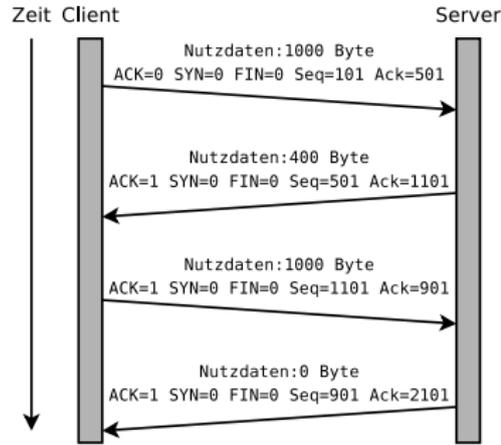


TCP-Datenübertragung

Um eine Datenübertragung zu zeigen, sind für die **Seq-Nummer** (Folgenummer aktuelles Segment) und die **Ack-Nummer** (Folgenummer nächstes erwartetes Segment) konkrete Werte nötig

- Im Beispiel ist zu Beginn des Dreiwege-Handshake die Folgenummer des Clients $x=100$ und die des Servers $y=500$
- Nach Abschluss des Dreiwege-Handshake: $x=101$ und $y=501$

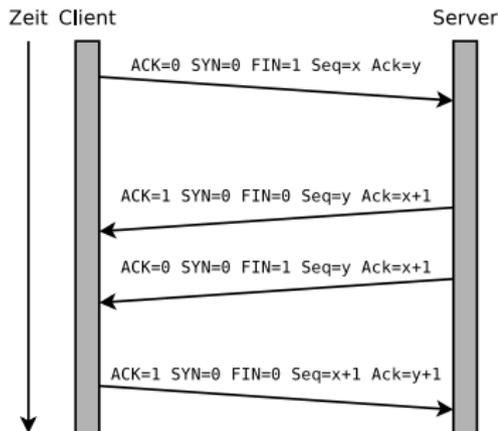
- 1 Client überträgt 1000 Byte Nutzdaten
- 2 Server bestätigt mit ACK=1 die empfangenen Nutzdaten und fordert mit der Ack-Nummer 1101 das nächste Segment an. Im gleichen Segment überträgt der Server 400 Bytes Nutzdaten
- 3 Client überträgt weitere 1000 Byte Nutzdaten. Zudem bestätigt er den Empfang der Nutzdaten mit ACK=1 und fordert mit der Ack-Nummer 901 das nächste Segment an
- 4 Server bestätigt mit ACK=1 die empfangenen Nutzdaten und fordert mit der Ack-Nummer 2101 das nächste Segment an



TCP-Verbindungsabbau

- Der Verbindungsabbau ist dem Verbindungsaufbau ähnlich
- Statt des SYN-Bit kommt das FIN-Bit zum Einsatz, das anzeigt, dass keine Nutzdaten mehr vom Sender kommen

- 1 Client sendet den Abbauwunsch mit FIN=1
- 2 Server sendet eine Bestätigung mit ACK=1
- 3 Server sendet den Abbauwunsch mit FIN=1
- 4 Client sendet eine Bestätigung mit ACK=1



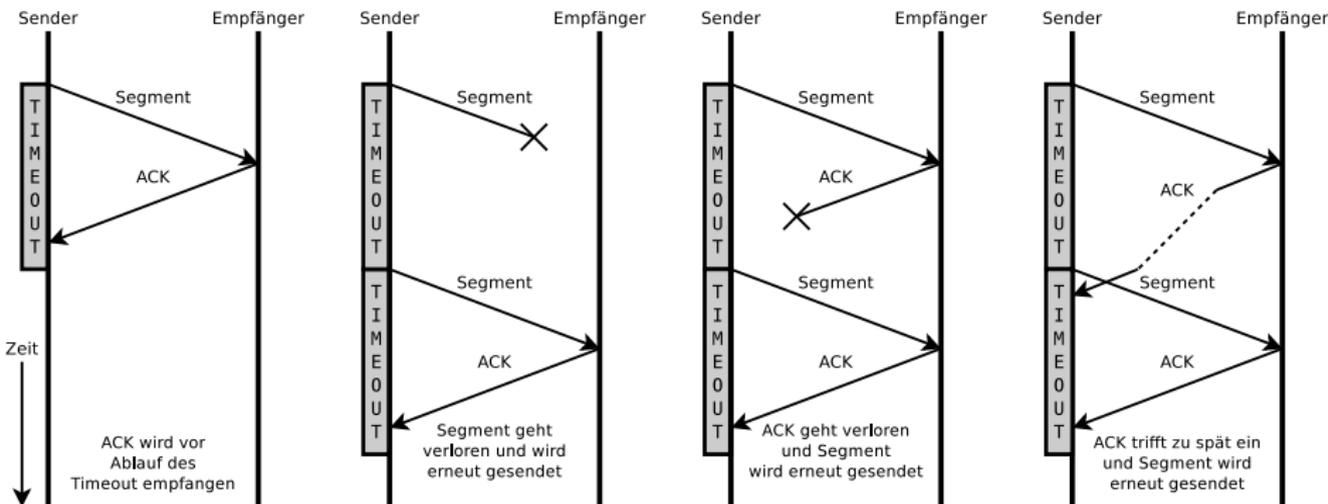
- Beim Verbindungsabbau werden keine Nutzdaten ausgetauscht

Zuverlässige Übertragung durch Flusskontrolle (*Flow Control*)

- Via Flusskontrolle steuert der Empfänger die **Sendegeschwindigkeit des Senders** dynamisch und stellt so und die **Vollständigkeit der Datenübertragung** sicher
 - Langsame Empfänger sollen nicht mit Daten überschüttet werden
 - Dadurch würden Daten verloren gehen
 - Während der Übertragung verlorene Daten werden erneut gesendet
- Vorgehensweise: **Sendewiederholungen**, wenn diese nötig sind
- Grundlegende Mechanismen:
 - **Bestätigungen** (Acknowledgements, ACK) als Feedback bzw. Quittung
 - **Zeitschranken** (Timeouts)
- Konzepte zur Flusskontrolle:
 - **Stop-and-Wait**
 - **Schiebefenster** (Sliding-Window)

Stop-and-Wait

- Nach dem Senden eines Segments wartet der Sender auf ein ACK
 - Kommt in einer bestimmten Zeit kein ACK an \implies Timeout
 - Timeout \implies Segment wird erneut gesendet



- **Nachteil:** Geringer Durchsatz gegenüber der Leitungskapazität

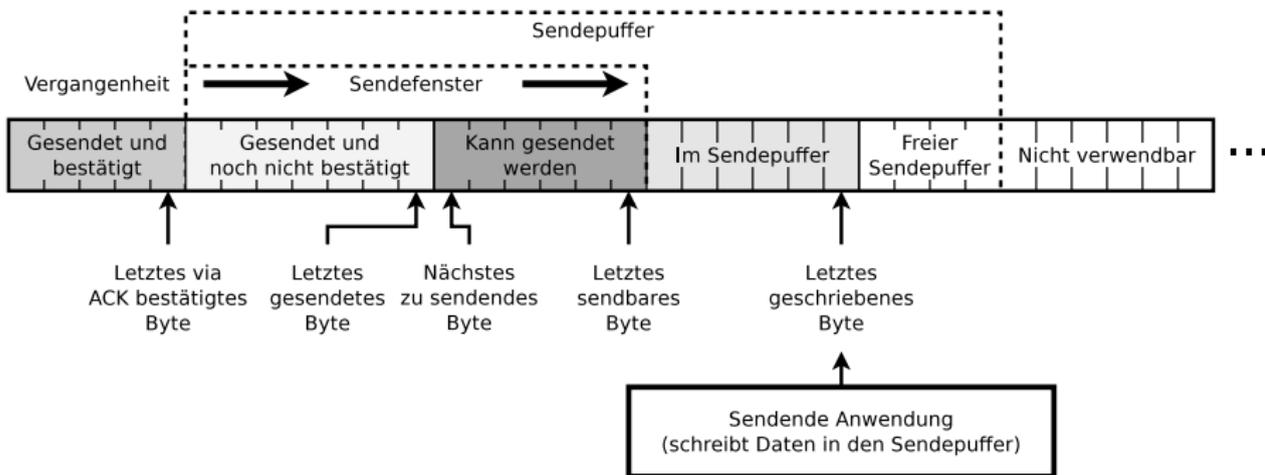
Das **Trivial File Transfer Protocol** (RFC 783) arbeitet nach dem Prinzip Stop-and-Wait

Schiebefenster (Sliding-Window)

- Ein **Fenster** ermöglicht dem Sender die Übertragung einer bestimmten Menge Segmente, bevor eine Bestätigung (Quittung) erwartet wird
 - Beim Eintreffen einer Bestätigung wird das Sendefenster verschoben und der Sender kann weitere Segmente aussenden
 - Der Empfänger kann mehrere Segmente auf einmal bestätigen
⇒ **kumulative Acknowledgements**
 - Beim Timeout übermittelt der Sender alle Segmente im Fenster neu
 - Er sendet also alles ab der letzten unbestätigten Sequenznummer erneut
- Ziel: Leitungs- und Empfangskapazität besser auslasten

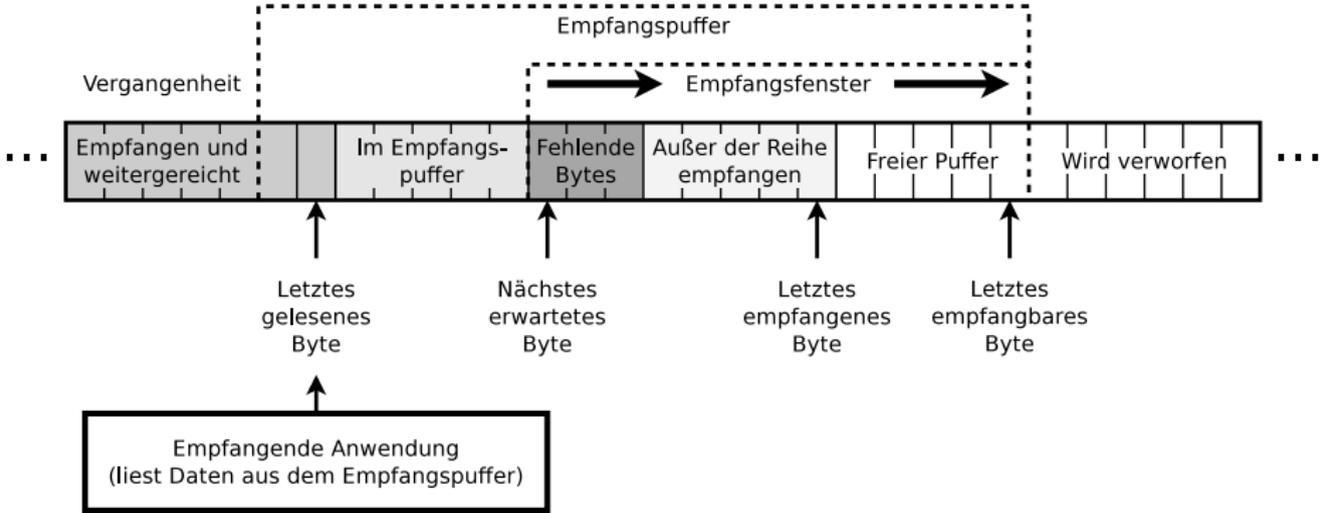
Schiebefenster – Vorgehensweise: Sender

- Der Sendepuffer enthält Daten der Anwendungsschicht, die...
 - bereits gesendet, aber noch nicht bestätigt wurden
 - bereits vorliegen, aber noch nicht gesendet wurden



Schiebefenster – Vorgehensweise: Empfänger

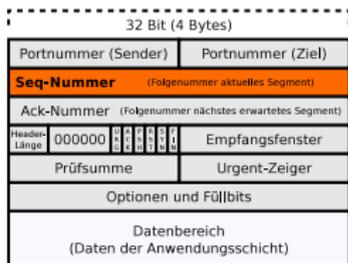
- Der Empfangspuffer enthält Daten für die Anwendungsschicht, die...
 - in der korrekten Reihenfolge vorliegen, aber noch nicht gelesen wurden
 - außer der Reihe angekommen sind



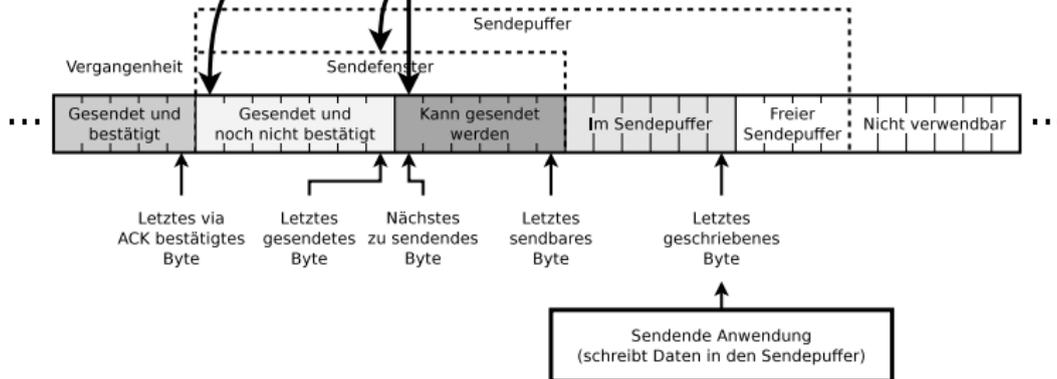
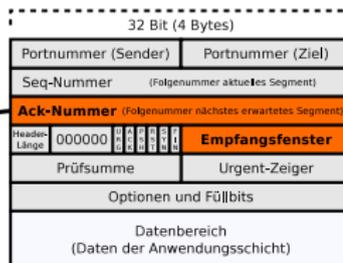
- Der Empfänger gibt dem Sender an, wie groß sein Empfangsfenster ist
 - Wichtig, um einen Pufferüberlauf zu vermeiden!

TCP-Flusskontrolle

zu sendendes Segment



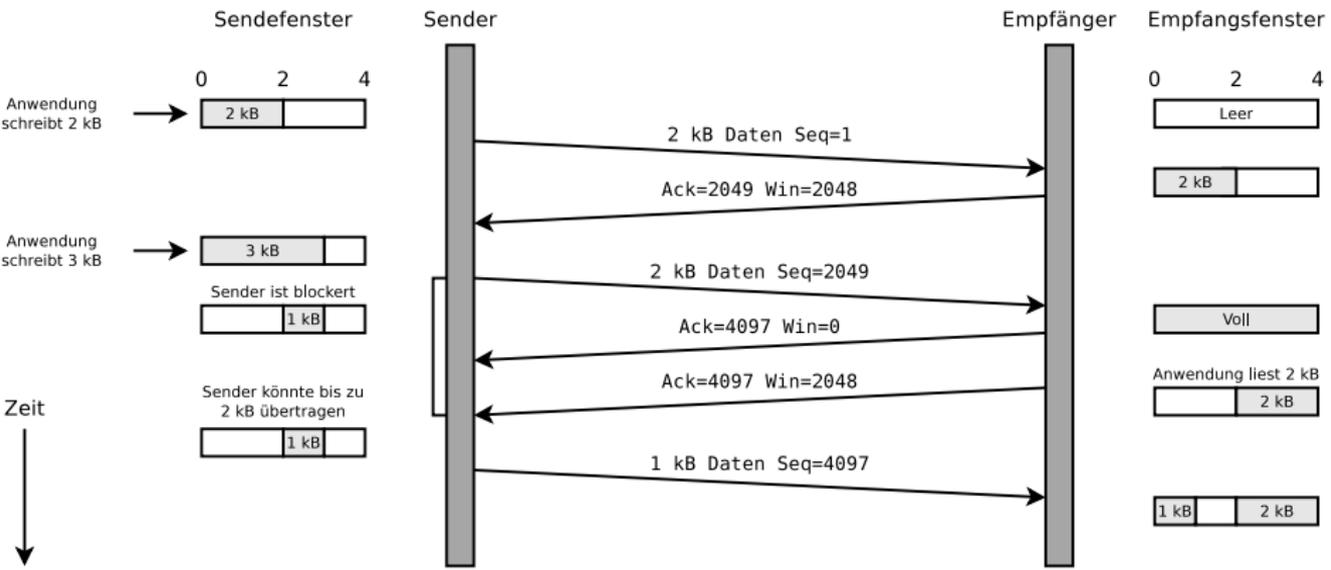
empfangenes Segment



Empfänger teilt hier mit, wie viele Bytes er empfangen kann

Beispiel zur Flusskontrolle bei TCP

- Empfänger informiert in jedem Segment über das freie Empfangsfenster
- Ist das Empfangsfenster voll, ist der Sender blockiert, bis er vom Empfänger erfährt, dass im Empfangsfenster freier Speicher ist
- Wird Kapazität im Empfangsfenster frei \implies **Fensteraktualisierung**



Silly Window Syndrom

- Gefahr des **Silly Window Syndrom**, bei dem sehr viele kleine Segmente geschickt werden, was den Protokoll-Overhead vergrößert
 - Szenario:
 - Ein überlasteter Empfänger mit vollständig gefülltem Empfangspuffer
 - Sobald die Anwendung wenige Bytes (z.B. 1 Byte) aus dem Empfangspuffer gelesen hat, sendet der Empfänger ein Segment mit der Größe des freien Empfangspuffers
 - Der Sender sendet dadurch ein Segment mit lediglich 1 Byte Nutzdaten
 - Overhead: Mindestens 40 Bytes für die TCP/IP-Header jedes IP-Pakets (Nötig sind: 1 Segment mit den Nutzdaten, 1 Segment für die Bestätigung und eventuell noch ein Segment nur für die Fensteraktualisierung)
 - Lösungsansatz: **Silly Window Syndrom Avoidance**
 - Der Empfänger benachrichtigt den Sender erst über freie Empfangskapazität, wenn der Empfangspuffer mindestens zu 25% leer ist oder ein Segment mit der Größe MSS empfangen werden kann

Gründe für Überlastung

- Mögliche Ursachen für Überlastungen:

- ① **Empfängerkapazität**

- Der Empfänger kann die empfangen Daten nicht schnell genug verarbeiten und darum ist sein Empfangspuffer voll
 - Bereits gelöst durch die **Flusskontrolle**

- ② **Netzkapazität**

- Wird ein Computernetz über seine Kapazität hinaus beansprucht, kommt es zu Überlastungen \implies **Congestion Control**
 - Einzige hilfreiche Reaktion bei Überlastungen: **Datenrate reduzieren**
 - TCP versucht Überlastungen durch dynamische Veränderungen der Fenstergröße zu vermeiden \implies **Dynamisches Sliding Window**

- Es gibt nicht *die eine* Lösung für beide Ursachen

- Beide Ursachen werden getrennt angegangen

Anzeichen für Überlastungen der Netzkapazität

- Paketverluste durch Pufferüberläufe in Routern
- Lange Wartezeiten durch volle Warteschlangen in Routern
- Häufige Übertragungswiederholungen wegen Timeout oder Paket-/Segmentverlust

Lösungsansatz gegen Überlastung

• Der Sender verwaltet 2 Fenster

① **Advertised Receive Window** (*Empfangsfenster*)

- Vermeidet Überlast beim Empfänger
- Wird vom Empfänger angeboten (*advertised*)

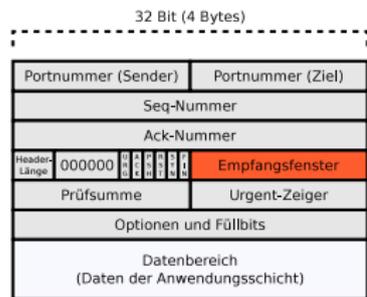
② **Congestion Window** (*Überlastungsfenster*)

- Vermeidet Überlastung des Netzes
- Legt der Sender fest

• Das Minimum beider Fenster ist die maximale Anzahl Bytes, die der Sender übertragen kann

• Beispiel:

- Kann der Empfänger zum Beispiel gemäß seinem Empfangsfenster 20 kB empfangen, aber der Sender erkennt, dass bei mehr als 12 kB das Netz verstopft, dann sendet er nur 12 kB.



• Woher weiß der Sender wie leistungsfähig das Netz ist?
 ⇒ Wie ermittelt der Sender die Größe des Überlastungsfensters?

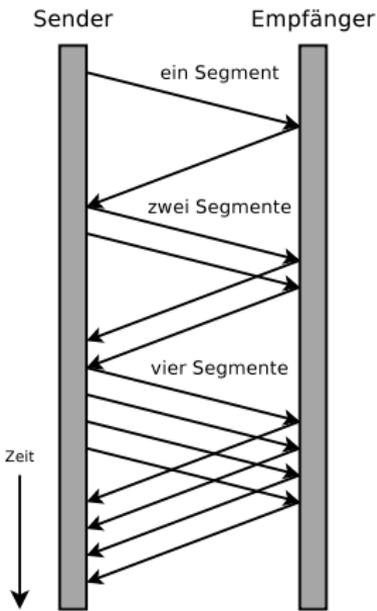
Größe des Überlastungsfensters festlegen

Sie wissen bereits...

- Der Sender kann genau sagen, wie groß das Empfangsfenster ist
- Grund: Der Empfänger teilt es ihm mit jedem Segment mit

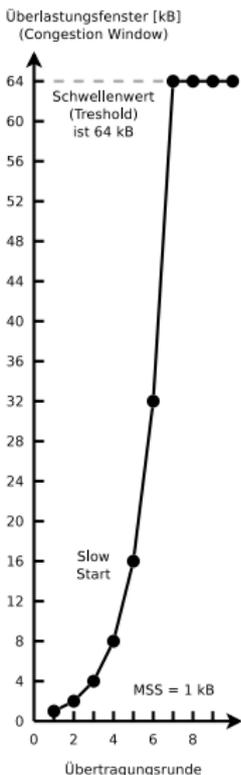
- Problem für den Sender: **Wie groß ist das Überlastungsfenster?**
 - Der Sender weiß zu keiner Zeit sicher, wie leistungsfähig das Netz ist
 - Die Leistungsfähigkeit der Netze ist nicht statisch
 - Sie hängt u.a. von der Auslastung und von Netzstörungen ab
- Lösungsweg: Der Sender muss sich an das Maximum dessen, was das Netzwerk übertragen kann, **herantasten**

Überlastungsfenster festlegen – Verbindungsaufbau



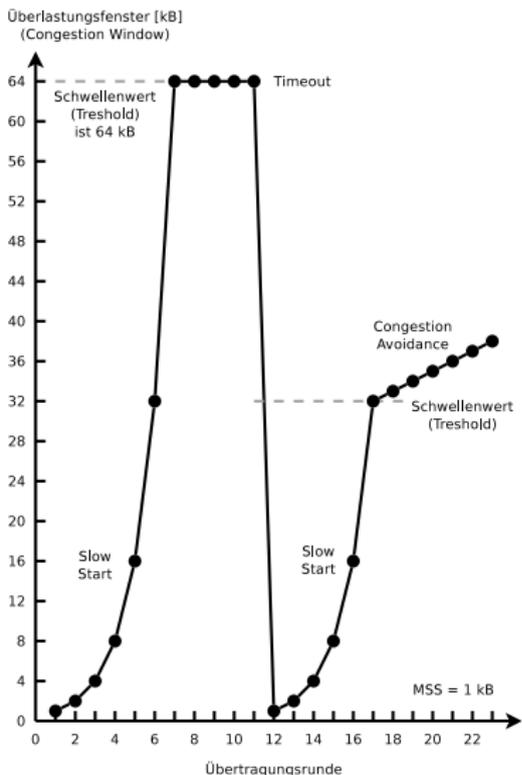
- Beim Verbindungsaufbau initialisiert der Sender das Überlastungsfenster auf die maximale Segmentgröße (MSS)
- Vorgehensweise:
 - 1 Segment mit der Größe MSS senden
 - Wird Empfang des Segments vor dem Timeout bestätigt, wird das Überlastungsfenster verdoppelt
 - 2 Segmente mit der Größe MSS senden
 - Wird der Empfang beider Segmente vor dem Timeout bestätigt, wird das Überlastungsfenster erneut verdoppelt
 - usw.

Überlastungsfenster festlegen – Slow Start



- Das Überlastungsfenster wächst exponentiell bis. . .
 - das vom Empfänger festgelegte Empfangsfenster erreicht ist
 - oder der **Schwellenwert** (Threshold) erreicht ist
 - oder es zum Timeout kommt
- Die exponentielle Wachstumsphase heißt **Slow Start**
 - Grund: Die niedrige Senderate des Senders am Anfang
- Hat das Überlastungsfenster die Größe des Empfangsfensters erreicht, wächst es nicht weiter
- Der Schwellenwert ist am Anfang der Übertragung 2^{16} Byte = 64 kB, damit er zu Beginn keine Rolle spielt
 - Das Empfangsfenster ist maximal $2^{16} - 1$ Bytes groß
 - Ist durch die Größe des Datenfelds **Empfangsfenster** im TCP-Header festgelegt

Überlastungsfenster festlegen – Congestion Avoidance

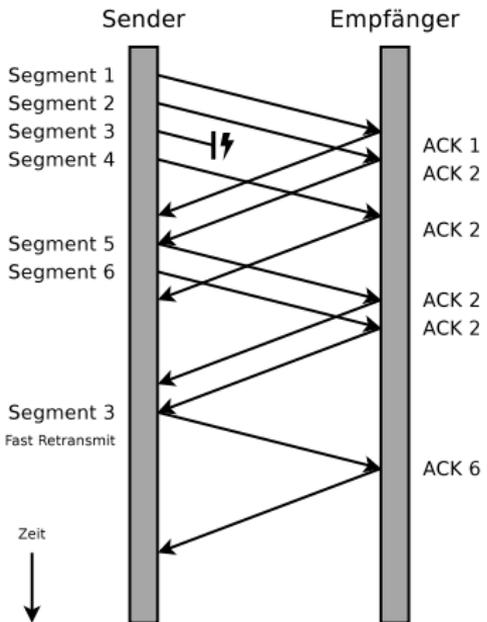


- Kommt es zum Timeout, wird...
 - der Schwellenwert auf die Hälfte des Überlastungsfensters gesetzt
 - und das Überlastungsfenster auf die Größe 1 MSS reduziert
- Es folgt erneut die Phase Slow Start
 - Wird der Schwellenwert erreicht, wächst das Überlastungsfenster linear bis...
 - das vom Empfänger festgelegte Empfangsfenster erreicht ist
 - oder es zum Timeout kommt
- Die Phase des linearen Wachstums heißt **Congestion Avoidance**

Gründe für einen Timeout und sinnvolles Vorgehen

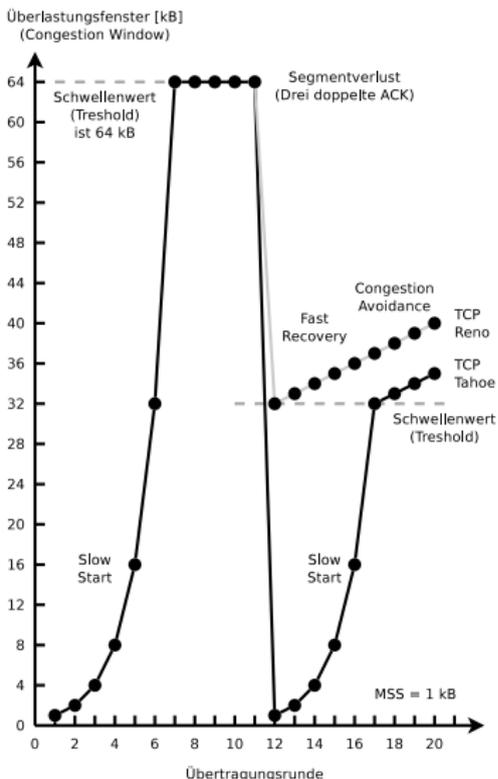
- Ein **Timeout** kann verschiedene Gründe haben
 - Überlast (\implies Verzögerung)
 - Verlust der Sendung
 - Verlust der Bestätigung (ACK)
- Nicht nur Verzögerungen durch Überlast, sondern auch jedes Verlustereignis reduziert das Überlastungsfenster auf 1 MSS
 - Entspricht dem Vorgehen der veralteten TCP-Version *Tahoe* (1988)
- Modernere TCP-Versionen unterscheiden zwischen...
 - Timeout wegen Netzüberlast
 - und **mehrfachem Eintreffen von Bestätigungen** (ACKs) wegen Verlustereignis

Fast Retransmit



- Geht ein Segment verloren, entsteht im Datenstrom beim Empfänger eine *Lücke*
 - Der Empfänger sendet bei jedem weiteren nach dieser Lücke empfangenen Segment ein ACK für das Segment vor dem verlorenen Segment
- Beim Segmentverlust ist eine Reduzierung des Überlastungsfensters auf 1 MSS unnötig
 - Grund: Für einen Segmentverlust ist nicht zwingend Überlastung verantwortlich
- TCP *Reno* (1990) sendet nach **dreimaligem Empfang eines doppelten ACK** das verlorene Segment neu
 ⇒ **Fast Retransmit**

Fast Recovery



- TCP *Reno* vermeidet auch die Phase Slow Start nach dreimaligem Empfang eines doppelten ACK
 ⇒ **Fast Recovery**
- Das Überlastungsfenster wird nach dreimaligem Empfang eines doppelten ACK direkt auf den Schwellenwert gesetzt
 - Das Überlastungsfenster wächst mit jeder bestätigten Übertragung linear...
 - bis das vom Empfänger festgelegte Empfangsfenster erreicht ist
 - oder es zum Timeout kommt

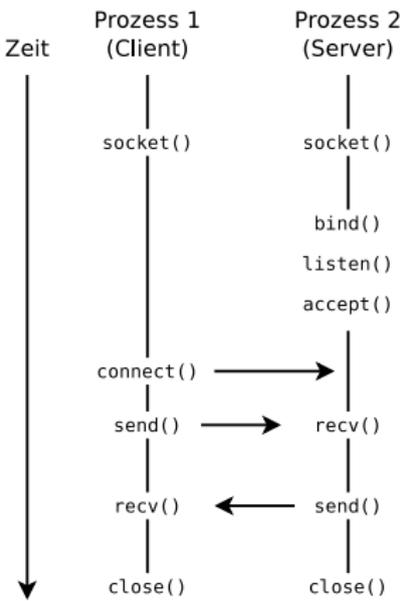
Additive Increase / Multiplicative Decrease (AIMD)

- AIMD ist das Prinzip/Konzept der Überlastkontrolle bei TCP
 - **Rasche Reduzierung** des Überlastungsfensters nach Timeout oder Verlustereignis und **langsames (lineares) Anwachsen** des Überlastungsfensters
- Grund für **aggressive Senkung** und **konservative Erhöhung** des Überlastungsfensters:
 - Die Folgen eines zu großen Überlastungsfensters sind schlimmer als die eines zu kleinen Fensters
 - Ist das Fenster zu klein, bleibt verfügbare Bandbreite ungenutzt
 - Ist das Fenster zu groß, gehen Segmente verloren und müssen erneut übertragen werden
 - Das vergrößert die Überlastung des Netzes noch mehr!
- Möglichst rasch muss der Zustand der Überlastung verlassen werden
 - Darum wird die Größe des Überlastungsfensters deutlich reduziert

Zusammenfassung zu Flusskontrolle und Überlastkontrolle

- Mit **Flusskontrolle** versucht TCP die Bandbreite eines verbindungslosen Netzes (\implies IP) effizient zu nutzen
 - Schiebefenster beim Sender (**Sendefenster**) und Empfänger (**Empfangsfenster**) dienen als Puffer zum Senden und Empfangen
 - Der Empfänger kontrolliert das Sendeverhalten des Senders
- Gründe für Überlastungen: **Empfangskapazität** und **Netzkapazität**
 - Empfangsfenster vermeidet Überlast beim Empfänger
 - Überlastungsfenster vermeidet Überlastung des Netzes
 - Effektiv verwendetes Fenster = Minimum beider Fenster
- Versuch der Maximierung der Netzauslastung und der schnellen Reaktion bei Überlastungsanzeichen
 - Prinzip des **Additive Increase / Multiplicative Decrease** (AIMD)

Verbindungsorientierte Kommunikation mit Sockets – TCP



• Client

- Socket erstellen (`socket`)
- Client mit Server-Socket verbinden (`connect`)
- Daten senden (`send`) und empfangen (`recv`)
- Socket schließen (`close`)

• Server

- Socket erstellen (`socket`)
- Socket an einen Port binden (`bind`)
- Socket empfangsbereit machen (`listen`)
 - Richtete eine Warteschlange für Verbindungen mit Clients ein
- Server akzeptiert Verbindungen (`accept`)
- Daten senden (`send`) und empfangen (`recv`)
- Socket schließen (`close`)

