TECHNISCHE UNIVERSITÄT DARMSTADT

Kommunikationsnetze 1

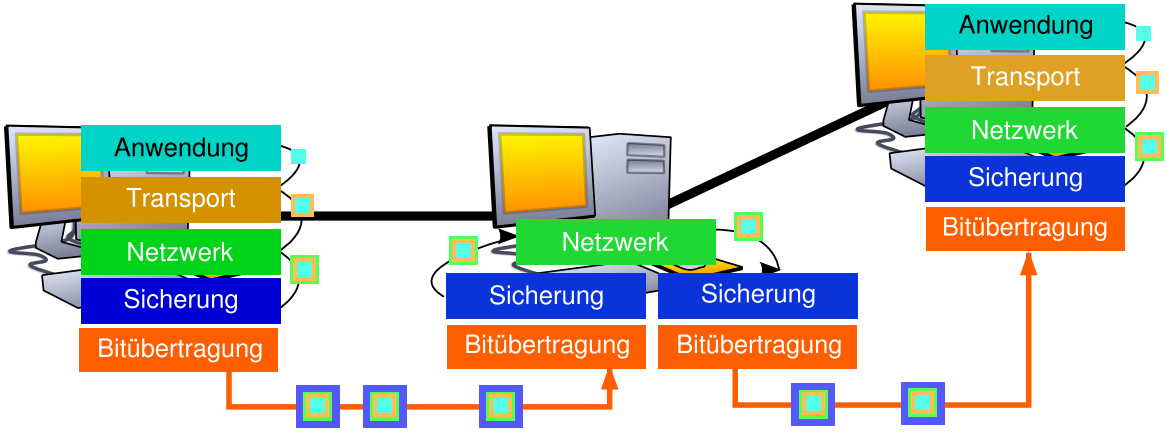
Kapitel 3: Transportprotokolle

Sommersemester 2024
Björn Scheuermann

Transportschicht: Einordnung

TECHNISCHE UNIVERSITÄT DARMSTADT

- Kommunikation von Anwendung zu Anwendung
- . . . und oft noch viel mehr



Allgemeines

TECHNISCHE UNIVERSITÄT DARMSTADT

- Die Transportschicht nutzt den Dienst der Netzwerkschicht und stellt ihrerseits der Anwendungsschicht Dienste zur Verfügung
  - im Internet setzt die Transportschicht also auf einem unzuverlässigen Datagrammtransport von Host zu Host auf
- Im Internet sind die Protokolle der Transportschicht *Ende-zu-Ende-Protokolle*
  - die Rechner im Netzwerkinneren „interessieren“ sich (eigentlich) nicht dafür
  - (es gibt durchaus Ausnahmen von dieser Regel…)

Dienstmodelle

TECHNISCHE UNIVERSITÄT DARMSTADT

- Verschiedene Transportprotokolle können unterschiedliche Dienstmodelle anbieten, oder sie können diese unterschiedlich implementieren
- Je nach Transportschichtprotokoll kann das Dienstmodell z. B. umfassen
  - Bytestromtransport oder Übertragung isolierter Nachrichten/Pakete
  - zuverlässigen oder unzuverlässigen Datentransport
  - Reihenfolgeerhaltung
  - zugesicherte feste oder minimal verfügbare Datenraten
  - Garantien zu Latenzzeiten und ihren Schwankungen (Jitter)
  - Zeitdauer bis zum abgeschlossenen Verbindungsaufbau
  - Schutz gegen Mithören, Verändern, Wiederholen, Unterdrücken,. . . von Daten
  - Priorisierung von Daten/Verbindungen
  - maximal entstehende Kosten für die Verbindung
  - . . .

Dienstmodelle

TECHNISCHE UNIVERSITÄT DARMSTADT

- Einige dieser Dienstmerkmale können unabhängig vom Dienstmodell der Netzwerkschicht realisiert werden
- Andere sind nur mit Unterstützung der darunterliegenden Schichten realisierbar
- …und damit im Internet oft gar nicht!
- Der Anwender muss sich überlegen, welche(s) Protokoll(e) er für seine Anwendung einsetzen möchte

UDP und TCP

TECHNISCHE UNIVERSITÄT DARMSTADT

- Im Internet sind zwei Transportprotokolle verbreitet:
  - das *User Datagram Protocol (UDP)*
    - datagrammorientiert
    - unzuverlässiger Best-Effort-Dienst
  - das *Transmission Control Protocol (TCP)*
    - bytestromorientiert
    - zuverlässig
    - …
- Weitere Transportprotokolle über IP sind möglich (und existieren), praktisch eingesetzt werden sie aber selten

Sockets und APIs

TECHNISCHE UNIVERSITÄT DARMSTADT

- Damit eine Anwendung mit dem Transportschichtprotokoll „sprechen“ kann, ist eine Programmierschnittstelle (*Application Programming Interface, API*) notwendig
- Die am weitesten verbreitete API für die Protokolle der TCP/IP-Familie ist die *Socket-API*
- Sie ermöglicht es Anwendungen z. B., TCP-Verbindungen auf- oder abzubauen und Daten darüber zu verschicken und zu empfangen
- Sehr flexibel (deswegen aber auch etwas gewöhnungsbedürftig und manchmal ein wenig „unhandlich“)

Sockets und APIs

TECHNISCHE UNIVERSITÄT DARMSTADT

- Je nach Programmiersprache können auch andere APIs zur Verfügung stehen
- Auch viele andere Programmierschnittstellen haben aber den Begriff *Socket* für einen Transportschicht-Endpunkt übernommen
- Hat sich deshalb generell eingebürgert
- Wichtig: Anwendungen können durchaus mehrere Sockets verwenden, z. B. wenn sie gleichzeitig mehrere Kommunikationsbeziehungen unterhalten

Kontaktaufbau

TECHNISCHE UNIVERSITÄT DARMSTADT

- Wenn zwei Rechner miteinander kommunizieren, muss einer von beiden den „ersten Schritt“ machen
- Im Kontext von Transportschichtprotokollen:
  - *Server* = Rechner, der auf eingehende Kontaktaufnahme wartet
  - *Client* = Rechner, der von sich aus Kontakt zu einem Server aufnimmt
- Dafür muss der Client die Netzwerkschicht-Adresse und die Portnummer des Servers kennen
  - es gibt viele Möglichkeiten (die auch genutzt werden)
  - häufiger Fall in TCP/IP-Netzen: IP-Adresse per DNS + von der Anwendung „üblicherweise“ verwendeter Port
- Peer-to-Peer-Anwendungen enthalten im selben Programm sowohl Client- als auch Serverfunktionalität
  - für einzelne Kommunikationsbeziehungen spricht man dann dennoch vom „Client“ und „Server“

Portnummern und Transportschicht-Multiplexing

TECHNISCHE UNIVERSITÄT DARMSTADT

- Die Transportschicht soll, basierend auf Host-zu-Host-Kommunikation, eine Anwendung-zu-Anwendung-Kommunikation (bzw. präziser: Socket-zu-Socket-Kommunikation) ermöglichen
- Dafür müssen den übertragenen Daten Informationen hinzugefügt werden, die den Ziel-Socket identifizieren
- Im Internet: *Portnummern* im Transportschicht-Header
- Dasselbe Paar von Quell- und Zieladresse auf der Netzwerkschicht kann damit Informationen für viele Kommunikationsbeziehungen auf der Transportschicht transportieren
- Man spricht deshalb von *Transportschicht-Multiplexing*

Portnummern

TECHNISCHE UNIVERSITÄT DARMSTADT

- Als Portnummern werden sowohl in TCP als auch in UDP 16-Bit-Zahlen verwendet, also der Bereich 0–65535
- Manche Portnummern werden üblicherweise für bestimmte Anwendungen verwendet
  - *„Well-known Ports“*
  - z. B. TCP Port 80 für HTTP, TCP Port 25 für SMTP, UDP+TCP Port 53 für DNS
  - Liste wird von der IANA verwaltet
- Eine Sonderrolle haben außerdem die sog. *„privilegierten Ports“* mit den Nummern 0–1023
  - die meisten Betriebssysteme erlauben nur Anwendungen mit Administratorrechten, diese Portnummern lokal zu verwenden

Achtung: TCP Port *x* ist etwas völlig anderes als UDP Port *x* – beide existieren unabhängig voneinander und können auch unabhängig voneinander genutzt werden

Protokollnummern

TECHNISCHE UNIVERSITÄT DARMSTADT

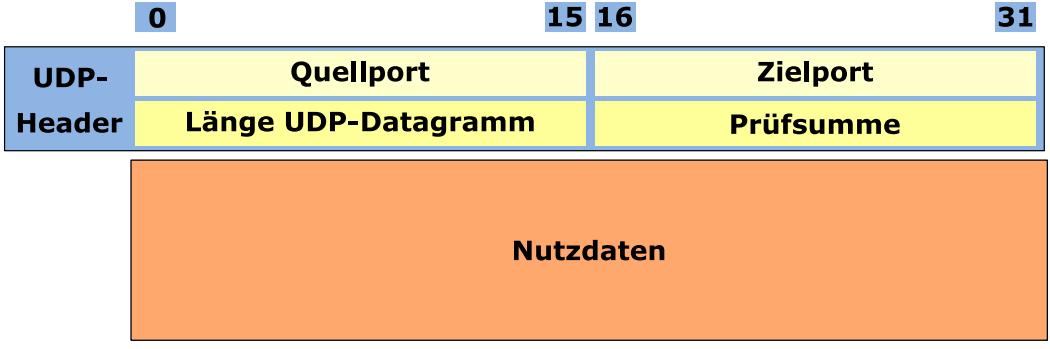
- Auf der Netzwerkschicht existiert ein verwandtes Adressierungsproblem: Das Netzwerkschichtprotokoll muss entscheiden können, an welche von mehreren Transportschichten ein eingehendes Paket weitergereicht werden soll
- In der TCP/IP-Welt wird hierfür die *Protokollnummer* verwendet
- Ein Eintrag im IP-Header gibt an, welches Transportschichtprotokoll verwendet wird, z. B.
  - 6 = TCP
  - 17 = UDP

User Datagram Protocol (UDP)

TECHNISCHE UNIVERSITÄT DARMSTADT

- UDP ist ein nahezu „minimales“ Transportprotokoll
- Verbindungsloser Datagrammdienst, der auf IP aufsetzt
- Spezifiziert in RFC 768
- UDP-Datagramme werden unabhängig voneinander unzuverlässig übertragen
- UDP „erbt“ den Best-Effort-Dienst von IP und fügt außer Portnummern kaum etwas hinzu
  - also insbes. keine Erkennung verlorener oder verdoppelter Pakete

UDP-Paketformat



- Länge = Länge des Datagramms in Byte, einschließlich UDP-Header
- Prüfsumme: damit kann der Empfänger überprüfen, ob das Datagramm korrekt angekommen ist
  - Prüfsumme ist optional; ein Wert von 0 bedeutet, dass keine Prüfsumme verwendet werden soll
  - in der Regel sollte sie aber verwendet werden (und wird sie auch)
  - ein empfangenes UDP-Datagramm mit fehlerhafter Prüfsumme wird verworfen

Transportschicht-(De)Multiplexing in UDP

TECHNISCHE UNIVERSITÄT DARMSTADT

Wie entscheidet ein UDP-Empfänger, an welchen Socket er ein eingehendes Datagramm zustellen soll?

- Er schaut sich erstens die Ziel-Portnummer an
- Bei einem Host mit mehreren IP-Adressen lauschen Sockets manchmal nur an einer davon
- Aus diesem Grund ist zweitens auch die Ziel-IP-Adresse aus dem IP-Header wichtig

Ein UDP-Socket wird eindeutig durch das Tupel aus Ziel-IP-Adresse und Ziel-Port identifiziert

- Datagramme mit unterschiedlichen Quell-IP-Adressen und/oder Quell-Portnummern, aber der selben Ziel-IP-Adresse und dem selben Ziel-Port werden an den selben Socket geliefert

Transportschicht-(De)Multiplexing in UDP

TECHNISCHE UNIVERSITÄT DARMSTADT

Wieviele UDP-Sockets mit derselben Portnummer darf es für eine IP-Adresse geben?

- Es darf maximal einen solchen Socket geben!

Wofür dient die Quell-Portnummer im UDP-Header?

- Eventuell nützlich für die Anwendung: Datagramme von unterschiedlichen Quellen kommen am selben Socket an
- Die Quell-Portnummer (zusammen mit der Quell-IP-Adresse) ist notwendig für das Senden einer Antwort

Wofür UDP?

TECHNISCHE UNIVERSITÄT DARMSTADT

- Der Einsatz von UDP kann sinnvoll sein, wenn es auf (Reaktions-)geschwindigkeit ankommt und mögliche Verluste akzeptabel sind
- Da kein Verbindungsaufbau nötig ist, entsteht hierfür auch keine Verzögerung
- Kommunikation mit mehreren Gegenstellen über einen einzigen Socket
  - Vorteil, wenn mit vielen gleichzeitig kommuniziert wird!
- Kleinen Header, daher nur wenige zusätzliche Bytes für Verwaltungsinformationen (kleiner *Protokolloverhead*)
- UDP kann immer so schnell senden, wie es die Anwendung wünscht (muss natürlich nicht alles ankommen. . .)
  - Vorteil oder Nachteil?
  - wir werden noch sehen, dass das bei TCP anders ist
- UDP wird aus diesen Gründen oft für Multimedia-Protokolle eingesetzt (aber z. B. auch für DNS)

TCP

TECHNISCHE UNIVERSITÄT DARMSTADT

- Das andere wichtige Transportprotokoll im Internet: Transmission Control Protocol (TCP)
- Spezifiziert in RFC 793 und vielen, vielen Erweiterungen
- Das krasse Gegenteil zum simplen UDP: TCP ist ein extrem komplexes Protokoll
- Viele zentrale Probleme aus dem Bereich Netzwerke tauchen im Zusammenhang mit TCP auf – TCP ist deshalb ein schönes Beispiel für viele grundlegende Fragestellungen
- Darum wechseln wir im Folgenden hin und her zwischen
  - grundlegenden Betrachtungen (Welche Probleme gibt es? Welche generellen Lösungsmöglichkeiten existieren?) und
  - der Diskussion der konkreten Mechanismen, die in TCP implementiert sind

TCP

TECHNISCHE UNIVERSITÄT DARMSTADT

- Das Dienstmodell von TCP umfasst die *zuverlässige, reihenfolgeerhaltende Übertragung eines Bytestromes*
- Dafür wird der Bytestrom beim Sender in Pakete (bei TCP *Segmente* genannt) zerlegt und beim Empfänger wieder zusammengesetzt
  - eine solche Zerlegung nennt man allgemein *Segmentierung*, das Zusammenfügen *Reassemblierung*
- TCP ermöglicht *Vollduplex-Kommunikation*
  - es werden also Byteströme in beide Richtungen übertragen
- Für viele Zwecke können wir die beiden Richtungen unabhängig betrachten und uns auf einen Bytestrom in eine Richtung konzentrieren
  - das macht die Dinge etwas einfacher
  - dabei aber im Hinterkopf behalten, dass das dann immer in gleicher Weise in beide Richtungen passiert
- TCP unterstützt nur *Unicast-Kommunikation* zwischen genau zwei Gegenstellen, also keine Übertragungen desselben Bytestroms an mehrere Empfänger

Verbindungen in TCP

TECHNISCHE UNIVERSITÄT DARMSTADT

- Für einen zuverlässigen, reihenfolgeerhaltenden Dienst ist es notwendig, dass Sender und Empfänger einen Zusammenhang zwischen den Paketen sehen, die sie austauschen
- Man könnte auch sagen, dass ein solches Protokoll zwangsläufig zustandsbehaftet sein muss (im Gegensatz zum zustandslosen Transport unabhängiger Datagramme bei UDP)
- Deshalb müssen sich Sender und Empfänger einig darüber sein, welche Segmente zusammengehören
  - wir brauchen also *Verbindungen*, die explizit aufgebaut (und abgebaut) werden müssen
- Wie das genau funktioniert, schauen wir uns gleich genauer an
- Zunächst hat das aber wichtige Folgen für das Transportschicht-Multiplexing

Transportschicht-Multiplexing TCP

TECHNISCHE UNIVERSITÄT DARMSTADT

Darf TCP Segmente, die von unterschiedlichen Quellen stammen, an denselben Ziel-Socket zustellen?

- Daten von unterschiedlichen Absender-Sockets beim Empfänger zu „mischen“ macht bei einem zuverlässigen Bytestrom-Dienst keinen Sinn – also Nein!

Wie entscheidet ein TCP-Empfänger, an welchen Socket ein eingehendes Segment gehen soll?

- Anhand des 4-Tupels (Quell-IP-Adresse, Quell-Portnummer, Ziel-IP-Adresse, Ziel-Portnummer)
- Hier sind also Quell-Portnummer und Quell-IP-Adresse wichtig für die richtige Zuordnung
- Eine Teilmenge des 4-Tupels ist im Allgemeinen nicht ausreichend – anders als beim verbindungslosen UDP!

Transportschicht-Multiplexing TCP

TECHNISCHE UNIVERSITÄT DARMSTADT

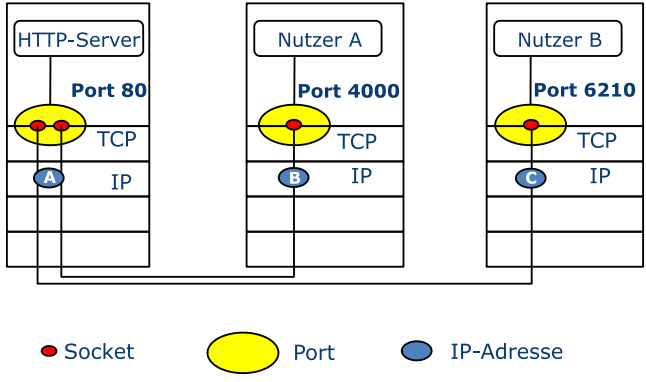
Kann es mehrere gleichzeitig aktive TCP-Sockets
mit derselben Portnummer für dieselbe (lokale) IP-Adresse geben?

- Ja – das kommt sogar sehr häufig vor!
- Wenn ein Server mit mehreren Clients spricht, dann sprechen alle mit demselben Port
- …aber für jeden der Clients wird ein eigener Socket erzeugt

Beispiel für HTTP-Verbindungen

TECHNISCHE UNIVERSITÄT DARMSTADT

TCP-Verbindungen sind eindeutig durch das 4-Tupel
(Ziel-Port, Ziel-IP-Adresse, Quell-Port, Quell-IP-Adresse) unterscheidbar



Zuverlässige Datenübertragung

TECHNISCHE UNIVERSITÄT DARMSTADT

- Schon mehrfach ist hier das Wort „zuverlässig“ gefallen
- An dieser Stelle ist es sinnvoll, genauer zu schauen, was damit eigentlich gemeint ist

Was genau bedeutet wohl „zuverlässig“ im Zusammenhang mit TCP?

- „Zuverlässig“ heißt hier *nicht*, dass Daten garantiert ankommen!
  - das kann man gar nicht garantieren: Wenn ich den Stecker ziehe, dann kommen sie garantiert niemals an… :-)
- Es bedeutet, dass der Sender irgendwann eine Bestätigung bekommt, dass die Daten zugestellt wurden – oder zumindest eine Fehlermeldung, wenn eben das nicht sichergestellt werden konnte

Zuverlässige Datenübertragung

TECHNISCHE UNIVERSITÄT DARMSTADT

- TCP bietet Zuverlässigkeit im eben diskutierten Sinne
- Zuverlässige Datenübertragung über einen unzuverlässigen Übertragungskanal ist eine elementar wichtige Fragestellung für das Internet (und andere Kommunikationsnetze)
- Es lohnt sich deshalb, sich für dieses Thema etwas Zeit zu nehmen
- Wir tasten uns an das Problem schrittweise heran, betrachten einige generelle Lösungsansätze und ihre Vor- und Nachteile, und schauen uns schließlich die Umsetzung in TCP als konkretes Anwendungsbeispiel an

Zuverlässige Übertragungskanäle

TECHNISCHE UNIVERSITÄT DARMSTADT

- Wir machen das zugrundeliegende Netzwerk Schritt für Schritt „schwieriger“ und überlegen jeweils, was für das Erreichen von Zuverlässigkeit notwendig ist

- Wir fangen mit einem absolut zuverlässigen Übertragfungskanal an: Alles, was abgeschickt wird, kommt garantiert auch richtig an

Wie würden Sie über einen solchen Übertragungskanal ein zuverlässiges Transportprotokoll realisieren?

Kanäle mit Übertragungsfehlern

TECHNISCHE UNIVERSITÄT DARMSTADT

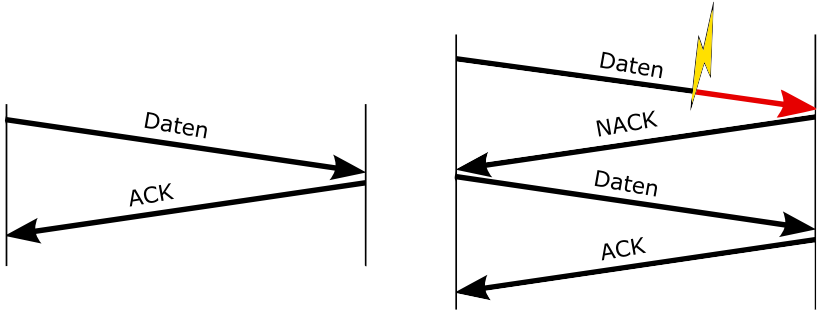
- Ok, das war zu einfach…
- Machen wir das ganze etwas anspruchsvoller: Angenommen, auf dem Kanal können Übertragungsfehler auftreten
- Der Empfänger wird jedes Paket bekommen, aber die Pakete sind möglicherweise „kaputt“
- Der Empfänger kann für jedes Paket feststellen, ob es korrekt empfangen wurde oder ob ein Fehler aufgetreten ist

Wie würden Sie in diesem Fall Zuverlässigkeit realisieren?

Kanäle mit Übertragungsfehlern

TECHNISCHE UNIVERSITÄT DARMSTADT

- Idee: Bestätigungsnachrichten
- Entweder eine positive Bestätigung des korrekten Empfangs (*Acknowledgment, ACK*)
- . . . oder ein „negatives ACK“ (*NACK*) zum Mitteilen eines Fehlers
- Beim Empfang eines NACK schickt der Sender das Paket nochmals



Kanäle mit Übertragungsfehlern

TECHNISCHE UNIVERSITÄT DARMSTADT

- Problem: Auch ACKs oder NACKs sind Pakete und können kaputtgehen!
- Dann kann die Senderseite nicht entscheiden, ob die Übertragung erfolgreich war oder nicht
- Wenn das Paket dann neu übertragen wird, kann es sein, dass es doppelt beim Empfänger ankommt

Wie können wir das Protokoll so erweitern, dass der Empfänger das feststellen kann?

Sequenznummern

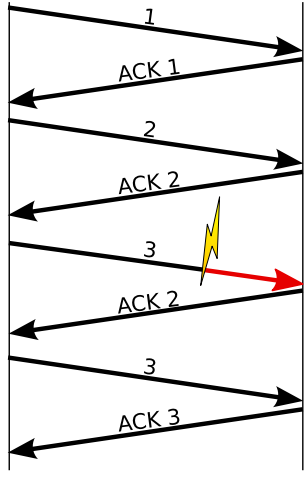
TECHNISCHE UNIVERSITÄT DARMSTADT

- Wir können die Pakete durchnummerieren
- An der *Sequenznummer* kann der Empfänger erkennen, dass er das Paket bereits hat, und kann das Duplikat verwerfen
- Wenn wir das umsetzen, dann können wir uns die NACKs sparen und nur mit ACKs arbeiten – das vereinfacht das Protokoll
- Dafür schreiben wir in das ACK-Paket jeweils die Sequenznummer des letzten korrekt empfangenen Paketes

Sequenznummern

TECHNISCHE UNIVERSITÄT DARMSTADT

- Bei einem fehlerhaft empfangenen Paket (oder einem Duplikat) wird einfach das letzte ACK wiederholt
- Ein „altes“ ACK signalisiert also dem Sender, dass das Paket nicht korrekt angekommen ist



Wiederholung von Sequenznummern

TECHNISCHE UNIVERSITÄT DARMSTADT

- Sequenznummern werden in vielen Protokollen benutzt
- Sie richtig zu wählen, kann schwierig sein!
- Sequenznummern sind endliche Zahlen – bei langen Verbindungen werden sie sich irgendwann wiederholen
  - es muss jederzeit sichergestellt sein, dass kein anderes Paket mit derselben Sequenznummer beim Empfänger ankommt
- Die Zeit bis zu einer Wiederholung muss länger sein, als ein Paket im Netz maximal „überlebt“ haben kann
- Gar nicht so einfach abzuschätzen (vor allem für zukünftige Netze, über die das Protokoll einmal verwendet werden könnte!)
  - hängt auch von der Rate ab, mit der Sequenznummern „abgearbeitet“ werden
  - also von der Übertragungsgeschwindigkeit im Netz

Wiederholung von Sequenznummern

TECHNISCHE UNIVERSITÄT DARMSTADT

- Auch die Wahl der Sequenznummer für das erste Paket ist ein interessantes Problem
- Ganz schlechte Idee: immer bei 1 (oder 0) anfangen
- Warum: Viele Netze (z. B. das Internet) garantieren keine maximale Zustelldauer: Was, wenn noch Pakete einer früheren Verbindung „herumgeistern“?
- Wir könnten uns theoretisch merken, welche Sequenznummer wir zuletzt benutzt hatten, und bei der neuen Verbindung mit der nächsthöheren weitermachen
- Aber was, wenn der Rechner abgestürzt war und die vorher verwendeten Sequenznummern „vergessen“ hat?
- Idee: nach jedem Systemabsturz mindestens so lange gar nichts senden, bis garantiert keine alten Pakete mehr „überlebt“ haben
- Elegant? Eher nein. . .

Sequenznummern und Sicherheit

TECHNISCHE UNIVERSITÄT DARMSTADT

- Ein Angreifer, der in der Lage ist, die richtige Sequenznummer zu „raten“, kann gefälschte Pakete in eine fremde (TCP-)Verbindung einschleusen, selbst wenn er den Datenverkehr nicht mitlesen kann
- Deshalb ist die Wahl der Startsequenznummer auch eine Sicherheitsfrage: sollte nicht vorhersehbar sein
- Heute oft praktiziert:
  - Startsequenznummer zufällig wählen
  - ausreichend großen Sequenznummernbereich verwenden
  - und das Beste hoffen

Kanäle mit Paketverlusten

TECHNISCHE UNIVERSITÄT DARMSTADT

- Zurück zu unseren unzuverlässigen Netzen
- Nehmen wir jetzt einen Übertragungskanal an, der jederzeit Pakete verlieren kann
- Das ist die Situation, wie sie im Internet vorliegt

Welches neue Problem ergibt sich jetzt?

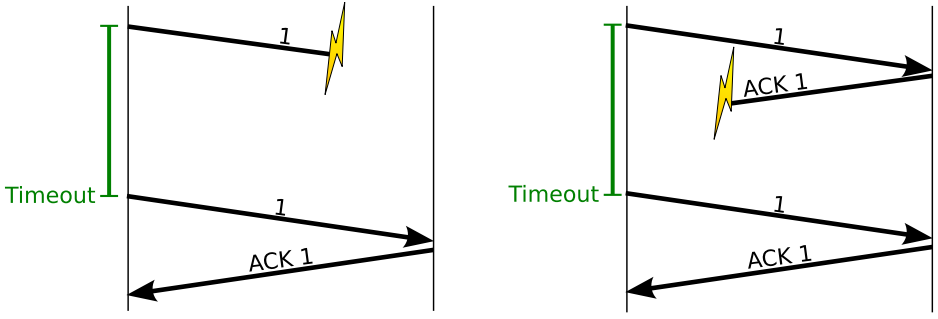
- Es kann sein, dass der Sender überhaupt keine Rückmeldung bekommt
  - entweder weil sein Paket verloren gegangen ist
  - . . . oder weil das ACK verloren ging

Ihr Lösungsvorschlag?

Kanäle mit Paketverlusten

TECHNISCHE UNIVERSITÄT DARMSTADT

- Der Sender wartet eine „vernünftige Zeitspanne“ (*Retransmission Timeout, RTO*) und überträgt das Paket dann erneut
- Passiert sowohl beim Verlust des Datenpaketes als auch beim Verlust eines ACK
  - . . . denn der Sender kann diese beiden Fälle nicht unterscheiden!



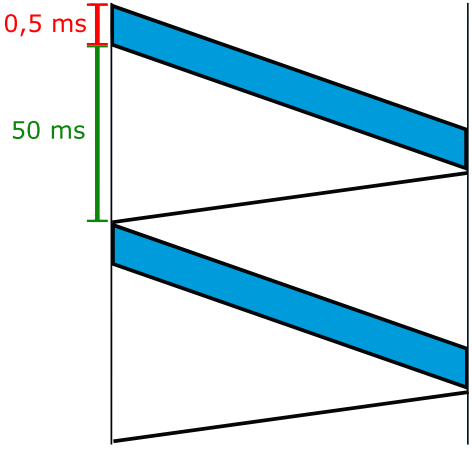
Stop-and-Wait?

TECHNISCHE UNIVERSITÄT DARMSTADT

- Bisher sind wir (implizit) immer davon ausgegangen, dass der Sender ein Paket schickt und dann auf Rückmeldung vom Empfänger wartet
- Dieses Vorgehen nennt man *Stop-and-Wait*
- Die Frage ist: Ist Stop-and-Wait (im Internet) sinnvoll?
- Nehmen Sie folgende Situation an:
  - zwischen zwei Hosts ist die RTT gleich 50 ms
  - das Netz kann zwischen den Hosts 2 MB/s übertragen
  - ein Paket enthält 1000 Byte Daten
  - Header und ACKs sind vernachlässigbar klein

Wie viele Daten pro Zeiteinheit werden mit Stop-and-Wait zugestellt?

Stop-and-Wait



- Das Abschicken eines Paketes dauert

$$\frac{1\,000\ \text{Byte}}{2\,000\,000\ \text{Byte/s}} = 0{,}5\,\text{ms}$$

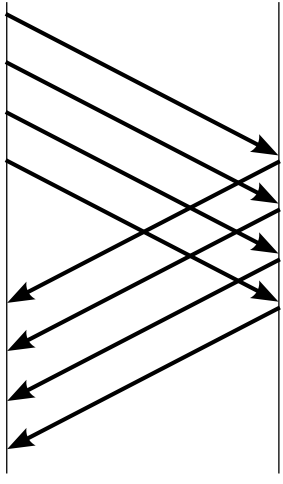
- Sobald das letzte Bit abgeschickt ist, dauert es noch 1 RTT ($= 50$ ms), bis das ACK eintrifft, danach beginnt der Versand des nächsten Paketes

- Es wird also alle
  $50\,\text{ms} + 0{,}5\,\text{ms} = 50{,}5\,\text{ms}$
  ein Paket von 1000 Byte verschickt

- Von den 2 MB/s werden damit nur
  $1\,000\ \text{Byte} \,/\, 0{,}0505\,s = 19\,801{,}98\ \text{Byte/s}$
  sinnvoll genutzt – das ist unter 1%!

Pipelining

- Lösung: Der Sender verschickt weitere Pakete, schon bevor das erste ACK eintrifft
- Das nennt man dann *Pipelining* (wie bei HTTP…)
- Die Folge: der Sender muss mehrere Pakete zwischenspeichern, um sie ggf. neu übertragen zu können
- Die nächsten Fragen ist damit klar: Wie soll ein solches Protokoll auf fehlende Pakete bzw. ACKs reagieren?
- Das schauen wir uns zunächst allgemein, dann konkret für TCP an



Grundschemata für Zuverlässigkeit mit Pipelining

TECHNISCHE UNIVERSITÄT DARMSTADT

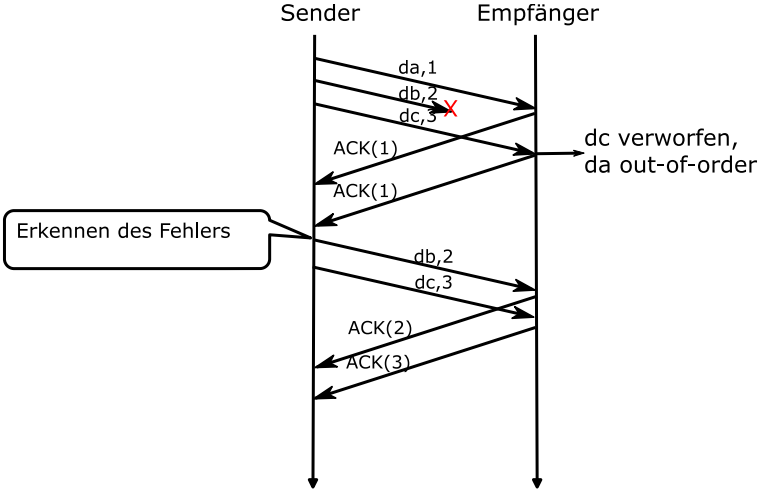
- Für die zuverlässige Datenübertragung mit Pipelining gibt es zwei weit verbreitete „Grundschemata“:
  - Go-back-n
  - Selective Repeat
- …und viele Varianten davon

Go-Back-n

TECHNISCHE UNIVERSITÄT DARMSTADT

- Sender:
  - ACK für Paket *x* bedeutet, dass auch alle vorangegangenen Pakete erfolgreich angekommen sind (*kumulatives ACK*)
  - Fehler werden durch einen Timeout erkannt
  - überträgt bei einem erkannten Fehler für Paket *x* alle Pakete ab *x* erneut
- Empfänger:
  - quittiert bei jedem Empfang das letzte in der richtigen Reihenfolge empfangene Paket (ggf. also viele ACKs mit derselben Sequenznummer)
  - verwirft alle Pakete, die nicht in der richtigen Reihenfolge ankommen
    - Variante/Erweiterung: Puffern von Paketen „außer der Reihe“ kann ggf. die Effizienz steigern

Go-Back-n

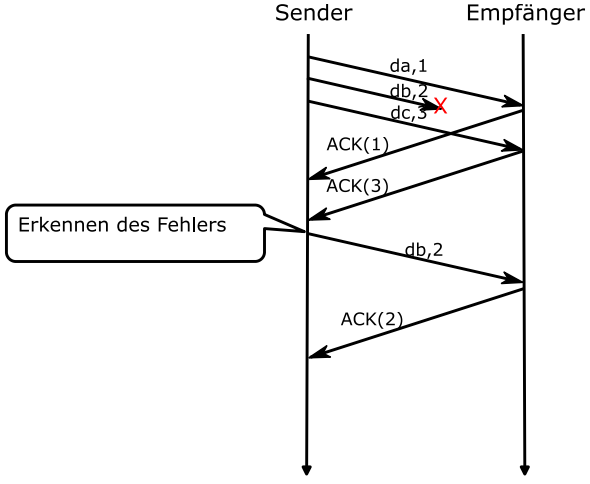

Sender
Empfänger
da,1
db,2
dc,3
X
dc verworfen,
da out-of-order
ACK(1)
ACK(1)
Erkennen des Fehlers
db,2
dc,3
ACK(2)
ACK(3)

Selective Repeat

TECHNISCHE UNIVERSITÄT DARMSTADT

- Sender:
  - erkennt Fehler an Lücken in Quittierung
  - wiederholt nur Übertragung nicht quittierter Pakete
  - Erkennung fehlender ACKs durch Sende-Timeouts
- Empfänger:
  - speichert alle empfangenen Pakete
  - quittiert jedes Paket
- Aufwändiger als Go-back-n – aber (je nach Umgebung, Einsatzzweck) evtl. auch effizienter

Selective Repeat



Zuverlässigkeit in TCP

TECHNISCHE UNIVERSITÄT DARMSTADT

- Schauen wir uns konkret an, wie Zuverlässigkeit in TCP implementiert ist
- Die TCP-Zuverlässigkeitsmechanismen orientieren sich (in erster Näherung) an Go-back-n
- Nicht vergessen: TCP überträgt in beide Richtungen je einen Bytestrom
  - kein Problem: die beiden Richtungen sind unabhängig
- TCP nutzt 32-Bit-Sequenznummern
- Für jede Richtung werden unabhängige Sequenznummern verwendet

ACKs in TCP

TECHNISCHE UNIVERSITÄT DARMSTADT

- Im TCP-Header gibt es ein ACK-Feld
  - genauer: ACK-Flag + ACK-Sequenznummernfeld
- Jedes TCP-Segment in eine Richtung kann gleichzeitig ein ACK in die andere Richtung sein (*piggybacked ACKs*)
- Wenn in die Gegenrichtung gerade keine Daten zu verschicken sind, dann ist ein ACK ein leeres TCP-Segment (Nutzdatenlänge 0) mit gesetztem ACK-Feld
- ACKs in TCP sind kumulativ

Kumulative ACKs

TECHNISCHE UNIVERSITÄT DARMSTADT

- Da ein (kumulatives!) ACK immer auch alle früheren Segmente bestätigt, können Segmente hinter einer „Lücke“ oder in falscher Reihenfolge eintreffende Segmente nicht bestätigt werden
- TCP wiederholt dann – genau wie klassisches Go-Back-n – jeweils einmal das vorherige ACK
- Die TCP-Spezifikation schreibt nicht vor, ob außer der Reihe eintreffende Segmente vom Empfänger zwischengespeichert werden
  - sie dürfen auch verworfen werden – wegen Neuübertragungen werden sie noch einmal ankommen
  - heutige TCP-Implementationen puffern solche Daten aber praktisch immer zwischen

Bytesequenznummern

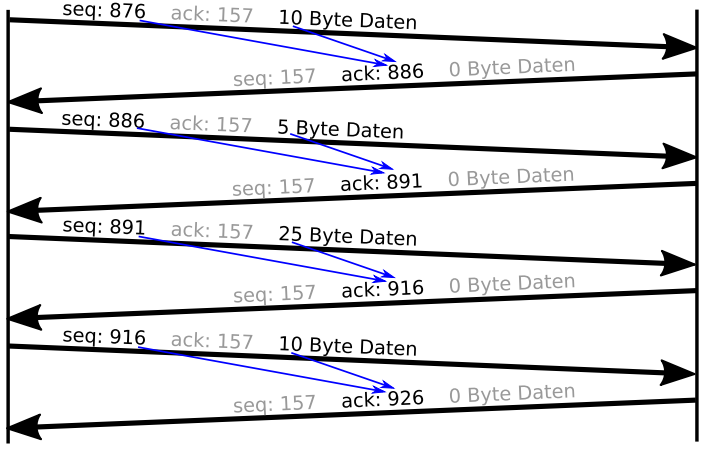
TECHNISCHE UNIVERSITÄT DARMSTADT

- TCP nummeriert nicht Segmente, sondern alle Bytes im übertragenen Bytestrom durch
- Zahl im Sequenznummern-Feld = Sequenznummer des ersten im Segment enthaltenen Bytes
  - hat ein Segment die Sequenznummer 3217 und enthält 500 Bytes Daten, dann hat das nächste Segment die Sequenznummer 3717
- Zahl im ACK-Feld = Sequenznummer des ersten Bytes an, das dem Sender des ACK im Bytestrom *noch fehlt*
  - also *nicht* (!!!) die höchste bereits empfangene Sequenznummer

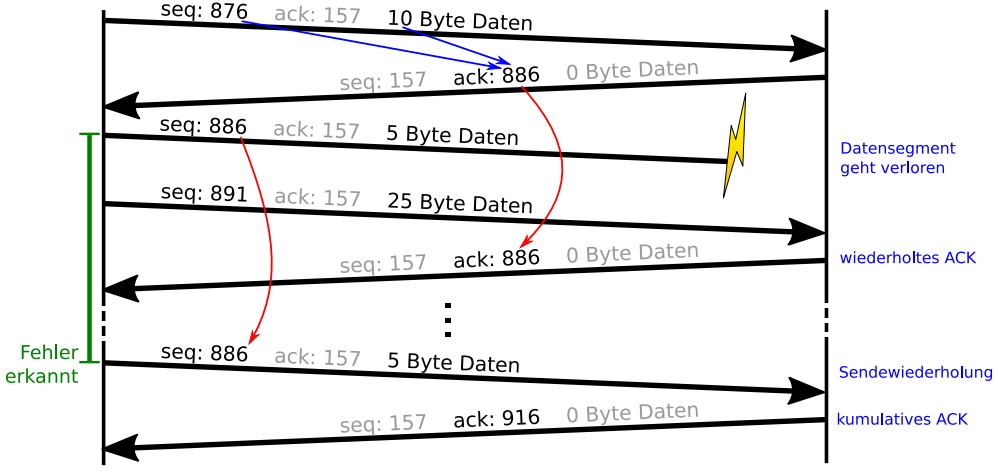
Beispiel: Störungsfreie Datenübertragung

TECHNISCHE UNIVERSITÄT DARMSTADT

(Annahmen: sofortige Bestätigung jedes Pakets, kein Pipelining)



Beispiel: Datenübertragung mit Paketverlust



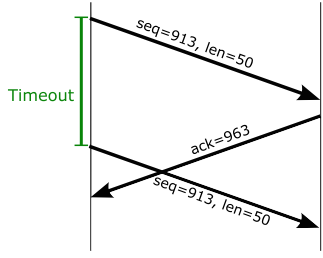
Retransmission Timeout

TECHNISCHE UNIVERSITÄT DARMSTADT

- Oben wurde gesagt, dass eine Segmentübertragung wiederholt werden soll (bzw. muss), wenn zu lange kein ACK eingetroffen ist

Wie lange ist „zu lange“?

- Verfrühte Timeouts sollten natürlich möglichst vermieden werden



- . . . zu langes Warten aber auch
- Hier muss ein guter Kompromiss gefunden werden!

Retransmission Timeout

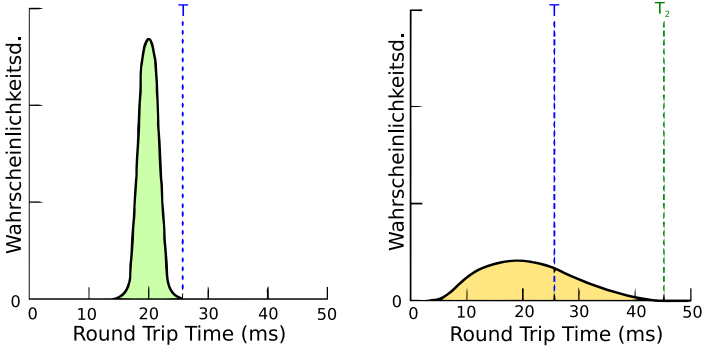
TECHNISCHE UNIVERSITÄT DARMSTADT

- Idee: Beobachte die typische RTT der Verbindung und verwende das als Grundlage für den Timeout
- Wenn ein ACK ankommt, liefert uns das automatisch auch eine Messung der RTT (wir wissen ja, wann wir das Segment abgeschickt hatten)
- Darüber können wir einen (exponentiell gleitenden) Mittelwert berechnen
- Die RTT selbst ist aber keine gute Wahl den Timeout
  - dann würde ja ein minimal verzögertes ACK schon eine Neuübertragung verursachen
- Wie wählt man also basierend auf den RTT-Messungen den „richtigen“ Timeout-Wert?
- Ursprünglich einmal in TCP: das Doppelte der gemessenen durchschnittlichen RTT
  - auch nicht gut – schauen wir einmal, warum!

Retransmission Timeout

TECHNISCHE UNIVERSITÄT DARMSTADT

- Zwei Wahrscheinlichkeitsdichten für RTT-Messungen innerhalb einer TCP-Verbindung – beide mit demselben RTT-Mittelwert



- Links genügt es, den Timeout T geringfügig höher als den Durchschnitt zu wählen, rechts wäre das zu früh

$\Rightarrow$ Der Durchschnitt der RTT sagt alleine nicht genug aus

Retransmission Timeout

TECHNISCHE UNIVERSITÄT DARMSTADT

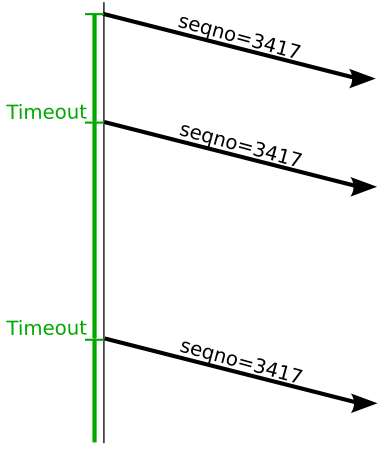
- Also: Standardabweichung (und damit „Breite“ der Verteilung) sollte mit einfließen
- TCP misst deshalb außerdem, wie weit die RTTs typischerweise vom Mittelwert abweichen (erneut gleitender Mittelwert)
- Der Timeout wird in TCP dann wie folgt gewählt:

$$\text{RTT} + 4 \cdot \text{Standardabweichung}$$

Exponentielle Timeouts

TECHNISCHE UNIVERSITÄT DARMSTADT

- Wenn nach einer Übertragungswiederholung noch immer kein ACK eintrifft, wird die Wartezeit bis zum nächsten Übertragungsversuch verdoppelt
- Die Zeit zwischen aufeinanderfolgenden Übertragungsversuchen wächst also *exponentiell* (bis zu einer oberen Grenze, oft 60 s)



Verzögerte ACKs

TECHNISCHE UNIVERSITÄT DARMSTADT

Heutige TCP-Implementationen verringern die Zahl generierter ACK-Nachrichten durch einen Mechanismus namens *Delayed ACKs (DACK)*; wenn ein TCP-Empfänger…

- …ein Segment in richtiger Reihenfolge mit der erwarteten Sequenznummer erhält und alle vorangegangenen Daten schon bestätigt wurden

⇒ das ACK wird verzögert und auf das nächste Segment gewartet; erst wenn nach (max.) 500 ms immer noch kein weiteres Segment ankommt, wird das ACK verschickt

- …ein Segment in richtiger Reihenfolge mit der erwarteten Sequenznummer ankommt und ein vorangegangenes ACK verzögert wurde

⇒ sofort ein gemeinsames (kumulatives!) ACK für beide Segmente verschicken

Verzögerte ACKs

TECHNISCHE UNIVERSITÄT DARMSTADT

- …ein Segment mit einer höheren Sequenznummer als erwartet empfängt

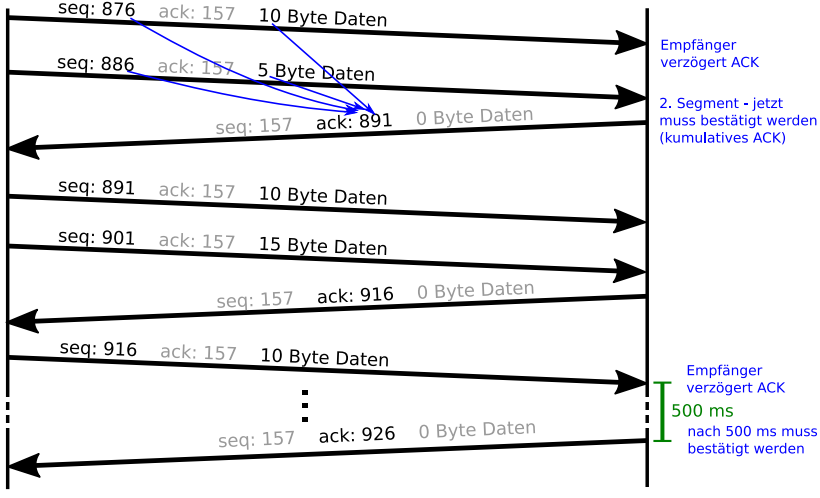
⇒ eine Lücke!

⇒ sofort das vorangegangene ACK wiederholen (mit der – noch immer – erwarteten nächsten Sequenznummer = Anfang der Lücke)

- …ein Segment erhält, das eine vorhandene Lücke (teilweise) schließt

⇒ sofort ein ACK mit dem nächsten noch fehlenden Byte verschicken

Beispiel: Datenübertragung mit DACK



Fast Retransmit

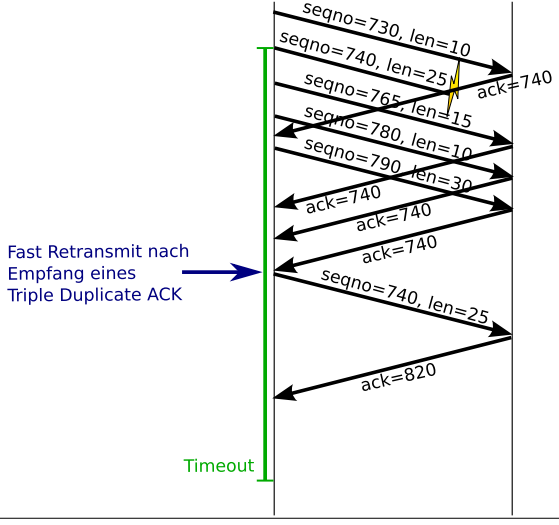
TECHNISCHE UNIVERSITÄT DARMSTADT

- Der TCP Retransmission Timeout ist oft relativ lang
- Das bedeutet lange Verzögerungen vor einer Übertragungswiederholung

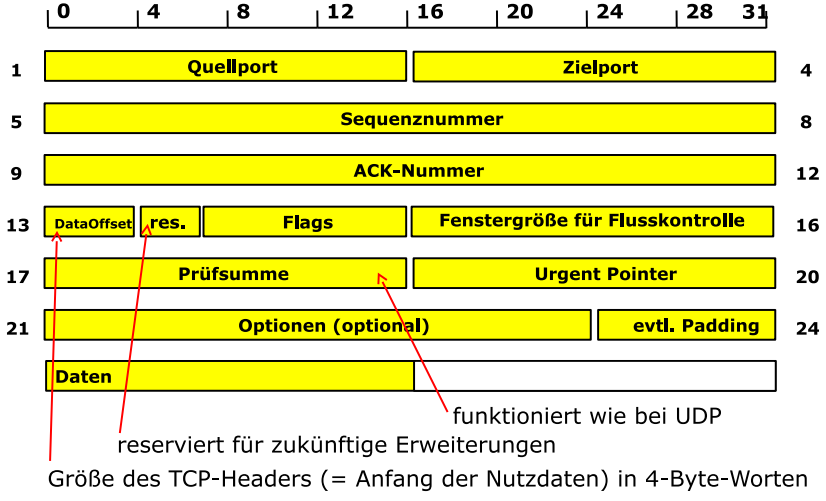
Was passiert, wenn ein einzelnes Segment verloren geht, nachfolgende aber ankommen?

- Der Sender wird mehrere ACKs mit derselben Sequenznummer bekommen
- Das lässt sich ausnutzen:
  - wenn ein Sender drei Duplikate eines ACKs erhalten hat (*Triple Duplicate ACK, TDACK*), nimmt er an, dass das Segment verloren gegangen ist
  - dann wird ein *Fast Retransmit* durchgeführt: das Segment wird schon vor dem Timeout wiederholt
- Neu übertragen wird nur das eine Segment

Fast Retransmit



Aufbau des TCP-Headers



TCP-Flags

TECHNISCHE UNIVERSITÄT DARMSTADT

- Insgesamt 9 Bits im TCP-Header sind sogenannte Flags
- Die wichtigsten davon:
  - ACK: Segment bestätigt korrekt empfangene Daten, der Inhalt des ACK-Nummern-Feldes ist gültig
  - RST: der Sender des Segments will die Verbindung abbrechen (Fehlerfall)
  - SYN: wird beim Verbindungsaufbau verwendet
  - FIN: Verbindungsabbau, es werden keine weiteren Daten in dieser Senderichtung folgen
- Außerdem: URG-Flag zum Anzeigen besonders wichtiger Daten (heute irrelevant) und drei Flags für spezielle Erweiterungen der Überlastkontrollmechanismen

TCP-Optionen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Wenn das Data-Offset-Feld im Header einen Wert > 5 hat (also mehr als $5 \cdot 4 = 20$ Byte Header-Länge), dann enthält der TCP-Header ab Byte 21 sogenannte Optionen
- Das erste Byte gibt den Typ der Option an
- Optionsfelder können unterschiedlich lang sein
  - für Typ ≥ 2 gibt das zweite Byte die Gesamtlänge in Byte an
- Ein paar Beispiele:

Optionstyp	Länge	Bedeutung
0	1	Ende der Optionsliste
1	1	Leere Option
2	4	MSS
3	3	Window Scaling
4	2	SACK-Unterstützung
5	var.	SACK-Daten
8	10	Timestamp

TCP-Optionen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Die Optionen 0, 1 und 2 müssen von jeder TCP-Implementation unterstützt werden
- 0 bedeutet: „Ende der Optionsliste“
  - der Rest ist Padding
- 1 bedeutet: „dieses Byte der Optionsliste wird nicht genutzt“
  - kann „Lücken“ zwischen Optionen füllen
- 2 (*Maximum Segment Size, MSS*) wird verwendet, um im ersten Segment einer neuen Verbindung mitzuteilen, wie groß die maximale Segmentgröße ist, die die TCP-Implementation verarbeiten kann
  - die MSS muss laut Standard mindestens 536 Byte sein
  - viele Implementationen können aber mehr
  - wenn die Option nicht verwendet wird, nimmt TCP eine MSS von 536 Byte an (schlecht!!)

Timestamp-Option

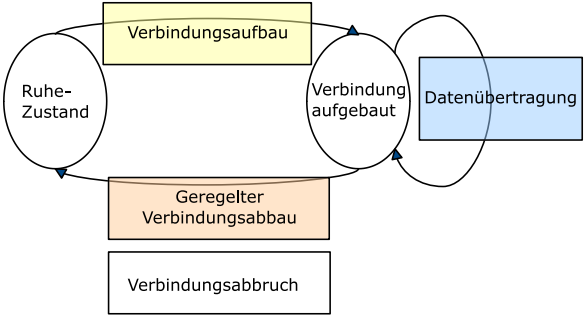
TECHNISCHE UNIVERSITÄT DARMSTADT

- Die Timestamp-Option ist eine sehr einfache und „typische“ TCP-Option
- Ziel: Mit jedem empfangenen ACK einen Zeitstempel transportieren, zu welchem Zeitpunkt das zugehörige Paket verschickt wurde
  - $\Rightarrow$ einfacher Mechanismus für sehr genaue RTT-Messungen!
- Das Verfahren wird angewandt, wenn der Client die Option im ersten Paket beim Verbindungsaufbau verwendet und der Server das in seinem ersten Antwortpaket ebenfalls tut
  - das bedeutet: beide Seiten unterstützen die Option
- Der Timestamp-Optionseintrag ist 10 Bytes lang
  - 1 Byte für den Optionstyp (Wert: 8)
  - 1 Byte für die Länge des Optionseintrags (immer 10)
  - 4 Byte für einen aktuellen Zeitstempel seiner Uhr in Senderichtung
  - 4 Byte spiegelt den empfangenen Zeitstempel in der Rückrichtung

Verbindungen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Jetzt kommen wir zu einem anderen, grundlegend wichtigen Thema: Die Verwaltung von Verbindungen
- Wieder betrachten wir das Problem erst allgemein, danach dann für den speziellen Fall TCP
- Auf einer abstrakten Ebene gibt es zwei Zustände, zwischen denen beim Verbindungsauf- und abbau gewechselt wird:



Dienstprimitive

TECHNISCHE UNIVERSITÄT DARMSTADT

- Der darüberliegenden Schicht werden über eine API (z. B. die Socket-API) *Dienstprimitive* angeboten, mit denen sie die Verbindung steuern und nutzen kann
- Dienstprimitive könnten zum Beispiel sein:

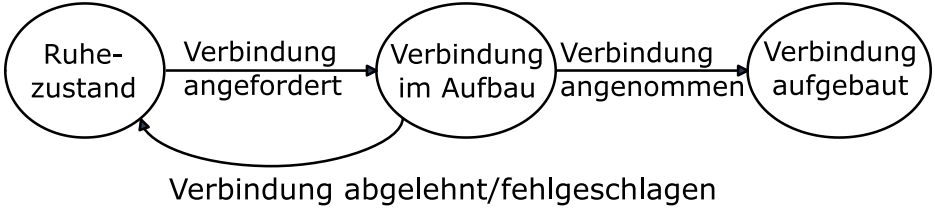
Primitiv	Bedeutung
LISTEN	blockierendes Warten auf ankommende Verbindung (= Serverseite)
CONNECT	Aufbau einer Verbindung zu einer wartenden Partnerinstanz (= Clientseite)
RECEIVE	blockierendes Warten auf ankommende Daten
SEND	Daten an Partnerinstanz senden
DISCONNECT	Verbindung beenden

- Die Protokollimplementation erzeugt dann Nachrichten entsprechend dem Protokoll
- Dienstprimitive spiegeln sich nicht unbedingt 1:1 in Protokollnachrichten wider
  - zum Beispiel erzeugt LISTEN auf der Serverseite keine ausgehende Protokollnachricht

Verbindungsaufbau

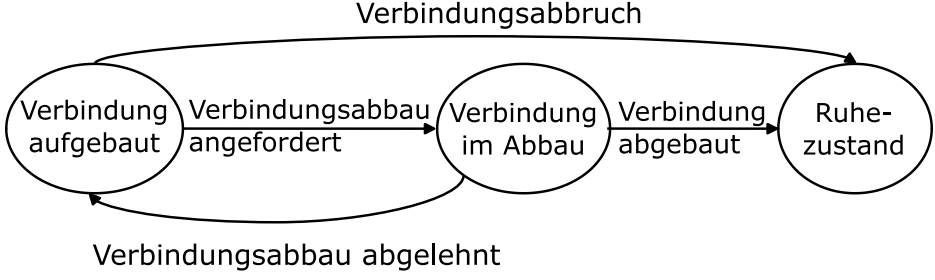
TECHNISCHE UNIVERSITÄT DARMSTADT

- Eine Verbindung „entsteht“ nicht plötzlich von selbst
- Dafür sind mindestens zwei Nachrichten (z. B. Verbindungsanfrage und Antwort auf die Verbindungsanfrage) nötig – je nach Protokoll eventuell auch noch mehr
- Deshalb gibt es nicht nur die Zustände „nicht verbunden“ und „verbunden“, sondern abhängig vom Protokoll noch weitere Zwischenzustände
- In einem ganz einfachen Fall z. B. so:



Verbindungsabbau

- Dasselbe gilt natürlich auch während des Verbindungsabbaus, z. B. so:



In der Praxis sind die Protokollzustände sowie der Wechsel zwischen ihnen (durch Dienstprimitive und/oder Nachrichten) also viel komplizierter als in dem einleitenden Beispiel!

- Schauen wir uns das also für TCP genauer an – dort ist (wie zu erwarten) tatsächlich alles noch viel komplizierter

Initiale Sequenznummern

TECHNISCHE UNIVERSITÄT DARMSTADT

- Zu Beginn einer Verbindung werden die TCP-Startsequenznummern in jede Richtung beliebig (normalerweise: zufällig) gewählt

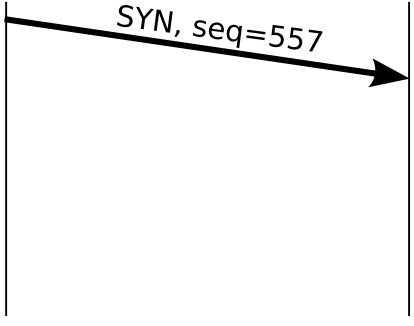
- Wir haben bereits diskutiert, warum das sinnvoll ist

TCP-Verbindungsaufbau

TECHNISCHE UNIVERSITÄT DARMSTADT

Zum Aufbauen einer Verbindung verwendet TCP einen sogenannten *Drei-Wege-Handshake*:

1. Zunächst schickt der Client (also der Endpunkt, der die Verbindung aufbaut) ein Segment mit gesetztem SYN-Flag an den Server; die von ihm gewählte Anfangssequenznummer steht im Sequenznummernfeld (*SYN-Segment*)

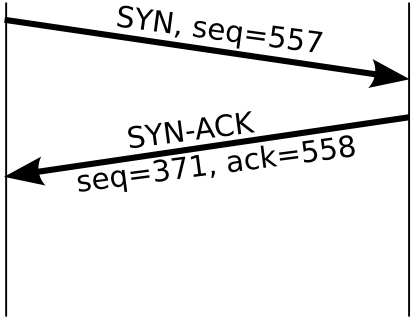


TCP-Verbindungsaufbau

TECHNISCHE UNIVERSITÄT DARMSTADT

Zum Aufbauen einer Verbindung verwendet TCP einen sogenannten *Drei-Wege-Handshake*:

2. Der Server antwortet mit einem *SYN-ACK-Segment*: er setzt das SYN- und das ACK-Flag, bestätigt im ACK-Nummern-Feld die initiale Sequenznummer des Clients +1 (!) und gibt im Sequenznummernfeld seine eigene Startsequenznummer an

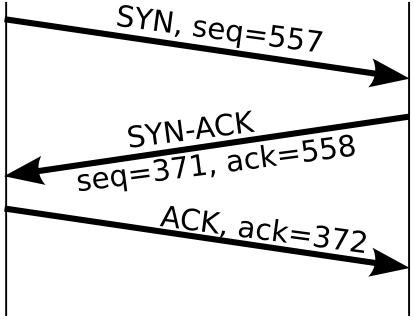


TCP-Verbindungsaufbau

TECHNISCHE UNIVERSITÄT DARMSTADT

Zum Aufbauen einer Verbindung verwendet TCP einen sogenannten *Drei-Wege-Handshake*:

3. Schließlich bestätigt der Client das SYN-ACK mit einem *ACK*, in dem er die initiale Sequenznummer des Servers – wieder +1 – bestätigt

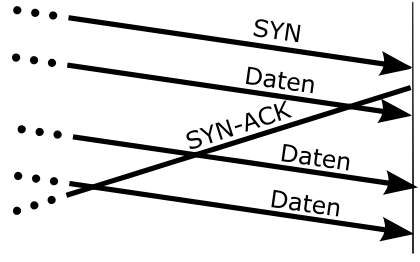


TCP-Verbindungsaufbau

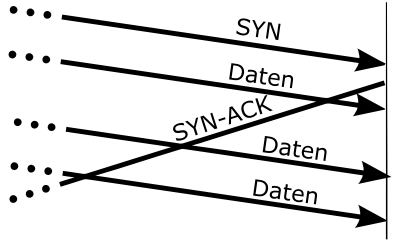
TECHNISCHE UNIVERSITÄT DARMSTADT

Wieso ist ein Drei-Wege-Handshake notwendig?
Warum genügt nicht auch ein SYN/SYNACK-Handshake?

- Es geht wiederum um „alte“ Segmente, die das Netz eventuell noch verlassen könnten
- Angenommen, ein Zwei-Wege-Handshake würde verwendet und die paar ersten Segmente einer alten Verbindung würden (evtl. ein zweites Mal) bei einem TCP-Server ankommen



TCP-Verbindungsaufbau

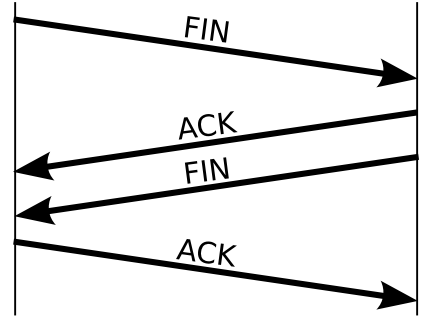


- Wenn nur ein Zwei-Wege-Handshake erfolgen müsste, dann müsste der Server die eingehenden Daten akzeptieren und an die Anwendung weitergeben
- Das dritte Handshake-Paket (genauer: die korrekt wiedergegebene Startsequenznummer des Servers) bestätigt dem Server also, dass der Client „lebt“ und *jetzt* die Verbindung aufbauen möchte
- Außerdem bietet es zusätzlichen Schutz vor einem Angreifer, der auf diese Weise keine TCP-Verbindung mit gefälschter Quelladresse vortäuschen kann, ohne die Antworten zu hören

TCP-Verbindungsabbau

TECHNISCHE UNIVERSITÄT DARMSTADT

- Für den Verbindungsabbau verwendet TCP einen Vier-Wege-Handshake
- Jede Richtung der Verbindung kann mit einem gesetzten FIN-Flag signalisieren, dass sie keine weiteren Daten senden wird



- Ein FIN der Gegenseite wird mit einem ACK bestätigt
  - wie beim SYN wird die ACK-Nummer um 1 erhöht
  - dieses ACK bestätigt also, dass alle Daten dieser Richtung korrekt empfangen wurden; es werden also keine weiteren Übertragungswiederholungen notwendig

Verbindungsabbau

TECHNISCHE UNIVERSITÄT DARMSTADT

- Die beiden Richtungen der TCP-Verbindung können unabhängig voneinander geschlossen werden
- Auch nachdem die Richtung $A \rightarrow B$ geschlossen wurde, können noch Daten von B zu A fließen
  - das nennt man „halbgeschlossene Verbindung“ (nicht verwechseln mit einer „halboffenen Verbindung“ vor Abschluss des 3-Wege-Handshakes!)
- Wenn auch die Gegenseite die Verbindung gleich schließt, ist es möglich, dass das ACK für das erste FIN und das zweite FIN zu einem Segment zusammengefasst werden
  - diese Nachrichten werden aber durch unterschiedliche Ereignisse (Empfang des FIN bzw. lokaler Aufruf des CLOSE-Dienstprimitivs) ausgelöst und deshalb zu unterschiedlichen Zeitpunkten erzeugt

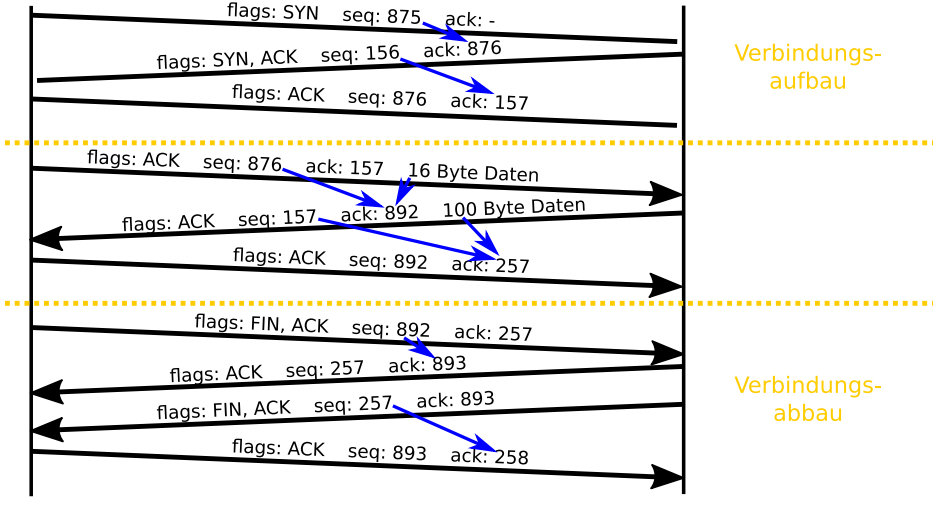
Wann kann ein Host die Informationen
zu einer geschlossenen TCP-Verbindung „vergessen“?

Verbindungsabbau

TECHNISCHE UNIVERSITÄT DARMSTADT

- Informationen über eine geschlossene Verbindung müssen so lange „gemerkt“ werden, bis der Host sicher sein kann, dass die Gegenstelle das ACK auf ihr FIN erhalten hat
  - sonst könnte ja eine Übertragungswiederholung des FIN eintreffen, auf die korrekt mit einem wiederholten ACK reagiert werden muss…
- Das lässt sich in einem Netz mit dem Dienstmodell des Internet nicht ohne weiteres garantieren – wer zuerst sein FIN geschickt hatte, hat deshalb ein Problem!
- In der Praxis merken sich die Implementationen die Informationen für eine gewisse Zeitspanne (z. B. 4 Minuten)
- Das ist ein echtes Problem für TCP-Implementationen!
  - ein Webserver mit 10 000 Verbindungen pro Sekunde muss bei 4 Minuten Wartezeit ständig Informationen über 2 Millionen alte TCP-Verbindungen vorhalten!

Eine komplette TCP-Sitzung



Reset (RST)

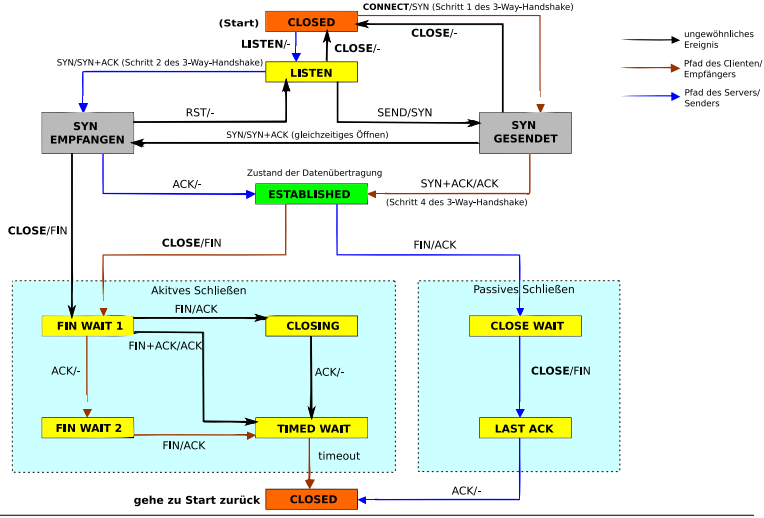
TECHNISCHE UNIVERSITÄT DARMSTADT

- Neben dem „geordneten“ Schließen mit FIN-Segmenten kann eine TCP-Verbindung auch abgebrochen werden
- Der Empfang eines Segmentes mit gesetztem RST-Flag (*Reset*) bedeutet, dass ab sofort
  - keine weiteren Segmente gesendet werden dürfen und
  - alle zukünftig eintreffenden Segmente verworfen werden sollen
- Zweck: Fehlerbehandlung, zum Beispiel nach dem Absturz von einem der Kommunikationspartner
  - RST wird deshalb vor allem dann gesendet, wenn ein Segment für eine unbekannte TCP-Verbindung eintrifft
- Nach einem Reset gibt es keine Garantien, ob gesendete Daten zugestellt wurden oder nicht!
- Ein RST darf niemals als Antwort auf ein eingehendes RST gesendet werden (um sog. „Reset Wars“ zu vermeiden)

Zustände einer TCP-Verbindung (vereinfacht)

TECHNISCHE UNIVERSITÄT DARMSTADT

- Das Dienstmodell des Internets macht den Verbindungsaufbau und -abbau in TCP offensichtlich sehr komplex
- Das resultiert auch in sehr vielen Protokollzuständen und komplizierten Übergängen zwischen diesen



(Abbildung: [Wikipedia])

Bytestrom-Segmentierung

TECHNISCHE UNIVERSITÄT DARMSTADT

- TCP unterteilt den verschickten Bytestrom selbständig in Segmente
- Betrachten wir nun das Problem der Segmentierung etwas genauer

Wie viele Daten sollten wir in ein Segment packen?

- So viele wie möglich, um den Overhead überschaubar zu halten!
- Aber wir können die Anwendung nicht „zwingen“, uns Daten zu geben
- Vielleicht hat sie zurzeit einfach wirklich keine mehr…

Interessanter Fall: Wir haben Daten vorliegen, aber nicht „genug“, um ein Paket ganz zu füllen

Nicht volle Segmente

TECHNISCHE UNIVERSITÄT DARMSTADT

- Angenommen, unser TCP-Endpunkt hat einen leeren Sendepuffer
- Wir kennen außerdem die maximale Nutzdatenmenge, die in ein Segment passt (= Maximum Segment Size, MSS)
- Wir bekommen von der Anwendung jetzt x Byte Daten, wobei $x < \text{MSS}$

Sollten wir ein „nicht-volles“ Segment verschicken?

- Natürlich, das müssen wir!
- Wer weiß, ob die Anwendung nochmals etwas senden würde, oder ob sie jetzt auf Antwort vom Kommunikationspartner wartet

Aber was, wenn die Anwendung immer gleich nach dem Absenden eines solchen „kleinen“ Segments wieder ein paar wenige Daten schreibt (*Tinygram-Problem*)?

Nagle-Algorithmus

TECHNISCHE UNIVERSITÄT DARMSTADT

- Ein TCP-Sender kann den *Nagle-Algorithmus (Nagle's Algorithm)* verwenden, um das Tinygram-Problem zu vermeiden

- Grundidee:
  - solange ein unbestätigtes Segment unterwegs ist
  - . . . und noch nicht genug Daten für ein volles Segment (also weniger als eine MSS) zum Senden vorliegen
  - . . . dann verzögere das Absenden des nächsten Segments

Ein Experiment

Empfangspuffer

TECHNISCHE UNIVERSITÄT DARMSTADT

- Ein Empfänger in einem Netzwerk kann Daten nicht beliebig schnell verarbeiten
- Im Fall von TCP wird die empfangende Anwendung in der Regel nur dann Daten entgegennehmen, wenn sie auch in der Lage ist, sie zu verarbeiten
  - wann und wie oft das passiert, ist Sache des Anwendungsentwicklers
- Wenn Sender und Netzwerk schneller sind als der Empfänger, dann kann es sein, dass dieser mit Daten „überrannt“ wird
- In einem zuverlässigen Protokoll dürfen empfangene (und bestätigte!) Daten aber nicht einfach weggeworfen werden
- Speicher zum Zwischenspeichern empfangener Daten sind allerdings auch nicht unendlich groß

Flusskontrolle

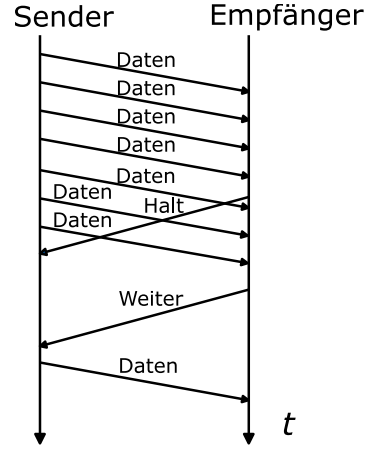
TECHNISCHE UNIVERSITÄT DARMSTADT

- Also ist ein Mechanismus nötig, der den Sender bei Bedarf „bremst“
- Einen solchen Mechanismus nennt man *Flusskontrolle*
- Prinzipielle Möglichkeiten für die Realisierung (zunächst wieder ganz unabhängig vom konkreten Protokoll TCP):
  - über Kommandos: Halt, Weiter
  - durch Verzögerung von ACKs
    - schwierig bei gleichzeitigem Einsatz von Timeouts für Übertragungswiederholungen
  - über einen Fenstermechanismus (Schiebefenster, Sliding Window)

Flusskontrolle mit Halt/Weiter-Meldungen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Einfachste Methode
  - bei drohendem Empfänger-Überlauf: Absenden einer „Halt“-Meldung
  - bei wieder möglichem Empfang: Absenden einer „Weiter“-Meldung
- Beispiel: XON/XOFF-Protokoll
  - Anschluss von Modem, Drucker (früher mal…)
  - Punkt-zu-Punkt-Verbindungen zw. Rechnern
  - ISO 7-Bit- (bzw. ASCII-) Alphabetzeichen
    - XON ist DC1 (Device Control 1)
    - XOFF ist DC3 (Device Control 3)



Problem: In Netzen wie dem Internet gibt es keine Garantie, dass/wann die „Halt“-Nachricht eintrifft

Halt/Weiter im Internet?

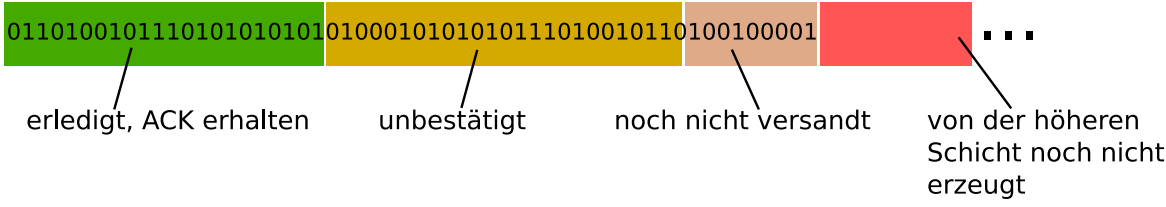
TECHNISCHE UNIVERSITÄT DARMSTADT

- Deswegen lässt sich nicht sicherstellen, dass der Sender nicht zu viele Daten auf den Weg bringt, bevor ihn das „Halt“ erreicht
- Wir brauchen (zumindest für solche Netze) also eine andere Idee…
- Ansatz: Präzisere Rückmeldung vom Empfänger an den Sender
- Um zu verstehen, was genau der Empfänger dem Sender mitteilen muss, schauen wir uns den Sequenznummernbereich aus Sicht beider Endpunkte genauer an

Sequenznummernraum beim Sender

TECHNISCHE UNIVERSITÄT DARMSTADT

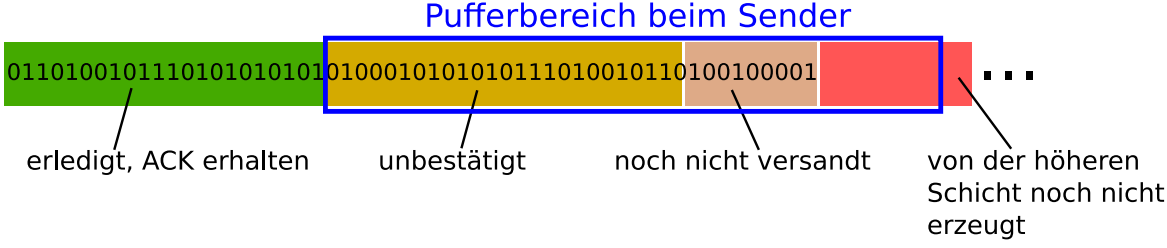
Aus der Sicht des Senders lässt sich der Sequenznummernraum in mehrere Bereiche unterteilen:



Welchen Teil des Sequenznummernraumes muss der Sender in seinem Sendepuffer zwischenspeichern?

Sequenznummernraum beim Sender

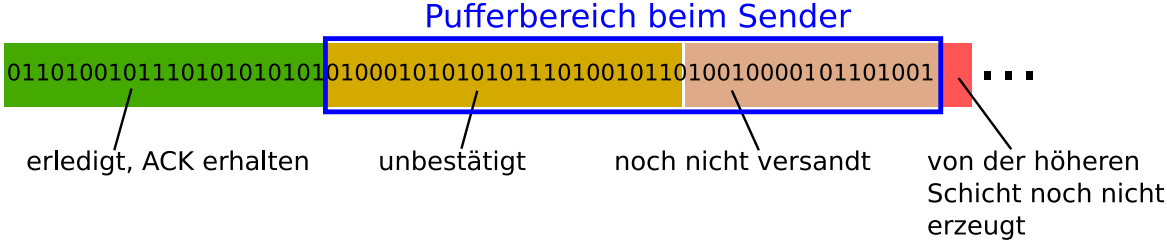
- Der Sender muss diejenigen Teile des Bytestroms puffern, die noch nicht bestätigt wurden
- Denn für diese Teile könnte eine Übertragungswiederholung notwendig werden!



Sequenznummernraum beim Sender

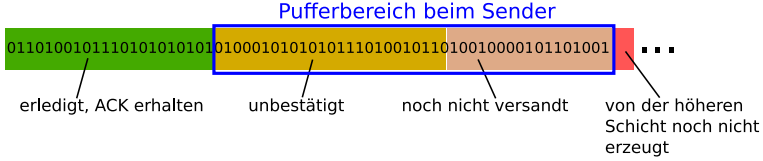
TECHNISCHE UNIVERSITÄT DARMSTADT

- Wenn die Anwendung weiter Daten erzeugt, ist der zur Verfügung stehende Platz für den Sendepuffer irgendwann voll:

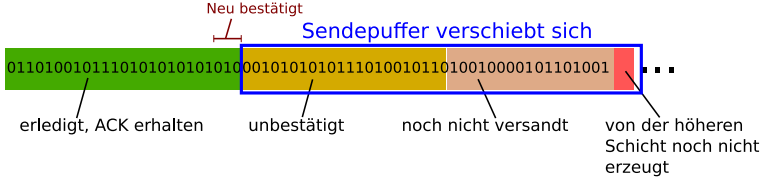


- Der Sender muss die Daten erzeugende höhere Schicht dann in irgendeiner Form „bremsen“
  - typisches Vorgehen bei TCP-Sockets: sendende Anwendung blockiert beim Versuch, auf einen Socket mit vollem Sendepuffer zu schreiben

Verschieben des Fensters

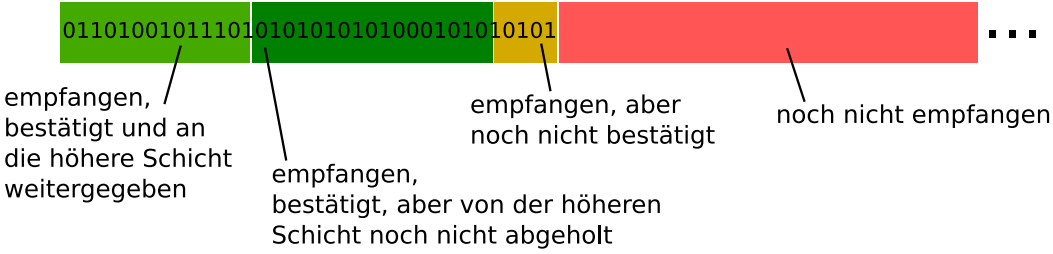


Wenn weitere Teile des Bytestroms bestätigt werden, verschiebt sich der Sendepuffer über dem Sequenznummernbereich – es wird wieder Platz frei:



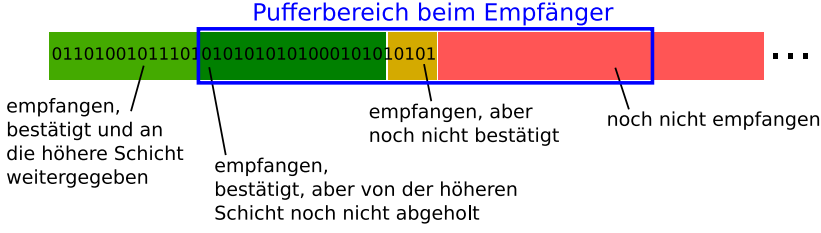
Man spricht von einem *Schiebefenster (Sliding Window)*

Sequenznummernraum beim Empfänger



Welcher Sequenznummernbereich muss empfängerseitig gepuffert werden?

Sequenznummernraum beim Empfänger



Beobachtungen:

1. Das Empfängerfenster verschiebt sich nach rechts, wenn empfangene Daten von der höheren Schicht abgeholt werden
2. Der rechte Rand des Fensters bestimmt, was die höchste Sequenznummer ist, die derzeit noch entgegengenommen werden kann

Herausforderung: Wie kann der *Sender* wissen,
wie viel beim *Empfänger* maximal noch zwischengespeichert werden kann?

Kreditbasierte Flusskontrolle

TECHNISCHE UNIVERSITÄT DARMSTADT

- Der Empfänger muss es schaffen, dass der Sender keine Daten verschickt, die nicht mehr in seinen Puffer passen würden
- Der Sender weiß nicht, bis zu welcher Sequenznummer die empfangende Anwendung die Daten schon abgeholt hat
- Der Sender weiß auch nicht, wie viel Pufferplatz dem Empfänger gerade zur Verfügung steht

$\Rightarrow$ Empfänger muss entsprechende Rückmeldung liefern
  - wird typischerweise gemeinsam mit ACKs übertragen

Kreditbasierte Flusskontrolle

TECHNISCHE UNIVERSITÄT DARMSTADT

Ein paar prinzipielle Möglichkeiten:

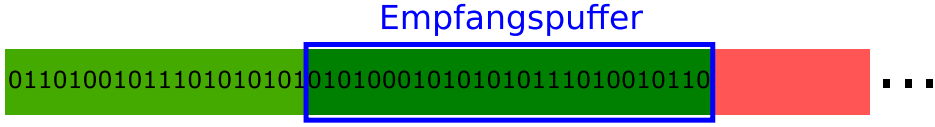
- Empfänger teilt dem Sender Anfang und Länge des derzeitigen Empfangsfensters mit
- Empfänger meldet die höchste Sequenznummer zurück, die noch in den Puffer passen würde
  - so muss man weniger Daten zum Sender schicken: nur eine Sequenznummer statt Sequenznummer + Fensterlänge
- Empfänger meldet zusammen mit jedem ACK die verbleibende Datenmenge, die *hinter den gerade bestätigten Daten noch in den Puffer passt („Sendekredit")*
  - diese *kreditbasierte Flusskontrolle* trifft man in der Praxis am häufigsten an – auch bei TCP
  - das *Receive-Window-Feld (RWIN)* im TCP-Header trägt genau diese Information (gemessen wiederum in Byte)

Flusskontrolle in Aktion

Volle Empfangspuffer

TECHNISCHE UNIVERSITÄT DARMSTADT

Der Puffer auf Empfängerseite kann volllaufen:



Welchen Sendekredit wird der Empfänger dem Sender mit seinem letzten ACK zurückmelden? Welches Problem ergibt sich?

Volle Empfangspuffer

TECHNISCHE UNIVERSITÄT DARMSTADT

- Es wird eine Fenstergröße von 0 zurückgemeldet
- Der Sender dürfte damit nie wieder Daten senden!
- …er würde deshalb auch nie wieder ein ACK bekommen
- …und so nie erfahren, dass wieder Platz im Empfangspuffer frei geworden ist

Ihr Lösungsvorschlag?

Volle Empfangspuffer

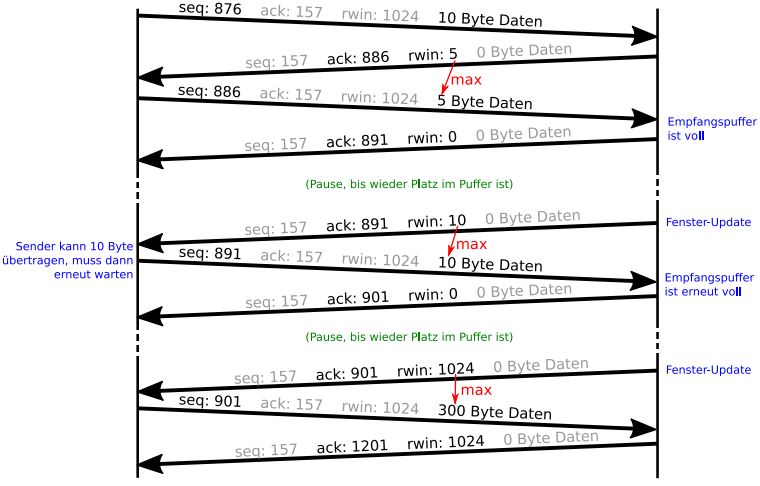
TECHNISCHE UNIVERSITÄT DARMSTADT

Lösung in TCP:

- der Sender darf auch bei Empfangsfenster 0 ab und zu ein Segment („*Probe Packet*“, 1 Byte groß) senden
- immer, wenn der *Persist-Timer* (= leichte Abwandlung des Retransmission Timers, der bei rwin $= 0$ aktiv wird) abläuft, wird ein neuer Versuch gestartet
- der Empfänger sendet dann jeweils ein ACK, darin steht die aktuelle rwin-Wert
- der Empfänger darf (muss nicht!) auch von sich aus erneut ein *Fenster-Update* (= wiederholtes ACK mit höherer Fenstergröße) schicken, sobald wieder Platz ist
  - sicherlich sinnvoll (und wird in der Praxis gemacht)
  - das alleine würde aber nicht ausreichen, denn das Fenster-Update könnte ja verloren gehen
  - der Probe-Mechanismus ist deshalb in jedem Fall notwendig!

Flusskontrolle mit volllaufendem Puffer

TECHNISCHE UNIVERSITÄT DARMSTADT



Silly Window

TECHNISCHE UNIVERSITÄT DARMSTADT

Was passiert, wenn der Empfänger den Puffer erst volllaufen lässt, und die höhere Schicht die Daten dann immer wieder in sehr kleinen Einheiten vom Socket liest?

- Das Fenster geht immer wieder um wenige Bytes auf
- Der Sender wird so gezwungen, viele sehr kleine Segmente zu verschicken
- Jeweils mit vollständigem TCP+IP+. . . -Header – das ist sehr ineffizient!
- Dieses Problem ist bekannt als *Silly Window Syndrome*
- Im Prinzip ist das ein Problem der Anwendung: schlecht implementiert!
- Aber schlecht implementierte Anwendungen gibt es häufig!

Silly Window

TECHNISCHE UNIVERSITÄT DARMSTADT

Lösung:

- Empfänger wartet, bis sinnvoll viel Puffer (z. B. halbe Puffergröße oder eine MSS) verfügbar ist
- Erst dann wird der Sender über das wieder geöffnete Fenster informiert
- Auch der Nagle-Algorithmus auf der anderen Seite der Verbindung, falls aktiviert, kann das Problem lösen

Maximale Fenstergröße

TECHNISCHE UNIVERSITÄT DARMSTADT

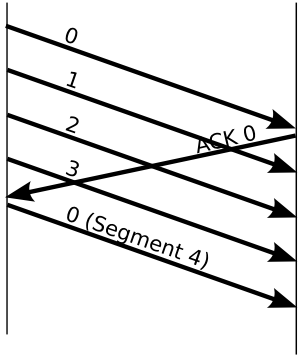
- Im Detail gibt es (wieder mal) eine Reihe von Stolperfallen betreffend Sequenznummern und Fenstergrößen
- Eine wichtige Einschränkung ergibt sich aus der Tatsache, dass Sequenznummern endlich lang sind
  - Überlauf: nach der höchsten Sequenznummern kommt wieder die 0
- Angenommen, wir haben Sequenznummern $0, \ldots, S-1$
- Betrachten wir den Fall *ohne* Beschränkung des Pipelining durch ein Fenster
- Sequenznummernbereich wird ausgeschöpft, der Sender überträgt immer weiter Daten

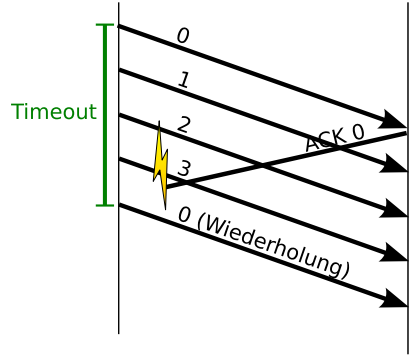
Welches Problem entsteht?

Maximale Fenstergröße

TECHNISCHE UNIVERSITÄT DARMSTADT

Zum Beispiel mit $S = 4$:





Beide Fälle sind für den Empfänger nicht unterscheidbar!

Maximale Fenstergröße

TECHNISCHE UNIVERSITÄT DARMSTADT

- Dieses Problem löst man über eine Beschränkung der Fenstergröße
- Idee: Erzwinge Eindeutigkeit, welche Sequenznummer der Sender trotz Modulo-Operation meinen kann
- Dafür: Empfangsfenster darf sich nicht für senderseitig noch unbestätigte Sequenznummern öffnen

In welchem Szenario mit möglichst kleinem Fenster ragt das Empfangsfenster wieder in senderseitig noch offene Sequenznummern?

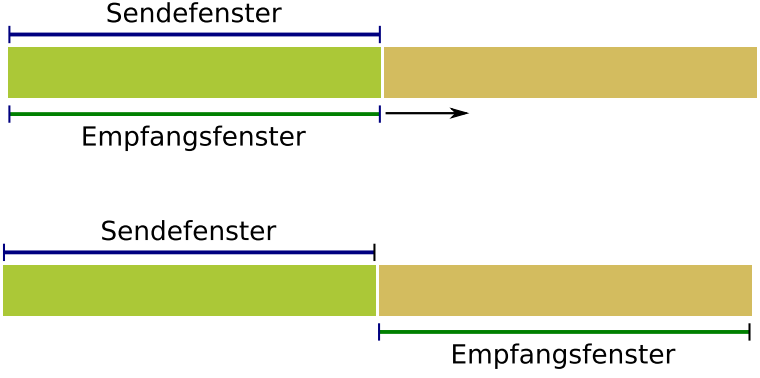
- Der Sender überträgt ein volles Fenster
- Alles kommt an, aber alle ACKs gehen verloren
- Der Empfänger verschiebt sein Fenster um eine volle Fenstergröße, der Sender bekommt davon nichts mit

Was bedeutet das für die maximal zulässige Fenstergröße?

Maximale Fenstergröße

TECHNISCHE UNIVERSITÄT DARMSTADT

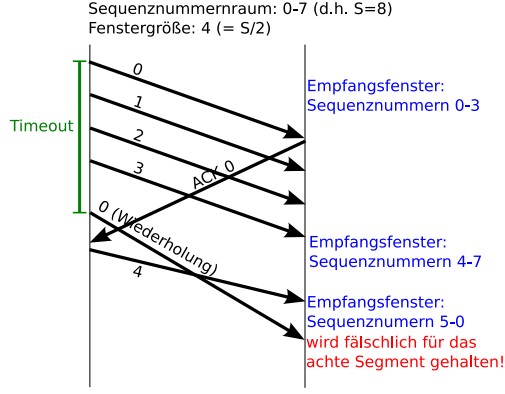
- Das Fenster darf maximal den *halben* Sequenznummernbereich überdecken
- Wenn es dann um eine volle Fenstergröße verschoben wird, werden dennoch alle vorher gültigen Sequenznummern ungültig



Weitere Probleme, weitere Einschränkungen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Effekte wie beispielsweise Reihenfolgevertauschungen können weitere Probleme verursachen
- Im nebenstehenden Beispiel wird eine Übertragungswiederholung für ein neues Segment gehalten – trotz Fenstergröße $S/2$
- Dafür sind nur sehr geringe Verzögerungen nötig – das ist inakzeptabel!



Noch weiterer „Sicherheitsabstand“ ist nötig!
Die Fenstergröße sollte $S/4$ nicht überschreiten.

Maximale Fenstergröße in TCP

TECHNISCHE UNIVERSITÄT DARMSTADT

- Die Fenstergröße in TCP ist ein 16-Bit-Feld im TCP-Header
- Der maximale Wert ist also (ganz unabhängig vom Sequenznummernfeld) $2^{16} - 1 = 65535$

Welche Auswirkungen hat das auf das Pipelining?

- Es können niemals mehr als 64 KB Daten gleichzeitig unterwegs sein
- Beispiel: Über eine Verbindung mit 100 ms RTT können nicht mehr als 640 KB/s übertragen werden (Effekt vergleichbar mit Stop and Wait)
- Für Verbindungen mit hoher RTT schränkt das die erreichbare Datenrate massiv ein!

BDP und Fenstergrößen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Allgemeiner können wir festhalten, dass die RTT und die erlaubte Fenstergröße W die erreichbare Datenrate R einschränken:

$$R = \frac{W}{\text{RTT}}$$

- Umgekehrt ergibt sich die notwendige Fenstergröße direkt aus dem Produkt von Datenrate und RTT der Netzwerkstrecke – das *Bandwidth-Delay Product (BDP)*:

$$W = R \cdot \text{RTT} = \text{BDP}$$

- Das ist ein grundlegender und wichtiger Zusammenhang für alle Protokolle mit Schiebefenster!

- Randnotiz: Bei konstant angenommener RTT kann ein künstlich klein gehaltenes Fenster dazu dienen, die Datenrate zu steuern – das wird später nochmal wichtig

Window Scaling

TECHNISCHE UNIVERSITÄT DARMSTADT

- Abhilfe für das konkrete Problem des zu kleinen RWIN-Feldes in TCP: Eine Header-Option namens „Window Scaling“
  - Optionstyp 3, definiert in RFC 1323
  - wird beim Verbindungsaufbau benutzt
  - muss im SYN- und SYNACK-Segment vorhanden sein, um aktiv zu werden
- Wenn beide Seiten Window Scaling unterstützen, wird dabei (für jede Richtung der Verbindung separat) ein Wert $0 \leq i \leq 14$ ausgehandelt
- Zukünftige Einträge im RWIN-Feld des TCP-Headers in die jeweilige Richtung werden dann immer mit 2^i multipliziert (= Shift Left um i Bit)
- Man verliert dadurch also Granularität, vergrößert aber die maximal mögliche Fenstergröße sehr stark

Window Scaling

TECHNISCHE UNIVERSITÄT DARMSTADT

Warum ist der maximal mögliche Skalierungsfaktor 2^{14}?

- Wir hatten schon gesehen, dass bei einem Sliding-Window-Verfahren die Fenstergröße maximal ein Viertel des Sequenznummernbereiches umfassen sollte
- Größe des Sequenznummernbereiches bei TCP: $S = 2^{32}$
- Maximale Fenstergröße mit Window Scaling:

$$\underbrace{2^{14}}_{\text{max. Skalierungsfaktor}} \cdot \underbrace{2^{16}}_{\text{max. Wert von rwin}} = 2^{30} = \frac{S}{4}$$

- Keine perfekte Lösung: Bei sehr hohem BDP kann das immer noch ein Engpass sein!

Noch ein Experiment

Überlastkontrolle

TECHNISCHE UNIVERSITÄT DARMSTADT

- Es kann nicht nur der Empfänger der Daten zu langsam sein, sondern es ist auch möglich, dass Pakete schneller verschickt werden als sie im Netzwerk transportiert werden können!
- Dieses Problem sieht dem Flusskontrollproblem auf den ersten Blick ähnlich – die Ursachen (und auch die Lösung) sind aber völlig anders!

Flusskontrolle = Regulieren der Senderate
zum Vermeiden von *Überlastung des Empfängers*

Überlastkontrolle = Regulieren der Senderate
zur Vermeidung von *Überlast im Netzwerk*

- Wir betrachten zunächst Überlasteffekte in Netzwerken aus einer allgemeinen Perspektive, danach dann die Überlastkontrollmechanismen in TCP

Überlast bei unbegrenzten Puffern

TECHNISCHE UNIVERSITÄT DARMSTADT

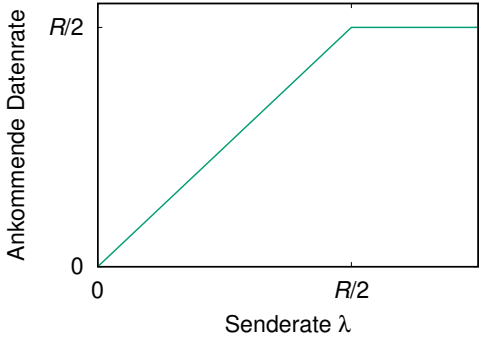
- Angenommen, durch ein (sehr einfaches) Netzwerk mit einem Router laufen zwei Verbindungen
- Beide Verbindungen senden zu zufälligen Zeitpunkten Pakete an denselben Zielrechner, jede Verbindung mit einer durchschnittlichen Datenrate von λ (in Bit/s)
- Wir betrachten (für den Moment) keine Zuverlässigkeitsmechanismen, Paketwiederholungen etc.
- Jede Leitung kann Daten mit der Rate *R* transportieren
- Die Warteschlange im Router kann beliebig lang werden (unendlich großer Puffer)

Wie verhält sich die am Zielrechner ankommende Datenrate,
wenn λ nach und nach erhöht wird?

Überlast bei unbegrenzten Puffern

TECHNISCHE UNIVERSITÄT DARMSTADT

Die Datenrate ist auf die Geschwindigkeit begrenzt, die die Leitung zum Zielrechner transportieren kann: $R/2$ Bit/s pro Verbindung

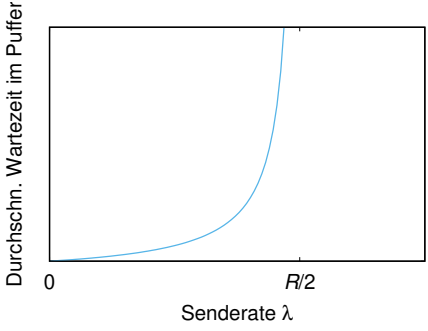


Und wie verändert sich die (durchschnittliche) Paketlaufzeit?

Überlast bei unbegrenzten Puffern

TECHNISCHE UNIVERSITÄT DARMSTADT

- Wir betrachten den eingeschwungenen Zustand nach einer langen Zeit mit konstanter Last
- Die durchschnittliche Zeit der Pakete in der Warteschlange *wird beliebig groß*, wenn λ sich an $R/2$ annähert



Überlast bei begrenzten Puffern

TECHNISCHE UNIVERSITÄT DARMSTADT

Was ändert sich, wenn die Größe der Warteschlange im Router beschränkt ist?

- Dann treten Paketverluste auf
- Das untersuchen wir ein bisschen genauer, denn es entstehen (zunächst vielleicht unerwartete) Effekte
- Zunächst ein paar grundsätzliche Überlegungen zu Paketverlustwahrscheinlichkeiten, die uns später helfen werden

Paketverlustraten bei geteilten Puffern

TECHNISCHE UNIVERSITÄT DARMSTADT

- Angenommen…
  - eine Leitung kann Datenrate R transportieren
  - darüber laufen mehrere Verbindungen $c_1, \ldots, c_n$, die Daten mit Raten $\lambda_1, \ldots, \lambda_n$ anliefern
  - wir betrachten den eingeschwungenen Zustand und Paketverlustraten im längerfristigen Mittel
  - alle Pakete sind gleich groß, Paketankunftszeiten sind statistisch unabhängig
  - *Drop-Tail-Puffer*: Paket wird verworfen, wenn bei seiner Ankunft der Puffer voll ist
  - eine Überlastsituation liegt vor, also

$$\sum_i \lambda_i > R$$

Welcher Anteil der an den Puffer am Anfang der Leitung angelieferten Pakete wird verworfen?

Paketverlustraten bei geteilten Puffern

TECHNISCHE UNIVERSITÄT DARMSTADT

- Bei hinreichender Puffergröße wird die Ausgangsleitung kontinuierlich genutzt, weil schneller angeliefert als abgeschickt wird
- Sei
$$L := \sum_i \lambda_i$$
- Von der angelieferten Gesamtdatenrate L wird R weitergeleitet; $L - R$ Bit/s müssen verworfen werden
- Damit kann also ein Anteil
$$p = \frac{L - R}{L} = 1 - \frac{R}{L}$$
der angelieferten Pakete nicht transportiert werden; dies entspricht der Verlustrate in diesem Puffer

Paketverlustraten bei geteilten Puffern

TECHNISCHE UNIVERSITÄT DARMSTADT

- Angenommen, die Verbindung c_1 verliert im längerfristigen Mittel einen Anteil p der von ihr angelieferten Pakete

Welchen Anteil ihrer jeweiligen Pakete verlieren die *anderen* Verbindungen?

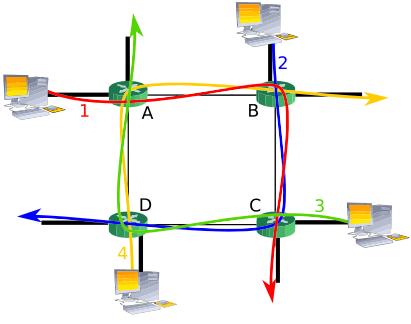
- Ein Paket geht verloren, wenn zum Zeitpunkt seiner Ankunft der Puffer voll ist (denn: Drop Tail!)
- Die Wahrscheinlichkeit, dass das passiert, ist unabhängig von der Verbindung, zu der das ankommende Paket gehört
- Für *alle anderen* Verbindungen ist die Wahrscheinlichkeit also ebenfalls gleich p

Die Verlustwahrscheinlichkeit einer Drop-Tail-Warteschlange ist bei zufälligen, unabhängigen Ankunftszeiten für alle darüberlaufenden Verbindungen gleich

Komplexeres Szenario

TECHNISCHE UNIVERSITÄT DARMSTADT

Betrachten wir folgendes Szenario mit vier Verbindungen:



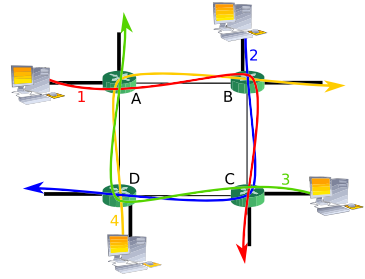
- die Leitungen zwischen den Routern können Rate R transportieren
- die anderen Leitungen sehr viel mehr (näherungsweise: beliebig viel)

Was passiert, wenn die Senderate λ der Verbindungen erhöht wird?

Komplexeres Szenario

TECHNISCHE UNIVERSITÄT DARMSTADT

- Betrachten wir als Beispiel Verbindung 1
- Wenn $\lambda \ll R/2$, werden praktisch alle Pakete zugestellt
- Pakete der Verbindungen 1 und 4 müssen sich den Ausganspuffer für die Leitung von *A* zu *B* teilen
- Wenn $\lambda \gg R/2$, kommen bei *A* viel mehr Pakete von Verbindung 1 als von Verbindung 4 an



- Freie Plätze im Puffer werden deshalb praktisch immer mit Paketen von Verbindung 1 belegt (Erinnerung: die Verlustraten sind gleich!)
- Die Pakete von 1 haben allerdings bei ihrer Ankunft bei *B* kaum eine Chance, dort einen freien Platz im Puffer zu erobern: Bei *B* kommen sehr viel mehr Pakete von Verbindung 2 an

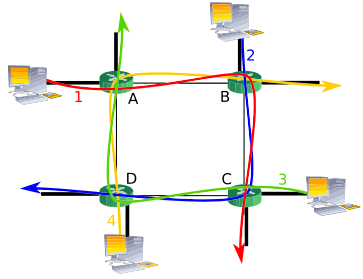
Durch die Überlast kommen also praktisch keine Daten mehr bei den Empfängern an.
Die „Arbeit“ der Router wird verschwendet!

Komplexeres Szenario

TECHNISCHE UNIVERSITÄT DARMSTADT

Nochmal etwas formaler:

- Anlieferungsraten $\lambda_1 = \ldots = \lambda_4 = \lambda$
- Datenraten auf den Links zwischen den Routern: $R_{AB} = \ldots = R_{DA} = R \ll \lambda$
- Sei α die ausgehende Rate der beim betrachteten Router erstmals weitergeleiteten Verbindung (z. B. Verbindung 1 bei Router *A*)



- β ist die Rate der Verbindung, die von dem Router zum zweiten Mal weitergeleitet wird (z. B. Verbindung 4 bei Router *A*)
  - Beobachtung: β ist zugleich der von den Verbindungen erzielte Durchsatz
- Somit ist $\alpha + \beta = R$

Komplexeres Szenario

TECHNISCHE UNIVERSITÄT DARMSTADT

- Sei p die (für alle Verbindungen gleiche!) Verlustrate vor einer Leitung zum nächsten Router, $q := 1 - p$
- Wir wissen damit, dass $\alpha = q\lambda$ und $\beta = q\alpha = q^2\lambda$ gelten muss
- Wegen $\alpha + \beta = R$ gilt also

$$q\lambda + q^2\lambda = R$$

und damit (pq-Formel)

$$q = -\frac{1}{2} + \sqrt{\frac{1}{4} + \frac{R}{\lambda}}$$

Komplexeres Szenario

TECHNISCHE UNIVERSITÄT DARMSTADT

- Für die Nutzdatenrate gilt deshalb

$$\begin{aligned}
\beta = q^2\lambda &= \left(-\frac{1}{2} + \sqrt{\frac{1}{4} + \frac{R}{\lambda}}\right)^2 \cdot \lambda \\
&= \left(\frac{\lambda}{2} + R - \sqrt{\frac{\lambda^2}{4} + R\lambda}\right) \cdot \frac{\frac{\lambda}{2} + R + \sqrt{\frac{\lambda^2}{4} + R\lambda}}{\frac{\lambda}{2} + R + \sqrt{\frac{\lambda^2}{4} + R\lambda}} \\
&= \frac{\left(\frac{\lambda}{2} + R\right)^2 - \left(\frac{\lambda^2}{4} + R\lambda\right)}{\frac{\lambda}{2} + R + \sqrt{\frac{\lambda^2}{4} + R\lambda}} = \frac{R^2}{\frac{\lambda}{2} + R + \sqrt{\frac{\lambda^2}{4} + R\lambda}}
\end{aligned}$$

  und damit

$$\lim_{\lambda \to \infty} \beta = 0$$

- Die Nutzdatenrate geht (schnell) gegen 0, wenn λ steigt!

Congestion Collapse

TECHNISCHE UNIVERSITÄT DARMSTADT

- Eine Situation, in der ein Netz aufgrund von Überlasteffekten praktisch vollständig zusammenbricht und kaum mehr sinnvoll Daten zustellt, nennt man *Congestion Collapse*

- Im Internet trat dies Mitte der 1980er Jahre massiv auf – zum Teil brach die erreichbare Datenrate um den Faktor 1000 ein!

- Deshalb ist Überlastkontrolle für ein Netzwerk quasi „lebensnotwendig“

Ansätze zur Überlastkontrolle

TECHNISCHE UNIVERSITÄT DARMSTADT

Es gibt zwei prinzipielle Möglichkeiten, in einem Netzwerk Überlastkontrolle zu implementieren:

1. mit Unterstützung durch das Netzwerkinnere
   - die weiterleitenden Netzwerkknoten signalisieren auftretende Überlast
   - dies kann in sehr unterschiedlicher Weise geschehen: von einem einzelnen durch den Router gesetzten Bit im Paket als Überlasthinweis bis hin zur expliziten Vergabe von Senderaten

2. Ende-zu-Ende
   - Überlast im Netz muss von den Endsystemen durch Beobachtung des Netzwerkverhaltens erkannt werden
   - TCP verfolgt diesen Ansatz

TCP-Überlastkontrolle

TECHNISCHE UNIVERSITÄT DARMSTADT

- Im Internet ist Überlastkontrolle auf der Transportschicht, insbesondere in TCP, implementiert
- Eigentlich ist es ein wenig seltsam, das Problem an dieser Stelle zu lösen:
  - Überlast ist ein Problem im Inneren des Netzwerkes
  - die Transportschicht arbeitet im Internet nur auf Endsystemen!
- Die Systeme im Netzwerkinneren kennen jedoch keine Verbindungen
- Außerdem entspricht es dem Ende-zu-Ende-Prinzip, komplexe Funktionalität (Überlastkontrolle ist *sehr* komplex!) möglichst in den Endsystemen zu implementieren
- Deshalb hat man im Internet entschieden, die Überlastkontrolle in TCP zu implementieren

TCP-Überlastkontrolle

TECHNISCHE UNIVERSITÄT DARMSTADT

Woran kann TCP Überlast im Netzwerk erkennen?

- TCP erkennt Überlast anhand von Segmentverlusten
- Annahme: Segmentverlust = Überlast
- Deshalb: Bei Segmentverlusten wird die Rate verringert, mit der TCP Daten absendet

TCP-Überlastkontrolle

TECHNISCHE UNIVERSITÄT DARMSTADT

Wie kann TCP erkennen, dass es seine Senderate (wieder) erhöhen darf?

- Das lässt sich überhaupt nicht direkt feststellen!
- Deshalb: Ausprobieren!
- Wenn *keine* Verluste auftreten, wird die Datenrate erhöht – so lange, bis wieder ein Segment verloren geht
- Daraus ergibt sich ein ständiges Auf und Ab der Senderate einer TCP-Verbindung („Sägezahnmuster“)
- Die Fragen, mit denen wir uns jetzt beschäftigen, sind:
  - Auf welche Weise steuert TCP die Senderate?
  - Wie wird in welchen Situationen die Senderate genau angepasst?

Anpassen von Senderaten

TECHNISCHE UNIVERSITÄT DARMSTADT

Auf welche Weise kann TCP überhaupt eine Senderate vorgeben?

- Gezielt eine bestimmte Datenrate zu erzeugen ist zwar prinzipiell möglich. . .
- . . . aber aufwändig: es wären z. B. viele hochauflösende Timer nötig
- Aber wir hatten im Zusammenhang mit der Flusskontrolle schon gesehen, dass die Größe des eines vom Sender beachteten Schiebefenster *implizit* die Datenrate begrenzt:

$$\text{Datenrate} = \frac{\text{Fenstergröße}}{\text{RTT}}$$

TCP-Überlastkontrolle

TECHNISCHE UNIVERSITÄT DARMSTADT

- Idee deshalb: ein zweites, unabhängiges Fenster zur Ratenbegrenzung
- Dieses *Überlastfenster (congestion window, congwin bzw. cwin)* wird *zusätzlich* zum Flusskontrollfenster beachtet
  - gesendet werden darf nur das, was in *beide* Fenster passt (also eine Datenmenge entsprechend dem Fenster, das gerade kleiner ist)
  - Vorsicht: Das erhöht die Gefahr, dass Sie Flusskontrolle und Überlastkontrolle verwechseln, noch mehr!
- congwin bestimmt deshalb die maximal mögliche Datenmenge pro RTT – und damit die Datenrate!
- Die Größe des Überlastfensters wird kontinuierlich angepasst, je nach wahrgenommener Überlastsituation

AIMD

TECHNISCHE UNIVERSITÄT DARMSTADT

- TCP verfolgt beim Anpassen der Senderate eine Strategie namens *Additive Increase, Multiplicative Decrease (AIMD)*

- Bis zum Auftreten eines Verlustes wächst congwin mit einer festen Rate (Additive Increase)

- Konkret: in jeder RTT wird die erlaubte Datenmenge um so viel erhöht, wie in ein volles Segment passt (eine *Maximum Segment Size, MSS*)

Slow Start

TECHNISCHE UNIVERSITÄT DARMSTADT

- Zu Beginn einer Verbindung wird congwin = 1 MSS (in manchen Implementationen auch 2 MSS) gesetzt
- Die mögliche Datenübertragungsrate im Netz kann aber sehr viel höher sein als 1 MSS/RTT
- Mit Additive Increase an den korrekten Wert heranzutasten könnte extrem lange dauern
- Zu Beginn erhöht TCP deshalb die Datenrate exponentiell
  - meist bis zum ersten Verlustereignis
  - oder bis zum Erreichen einer Schwelle (je nach Implementation, s. u.)
- Dieser Mechanismus heißt (etwas missverständlich…) *Slow Start*

Slow Start

TECHNISCHE UNIVERSITÄT DARMSTADT

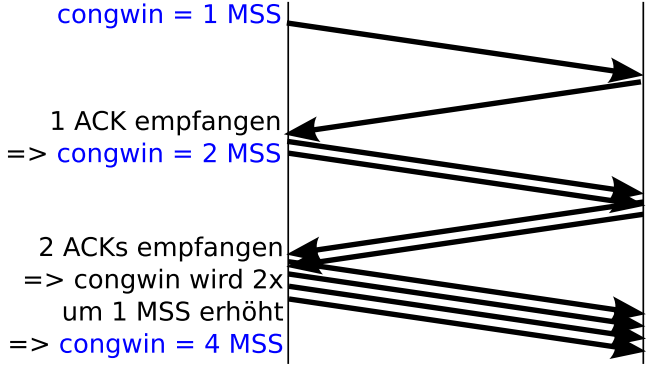
- TCP passt congwin immer beim Eintreffen eines ACK an

- Während der Slow-Start-Phase möchte TCP die Senderate in jeder RTT verdoppeln

Wie muss beim Empfang eines ACK das Überlastfenster verändert werden, damit sich insgesamt eine Verdopplung in jeder RTT ergibt?

Slow Start

- Mit jedem ACK muss congwin um 1 MSS vergrößert werden:



- Die Wirkung: TCP sendet zu Beginn sehr langsam, erhöht die Senderate dann aber sehr schnell

Verhalten bei Verlusten

TECHNISCHE UNIVERSITÄT DARMSTADT

- Bei beobachteten Paketverlusten wird die Fenstergröße verkleinert
- Das Verhalten von TCP unterscheidet sich, je nachdem, wie der Segmentverlust erkannt wurde
- Ein *Triple Duplicate ACK (TDACK)* bedeutet, dass das Netz noch in der Lage ist, zumindest einen Teil der Segmente zuzustellen (siehe Fast Retransmit!)
- TCP nimmt deshalb in diesem Fall moderate Überlast an:
  - congwin wird halbiert (Multiplicative Decrease)
  - danach wächst es wieder linear an (diese Betriebsart von TCP nennt man *Congestion Avoidance*)

Verhalten bei Verlusten – ssthresh

TECHNISCHE UNIVERSITÄT DARMSTADT

- Ein *Timeout* deutet hingegen auf schwere Überlast hin, entsprechend reagiert TCP stärker
- congwin wird deshalb in diesem Fall auf 1 oder 2 MSS gesetzt, danach erfolgt ein neuer Slow Start
- Er wird bei Erreichen von 50% der vorherigen Fenstergröße beendet
  - diese Schwelle nennt man *Slow Start Threshold (ssthresh)*
  - ssthresh wird in einer Variablen gespeichert
  - sie wird immer beim Eintreten eines Verlustereignisses auf die Hälfte der vorher erreichten Fenstergröße gesetzt
  - manche Implementationen von TCP geben zu Beginn einer Verbindung einen ssthresh vor, sodass der erste Slow Start immer an einem bestimmten Punkt abbricht; bei anderen (z. B. Linux) endet der erste Slow Start stets mit dem ersten Verlustereignis
- Nach Überschreiten des ssthresh wächst congwin linear weiter
  (= Übergang zu Congestion Avoidance)

Congestion Avoidance

TECHNISCHE UNIVERSITÄT DARMSTADT

- Während der Congestion-Avoidance-Phase soll die Fenstergröße während jeder RTT um 1 MSS wachsen

- Wieder erfolgt die Anpassung von congwin immer dann, wenn ein ACK eintrifft

Wie muss während dieser Phase
die Fenstergröße beim Eintreffen eines ACK verändert werden?

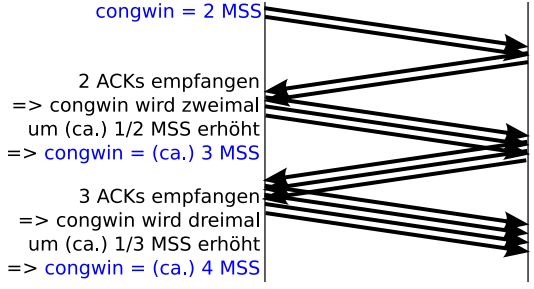
Congestion Avoidance

TECHNISCHE UNIVERSITÄT DARMSTADT

- Wenn momentan congwin $=$ *x* MSS ist und kein Verlust auftritt, dann kommen während einer RTT *x* ACKs an
- Wenn nach der RTT das Fenster um 1 MSS gewachsen sein soll, dann muss pro ACK

$$\text{congwin} = \text{congwin} + \frac{\text{MSS}}{\text{congwin}} \cdot \text{MSS}$$

gesetzt werden[1]



[1] Die Berechnung ist eine Näherung (vgl. RFC 5681)

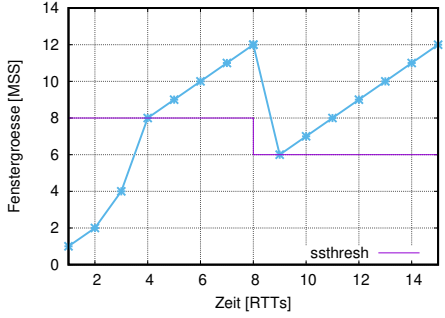
congwin und ssthresh: Beispiele

TECHNISCHE UNIVERSITÄT DARMSTADT

- Angenommen eine TCP-Verbindung beginnt bei $t = 0$, erstes Datensegment verschickt bei $t = 1$
- Initialer Wert von congwin $= 1\,\text{MSS}$
- Initialer Wert von ssthresh $= 8\,\text{MSS}$
- Vereinfachende Annahmen:
  - RTT ist konstant
  - Zeit diskretisiert auf volle RTT-Intervalle (z. B.: alle ACKs für am Beginn eines RTT-Intervalls versandte Pakete kommen zeitgleich am Ende der RTT an)
- Unmittelbar nach $t = 8\,\text{RTT}$ tritt ein TDACK auf

Wie entwickeln sich congwin und ssthresh im Zeitverlauf?

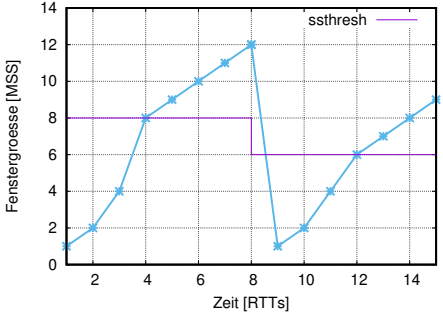
congwin und ssthresh: Beispiele



- Bei $t = 4$ wird der Anfangswert von ssthresh ($= 8$) überschritten ($\Rightarrow$ Übergang Slow Start $\rightarrow$ Congestion Avoidance)
- Nach $t = 8$ wird ein TDACK beobachtet ($\Rightarrow$ congwin wird halbiert)

Und wenn nach $t = 8$ stattdessen ein Timeout passiert?

congwin und ssthresh: Beispiele

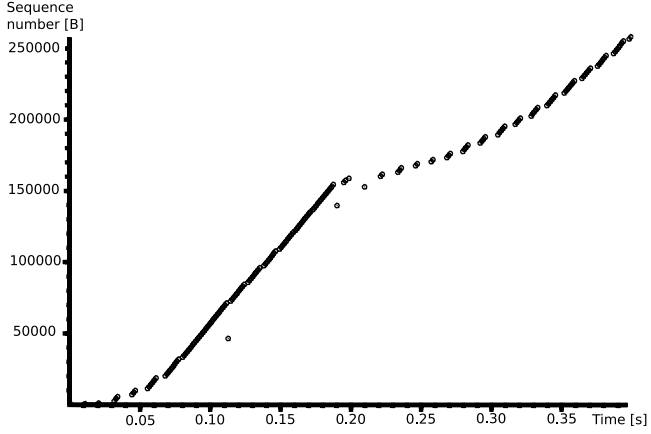


- Nach $t = 8$ tritt diesmal ein Timeout auf
  ($\Rightarrow$ congwin wird auf 1 MSS zurückgesetzt, ssthresh $= 12/2 = 6$)
- Bei $t = 12$ wird der neu gesetzte ssthresh überschritten

Zeit-Sequenznummern-Plot

TECHNISCHE UNIVERSITÄT DARMSTADT

- Hier eine Grafik, die den Verlauf der gesendeten Sequenznummer über die Zeit einer realen TCP-Verbindung zeigt; Slow Starts und ein paar Übertragungswiederholungen sind erkennbar



Self Clocking

TECHNISCHE UNIVERSITÄT DARMSTADT

- TCP benötigt für das Anpassen der Überlastfenstergröße keine Timer
- Die RTT im Netzwerk gibt den „Takt“ vor, nach dem das Fenster zu höheren Sequenznummern hin verschoben (und gleichzeitig vergrößert) wird
- …und damit auch den Takt, in dem neue Segmente den Sender verlassen dürfen
- Diese Eigenschaft nennt man auch *Self Clocking*
- Sie hilft bei der Implementierung: TCP-Sender brauchen trotz der komplexen Protokollregeln wenige Ressourcen

Warum exponentielle Timeouts?

TECHNISCHE UNIVERSITÄT DARMSTADT

- Wir hatten besprochen, dass Retransmission Timeouts in TCP exponentiell wachsen (Verdopplung der Wartezeit vor jeder Neuübertragung) – warum eigentlich?
- Um Überlast zu vermeiden!
- Neuübertragungen erzeugen zusätzliche Last, Congestion Collapse nur wegen Neuübertragungen droht
- Intuition:
  - angenommen, Neuübertragungen würden in festen Intervallen erfolgen, also nicht exponentiell
  - dann würde jede TCP-Verbindung, die gerade versucht, ein Paket zuzustellen, mit einer festen Rate (Paketgröße/Übertragungswiederholungsintervall) senden
  - wir haben aber schon gesehen, dass feste Raten zum Congestion Collapse führen können!
- Man kann zeigen, dass Übertragungswiederholungs-Intervalle, die langsamer als exponentiell wachsen, ein Netzwerk i. A. nicht aus so einer Situation „retten“ könnten

Fairness

TECHNISCHE UNIVERSITÄT DARMSTADT

- Angenommen, n Verbindungen teilen sich eine gemeinsame Leitung mit Übertragungskapazität R

Ist die TCP-Überlastkontrolle „fair“?

Vereinfachende Annahmen (die wir z. T. später fallen lassen):

- alle Verbindungen haben identische RTTs
- Verlusterkennung erfolgt stets durch TDACKs
- sonst keine Engpässe im Netz (d. h. keine anderen Paketverluste)
- Betrachtung im längeren zeitlichen Mittel

Fairness

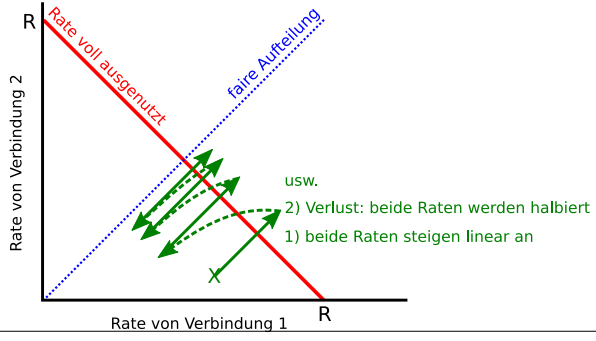
TECHNISCHE UNIVERSITÄT DARMSTADT

- Das Gesamtsystem konvergiert gegen eine gleichmäßige Aufteilung der Datenrate
- Erklärungsansatz: Die momentan schnelleren Verbindungen werden häufiger und stärker „gebremst“
- Warum?
  - Verlustwahrscheinlichkeit ist für alle Pakete gleich
  - Verbindungen mit momentan höherer Datenrate haben mehr Pakete unterwegs, sehen deshalb mit höherer Frequenz Verluste
  - wegen des Multiplicative Decrease wird außerdem von der schnelleren Verbindung bei einem Verlust mehr „abgezogen“

Fairness

TECHNISCHE UNIVERSITÄT DARMSTADT

- Betrachten wir die Entwicklung des Durchsatzes von zwei TCP-Verbindungen (mit gleicher RTT) über einen gemeinsamen Engpass in einem vereinfachten Modell
  - solange keine Verluste $\Rightarrow$ beide Verbindungen erhöhen ihre Rate gleich schnell
  - wenn der Gesamtdurchsatz zu hoch wird, treten Verluste auf und beide Verbindungen halbieren ihre Rate
- Konvergiert gegen eine gerechte Aufteilung!



(Abbildung: [nach Kurose, Ross])

Fairness: Unterschiedliche RTTs

TECHNISCHE UNIVERSITÄT DARMSTADT

Was passiert, wenn die Verbindungen unterschiedliche RTTs haben?

- Das Überlastfenster wächst (während Congestion Avoidance) linear um 1 MSS/RTT
- Wenn die RTT länger ist, wächst das Fenster also langsamer
  - das ist eine Nebenwirkung der Self-Clocking-Eigenschaft
- Dadurch ist eine TCP-Verbindung mit längerer RTT weniger „aggressiv“
- Wenn sie sich einen Engpass mit einer aggressiveren Verbindung (d. h. einer mit kürzerer RTT) teilt, dann bekommt sie weniger Bandbreite ab
- TCP-Verbindungen mit längerer RTT werden deshalb benachteiligt

Fairness: UDP

TECHNISCHE UNIVERSITÄT DARMSTADT

Was passiert, wenn sich TCP-Verbindungen einen Engpass mit UDP-Datenübertragungen teilen?

- UDP hat keine Überlastkontrolle
- Wird also nicht langsamer, wenn es „eng“ wird, nimmt keine Rücksicht auf TCP
- Natürlich *sollten* entsprechende Mechanismen in gefährdeten UDP-Anwendungen implementiert sein
- Aber wenn es hart auf hart kommt, dann zieht TCP den Kürzeren

Fairness: Parallele Verbindungen

TECHNISCHE UNIVERSITÄT DARMSTADT

Was passiert, wenn Anwendung *A* (z. B. ein Webbrowser) mehrere parallele TCP-Verbindungen öffnet, während Anwendung *B* gleichzeitig nur eine einzelne nutzt?

- Jede TCP-Verbindung bekommt denselben Anteil der Datenrate ab
- Die Anwendung mit mehreren parallelen Verbindungen bekommt also einen entsprechend größeren Anteil

Eignung für Multimedia-Übertragungen

TECHNISCHE UNIVERSITÄT DARMSTADT

- Für Multimedia-Übertragungen ist TCP nicht gut geeignet:
  - benötigt wird eine gleichmäßige Datenübertragung – TCP liefert ein Sägezahnmuster
  - benötigt werden möglichst konstante Latenzen – TCP wartet vor Übertragungswiederholungen
  - benötigt wird häufig keine Zuverlässigkeit – TCP hält sogar eingegangene Daten noch zurück, bis alles in der richtigen Reihenfolge abgeliefert werden kann
- Es existieren Ansätze, die auf Basis von UDP eine Überlastkontrolle mit gleichmäßiger Rate (ohne „Sägezähne“) realisieren
  - in der Praxis kaum verwendet
- Lösung meist: entweder mit den Einschränkungen von TCP leben, oder UDP verwenden

TCP über Long Fat Pipes

TECHNISCHE UNIVERSITÄT DARMSTADT

- *„Long Fat Pipes“* (oft auch *„Long Fat Networks“, LFNs*) sind Netzwerke mit sehr hohem Bandwidth-Delay Product (BDP)
- Sie stellen eine Herausforderung für das Verhalten von TCP dar
- Eines der Probleme hatten wir schon im Zusammenhang mit der Flusskontrolle gesehen, wurde dort mit Window Scaling gelöst
- Durch die Überlastkontrolle kommen nochmals neue Schwierigkeiten hinzu
- Beispiel: Übertragung eines (langen) Datenstroms von Brisbane (Australien) nach Perth (Australien) über gut 4300 km
  - RTT $= 70\,\text{ms} \;\Rightarrow$ 14,3 Data-ACK-Zyklen pro Sekunde
  - Datenrate der Leitung $R = 10\,\text{GBit/s} \;\Rightarrow\; \text{BDP} = 700\,\text{MBit}$
  - Paketgröße $=$ MSS $= 1500$ Byte (12 000 Bit)

TCP über Long Fat Pipes

TECHNISCHE UNIVERSITÄT DARMSTADT

- Bei Start mit congwin $=$ 1 MSS wird in der 17. RTT die Leitungskapazität überschritten
- Nach ca. $17 \cdot 70\text{ms} = 1{,}2\text{s}$ treten Duplicate ACKs auf
- Dann: halbieren der Rate, Eintritt in Congestion-Avoidance-Phase
- Congestion Avoidance beginnt mit einer anfänglich Übertragung von etwa 5 Gbit/s
- Pro RTT Steigerung der Datenrate um 171 kbit/s gesteigert
  (eine zusätzliche MSS alle 70 ms)

Wie lange dauert es, bis die Datenrate der Leitung erneut erreicht wird?

TCP über Long Fat Pipes

TECHNISCHE UNIVERSITÄT DARMSTADT

- Nach 29 240 RTTs bzw. ca. 34 min (!) sind 10 Gbit/s überschritten, sodass erneut ein Verlust auftritt
- Wiederum Drosseln auf 5 Gbit/s, usw.
- Die Congestion-Avoidance-Phase zeigt also ein Sägezahn-Verhalten mit einer Periode von rund 34 min!
- Es ergibt sich eine langfristige durchschnittliche Datenrate von 7,5 Gbit/s – im besten Fall
- Wenn gelegentlich andere Verluste auftreten, ist die erreichte Datenrate sehr viel niedriger
  - „gelegentlich“ heißt *sehr* gelegentlich: der o. g. Durchsatz wird nur dann erreicht, wenn während eines typischen 34-Minuten-Intervalls überhaupt *kein* Verlust eintritt
- Deshalb: Entwicklung mehrerer anderer TCP-Varianten

TCP in drahtlosen Netzwerken

TECHNISCHE UNIVERSITÄT DARMSTADT

- Auch in drahtlosen Netzwerken hat TCP Probleme
- Dort treten Paketverluste durch Übertragungsfehler relativ häufig auf
- TCP nimmt dann aber Überlast an
- Eine zügige Übertragungswiederholung wäre sinnvoll – TCP wartet jedoch auf Timeout (oder TDACK) und verringert dann auch noch unnötigerweise die Datenrate
- Deshalb: Entwicklung von noch mehr anderen TCP-Varianten

TCP-Varianten-Zoo

TECHNISCHE UNIVERSITÄT DARMSTADT

- Es gibt (sehr!) viele Varianten der TCP-Überlastkontrolle (hunderte, vielleicht tausende)
- Unterscheiden sich darin, wie Überlast erkannt und behandelt wird
- Die „klassische“ Variante, die wir hier (mit Vereinfachungen. . .) besprochen haben, heißt TCP Reno
- Ziel ist immer:
  - kompatibel sein mit anderen TCP-Varianten (jeder soll mit jedem reden können)
  - möglichst fair sein zu anderen TCP-Varianten (beim gemeinsamen Durchqueren eines Engpasses)

TCP-Varianten-Zoo

TECHNISCHE UNIVERSITÄT DARMSTADT

Wichtige und bekannte TCP-Spielarten sind z. B.:

- TCP Tahoe: Der „Urahn“ und Vorläufer von TCP Reno; keine Sonderbehandlung von TDACKs, nach jedem Verlust ein Slow Start
- TCP NewReno: veränderte Regeln, wie/wann/welche Segmente nach TDACKs übertragen werden
- TCP Vegas: beobachtet zum Zweck der Überlastkontrolle Veränderungen der RTT; kann auf diese Weise wachsende Warteschlangen früh erkennen dadurch schon vor dem Entstehen von Verlusten auf bevorstehende Überlast reagieren
- CUBIC (derzeit in Linux) und Compound TCP (zeitweise in Windows): Anpassungen für Long Fat Pipes
- BBR („Bottleneck Bandwidth and Round-trip propagation time“): kombiniert Ideen von TCP Vegas (Beobachtung der RTT) mit „Probing“ um die Bottleneck-Rate zu ermitteln

Grundidee hinter CUBIC

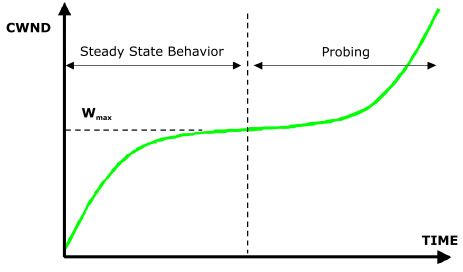
TECHNISCHE UNIVERSITÄT DARMSTADT

- Standard-Variante von TCP im Linux-Kernel seit Version 2.6.19
- …mit einem völlig anderen Verfahren zum Bestimmen des Congestion Window:
  - keine Unterscheidung Slow Start / Congestion Avoidance
  - nicht mehr self-clocking, Fensterwachstum unabhängig vom Eintreffen von ACKs (stattdessen zeitabhängig)
- Die Größe des Fensters über die Zeit wird von einer kubischen Funktion beschrieben
- Bei einem Verlustereignis wird die Fenstergröße reduziert
- Wächst dann wieder entsprechend einer kubischen Funktion
- Die Parameter der kubischen Funktion für die neue „Runde“ werden abhängig von der bis zum Verlust erreichten Fenstergröße neu bestimmt

Fensterwachstum bei CUBIC

TECHNISCHE UNIVERSITÄT DARMSTADT

- Die kubische Funktion führt
  - erst zu einem schnellen Wachsen des Fensters: zügiges Wiederherstellen der Höhe vor dem Verlust
  - dann zu einem „Plateau“, während dem das Fenster weitgehend unverändert bleibt: Parameter werden so justiert, dass die Höhe dem letzten funktionierenden Fenster vor dem Verlust entspricht (in der Abbildung W_{max})
  - und dann wieder zu einem schnellen Anstieg: Austesten, ob noch mehr geht



Eigenschaften von CUBIC

TECHNISCHE UNIVERSITÄT DARMSTADT

- Vermeidet Unfairness zwischen Verbindungen mit unterschiedlichen RTTs, da das Fensterwachstum von der RTT unabhängig ist
- Kann Long Fat Pipes besser ausnutzen, da das Wachstum sich immer schnell dem vorherigen Fenster annähert
- Parameter sind so justiert, dass das Teilen eines Engpasses mit „klassischen“ TCP-Varianten ebenfalls (einigermaßen) fair geschieht

Zusammenfassung

TECHNISCHE UNIVERSITÄT DARMSTADT

- In diesem Kapitel haben wir die Transportschicht betrachtet
- Wir haben uns (kurz) UDP und (lange) TCP angeschaut
- Wir haben uns über die unterschiedlichen Dienstmodelle der Protokolle unterhalten
- Wir haben gesehen, wie man Zuverlässigkeit über einen unzuverlässigen Übertragungskanal (wie z. B. die Internet-Netzwerkschicht) erreichen kann
- Wir haben die TCP-Zuverlässigkeitsmechanismen kennen gelernt
- Und wir haben erkannt, dass Flusskontrolle und Überlastkontrolle notwendig sind und wie sie in TCP umgesetzt werden
- Jetzt: Wieder ein Schritt weiter nach unten im Protokollstapel, in die Netzwerkschicht