinführung und Netzwerktools
S-1	44	Video	IPv4
P-2	45	Praktikum	Adressierung
P-3	46	Praktikum	IPv4 und Autokonfiguration
P-4	47	Praktikum	IPv6 und Autokonfiguration
P-5	48	Praktikum	Routing
P-6	49	Praktikum	Switching
P-7	50	Praktikum	Transportprotokolle
S-2	51	Experiment	VPN
S-2	52	Experiment	VPN
	2	/	/
P-8	3	Praktikum	DNS
P-9	4	Praktikum	Webkommunikation

UE - Übung laut Stundenplan in den Seminarräumen

P - Praktikum in C055

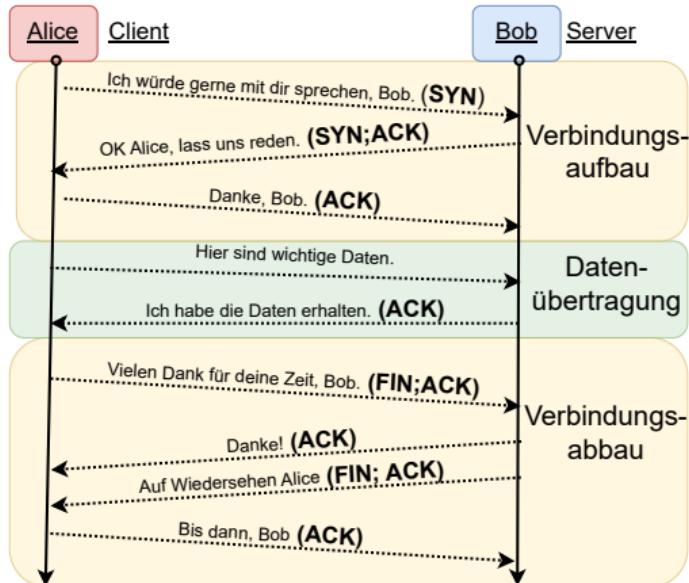
S - Selbststudium KEINE Präsenz

- TCP überträgt Daten zuverlässig über ein unzuverlässiges IP-Netz.
- TCP baut vor der Datenübertragung eine Verbindung mittels Drei-Wege-Handshake auf.
- TCP reguliert den Datenfluss, um Überlastungen des Empfängers und des Netzes zu vermeiden (Flow Control und Congestion Control).
- TCP segmentiert den Byte-Stream gemäß der **Maximum Segment Size (MSS)**; die Fragmentierung erfolgt ggf. auf IP-Ebene.

TCP

TCP bietet einen verbindungsorientierten, zuverlässigen und voll-duplex Byte-Stream-Service mit Fluss- und Staukontrolle.

Verbindungsorientiert

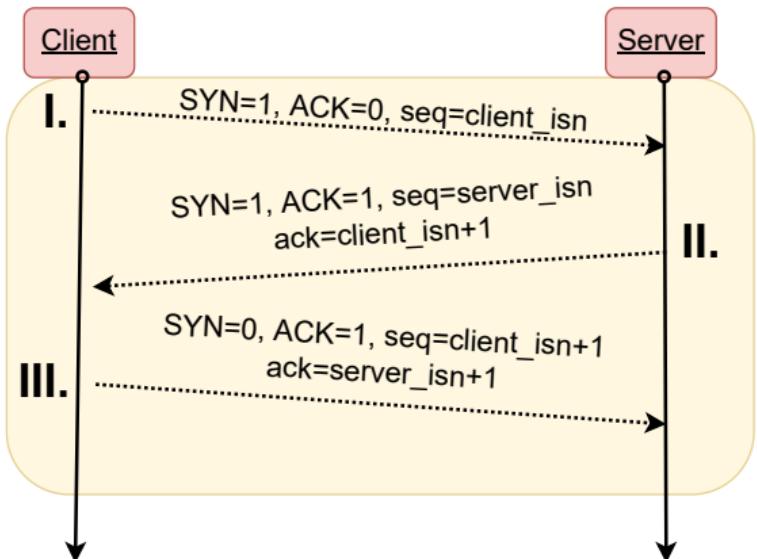


TCP

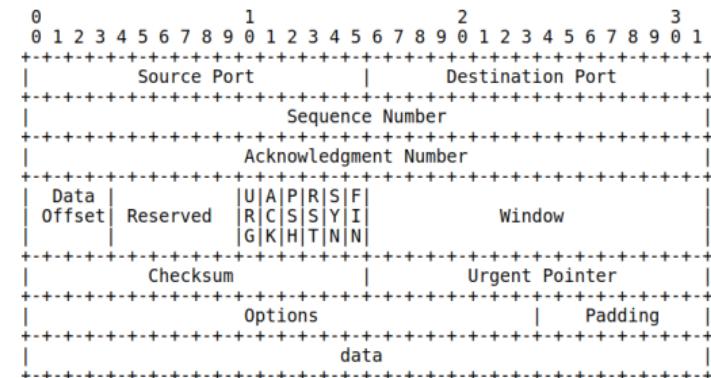
TCP bietet einen **verbindungsorientierten**, zuverlässigen und voll-duplex Byte-Stream-Service mit Fluss- und Staukontrolle.

- Verbindungsauftau (3-Wege-Handshake)
 - SYN → SYN-ACK → ACK
- Datenübertragung – bidirektional (full-duplex)
- Verbindungsabbau (4-Wege-Handshake)
 - FIN / ACK in zwei Richtungen
 - Kann von beiden Parteien initiiert werden
 - Schließen einer Richtung möglich (half-close)

TCP - 3 Wege Handshake



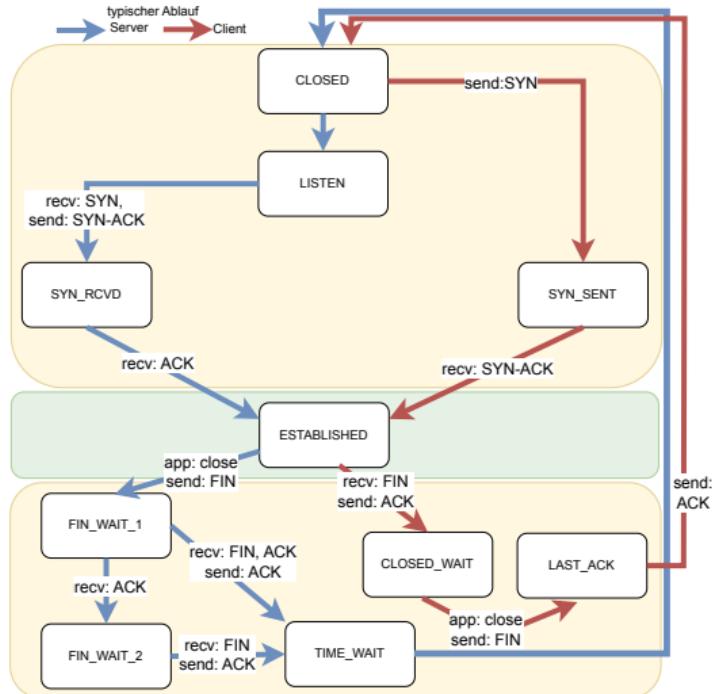
3 Wege Handshake mit TCP



TCP Header Format [2]

- seq = Sequence Number
- isn = initial sequence number
- ack= Acknowledgment Number

Die TCP-State-Machine

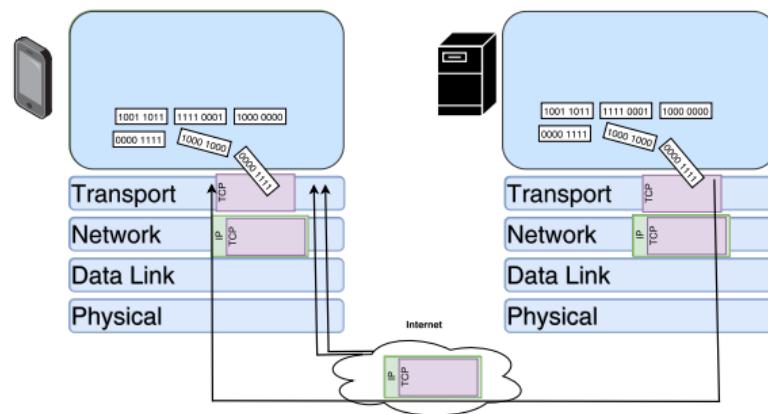


TCP State-Machine [2],
vereinfacht, korrigierte Version

Übertragung eines Byte-Stroms

TCP

TCP bietet einen verbindungsorientierten, zuverlässigen und voll-duplex **Byte-Stream-Service** mit Fluss- und Staukontrolle.



- **TCP ist byteorientiert, nicht nachrichtenorientiert.**
- Der TCP-Strom wird in **Segmenten** übertragen.
- Ein Segment wird typischerweise gesendet, wenn:
 - genügend Daten für ein MSS-Segment vorliegen,
 - ein interner TCP-Timer das Senden auslöst (z. B. Delayed-ACK-/Persist-Timer), oder
 - die Applikation das sofortige Senden anfordert (z. B. mittels **TCP_Nodelay / Push**).

TCP

TCP bietet einen verbindungsorientierten, **zuverlässigen** und voll-duplex Byte-Stream-Service mit Fluss- und Staukontrolle.

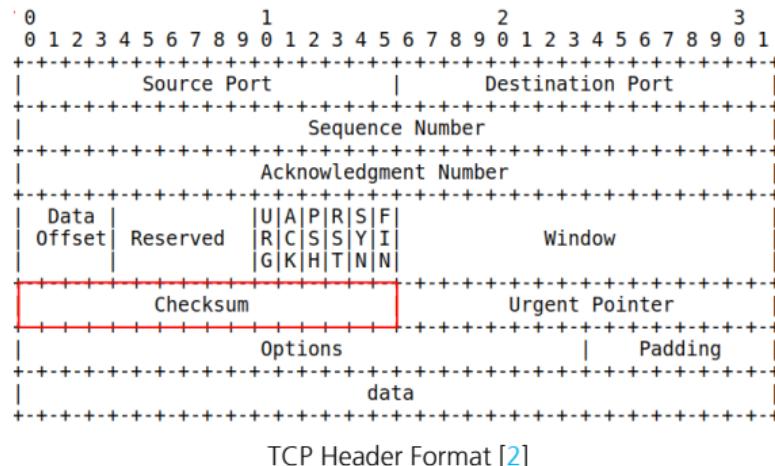
Zuverlässigkeit im Falle von TCP:

- Fehlerhafte oder verlorene Daten werden erkannt und erneut übertragen
- Alle Daten werden vollständig übermittelt
- Duplikate werden eliminiert
- Daten werden in der korrekten Reihenfolge ausgeliefert

→ Umsetzung des Ende-zu-Ende-Prinzips für Zuverlässigkeit.

Typische Fehler und deren Erkennung

- Bitfehler → Internet Checksum (siehe UDP)
- Verlust kompletter Nachrichten → Fehlende Bestätigung (**Ack**nowledgement)
- Duplikate → Sequenznummer
- Falsche Reihenfolge → Sequenznummer

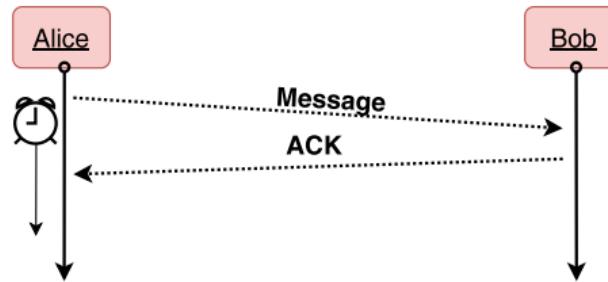


Automatic Repeat reQuest (ARQ)

Die wichtigste Strategie zur Erkennung (und Korrektur) des Verlusts ganzer Nachrichten wird **ARQ** genannt.

1. Der Sender fordert vom Empfänger eine Bestätigung (ein ACK), dass die Daten korrekt empfangen wurden.
2. Kommt nach Ablauf eines Timeouts keine Bestätigung, wird das Paket erneut gesendet.

Sendet man die nächste Nachricht erst, wenn die vorherige erfolgreich übertragen wurde, spricht man von **Stop-and-Wait**.

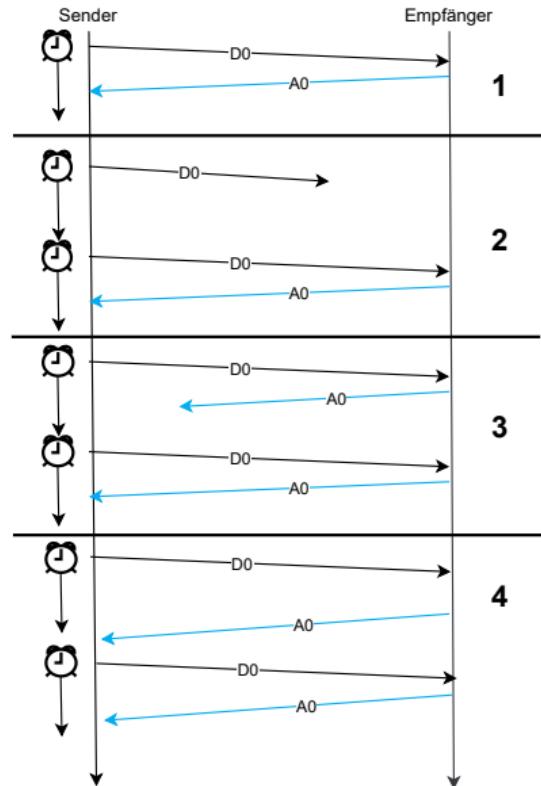


! Bereits bekannte Strategie von WLAN: CSMA/CA

ARQ - Beispielhafte Szenarien

Szenarien

1. Übertragung erfolgreich
2. Paket geht verloren oder ist beschädigt (Timeout)
3. ACK geht verloren oder ist beschädigt (Timeout)
4. ACK kommt zu spät an (Timeout)

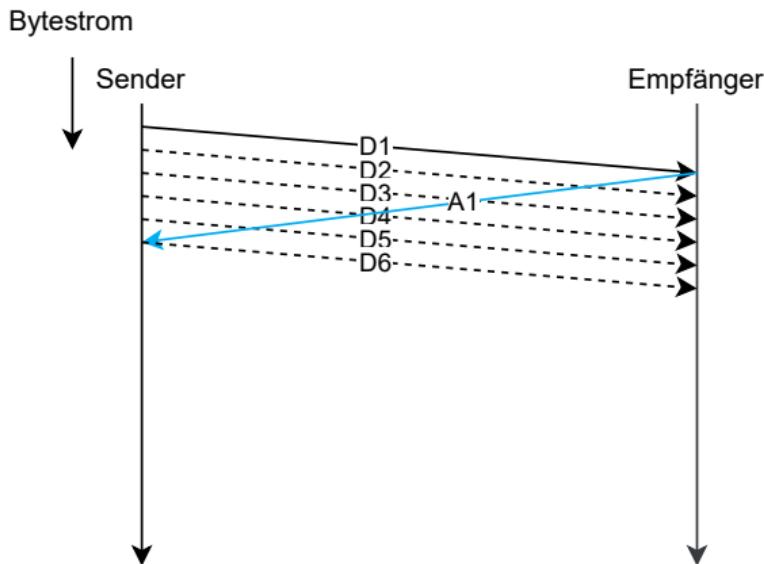


Stop and Wait ist ineffizient

- Obwohl der Sender viele Daten und mehrere Segmente (D2 bis D6) verschicken könnte, muss er **warten** bis D1 durch A1 bestätigt wurde
- Sehr **ineffizient** bei kleinen Datenpaketen und langen Laufzeiten zwischen Sender und Empfänger

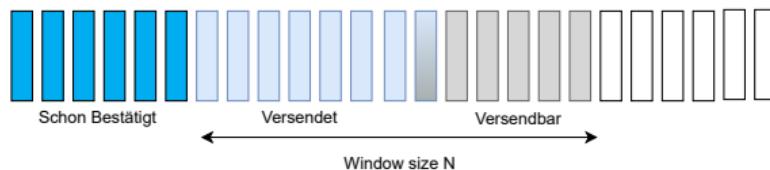
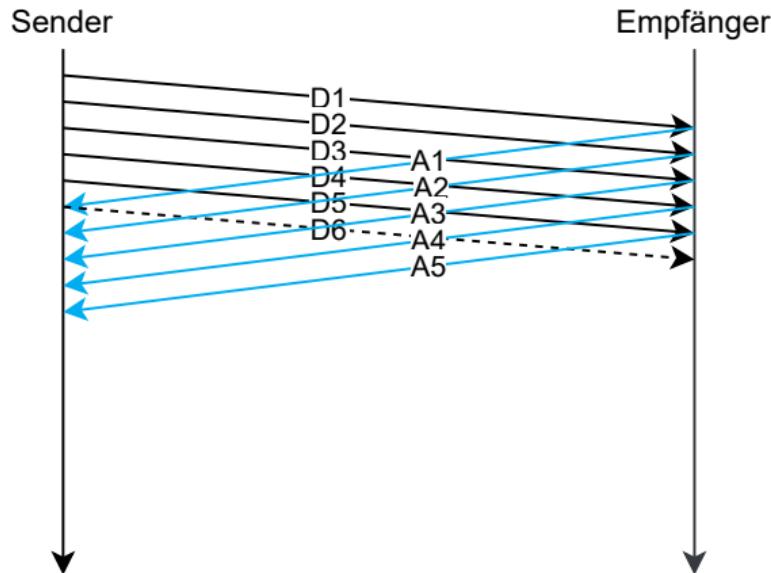
Beispiel:

- Datenpakete: 500 Byte
- RTT: 20 ms
- Übertragungsrate im besten Fall:
$$\frac{500\text{Byte}}{20\text{ms}} = 0.2Mb/s$$



Verbesserung: Sliding Window

- Der Sender darf nicht nur ein Segment versenden sondern N, bevor er ein ACK erwartet.
- Wenn Segmente fortlaufend nummeriert werden (Sequenznummer) entsteht ein sogenanntes **Window** welches die aktuell zu versendeten Segmente angibt.
- Sobald ein Paket(e) bestätigt wird, verschiebt sich Anfang und Ende des Window.
- Es entsteht ein **Sliding Window**.

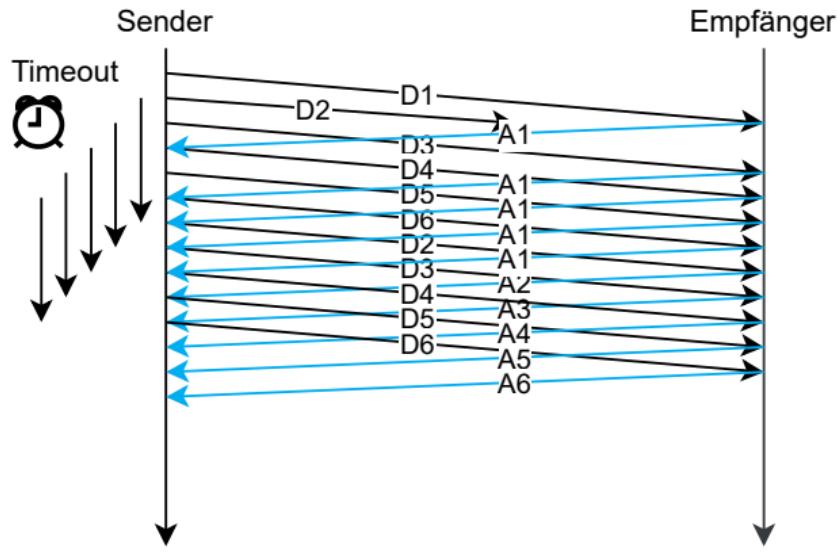


Herausforderung für das Sliding Window

Herausforderung: Wie geht man mit Übertragungsfehlern um? Bedenke, die Reihenfolge beim Empfänger muss eingehalten werden! Verschiedene Strategien:

- **Go-Back-N**
- **Selective Repeat**
- Selective Reject
- ...

Go-Back-N

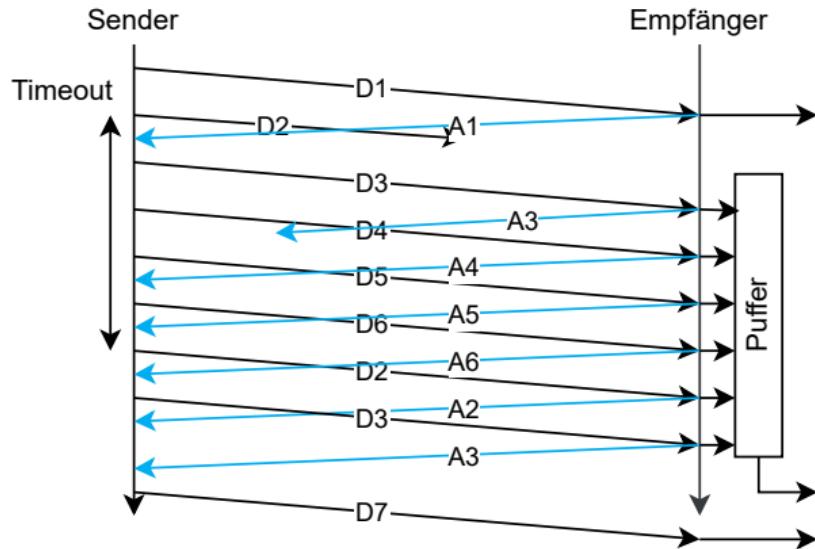


Empfänger:

- Wenn ein Segment mit Sequenznummer n empfangen wird **und** die Reihenfolge korrekt ist, sende ein $\text{ACK}(n)$
- Sollte eine falsche Sequenznummer eintreffen ($n+2$), sende weiterhin ACKs mit n bis $n+1$ eintrifft
- Sehr einfache Implementierung beim Empfänger (ohne Puffer!)

Viele Segmente sind ggf. gleichzeitig unterwegs. Verlust eines einzelnen Segments führt zu vielen (unnötigen) Wiederholungen.

Selective Repeat

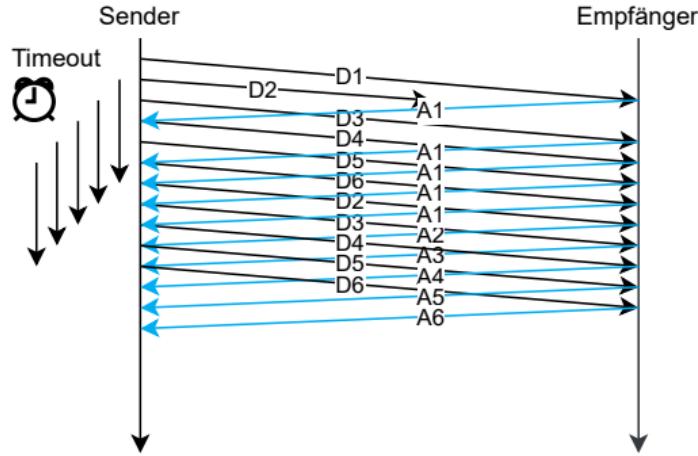


Empfänger:

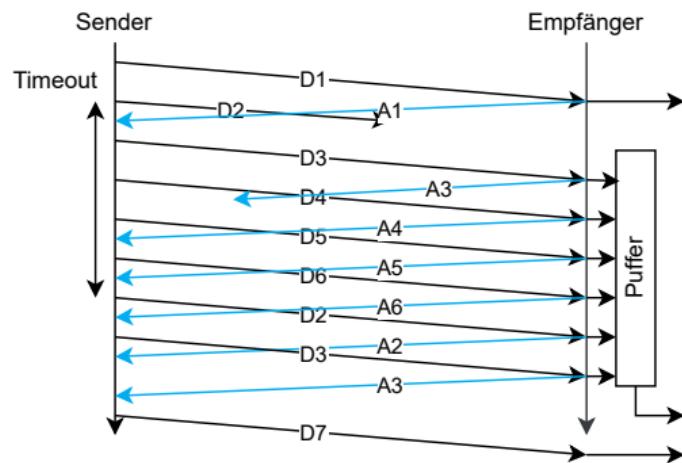
- Individuelle Bestätigung der Segmente, selbst wenn diese in falscher Reihenfolge eintreffen
- Empfangspuffer, um auf fehlende Lücken warten zu können.

Go-Back-N vs. Selective Repeat

Go-Back-N



Selective Repeat



Besseres Verständnis durch die folgende Animation:

https://www2.tkn.tu-berlin.de/teaching/rn/animations/gbn_sr/

Herausforderung: Welchen Wert sollte das TCP (ACK) Timeout haben?

- Länger als die RTT
 - Die RTT variiert aber: WLAN vs. Glasfaser, Entfernung zwischen Sender und Empfänger,
...
 - Ändert sich auch während der Lebenszeit der Verbindung.
-
- RTT zu kurz: Unnötig verworfene Segmente
 - RTT zu lang: Langsame Reaktion bei Fehlern

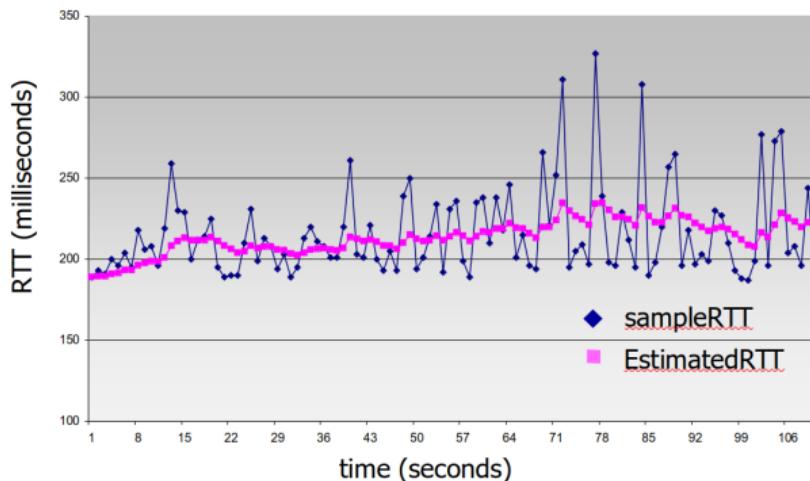
TCP: RTT-Messung und Estimated RTT

Grundidee: TCP misst kontinuierlich die Round-Trip Time (RTT) über erfolgreiche ACKs.

- Nutzung eines Exponential Weighted Moving Average (EWMA), um aktuelle Werte stärker zu gewichten
- Glättung reduziert Einfluss einzelner Ausreißer

$$EstimatedRTT = (1 - \alpha) \cdot EstimatedRTT + \alpha \cdot SampleRTT$$

$$\alpha \approx 0.125$$



Quelle: [9]

TCP: Retransmission Timeout (RTO)

Motivation: Ein Timeout muss nicht nur die gemessene RTT berücksichtigen, sondern auch deren Schwankungen (Jitter).

- TCP schätzt zusätzlich die Varianz der RTT (DevRTT)
- RTO passt sich instabilen Netzwerkbedingungen an

Varianzschätzung:

$$DevRTT = (1 - \beta) \cdot DevRTT + \beta \cdot |SampleRTT - EstimatedRTT|$$

$$\beta \approx 0.25$$

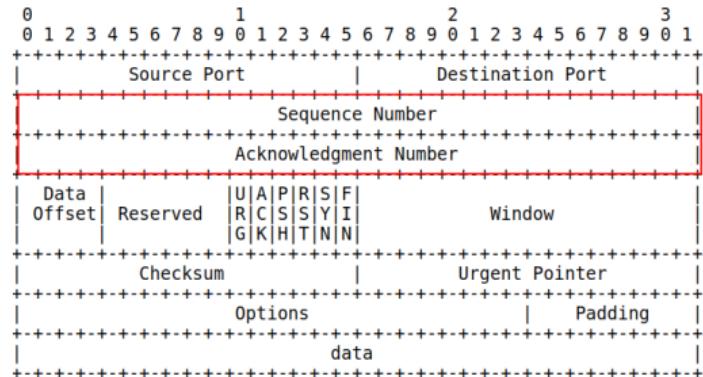
Retransmission Timeout:

$$RTO = EstimatedRTT + 4 \cdot DevRTT$$

RTO entscheidet, wann ein Segment erneut gesendet wird.

TCP ARQ in der Realität

- In der Praxis werden nicht ganze Segmente, sondern **Bytes** (als Teil des Bytestroms) bestätigt.



- „Sequence Number“ (Seq)
- „Acknowledgment Number“ (Ack)

- Moderne TCP-Implementierungen nutzen ein zuverlässiges Übertragungsverfahren, das Merkmale von Go-Back-N und Selective Repeat kombiniert; mit Selective Acknowledgment (SACK)-Unterstützung verhält sich TCP in der Praxis weitgehend wie ein **Selective-Repeat-ähnliches Verfahren**.

Wireshark Beispiel für TCP Seq und Ack

http_witp_jpegs.cap

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

tcp.stream eq 18

Interface Channel 802.11 Preferences

No.	Time	Source	Destination	Length	Protocol	Info
275	10.827791	10.1.1.101	10.1.1.1	62	TCP	3200 → 80 [SYN] Seq=0 Win=0 Len=0 MSS=1460 SACK_PERM
276	10.828277	10.1.1.1	10.1.1.101	62	TCP	80 → 3200 [SYN, ACK] Seq=0 Ack=1 Win=5840 Len=0 MSS=14
277	10.828321	10.1.1.101	10.1.1.1	54	TCP	3200 → 80 [ACK] Seq=1 Ack=1 Win=65535 Len=0
278	10.836425	10.1.1.101	10.1.1.1	691	HTTP	GET /Websidan/2004-07-SeaWorld/fullsize/DSC07858.JPG H
279	10.837691	10.1.1.1	10.1.1.101	60	TCP	80 → 3200 [ACK] Seq=1 Ack=638 Win=7007 Len=0
280	10.840658	10.1.1.1	10.1.1.101	1514	TCP	80 → 3200 [ACK] Seq=1 Ack=638 Win=7007 Len=1460 [TCP s
281	10.841903	10.1.1.1	10.1.1.101	1514	TCP	80 → 3200 [ACK] Seq=1461 Ack=638 Win=7007 Len=1460 [TC
282	10.841956	10.1.1.101	10.1.1.1	54	TCP	3200 → 80 [ACK] Seq=638 Ack=2921 Win=65535 Len=0
283	10.844315	10.1.1.1	10.1.1.101	1514	TCP	80 → 3200 [ACK] Seq=2921 Ack=638 Win=7007 Len=1460 [TC
284	10.844383	10.1.1.101	10.1.1.1	54	TCP	3200 → 80 [ACK] Seq=638 Ack=4381 Win=65535 Len=0

▼ Transmission Control Protocol, Src Port: 80, Dst Port: 3200, Seq: 1461, Ack: 638, Len: 1460
Source Port: 80
Destination Port: 3200
[Stream index: 18]
[Conversation completeness: Complete, WITH_DATA (31)]
[TCP Segment Len: 1460]
Sequence Number: 1461 (relative sequence number)
Sequence Number (raw): 937640163
[Next Sequence Number: 2921 (relative sequence number)]
Acknowledgment Number: 638 (relative ack number)
Acknowledgment number (raw): 886164448

Sequence Number (tcp.seq), 4 bytes Packets: 483 - Displayed: 209 (43.3%) Profile: Default

Sender: 1461 (Seq) + 1460 (Len)
Empfänger: 2921 (ACK)



TCP

TCP bietet einen verbindungsorientierten, zuverlässigen und voll-duplex Byte-Stream-Service mit **Fluss- und Staukontrolle**.

Flow Control



Ein Sender schickt zu viele Daten zu schnell für einen Empfänger.

Congestion Control



Viele Sender schicken zu viele Daten zu schnell und überlasten das Netzwerk.

TCP Flow Control

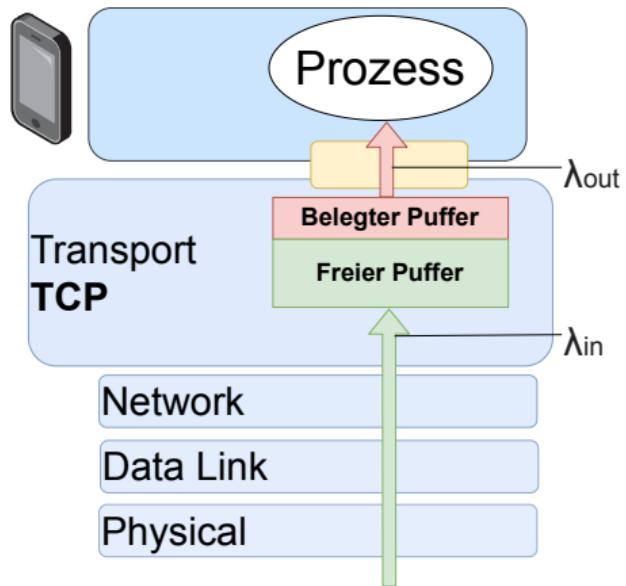
Situation: Der Sender überträgt Daten schneller, als der Prozess auf Empfängerseite sie aus dem Puffer lesen kann ($\lambda_{in} > \lambda_{out}$).

Folge: Der Empfangspuffer kann überlaufen; Segmente würden verworfen → unnötige Wiederholungen.

Lösungsansatz: Der Empfänger informiert den Sender über den aktuell verfügbaren freien Pufferplatz.

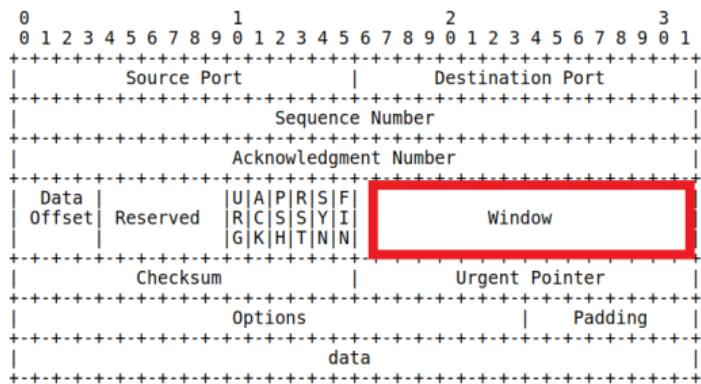
Ziel: Der Empfänger steuert die Sendegeschwindigkeit des Senders.

Empfänger:



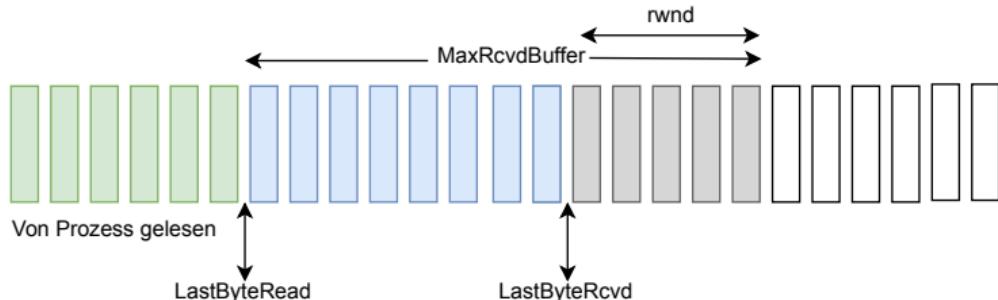
TCP Flow Control

- Der Empfänger teilt die Größe des aktuell freien Puffers (receive window (rwnd)) im Window Feld des TCP headers dem Sender mit
- Sender limitiert sein Sendefenster basierend auf der Größe des Puffers beim Empfänger



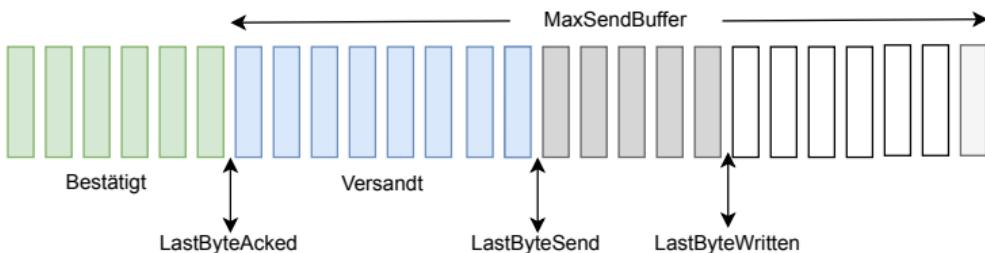
TCP Flow Control - Puffer auf Sender/Empfängerseite

Empfänger:



$$\text{rwnd} = \text{MaxRcvBuffer} - [\text{LastByteRcvd} - \text{LastByteRead}]$$

Sender:



$$\text{LastByteSent} - \text{LastByteAcked} \leq \text{rwnd}$$



Stau (Congestion) in Netzwerken

Situation: Viele Sender übertragen gleichzeitig so viele Daten, dass das Netzwerk überlastet wird. **Folgen:**

- **Lange Verzögerungszeiten** durch wachsende Warteschlangen in Routern
- **Paketverluste** aufgrund überlaufender Routerpuffer
- Paketverluste führen zu **wiederholten Übertragungen** – was die Stausituation weiter verschärft.

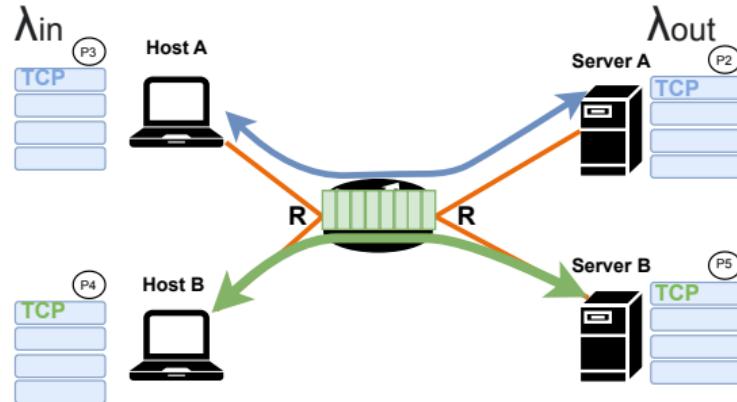


Hinweis: Staukontrolle ist deutlich komplexer als Flow Control, da nicht nur Sender und Empfänger, sondern das gesamte Netzwerkverhalten berücksichtigt werden muss.

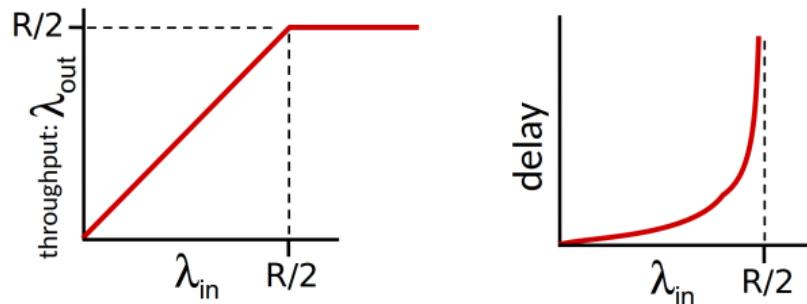
Stau (Congestion) in Netzwerken: Einfaches künstliches Beispiel

Einfaches Szenario:

- Ein Router, **unendliche** Puffer
- Kapazität: R
- Zwei Verbindungen
 - $\lambda_{in}, \lambda_{out}$
- Keine Verluste/Wiederholungen



Was passiert, wenn λ_{in} sich $R/2$ annähert:

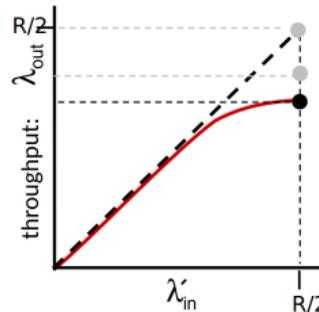
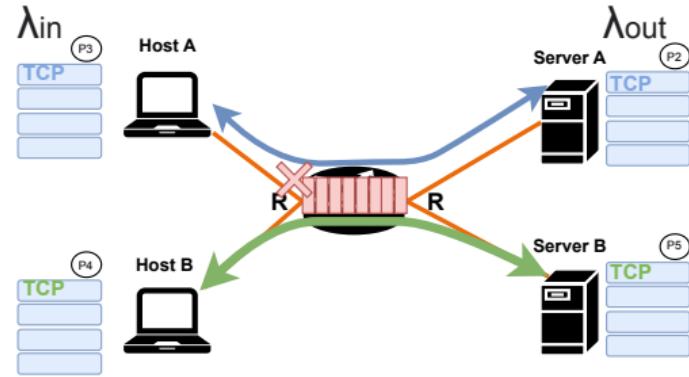


Stau (Congestion) in Netzwerken: Einfaches realistischeres Beispiel

Einfaches Szenario:

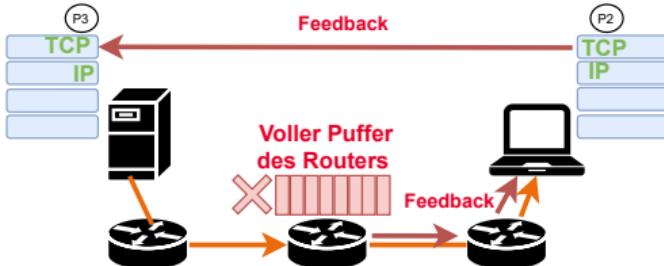
- Ein Router, **endliche** Puffer
- Kapazität: R
- Zwei Verbindungen
 - $\lambda_{in}, \lambda_{out}$
- Wiederholungen von Paketen durch verwarfene Pakete an vollen Puffern
- Timeout von Paketen (ACK-Timeout) und dadurch unnötige doppelte Pakete

Die Kapazität von λ_{in} von $R/2$ kann nicht erreicht werden.

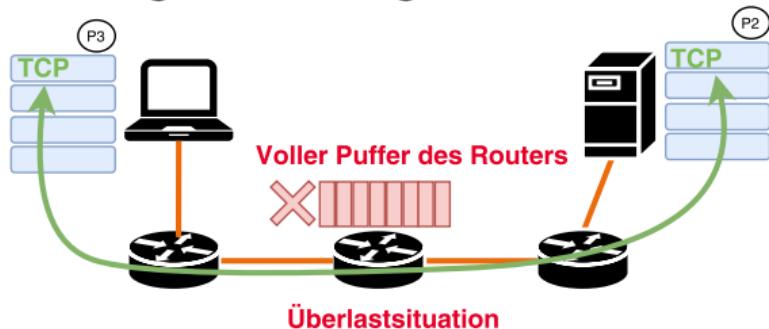


Ansätze für Congestion Control

Netzgestützte Congestion Control



Abgeleitete Congestion Control



- Router signalisieren Überlastsituationen an Empfänger, welcher die Information an den Sender kommuniziert.
 - Anpassung bzw. Nutzung von IP/TCP-Headerfeldern
- Beispiel: **TCP ECN** (Explicit Congestion Notification), RFC 3168 [6] und RFC 8311 [3]

- Überlast wird aus Verbindungsdaten abgeleitet
 - Segmentverluste, RTT-Anstieg, Jitter, ...
- Beispiel: **TCP** (klassisch verlustbasiert)

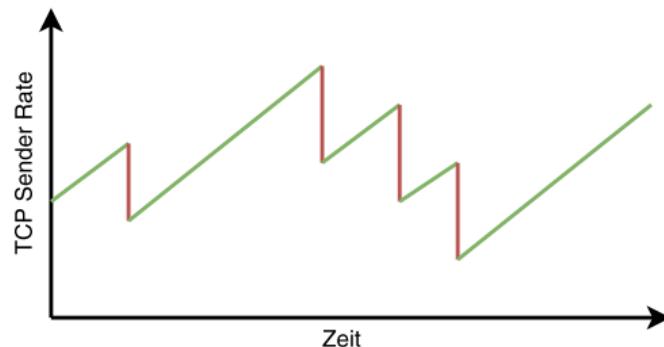
TCP Congestion Control: AIMD

Grundidee: **Sender erhöhen die Sendedatenrate sukzessive und reduzieren sie bei Paketverlusten (Congestion)** → „Sondierung“ der maximal möglichen Datenrate.

Additive Increase Multiplicative Decrease (AIMD)

Additive Increase: Erhöht das Congestion Window pro RTT um **+1 MSS** bei erfolgreicher Übertragung.

Multiplicative Decrease: Reduziert das Congestion Window z. B. um die Hälfte (**/2**) bei Verlusten.



TCP Congestion Control: Wichtige TCP-Algorithmen

Genaues Verhalten hängt vom verwendeten Algorithmus ab:

- **TCP Tahoe** – klassischer Algorithmus aus [8] (kein Fast Recovery; Verlust → cwnd = 1 MSS)
- **TCP Westwood** – Forschungsalgorithmus aus [10], nie als RFC standardisiert
- **TCP Reno** – RFC 5681[4] (definiert AIMD, Fast Retransmit, Fast Recovery)
- **TCP NewReno** – RFC 6582[7] (verbessertes Fast Recovery)
- **TCP Cubic** – RFC 8312[12] (Standard in Linux; gut für High-BDP-Netze)



Additive Increase Multiplicative Decrease (AIMD)

Warum **multiplikative Verringerung**?

Herausforderung: Mit welcher „**Sender Rate**“ sollte gestartet werden?

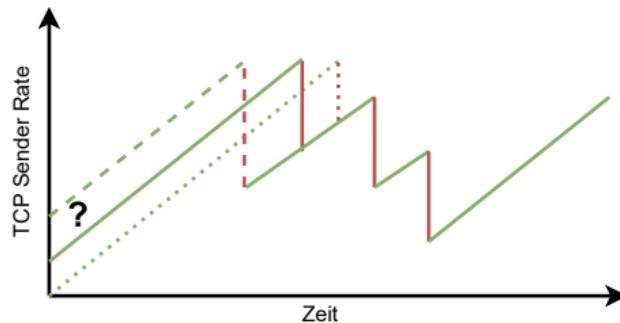
- Congestion ist schlecht für alle Teilnehmer (**Fairness**) daher aggressive Reaktion nötig

Warum **additive Erhöhung**?

- Sukzessive Erhöhung um ein Gleichgewicht zu finden

Sender Rate \approx Anzahl an MSS Bytes die gleichzeitig verschickt werden!
Diese Sender Rate nennt man

Congestion Window.



- Zu Hoch: Neue Verbindungen überlasten das Netz
- Zu Niedrig: Sehr lange Zeit bis hohe Datenraten erreicht werden

Initial Congestion Window (IW10)

Motivation: Zu Beginn einer TCP-Verbindung ist der Congestion Window (CWND) sehr klein → langsamer Start bei Web- und Echtzeitanwendungen.

Historische Entwicklung des Initial Window (IW):

- Ursprünglich: **IW = 1 MSS** (RFC 2001 [13])
- Ab 1998: **IW = 2–4 MSS** (RFC 2414 [11])
- Heute: **IW = 10 MSS** (RFC 6928 [5])

Warum IW10?

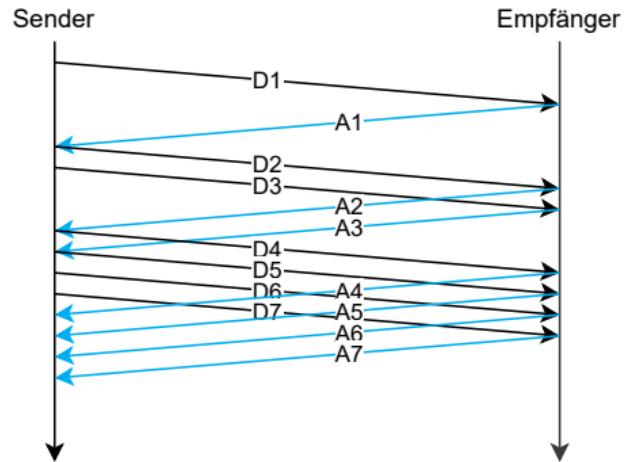
- Schnellere Ladezeiten für Webtraffic (v. a. HTTPS mit vielen kleinen Objekten)
- Bessere Performance über moderne, latenzoptimierte Netze
- Keine signifikante Zunahme von Paketverlusten in Internet-Messkampagnen

RFC-Basis: RFC 6928 – “Increasing TCP’s Initial Window”



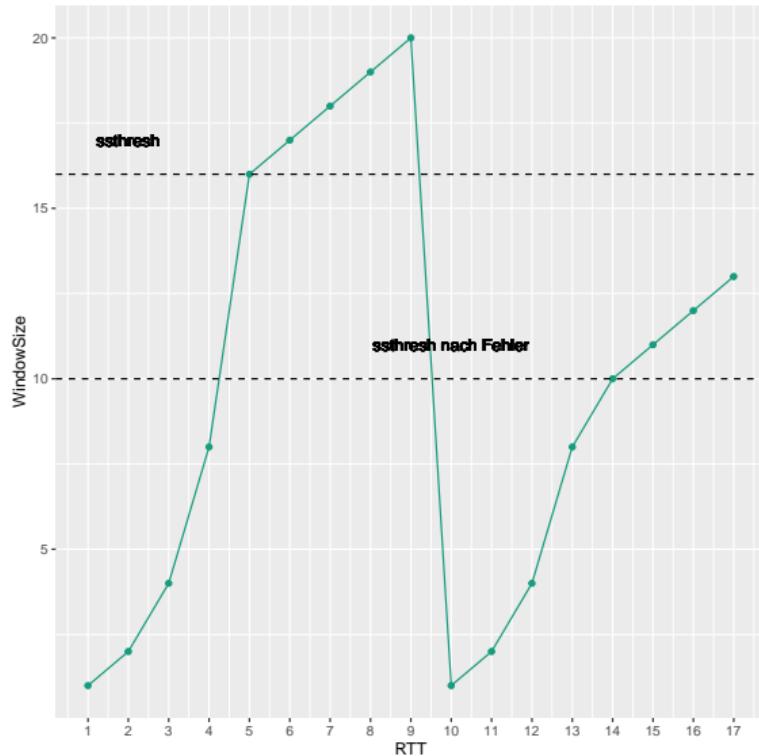
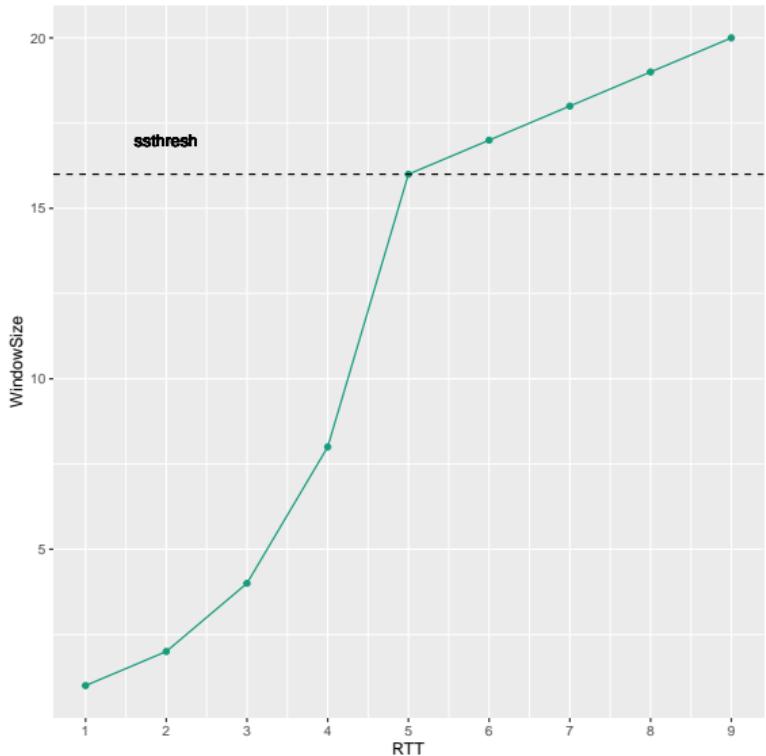
TCP Slow Start

- Start mit dem **Initial Window**
(Abbildung benutzt RFC 2001 mit 1 MSS - historisch)
- Für jede erfolgreiche RTT: **Verdopplung der Senderate** → exponentielles Wachstum
- Dies widerspricht nicht AIMD — Slow Start gilt **vor** AIMD/Congestion Avoidance
- Übergang zur Congestion Avoidance, sobald **ssthresh** erreicht oder überschritten ist → dann nur noch **+1 MSS pro RTT** (Additive Increase)

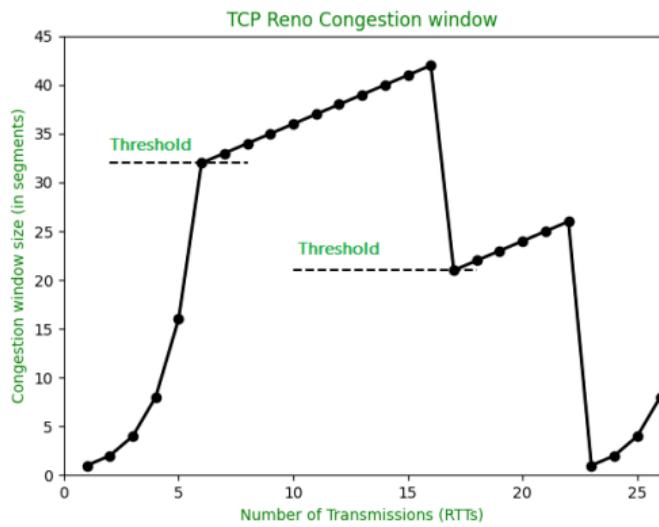
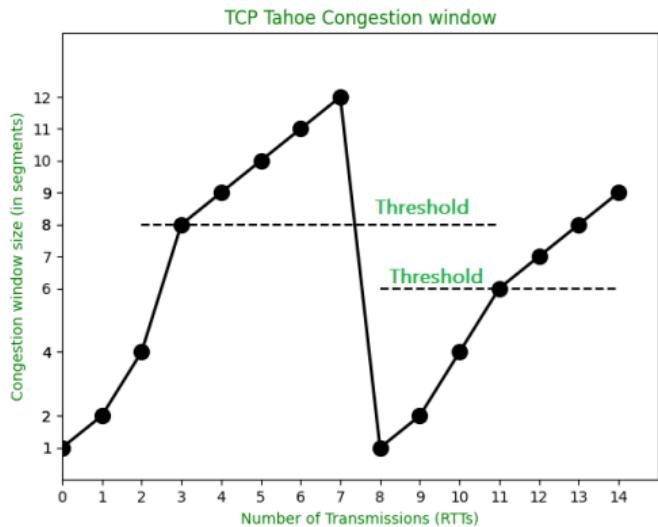


Slow Start betrifft nicht das langfristige Wachstum, sondern die **initiale Erhöhung** der Senderate.

TCP Slow Start — Hier TCP Tahoe



TCP Algorithmen – TCP Tahoe vs Reno



Quelle: [1]

Unterschied im Verhalten des Congestion Windows (cwnd):

- **TCP Tahoe:** Bei **jedem Verlust** (Timeout oder 3-Duplikat-ACKs) → cwnd wird auf **1 MSS** zurückgesetzt und Slow Start beginnt erneut. ⇒ sehr konservativ.
- **TCP Reno:** Bei **3 Duplikat-ACKs** → **Fast Retransmit + Fast Recovery**: cwnd wird nur auf **cwnd/2** gesetzt (statt auf 1 MSS). Erst bei echten Timeouts fällt cwnd auf 1 MSS. ⇒ weniger drastische Reduktion, höhere Effizienz.

Receiver Window und Congestion Window

TCP Flow Control



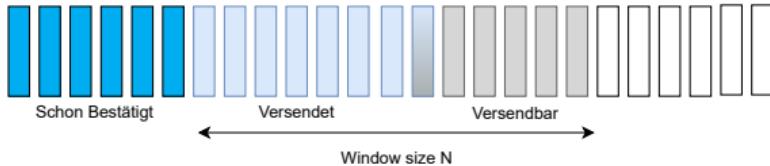
- **Receiver Window** wird dem Sender über das Window Feld mitgeteilt

TCP Congestion Control



- **Congestion Window** wird durch den Sender selbst bestimmt

Window size = min{congestion window, receiver window}



Bufferbloat: Große Puffer, große Probleme

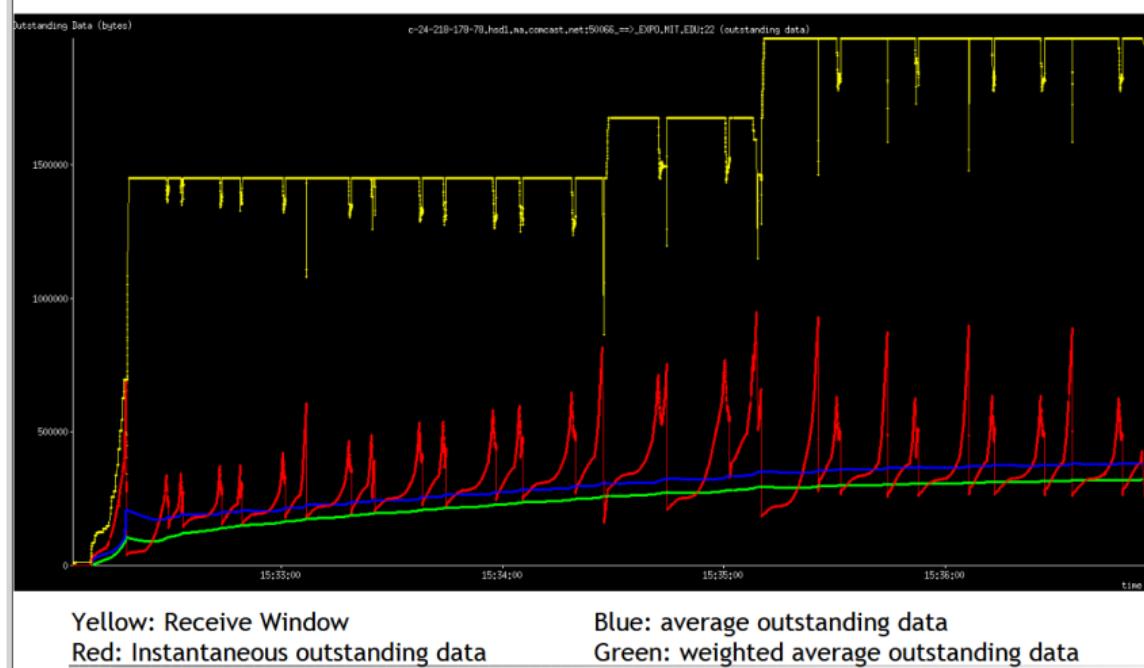
Problem: In modernen Routern und Endgeräten führen **überdimensionierte Puffer** zu extrem langen Warteschlangen – obwohl sie Paketverluste verhindern.

Folgen:

- stark erhöhte **Latenz**
- hoher **Jitter**
- verzögerte Reaktion der **TCP-Congestion-Control**

Wichtig: Bufferbloat beeinträchtigt nicht nur TCP. Auch **UDP-Echtzeitanwendungen** wie VoIP, Gaming oder Videokonferenzen leiden unter der erhöhten Latenz und dem Jitter – selbst ganz ohne Paketverluste.

Bufferbloat sichtbar gemacht (Jim Gettys, IETF 80)



Messung aus Jim Gettys' Vortrag: „*Bufferbloat – Dark Buffers in the Internet*“ (IETF 80, 2011)
<https://www.ietf.org/proceedings/80/slides/tsvarea-1.pdf>

(Eine detaillierte Erklärung der Messung folgt auf der nächsten Folie.)

Wie die Messung zu interpretieren ist:

- Die rote Linie zeigt die **instantane Menge an Daten im Flug** (*instantaneous outstanding data*).
- Starker Anstieg bedeutet: Es befinden sich deutlich mehr Daten im Netz als das Bandbreiten-Verzögerungsprodukt (BDP) erlaubt.
- Ursache: Überdimensionierte Routerpuffer halten Pakete lange zurück → TCP erhält verspätet Rückmeldung über Congestion.
- Folge: **Ungewöhnliches TCP-Verhalten** (Duplicate ACKs, Retransmissions, viele SACKs).

Jim Gettys

Kernproblem: TCP erkennt Überlast primär durch
Paketverluste oder steigende **RTT**.

- Große Puffer halten Pakete lange zurück → RTT steigt, aber **keine Verluste**.
- TCP interpretiert dies nicht sofort als Stau und sendet weiter schnell.
- Warteschlange wächst weiter → **extreme Latenz** bei gefüllten Puffern.
- Verluste treten erst spät auf → langsame und ineffiziente Reaktion.

Ergebnis: Das Netzwerk ist faktisch „verstopft“, aber TCP merkt es erst sehr spät.

Hinweis: **Jim Gettys** machte das Bufferbloat-Problem erstmals breit sichtbar und initiierte die Diskussion über moderne AQM-Verfahren.



Entwickler von X Window und Mitbegründer des **Bufferbloat Projects**. Seine Messungen und Analysen lösten den modernen Umgang mit zu großen Puffern aus.

Zusammenfassung: TCP, UDP und QUIC

TCP

- Verbindungsorientiert
- Zuverlässige Übertragung
- Reihenfolge garantiert
- Flusskontrolle (Flow Control)
- Überlastkontrolle (Congestion Control)
- Fragmentierung und Reassembly
- Head-of-Line (HoL) Blocking auf Transportebene
- Erweiterungen über RFCs (z. B. ECN, CUBIC)

UDP

- Verbindungslos
- „Schnörkellose“ Erweiterung von IP
- Keine Zuverlässigkeit
- Keine Reihenfolge
- Keine Fluss- oder Überlastkontrolle
- Minimaler Overhead
- Grundlage für Echtzeitprotokolle (z.B. RTP)

QUIC

- Läuft über UDP (User-Space Transport)
- Integrierte TLS 1.3 Verschlüsselung
- Multiplexing ohne Transport-HoL-Blocking
- Zuverlässige Streams, optional unzuverlässig
- Eigene Congestion-Control
- Schnellere Verbindungsaufnahme (0-RTT, 1-RTT)
- Heute Basis von HTTP/3



Quellen I

- [1] Tcp tahoe and tcp reno - geeksforgeeks.
<https://www.geeksforgeeks.org/tcp-tahoe-and-tcp-reno/>.
(Accessed on 06/06/2024).
- [2] Transmission Control Protocol.
RFC 793, September 1981.
- [3] Black, D. L.
Relaxing Restrictions on Explicit Congestion Notification (ECN) Experimentation.
RFC 8311, January 2018.
- [4] Blanton, E., Paxson, D. V., and Allman, M.
TCP Congestion Control.
RFC 5681, September 2009.
- [5] Chu, J., Dukkipati, N., Cheng, Y., and Mathis, M.
Increasing TCP's Initial Window.
RFC 6928, April 2013.
- [6] Floyd, S., Ramakrishnan, D. K. K., and Black, D. L.
The Addition of Explicit Congestion Notification (ECN) to IP.
RFC 3168, September 2001.
- [7] Gurtov, A., Henderson, T., Floyd, S., and Nishida, Y.
The NewReno Modification to TCP's Fast Recovery Algorithm.
RFC 6582, April 2012.
- [8] Jacobson, V.
Congestion avoidance and control.
In *Proceedings of the ACM SIGCOMM* (1988), pp. 314–329.
- [9] Kurose, J. F., and Ross, K. W.
Computer Networking: A Top-Down Approach, 7 ed.
Pearson, Boston, MA, 2016.



Quellen II

- [10] Mascolo, S., Casetti, C., Gerla, M., Sanadidi, M. Y., and Wang, R.
Tcp westwood: Bandwidth estimation for enhanced transport over wireless links.
In *Proceedings of the 7th Annual International Conference on Mobile Computing and Networking* (New York, NY, USA, 2001), MobiCom '01, Association for Computing Machinery, p. 287–297.
- [11] Partridge, D. C., Floyd, S., and Allman, M.
Increasing TCP's Initial Window.
RFC 2414, September 1998.
- [12] Rhee, I., Xu, L., Ha, S., Zimmermann, A., Eggert, L., and Scheffenegger, R.
CUBIC for Fast Long-Distance Networks.
RFC 8312, February 2018.
- [13] Stevens, W. R.
TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms.
RFC 2001, January 1997.