Transportprotokolle

1. Protocol-Port Konzept
2. Socket Programmierung
3. User Datagram Protocol (UDP)
4. Transmission Control Protocol (TCP)
   1. Verbindungsmanagement
   2. Sicherung
   3. Flußkontrolle, Staukontrolle
   4. Optimierungen
5. Neuere Entwicklungen

Zum Inhalt

In diesem Abschnitt werden wir uns den Transport-protokollen im Internet widmen. Ihre Wahl und Eigenschaften werden direkt aus der Anwendungs-entwicklung heraus angesprochen. Deshalb spielt die Programmierung des „Socket"-Interfaces hier eine besondere Rolle.

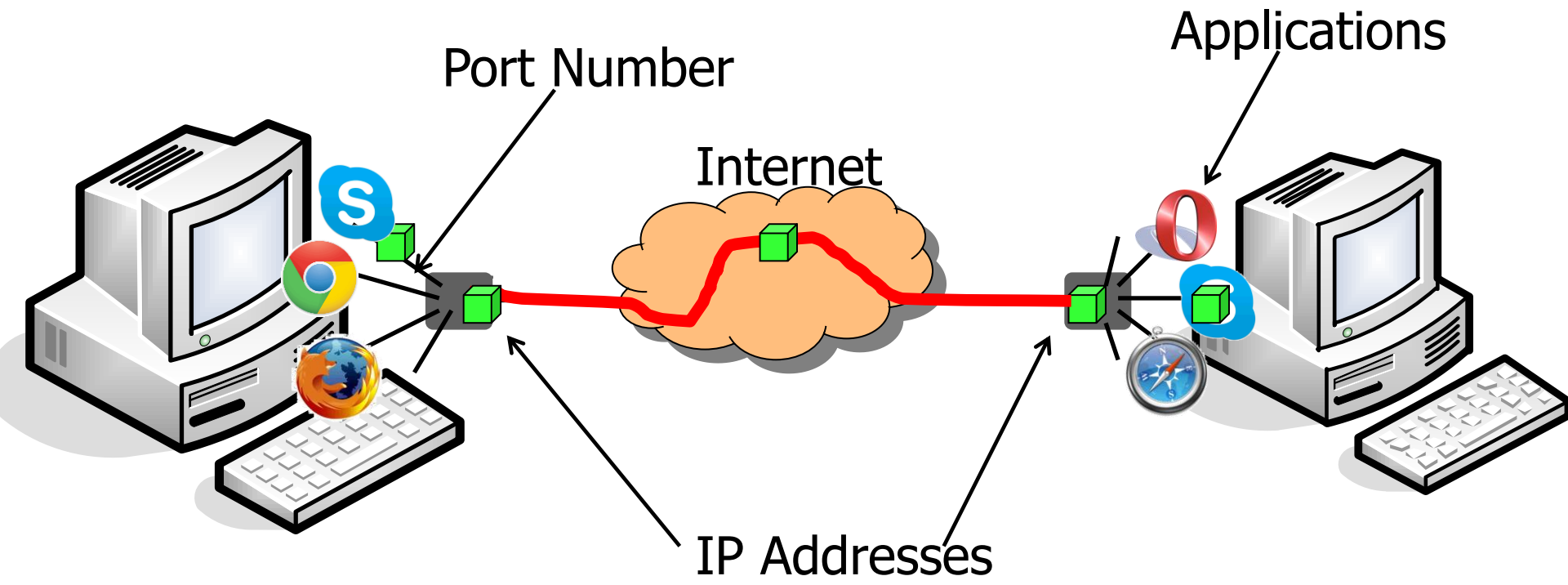
Das zugehörige Kapitel im Tanenbaum ist 6, im Meinel/Sack ist es 7 – aber für die Socket-Programmierung bitte Tanenbaum überspringen. Besser: Herbert Wiese: *Das neue Internetprotokoll IPv6*, Hanser, 2002

1. Protokoll-Port-Konzept

Wie werden Kommunikationsprozesse auf einem Rechner identifiziert?

- Direkte Prozessadressierung ist problematisch:
  - Prozesslogik ist betriebssystemabhängig
  - Ziel der Adressierung ist ein Dienst, nicht der Prozeß
  - Prozesse können mehr als einen Dienst anbieten
- Lösung: Verwendung von abstrakten Protokoll Ports
  - Betriebssystem stellt Spezifikation und Zugriff
  - Protokoll-Software (stack) synchronisiert den Zugriff mittels Warteschlangen und Belegungen

1. Multiplexen und Demultiplexen



- IP stellt den Pfad von Quell- zu Zielrechner bereit
- Die Transportschicht multiplext lokal mithilfe der Ports

1. Netzwerkports

```
atlantis %>netstat -a (Auszug)

TCP
   Local Address         Remote Address    Swind Send-Q Rwind Recv-Q  State
-------------------- -------------------- ----- ------ ----- ------ -------
      *.*                  *.*                0      0     0      0 IDLE
      *.sunrpc             *.*                0      0     0      0 LISTEN
      *.*                  *.*                0      0     0      0 IDLE
      *.32771              *.*                0      0     0      0 LISTEN
      *.ftp                *.*                0      0     0      0 LISTEN
      *.telnet             *.*                0      0     0      0 LISTEN
      *.shell              *.*                0      0     0      0 LISTEN
      *.login              *.*                0      0     0      0 LISTEN
      *.exec               *.*                0      0     0      0 LISTEN
      *.uucp               *.*                0      0     0      0 LISTEN
      *.finger             *.*                0      0     0      0 LISTEN
      *.time               *.*                0      0     0      0 LISTEN
      *.x-server           *.*                0      0     0      0 LISTEN
   sol01.rz.fhtw-berlin.de.1021 amor.rz.fhtw-berlin.de.nfsd  8760      0  8760
                                                                    0 ESTABLISHED
localhost.32785      localhost.32783      32768      0 32768  0 ESTABLISHED
localhost.32783      localhost.32785      32768      0 32768  0 ESTABLISHED
   sol01.rz.fhtw-berlin.de.1023 merlin.rz.fhtw-berlin.de.login 49152   0  8760
                                                                    0 ESTABLISHED
   sol01.rz.fhtw-berlin.de.x-server merlin.rz.fhtw-berlin.de.25478 49152     0
                                                          8760      0 ESTABLISHED
```

1. Zentrale Portvergabe

Zur Kommunikation miteinander müssen sich zwei Rechner auf Portnummern einigen. Hierfür gibt es eine

- Zentrale Vergabe - universal binding to well known ports festgelegt durch die IANA (port < 1024)

  Bsp:

Service	Portnummer	Protokoll
ftp-data	20	tcp
ftp	21	tcp
telnet	23	tcp
smtp	25	tcp
tftp	69	udp
finger	79	tcp
portmap	111	tcp
portmap	111	udp

- Ports < 1024 sind privilegiert (geschützt über Systemrechte)!

1. Dynamische Portvergabe

- Dynamische Zuordnung - (dynamic binding)
- Wird von Anwendungsprogrammen implementiert
- Portnummern sollten > 1024 sein
- Die Festlegung der Dienste und der zugehörigen Port-Nummern befindet sich in der Datei /etc/services
- Diese kann um eigene Definitionen erweitert werden.

1. Transportprotokolle

- IP adressiert nur Zielrechner, nicht einzelne Programme
- Damit Anwendungsprogramme Datagramme senden und empfangen können, stehen traditionell UDP und TCP als Transportprotokolle bereit (Layer 4)
- Beide verwenden das Prinzip der abstrakten Ports
- Beide können via Sockets programmiert werden

2. Programmierschnittstelle

Als Programmierschnittstelle für Software mit Kommunikations-kanälen haben sich die sog. Berkeley Sockets etabliert:

- Windows: winsock.dll
- Unix: libsocket.so, <sys/socket.h>
- Java: java.net.*

Die beiden Tripel (Internet-Adresse, Protokoll, Port) von Sender und Empfänger bilden die (eindeutigen) Kommunikationseckpunkte

Kommunikationsparameter können mit ‚getsockopt' gelesen und mit ‚setsockopt' verändert werden

2. Applikationsprogramm-Interface (API)

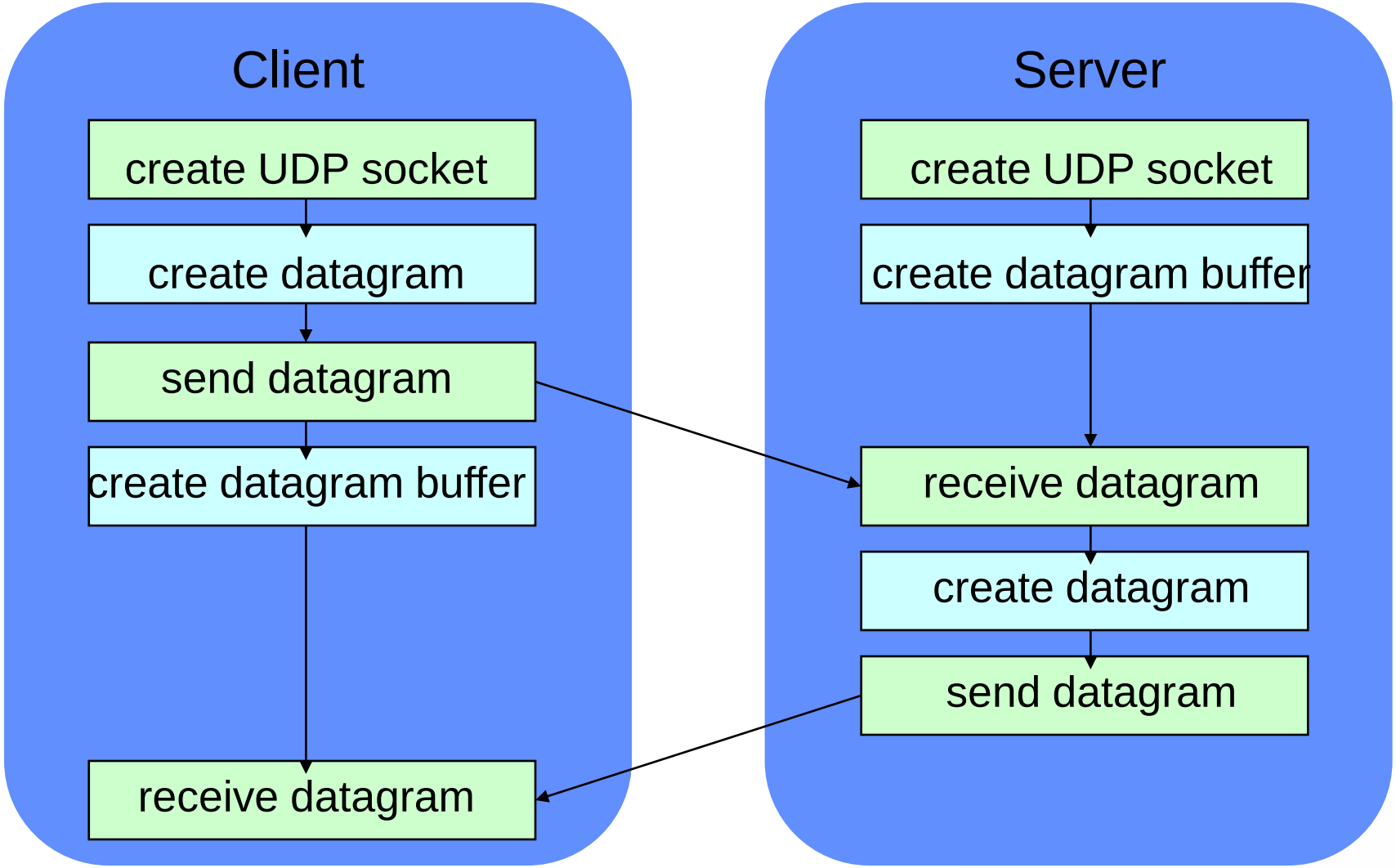
Wie können Anwendungsprogramme einfach in einer Netzwerkschnittstelle lesen und schreiben?

- Zielstellung: Nutzung wie `read` and `write`
- Probleme:
  - Netzwerkschnittstelle komplexer als Filesystem (Protokolle, -arten, Sicherheit)
  - Andersartige Kommunikationsparadigmen (C/S, Message Passing, Broadcast,…)
  - Systemübergreifende Realisierung (Vielfältige OSes, Enkodierungen & Programmiersprachen)
- Lösung: Standardisierte Netzwerk-API zur Bedienung von Standardprotokollen - Berkeley Sockets und SystemV Transport Layer Interface (TLI)

2. Sockets

- Ursprünglich Netzwerk-API von BSD 4.3 (Unix)
- Seit Jahren am meisten verbreiteter Programmierstandard (C, C++, Java, …)
- Typen:
  - Stream (`SOCK_STREAM`) für TCP
  - Datagram (`SOCK_DGRAM`) für UDP
  - Raw (`SOCK_RAW`) für IP & ICMP
- Erstellung: `s = socket(domain, type, protocol)`

2. Verbindungslose Kommunikation: Struktur eines UDP-Programms



2. Beispiel: UDP-Client in Java

```
import java.net.*;
import java.io.*;
public class UDPClient{
    public static void main(String args[]){
        // args give message contents and server hostname
          try {
                aSocket = new DatagramSocket();
                byte [] m = args[0].getBytes();
                InetAddress aHost = InetAddress.getByName(args[1]);
                int serverPort = 6789;
                DatagramPacket request = new DatagramPacket(m, args[0].length(),
                                                            aHost, serverPort);
                aSocket.send(request);
                byte[] buffer = new byte[1000];
                DatagramPacket reply = new DatagramPacket(buffer, buffer.length);

                aSocket.receive(reply);
                System.out.println("Reply: " + new String(reply.getData()));
                aSocket.close();
          }catch (SocketException e){System.out.println("Socket: " +
e.getMessage());
          }catch (IOException e){System.out.println("IO: " + e.getMessage());}}
    }
}
```

Sendet Nachricht an Server

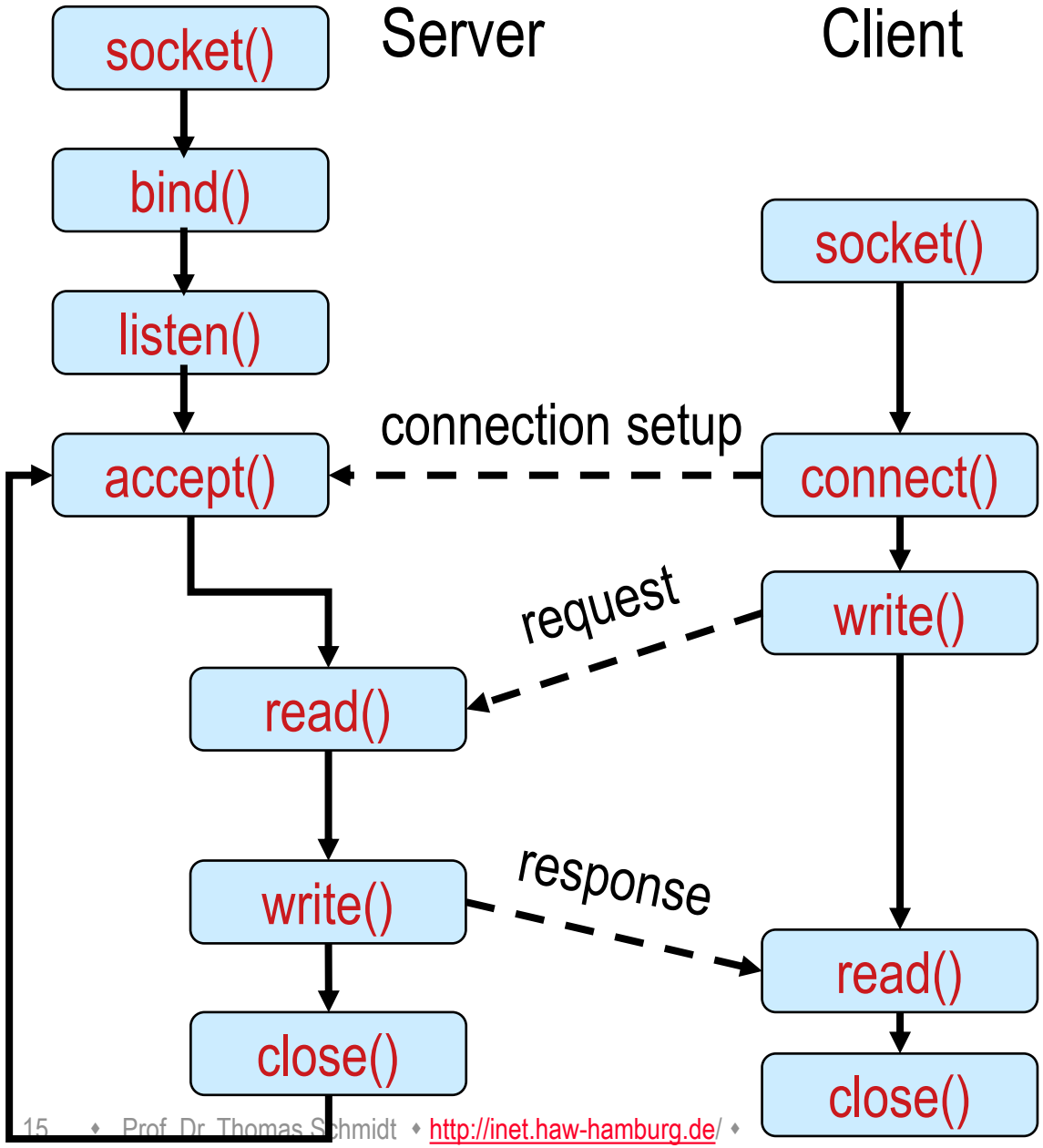
Empfängt Antwort vom Server

2. Beispiel: UDP-Server in Java

```
import java.net.*;
import java.io.*;
public class UDPServer{
        public static void main(String args[]){
            try{
                aSocket = new DatagramSocket(6789);
                byte[] buffer = new byte[1000];
                while(true){
                    DatagramPacket request = new DatagramPacket(buffer, buffer.length);
                    aSocket.receive(request);
                    DatagramPacket reply = new DatagramPacket(request.getData(),
                                                              request.getLength(),
                                                              request.getAddress(),
                                                              request.getPort());
                    aSocket.send(reply);
                }
             }catch (SocketException e){System.out.println("Socket: " + e.getMessage());
            }catch (IOException e) {System.out.println("IO: " + e.getMessage());}
    }
}
```

Empfängt Nachricht von Client

Sendet Antwort an Client



Verbindungs-orientierte Kommunikation: Call-Sequenz für TCP

2. Wichtige Funktionen (C)

- int socket (int af, int type, int protocol);
- int bind (int s, const struct sockaddr *addr, socklen_t addrlen);
- int listen(int s, int backlog);
- int accept(int s, struct sockaddr *addr, socklen_t *addrlen);
- int connect(int s, const struct sockaddr *serv_addr, socklen_t addrlen);
- ssize_t send(int s, const void *buf, size_t len, int flags);
- ssize_t recv(int s, void *buf, size_t len, int flags);
- int close(int s);
- int set/getsockopt(int s, int level, int optname, const void *optval, socklen_t optlen);

2. IPv4 → IPv6 Koexistenz – RCF 3493: Versionsübergreifende Adress-API

	IPv4	IPv6	
Data structures	AF_INET	AF_INET6	
	in_addr sockaddr_in	in6_addr sockaddr_in6	
Address conversion functions	inet_aton() inet_addr()	inet_pton()	IPv4 and IPv6 functions
	inet_ntoa()	inet_ntop()	
Name-to-address functions	gethostbyname() gethostbyaddr()	getnameinfo() * getaddrinfo() *	

*** POSIX protocol independant functions**

2. Benutzung der Koexistenz-API

Problem: `sockaddr_in` und `sockaddr_in6` sind inkompatible Datenstrukturen ...

Lösung:

- Eine Indirektionsstruktur `addrinfo` erlaubt den transparenten Zugriff auf die (versionsabhängigen) `sockaddr*` Strukturen
- Diese werden automatisch gefüllt durch `getaddrinfo`: liefert im `result` einen Pointer auf eine verkettete Liste von `addrinfo` Adressstrukturen.
- Um einen erfolgreichen Kommunikationsweg zu finden, müssen die Adressen der Liste ausprobiert werden.
- Zusätzlich ist `sockaddr_storage` eine versionsübergreifende Datenstruktur, die gem. Versionen gecasted werden kann.

2. Versionsneutrale Programmierung

Client Seite:

- Der Client muss eine IP-Version wählen, die der Server versteht. Hierzu nutzt er den DNS (`getaddrinfo`), ggfs. Anwendereingaben bzw. probiert die Versionen durch.
- In der Regel wählt der Client also zwischen mehreren Adressoptionen des Servers zur Kommunikationsaufnahme.

Server Seite:

- Der Server muss in allen IP-Versionen ansprechbar sein. Hierfür benötigt er eine Adressstruktur, die für alle Adressen geeignet ist (`sockaddr_storage`). Gegenwärtig ist diese identisch mit `sockaddr_in6` und IPv4 wird eingebettet.
- Der Server antwortet in derselben Version, in der er angesprochen wurde.

2. Programmbeispiele

In der Vorlesung

client.c **server.c**

2. Für richtige Programmierer

- Namen/Adressen werden dynamisch gesetzt – im Programm stehen natürlich keine ‚Magic Numbers'
- Server-Implementierungen multiplexen normalerweise mehrere Verbindungen
  - Das geht ohne Threads im *Reactive Pattern*
  - In Posix mit `Select()` - wählt Daten-Events aus Descriptor-Menge: `fd_set`
  - Performanter ist `epoll()` unter Linux
- UDP multiplext Pakete unterschiedlicher Sender. Anwendungen unterscheiden Sender mit `recvfrom()`

2. Wichtige Funktionen (JAVA)

Socket (Hostname/InetAddress, Port) /
ServerSocket (Port) / DatagramSocket (Port)

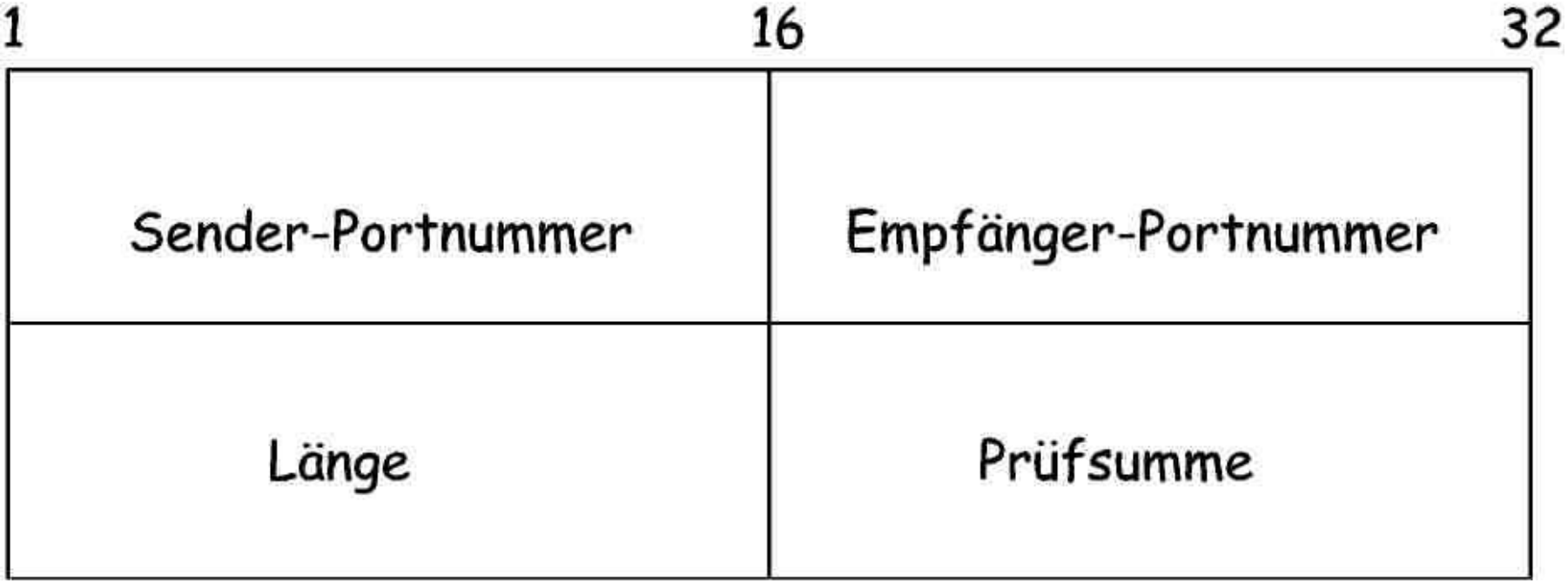
mit den Methoden:

- bind
- connect/accept
- close
- getInputStream/getOutputStream

3. User Datagram Protocol

- UDP (RFC 768) ist ein ungesicherter, verbindungsloser Transportdienst
- UDP verarbeitet Daten paket- oder message-weise
- Es besitzt eine optionale Checksumme für transferierte Daten
- UDP unterstützt das Multiplexing zwischen verschiedenen Anwendungsprogrammen auf einem Rechner
- UDP besitzt minimalen Overhead
- UDP veranlasst selbst keine Paketwiederholungen
- UDP kennt keine Mechanismen der Flusskontrolle

3. UDP Datagramm



Länge	Anzahl der Bytes des gesamten Datagramms
Prüfsumme	über Pseudo-Header und Datenteil (optional in IPv4)

3. Pseudo-Header

- Die Checksum in UDP (wie in TCP) wird als 1-er Kompliment der Summe der 1-er Komplimente aller 16-bit Worte berechnet über den UDP Header und die Daten, mit vorangestelltem Pseudo-Header:

Source address (IP)		
Destination address (IP)		
00000000	Protocol = 17	Length of the UDP datagram

3. Anmerkung zum Pseudo-Header

- Checksum bewirkt eine sogenannte layering violation
- In der Transportschicht werden IP bits verarbeitet
- Ursprünglich kein Problem, da diese IP-Informationen auf der Transportschicht zur Verfügung standen
- Heute ein Problem für NATs, insbesondere wenn Datagramme fragmentiert werden.

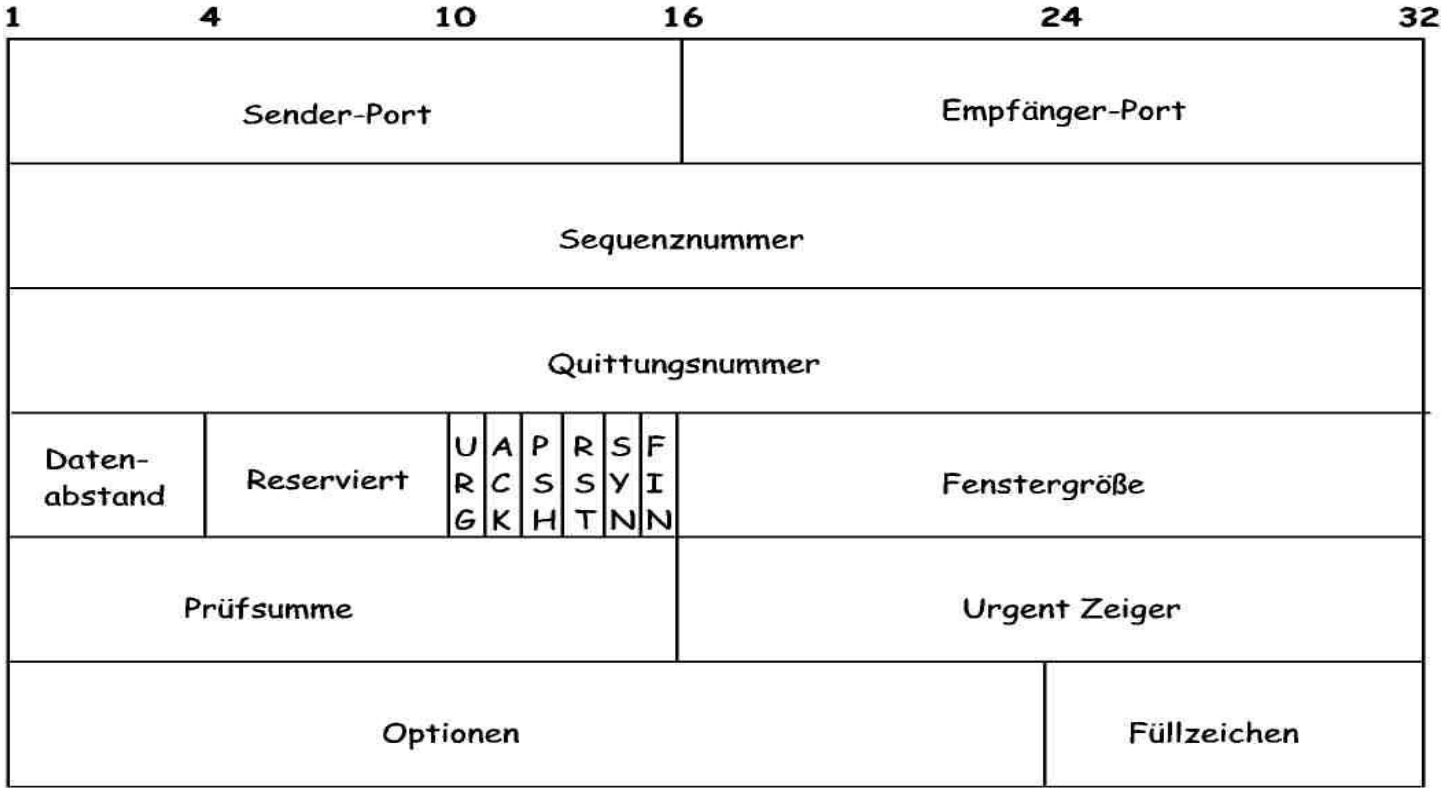
4. Transmission Control Protocol - TCP

- TCP ist der zentrale Transportdienst im Internet
- Spezifiziert in RFC 793
- TCP stellt einen verbindungsorientierten, gesicherten Transferdienst zur Verfügung
- TCP-Pakete heißen Segmente
- TCP sendet Daten als Byte-Strom (stream oriented)
- TCP unterstützt Full Duplex

4. Eigenschaften von TCP

- Virtuelle Verbindung (virtual circuit connection)
  - verdeckt Details zu Verbindungsaufbau und -sicherung
  - erscheint dadurch wie eine direkte Hardware-Verbindung
- Jedes Segment wird (initial) in einem IP-Datagram transportiert. Daraus ergibt sich für die (konfigurierbare) TCP Maximum Segment Size MSS $\leq$ MTU - 40 Bytes
- Das Receive-TCP quittiert den Empfang von Segmenten
- Das Send-TCP veranlaßt ein erneutes Versenden, wenn die Quittung ausbleibt

4. TCP Segment



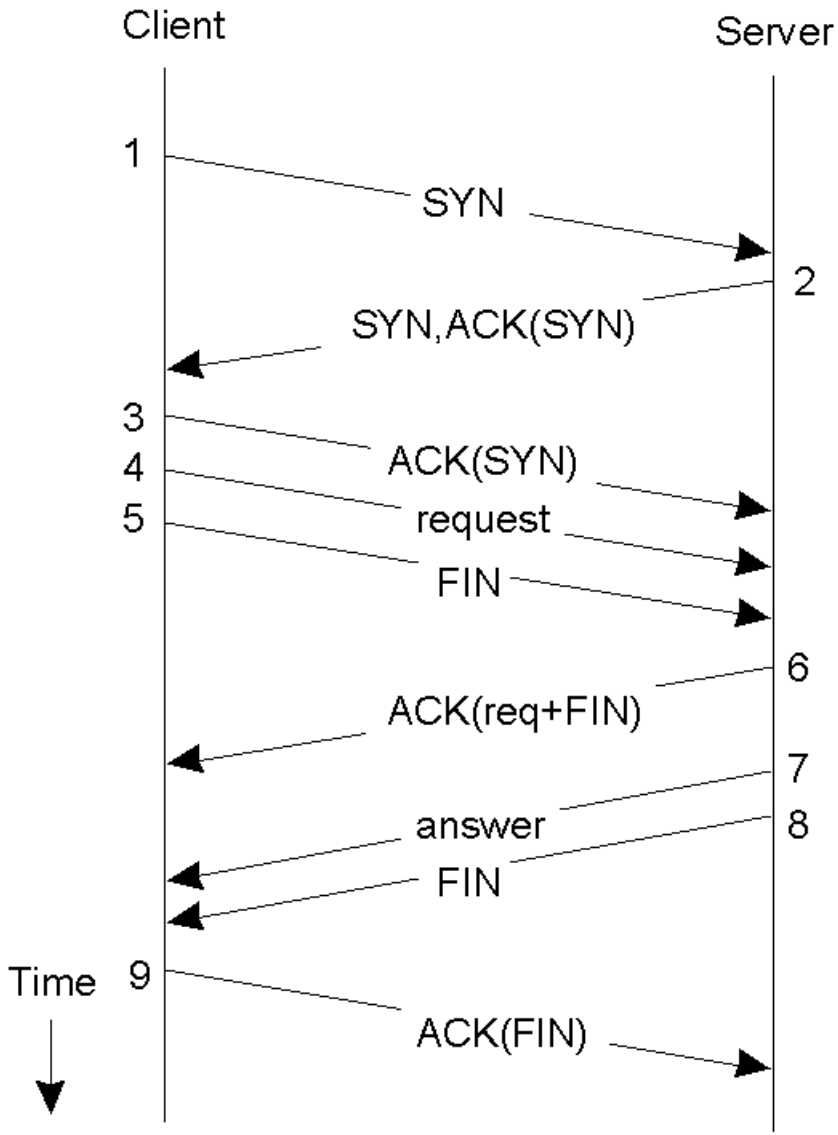
Sequenznummer	Nummer des ersten Bytes in der Sequenz
Quittungsnummer	Nummer des letzten quittierten Segment-Bytes
Datenabstand	Länge des Headers in 32-bit Worten
Fenstergröße	Anzahl der Bytes, die der Empfänger abnimmt
Prüfsumme	über Header und Datenteil (obligatorisch)

4. Coded Bits

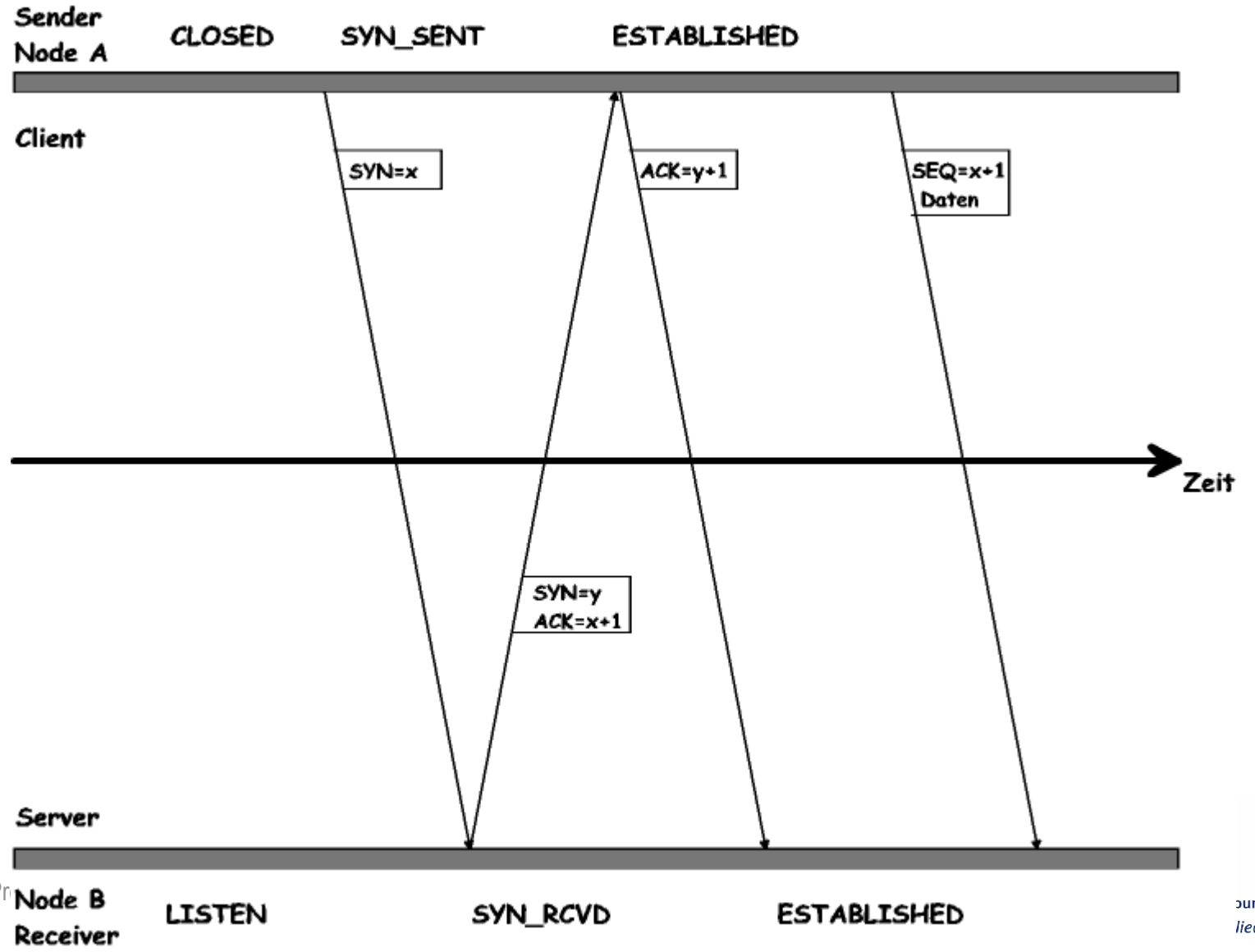
- URG - Urgent Pointer ist gültig: Setzt den Empfänger in ‚Urgent Mode' bis der Urgent Pointer passiert ist.
- ACK - Acknowledgment ist gültig
- PSH - Push Flag: ‚verarbeite Daten sofort'
- RST - Reset: Setze Verbindung zurück
- SYN - Synchronize zur Verbindungsaufnahme
- FIN - Finish zum Verbindungsabbau

4.1 TCP Verbindungsmanagement

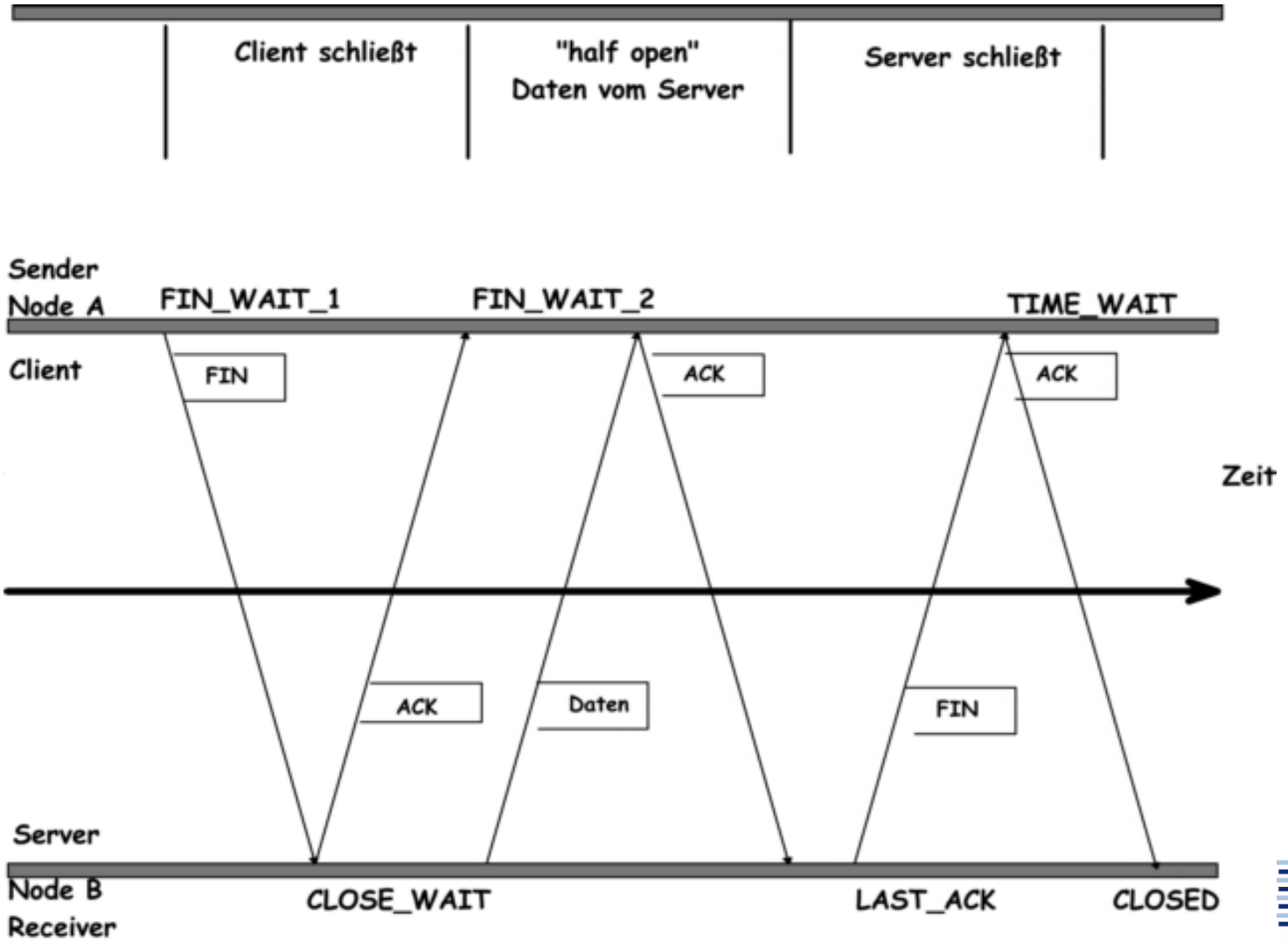
- TCP beinhaltet ein automatisches Verbindungsmanagement
- Realisiert im Client-Server Modell: Client initiiert, Server akzeptiert Verbindung
- Eine Verbindung besteht, wenn Client und Server den zugehörigen Verbindungszustand etabliert haben: Hierzu ist (mindestens) ein Drei-Wege-Handshake erforderlich



4.1 TCP Verbindungsaufbau



4.1 TCP Verbindungsabbau



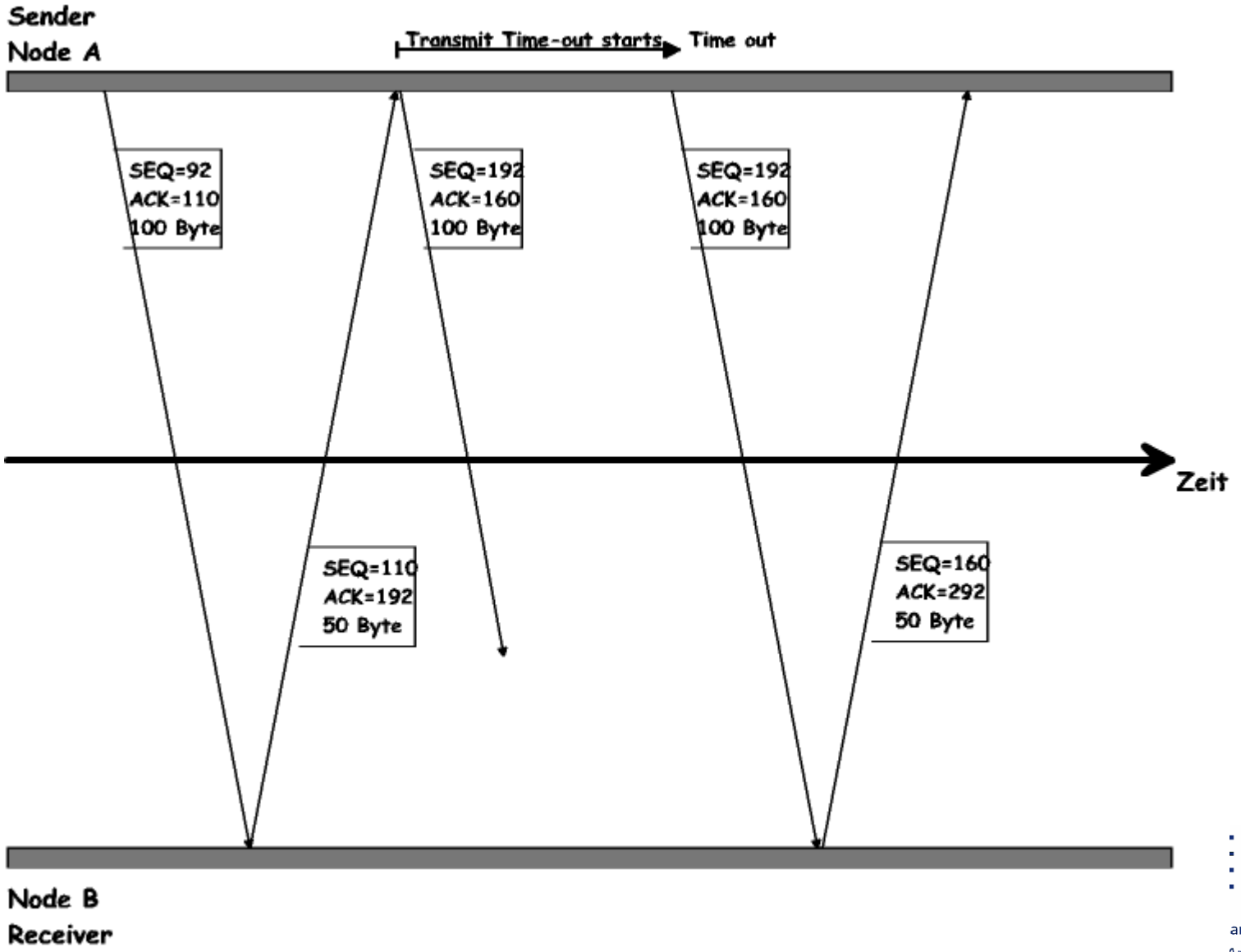
4.2 TCP Sicherung

- TCP sichert den Transport seiner Segmente so ab, dass beim Empfänger ein vollständiger, geordneter Datenstrom erhalten wird
- Fehlt ein Segment beim Empfänger, wird der Strom angehalten und auf das fehlende Paket gewartet - „Head of Line Blocking"
- Datenverluste werden an der Sequenznummer erkannt
- TCP meldet jedoch keine Verluste, sondern versendet Empfangsquittungen (ACKs)
- Dabei quittiert TCP (ursprünglich) das letzte zusammenhängende Segment – „Cumulative acknowledgement"
- Nach einem Verlust beginnt TCP (ursprünglich), vom verlorenen Segment an den Strom neu zu senden – „go back N"

4.2 Empfangsquittungen (ACK)

- TCP ordnet die empfangenen Segmente in ihre ursprüngliche Reihenfolge (gem. Sequenznummer)
- Sobald Segmente in zusammenhängender Reihenfolge eingetroffen sind, ist TCP ‚quittungsbereit'
- TCP versucht, ACK gemeinsam mit Daten zu senden (Piggybacking)
  - Sind keine Daten ‚versandfertig', wird ACK verzögert
  - Nach (typisch) wenigen ms wird ACK auch alleine versandt

4.2 Quittung & Retransmission



4.2 Retransmission

Wann eine Sendung von TCP-Messages wiederholen?

- Feste Timeouts problematisch bei variabler Verzögerung
  - zu groß: Performanceverlust
  - zu klein: unnütze Netzlast durch Wiederholungen
- Lösung: Round Trip Time = durchschnittl. Verzögerung zwischen Aussenden und Quittungsempfang beachten
- TCP ermittelt RTT für jede Verbindung
  - Retransmit Timer basiert auf der RTT und ihrer Variation
  - dennoch: Probleme bei schnell veränderlicher RTT!

4.2 Retransmit Timing

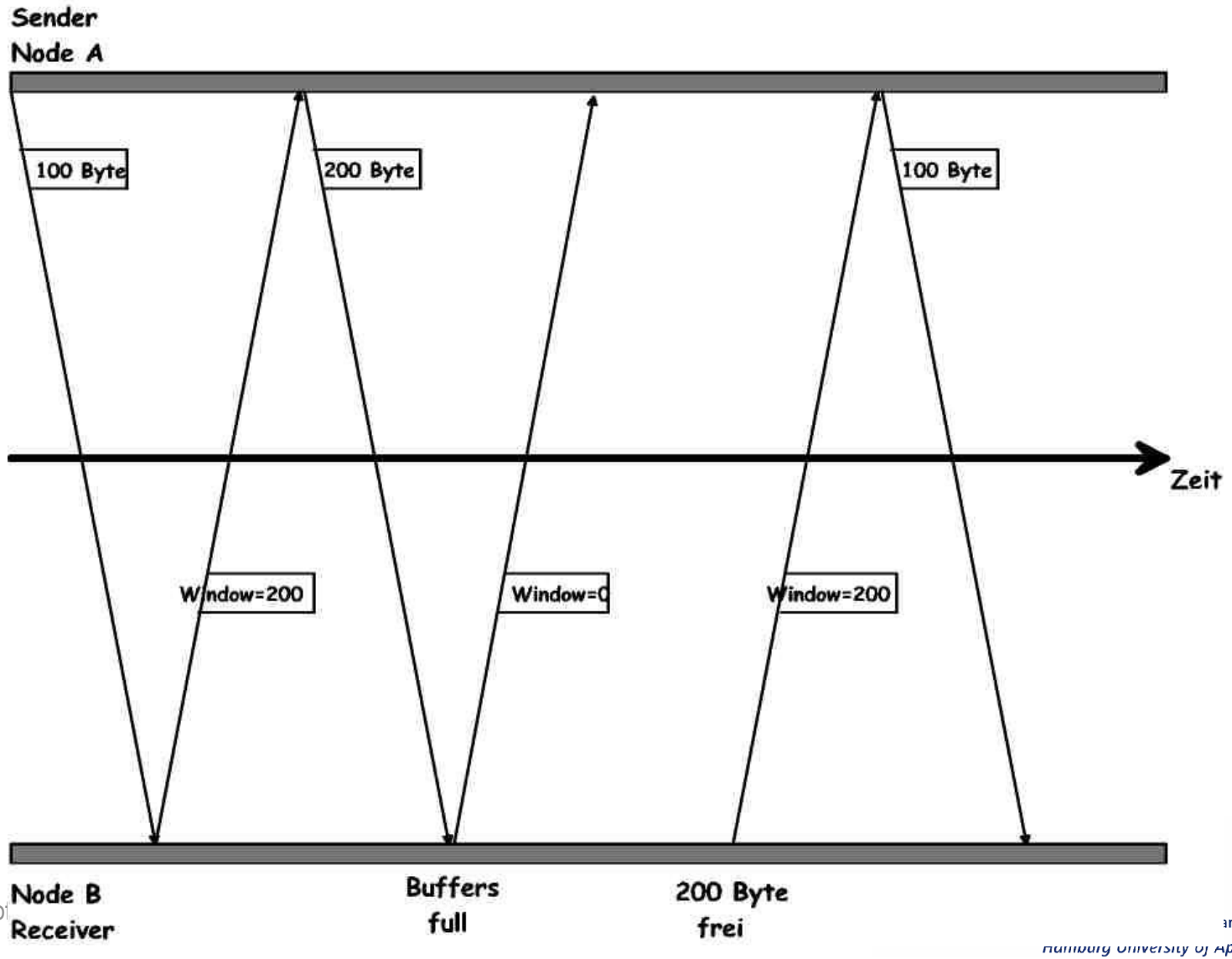
Adaptive Zeitsteuerung ist komplex:

- TCP ermittelt erwartete RTT, Delay-Variation (Jitter) und Timeout
  - Timeout = estRTT + 4 * D-V (Initialisierung)
  - Timeout-Verdopplung bei Retransmit (Exponential Backoff)
- TCP interpretiert Datenverlust (Retransmit) als Anzeichen einer Netzwerküberlastung
  - TCP wendet Strategie zur Stauvermeidung an

4.3 Flußkontrolle zum Empfänger

- Prinzip der dynamischen Flusskontrolle:
- TCP teilt den Datenfluss zur Übertragung in Segmente ein
- Der Empfänger steuert den Datenfluss durch Mitteilen der verfügbaren (Empfangs-) Puffergrößen:
  receive window (rwnd)
- Ein Fenster der Größe 0 stoppt den Fluss
- Der Empfänger kann zusätzliche ACKs schicken, um den Fluss wieder in Gang zu setzen

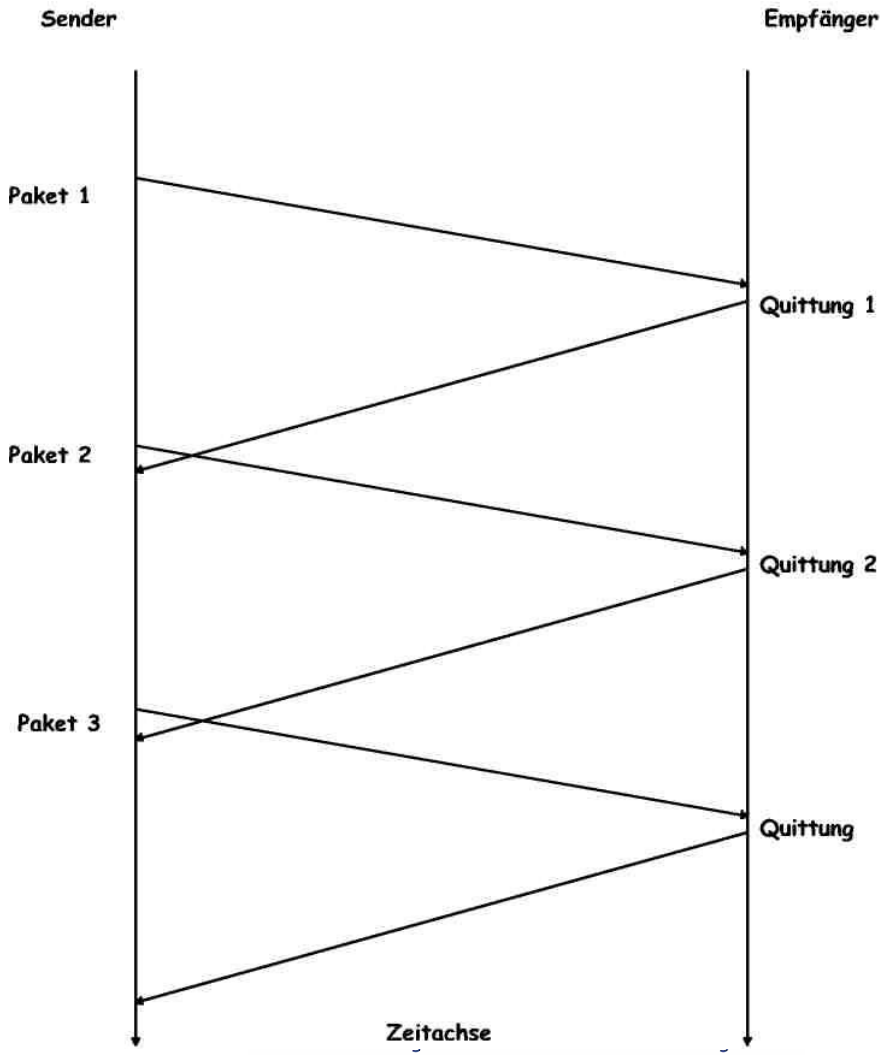
4.3 Window Advertisement



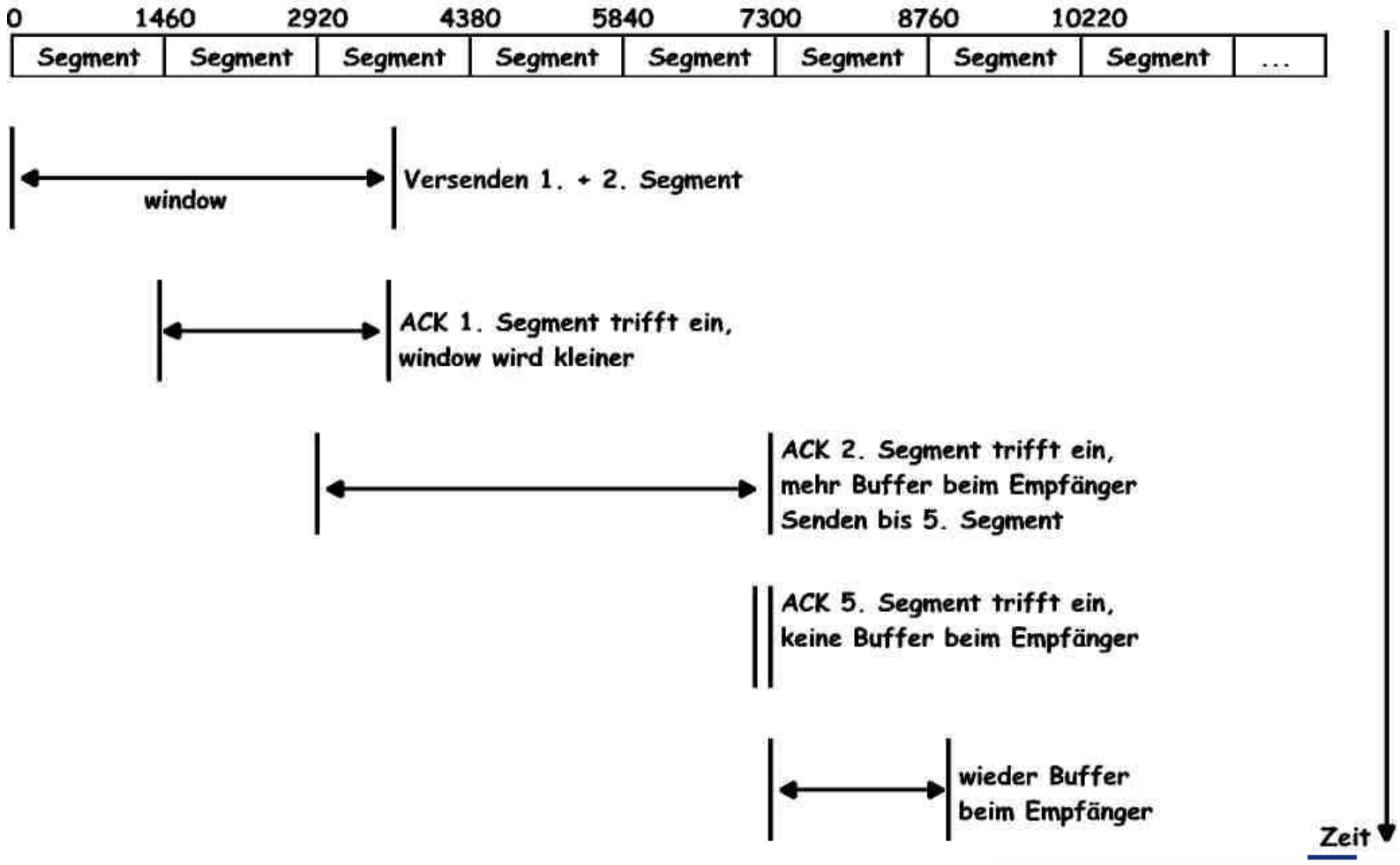
4.3 Sliding Window

Optimierung der Flusskontrolle:

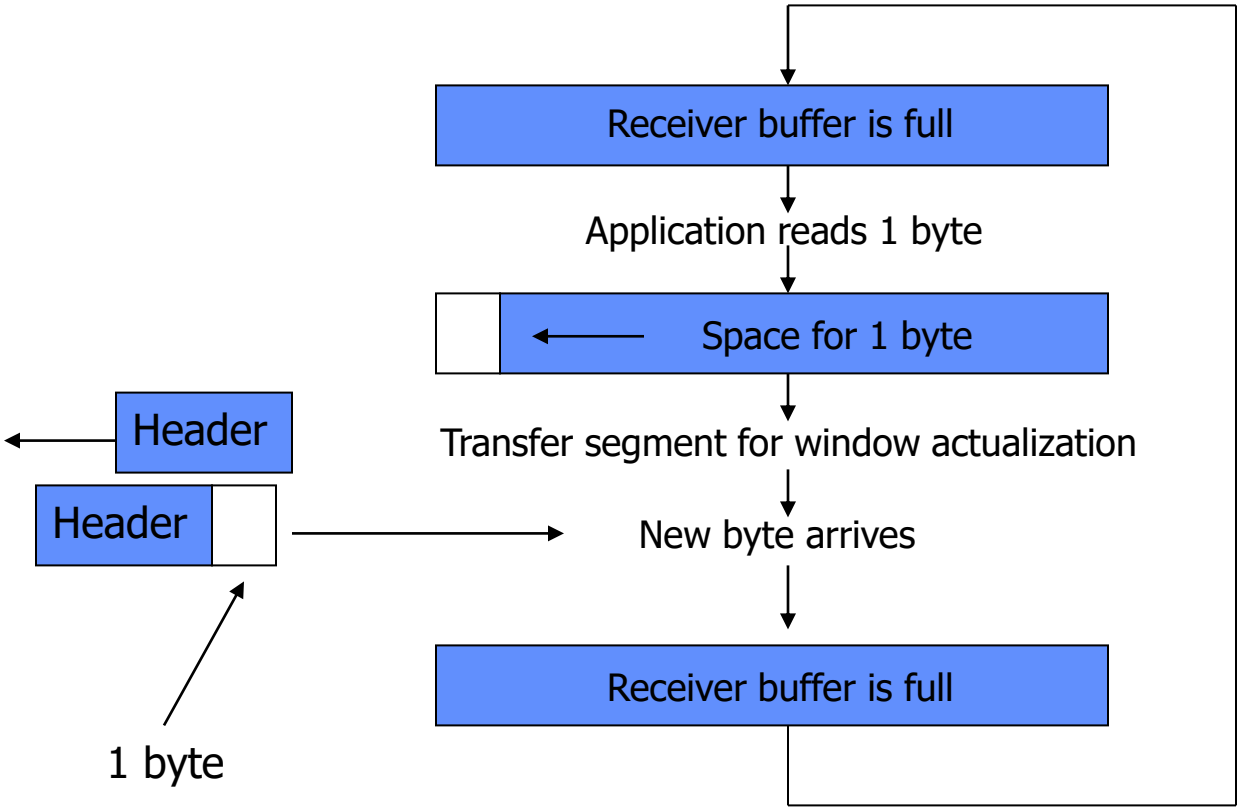
- Jeder darf die Anzahl Bytes im Window senden, ohne auf eine Quittung zu warten
- Typische default Fenstergrößen: 4096 - 16384 Bytes
- Höhere Werte insbesondere bei ‚long fat pipes‘
- Max. Fenstergröße: Receive Buffer Size
- Das Empfangsfenster wandert mit jedem ACK entsprechend weiter



4.3 Datenstrom und Datenfluss



4.3 “Silly Window” Syndrom



- Empfangsfenster in TCP haben immer mindestens die Größe eines Segments (MSS)

4.3 Stauvermeidung - V. Jacobson `88

- Wie viele Daten darf eine Quelle auf einmal senden?
  - Ursprünglich unkontrolliert → Gefahr des Netzkollaps!
- TCP operiert zur Vermeidung von Netzwerkstaus vorsichtig:
  Congestion Avoidance und einen Slow Start
  - Congestion Window (cwnd) wird beim Sender geführt
- Congestion und Receiver Window begrenzen das Senden: Allowed-window = min(cwnd, rwnd)

4.3 Stauvermeidung: Slow Start

- Welche Datenrate verkraftet das Netzwerk?
  - TCP tastet sich im Slow Start von unten an die Grenze
- Ein TCP Sender führt sein Congestion Window so:
  - Es startet mit einem Anfangswert: 1-4 Segmente
  - Per ACK erhöht sich die Fenstergröße um 1 Segment
  - Der Slow Start endet bei einem Schwellwert: ssthresh
  - ssthresh wird adaptiv an Verlustraten angepasst

4.3 Stauvermeidung

- Wie schnell soll das Congestion Window wachsen?
  - Für cwnd > ssthresh besteht Staugefahr!
- Ein TCP Sender folgt nun dem Congestion Avoidance Modus:
  - Das Congestion Window wird nun pro Round Trip Time (RTT) um 1 Segment erhöht
  - Das ist langsamer, da mehrere Segmente parallel verarbeitet werden (Sliding Window)
  - Bei Stauanzeichen (timeout) wird cwnd = 1 gesetzt und TCP fällt in den Slow Start zurück

4.3 Jacobson Fast Retransmit

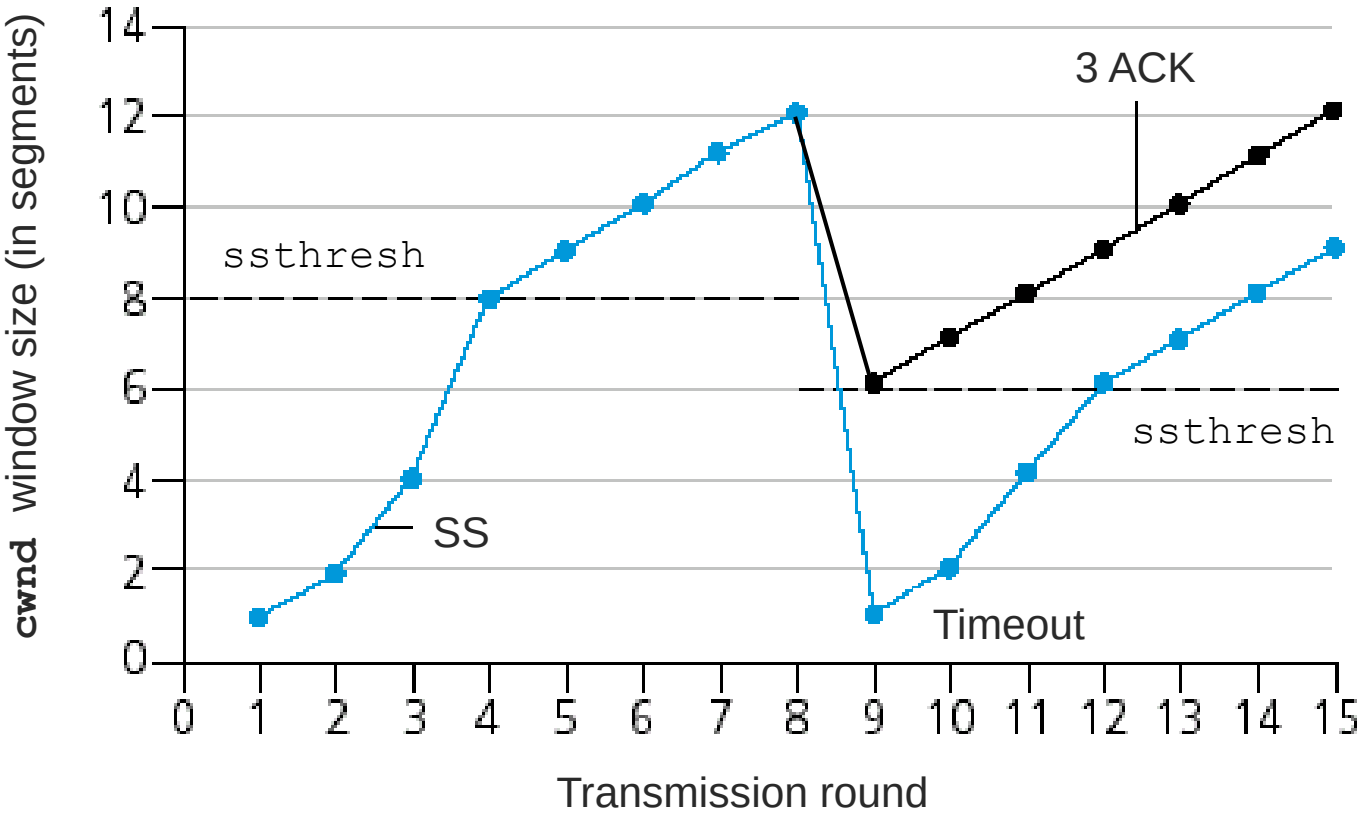
TCP quittiert empfangene Pakete kontinuierlich

- Wird innerhalb eines sliding Window ein Segment verloren, muss das ganze Fenster erneut gesendet werden (und TCP fällt in Slow Start)

Einfache Idee zur Verbesserung:

- Wenn ein ‚out-of-order' Segment eintrifft, sendet TCP eine erneute Quittung des kumulativ erhaltenen Stroms (duplicate ACK).
- Annahme: wenn mehrere gleiche ACKs eintreffen, ist wahrscheinlich nur ein Zwischensegment verloren worden.
- Fast Retransmit:
  - Wird das dritte duplicate ACK erhalten, schickt der Sender das erste nicht bestätigte Segment unverzüglich erneut.
  - Der Sender wechselt nicht in die Slow Start Prozedur, sondern auf ssthresh

4.3 TCP Leistungsverhalten



4.4 TCP Optimierungen

- TCP besteht seit seiner Erfindung unverändert ‚on the wire' – TCP kennt keine Versionsnummer
- Dennoch hat TCP kontinuierlich Optimierungen und Erweiterungen erfahren
  - Um ‚besser', d.h. effizienter und leistungsfähiger zu werden
  - Um sich den veränderten Übertragungsanforderungen anzupassen (z.B. Wireless)
  - Um die gestiegenen Endgeräte-Kapazitäten zu nutzen
- Dabei liegt die Herausforderung darin, gleichmäßig kompatibel zu bleiben

4.4 Vermeidung von Tinygrams

Einzeln versendete Datenbytes haben (auch) 40 Byte Header. Zur Vermeidung unnötiger Overheads:

- Nagle Algorithmus:
  - Sende unvollständige Segmente erst nach vollständigen ACKs
  - Während der Wartezeit werden Daten gesammelt
- Problem: Graphische Interaktion und verzögertes ACK
  - Nagle Algorithmus kann mit Socket-Option TCP_NODELAY ausgeschaltet werden.

4.4 Selective Acknowledgment

TCP quittiert empfangene Pakete nur zusammenhängend

- Werden innerhalb eines sliding Window Segmente verloren, muß das ganze Fenster erneut gesendet werden.

Komplexerer Lösungsansatz:

- Segmentbereiche (innerhalb eines sliding Window) können diskontinuierlich quittiert werden (SACK)
- Der Sender hat dann die Option, zunächst die unquittierten Segmente erneut zu senden
- Erfolgt ein erneutes TIMEOUT, wird von der letzten (kumulativen) Standardquittung an erneut gesendet
- Erfolgt ein kumulatives ACK, wird der Vorgang abgebrochen

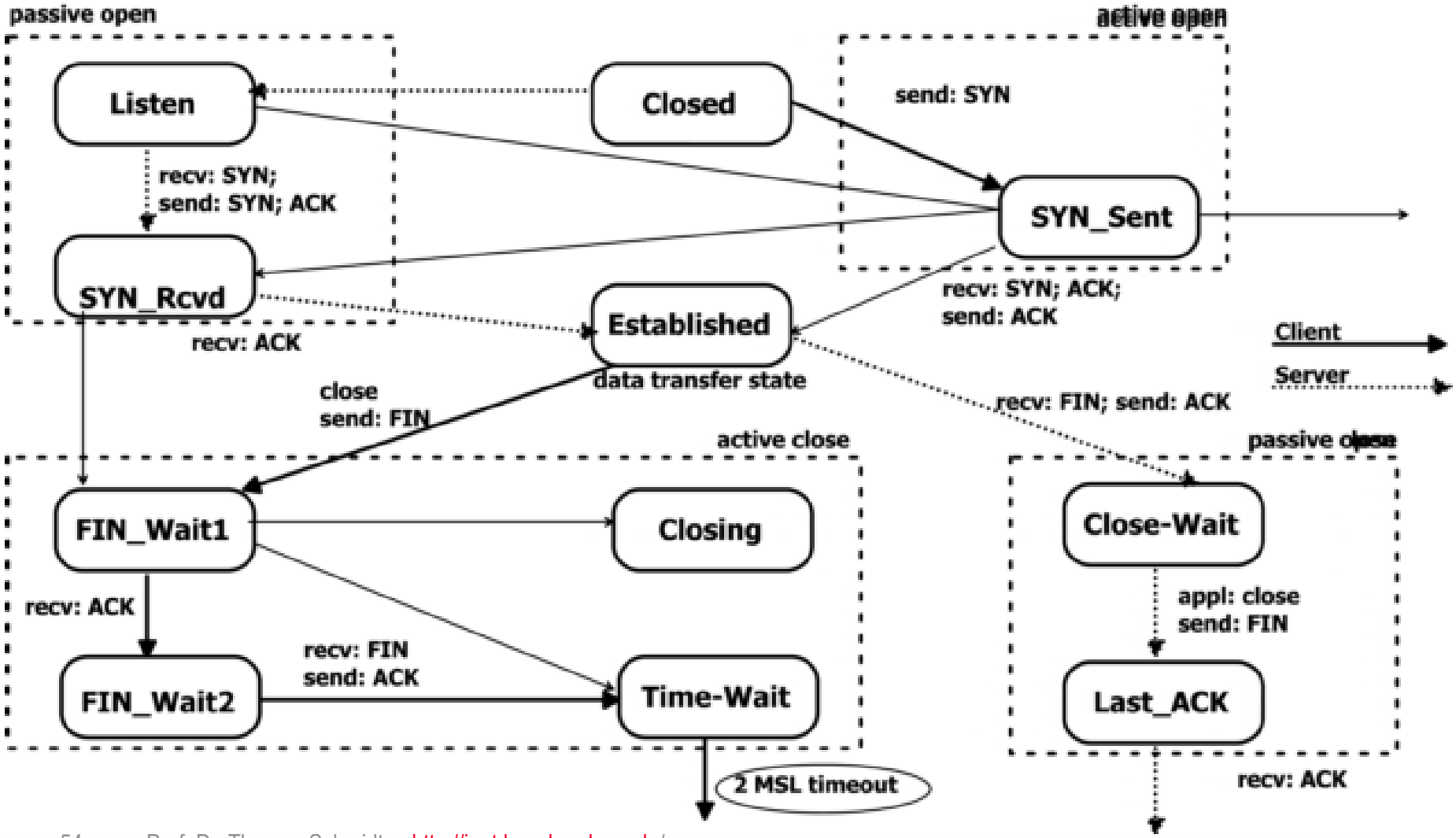
4.4 SACK

- Spezifiziert in RFC 2018
- Wird initial verhandelt bei dem Verbindungsaufbau (SYN)
- Selektive Quittungen (des Empfängers) erfolgen in den TCP-Header Options
- Der Sender muß eine separate ‚Sack'-Tabelle führen
- Der Sender ist berechtigt, auf selektive Quittung nicht zu reagieren

4. TCP Connection Timer

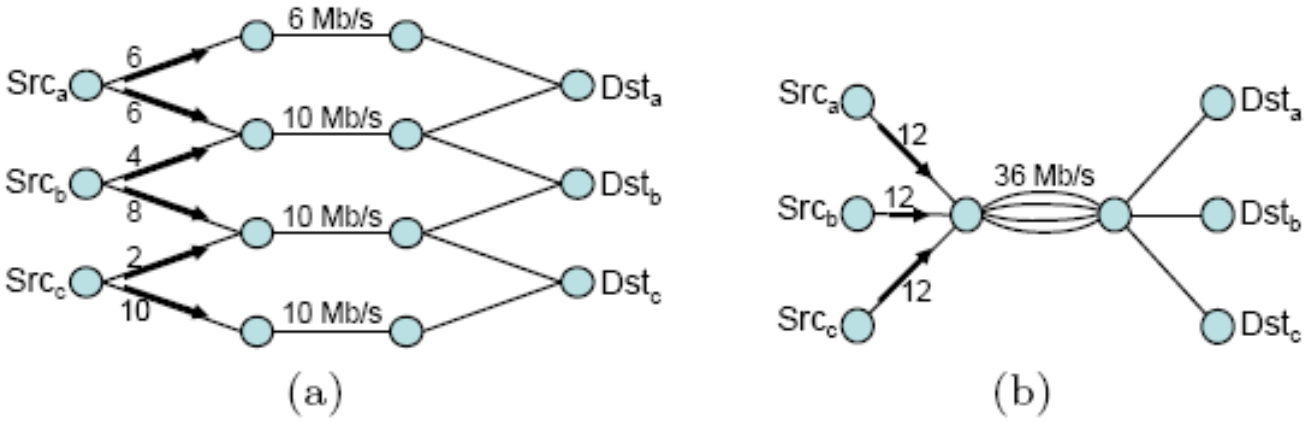
- Retransmission: Dauer, in welcher ein ACK erwartet wird.
- Persist: Dauer, nach welcher das Fortbestehen eines geschlossenen Empfangspuffers überprüft wird.
- Keepalive: Dauer, nach welcher die Verbindungsgegenstelle um ein Lebenszeichen gebeten wird.
- 2MSL: Dauer, in welcher TCP-Segmente im Netz gültig sind (Maximum Segment Lifetime).

4. TCP Zustandsdiagramm



4.5 Multipath TCP: Die Idee

- Resource Pooling zur Verbesserung von Durchsatz und Zuverlässigkeit
  - Gebündelte Ressourcen sollen als Eins erscheinen

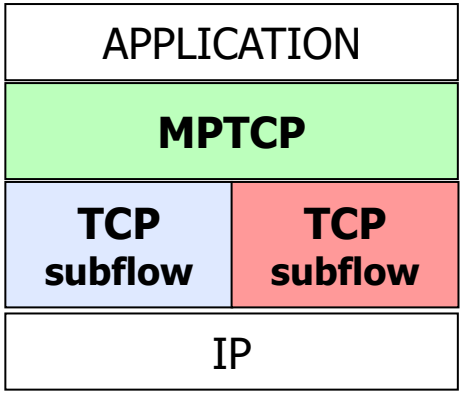


- Problem: Auf welcher Schicht passt das am besten?

4.5 Multipath TCP (MPTCP)

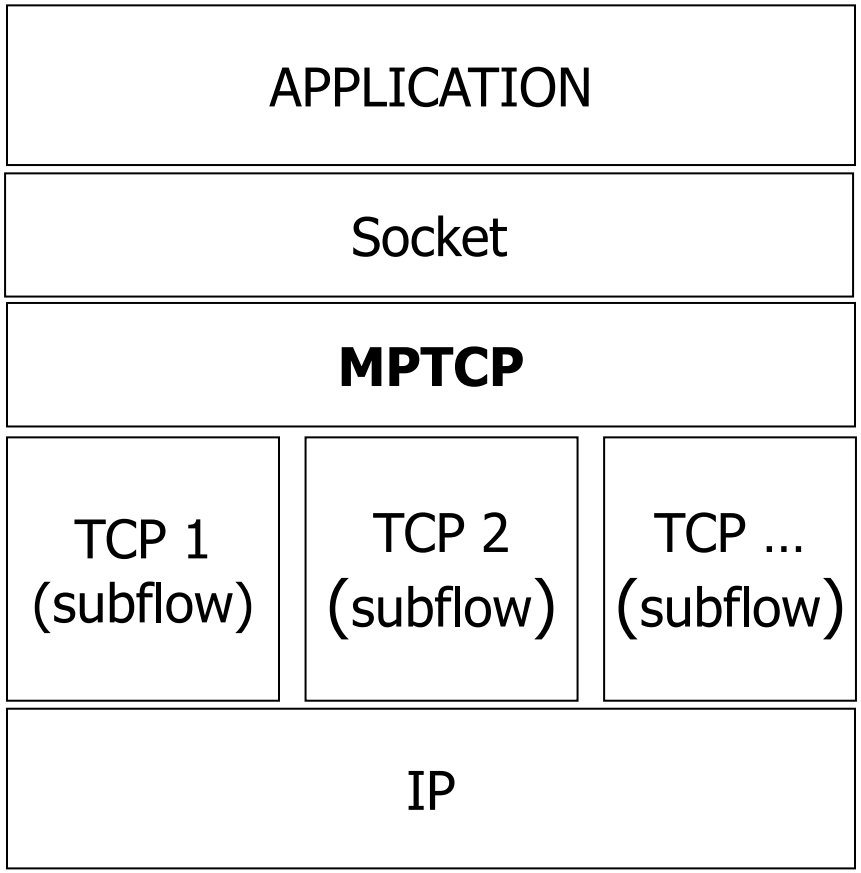
- Argumente für TCP (Transport):
  - TCP dominiert im Internet
  - Transport verfügt über Informationen am Endpunkt
    - Netzwerkzugänge (Interfaces)
    - Verbindungsmanagement, Leistungsstatistik
    - Anwendungsübergang (Datenstrom)
  - Endsystem verfügt über variable Ressourcen (Buffer, Prozesslogik, ...)
- MPTCP bleibt abwärtskompatibel und infrastrukturneutral
- Spezifiziert in RFC 6824, RFC 6897, RFC 6356

4.5 MPTCP Design



- Host-lokale MPTCP-Schicht
  - Signalisiert über TCP Options
  - Verhält sich wie TCP für eine einzelne Verbindung
  - Kann TCP-Subflows für mehrere IP-Interfaces öffnen
    - Subflows korrespondieren zu einzelnen TCP-Verbindungen
    - MPTCP vereint Subflows zu *einer* gemeinsamen Verbindung
  - MPTCP verwaltet mehrere Sequenznummern
    - Für jeden Subflow – TCP Sequenznummern
    - Für die Gesamtverbindung – MPTCP Sequenznummern

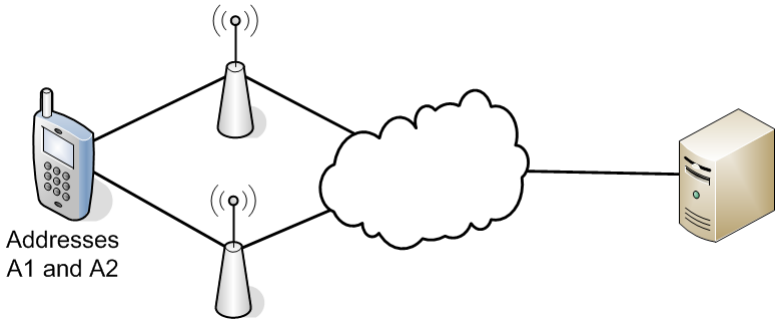
4.5 MPTCP Systemarchitektur

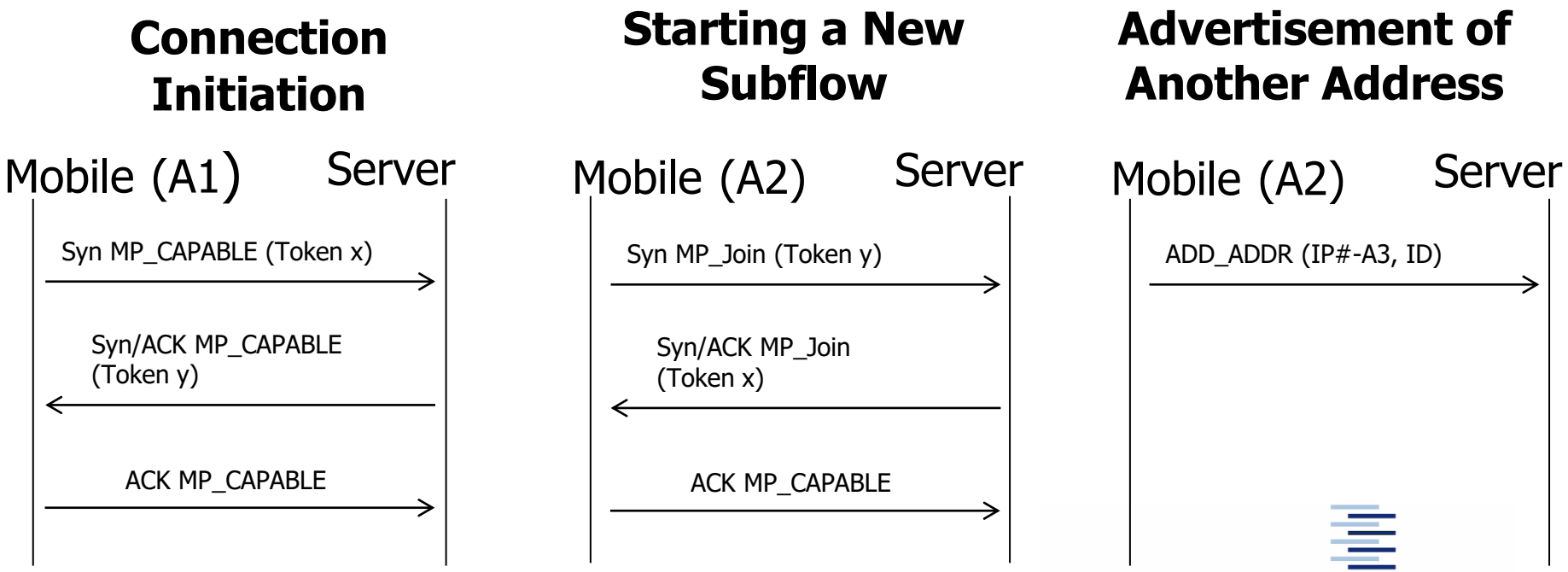


Manages different subflows
Manages retransmission of data between subflows

Subflows should look like regular TCP connections

4.5 MPTCP Protokollablauf

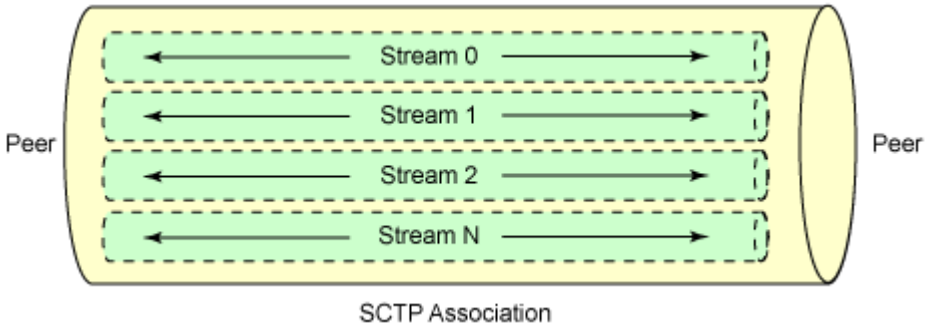




5. Streaming Control Transmission Protocol (SCTP)

- Verbindungsorientiertes Transportprotokoll
- Spezifiziert in RFCs 4960, 6096, 6335, 8260
- Message-orientiert:
  - Unterstützt Messages beliebiger Größe (fragmentiert)
  - Kann kleine Messages in einem SCTP-Paket bündeln
- Ermöglicht mehrere ‚Streams' pro Verbindung (analog SS7)
- Stream-Eigenschaften separat konfigurierbar
- Unterstützt Multi-Homing, Erweiterungen für Mobility
- Implementiert SACK

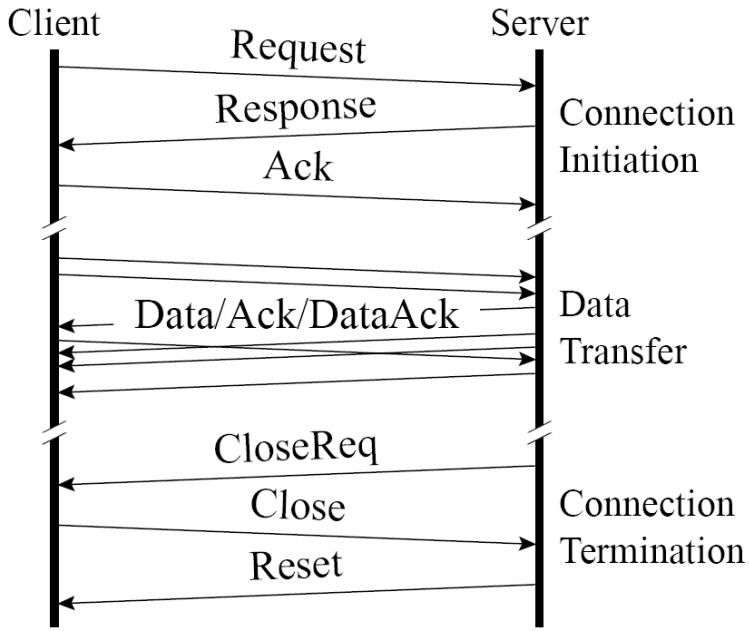
5.1 SCTP Streams



- Streams können unterschiedlich konfiguriert werden
  - Individuelle (partielle) Zuverlässigkeit
  - Individuelle (partielle) Ordnung
  - Individuelle Fluss- und Staukontrolle
  - Individuelle Priorität
- Design-Ziel: Differenzierung von Anwendungsdaten
  - Z.B. Echtzeit Audio/Video versus Massendaten
- Praktischer Einsatz in Browser-Browser Kommunikation
  - WebRTC über UDP

6. Datagram Congestion Control Protocol (DCCP - RFC 4340)

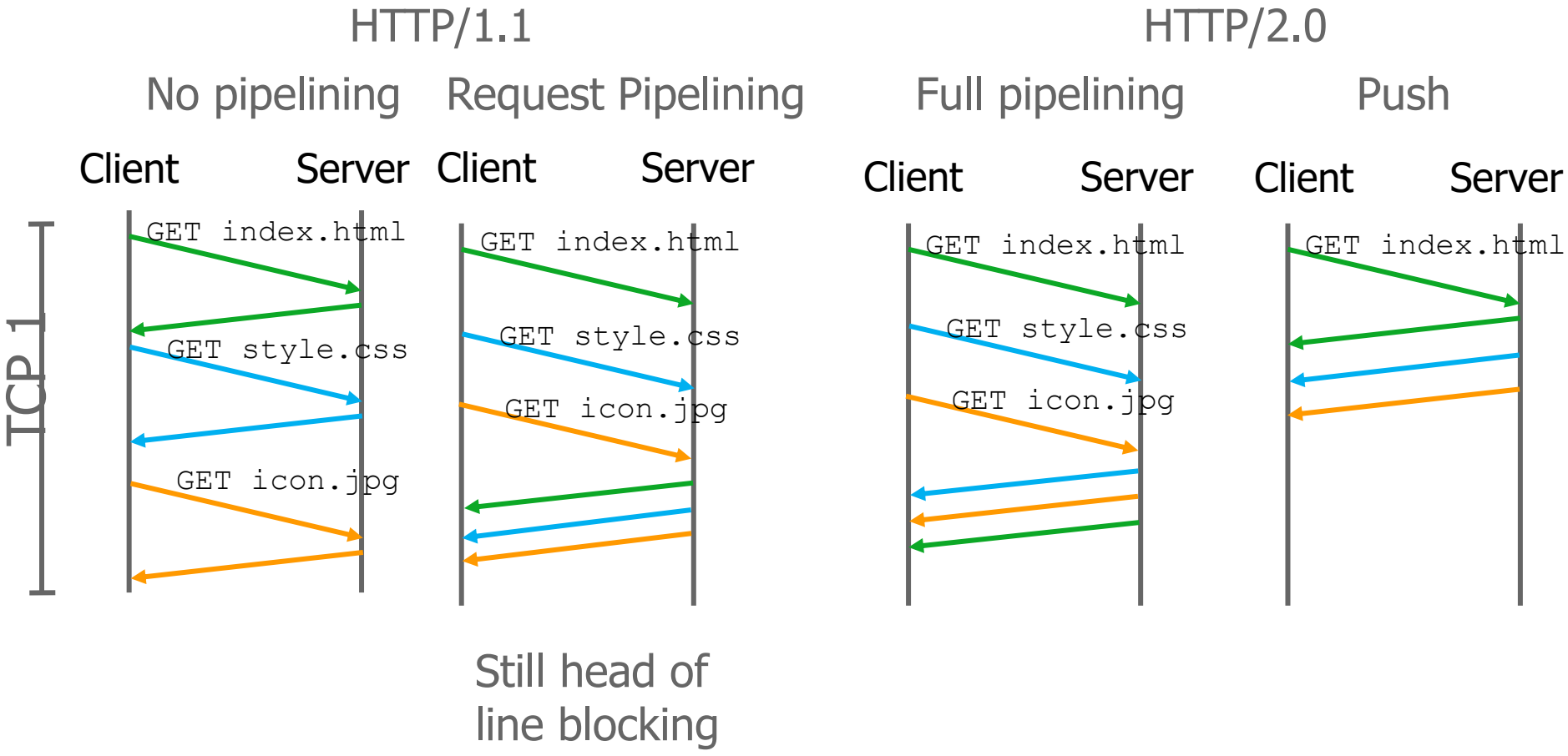
- Protokoll für ungesicherten Unicast Transport mit Staukontrolle
- Entworfen für Echtzeitanwendungen
- Verbindungsorientiert, entdeckt Packetverlust, ohne Pakete zu wiederholen
- Bietet den Rahmen für verschiedene Staukontrollmechanismen (Window-/Raten-basiert)
- Implementierungen Linux & BSD



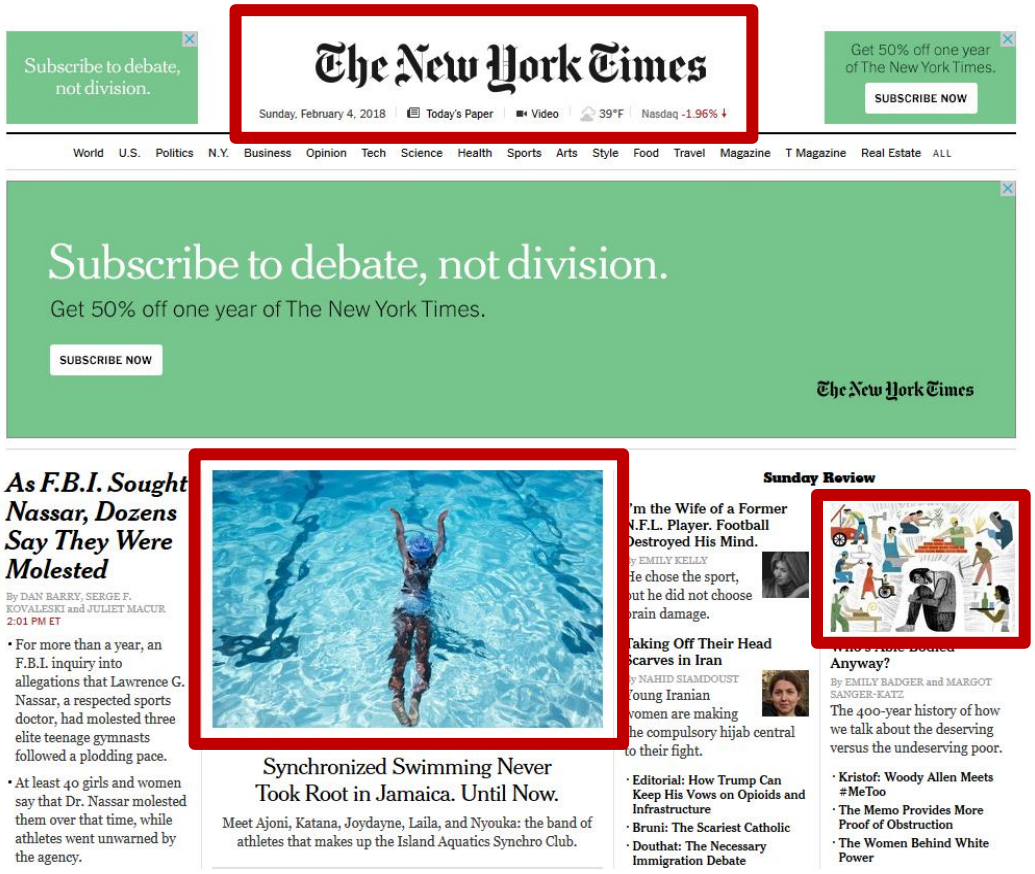
7. HTTP/2 Evolution

- Problem: Webseiten setzen sich aus vielen Downloads zusammen – sequentiell ist das Laden langsam
- TCP hält den Download an, sobald der Strom unterbrochen ist – Head of Line Blocking
- HTTP/2 multiplexes Downloads
  - Requests/Responds werden individuell zugeordnet
  - Kann dadurch in einer TCP-Verbindung Requests parallelisieren
  - Erlaubt Servern Content-Push in Browser Caches

7. Wachsende Nebenläufigkeit in HTTP



7. QUIC - Motivation



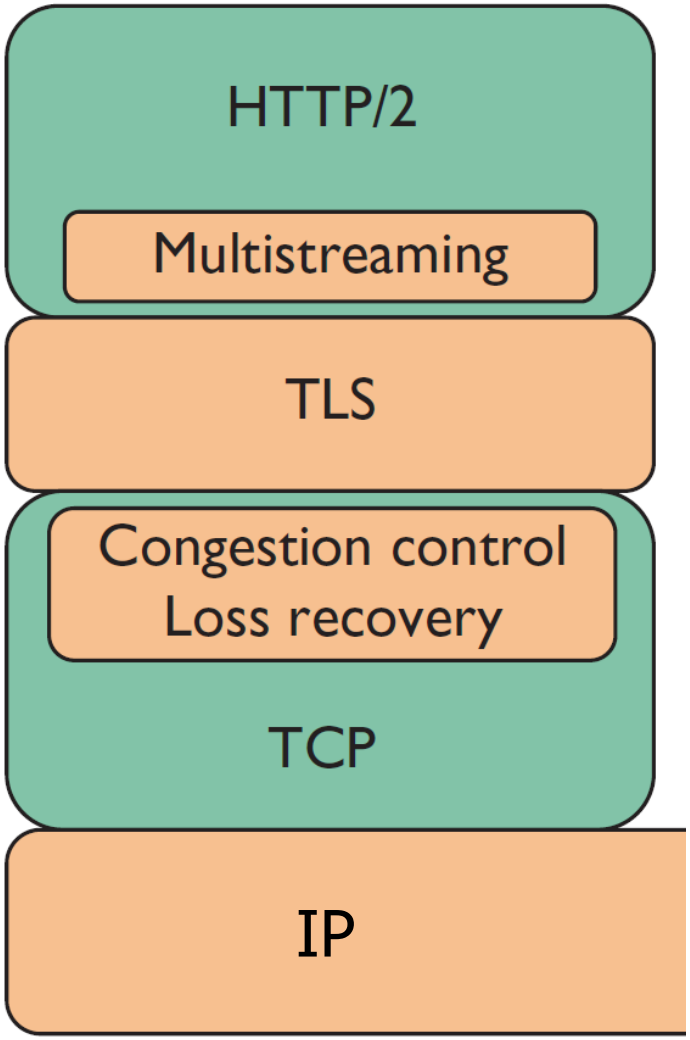
Problem: TCP
Head of Line Blocking

Im Web:

- http/1.1 implementiert Pipelining
- http/2.0 implementiert Multiplexing

Reicht das zur schnellen Webanzeige?

7. TCP bremst den Web Download



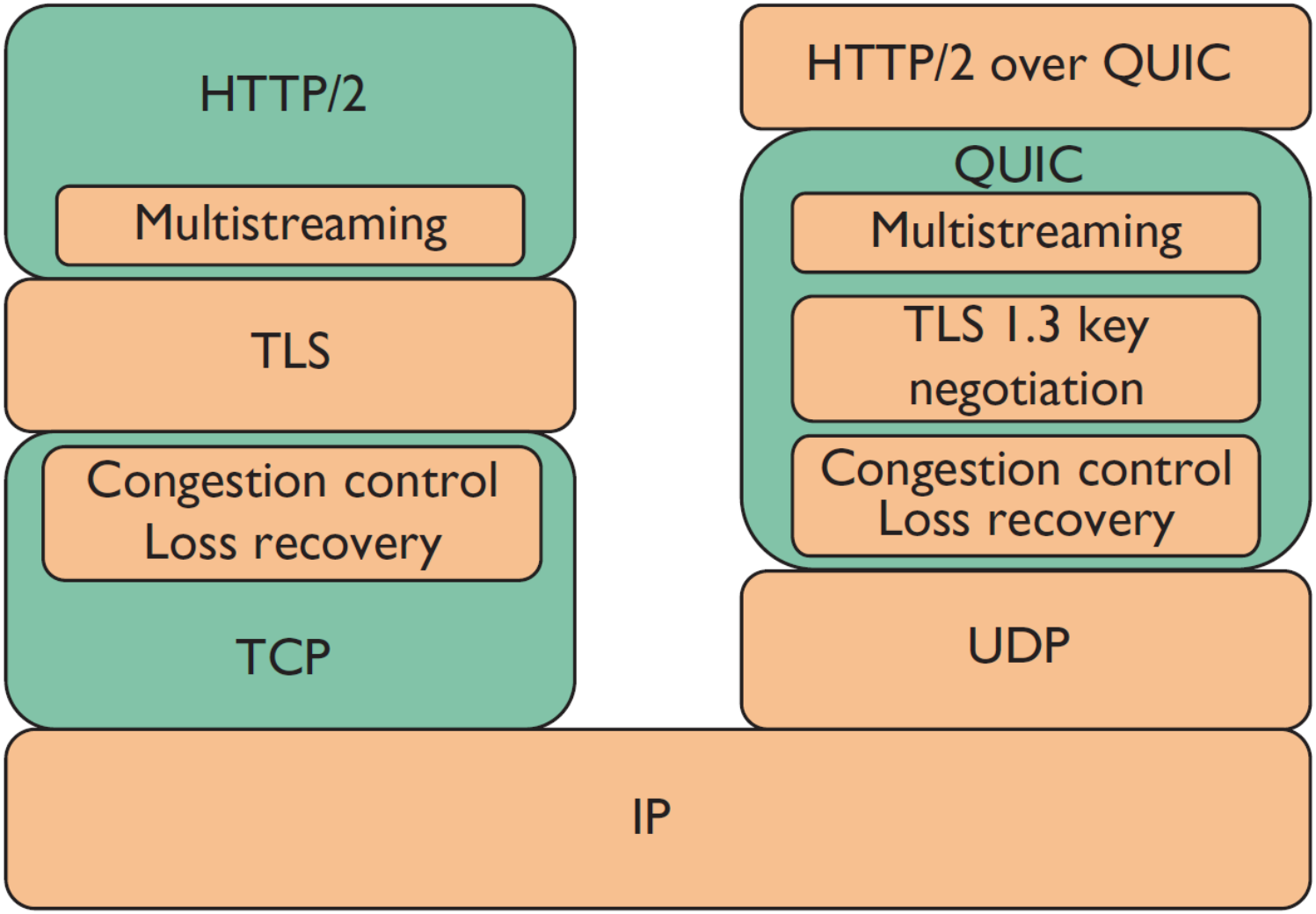
- HTTP(s) nutzt TCP
- Ein einzelnes verlorenes Paket im TCP-Strom hält alle HTTP/2-Ströme an
- Mehrere TCP-Ströme zu komplex
- SCTP nicht universell einsetzbar
- Einfachste Abhilfe: HTTP/2 über UDP + Verbindungslogik

7. QUIC: Quick UDP Internet Connections

Gegenwärtig in der Standardisierung - Zielstellungen:

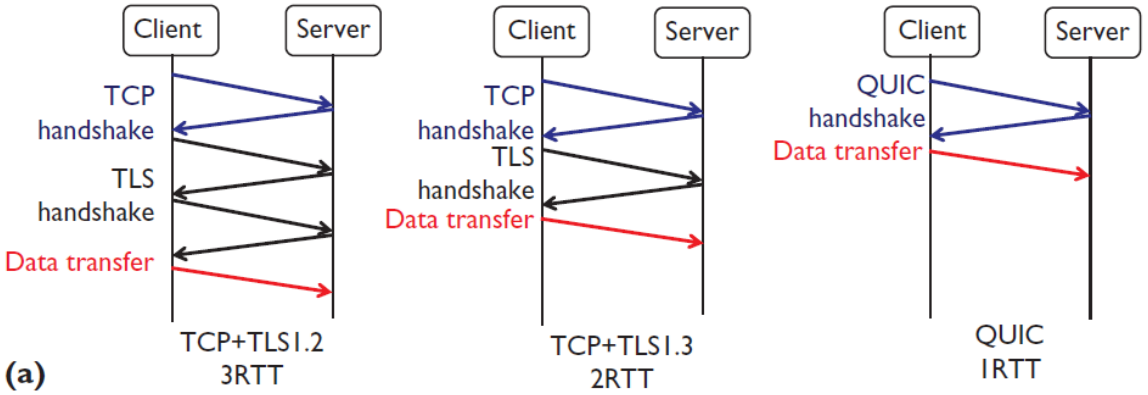
- Einfache Verbreitung – Google (Browser + Server!)
- Schnelles Verbindungsmanagement
  - Schneller Aufbau
  - Volles Multiplexing von (HTTP-)Streams
- Multipath
- Verbesserte Verlust- und Staubehandlung
- Integrierte TLS-Verschlüsselung
- Diverse Protokoll-Verschleierungen

7.1 Web Protokoll-Stack mit QUIC

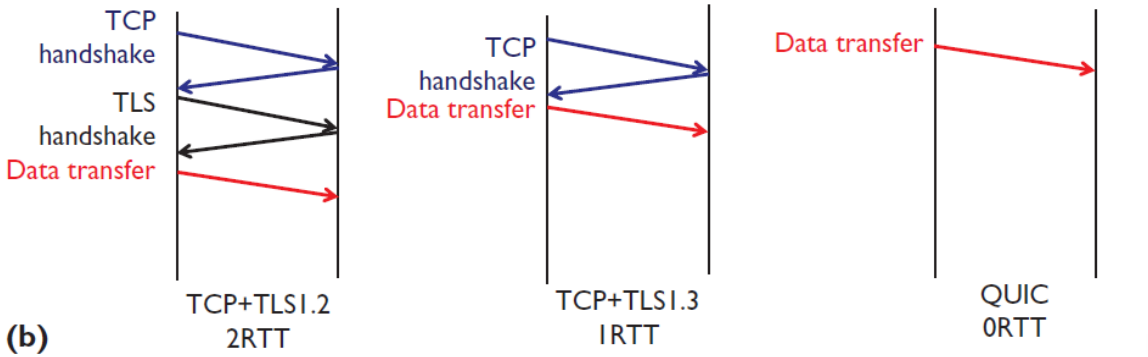


7.2 Handshakes: TCP vers. QUIC

Connection establishment



Subsequent connections



Cui et al., 2017

Selbsteinschätzungsfragen

1. Welches Transportprotokoll eignet sich zur Übertragung von Dateien, welches zur Gruppenkommunikation?
2. Wozu dient bei Verbindungsbeendigung der TCP Zustand CLOSE_WAIT?
3. Wie unterstützt TCP Stauvermeidung im Netz? Warum wird TCP auch als ‚höflich' bezeichnet?
4. Wie entscheidet TCP, ein Segment wiederholt zu versenden? Welche Erweiterungen gibt es?
5. Warum ist es im gegenwärtigen Internet schwierig, neue Transportprotokolle zu verbreiten?
6. Warum ist HTTP/2 über QUIC schneller als HTTP mit TCP-Verbindungs-Multiplexen?