Rechnernetze:
(5) Transportschicht

Prof. Dr. Klaus-Peter Kossakowski

Gliederung der Vorlesung

- **Einführung und Historie des Internets**
- **Schichtenmodell**
- **Netzwerk als Infrastruktur**
- **Layer 7: Anwendungsschicht**
- **Layer 4/7: Socketprogrammierung**
- **Layer 4: Transportschicht**
  - UDP – Verbindungsloser Transport
  - TCP – Verbindungsorientierter Transport
- **Layer 3: Netzwerkschicht**
- **Layer 2: Sicherungsschicht**

Inhalte dieses Kapitels

In diesem Kapitel behandeln wir die beiden gängigen Transportprotokolle UDP und TCP, wobei aufgrund der Komplexität der Schwerpunkt auf TCP liegt.

Die Rolle der Ports für das Multiplexen verschiedener Anwendungen zwischen zwei Rechnern wird erläutert. Die Protokollheader werden mit ihren Daten besprochen.

Besondere Funktionen von TCP wie der TCP-Handshake, die Flusskontrolle und Methoden zur Staukontrolle werden auf Konzeptebene erklärt und vertieft.

Ziele dieses Kapitels

Sie können das Multiplexen von Anwendungen über Ports bei UDP- und TCP-Anwendungen erklären.

Sie kennen die Protokollheader und insbesondere die Verwendung der TCP-Flags.

Sie können den protokollkonformen Ablauf einer TCP-Verbindung (Aufbau, Transfer, Abbau) und hierbei die Verwendung von Flags und Sequenznummern erläutern.

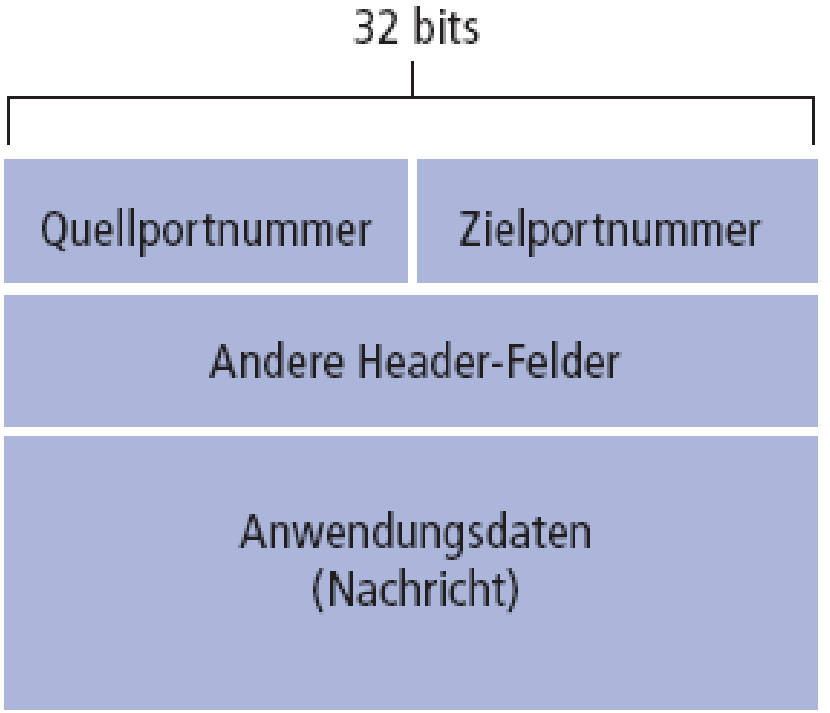
Sie können Flusskontrolle sowie Staukontroll-mechanismen beim Pipelining erklären.

Zentrale Portvergabe

Zur Kommunikation miteinander müssen sich zwei Rechner auf passende Portnummern einigen. Hierfür gibt es eine zentrale Vergabe bekannter Portnummern (sogenannter „well known ports“) zumindest für Server-Prozesse

- festgelegt durch die IANA für < 1024
- Ports < 1024 sind oft auch auf Betriebs-systemebene privilegiert, d.h. durch das OS geschützt
  - siehe z.B. in der Datei /etc/services

Zuordnung per Portangabe



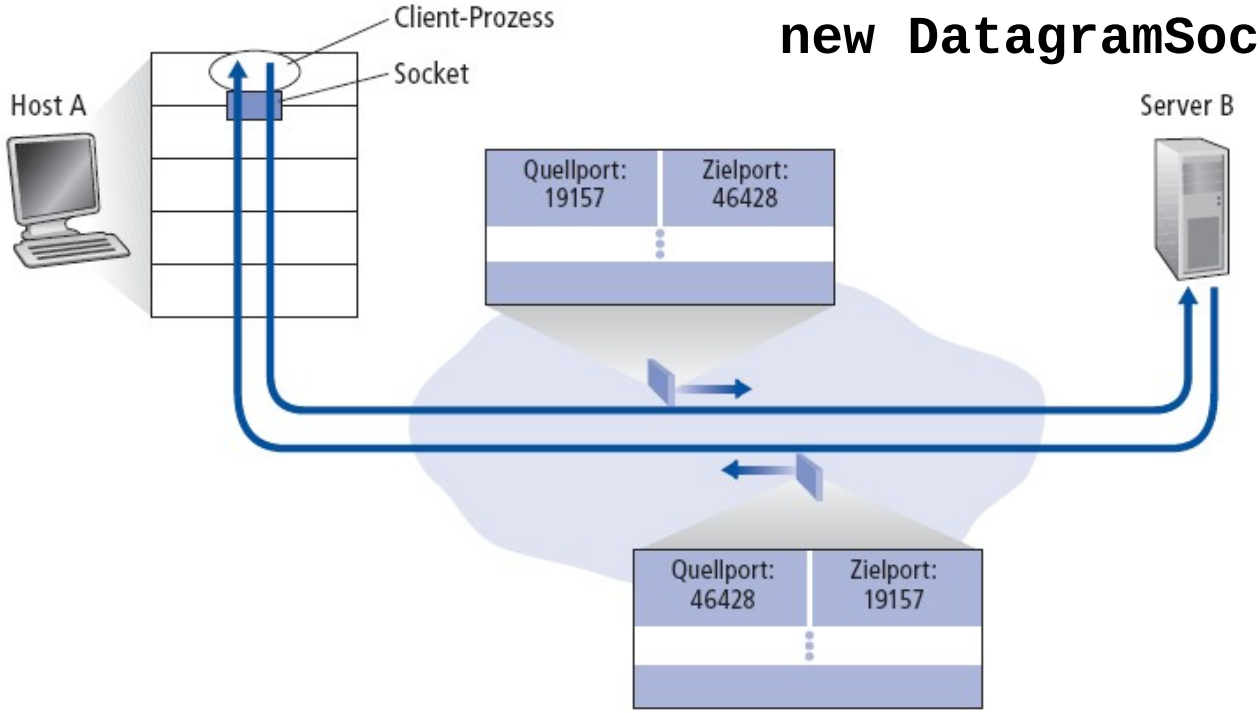
Portangaben sowohl für Quelle (Sender) als auch für das Ziel (Empfänger) sind die wichtigsten Felder des Headers!

Header-Felder unterscheiden sich bei Transportprotokollen

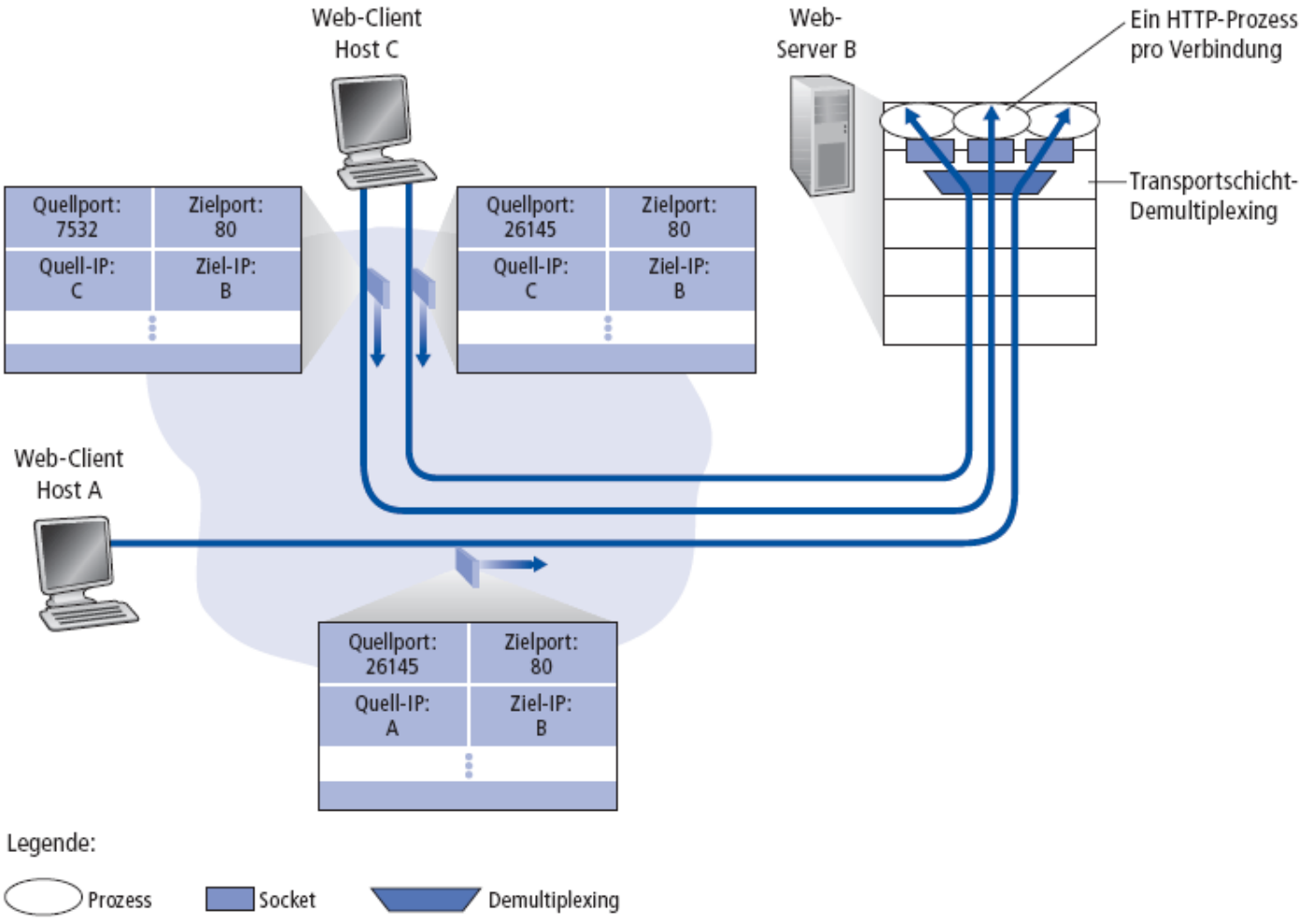
Port == Anwendungsprozess

```
DatagramSocket
host_A_Socket =
new DatagramSocket(19157);
```

```
DatagramSocket
server_B_Socket =
new DatagramSocket(46428);
```



De-Multiplexing beim Web-Server



Dynamische Portvergaben

- **Clients (= Hosts) fordern vom Betriebssystem einen „freien“ Port**
  - Portnummern sollten > 1024 sein
  - „bestimmter“ Portnummer nicht wichtig
- **Dynamische Zuordnung kann von Anwendungs- bzw. Dienstprogrammen selbst implementiert werden**
  - z.B. Portmapper auf *NIX-Systemen ist ein Verbindungsmanager, auf dem z.B. entfernte Clients nach dem aktuellen Port eines Services „fragen“ können

Warum Transportprotokolle?

- **IP adressiert nur Zielrechner, nicht einzelne Programme oder laufende Prozesse**
- **Die Verteilung auf Anwendungsprogramme (Multiplexing) und die Abbildung der Qualitätsansprüche muss oberhalb der Netzwerkschicht geregelt werden**
- **Zur Adressierung gibt es bei den beiden Transport-Protokollen UDP und TCP jeweils 2^{16} Ports**
  - Vergleichbar mit Postfächern (= Ports) in einem Mehrfamilienhaus (= Zielrechner)

Die Welt liebt Pakete!

Gliederung der Vorlesung

- **Einführung und Historie des Internets**
- **Schichtenmodell**
- **Netzwerk als Infrastruktur**
- **Layer 7: Anwendungsschicht**
- **Layer 5: Sitzungsschicht**
- **Layer 4: Transportschicht**
  - UDP – Verbindungsloser Transport
  - TCP – Verbindungsorientierter Transport
- **Layer 3: Netzwerkschicht**
- **Layer 2: Sicherungsschicht**

Pakete – schlicht und schnell!

TCP – Dienste:

- Zuverlässig
- Datenstrom
- Reihenfolge erhaltend
- Flusskontrolle durch Empfänger
- Staukontrolle
- Nicht geboten:
  - Garantien über Verzögerung oder Kapazität

UDP – Dienste:

- Unzuverlässig
- einzelne Pakete
- geringer Overhead
- Nicht geboten:
  - Verb.-aufbau
  - Flusskontrolle
  - Staukontrolle
  - Garantien über Verzögerung und Kapazität

UDP:
User Datagram Protocol [RFC 768]

- UDP ist ein Protokoll, das den Anwendungen eine Prozedur zur Verfügung stellt, um mit minimalen Protokoll-mechanismen Daten an andere Hosts zu schicken
- UDP ist ungesichert, d. h. es erfolgt keine Quittierung der Daten. Eine erneute automatisierte Übertragung von fehlerhaften Daten findet nicht statt
- Multiplexen von Verbindungen erfolgt mit Hilfe des Port-Mechanismus

UDP:
User Datagram Protocol (2)

- **Fehlererkennung der empfangenen Daten beruht allein auf einfachen Prüfsummen**
  - Fehlerhafte Daten werden verworfen
- **UDP wird nicht durch einen Zustandsautomaten beschrieben**
- **Übertragungen werden nicht durch die Host-to-Host-Schicht zeitüberwacht**

UDP:
User Datagram Protocol (3)

- **UDP als „nacktes“ Transport Protokoll stellt nur einen „best effort“ Dienst bereit, daher können UDP-Datagramme**
  - verloren gehen,
  - in falscher Reihenfolge ankommen
  - oder doppelt ausgeliefert werden
- **Verbindungslos:**
  - kein “Handshake” zwischen UDP-Sender und Empfänger
  - Voneinander unabhängiger Pakettransport

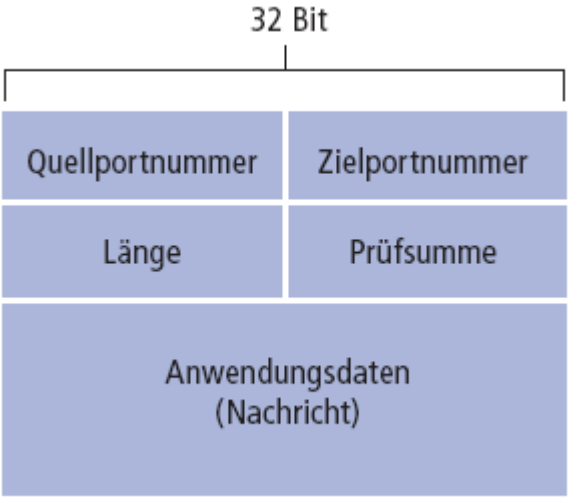
UDP füllt eine Nische

- **Warum gibt es UDP überhaupt?**
  - Keine Verzögerung durch Verbindungsaufbau
  - Kein Verwaltung von Verbindungszu-ständen bei Sender und Empfänger
  - Kleiner UDP-Header führt zu kleineren Paketen
  - Keine Staukontrolle ermöglicht unmittelbare Übertragung von Paketen ohne Warten

UDP füllt eine Nische (2)

- **Wird oft für Multimedia-Anwendungen (Streaming) verwendet**
  - Anwendung ist tolerant gegenüber Paketverlust
  - Datenrate ist allerdings kritisch und bevorzugt deswegen geringen Overhead
- **Ansonsten ist UDP beschränkt auf:**
  - DNS
  - SNMP (Netzwerk-Management)

UDP-Header



source port

- Adresse des UDP-Sender-Prozesses

destination port

- Adresse des UDP-Empfänger Prozesses

length

- Länge des UDP Paketes inklusive aller UDP-Header und UDP-Daten in Bytes (>= 8 Bytes)

UDP-Prüfsumme (checksum)

Ziel: Fehlerentdeckung im übertragenen Segment (z.B. falsche Bits)

- **Absender:**
  - Betrachte den Segmentinhalt als Sequenz von 16-bit Integer-Zahlen
  - Addition der Sequenz der Segmentinhalte zu einer 16-bit Integer-Zahl
  - Einerkomplement, also das invertierte Ergebnis, der Prüfsumme wird in das Datenfeld innerhalb des Headers **nach der Berechnung** eingesetzt

Beispiel für UDP-Prüfsumme

- Wenn Zahlen addiert werden, dann wird ein Übertrag aus der höchsten Stelle zum Resultat an der niedrigsten Stelle addiert!

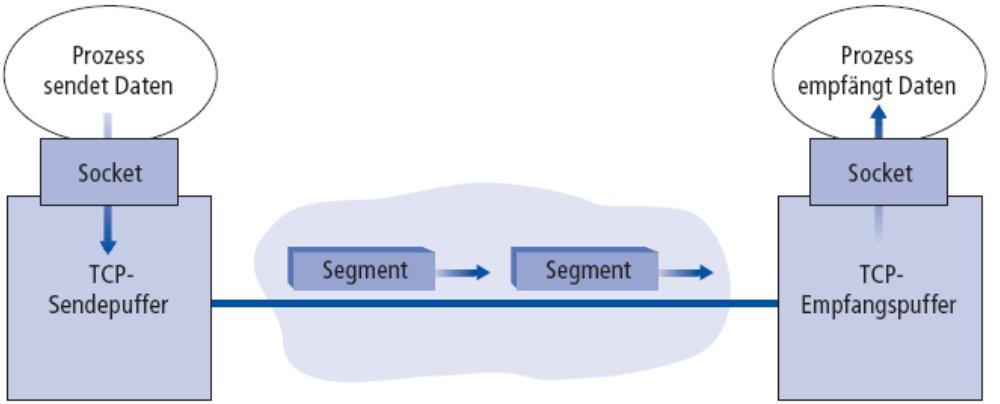
	1	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0
	1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
Übertrag	1	1	0	1	1	1	0	1	1	1	0	1	1	1	0	1	1
Summe	1	0	1	1	1	0	1	1	1	0	1	1	1	1	0	0
Prüfsumme	0	1	0	0	0	1	0	0	0	1	0	0	0	0	1	1

UDP-Prüfsumme (checksum, 2)

Ziel: Fehlerentdeckung im übertragenen Segment (z.B. falsche Bits)

- **Empfänger:**
  - Berechnet Prüfsumme des empfangenen Segments analog zum Sender
  - Überprüft, ob berechnete Summe dem übertragenen Wert des Headers entspricht:
    - NEIN – Fehler gefunden
    - JA – kein Fehler gefunden
- **Könnten da doch noch Fehler sein?**
  - Klar

Die Welt nutzt auch Verbindungen gerne!



Gliederung der Vorlesung

- Einführung und Historie des Internets
- Schichtenmodell
- Netzwerk als Infrastruktur
- Layer 7: Anwendungsschicht
- Layer 5: Sitzungsschicht
- Layer 4: Transportschicht
  - UDP – Verbindungsloser Transport
  - TCP – Verbindungsorientierter Transport
- Layer 3: Netzwerkschicht
- Layer 2: Sicherungsschicht

Qual der Wahl: TCP oder UDP?

TCP – Dienste:

- Zuverlässig
- Datenstrom
- Reihenfolge erhaltend
- Flusskontrolle durch Empfänger
- Staukontrolle
- Nicht geboten:
  - Garantien über Verzögerung oder Kapazität

UDP – Dienste:

- Unzuverlässig
- einzelne Pakete
- geringer Overhead
- Nicht geboten:
  - Verb.-aufbau
  - Flusskontrolle
  - Staukontrolle
  - Garantien über Verzögerung und Kapazität

Eigenschaften von TCP

- **Vor der eigentlichen Datenübertragung wird eine Verbindung zwischen den beiden TCP-Instanzen aufgebaut und initialisiert**
- **Diese Verbindungen sind jeweils unidirektional und nur zusammen bidirektional:**
  - Client → Server
    Server → Client
- **Die zu sendenden Daten werden segmentiert, jedes Segment bekommt einen eigenen TCP-Header**

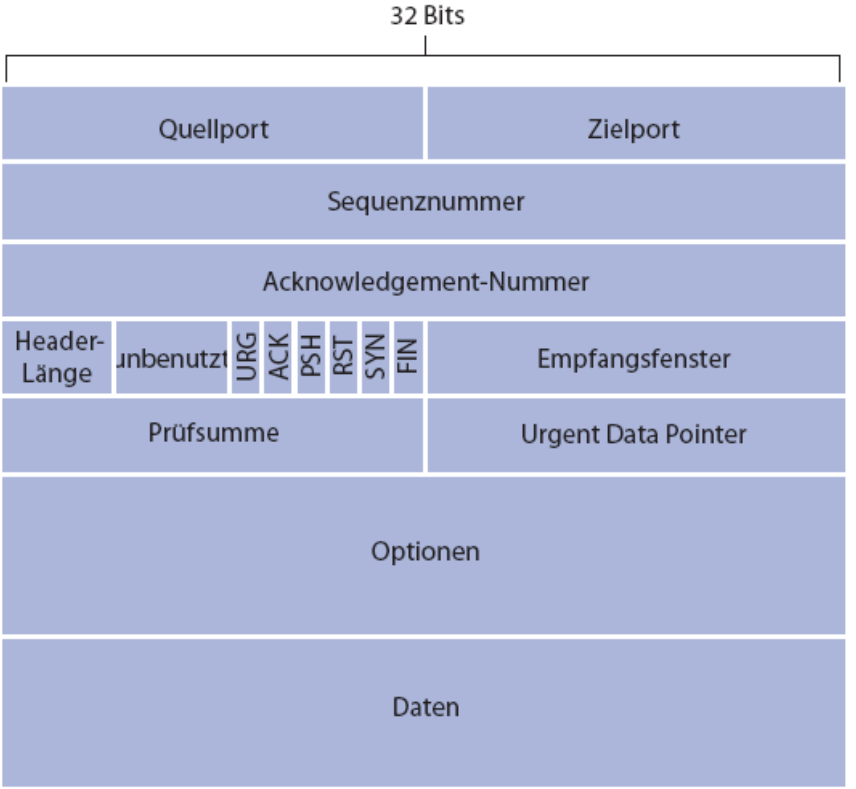
Eigenschaften von TCP (2)

- **Im TCP-Header befinden sich**
  - Portnummern jeweils für Ziel und Quelle
  - Sequenznummern zur nummerierten Datenübertragung ebenfalls für Ziel/Quelle
- **Der Header und der gesamte Inhalt des Segmentes werden analog zu UDP durch eine einfache Prüfsummen gesichert**
- **Jedes fehlerfrei empfangene Paket wird vom Empfänger bestätigt**
  - dazu wird die Sequenznummer des Senders verwendet

Eigenschaften von TCP (3)

- Fehlerbehaftete Pakete werden hingegen verworfen
- TCP sichert die Reihenfolge der Daten durch die Sequenznummern. Zugleich werden hierdurch duplizierte Pakete erkannt, die dann verworfen werden
- Der Sender überwacht die Quittierung von Paketen durch einen Timer
  - Erhält der Sender innerhalb der Laufzeit des Timers keine Quittung für ein gesendetes Paket, sendet er das Paket nochmal

TCP-Header



Länge des TCP Paketes inklusive möglicher TCP-Optionen ist immer wenigstens 20 Bytes!

Optionen sind jeweils Vielfaches von 4 Bytes!

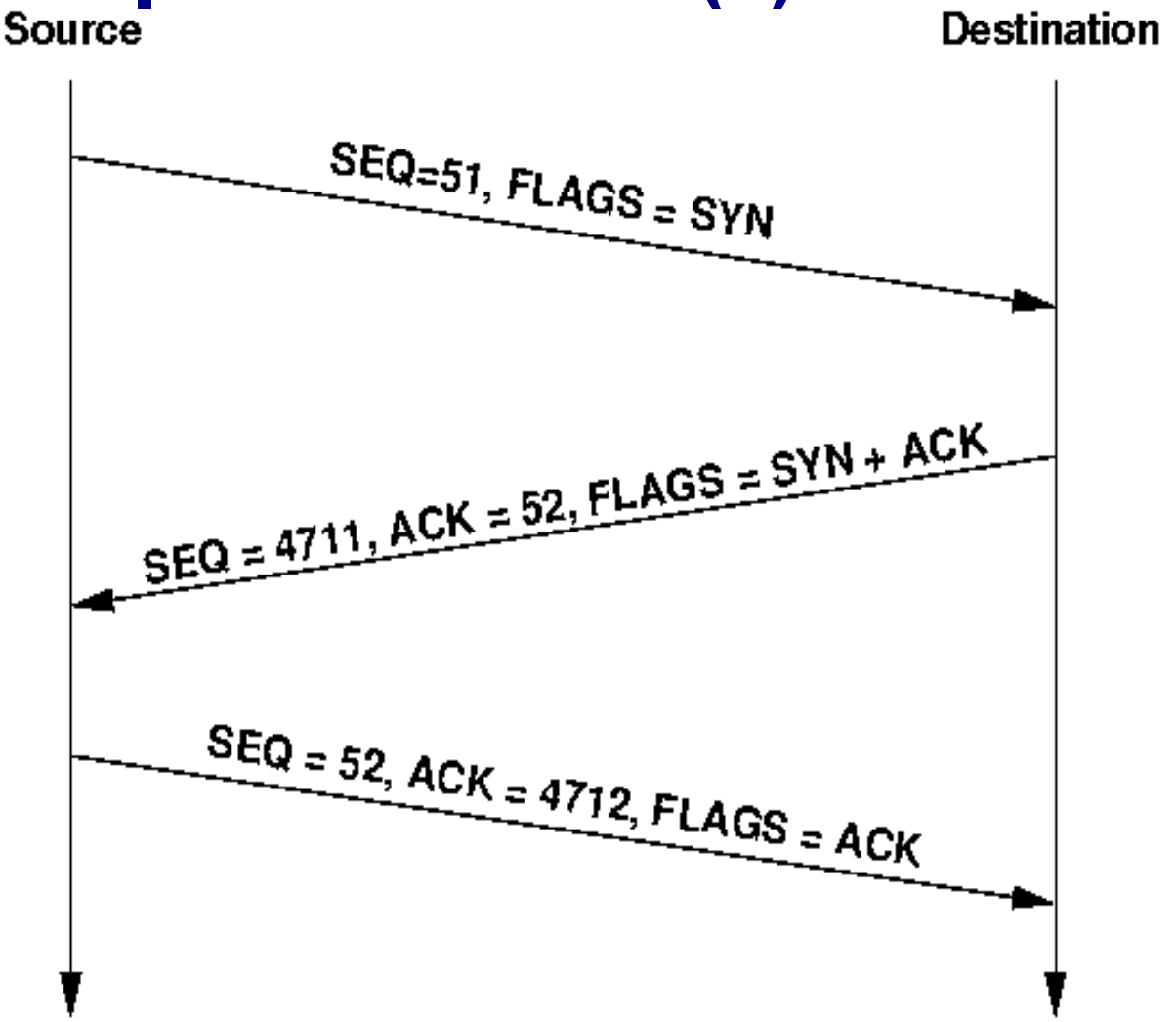
TCP-Header (2)

source port + destination port (je 16 bit)

- lokale Endpunkte einer Verbindung
- zeigt Anwendungszweck der Daten an
- 65536 Portnummern sind möglich
  - Die Portnummern 1 bis 1024 sind reserviert und haben oft vorgegebene Funktionen
- UNIX/LINUX: /etc/services
- Beispiele:

```
Ftp          21/tcp    # File Transfer
http         80/tcp    # World Wide Web HTTP
www          80/tcp    # World Wide Web HTTP
www-http     80/tcp    # World Wide Web HTTP
```

TCP-Sequenznummern (2)



TCP-Header (3)

sequence number (32 bit)

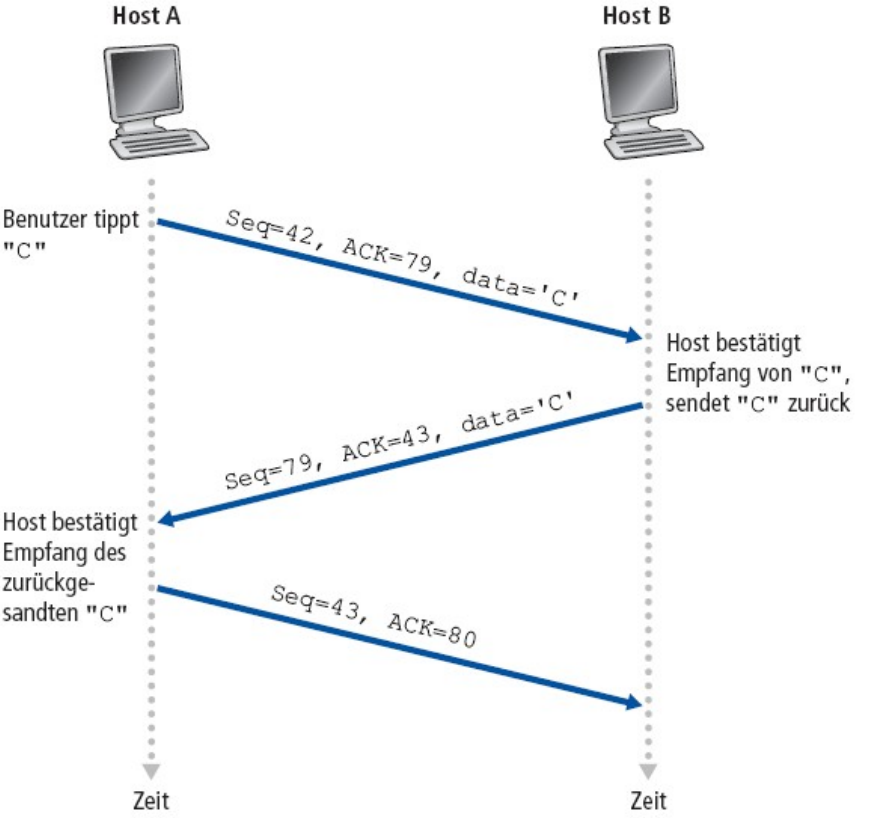
- Die Position der gesendeten Daten im Datenstrom in Bytes
  - nicht die Nummerierung der Segmente

acknowledge number (32 bit)

- Die bisher höchste Sequenznummer plus 1, also das nächste Byte im Datenstrom, das noch nicht bestätigt wurde

TCP-Sequenznummern

- **Daten bilden einen fortlaufenden Datenstrom**
  - Mittels der sequence number und der acknowledge number wird die Einhaltung der Reihenfolge realisiert



TCP-Header (4)

data offset (4 bit)

- Gibt an, wie viele 32 bit Wörter im TCP-Header enthalten sind. Da das Option-Feld eine variable Länge hat, wird so der Anfang der Nutzdaten ermittelt

reserved (4 bit)

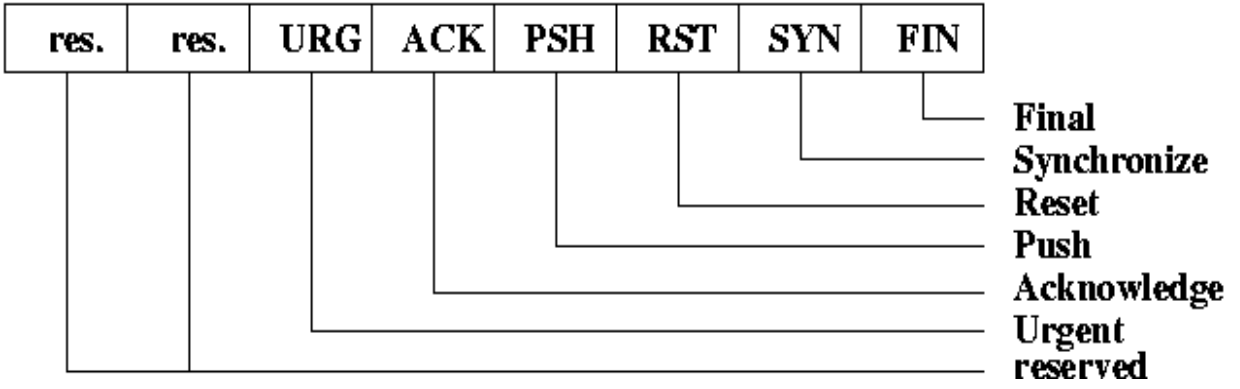
- Ungenutzt ...

flags (8 bit)

- Kodiert bestimmte Signale zwischen Sender und Empfänger und beeinflusst damit die Interpretation der eingehenden Daten

TCP-Header (5) – hier: Flags

- **SYN-Bit**
  - Für Aufbau von Verbindungen
- **ACK-Bit**
  - Signalisiert eine gültige Bestätigungs-nummer



TCP-Header (5) – hier: Flags (2)

- **FIN-Bit**
  - Anzeige, die Verbindung abzubauen
- **URG-Bit**
  - Falls gesetzt, ist der urgent pointer gültig
- **PSH-Bit**
  - Sogenannte PUSH-Daten, die nicht zwischengespeichert werden sollen, sondern sofort an die Anwendung übergeben werden sollen

TCP-Header (5) – hier: Flags (3)

- **RST-Bit**
  - Bei einer Störung kann die Verbindung zurückgesetzt werden

TCP-Header (6)

window size

- Anzahl der Bytes, die der Sender des Pakets für den Empfang zur Verfügung hat → freier Puffer für die Flusskontrolle

checksum

- Obligatorisch über Header- und Payload-Daten, analog UDP-Prüfsumme

urgent pointer

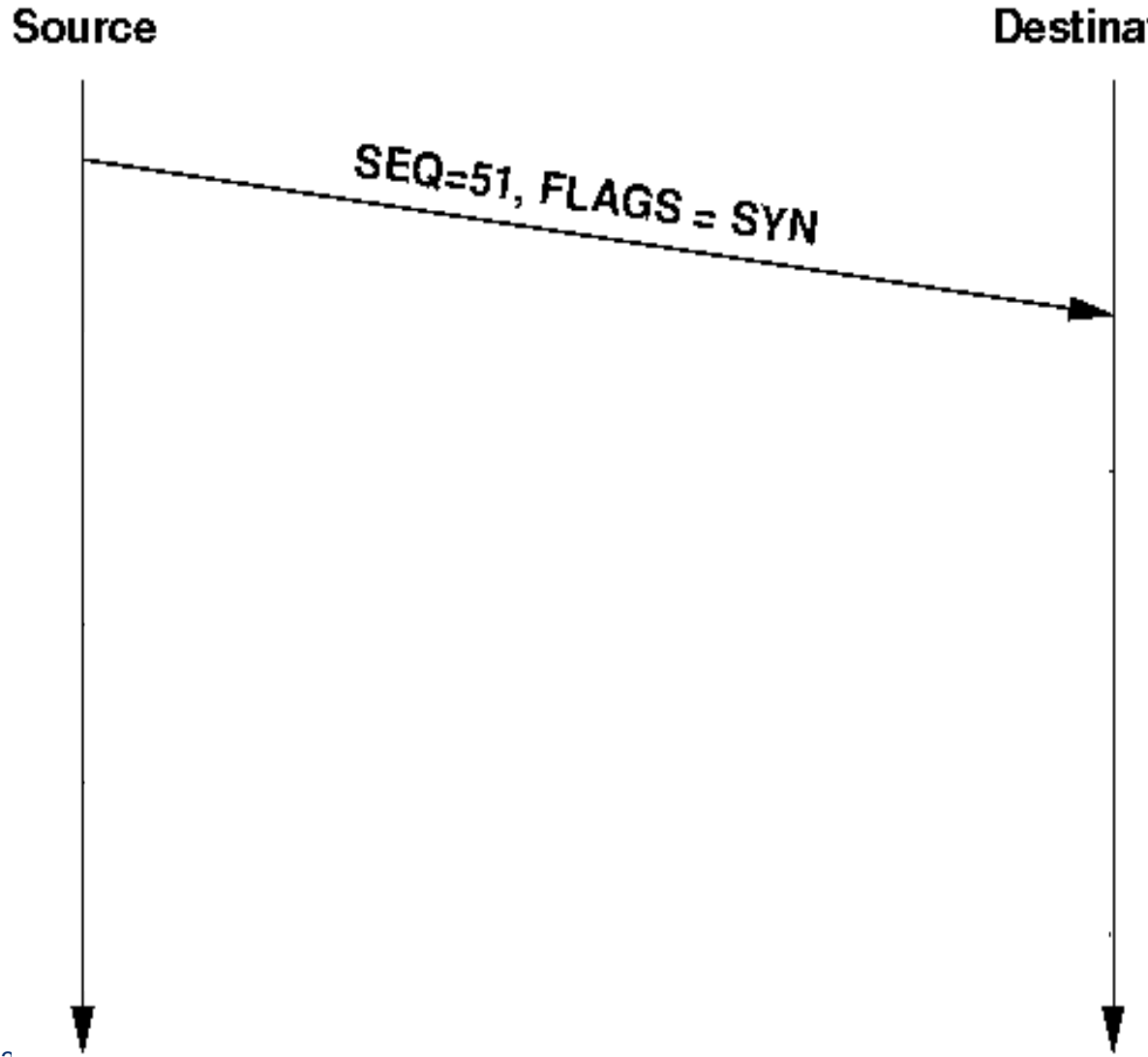
- Byteversatz von der aktuellen Folge-nummer, an der „dringende“ Daten vorgefunden werden

TCP-Verbindungsaufbau (a.k.a. Three-Way-Handshake)

Eine TCP-Verbindung wird durch einen sogenannten Three-Way-Handshake eröffnet

- Die Server-Applikation meldet sich an Socket an, die TCP-Instanz ist im Zustand LISTEN
- Die Client-Applikation fordert eine Verbindung zum Server. Es wird ein SYN-Paket gesendet mit
  - Sequence Number (hier: 51) und
  - weiteren Werten, z.B. für window size

TCP-Verbindungsaufbau: Schritt 1

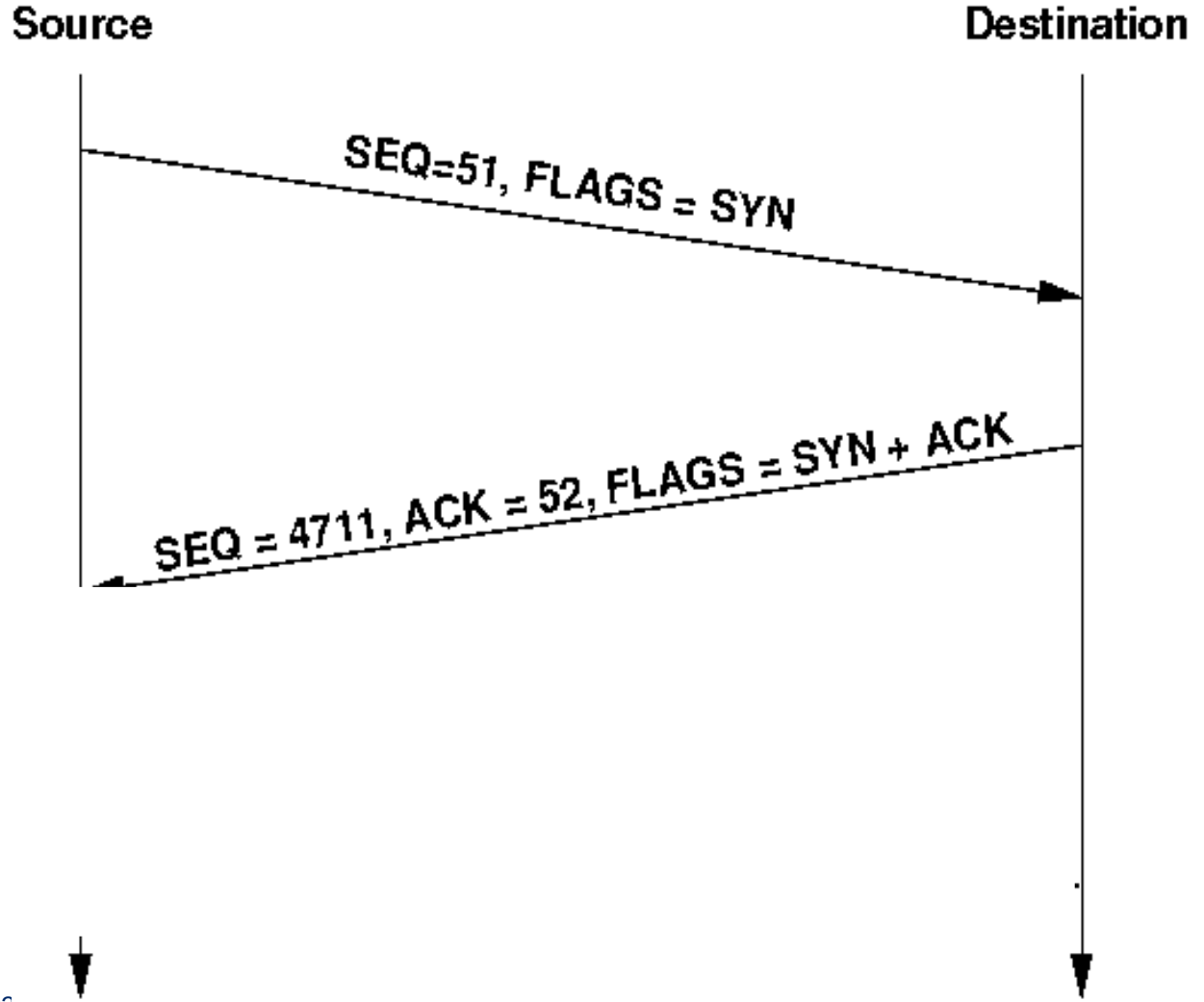


TCP-Verbindungsaufbau (a.k.a. Three-Way-Handshake)

(Fortsetzung – Schritt 2)

- Der Server antwortet mit einem SYN / ACK-Paket, das enthält:
  - Seine eigene Sequence Number (hier: 4711)
  - sowie weitere Werte, also auch window size
- Gleichzeitig wird der Empfang bestätigt, indem der nächste erwartete Wert für die Sequence Number des Clients (hier: 51+1 = 52) gesendet wird

TCP-Verbindungsaufbau: Schritt 2

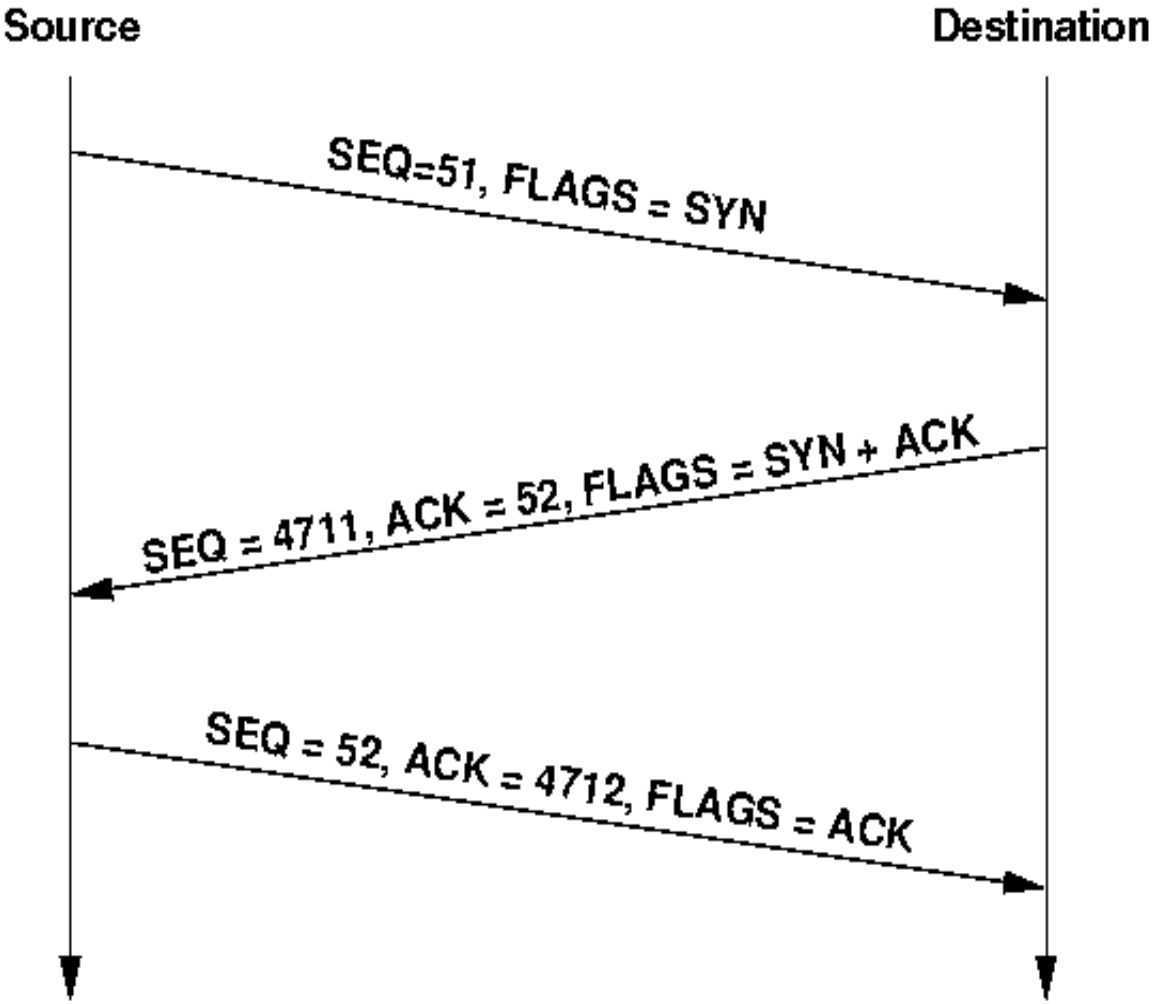


TCP-Verbindungsaufbau (a.k.a. Three-Way-Handshake)

(Fortsetzung – Schritt 3)

- Der Client bestätigt mit einem ACK-Paket die Sequence Number des Servers (hier: 4711+1 = 4712)
- Gleichzeitig wird die eigene Sequenznummer ebenfalls erhöht (hier: 51+1 = 52)

TCP-Verbindungsaufbau: Schritt 3

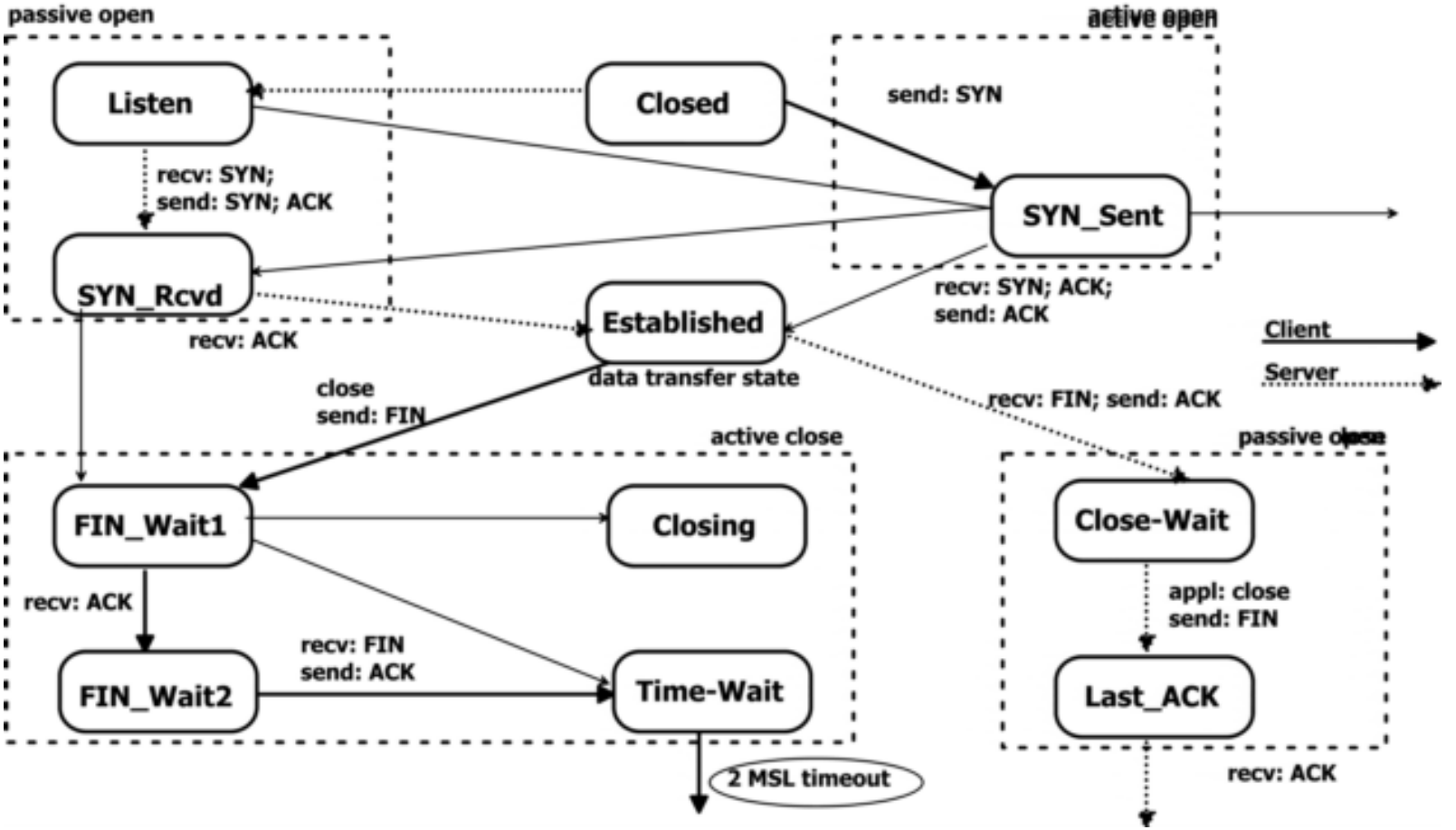


Zustände einer TCP-Verbindung

- Jeweils auf Sender- und Empfängerseite!

Zustand	Beschreibung
CLOSED	Keine Verbindung aktiv oder anstehend
LISTEN	Der Server wartet auf eine ankommende Verbindung
SYN RCVD	Ankunft einer Verbindungsanfrage und Warten auf Bestätigung
SYN SENT	Die Anwendung hat begonnen, eine Verbindung zu öffnen
ESTABLISHED	Zustand der normalen Datenübertragung
FIN WAIT 1	Die Anwendung möchte die Übertragung beenden
FIN WAIT 2	Die andere Seite ist einverstanden, die Verbindung abzubauen
TIMED WAIT	Warten, bis keine Pakete mehr kommen
CLOSING	Beide Seiten haben versucht, gleichzeitig zu beenden
CLOSE WAIT	Die Gegenseite hat den Abbau eingeleitet
LAST ACK	Warten, bis keine Pakete mehr kommen

TCP-Statusdiagramm

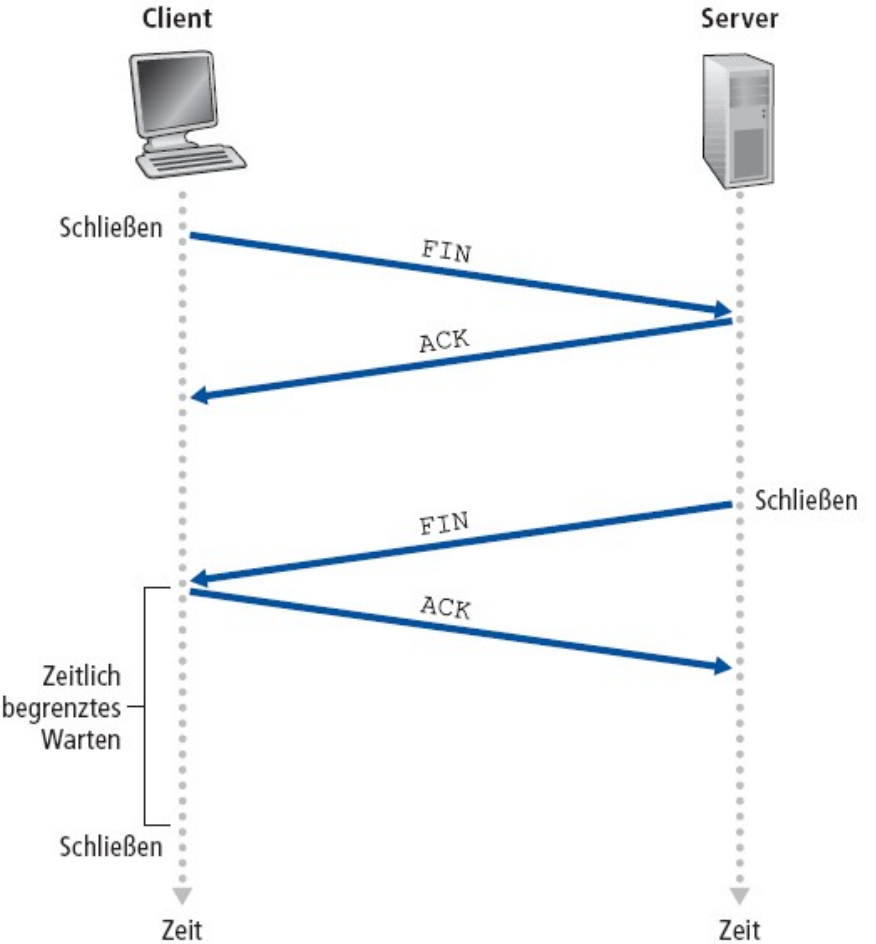


… und beim Abbau

```
clientSocket.close();
```

Server kann sein FIN zusammen mit dem ACK schicken.

Client wartet, weil sein ACK verloren gehen könnte



Sicherung des TCP-Protokolls

TCP sichert den Transport seiner Segmente immer so ab, dass beim Empfänger ein vollständiger und geordneter Datenstrom ankommt!

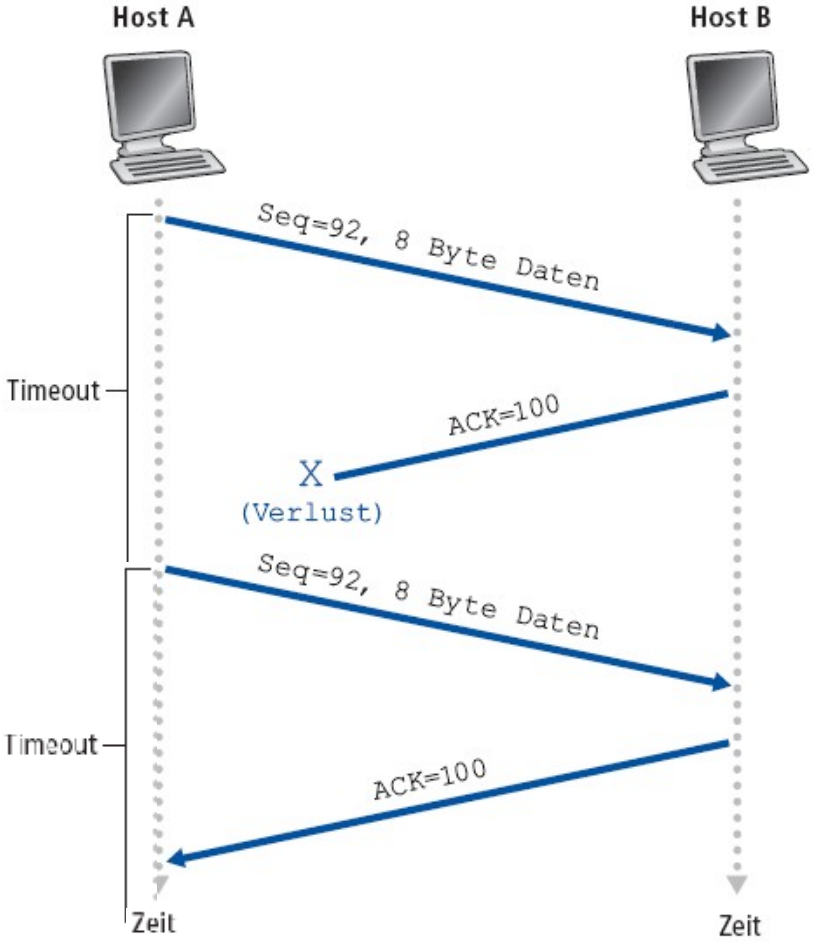
- Fehlt ein Segment, wird der Strom angehalten – d.h. eintreffende Pakete werden verworfen – und auf das fehlende Paket gewartet: „Head of Line Blocking“
- Datenverluste werden anhand der Sequenznummern erkannt

(Optimierte Variante nur als Option!)

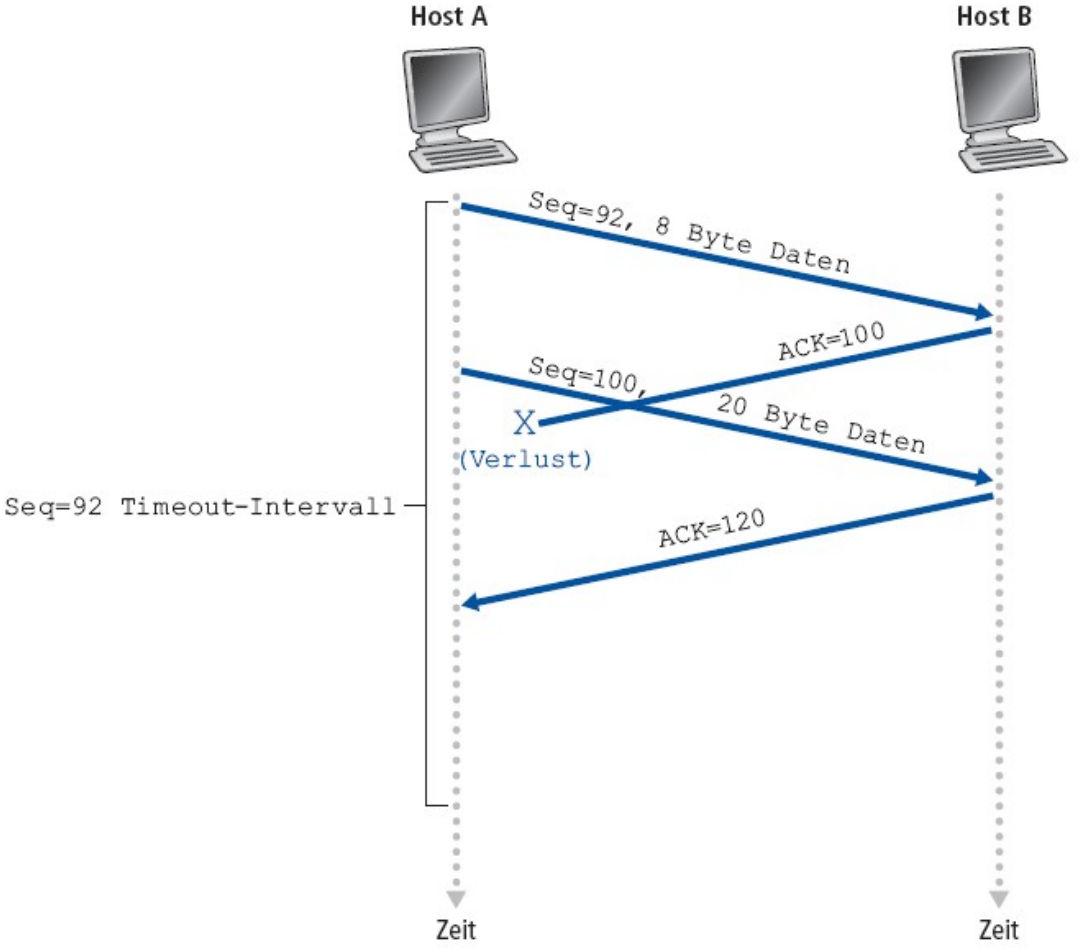
Sicherung des TCP-Protokolls (2)

- TCP meldet generell keine Verluste, sondern versendet Quittungen (ACKs) für korrekt eingegangene Segmente
- Per Definition wird immer das letzte zusammenhängend angekommene Segment quittiert – auch bei Fehler in der Reihenfolge:
  „Cumulative acknowledgement“
- Läuft der Timer ab, werden alle noch nicht bestätigten Segmente erneut gesendet, sogenanntes „go back N“

TCP-Quittung und Re-Transmission



Kumulative TCP-Quittung



Sicherung des TCP-Protokolls (3)

- **Empfangene Segmente werden anhand der Sequenznummern in ihrer ursprünglichen Reihenfolge erkannt**
  - Bei einer zusammenhängenden Kette von Segmenten ist TCP „quittungsbereit“
- **TCP versucht, ACK gemeinsam mit Daten zu senden: „Piggybacking“**
  - Sind keine Daten „versandfertig“, wird das ACK verzögert
  - Treffen innerhalb eines Zeitintervalls (typisch sind wenige ms) keine Daten ein, wird das ACK auch alleine versandt

Erzeugung von TCP ACKs

Ereignis	Aktion des TCP-Empfängers
Ankunft des Segmentes in der richtigen Reihenfolge mit der erwarteten Sequenznummer. Alle Daten bis zur erwarteten Sequenznummer sind bereits bestätigt.	Verzögertes ACK. Wartet bis zu 500 ms auf die Ankunft eines anderen Segmentes in richtiger Reihenfolge. Wenn das nächste Segment nicht in diesem Zeitintervall eintrifft, wird ein ACK gesendet.
Ankunft eines Segmentes in der richtigen Reihenfolge mit erwarteter Sequenznummer. Ein anderes Segment in der korrekten Reihenfolge wartet auf die ACK-Übertragung.	Sendet sofort ein einzelnes kumulatives ACK, bestätigt beide in richtiger Reihenfolge eingetroffene Segmente.
Ankunft eines Segmentes außerhalb der Reihenfolge mit einer Sequenznummer, die größer ist als erwartet. Lücke im Bytestrom aufgetreten.	Sendet sofort ein doppeltes ACK, in dem er die Sequenznummer des nächsten erwarteten Bytes angibt.
Ankunft eines Segmentes, das die Lücke in den erhaltenen Daten ganz oder teilweise ausfüllt.	Sendet sofort ein ACK, vorausgesetzt, das Segment beginnt mit der Sequenznummer des nächsten erwarteten Bytes. Bestätigt alle nun lückenlos vorliegenden Bytes.

Wann erfolgt eine Re-Transmission?

- **Feste Timeouts sind problematisch bei variabler Verzögerung**
  - zu groß: Performanceverlust
  - zu klein: unnütze Wiederholungen
- **Lösungsansatz: Messen der sogenannten Round Trip Time (= durchschnittl. Verzögerung zwischen Aussenden der Daten und Eingang der Quittung)**
  - TCP ermittelt RTT für jede Verbindung
  - Aber nicht für jedes Segment

Wann erfolgt eine Re-Transmission?

- **TCP ermittelt RTT für jede Verbindung**
  - Retransmit Timer basiert auf der RTT und ihrer Variation
- **dennoch: Probleme bei schnell veränderlicher RTT!**

Re-Transmit Timeout

- **TCP ermittelt „erwartete“ expRTT aus den gemessenen RTT-Werten (mit x=0,25, y=x/2):**

  $expRTT_{N+1} = (1-y) * expRTT_N + y * RTT_N$

- **sowie die Variation (Jitter) der Verzögerungen:**

  $Jitter_{N+1} = (1-x) * Jitter_N + x * | RTT_N - expRTT_N |$

- **und daraus den Timeout:**

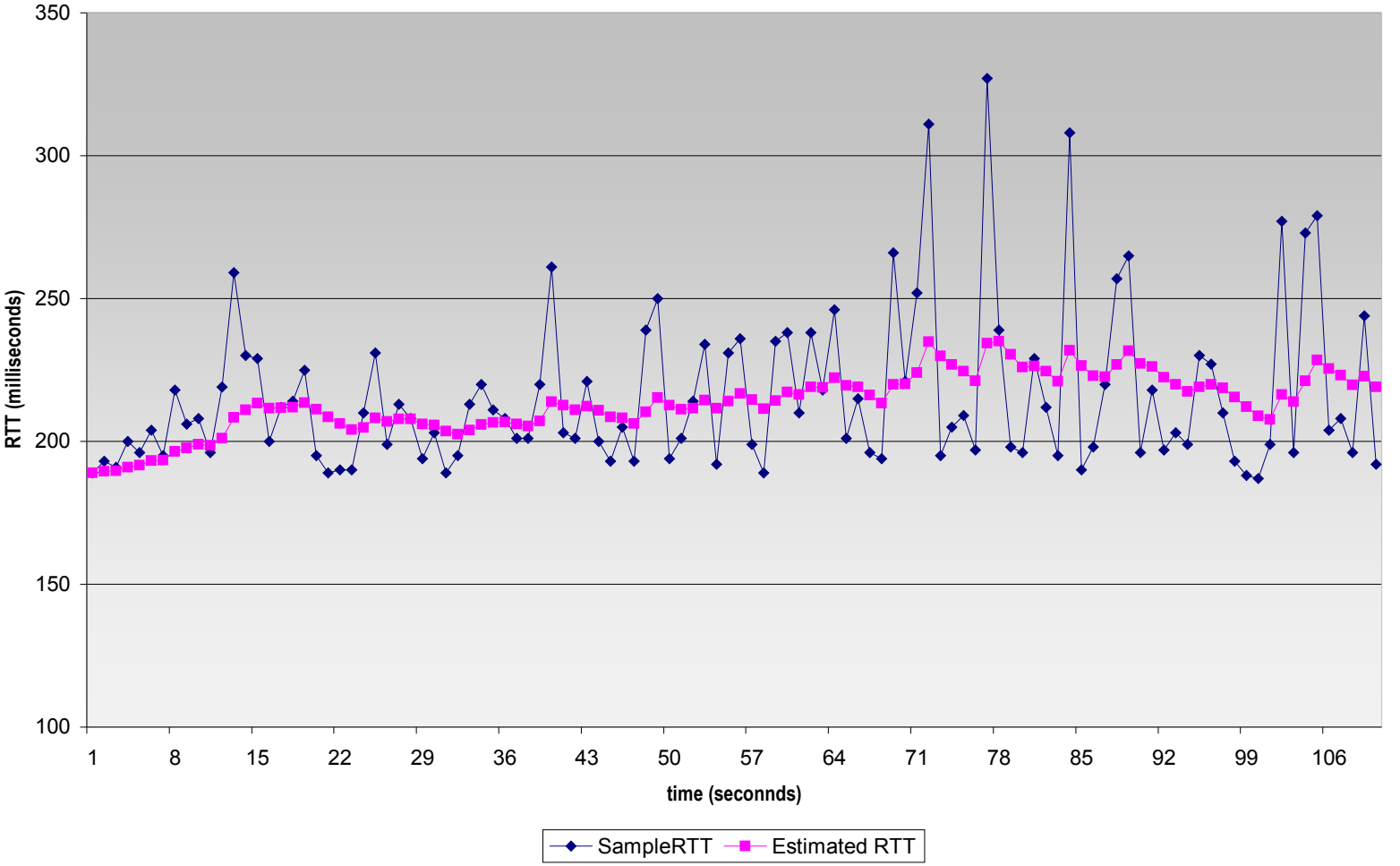
  $Timeout_{N+1} = expRTT_N + 4 * Jitter_N$

- **Anpassung bei Re-Transmit:**

  $Timeout_{N+1} = 2 * Timeout_N$

$Timeout_0 = 1\ s$

Beispielhafter Verlauf



… und wie sind TCP und UDP im Vergleich?

Qual der Wahl: TCP oder UDP?

TCP – Dienste:

- Zuverlässig
- Datenstrom
- Reihenfolge erhaltend
- Flusskontrolle durch Empfänger
- Staukontrolle
- Nicht geboten:
  - Garantien über Verzögerung oder Kapazität

UDP – Dienste:

- Unzuverlässig
- einzelne Pakete
- geringer Overhead
- Nicht geboten:
  - Verb.-aufbau
  - Flusskontrolle
  - Staukontrolle
  - Garantien über Verzögerung und Kapazität

Vergleich TCP - UDP

Funktion	TCP	UDP
Ende-zu-Ende-Kontrolle	ja	nein
Zeitüberwachung der Verbindung	ja	nein
Flusskontrolle über das Netz	ja	nein
Zuverlässige Datenübertragung	ja	nein
Geschwindigkeit	normal	hoch
Erkennung von Duplikaten	ja	nein
Reihenfolgerichtige Übertragung	ja	nein
Verbindungsaufbau	ja	nein
Multiplexen von Verbindungen	ja	ja

Qual der Wahl: TCP oder UDP?

TCP – Dienste:

- Zuverlässig
- Datenstrom
- Reihenfolge erhaltend
- Flusskontrolle durch Empfänger
- Staukontrolle
- Nicht geboten:
  - Garantien über Verzögerung oder Kapazität

UDP – Dienste:

- Unzuverlässig
- einzelne Pakete
- geringer Overhead
- Nicht geboten:
  - Verb.-aufbau
  - Flusskontrolle
  - Staukontrolle
  - Garantien über Verzögerung und Kapazität

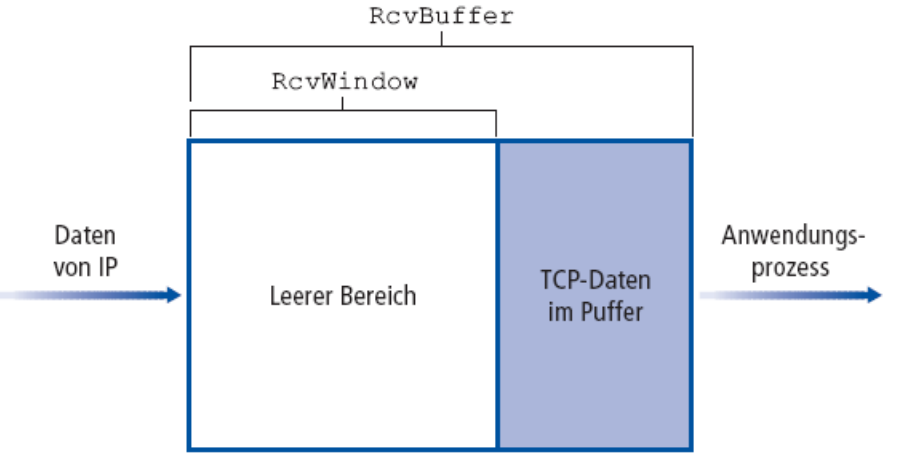
… wie macht TCP das denn?

→ Flusskontrolle

Dynamische Flusskontrolle

- TCP teilt den Datenstrom zur Übertragung in Segmente (= Übertragung in einem TCP-Paket) ein
- Der Empfänger gibt dem Sender zurück, wie groß sein freier Puffer ist:

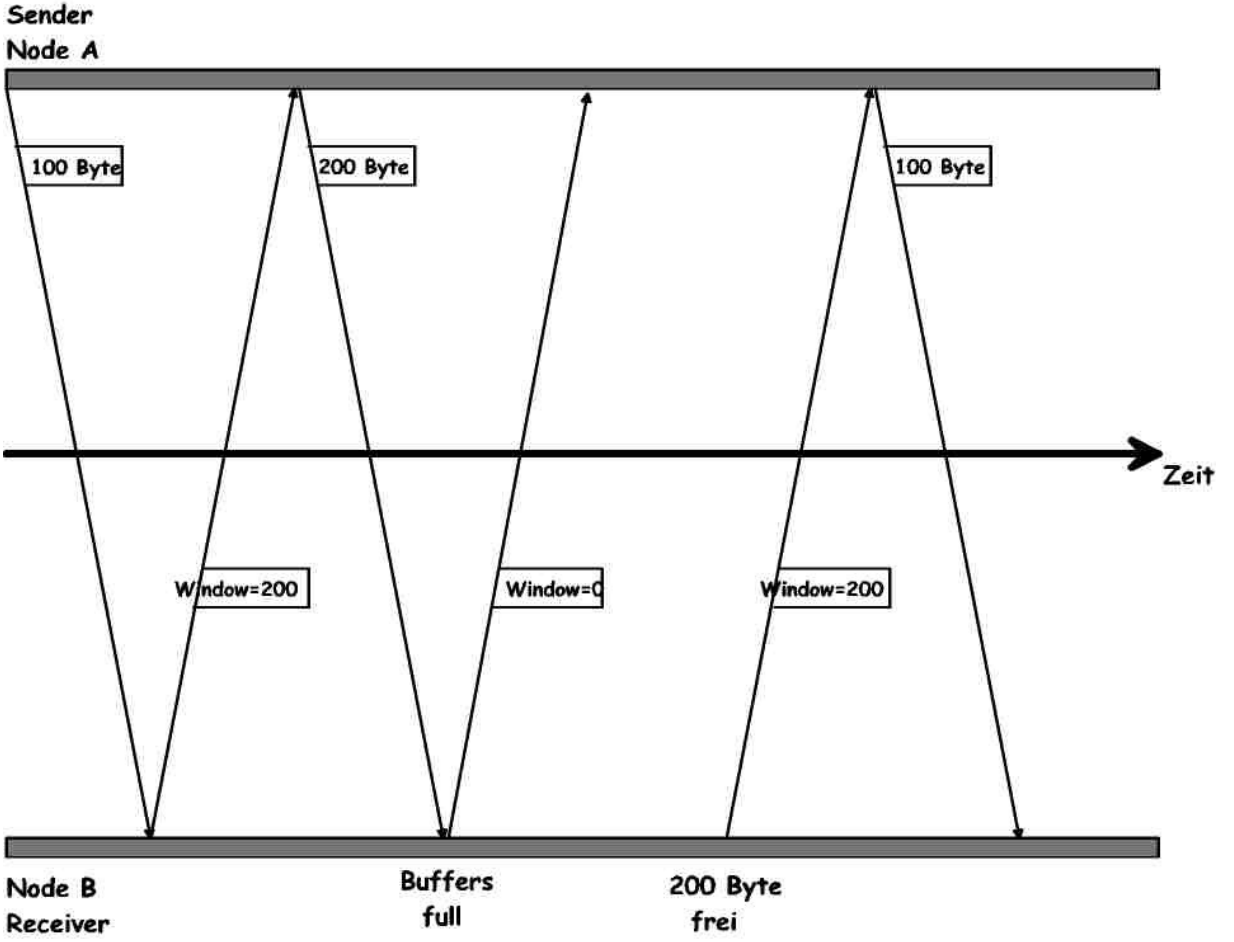
  window size
- Ein Fenster der Größe „0“ stoppt den Fluss



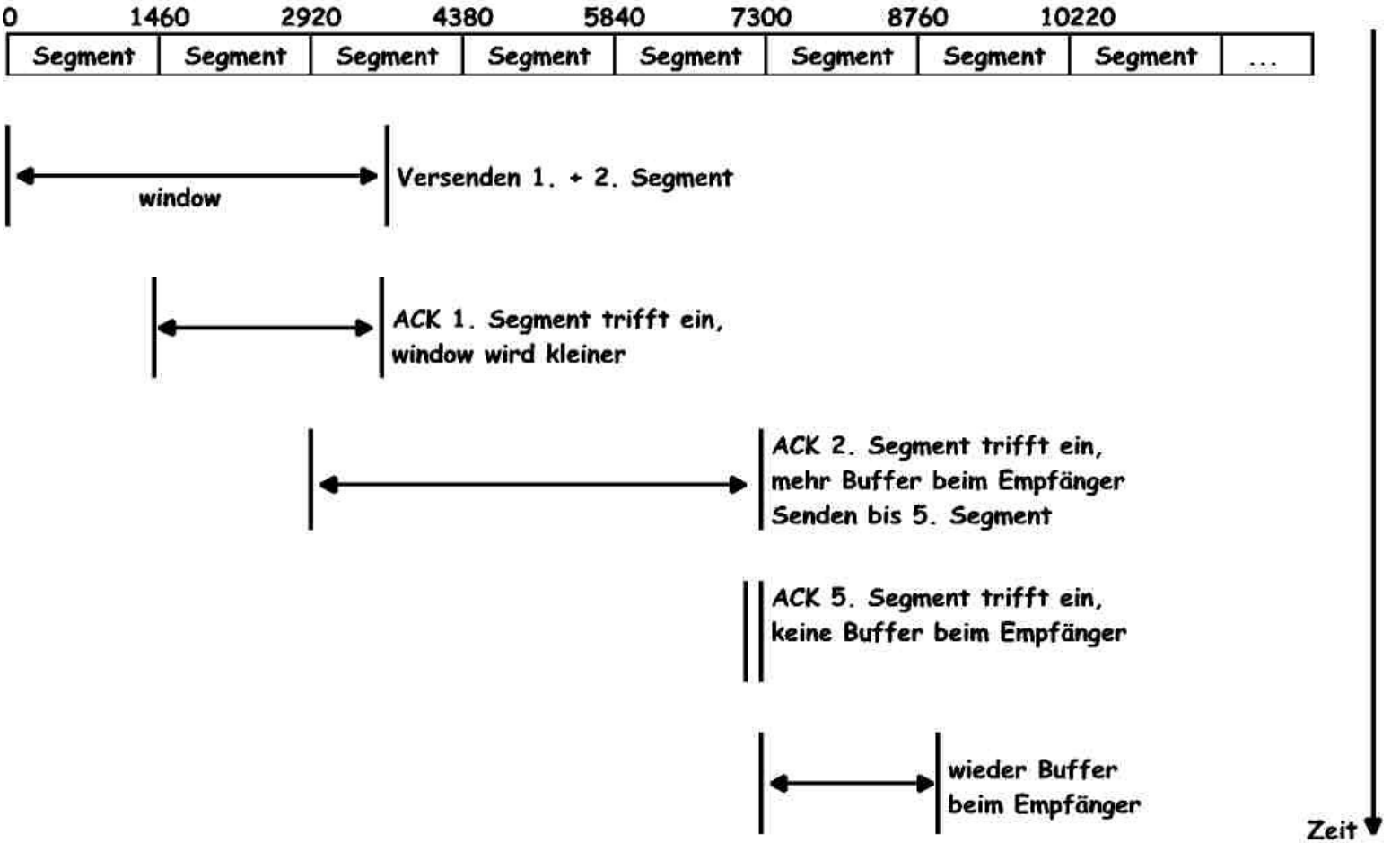
Dynamische Flusskontrolle

- TCP teilt den Datenfluss zur Übertragung in Segmente (= Übertragung in einem TCP-Paket) ein
- Der Empfänger teilt dem Sender mit, für groß sein freier Puffer ist: **window size**
- Der Sender begrenzt die Menge der unbestätigt gesendeten Daten auf die Größe des verfügbaren Puffers

Dynamische Flusskontrolle



Dynamische Flusskontrolle (2)



… wie macht TCP das denn?

→ Staukontrolle

Prinzipien der Staukontrolle (Überlastkontrolle)

“zu viele Quellen senden zu viele Daten zu schnell, das Netzwerk kann sie nicht alle bearbeiten”

- **Erfordert andere Maßnahmen als die Flusskontrolle, die die Kommunikation zwischen Sender und Empfänger steuert**
- **Staus feststellbar durch:**
  - verlorene Pakete:
    Pufferüberlauf in den Routern
  - große Verzögerungen:
    lange Queues in den Puffern der Router

Kosten von Staus

- **Verlorengegangene Pakete müssen wiederholt werden**
- **Starke Verringerung der Übertragungsrate (“Durchsatz”)**
  - Tendenz: → 0 bei dauerhafter Überlast
- **Große Paketverzögerungen**
  - Tendenz: → ∞ bei dauerhafter Überlast

Stauvermeidung - V. Jacobson `88

- **Wie viele Daten darf eine Quelle auf einmal senden?**
  - Ursprünglich unkontrolliert, bestand immer die Gefahr eines Problems!
- **TCP vermeidet Netzwerkstaus durch drei Maßnahmen:**
  - Slow Start
  - Congestion Avoidance
  - Fast Retransmit
- **Der Sender beobachtet das Netz und leitet daraus die „richtige“ Maßnahme ab!**

Stauvermeidung - V. Jacobson `88

- **Ein „Congestion Window“ wird beim Sender geführt und begrenzt die Übertragungsrate**
  - Größe gemessen in Zahl von Segmenten
  - Zunächst mit Wert „1“ initialisiert
- **Es gibt einen „Threshold“, bis zu dessen Erreichen trotz des „Slow Starts“ mit jedem bestätigten Segment das Fenster stark ansteigt**
- **Danach wird versucht, durch „Congestion Avoidance“ einen Stau zu vermeiden**

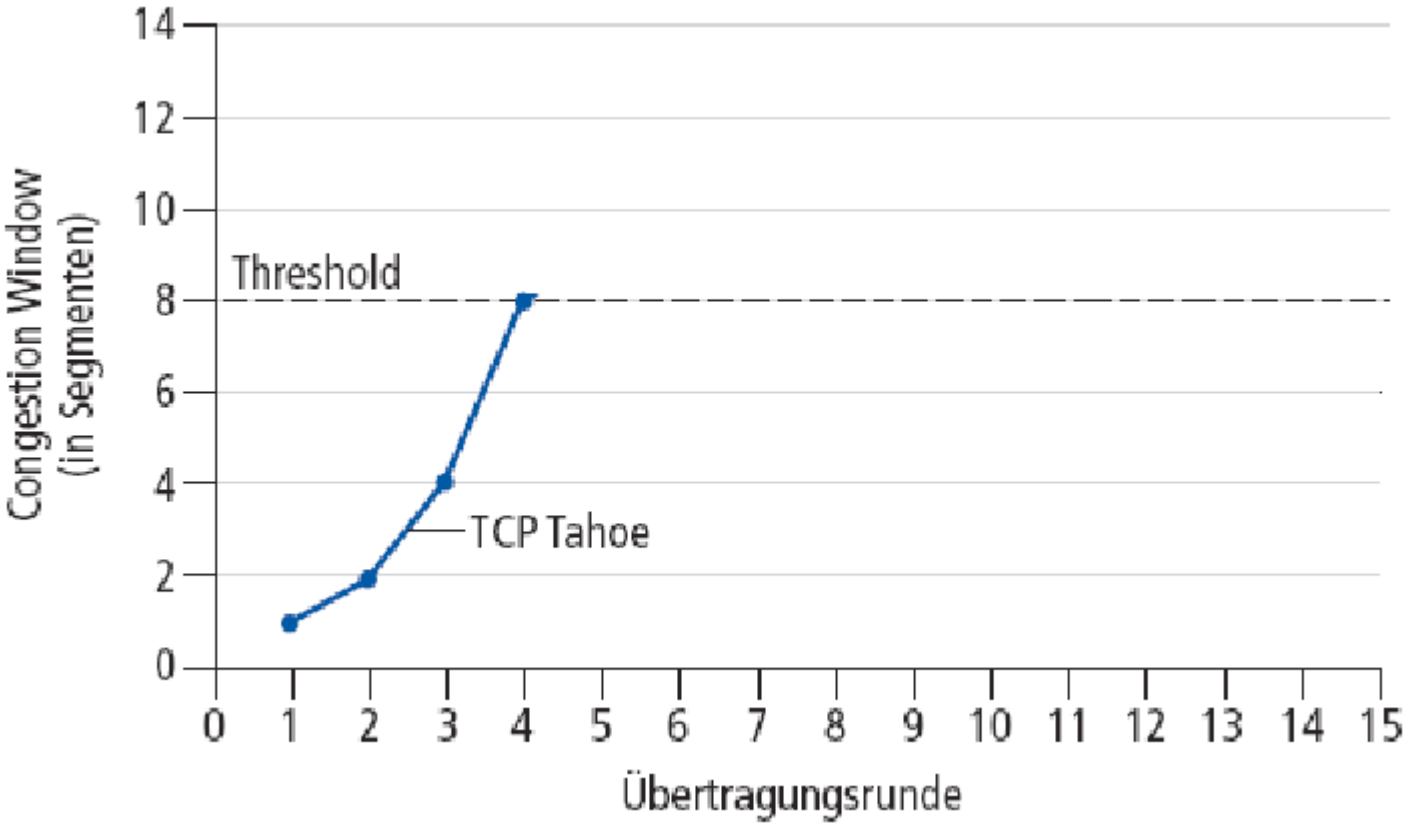
Stauvermeidung - V. Jacobson `88

Slow Start

- **Der Name „Slow Start“ bezieht sich auf den kleinen initialen Wert des „Congestion Windows“**
  - Es startet mit dem Wert von „1“
  - Pro ACK erhöht sich das „Congestion Window“ um 1 weiteres Segment
    - Dies hat ein exponentielles Wachstum zur Folge

      Segment 1 → Segment 2 → Segment 4
      → Segment 5
      → Segment 3 → Segment 6
      → Segment 7

Slow Start
= möglichst schnelle Annäherung

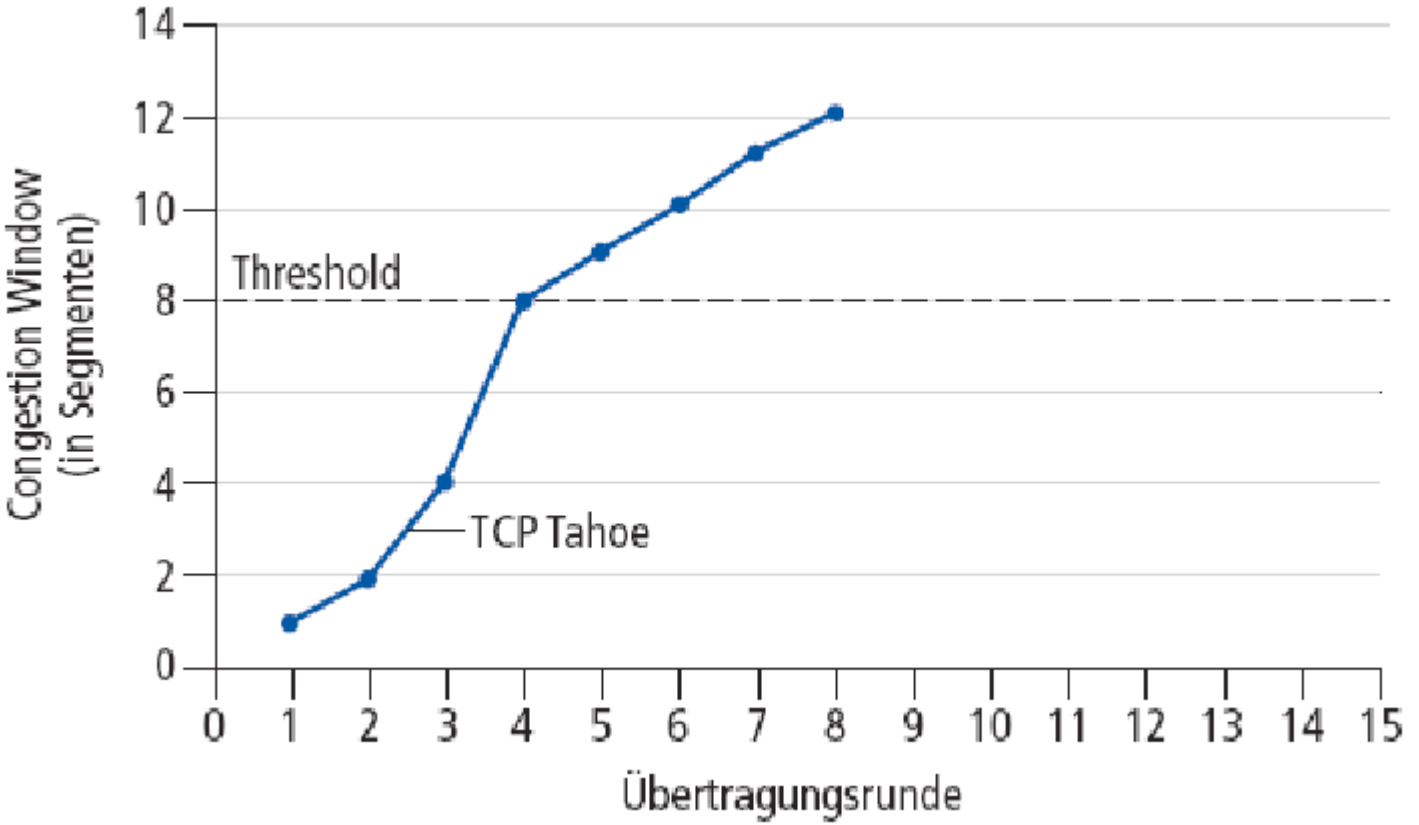


Stauvermeidung - V. Jacobson `88

Congestion Window

- **Wenn „Congestion Window“ > „Threshold“ ist, ist die „Slow Start“ Phase vorbei**
  - Danach wächst das „Congestion Window“ nur noch linear → max. +1 pro RTT
  - Dies vermeidet, durch weiteres exponentielles Wachstum einen Stau zu provozieren

Threshold
= Indikator für bisheriges Verhalten



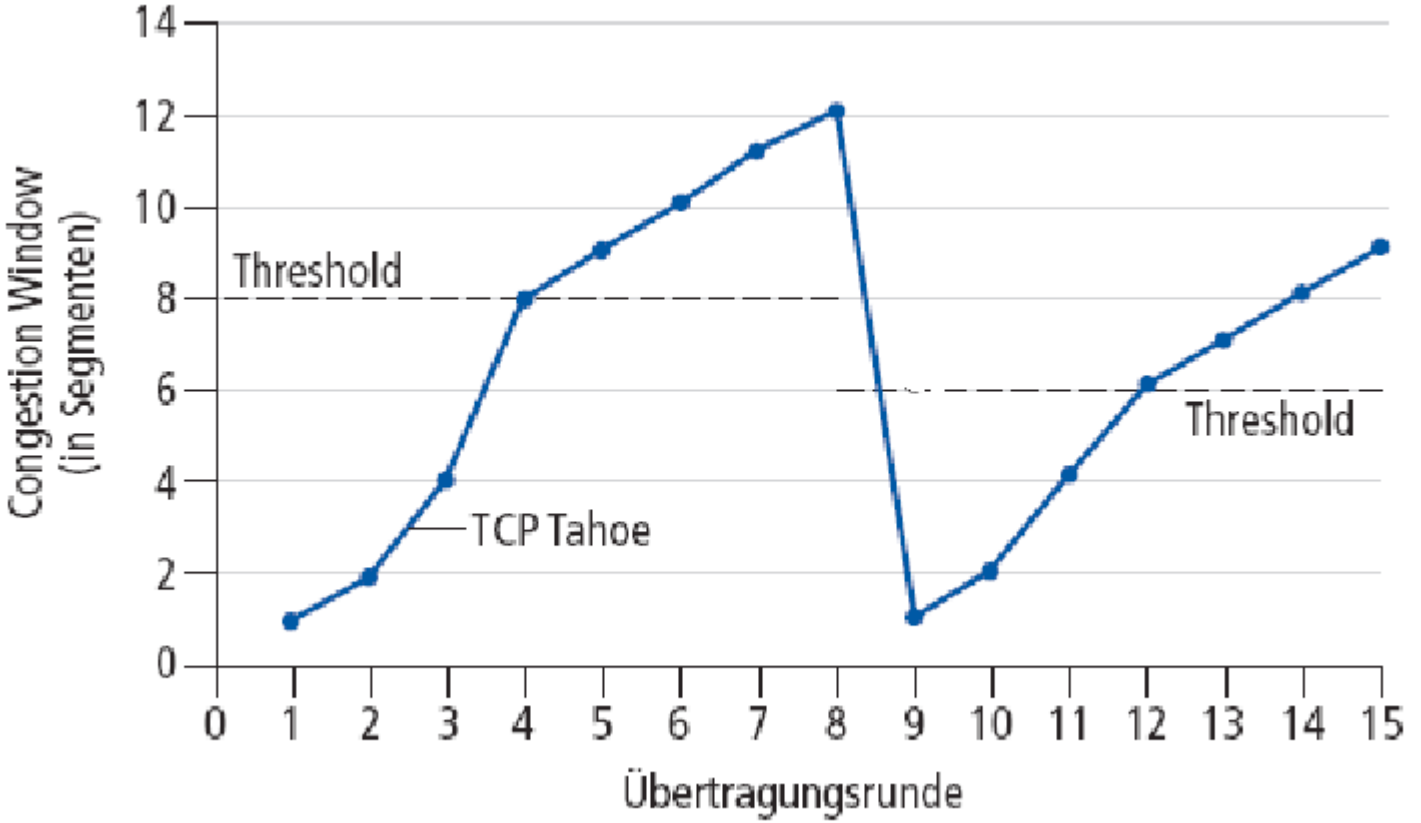
Stauvermeidung - V. Jacobson `88
Warten auf Paketverluste

- **Woran werden Paketverluste erkannt?**
  - **1. Timeout beim Sender wird erreicht**
    - Entweder das gesendete Paket oder das ACK sind verloren gegangen
  - **2. Beim Empfänger gehen ACKs ein, jedoch wird anhand der Sequenznummern deutlich, dass (mind.) ein Paket der Kette verloren gegangen ist**
    - „doppelte“ ACKs werden nach Eingang eines Segments gesendet, wenn es nicht fortlaufend ist

Stauvermeidung - V. Jacobson `88
Warten auf Paketverluste (2)

- **Wenn „doppelte“ ACKs ankommen, sind ja auch Segmente beim Empfänger angekommen – es gibt aber eine Lücke**
  - wahrscheinlich handelt es sich nur um den Verlust eines einzelnen Segments
  - sonst würden die ACKs nicht ankommen bzw. Timeouts entstehen
- **Erst bei konkretem Verdacht auf Paket-verlust**
  - „Threshold“ auf die Hälfte des aktuellen Congestion Windows und „Slow Start“

Neuer Threshold
= Hälfte des Congestion Windows



Jacobson Fast Retransmit

Wird auch nur ein Segment verloren, muss dieses – und alle danach gesendeten – wieder übertragen werden

Außerdem muss mit „Slow Start“ wieder neu angefangen werden

Einfache Idee zur Verbesserung:

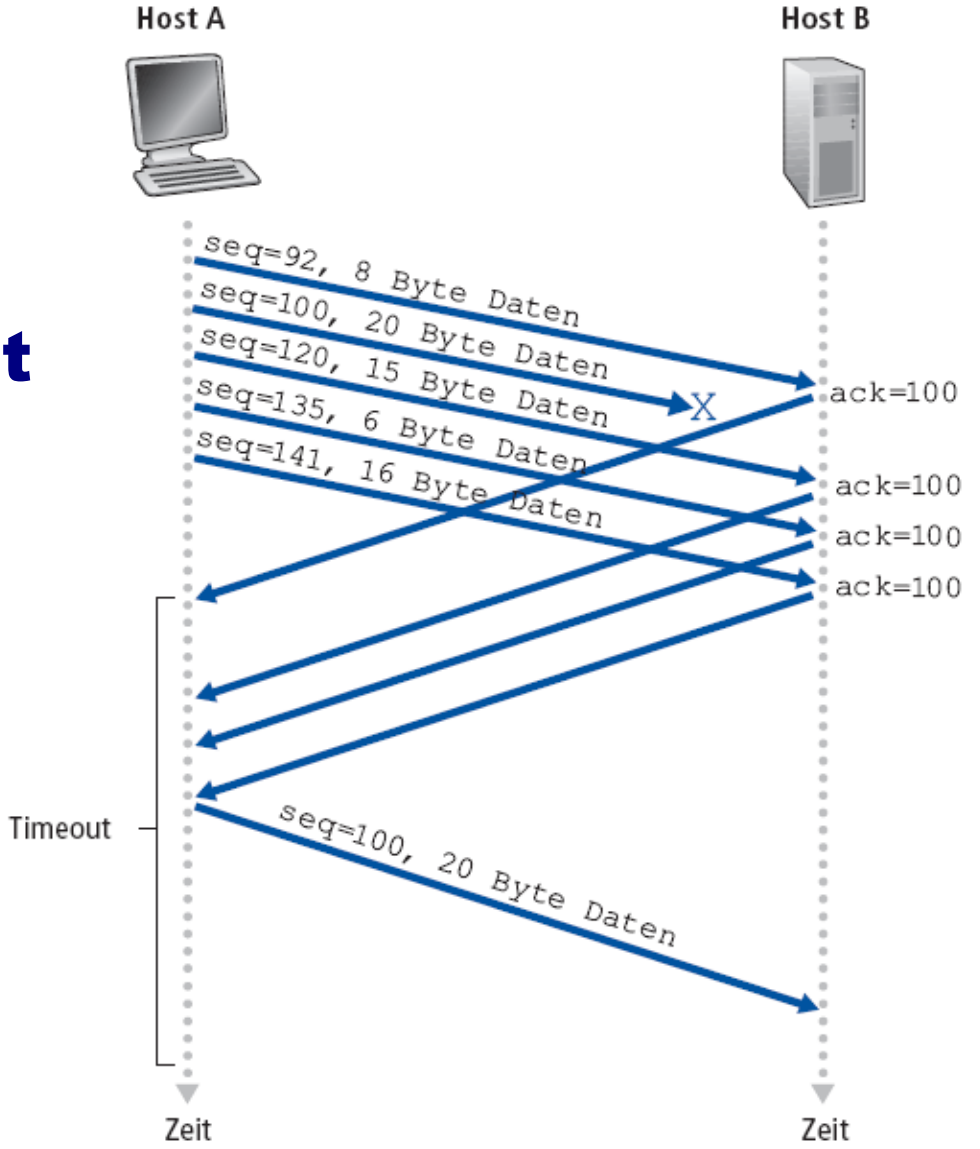
- Empfänger sendet erneute Quittung für die bisher korrekt übertragene Kette
- Das Congestion Window wird nicht auf „1“ gesetzt, wenn das fehlende Segment schnell eintrifft (ohne Timeout)

Jacobson Fast Retransmit

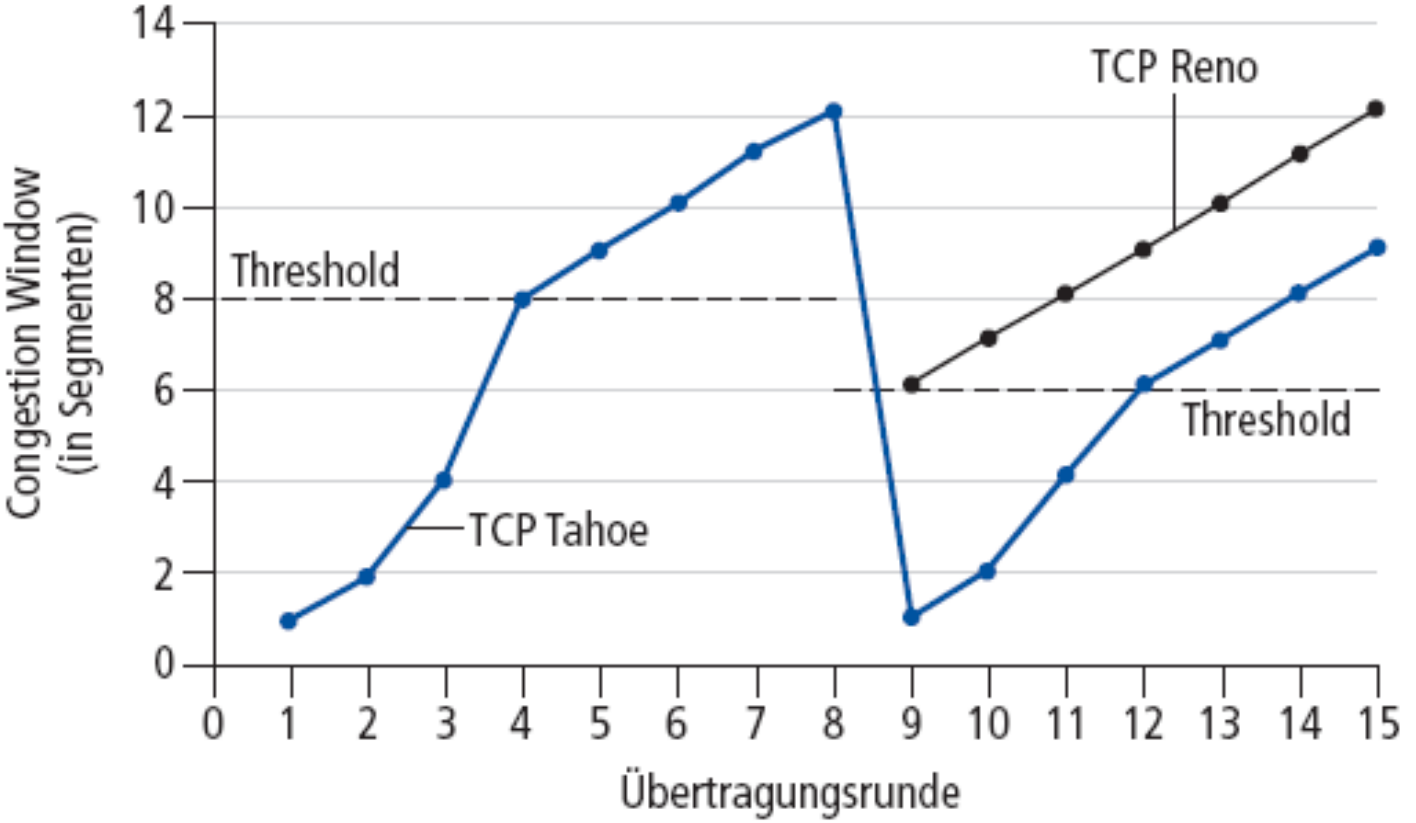
Wenn mehrere doppelte ACKs eintreffen, ist wahrscheinlich tatsächlich nur ein einzelnes Segment verloren worden, weil sonst auch die ACKs nicht ankommen würden!

- **Fast Retransmit:**
  - Wird das dritte „duplicate ACK“ empfangen, schickt der Sender das erste bisher nicht bestätigte Segment erneut
  - Der Sender wechselt nicht in die Slow Start Prozedur, sondern beginnt gleich bei Threshold

Beispiel für Fast Retransmit



Kein Slow Start
= Beginn bei Threshold



Weitere kleinere Optimierungen

Weitere Optimierungen

- **TCP besteht seit seiner Erfindung unverändert ‚on the wire' – und kennt auch keine Versionsnummern**
- **Warum also überhaupt optimieren?**
  - effizienter und leistungsfähiger
  - veränderte Übertragungsanforderungen z.B. Wireless
  - Gestiegene Kapazitäten der Endgeräte
- **Herausforderung liegt darin, gleichzeitig kompatibel zu bleiben!**

Keine „Tinygrams“

- **Auch einzeln versendete Datenbytes benötigen den 40 Byte langen TCP-Header**
- **Nagle Algorithmus vermeidet kleine Pakete:**
  - Statt kleiner Pakete werden Daten so lange gesammelt, wie es passt
- **Problem dabei:**
  - Graphische Interaktionen und Tastatur

    => Socket-Option TCP_NODELAY schaltet den Algorithmus aus

Selective Acknowledgment / SACK

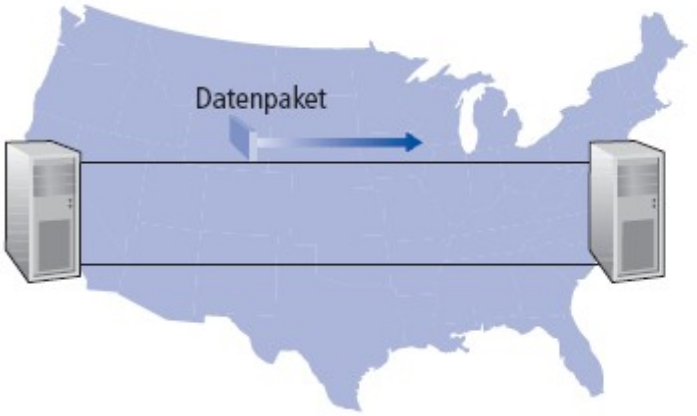
- **Komplexerer Lösungsansatz für das gleiche Problem optimiert Datenübertragung**
- **Bereiche innerhalb eines „sliding“ Windows können auch nur teilweise quittiert werden (SACK)**
  - Der Sender hat dann die Möglichkeit, die unquittierten Segmente erneut zu senden
- **Erfolgt ein erneuter Timeout, wird von der letzten kumulativen Standardquittung an erneut gesendet**

Selective Acknowledgment / SACK

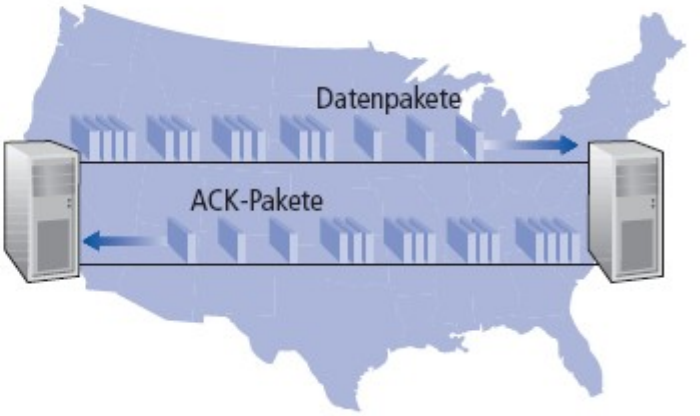
- **Muss initial bereits bei dem Verbindungsaufbau (SYN) verhandelt werden**
  - Selektive Quittungen des Empfängers erfolgen in dem Feld options des TCP-Headers
- **Der Sender muss dafür eine separate ‚SACK'-Tabelle führen**
  - Allerdings kann der Sender die Option auch einfach ignorieren
  - Weiterhin Meta-Daten im TCP-Header

… also mehr als ein Paket gleichzeitig?

… besser sind Pipelines!



a Ablauf bei Stop-and-Wait



b Ablauf bei Pipelining

Pipelining

… wird möglich, wenn der Sender Pakete unterscheiden kann, die noch „unterwegs" sind und bestätigt werden müssen

- Bereich der Sequenznummern muss vergrößert werden
- Puffer müssen beim Sender und ggf. beim Empfänger bereitgestellt werden

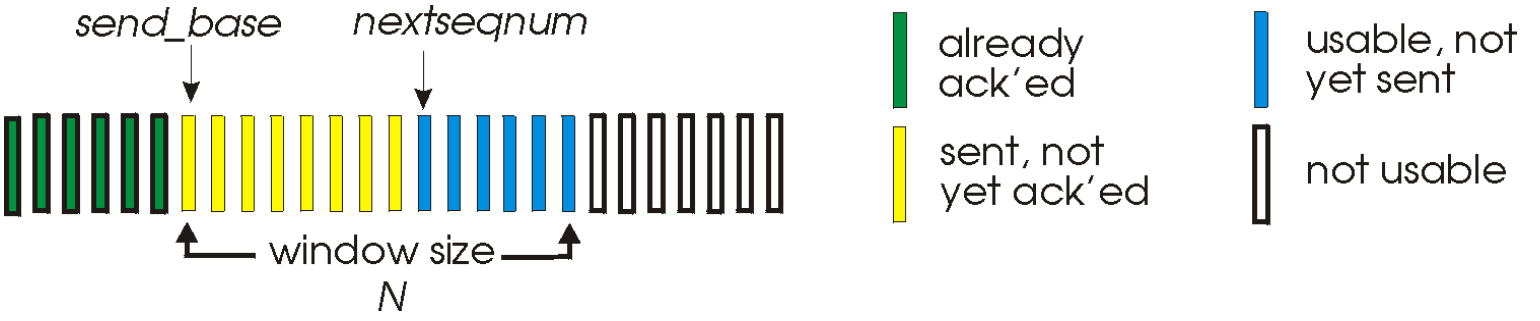
Zwei grundsätzliche Arten:

- Go-Back-N
- Selective Repeat

Go-Back-N

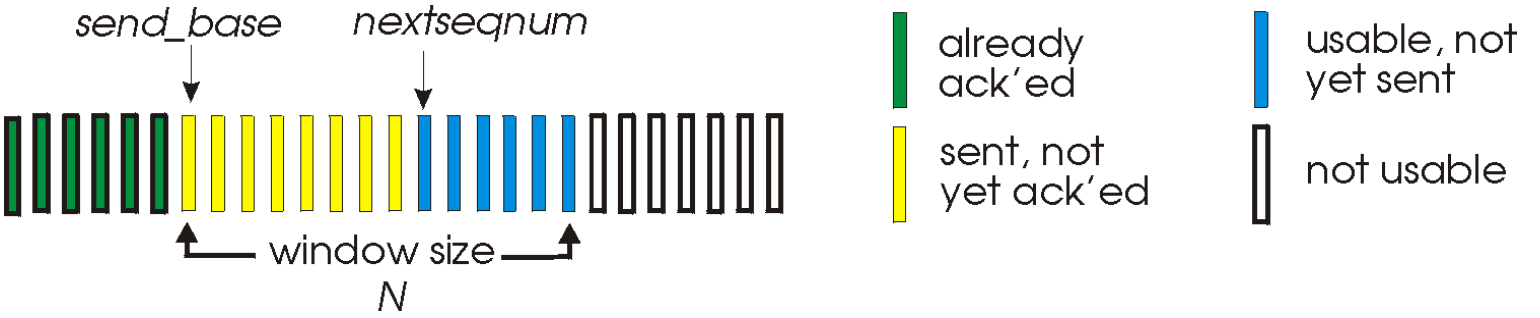
Sender:

- k-bit Sequenznummer im Paket-Header
- Fenster (“window”) erlaubt bis zu N aufeinanderfolgende, nicht bestätigte Pakete

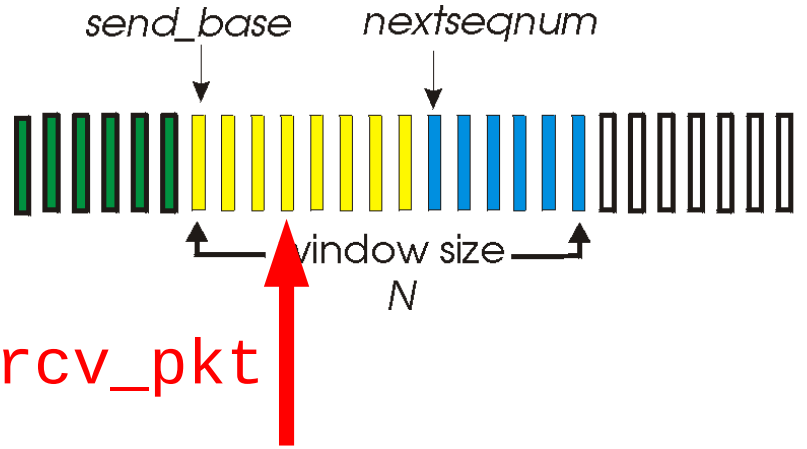


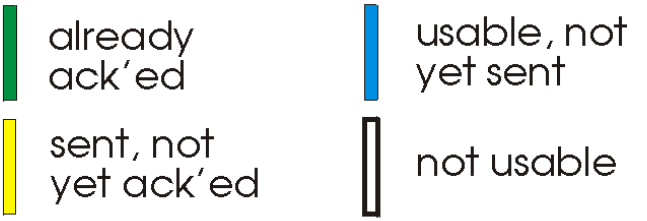
Go-Back-N (2)

- ACK(n) bestätigt alle Pakete incl. dem mit Sequenznummer n - “Kumulatives ACK”
- Timer für das älteste noch nicht bestätigte Paket (send_base)
- timeout(n): Sendewiederholung von Paket n und Pakete mit höherer Sequenznummer



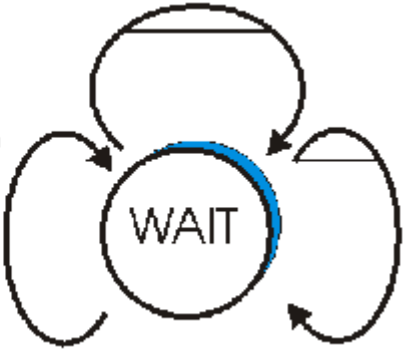
Go-Back-N: Erweiterte FSM des Senders – Eingang einer Bestätigung



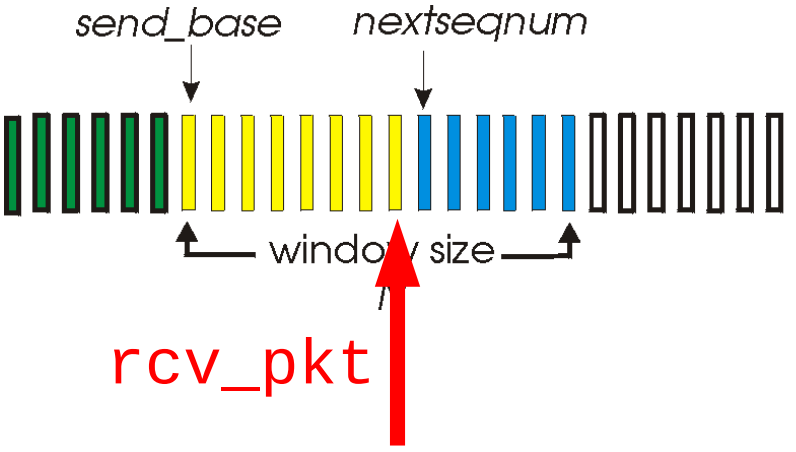


rdt_rcv(rcv_pkt) && notcorrupt(rcvpkt)

```
base = getacknum(rvcpkt)+1
if (base == nextseqnum)
   stop_timer
 else
   start_timer
```



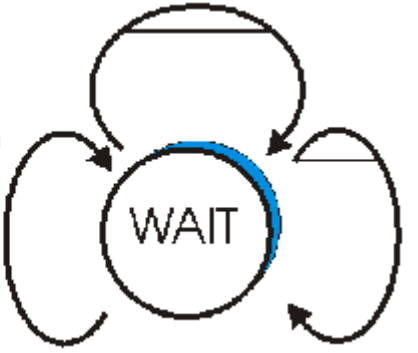
Go-Back-N:
Erweiterte FSM des Senders
– Bestätigung aller ausstehenden



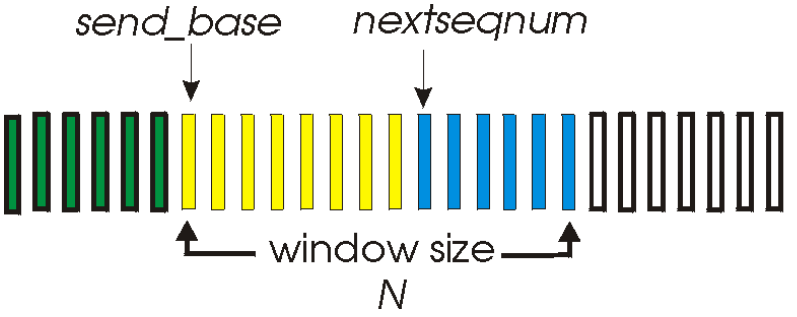
already ack'ed
sent, not yet ack'ed
usable, not yet sent
not usable

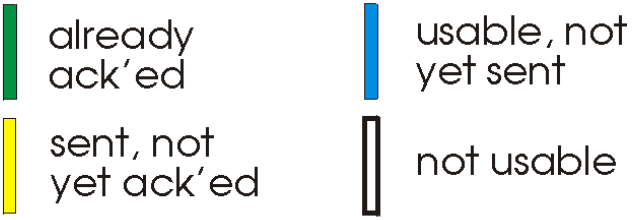
rdt_rcv(rcv_pkt) && notcorrupt(rcvpkt)

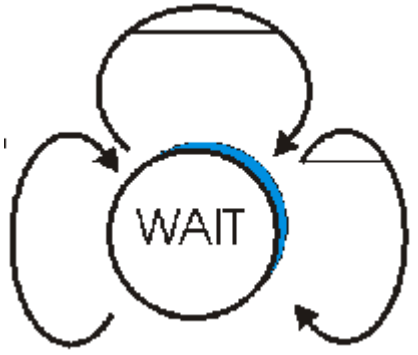
```
base = getacknum(rcvpkt)+1
if (base == nextseqnum)
   stop_timer
 else
   start_timer
```



Go-Back-N: Erweiterte FSM des Senders – Neue Daten zum Senden



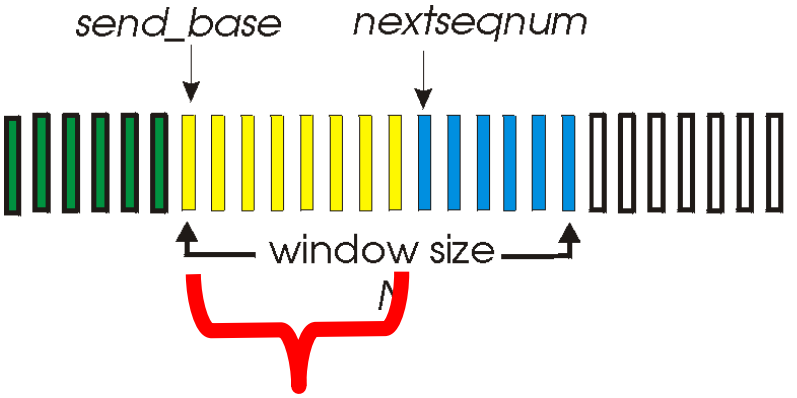


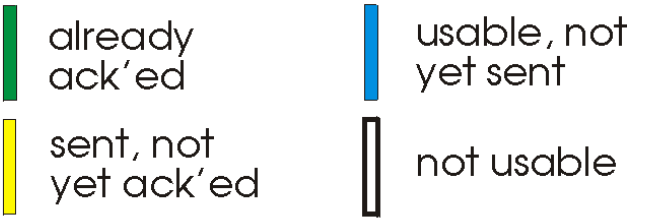


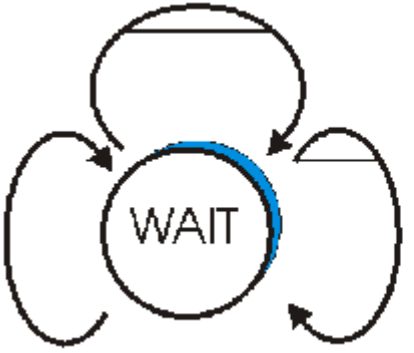

```
rdt_send(data)

if (nextseqnum < base+N) {
   compute chksum
   make_pkt(sndpkt(nextseqnum),nextseqnum,data,chksum)
   udt_send(sndpkt(nextseqnum))
   if (base == nextseqnum)
      start_timer
   nextseqnum = nextseqnum + 1
   }
else
   refuse_data(data)
```

Go-Back-N: Erweiterte FSM des Senders – Timeout



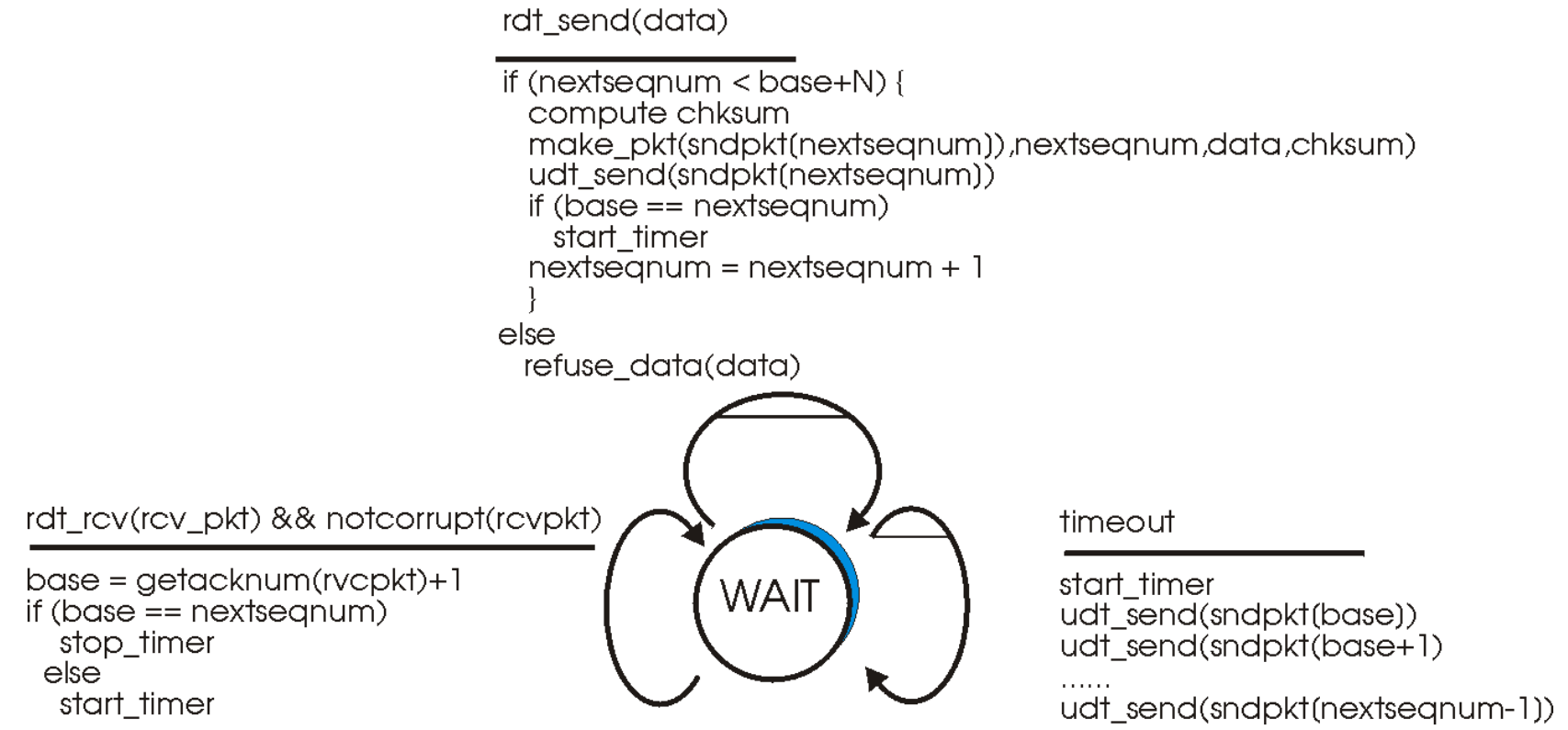




timeout

start_timer
udt_send(sndpkt(base))
udt_send(sndpkt(base+1)
.....
udt_send(sndpkt(nextseqnum-1))

Go-Back-N:
Erweiterte FSM des Senders



Für alle Additionsoperationen gilt: mod N (Window Size)

Go-Back-N:
Erweiterte FSM des Empfängers

Paket nicht korrekt

oder außerhalb der Reihenfolge:

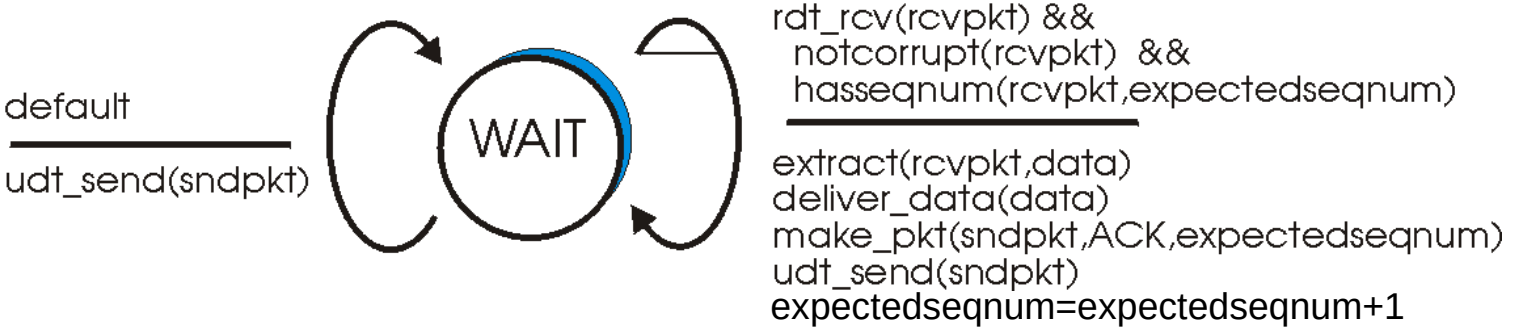
- Verwerfen des Pakets
  - kein Puffer auf Seiten des Empfängers!
- ACK für das Paket mit der höchsten Sequenznummer in richtiger Reihenfolge – letztes korrektes Paket – senden
  - Empfänger kann dadurch Duplikat – ACKs produzieren

Go-Back-N:
Erweiterte FSM des Empfängers

Paket korrekt und innerhalb der Reihenfolge:

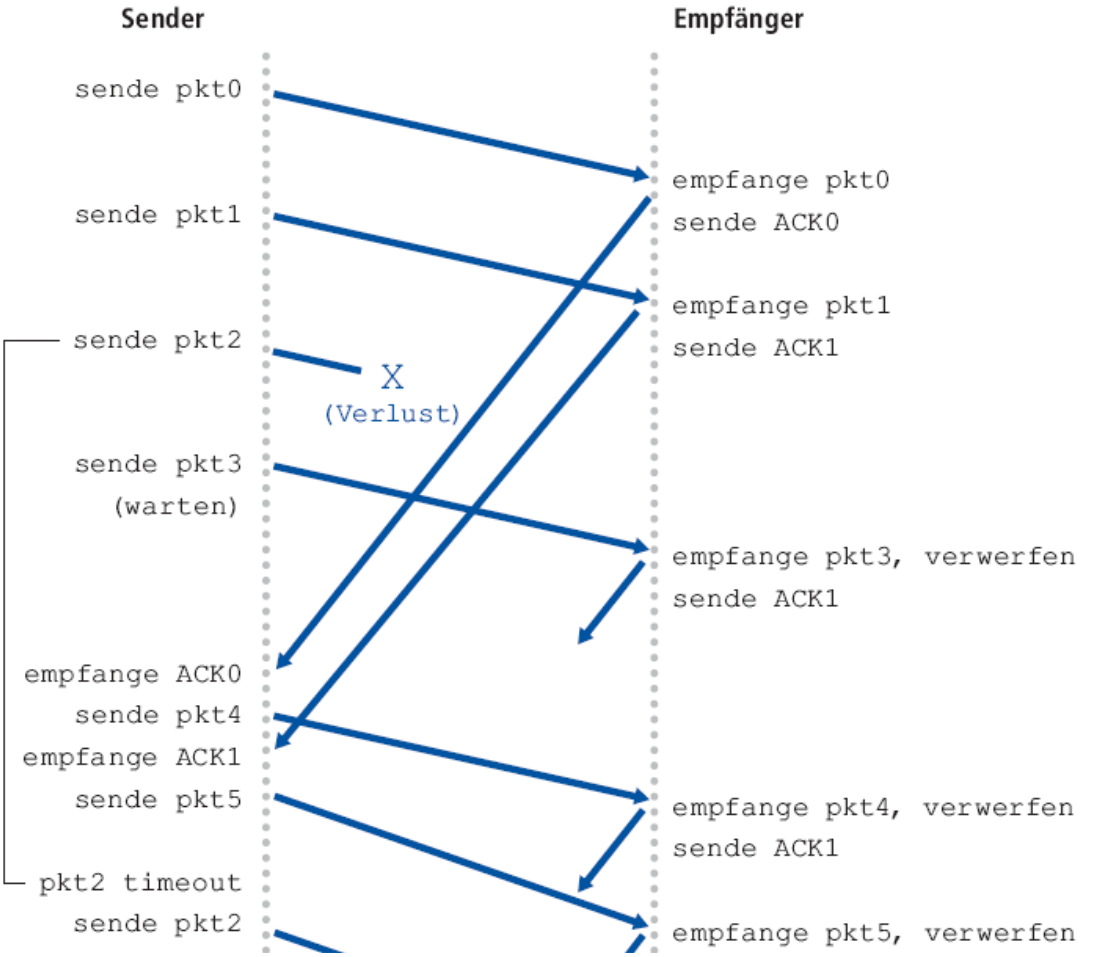
- Sende ACK für das empfangene Paket
- Erhöhe `expectedseqnum`

ansonsten schicke das letzte ACK



Für alle Additionsoperationen gilt: mod N (Window Size)

Go-Back-N in Aktion



Zur Erinnerung:
Pipelining

… wird möglich, wenn der Sender Pakete unterscheiden kann, die noch „unterwegs" sind und bestätigt werden müssen

- Bereich der Sequenznummern muss vergrößert werden
- Puffer müssen sowohl bei Sender als auch Empfänger bereitgestellt werden

Zwei grundsätzliche Arten:

- Go-Back-N
- Selective Repeat

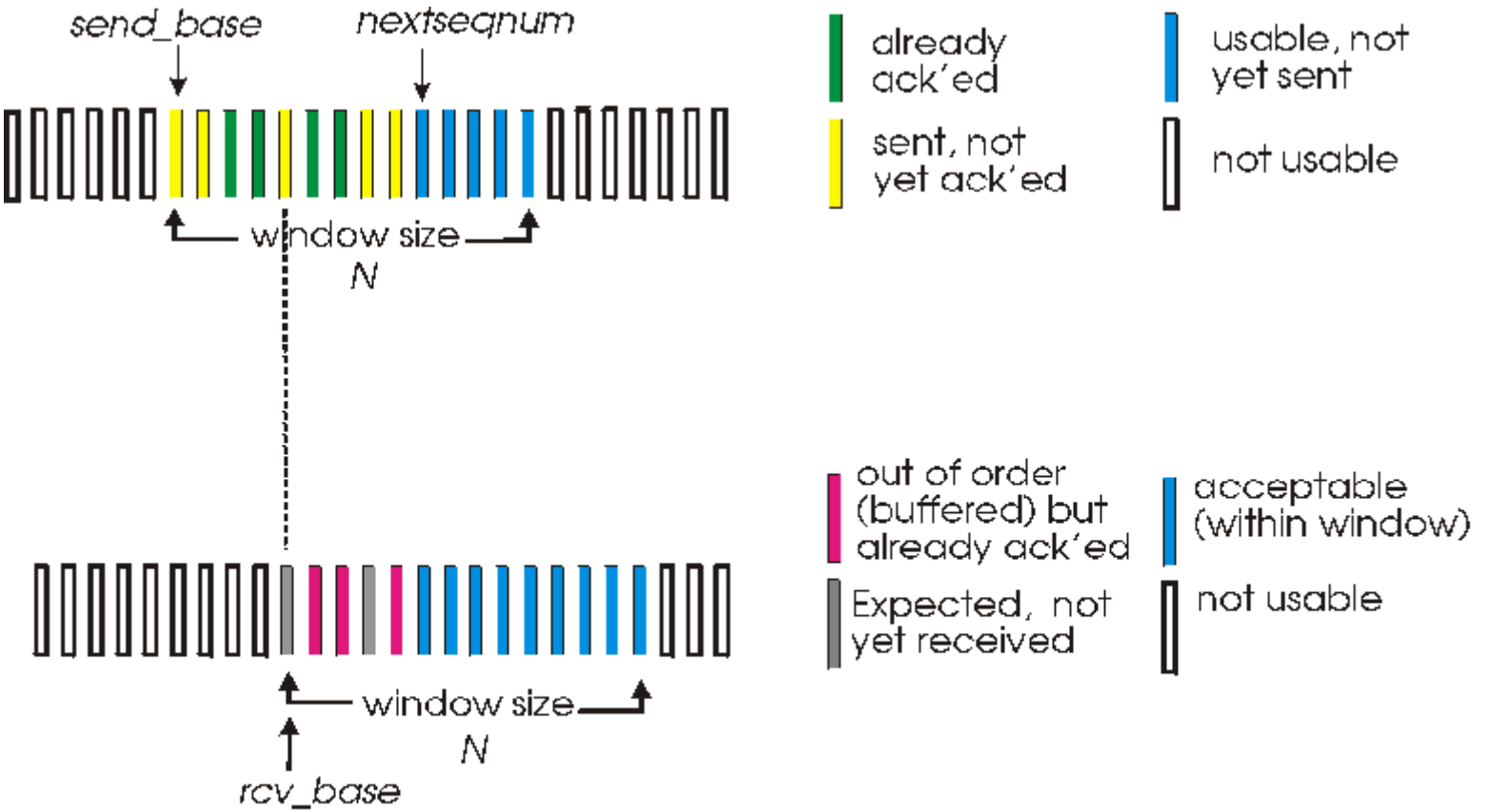
Selective Repeat

- **Empfänger bestätigt individuell alle korrekt eingegangenen Pakete**
  - Dazu werden – wenn erforderlich – Pakete zwischengespeichert, bis diese an die höhere Schicht in richtiger und lückenlosen Reihenfolge übergeben werden
  - Empfangspuffergröße = Sendepuffergröße
- **Vermeidet die eigentlich unnötigen Übertragungswiederholungen bei Go-back-N**

Selective Repeat im Detail

- **Empfänger bestätigt individuell alle korrekt eingegangenen Pakete**
- **Sender wiederholt nur die Pakete, für die er kein ACK erhält**
  - Sender braucht für jedes unbestätigte Paket einen eigenen Timer
- **Sendefenster**
  - N Pakete mit aufeinanderfolgenden Sequenznummern
  - wieder wird die Anzahl gesendeter, nicht bestätigter Sequenznummern begrenzt

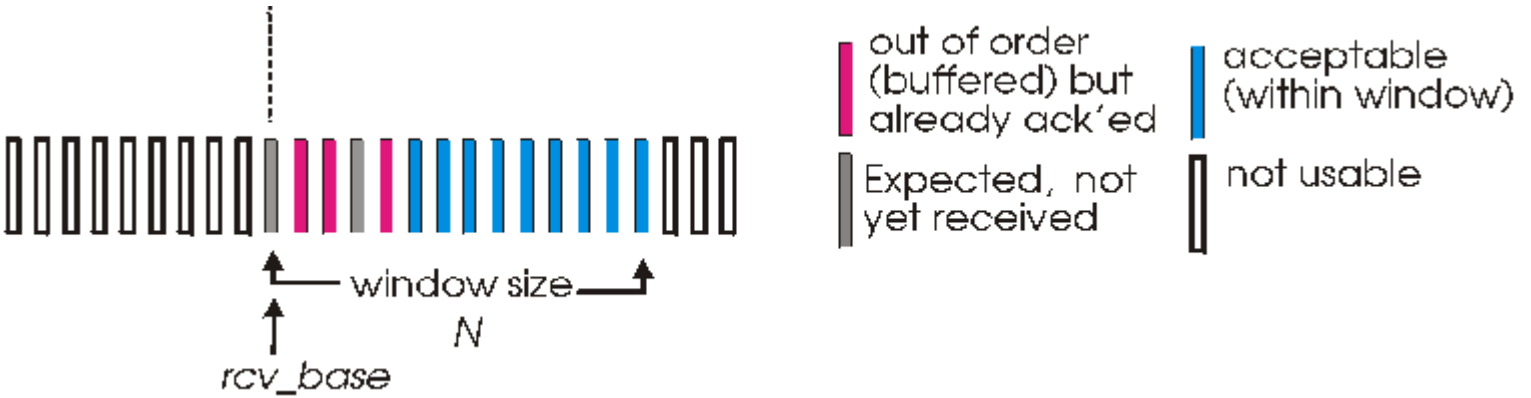
Selective Repeat: Sender / Empfängerfenster



Selective Repeat
Empfänger

Paket n in [rcvbase, rcvbase+N-1]

- sende ACK(n)
- in richtiger Reihenfolge, oder
- Paket außerhalb der Reihenfolge



Selective Repeat
Empfänger

Paket n in [rcvbase, rcvbase+N-1]

- sende ACK(n)
- in richtiger Reihenfolge:
  - Abliefern – mit allen bisher nicht gelieferten, aber gespeicherten Paketen, die dann in der richtigen Reihenfolge sind
  - Schiebe Fenster (rcvbase = n+1) auf nächstes nicht empfangenes Paket vor
- Paket außerhalb der Reihenfolge:
  - Zwischenspeichern

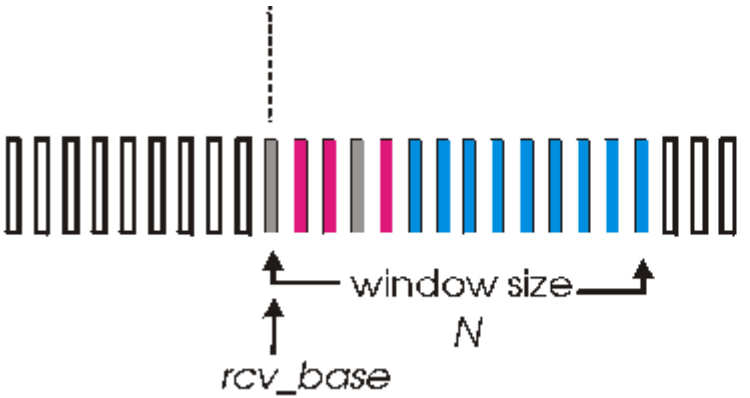
Selective Repeat
Empfänger

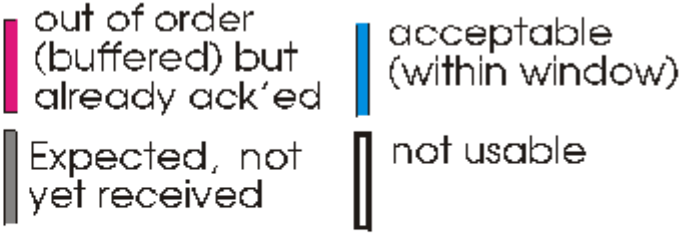
Paket n in [rcvbase-N, rcvbase-1]

- Sende ACK(n)

sonst:

- ignorieren





Selective Repeat

Sender

Daten von “oben”:

- Wenn nächste Sequenznummer im Fenster liegt, sende Paket

timeout(n):

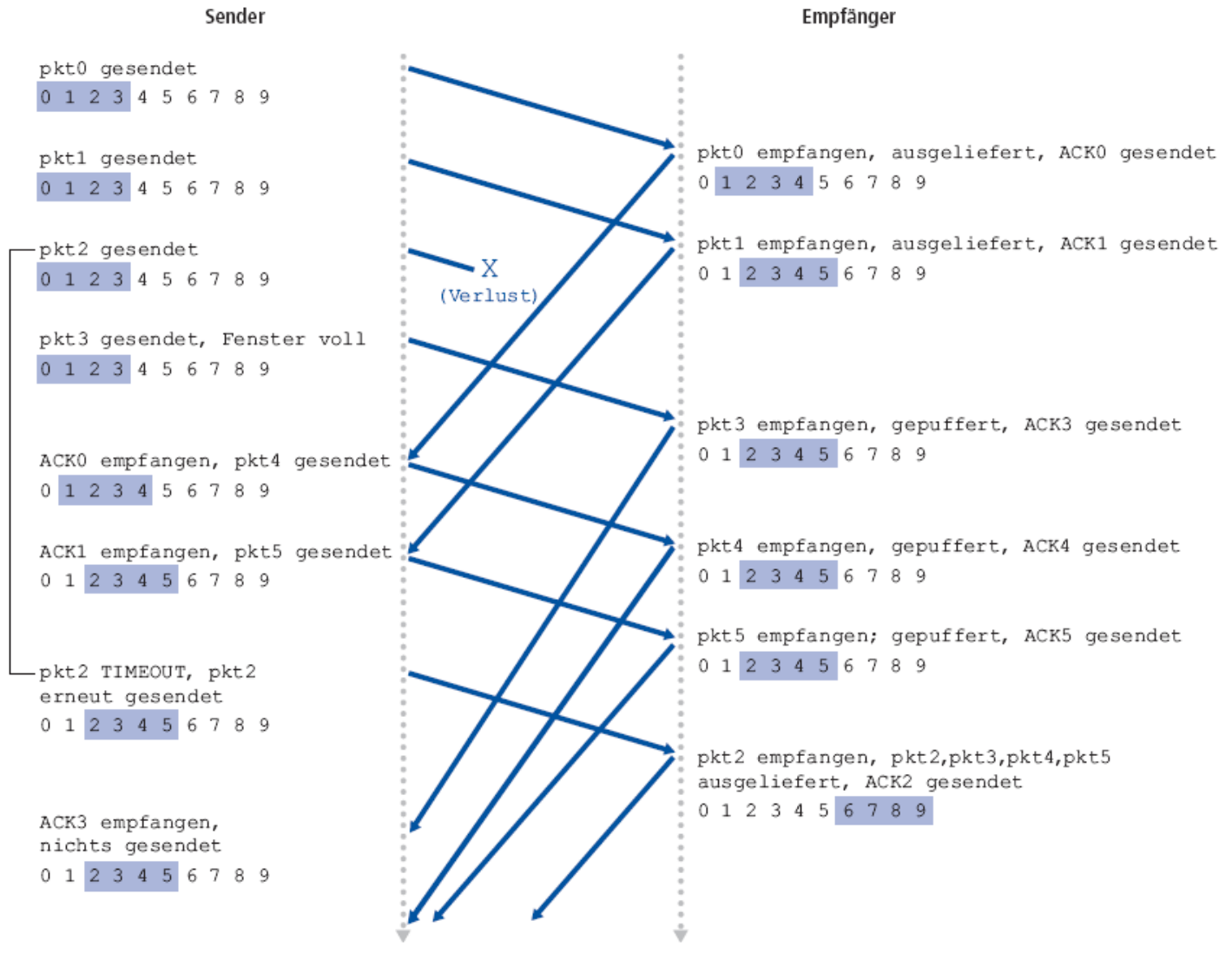
- Wiederholtes Senden von Paket n
- Timer neu starten

Selective Repeat
Sender (2)

ACK(n) in [sendbase, sendbase+N-1]:

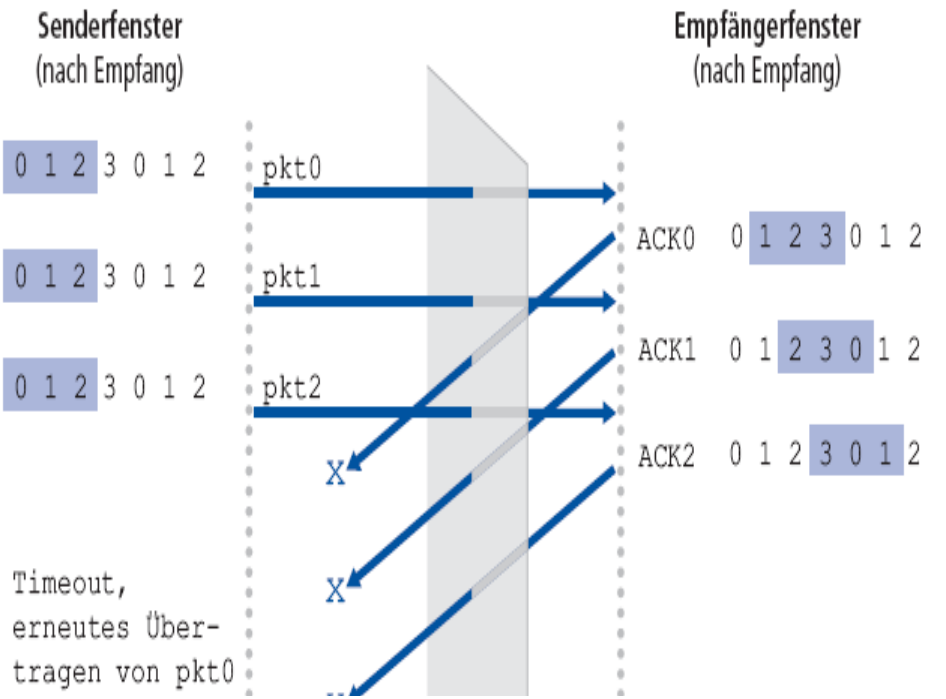
- markiere Paket n als „Empfangen“
- wenn n bisher die kleinste, nichtbestätigte Sequenznummer war, setze send_base auf die nunmehr kleinste, nicht bestätigte Sequenznummer

Selective Repeat in Aktion



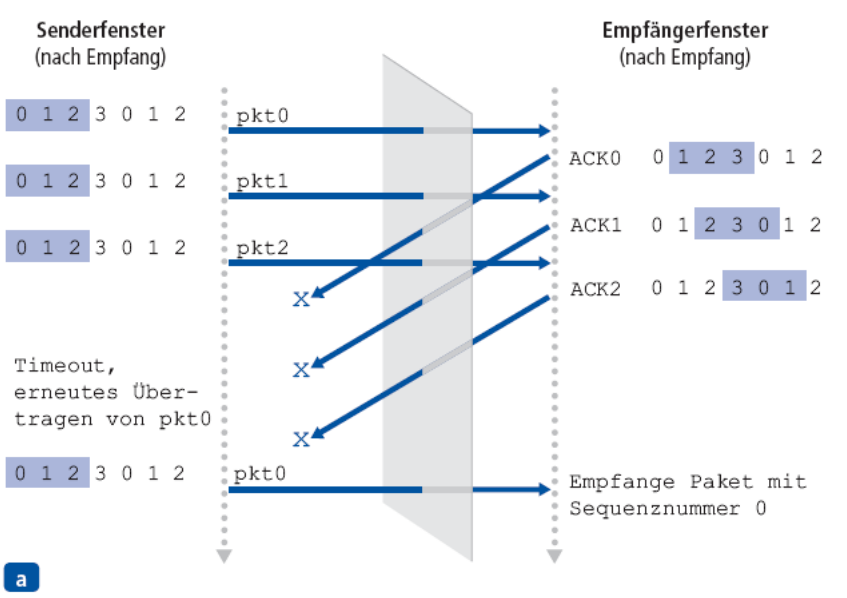
Selective Repeat:
Ist das Protokoll „narrensicher“?

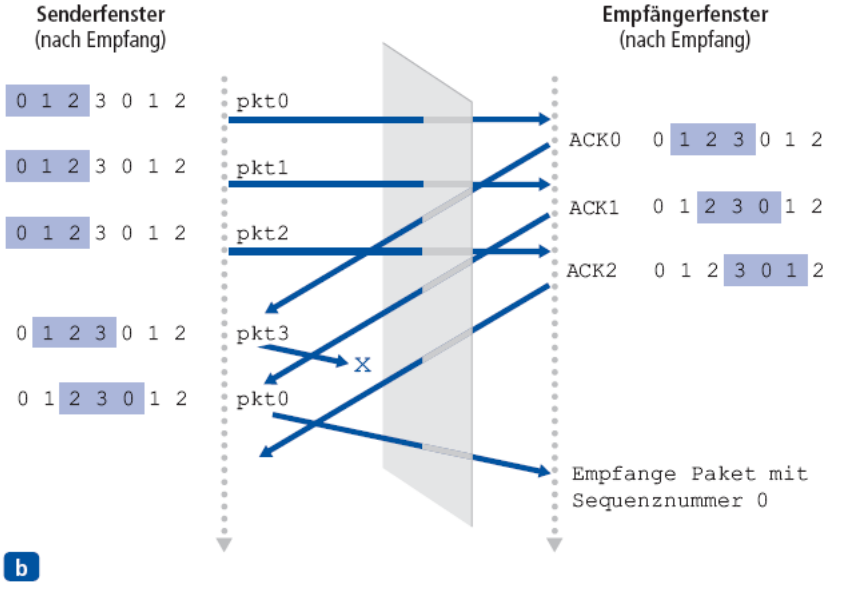
- Sequenznummer 0, 1, 2, 3
- Fenstergröße = 3



Selective Repeat: Dilemma

- Empfänger kann keinen Unterschied zwischen beiden Szenarien sehen!
- Empfänger liefert Duplikate fälschlicherweise als neue Daten ab (a)





Neuere Transportprotokolle!

Streaming Control Transmission Protocol (SCTP, RFC 2960)

- **Verbindungsorientiert, Message-orientiert:**
  - Unterstützt Nachrichten beliebiger Größe, allerdings fragmentiert
  - Kann kleine Nachrichten in einem SCTP-Paket bündeln
  - skalierbare Retransmission mit SACK
- **Ermöglicht mehrere „Streams“ für einzelne Verbindungen**
  - Stream-Eigenschaften separat definierbar
- **Unterstützt Multi-Homing sowie Erweiterungen für Mobility**

Datagram Congestion Control Protocol (DCCP, RFC 4340)

- **Protokoll für ungesicherten Transport**
  - Verbindungsorientiert
  - Entworfen für Echtzeitanwendungen
  - Implementierungen für Linux und BSD
- **Packetverluste werden entdeckt, ohne Pakete zu wiederholen**
- **Bietet den Rahmen für verschiedene Staukontrollmechanismen, z.B. Window- oder Ratenbasiert**

Zusammenfassung

Prinzipien der Zuverlässigkeit

- **Prüfsumme:**
  - Erkennen eines verfälschten Pakets beim Empfänger
- **Quittung (ACK):**
  - Rückmelden des Empfängerzustands
- **Wiederholung:**
  - Reparieren von Fehlern durch Sender
- **Sequenznummer:**
  - Entdecken von Duplikaten
    bzw. fehlenden Paketen beim Empfänger

Prinzipien der Zuverlässigkeit (2)

- **Timer:**
  - Entdecken komplett verloren gegangener Pakete beim Sender
- **Größe des Sendefensters:**
  - Anpassen der Sendegeschwindigkeit an die verfügbaren Puffer des Empfängers

    → Flusskontrolle

Kontakt

Prof. Dr. Klaus-Peter Kossakowski

Email: klaus-peter.kossakowski@haw-hamburg.de

Mobil: +49 171 5767010