Willkommen bei Kommunikations- und Netztechnik!

– *Von Kupferkabel, Glasfaser und Mikrowelle*
über Telefon, Ethernet und TCP
zu E-Mail, Webserver und REST.

– ‥‿

– Heute: **Transportschicht: Von Anwendung zu Anwendung.**

Organistatorisches

- sci-hub und libgen bekannt?

Wiederholung

- Routing: Quell- Senken Bäume
- VC (virtuelle Verbindung) vs. Paket
- Fluten + Optimierung
- Routing-Tabellen
- Warteschlangen verstehen
- Drosseln
- IPv4 vs IPv6

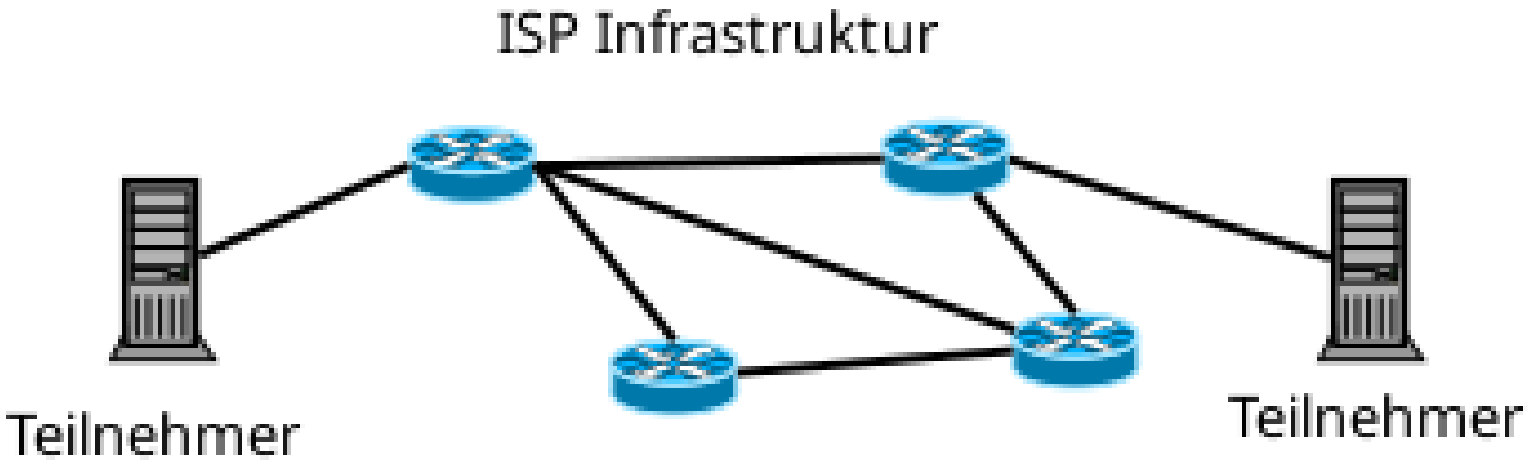
Ziele heute I

- Sie verstehen, dass die Transportschicht Segmente mit eigenem Header an die Vermittlungsschicht reicht
- Sie verstehen, dass die Transportschicht Prozesse verbindet und über Ports addressiert und IPs über die Vermittlungsschicht laufen
- Sie können ein 3-way Handshake-Diagramm aufschreiben
- Sie können ein 3-way Verbindungsabbau-Diagramm aufschreiben
- Sie wissen, dass es bei 2 Teilnehmenden immer essentielle Nachrichten gibt, die nicht verloren gehen dürfen
- Sie wissen, dass Datenverlust bei Servercrash unvermeidbar ist
- Sie verstehen AIMD (additive increase multiplicative decrease)

Ziele heute II

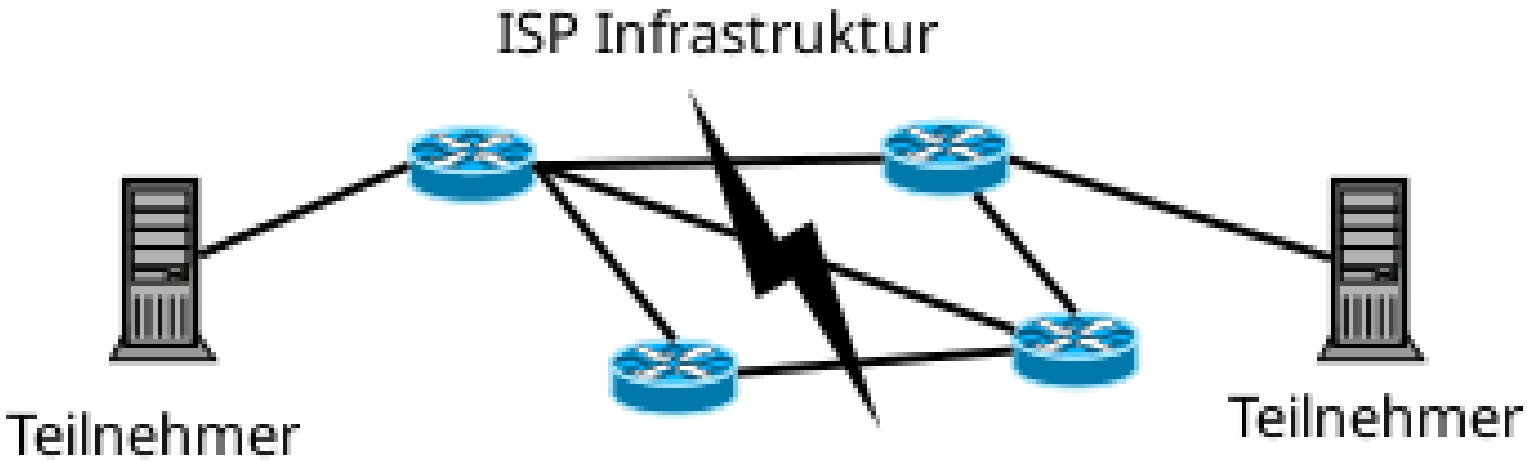
- Sie kennen den TCP-Header
- Sie können einen Varianzbasierten RTO berechnen (retransmission timeout)

Gründe für Transportschicht



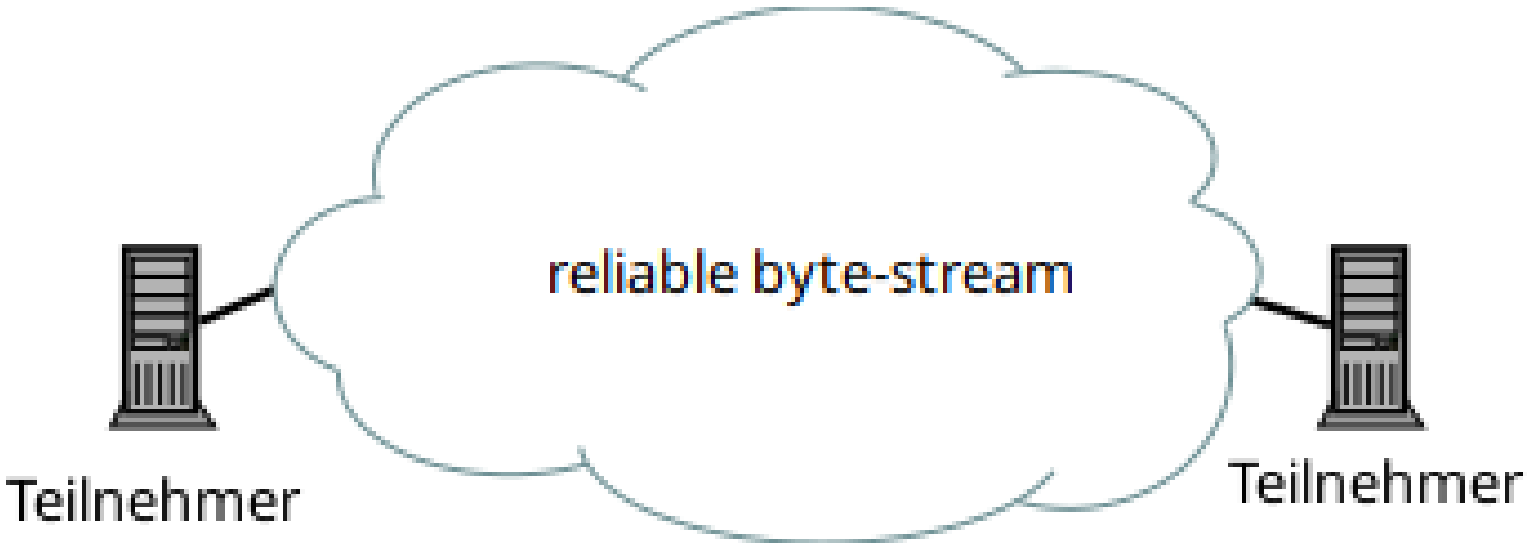
- Transportschicht: läuft auf Computer der Teilnehmer
- Netzwerkschicht: läuft auf Netzwerk-Hardware der Provider

Gründe für Transportschicht



- in der Netzwerkschicht kommt es zu Problemen
- diese können nicht auf Netzwerkschicht behoben werden

Gründe für Transportschicht



Transportschicht macht unzuverlässige Netzwerkschicht zuverlässig

Aufgaben der Transportschicht

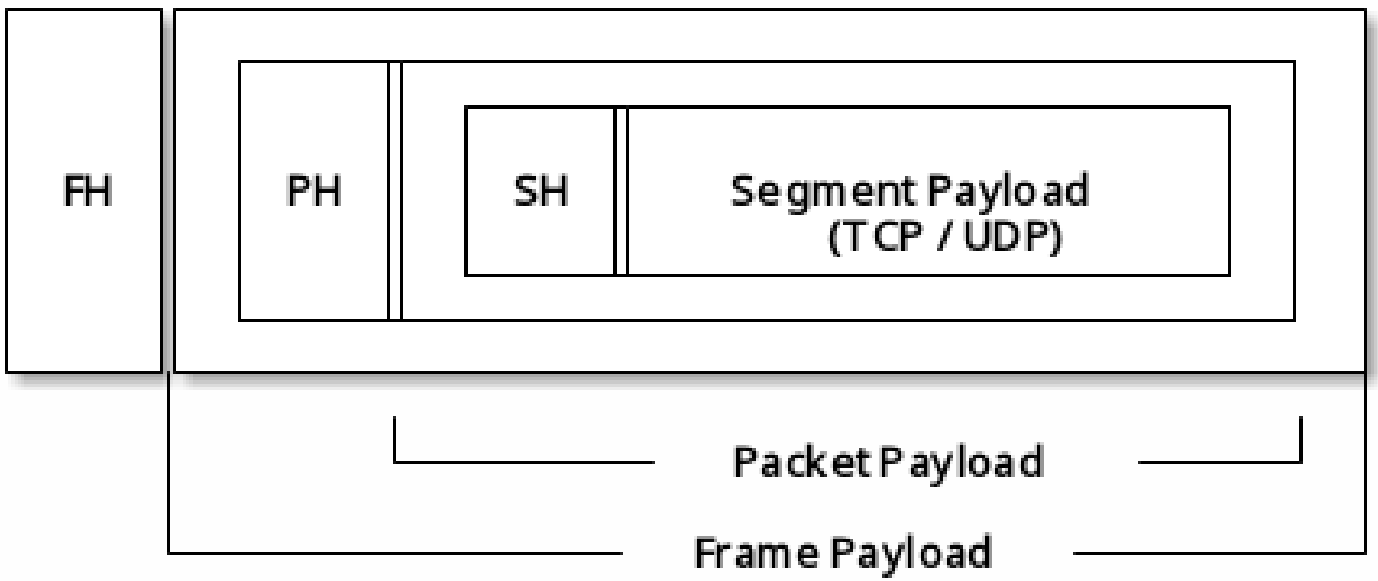
- Zuverlässigkeit
- Effizienz
- Flusskontrolle
- Überlastkontrolle

Gemeinsamkeiten Vermittlungsschicht

- verbindungslos und -orientiert möglich
- Adressierung von Hosts
- Flusskontrolle

Segmente: Noch ein Header

Die Transportschicht verschickt Segmente, die in Netzwerkschicht Pakete eingebettet sind.



Praktisch: Berkeley Sockets

Werden in vielen OS (Operating System) verwendet.

Function	Bemerkung
`socket()`	definiere verwendetes Protokoll
`bind()`	ordne Socket eine Netzwerkadresse zu
`listen()`	erzeuge Queue, bereit für Verbindung
`connect()`	blocking; baut Verbindung zu Server auf
`accept()`	blocking; erzeugt Filedescriptor für Verbindung
`send()`, `receive()`	sende und empfange Daten
`close()`	beende Verbindung

Zusammenfassung

- Kanal zwischen **Prozessen**
- Segmente in Paketen

Eigenschaften von Transport Protokollen

- Probleme bei Paketen:
  - out-of-order
  - packet loss
  - duplication
- Aufgaben Transportschicht:
  - Fehler-
  - Fluss-
  - Überlastkontrolle
  - Sequenzierung

Adressierung

- Netzwerkschicht: Adresse (Bsp: IP)
- Transportschicht: Ports (Bsp: 80 HTTP)
- Ports werden verwendet, um eine IP für mehrere Prozesse zu teilen
- Problem: auf welchen Port soll connected werden?
  - well known ports: 22, 25, 80, 443
  - portmapper: wie Telefonauskunft

Problem: doppelte Segmente

Folgendes Szenario:

- Überweisung per Online Banking
- Segment mit Überweisung benötigt zu lange
- wiederholte Übermittlung
- beide Segmente kommen an
- Überweisung wird doppelt ausgeführt

Lösung: Ids?

- jedes Segment verfügt über eine Id
- Nachteil: Sender & Empfänger müssen Buch führen
- Buch muss auch Neustarts überleben
- –> teuer

Beschränkung der Lebenszeit

- Lebenszeit von Segmenten beschränken
  - Hop Counter
  - Timestamp
- –> garantiert, dass Segmente sterben können
- Id kann nach Periode T wiederverwendet werden
- T ist mehrfaches der maximalen Paketlebenszeit
- im Internet: 120s

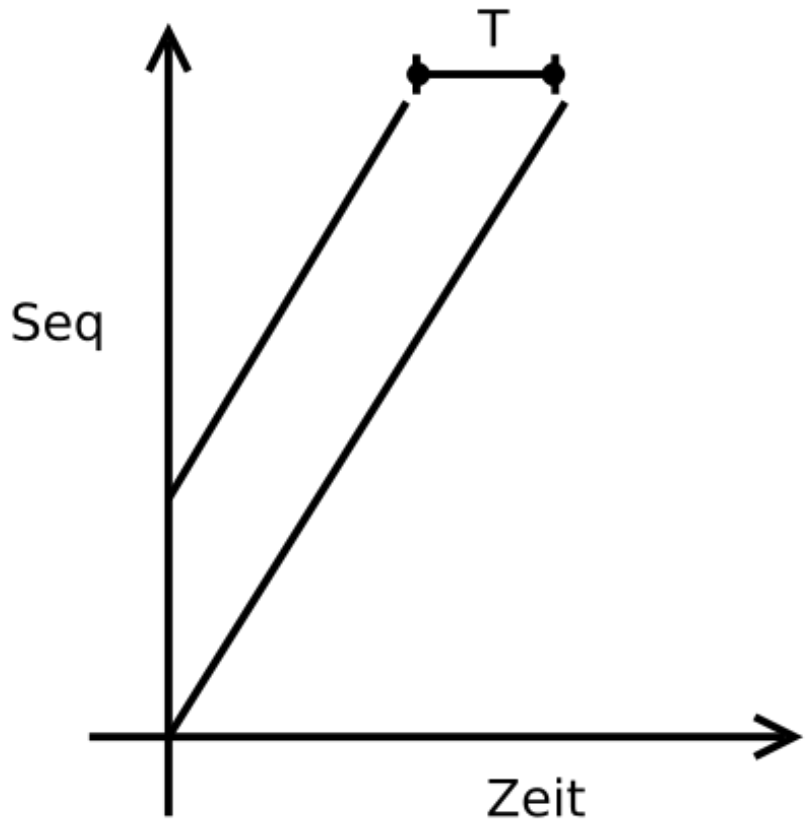
Umsetzung nach Sunshine und Dalal

- Segmente werden mit Sequenznummer versehen
- Sequenznummer ist innerhalb T eindeutig
- Maximum Seq abhängig von Rate und T
- Duplikate können immer noch existieren

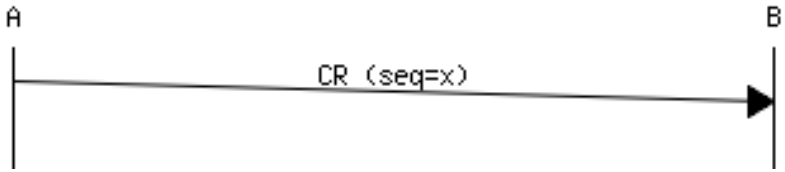
Was passiert bei einem Neustart?

- wie wird die Sequenznummer initialisiert?
  - muss die Garantie erhalten
- Möglichkeiten:
  - warte T Sekunden nach Neustart
  - Sequenznummer basiert auf Uhr, die auch während Ausfall weiterläuft

T und die Segmentrate

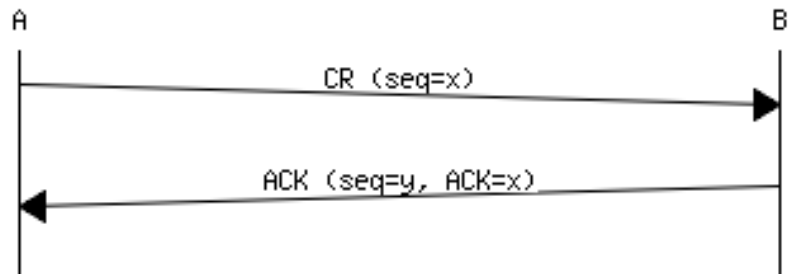


3-way Handshake



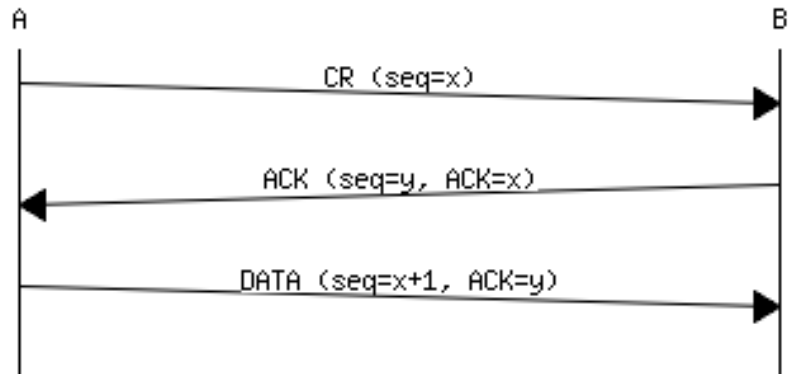
- A möchte sich mit B verbinden
- A sendet `CONNECTION REQUEST` mit eigener Sequenznummer x

3-way Handshake



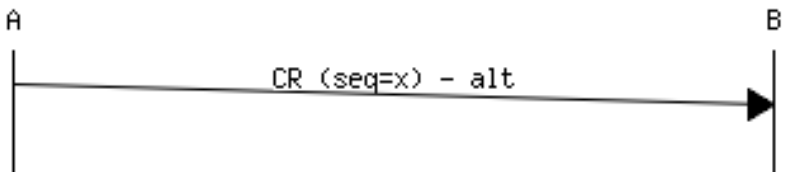
- B bestätigt `CR` mit Bestätigung `ACK` (Acknowledgment)
- Bestätigung enthält nächste Sequenznummer (x+1)
- sendet dabei eigene Sequenznummer y

3-way Handshake



- A sendet Daten und bestätigt dabei y

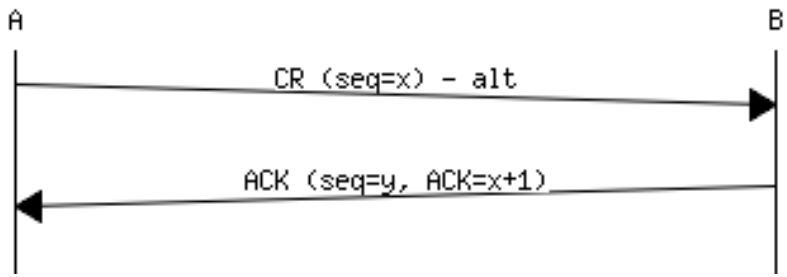
Handshake mit alten Segmenten



- B empfängt `CR`
- `CR` ist alt
- B kann das nicht erkennen

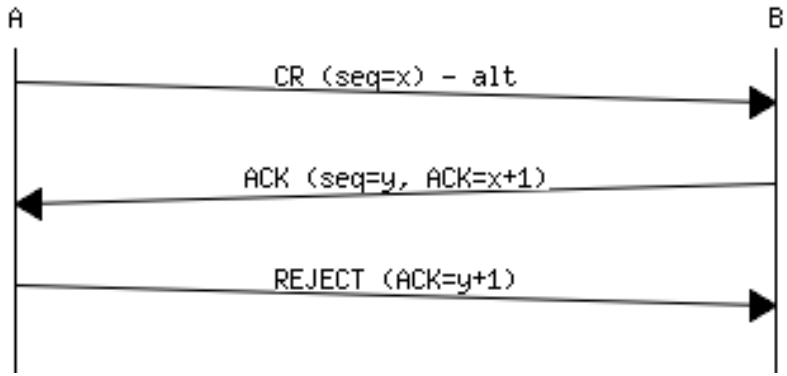
3-way Handshake

–



- B antwortet wie gewohnt mit `ACK`

3-way Handshake

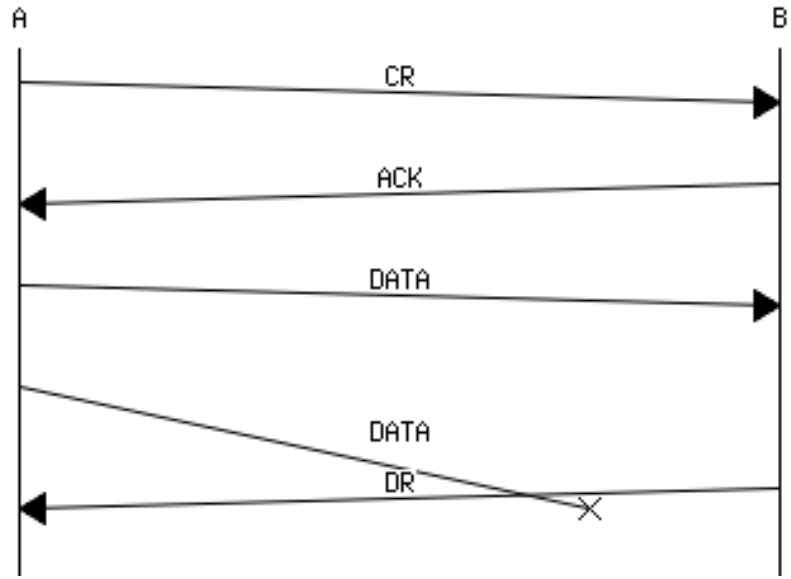


- A erhält `ACK` für x+1
- A weiß, dass x+1 alt ist
- A lehnt Verbindung ab

Sequenznummern und TCP

- Netzwerkverbindungen werden immer schneller
  - TCP verwendet 32 bit Sequenznummer
  - –> zusätzlicher Timestamp, um Verbindungen zu schützen
- weiteres Problem: vorhersagbare Sequenznummern
  - problematisch für Sicherheit
  - –> Initialisierung mit zufälligem Wert

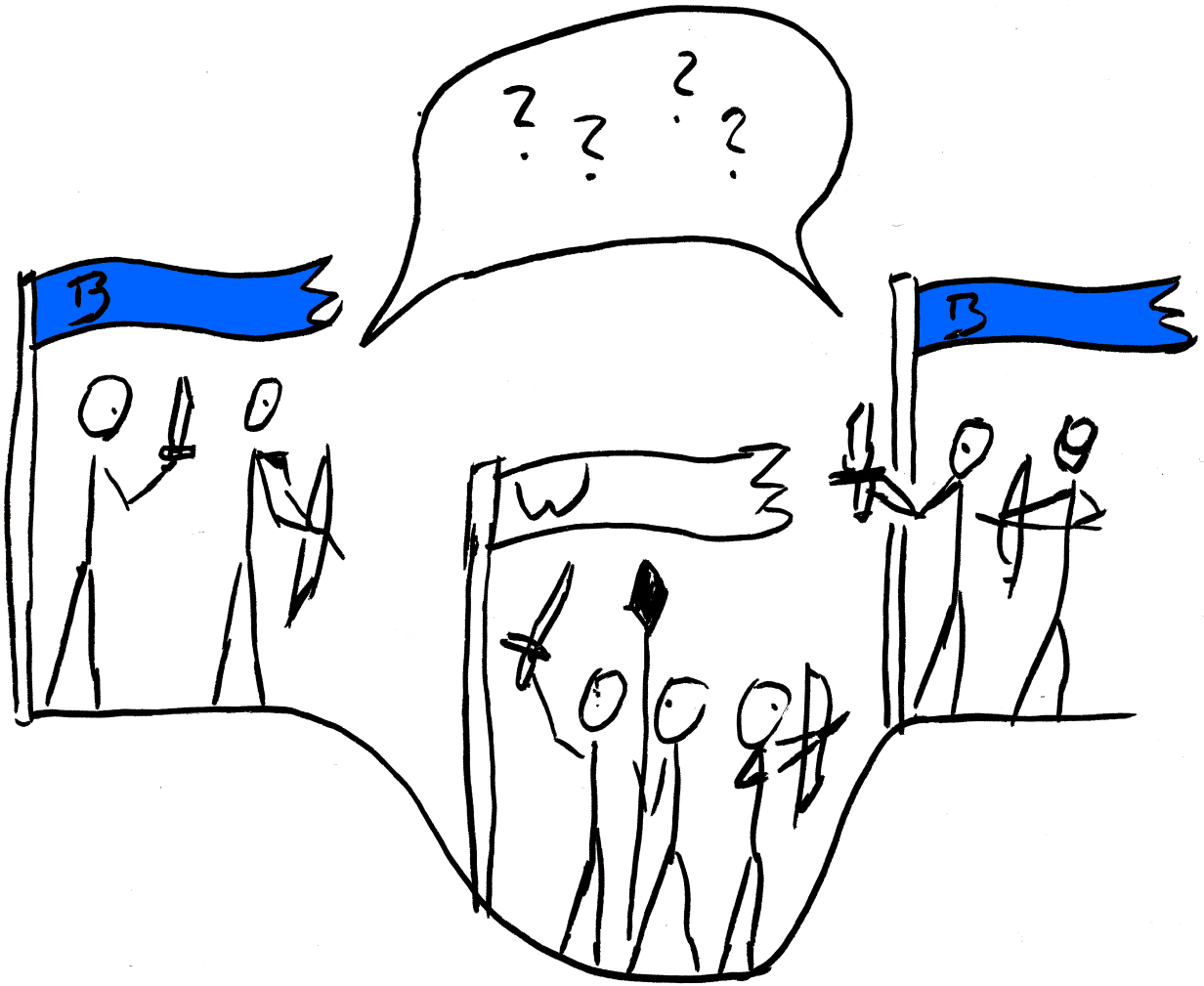
Verbindungsabbau - Datenverlust



- asymmetrisch: Verbindung wird geschlossen, sobald ein Partner beendet
- Daten können verloren gehen
- DR: Disconnection Request (hier nach timeout)

3-way Handshake

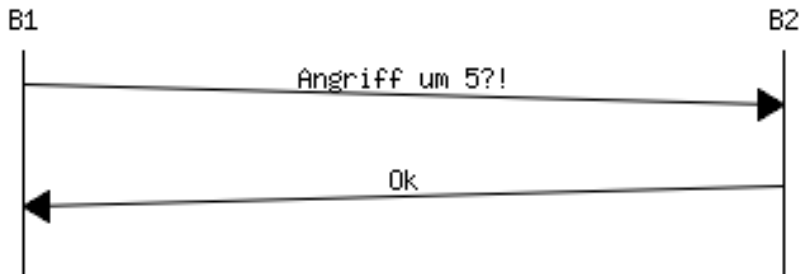
Two Army Problem



Two Army Problem, Fragen

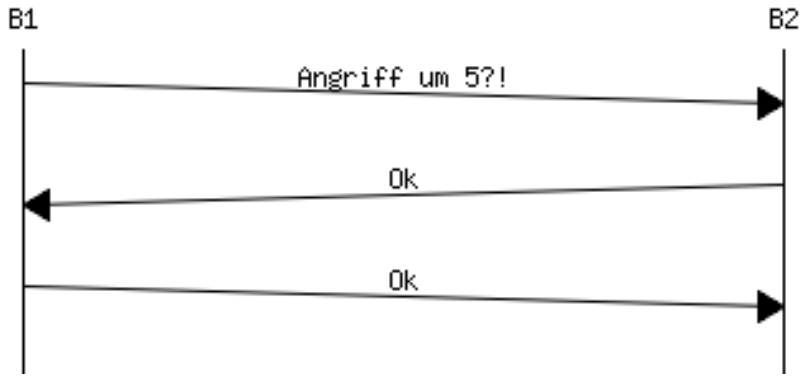
- Blaue Armeen können einzeln nicht die weiße Armee besiegen
- –> müssen ihren Angriff koordinieren
- dazu werden Boten verwendet
- Boten können gefangen werden
- Kann ein gemeinsamer Angriff koordiniert werden?

3-way Handshake



- wird Armee B2 angreifen?
- B2 kann sich nicht sicher sein, dass `Ok` ankam
- 3-way handshake als Lösung?

3-way Handshake



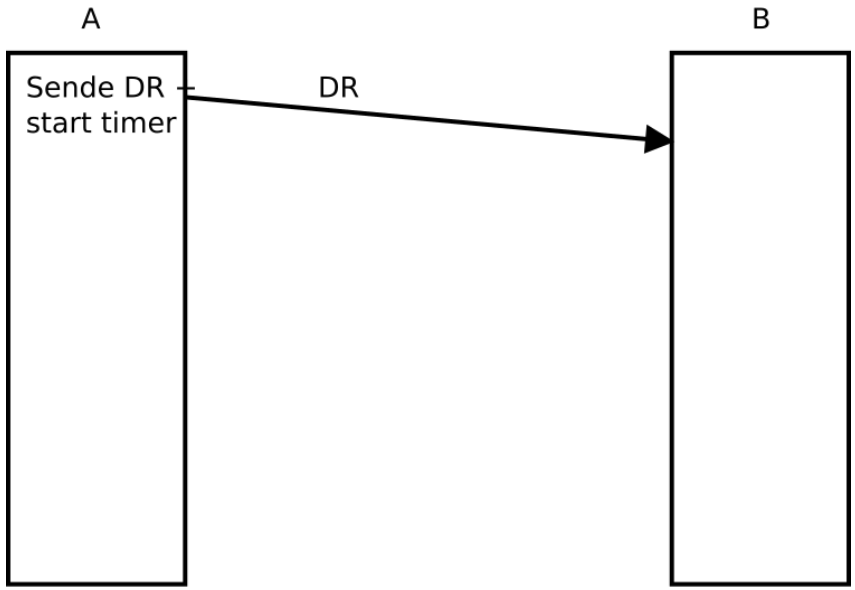
- nun kann sich B1 nicht sicher sein, ob `Ok` ankam
- 4-way handshake? nein

–

Allgemein:

- angenommen es existiert ein Protokoll
- letzte Nachricht ist entweder essentiell oder nicht
- falls nicht kann sie entfernt werden
- jede nicht essentielle Nachricht wird entfernt
- –> Protokoll besteht nur noch aus essentiellen Nachrichten
- –> sobald eine essentielle Nachricht verloren geht, funktioniert das Protokoll nicht mehr

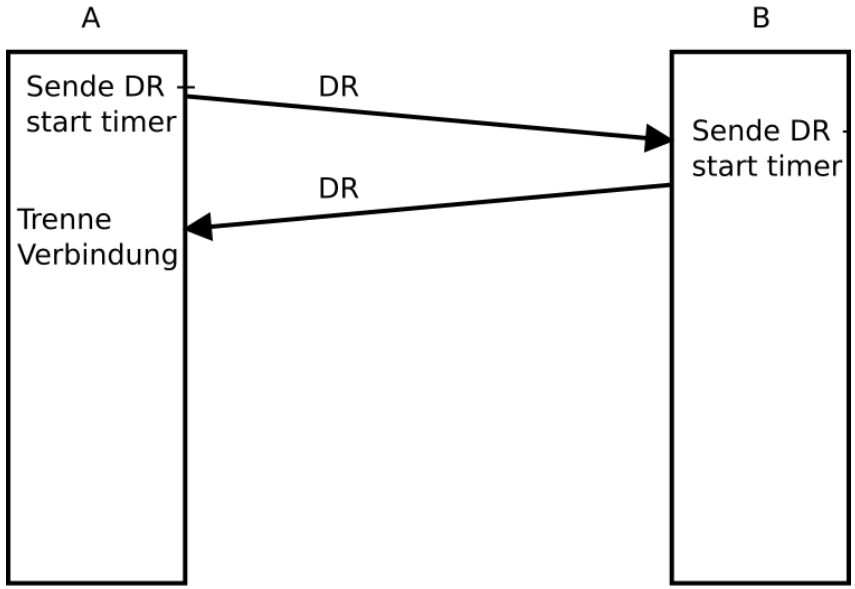
3-way handshake Verbindungsabbau



- A möchte Verbindung schließen
- sendet dazu `DR`
- startet gleichzeitig einen Timer

3-way handshake Verbindungsabbau

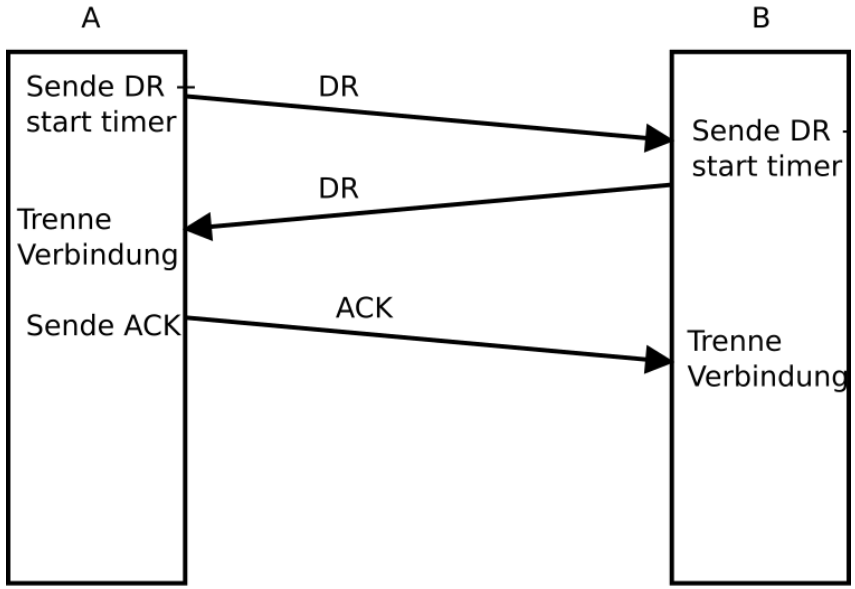
Verbindungsabbau Schritt 1: DR



- B empfängt `DR`
- B startet Timer
- B sendet `DR`
- A empfängt `DR`
- A trennt Verbindung

3-way handshake Verbindungsabbau

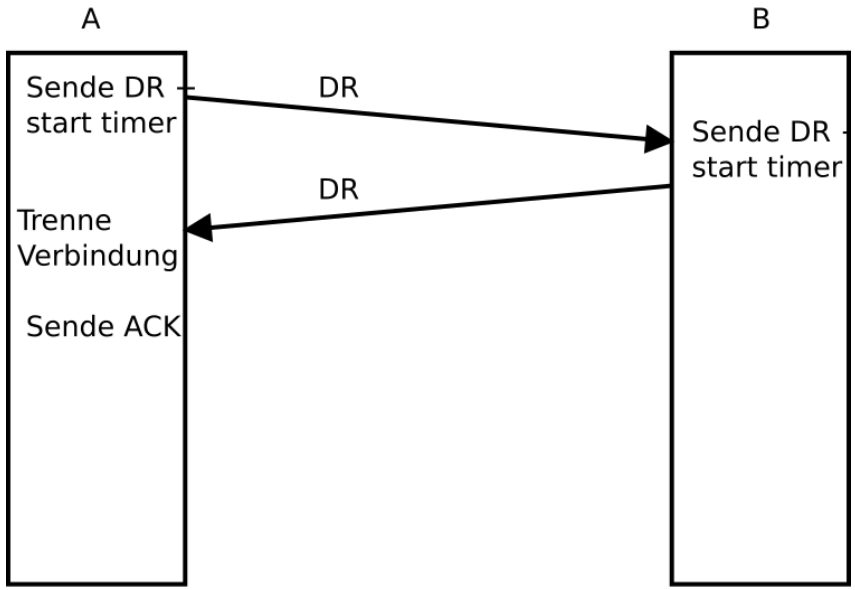
Verbindungsabbau Schritt 2: ACK



- A sendet `ACK`
- B empfängt `ACK`
- B trennt Verbindung

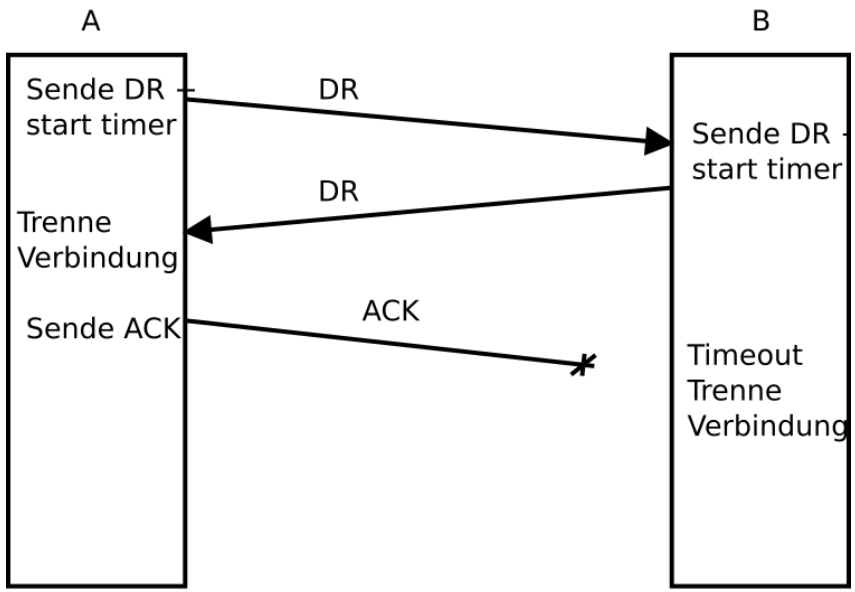
Was kann hier alles **schief gehen**?

Beispiel: ACK-Verlust



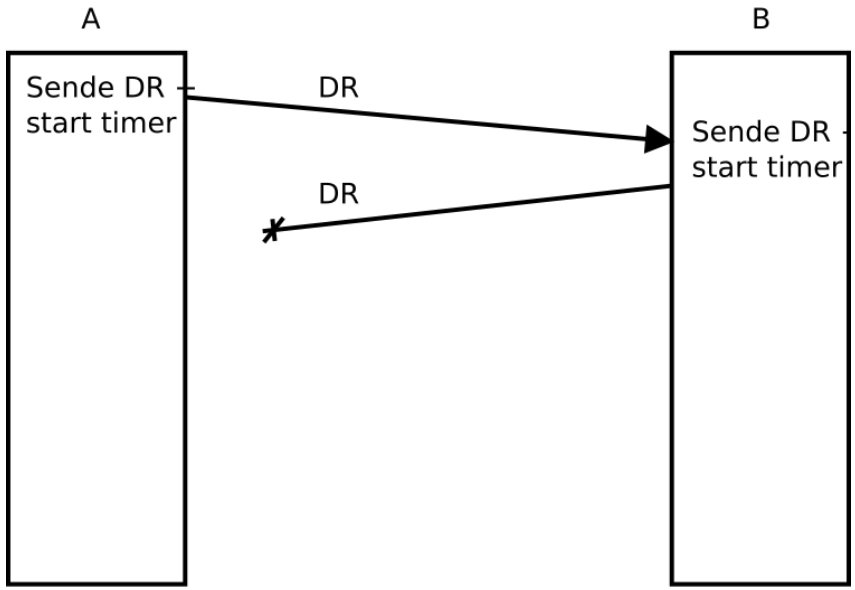
- Anfang wie gewohnt
- aber: `ACK` geht verloren
- was passiert?

Beispiel: ACK-Timeout



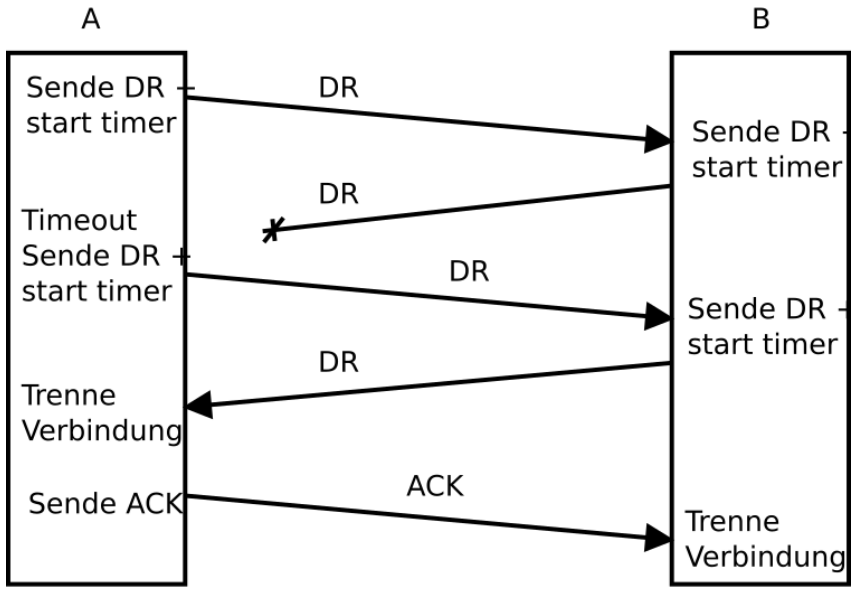
- B erreicht Timeout
- B trennt Verbindung

Beispiel: DR-Verlust



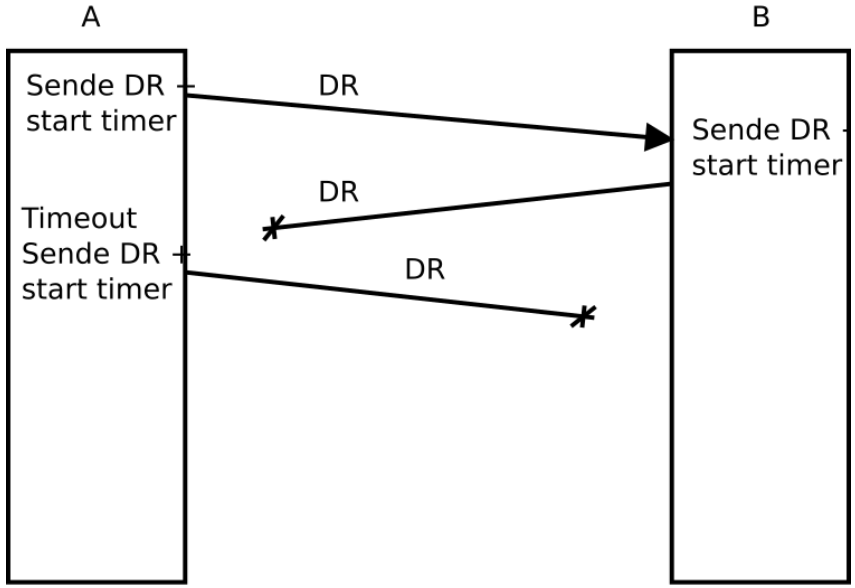
- B's `DR` geht verloren
- wie reagiert A?

Beispiel: DR-Verlust -> Timeout mit Retransmission



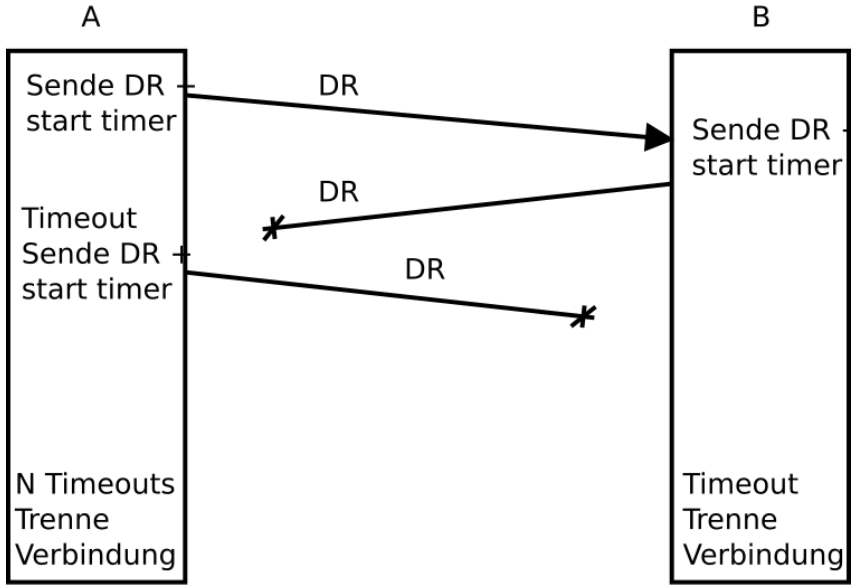
- A erreicht Timeout
- A sendet erneut `DR`
- Verbindung wird wie gewohnt getrennt

Beispiel: Doppelverlust



- wie zuvor
- aber A's zweite DR geht auch verloren
- dies wiederholt sich
- wird die Verbindung je getrennt?

Beispiel: Aufgeben



- B erreicht Timeout
- B trennt Verbindung
- A erreicht wiederholt, alle `DR` gehen verloren
- A gibt nach N Versuchen auf und trennt Verbindung

Fazit Verbindungsabbau

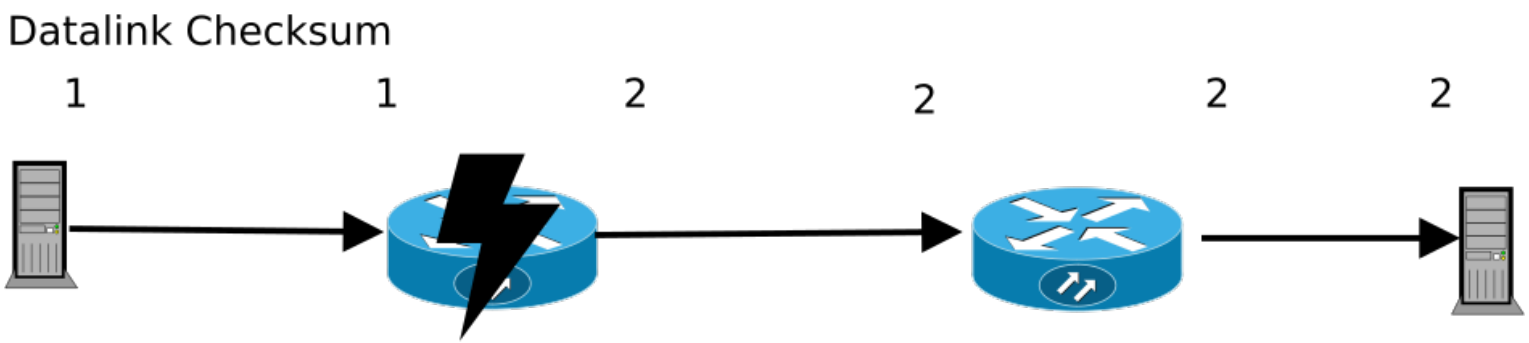
- Datenverlust kann nicht verhindert werden
- Lösung mit Timern aber 'good enough'
- –> Problem muss auf höherer Schicht gelöst werden
- Wie sieht es bei HTTP aus?

HTTP

- HTTP folgt Request/Response Prinzip
- Ablauf ohne Verbesserungen (persistent connection etc.):
  - TCP Verbindungsaufbau (3-way)
  - HTTP Request Browser -> Server
  - ACK Server -> Browser
  - HTTP Response Server -> Browser
  - ACK Browser -> Server
  - TCP Verbindungsabbau (3-way)
- –> nach Response sind alle Daten da, Verbindungsabbau ungefährlich

Fehlerkontrolle

- ähnlich Sicherungsschicht (Datalink)
- CRC Checksumme o.ä.
- aber:
  - Sicherungsschicht sichert nur zwischen Geräten
  - Transport Layer sichert Ende-zu-Ende



Flusskontrolle

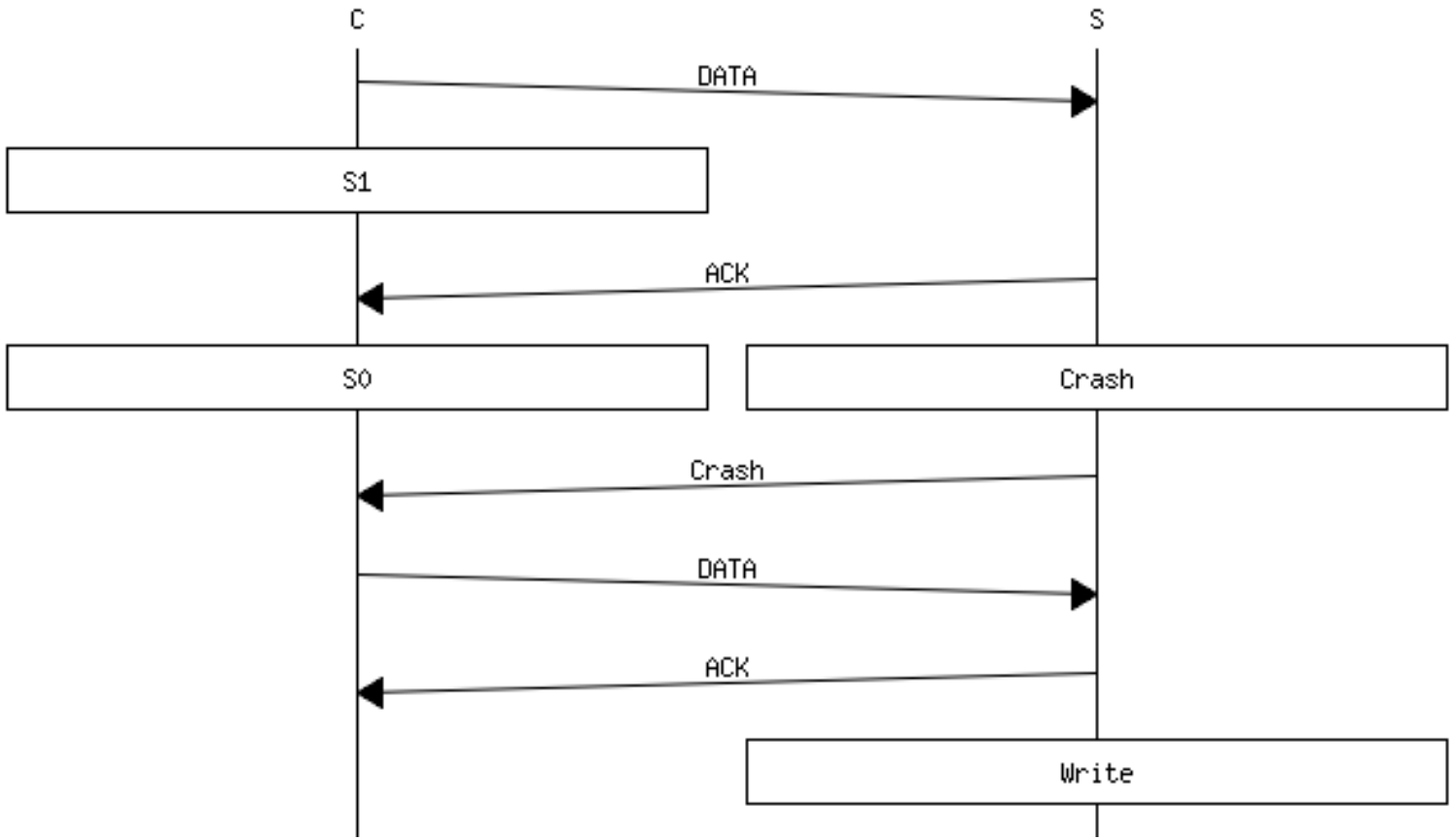
- ähnlich Sicherungsschicht
- Stop and Wait?
  - $\tau = 100$ms
  - –> Zeit für `DATA` und `ACK`: 200 ms
  - –> 5 Segmente pro Sekunde `:-/`
- –> TCP verwendet große Sliding Windows

Crash Recovery mit Anwendung

- Ausgangssituation: Server crasht und informiert Clients danach über Crash
- Client kann in 2 Zuständen sein
  - `S0`: no segment outstanding
  - `S1`: one segment outstanding
- Server:
  - `AW`: first `ACK`, then write
  - `WA`: first write, then `ACK`
  - `C`: Wo er crasht

-

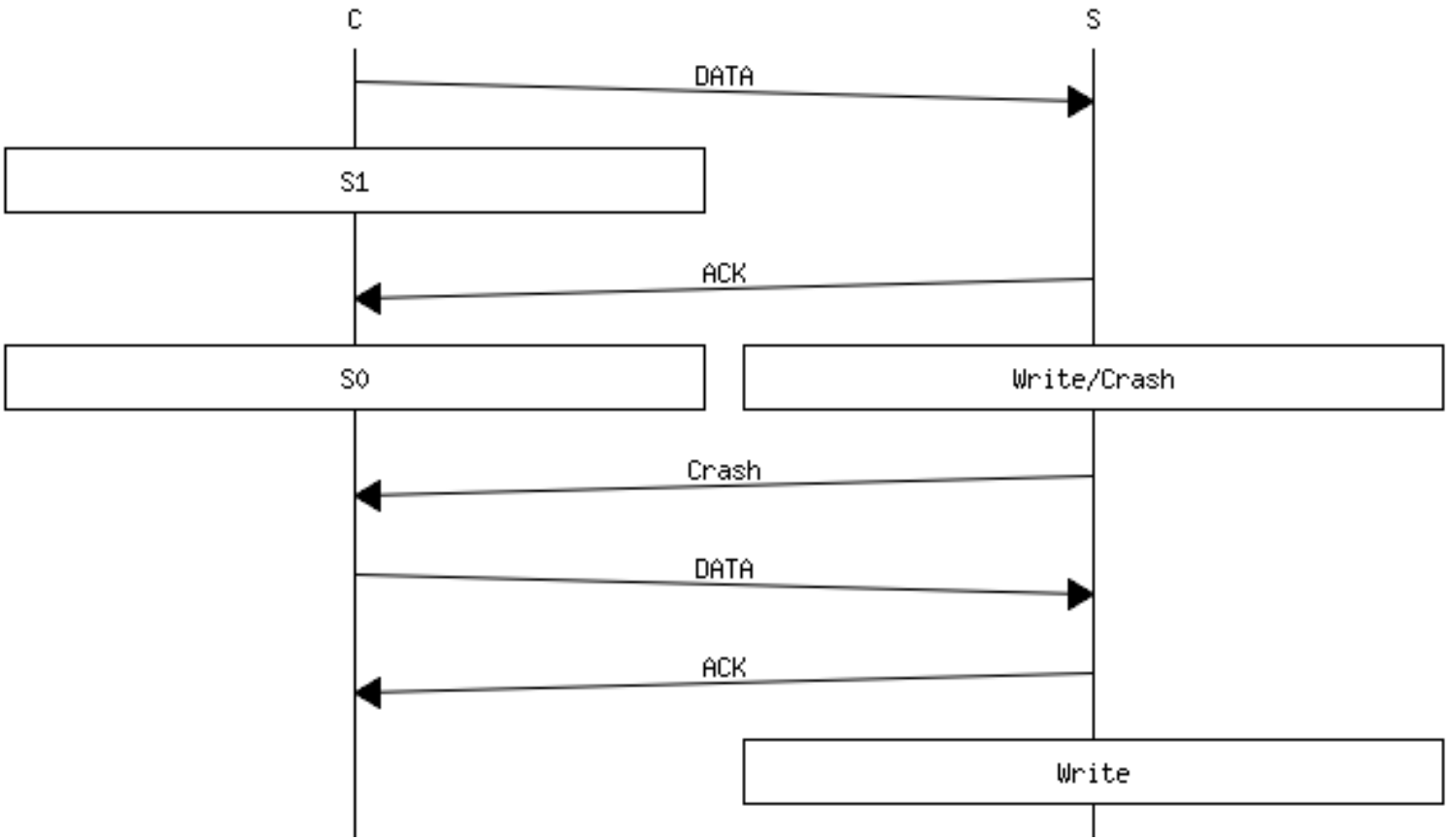
Beispiel: Client (Retransmit in S0), Server (AC(W))



–> alles ist Ok

Übung: Client (Retransmit in S0), Server (AWC)

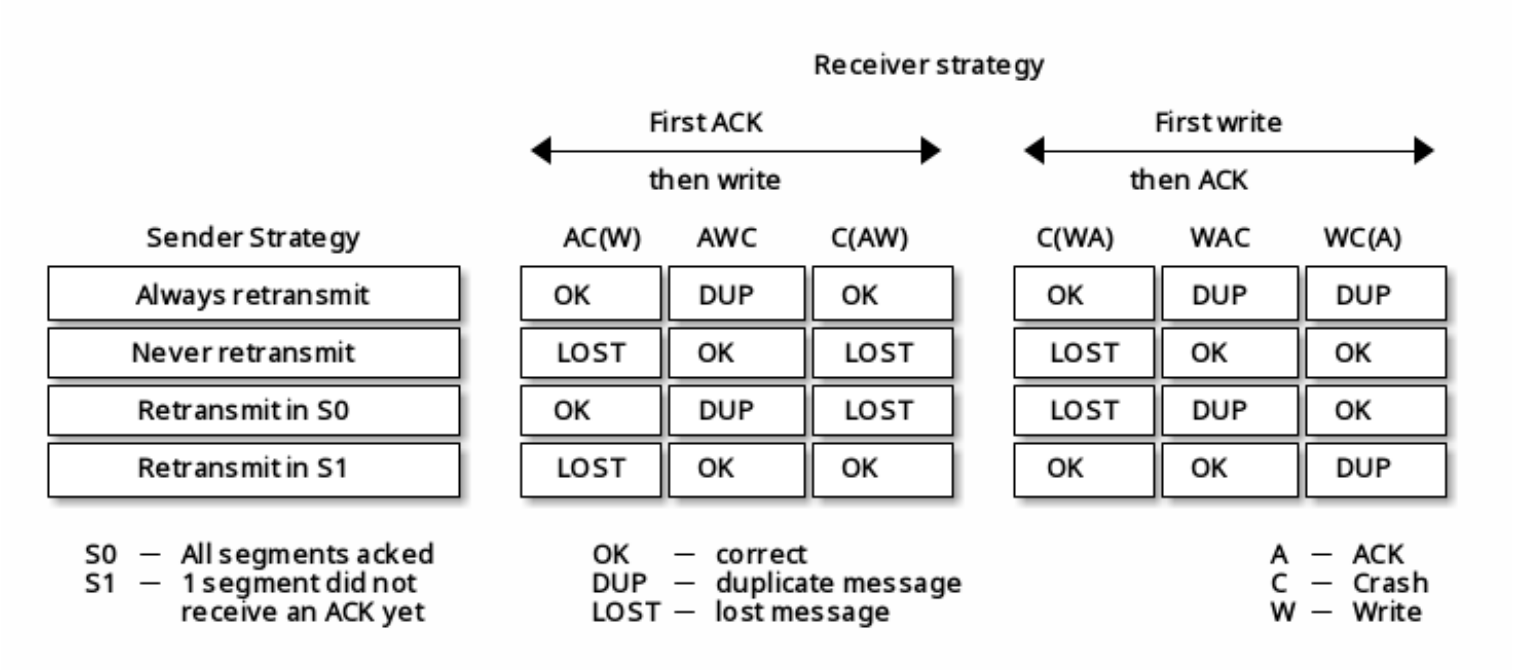
Lösung: Client (Retransmit in S0), Server (AWC)



–> doppelte Daten

Fehlerkontrolle

-



–> kann nicht transparent gestaltet werden, muss in höherem Layer behandelt werden

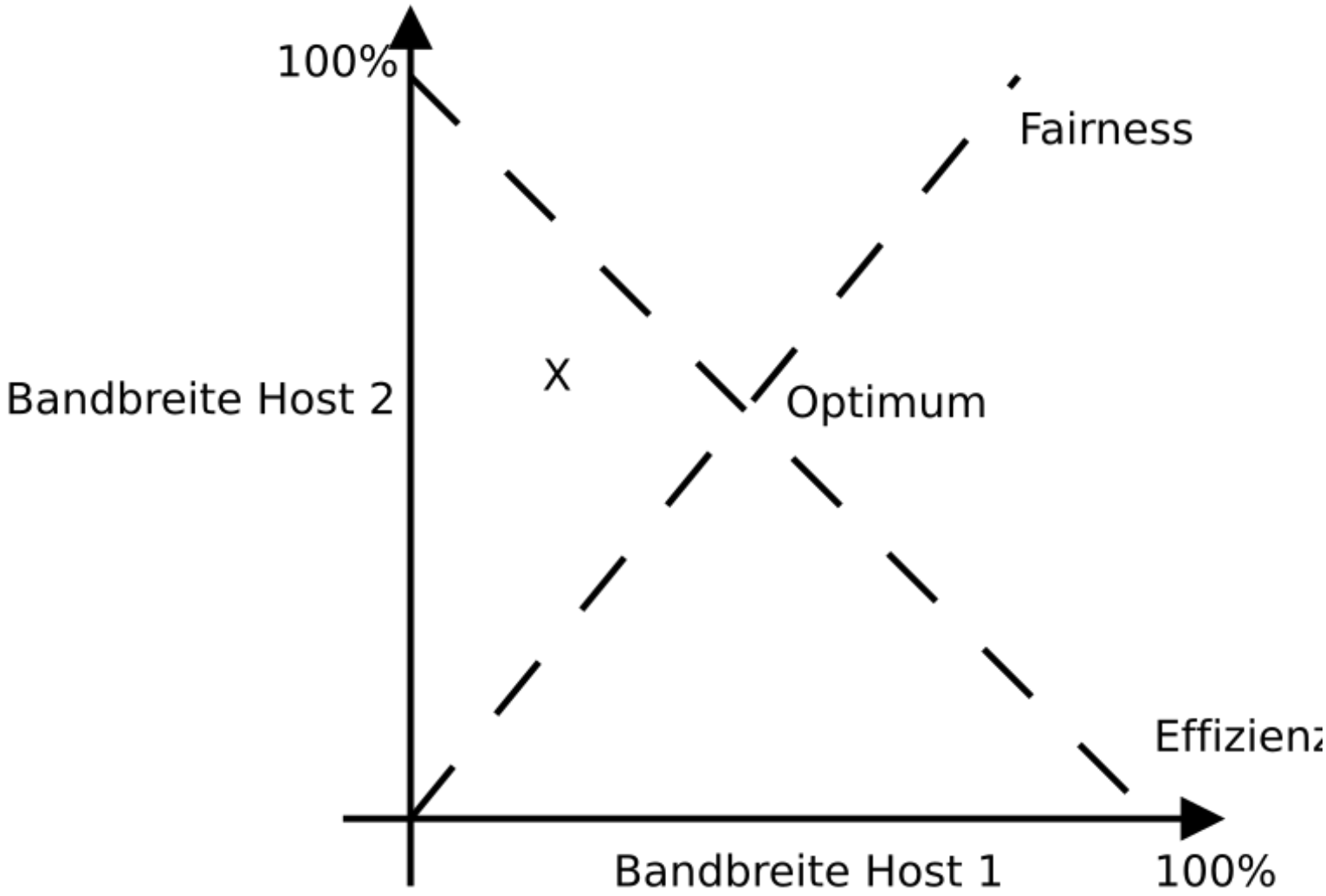
Ende-zu-Ende Überlastkontrolle - 3 Kriterien

- Effizienz: wie ist das Netzwerk insgesamt ausgelastet?
- Fairness: hat jeder Teilnehmer einen fairen Anteil an der Kapazität?
- Konvergenz: Algorithmus soll schnell konvergieren
  - Netzwerklast ändert sich ständig

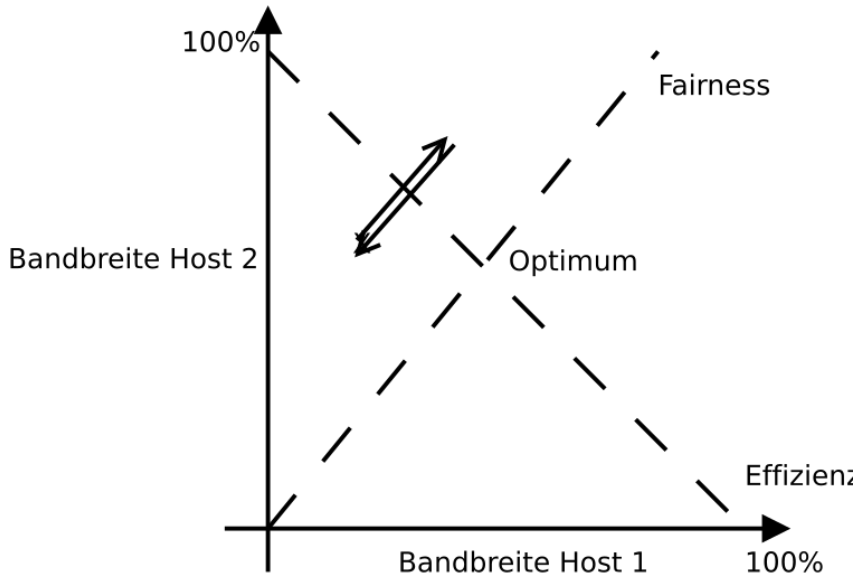
Überlastkontrolle - Anpassung der Senderate

- Senderate kann erhöht und gesenkt werden
- dies geschieht entweder:
  - additiv: $R_n = R_{n-1} + C$
  - multiplikativ: $R_n = R_{n-1} * C$
  - R: Senderate
  - C: Konstante

Startpunkt der Bandbreiten-Nutzung

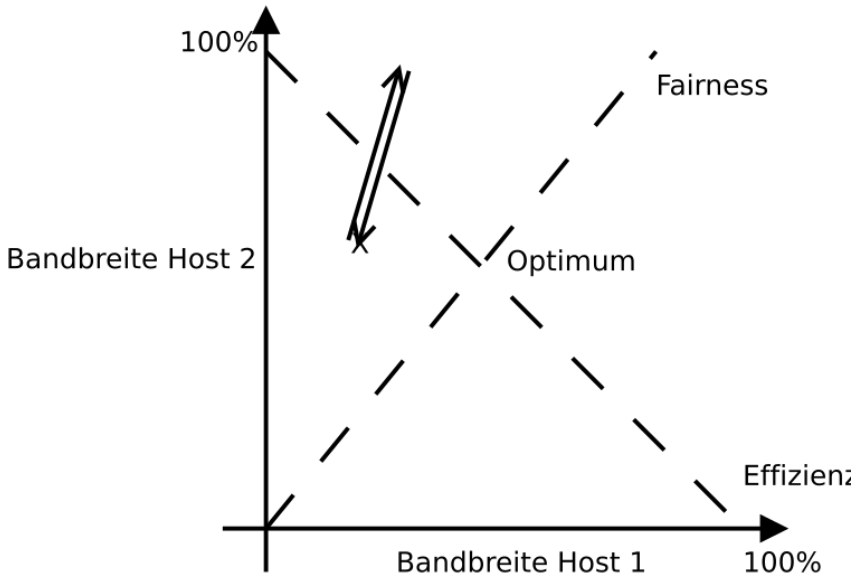


Additive Erhöhung/Senkung der Senderate



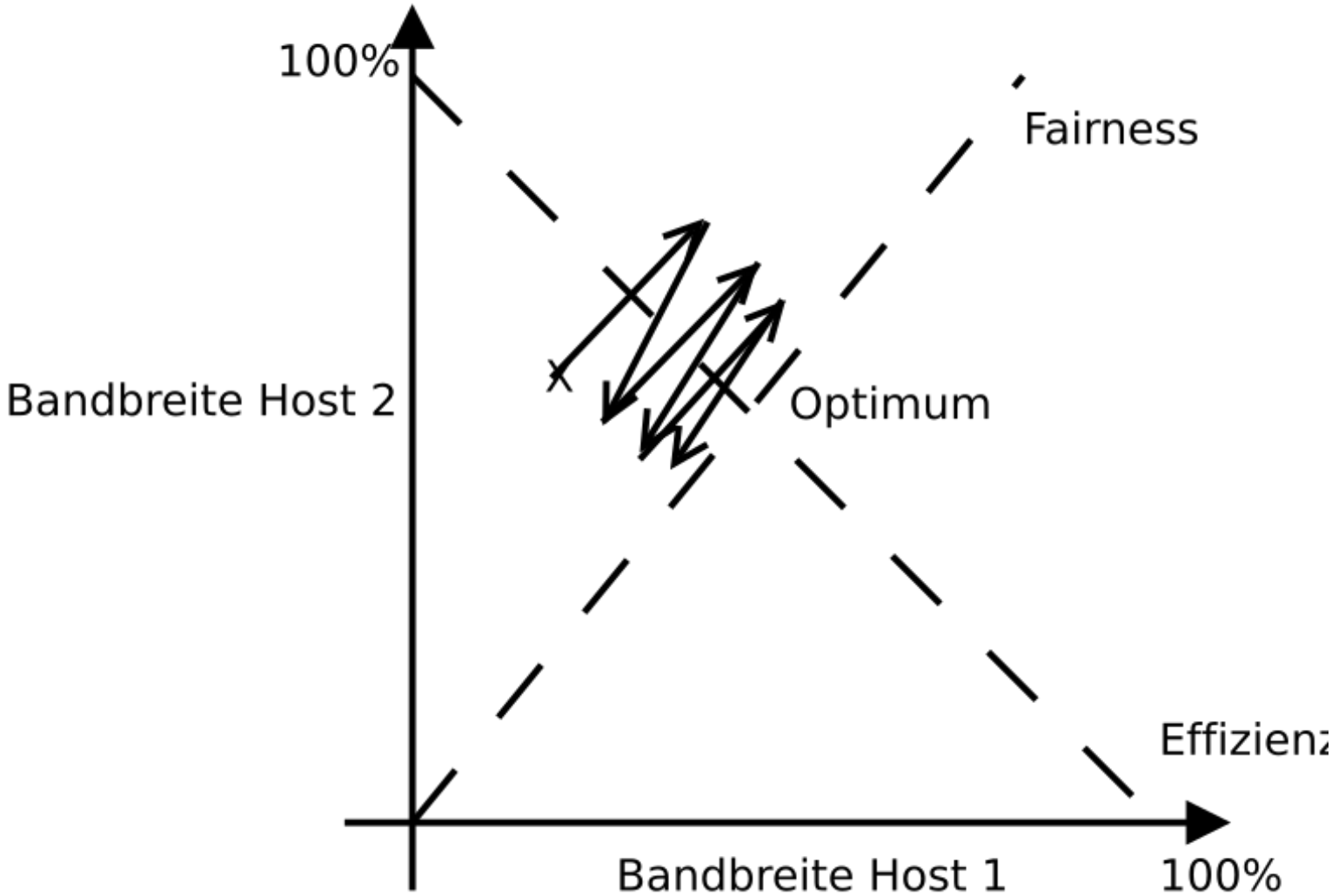
- Gerade durch Punkt mit Steigung 1
- Arbeitspunkt oszilliert um Effizienz
- –> keine Konvergenz

Multiplikative Erhöhung/Senkung



- Gerade durch Arbeitspunkt und Ursprung
- Arbeitspunkt oszilliert um Effizienz
- –> keine Konvergenz

AIMD (additive increase multiplicative decrease)



Zusammenfassung

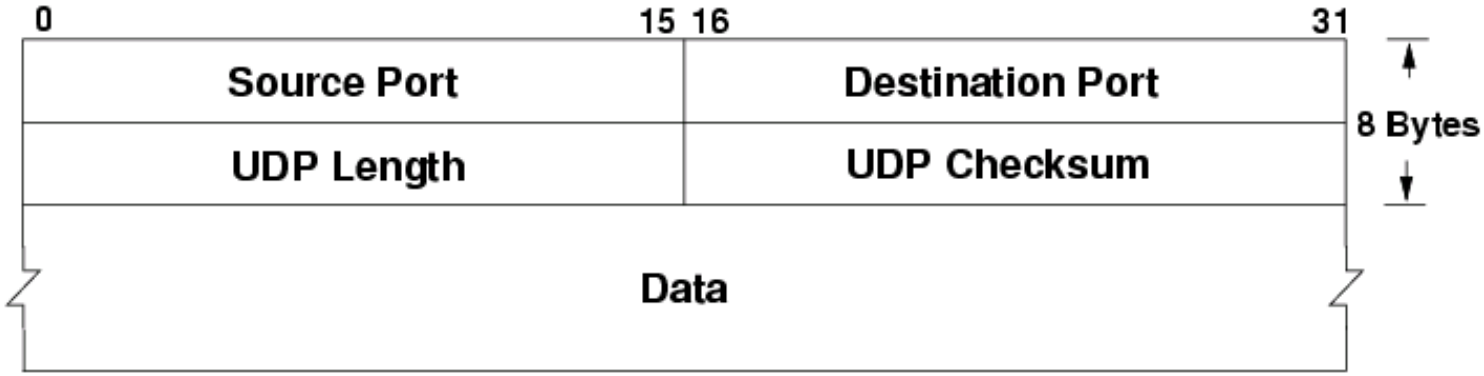
Außerdem: TCP and The Lower Bound of Web Performance
https://www.youtube.com/watch?v=C8orjQLacTo

- 3-way Aufbau
- 3-way Abbau
- AIMD (additive increase multiplicative decrease): Optimiert Bandbreitennutzung

UDP (User Datagram Protocol)

- verbindungsloses Protokoll
- fire and forget
- Vorteile ggü IP:
  - Ports
  - Checksumme über Daten (Ende zu Ende)
- verwendet für RPC, Realtime Protokolle (Media, Games)

UDP Header



- Source und Destination Ports
  - Bsp: 53 (DNS)
- Length: beinhaltet Header und Data
  - min: 8 Bytes
  - max: 65.515 Bytes (begrenzt durch IP)
- Checksum:
  - optional

UDP - nicht enthalten

- Flusskontrolle
- Überlastkontrolle
- Zuverlässigkeit
- –> muss in Anwendungsschicht implementiert werden, falls gewünscht

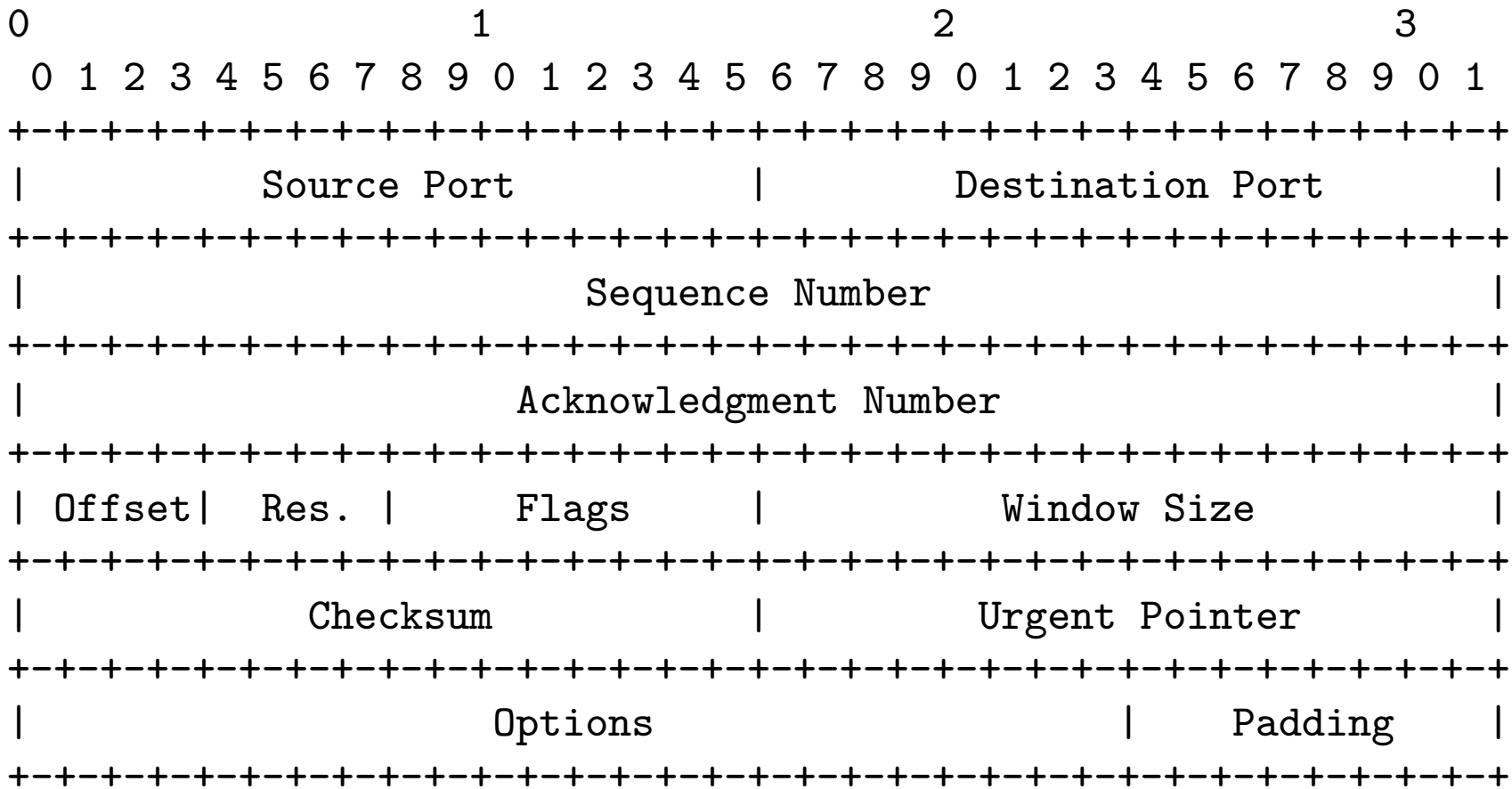
Zusammenfassung

- UDP ist minimal: Port + Checksumme (+ Länge)
- Addressiert Prozesse (Port) statt Rechner (IP)

TCP (Transmission Control Protocol)

- zuverlässiger, Ende zu Ende Byte Stream über unzuverlässiges Netzwerk
- full-duplex, point-to-point
  - kein multi- oder broadcasting
- Aufteilung des Bytestreams in Pakete ist transparent
  - Daten + Header müssen in 65.515 Byte IP Paket passen
  - jedes Segment muss in kleinste MTU (Maximum Transmission Unit)
- Segmente ohne Daten sind legal

TCP Header


0 1 2 3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
Source Port
Destination Port
Sequence Number
Acknowledgment Number
Offset
Res.
Flags
Window Size
Checksum
Urgent Pointer
Options
Padding

Source / Destination Port

- Bsp: 80 (HTTP)
- Port + IP definieren einen Endpunkt
- 2 Endpunkte definieren eine Verbindung

Sequence / Acknowledgment Number

- 32 Bit um jedes Byte Daten zu nummerieren
- ACK Number spezifiziert das nächste erwartete Byte
- cumulative acknowledgement: 23 bedeutet, dass Byte 0-22 erhalten wurde
- Offset: wieviele 32 Bit Wörter sind im TCP header?
  - wo beginnen die Daten (Options variable Länge)?
- Res: unbenutzte reservierte Bits für Erweiterungen
  - ursprünglich 6, jetzt nur noch 4

Auswahl Flags

- ACK: 1 wenn ACK Number valide ist (Segment enthält ACK)
- PSH: Daten sollen direkt an Anwendung gereicht werden (kein Buffering)
- RST: Connection Reset
- SYN: Verbindungsaufbau
  - Connection Request (SYN=1, ACK=0)
  - Connection Accepted (SYN=1, ACK=1)
- FIN: Verbindungsabbau

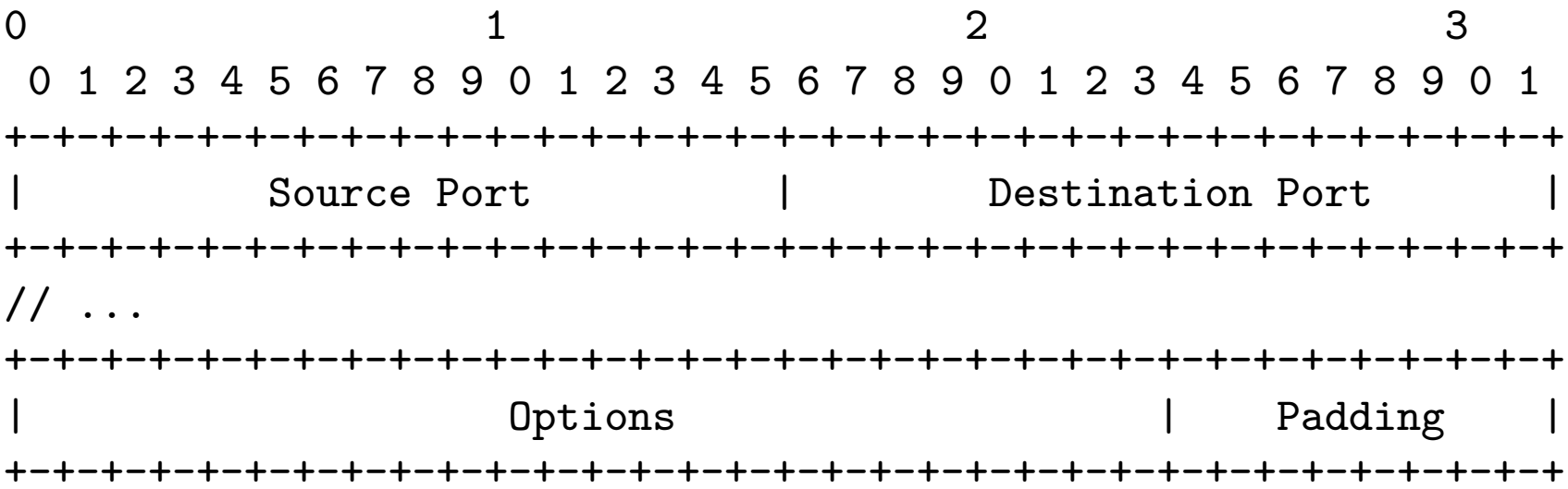
Window Size

- wieviele Bytes dürfen ab ACK Number gesendet werden?
- 0 ist legaler Wert
  - Weiteres Senden möglich nach Segment mit gleicher ACK No und Window Size > 0

Weitere

- Checksum
  - nicht optional
  - sicher Segmente Ende zu Ende
- Urgent Pointer
  - gibt Position von wichtigen Daten im Stream an
  - Bsp: CTRL + c

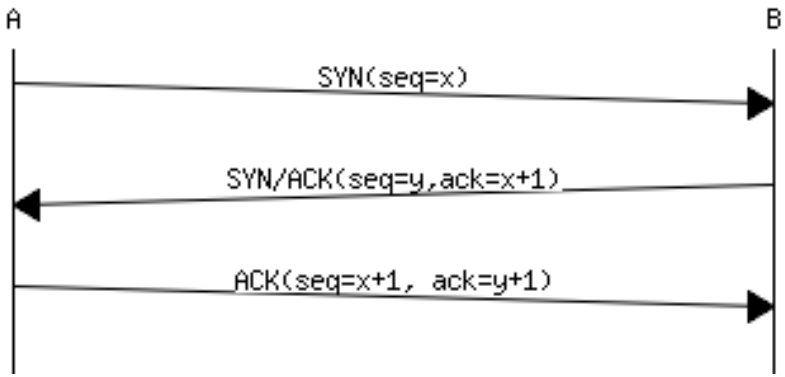
Optionen



- Window Scale: Window Size wird so weit nach links geshiftet
  - erlaubt Windows mit 2^{30} Bytes (sonst nur 64 KB)
- Timestamp: Berechnung von round trip time (RTT)
  - Erweiterung Sequenznummer für schnelle Verbindungen

Variable Länge, Vielfaches von 32 Bit lang! $\Rightarrow$ Padding.

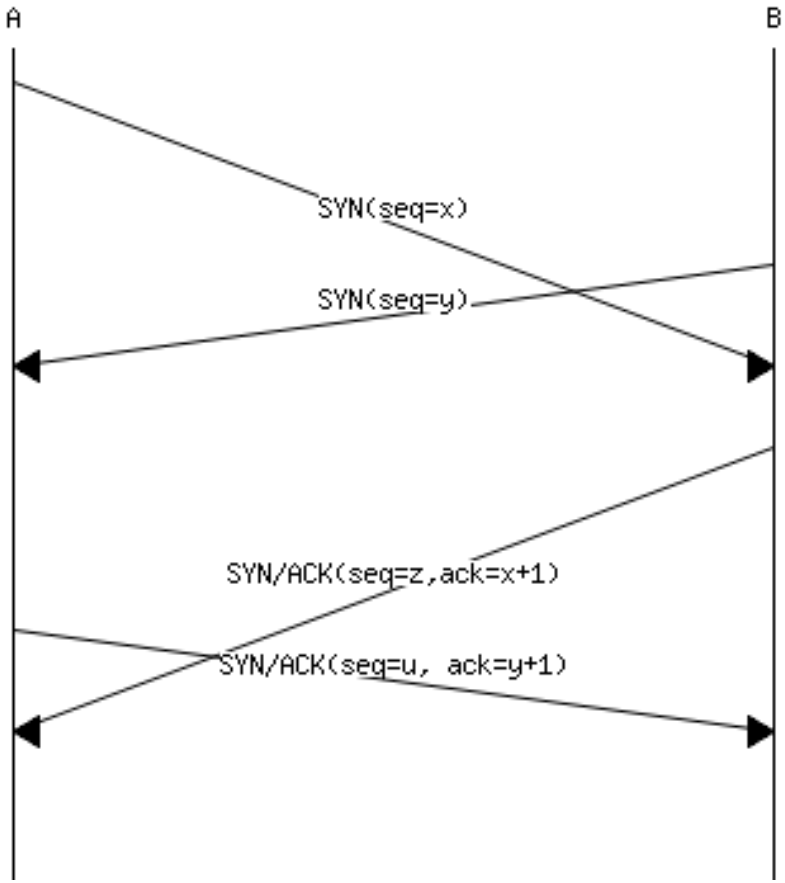
TCP Verbindungsaufbau



- initiale seq wurde früher mit externer Uhr generiert
  - passiver Teilnehmer muss seq von `SYN` Segmenten speichern
  - DOS (Denial of Service) möglich durch senden vieler `SYN` Segmente
  - Lösung SYN Cookies : Generierung initiale seq bei Bedarf

TCP Verbindungsaufbau

TCP doppelte Verbindung?

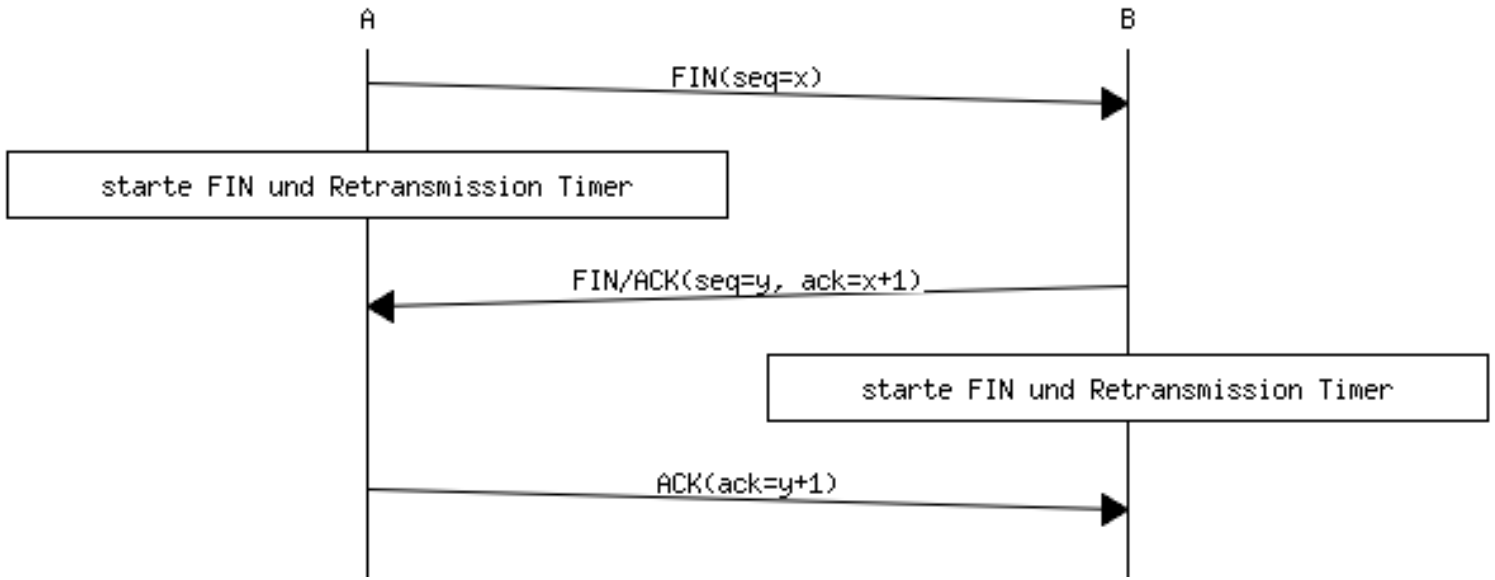


Wieviele Verbindungen werden geöffnet?

TCP Verbindungsabbau

- verwendet symmetrisches Verfahren
- Um eine Seite zu schließen:
  - Sende `FIN`
  - sobald `ACK` für `FIN` empfangen wird, trenne Verbindung
- `FIN` startet Timer (2 max Paketlebenszeiten)
  - Timeout schließt Verbindung
- `FIN` Retransmission wird durch normalen Retransmission Timer sichergestellt

TCP Verbindungsabbau, Diagramm

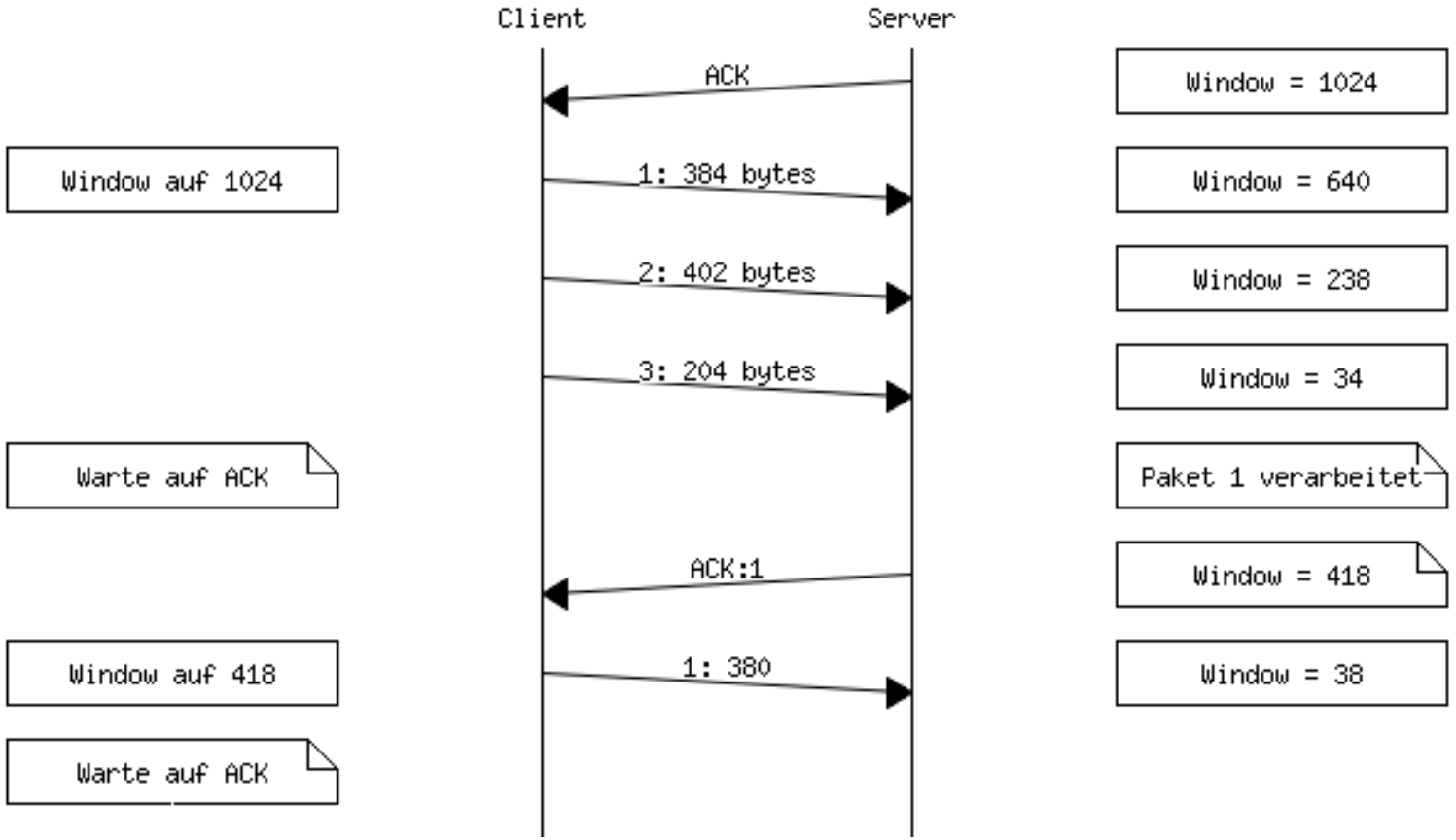


TCP Sliding Window

- Window = 0 ist legaler Wert
  - bedeutet: keine weiteren Daten senden
  - außer: urgent data
  - oder: Window Probe
- Sender können Daten buffern vor dem Senden
- Empfänger müssen `ACK` nicht sofort senden

TCP Sliding Window

–



–

Problem: Telnet Verbindung

- jeder Tastendruck erzeugt 21 Byte TCP Segment
- Empfänger sendet 20 Bytes `ACK`
- Empfänger sendet 21 Bytes für Bildschirmupdate

Lösung: Delayed Acknowledgments

- `ACK` darf bis zu 500 ms verzögert werden
- erlaubt `ACK` von mehreren Segmenten gleichzeitig

Weiteres Problem: Sender sendet immer noch 1 Segment pro Tastendruck

Nagles Algorithmus:

- wenn viele kleine Segmente gesendet werden sollen
- sende erstes Segment
- buffere alle weiteren Segmente
- sende alle gepufferten Segmente sobald `ACK` eintrifft
- kann mit der `TCP_NODELAY` Option ausgeschaltet werden
  - besser: `TCP_QUICKACK`

Nagle zu Delayed ACK

https://news.ycombinator.com/item?id=10607422

> *Unfortunately by the time I found about delayed ACKs, I had changed jobs, was out of networking, and doing a product for Autodesk on non-networked PCs.*

Silly Window Syndrome

- Anwendung auf Empfängerseite liest immer nur 1 Byte
- wenn Window voll ist, zwingt dies TCP nur noch 1 Byte große Segmente zu verschicken

Lösung:

- Empfänger sendet nur window update sobald mindestens die maximal Segmentgröße Platz ist

TCP Timer

- RTO (Retransmission Timeout): muss ein Segment erneut gesendet werden?
- Persistence Timer: verhindert Deadlock
  - Empfänger: Window Size=0
  - Window Update geht verloren
  - Persistence Timeout triggert Window Probe
- Keepalive Timer: sende Segmente, um Verbindung offen zu halten
- FIN Timer: beende Verbindung nach Timeout

TCP - Retransmission Timer

Problem:

- zu kurz: viele Retransmissions
- zu lang: hohe Latenz bei Packet Loss
- ständige Veränderung durch Überlastkontrolle

Lösung:

- dynamischer Algorithmus
- Berechne Smoothed Round Trip Time (SRTT)
  - $\text{SRTT}_{i+1} = \alpha\text{SRTT}_i + (1-\alpha)\text{RTT}_{i+1}$
  - $\alpha = \frac{7}{8}$ Smoothing Factor
- Früher: RTO = 2*SRTT_{i+1}

Varianz!

Problem: betrachtet nicht die Varianz:
wenige Pakete stark verzögert.

Lösung:

- berechne Round Trip Time Variation (RTTVAR)
  - $\text{RTTVAR}_{i+1} = \beta \text{RTTVAR}_i + (1-\beta)\,|\text{SRTT}_i - \text{RTT}_{i+1}|$
  - $\beta = \frac{3}{4}$
- $\text{RTO}_{i+1} = \text{SRTT}_{i+1} + 4\text{RTTVAR}_{i+1}$
- keine echte Varianz aber gute Näherung
- SRTT_{i+1} berechnet wie vorher.
- RTO mindestens 1 Sekunde

Initialisierung:

- SRTT = RTT
- RTTVAR = RTT / 2
- RTO = SRTT + 4 * RTTVAR
- erfolgt bei erster RTT Messung

Übung RTO

Berechne RTO in T_{1}. RTT_{0} = 50 ms, RTT_{1} = 30 ms.

Lösung RTO

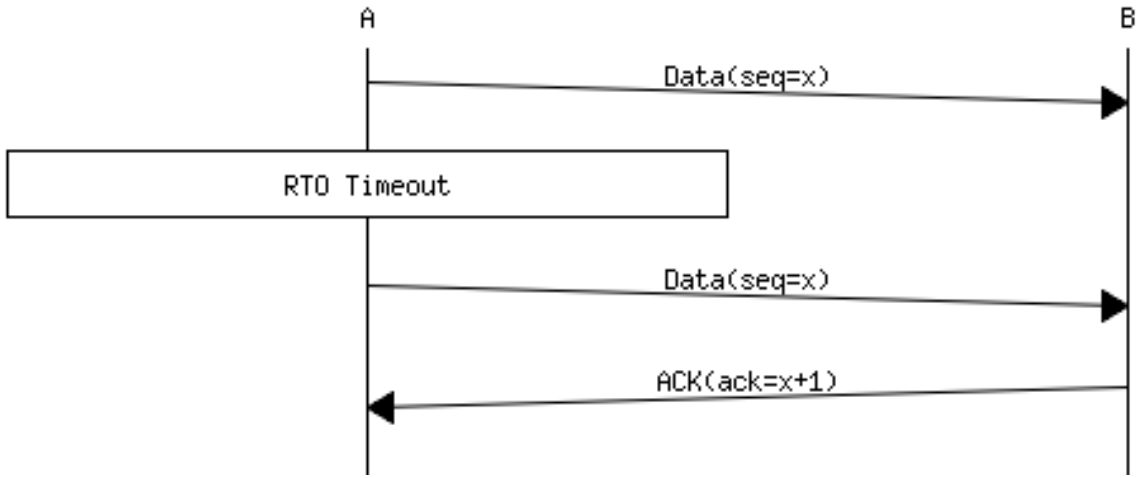
$$SRTT_1 = \alpha 50 + (1 - \alpha)30 = 47,5$$

$$RTTVAR_1 = \beta 25 + (1 - \beta)\,|50 - 30| = 23,75$$

$$RTO_1 = 47,5 + 4 * 23,75 = 142,5$$

- $\beta = \frac{3}{4}$
- $\alpha = \frac{7}{8}$

Problem Messung RTT



Auf welches Segment bezieht sich `ACK`?

Lösung: Karn Algorithmus

- kein Update der Werte bei wiederholter Übermittlung
- jeder Timeout verdoppelt RTO

TCP Überlastkontrolle

- TCP verwaltet congestion window
  - wieviele Bytes können sich auf einmal im Netzwerk befinden
- Größe wird mit AIMD angepasst
- zusätzlich zu window der Flusskontolle
- TCP hört auf zu senden sobald eins der Fenster voll ist
- Packet Loss als binäres (impräzises, implizites) Signal

ACK Clock

- mehrere kleine Pakete werden gesendet
- werden in Router vor Flaschenhals gepuffert
- Empfänger bestätigt einzelne Pakete
- `ACKs` treffen bei Sender ein
- Rate der `ACKs` entspricht optimaler Senderate

AIMD - Initialisierung

- Annahme: window wird mit 1 Paketgröße initialisiert
- additive Erhöhung recht langsam
- sollte das window größer initialisiert werden?
  - kann zu Problemen bei langsamen oder kleinen Leitungen führen

Slow Start nach Jacobson

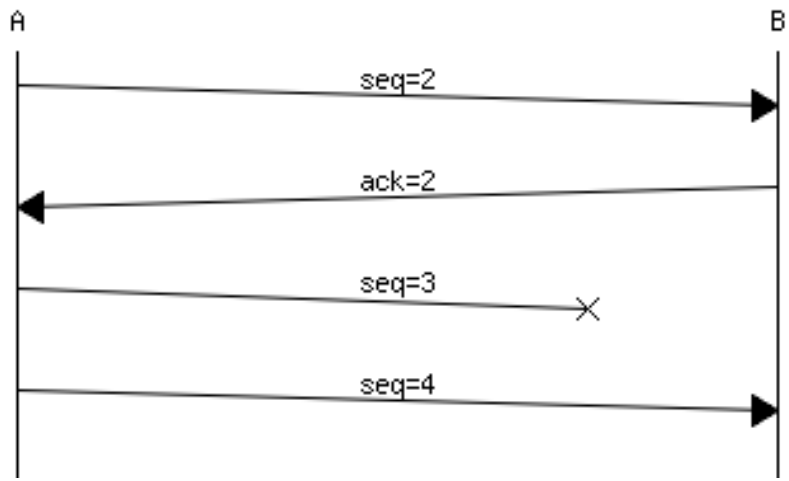
- starte mit kleinem congestion window
- pro Segment, das vor RTO bestätigt wird: vergrößere window um 1 Segment
- exponentielles Wachstum führt zu Überlast
- verwalte zusätzlich einen Schwellenwert (Slow Start Threshold)
- bei Packet Loss: setze Threshold auf Hälfte des congestion windows
- bei Überschreitung der Threshold: verwende additive increase
  - 1 Segment pro RTT (auch hier mit ACK Clock)

Fast Retransmission

Problem: verlorene Pakete erzeugen hohe Latenz (Warten auf RTO)

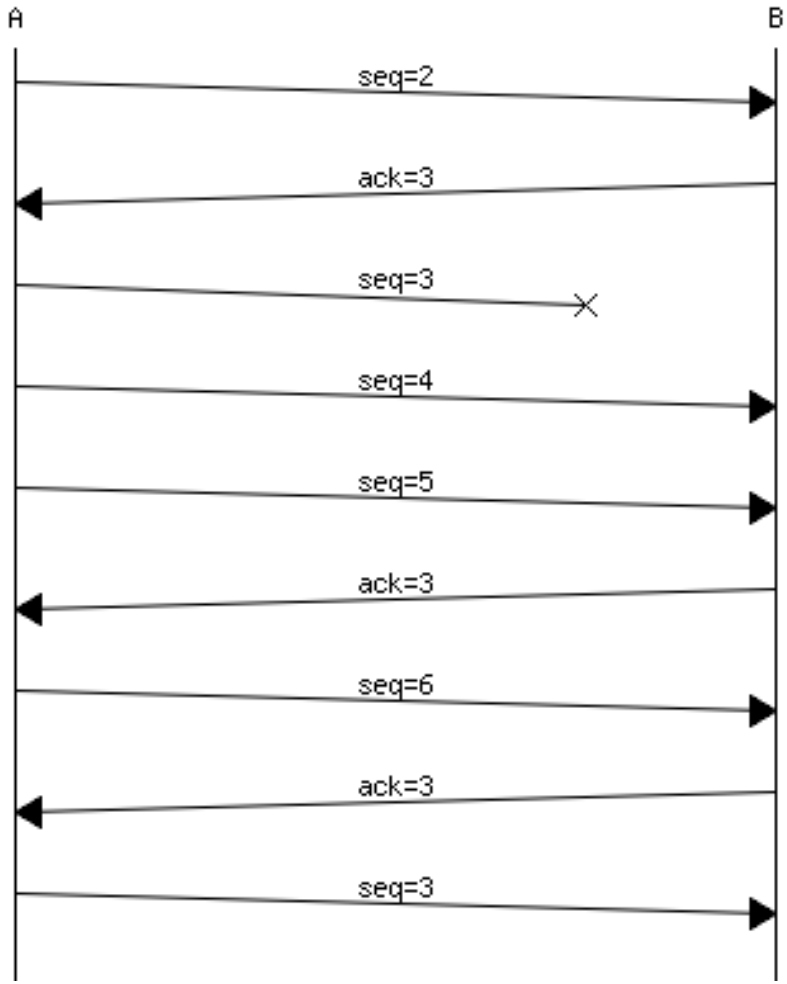
Lösung: duplicate `ACK`

- 3 duplicate `ACK` signalisieren Packet Loss



AIMD - Initialisierung

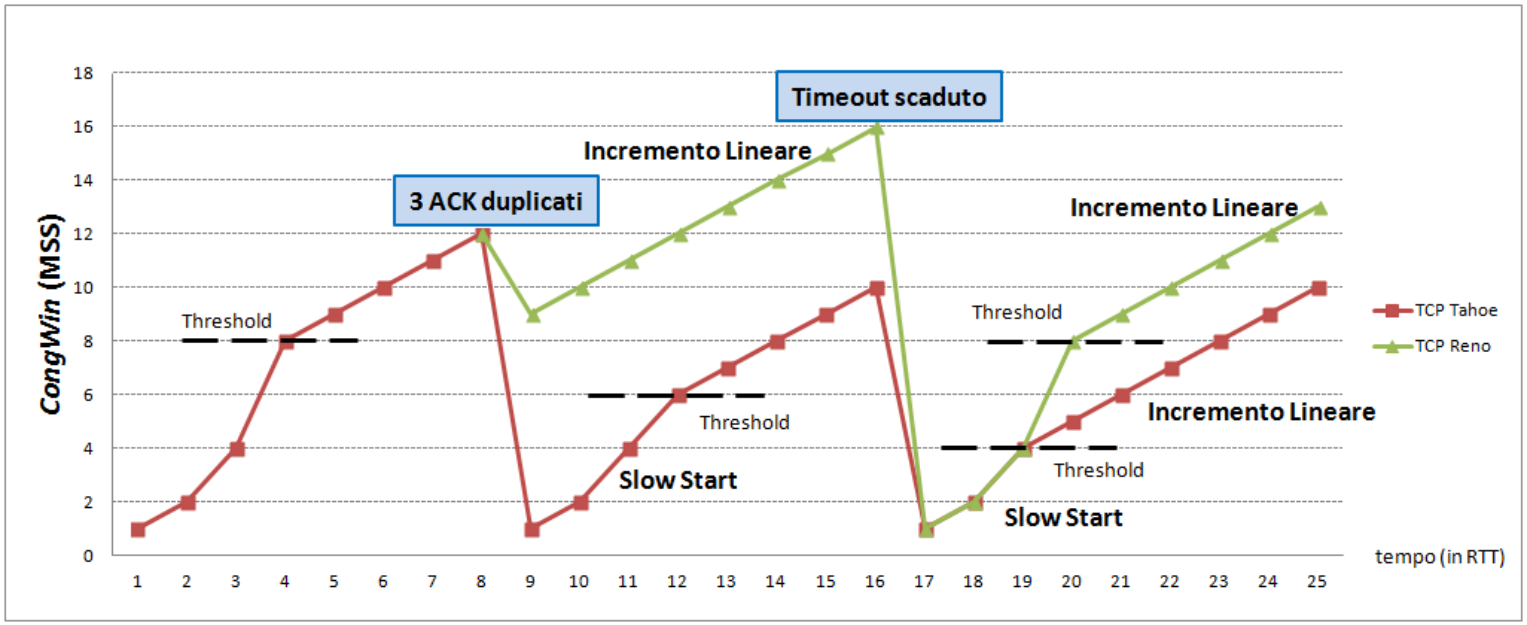
–



Fast Recovery nach Jacobson (Reno)

- bei 3 duplicate `ACKs`
  - verkleinere window auf Hälfte (anstatt 1 Segmentgröße)
  - setze Threshold auf neue window Größe

Effekt von Fast Recovery



- TCP Tahoe: slow start + AIMD + fast retransmit
- TCP Reno: Tahoe + fast recovery

Selective Acknowledgment (SACK)

- listet in Options bis zu 3 Intervalle auf, die bestätigt werden
- wird beim Verbindungsaufbau ausgehandelt
- weit verbreitet
- verbessert Retransmission

Zusammenfassung

- 3-way Aufbau
- 3-way Abbau
- AIMD-Optimierungen:
  - Slow Start ist schnell
  - Fast Retransmission
  - Fast Recovery

QUIC: Wieso?

- TCP: Head of line blocking
- TCP: Window 0 betrifft alle streams
- TCP: 6 Pakete für Verschlüsselung: 3 x TLS + 3 x TCP

Details:
https://www.potaroo.net/ispcol/2022-11/quicvtcp.html

Als kleiner Player passt ihr eure Software den Protokollen an. Als Big Player passt ihr die Protokolle für eure Software an.

QUIC: Verschlüsselte Streams über UDP

- Verschlüsselt: Garantiert Opaque Daten (weniger Abstraktionsverletzungen).
- Mehrere verlässliche TCP-ähnliche Streams über UDP
- Auch unzuverlässige Paketübermittlung
- Wiederaufnahme nach IP-Wechsel (mit Challenge mit voriger Verschlüsselung)
- Grundlage für HTTP/3

QUIC: Setup-Pakete reduzieren

- Initialdaten für Setup + Verschlüsselung gleichzeitig $\Rightarrow$ 3 Pakete statt 6
- Bei gleicher Session in Paket 0 schon Daten schicken
- Erstes Paket auf max MTU (Paketgröße) padden, um Pfade mit zu kleinen Paketgrößen zu vermeiden.

Zusammenfassung

- Kanal Zwischen Prozessen
- 3-way Aufbau
- 3-way Abbau
- UDP ist minimal: Port und Länge zu IP dazu
- AIMD-Optimierungen:
  - Slow Start ist schnell
  - Fast Retransmission
  - Fast Recovery

Sie kennen nun die Grundlagen der Netztechnik

Ich hoffe, ich konnte Ihnen auch Verständnis der wichtigsten Aspekte vermitteln.

Ab jetzt wird es Zeit zu handeln.

Sie sind bereit mit einer Hand an eigenem Code und einer in Wikipedia die Tiefen der Kommunikations- und Netztechnik zu erkunden.

(zumindest, wenn man jemals wirklich bereit dafür sein kann)

Viel Erfolg!

Nächstes Mal geht es weiter mit Anwendungen.

Hamming-Beispiele

- Der vorherige 11,7: https://hg.sr.ht/~arnebab/wisp/browse/examples/hamming.w?rev=ad2b1867648a
- Generisch, ineffizient, mit Bugs: https://hg.sr.ht/~arnebab/wisp/browse/examples/hamming-file.w?rev=ad2b1867648a
- 7,4 Hamming Code-Golf: https://codegolf.stackexchange.com/questions/45684/correct-errors-using-hamming7-4

Verweise I

Bilder:

Abkürzungen

ACK	Acknowledgment
AIMD	Additive Increase Multiplicative Decrease
CR	CONNECTION REQUEST
DOS	Denial of Service
DR	DISCONNECTION REQUEST
IP	Internet Protocol
ISP	Internet Service Provider
MTU	Maximum Transmission Unit
OS	Operating System
RPC	Remote Procedure Call
RTO	Retransmission Timeout
RTTVAR	Rount Trip Time Variation
SACK	Selective Acknowledgment

Übungsblatt Transportschicht

Besprechung in der nächsten Vorlesung.

Bei Multiple Choice Aufgaben reicht eine Lösung nach folgendem Muster:

Übungsblatt Transportschicht

Beispiel Aufgabe Multiple Choice

Kreuze die korrekten Aussagen an:

1. ☐ Die letzte Vorlesung war viel zu schnell
2. ☐ Sriracha passt zu allem
3. ☐ Tabs sind besser geeignet für die Einrückung von Quellcode

Übungsblatt Transportschicht

Beispiel Lösung Multiple Choice

1, 2

Übungsblatt Transportschicht

Aufgabe 1

Kreuze die korrekten Aussagen an:

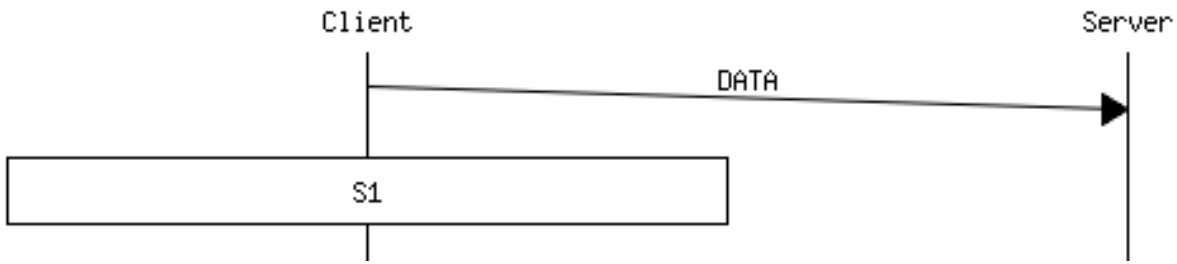
1. ☐ UDP Segmente kommen immer in Absendereihenfolge beim Empfänger an.
2. ☐ UDP Segmente können verloren gehen.
3. ☐ Erfolgreich empfangene UDP Segmente können beschädigt sein.
4. ☐ Segmente können vom Netzwerklayer dupliziert werden.
5. ☐ Die function `accept()` wird in der Regel client-seitig aufgerufen.
6. ☐ Ein Telefongespräch wird symmetrisch getrennt.

Übungsblatt Transportschicht

Aufgabe 2

Zeichne das Sequenzdiagramm für folgende Client- und Serverkonfiguration. Kommt es zu duplizierten/verlorenen Daten oder ist alles in Ordnung?

- Client: Always retransmit
- Server: First write then `ACK`
- Eventreihenfolge: Write, Crash, Ack



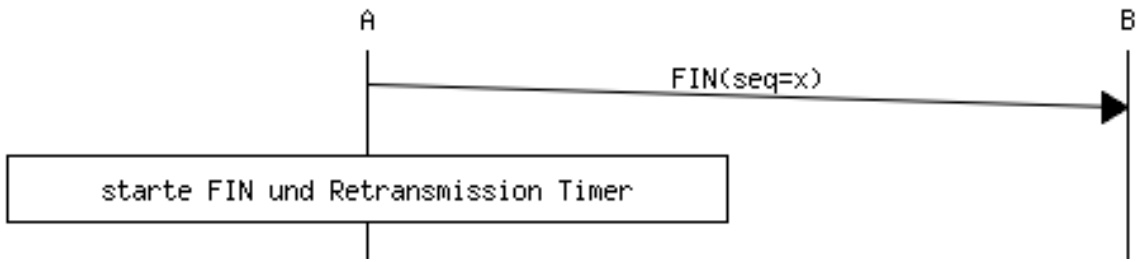
Übungsblatt Transportschicht

Aufgabe 3

Ekläre die 3 Kriterien: Effizienz, Fairness und Konvergenz.

Übungsblatt Transportschicht

Aufgabe 4



Vervollständige das Sequenzdiagramm für folgende 3 Fälle bis zur Trennung der Verbindung:

1. `FIN` Timer wird ausgelöst.
2. Retransmission Timer wird ausgelöst.
3. B sendet ein `ACK` Segment mit seq=y und ack=x+1

Übungsblatt Transportschicht

Aufgabe 5

Zeichne ein Diagramm mit Congestion Window Größe (in Segmenten) auf der y-Achse und Transmission Round (von 0 bis 8) auf der x-Achse für folgende Parameter:

- Threshold = 16 Segmente
- Packet Loss in Transmission Round 6
- Verwendung von Slow Start und Fast Recovery