

# Datenkommunikation und Sicherheit

## Kapitel 5: Transportschicht

Klaus Wehrle

Communication and Distributed Systems

Chair of Computer Science 4

RWTH Aachen University

<http://www.comsys.rwth-aachen.de>



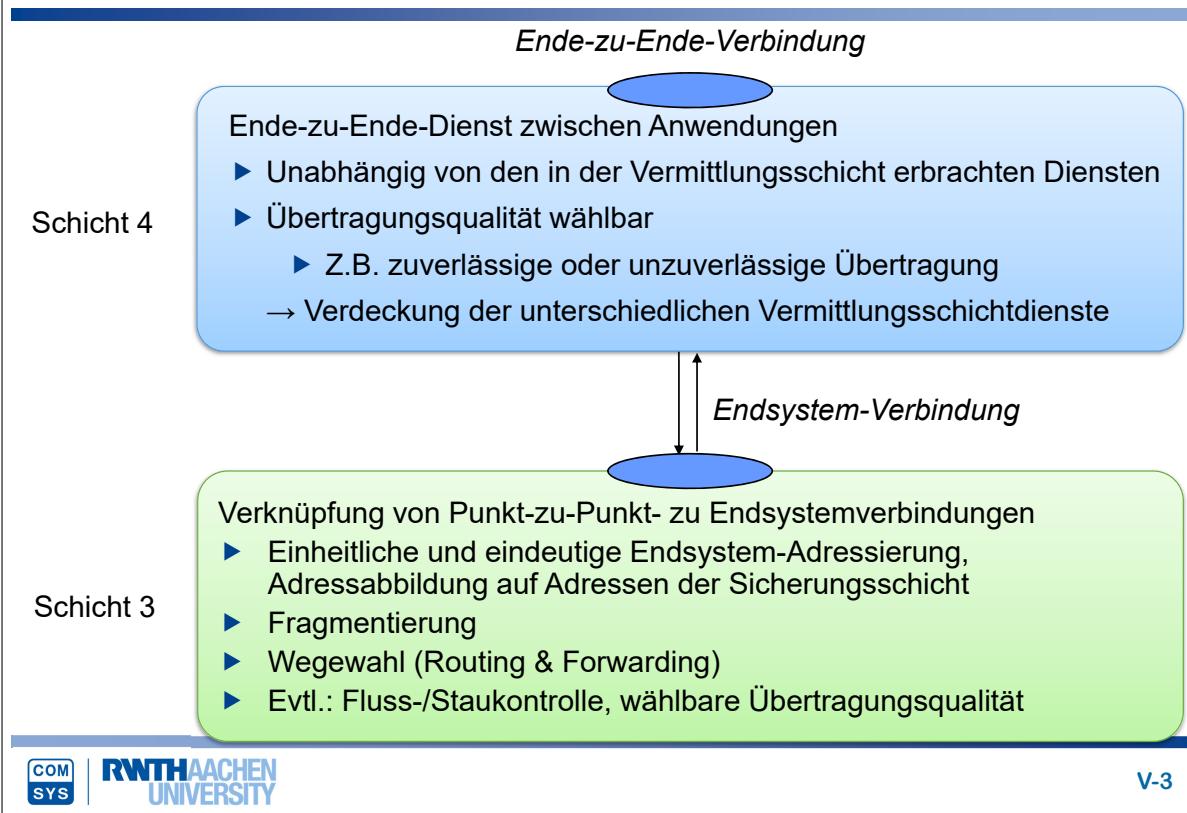
V-1

# Themenübersicht

## • Datenkommunikation und Sicherheit

- ▶ Einführung, Begriffe und allgemeine Grundlagen
- ▶ Technische und nachrichtentechnische Grundlagen: Medien, Signale, Bandbreite, Leitungscodes und Modulation, Multiplexing
- ▶ Lokale Netze: Strukturierung des Datenstroms, Fehlererkennung/-behebung, Flusssteuerung, Medienzugriff, Ethernet
- ▶ Internet und Internet-Protokolle
  - Vermittlungsschicht: IP, Routing
  - Transportschicht: TCP, UDP, QUIC
- ▶ Grundlagen der Sicherheit in/von Kommunikationsnetzen
  - Grundlagen: Verschlüsselung, Authentifizierung, Integrität
  - Sichere Internet-Protokolle

## Aufgaben der Transportschicht



Anwendungen benötigen für die Kommunikation untereinander einen meist zuverlässigen und einheitlichen Dienst, welcher die unterschiedlichen Leistungen der Vermittlungsschicht verdeckt. Dieser Dienst wird von der vierten Schicht im ISO/OSI-Basisreferenzmodell, der *Transportschicht*, erbracht, welche in diesem Kapitel behandelt wird.

Die Transportschicht ist nicht einfach „nur eine weitere Schicht“ im OSI-Modell. Sie repräsentiert das *Kernstück* der gesamten Protokollhierarchie, welche die netzwerkbezogenen, technischen Eigenschaften des Kommunikationssystems von den anwendungsorientierten Aspekten trennt. Die Anwendungen können somit auf einen einheitlichen (und, je nach gewählter Protokollinstanz, auch zuverlässigen) Dienst zugreifen, unabhängig von den verwendeten Technologien und der Topologie des darunterliegenden Netzwerkes. Im Gegensatz zur Vermittlungsebene, bei der sich die Kommunikation auf Rechnerknoten bezieht, werden in der Transportschicht *Anwendungen* adressiert, die miteinander kommunizieren.

Instanzen der Transportschicht finden sich nur in den Endsystemen, d.h. die Instanzen sind völlig unabhängig von den unterliegenden Netzen. Außerdem werden auf der Schicht 4 nicht Rechnersysteme adressiert sondern Anwendungen, d.h. es werden evtl. mehrere Transportschicht-Verbindungen zwischen zwei Anwendungen über eine Schicht-3-Verbindung geführt (Multiplexing). Außerdem ist es möglich, Quality of Service (Dienstgüte, Dienstqualität) für eine Transportverbindung festzulegen.

Funktionalitäten zur Erbringung solcher Dienste sind die Flusssteuerung des Datenstromes zwischen zwei Teilnehmern, das Multiplexen mehrerer Verbindungen auf eine Endsystemverbindung (zur Optimierung der Nutzung einer zur Verfügung stehenden Endsystemverbindung) und das Splitten einer Schicht-4-Verbindung auf mehrere Schicht-3-Verbindungen zur Erhöhung von Qualität und Sicherheit.

Der Transportschicht kommt – besonders im Internet – eine zentrale Bedeutung zu, da das dort verwendete IP zur Kommunikation zwischen Rechnern verbindungslos und unzuverlässig arbeitet:

- Eine Reihenfolgetreue der Pakete ist nicht gewährleistet
- Pakete können während der Übertragung verloren gehen oder verfälscht werden

Die Behandlung dieser Probleme wird – wenn benötigt – von der Transportschicht übernommen.

# Kapitel 5: Transportschicht

## • Protokollmechanismen der Transportschicht

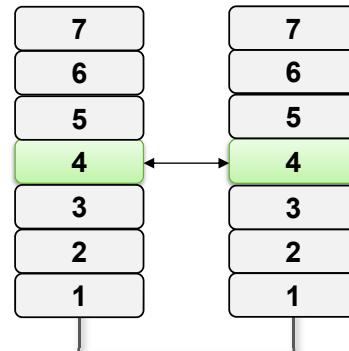
- ▶ Prozessadressierung, strombasierte vs. paketbasierte Kommunikation, verbindungsorientiert/verbindungslos

## • Die Transportschicht im Internet

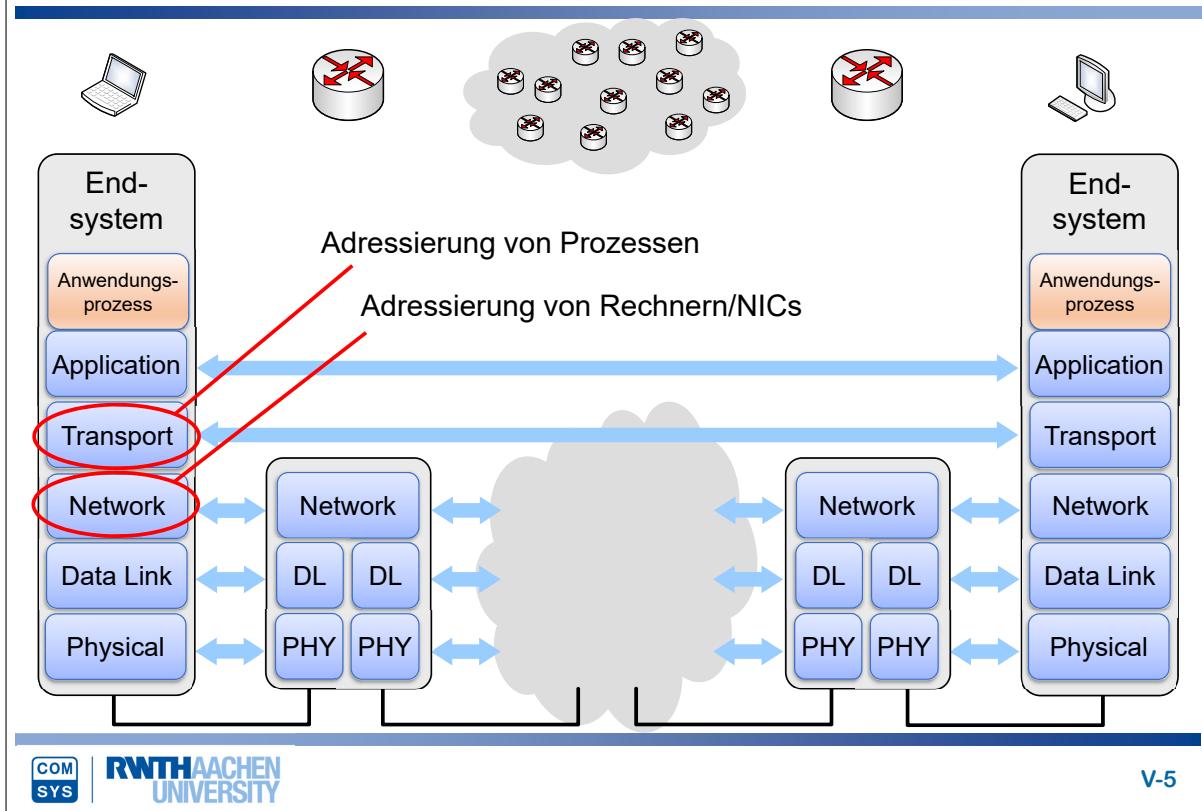
- ▶ TCP (Transmission Control Protocol)

- Adressierung, Sockets
- TCP-Verbindung
- Flusskontrolle
- TCP-Header
- Staukontrolle (Congestion Control)

- ▶ UDP (User Datagram Protocol)



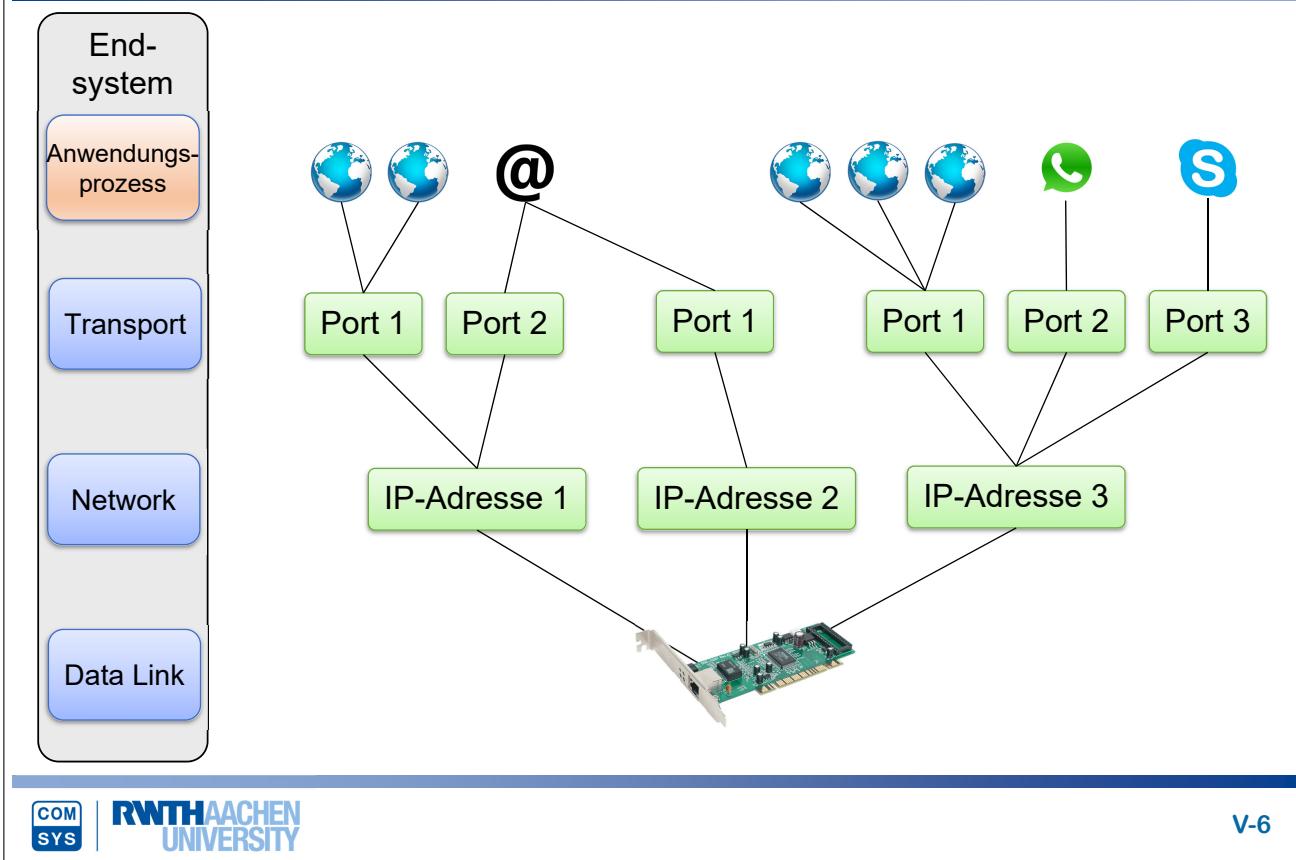
## Transport- und Vermittlungsschicht



Da die Transportschicht die kommunizierenden Anwendungen auf den Endrechnern adressiert und vom Netzwerk trennt, ist ihre Implementierung auch nur auf den Endsystemen notwendig. Router (oder generell: Vermittlungsknoten) im Netz brauchen sie nicht zu implementieren.

Während auf der Vermittlungsschicht (Schicht 3) eine Endsystemverbindung besteht, wird auf der Transportschicht eine Ende-zu-Ende-Verbindung zwischen den Teilnehmern (Anwendungen) aufgebaut. Die Transportinstanzen verständigen sich untereinander über das *Transportprotokoll*, also durch den Austausch von Transport-Protokolldateneinheiten (TPDUs: Transport Protocol Data Units). Diese TPDUs werden über die Endsystemverbindung übertragen.

# Rechner- und Prozessadressierung



Jeder Rechner muss anhand einer IP-Adresse adressierbar sein – aber man kann durchaus einem Rechner bzw. genauer gesagt einer Netzwerkkarte auch mehrere IP-Adressen zuweisen.

Die Transportschicht dient dazu, mehrere Anwendungsprozesse auf eine IP-Adresse zu multiplexen. Dazu werden sogenannte Port-Nummern verwendet: verschiedene gleichzeitig auf einem Rechner laufende Anwendungen können anhand eines lokalen Index voneinander unterschieden werden.

Unter Umständen ist es sogar möglich, mehrere Anwendungsprozesse gemeinsam über einen Port zu adressieren.

# Anwendungscharakteristika

## • Große Vielfalt an Anwendungen

### ► E-Mail

- Korrekte Übertragung von E-Mails, Verzögerung von einigen Minuten meist unkritisch

Zuverlässigkeit

### ► Webbrowsing

- Korrekte Übertragung von (größeren) Webseiten, keine zu großen Verzögerungen (Nutzerzufriedenheit)

Zuverlässigkeit  
(Latenz)  
(Durchsatz)

### ► Remote Work (SSH)

- Korrekte Übertragung von Zeichen, geringe Latenz notwendig (Interaktivität)

Zuverlässigkeit  
Latenz

### ► Namensauflösung (DNS)

- Kurze Nachrichten, möglichst keine großen Verzögerungen

(Latenz)

### ► Voice over IP

- Geringe Latenz zwingend notwendig (Interaktivität)

Latenz

Die Transportschicht hat aber nicht nur die Aufgabe, die Kommunikationsverbindungen verschiedener Anwendungsprozesse auf eine Netzwerkkarte zu multiplexen. Die zweite wesentliche Aufgabe ist, den Anwendungsprozessen einen Kommunikationsdienst bereitzustellen, der ihre Anforderungen erfüllt – dazu muss die Transportschicht Probleme, die auf der Netzwerkschicht auftreten können, vor den Anwendungen verbergen.

Das große Problem für die Transportschicht ist, dass sie mit dem Best-Effort-Dienst von IP leben und versuchen muss, die Anforderungen der verschiedenen Anwendungen zu erfüllen. Anwendungen wie E-Mail oder Webanwendungen benötigen eine zuverlässige Übertragung (vollständig, reihenfolgetreu) und darüber hinaus eventuell noch mehr – Webseiten sind heute oft sehr umfangreich, aber die Nutzer erwarten eine schnelle Anzeige der Inhalte, so dass auch eine relativ geringe Latenz und ein hoher Durchsatz benötigt werden. Noch kritischer sind Remote-Login-Anwendungen wie SSH: eine zuverlässige Übertragung der Kommandos ist notwendig, aber um dem Anwender ein Gefühl der Interaktivität zu vermitteln, muss auch eine geringere Latenz vorliegen.

DNS wird verwendet, um logische Rechnernamen in IP-Adressen aufzulösen, so dass wir Rechner im Web mit Namen adressieren können. Die hier ausgetauschten Nachrichten sind relativ klein und die Namensauflösung sollte schnell sein, um schnell die eigentliche Datenübertragung hin zur aufgelösten IP-Adresse beginnen zu können.

Interaktive Anwendungen, bei denen Sprache und/oder Bild übertragen werden, benötigen zwingend eine geringe Latenz. Wird Video übertragen, ist noch dazu ein relativ großer Durchsatz wichtig. Zuverlässigkeit spielt hingegen keine allzu große Rolle, da einzelne fehlende Teile eines Videos oder einer Sprachsequenz dem Benutzer nicht auffallen.

Ganz generell lassen sich diese Anforderungen in Zuverlässigkeit einerseits und Latenz

andererseits aufteilen. Als Lösung hat man auf der Transportschicht mehrere Protokolle definiert, die unterschiedliche Anforderungen erfüllen: TCP für Zuverlässigkeit, UDP für Einfachheit und damit geringe Latenz. Im Laufe der Zeit wurden weitere Transportprotokolle entwickelt, die sich alle nicht durchgesetzt haben; erst aktuell kommt mit QUIC ein neues Transportprotokoll, dass gleichzeitig Zuverlässigkeit und geringe Latenz erreichen soll.

# Verbindungslos oder verbindungsorientiert?

## • Anforderung Zuverlässigkeit

- ▶ Oft größere Menge an Daten (z.B. komplexe Webseiten)
  - **Strombasiert** – gesamter Datenstrom muss in Segmente unterteilt und beim Empfänger reassembliert werden
    - Übertragung der Segmente in unabhängigen IP-Paketen
  - Kontext zwischen den Segmenten notwendig
- *Verbindungsorientierte Kommunikation*

## • Anforderung Latenz

- ▶ Oft geringe Mengen an Daten (ein Paket) oder zeitkritische Daten ohne hohe Zuverlässigkeitssanforderungen
  - **Paketbasiert** – Versendung einzelner Pakete ohne Kontext oder Kontextverwaltung in der Anwendung
  - Kein Kontext auf Transportschicht notwendig
- *Verbindungslose Kommunikation*

Die große Vielfalt an Anwendungen sorgt für unterschiedliche Anforderungen – die nicht alle in einem einzigen Transportprotokoll erfüllt werden können. Denn zur Erfüllung der Anforderungen sind bereits unterschiedliche Grundprinzipien nötig.

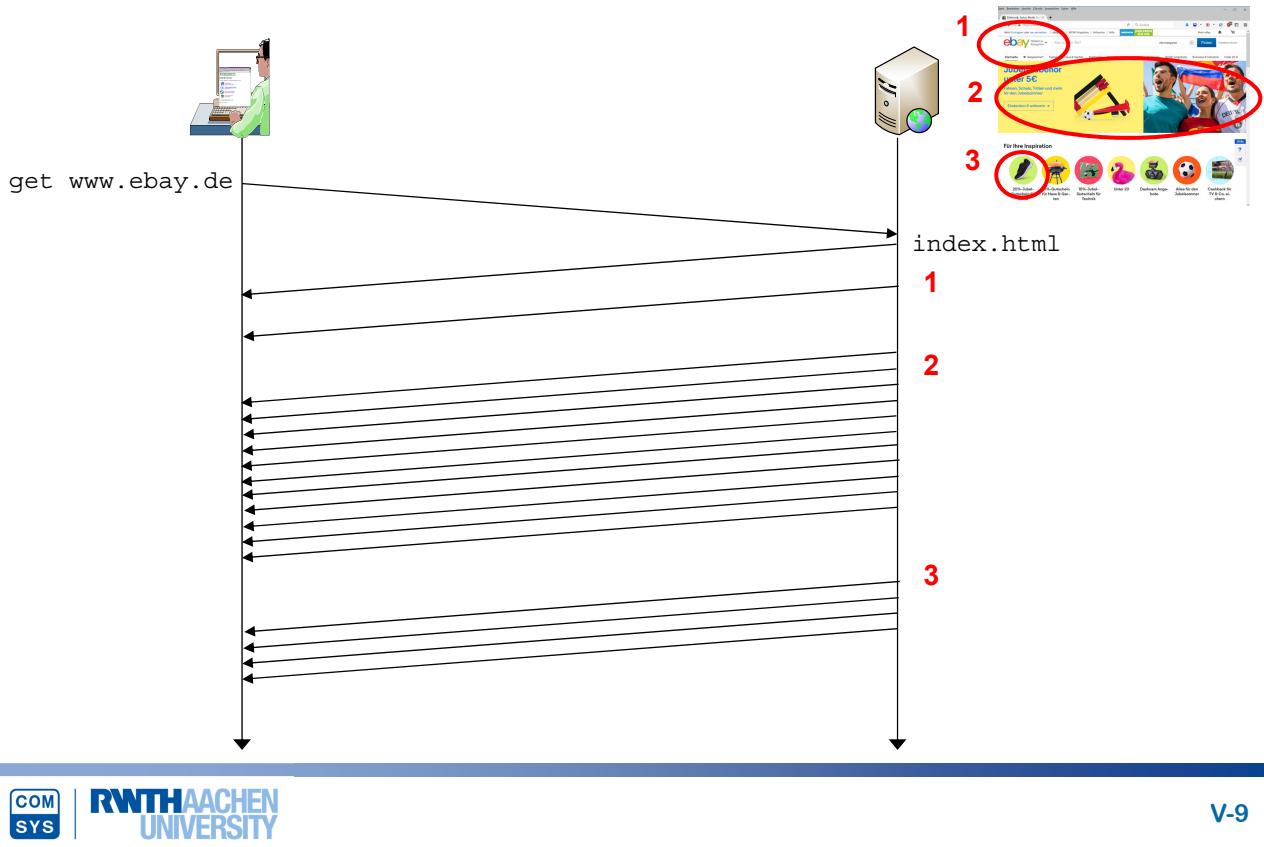
Um Zuverlässigkeit erreichen zu können, muss man mit Quittungsmechanismen arbeiten – und oft sind die Datenmengen so umfangreich, dass die zu übertragenden Daten als Datenstrom aufgefasst werden können, der in mehrere Pakete aufgeteilt werden muss. Daher benötigt man einen Kontext bei der Übertragung, muss also ein verbindungsorientiertes Protokoll definieren.

Um geringe Latenz zu erzielen, stört ein Verbindungsaufbau – er fügt eine Latenz hinzu. Gerade wenn nur kurze Nachrichten übertragen werden, die in einzelne Pakete passen (wie z.B. bei DNS), wäre eine verbindungslose Übertragung sinnvoller. Paketverlust ist in den meisten Fällen zwar vielleicht ärgerlich, aber nicht kritisch, so dass eine Anfrage einfach noch einmal wiederholt werden kann, wenn keine Antwort kommt.

Bei Anwendungen wie Telefonie ist eine verbindungsorientierte Übertragung auch hinderlich, da ihre Kontrollmechanismen einen Overhead erzeugen können, der die Übertragung verzögert. Hier bevorzugt man verbindungslose Kommunikation und übernimmt eine Kontextverwaltung auf Anwendungsebene. Zuverlässigkeit ist nicht kritisch, da das menschliche Hirn kleinere Mengen an fehlenden Sprachinformationen ausfiltert.

Neben diesen beiden Anforderungen ist auch Durchsatz wichtig – für die Übertragung moderner Webseiten, die dem Benutzer ein Gefühl von Interaktivität geben sollen, oder auch für z.B. Videostreaming. Einen hohen Durchsatz könnte man allerdings mit beiden Techniken erreichen. (Allerdings muss man vorsichtig vorgehen, um das Netz nicht zu überlasten, wie im Folgenden dargestellt.)

# Strombasierte Übertragung



Ein gutes Beispiel für strombasierte Übertragung ist die Übertragung von Webseiten. Moderne Webseiten bestehen aus vielen unterschiedlichen Elementen, die oft sehr groß sind und über mehrere Pakete verteilt werden müssen. Hier im Beispiel wird zuerst das Gerüst der Webseite übertragen, danach in einem Paket das Logo, dann eine sehr große Abbildung, zum Schluss eine kleinere Abbildung.

Man muss eine variable Menge an Daten in Pakete zerteilen – dazu werden die gesamten zu übertragenden Daten als Bytestrom gesehen, der in Pakete segmentiert werden muss.

# Welche Probleme kann IP verursachen?

- ▶ Paketverlust
  - Oft keine Fehlerbehebung auf Schicht 2 implementiert
  - Router können Pakete verwerfen, z.B. bei Überlast
- ▶ Pakete können in falscher Reihenfolge ankommen
  - Durch Übertragung auf unterschiedlichen Pfaden
- ▶ Pakete können dupliziert werden
  - Durch Neuübertragung bei verlorengegangener Quittung
- ▶ Verzögerung von Paketen
  - Durch hohe Routerlast
- ▶ Keine Flusskontrolle vorhanden
  - Flusskontrolle auf Schicht 2 nur für einzelne Hops
    - Keine Ende-zu-Ende-Vermeidung von Überlastung eines Empfängers
  - Keine Vermeidung von Router-Überlastung

Fehlerbehebung (ARQ)  
Stauvermeidung

Reihenfolge- und  
Duplikaterkennung

Stauvermeidung

Flusskontrolle  
Stauvermeidung

IP bietet weder Zuverlässigkeit noch gibt es die Möglichkeit, eine bestimmte Latenz zu garantieren. Selbst wenn auf Schicht 2 entsprechende Mechanismen z.B. zur Sicherstellung der Zuverlässigkeit realisiert sind, treten auf Schicht 3 wieder neue Probleme auf, so dass die Transportschicht wieder mit vielen Problemen konfrontiert ist, die bereits auf Schicht 2 existierten – nur jetzt auf globaler Ebene.

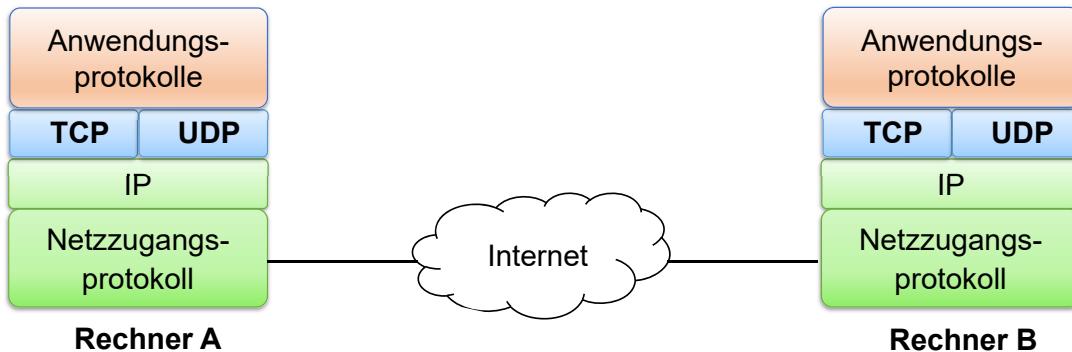
## Umsetzung der Transportschicht

- ***Verbindungsorientiert*, Hauptaspekt Zuverlässigkeit**
  - ▶ Sequenznummern und ARQ zur Behebung von Paketverlust, Reihenfolgeumstellung, Paketduplicierung
  - ▶ Ende-zu-Ende-Flusskontrolle pro Anwendung
  - ▶ Staukontrolle zur Vermeidung von Routerüberlast
  - ▶ Möglichkeit eines hohen Durchsatzes trotz Staukontrolle
- ***Verbindungslos*, Hauptaspekt Latenz**
  - ▶ Keine Kontrollmechanismen (Vermeidung von Overhead)
  - ▶ Stauvermeidung dennoch sinnvoll

## Die Transportschicht im Internet

- Hauptsächlich zwei Protokolle:

- ▶ **TCP (Transmission Control Protocol)**: Zuverlässiges, verbindungsorientiertes Transportprotokoll über unzuverlässigem IP
- ▶ **UDP (User Datagram Protocol)**: Verbindungsloses Transportprotokoll, fügt lediglich eine Anwendungsschnittstelle zu IP hinzu



Im Internet werden auf Transportebene vorwiegend TCP und UDP genutzt. Der wesentliche Unterschied zwischen den beiden Transportprotokollen ist, dass TCP verbindungsorientiert arbeitet, während UDP eine Paket-orientierte Übertragung anbietet. Häufig wird TCP in einem Atemzug mit IP genannt: man spricht von TCP/IP. Das deutet bereits darauf hin, dass TCP im Vergleich zu UDP die bessere Ergänzung zum Paket-orientierten IP darstellt. Die Unzuverlässigkeit des auf der Vermittlungsschicht eingesetzten IP wird durch die Verwendung von TCP vollständig verdeckt. Dadurch können die Anwendungen auf einen gesicherten Dienst zugreifen.

Die Dienste der Transportschicht können von den Anwendungsprotokollen genutzt werden, wobei diese in den meisten Fällen auf TCP aufsetzen. Beispiele für solche Anwendungsprotokolle sind FTP (File Transfer Protocol, Dateiübertragung), SSH (Secure Shell, Remote Terminal), SMTP (Simple Mail Transfer Protocol, E-Mail) oder HTTP (HyperText Transfer Protocol, Webseiten).

Es gibt aber auch Anwendungen, die UDP verwenden - vor allem, wenn der Zusatzaufwand der gesicherten Übertragung nicht benötigt wird, da nur kurze, nicht-zeitkritische Nachrichten versendet werden. Dazu zählen beispielsweise SNMP (Simple Network Management Protocol, Netzwerkmanagement) oder DNS (Domain Name System, Abbildung logischer Namen auf IP-Adressen). Es gibt sogar zeitkritische Anwendungen, die auf keinen Fall erneute Paketübertragungen wollen, weil die Neuübertragung den ganzen Datenstrom ausbremst und die neuübertragenen Daten bei Ankunft eventuell schon veraltet sind. Beispiele hierfür sind Audio- und Videoübertragungssysteme, die ihre Daten in Echtzeit übertragen (Live-Streaming, Videokonferenzen). Diese Anwendungen nehmen eher einen Paketverlust in Kauf (was kurzfristig zu einer schlechteren Ton- oder Bildqualität führt) als eine Paketwiederholung (d.h. Anhalten des Audio-/Videostroms und warten auf das fehlende Paket).

Neben TCP und UDP sind mehrere weitere Transportprotokolle definiert worden, von denen viele allerdings nie oder nur sehr rudimentär eingesetzt wurden. Neuere Entwicklungen, die noch weiter Verbreitung finden könnten, sind SCTP (Stream Control Transmission Protocol) und QUIC (Quick UDP Internet Connections); diese werden in dieser Vorlesung allerdings nicht weiter betrachtet. Gerade QUIC stellt eine wichtige Entwicklung dar, da es verbindungsorientierte Kommunikation mit reduzierten Latenzen ermöglichen soll.

# Kapitel 5: Transportschicht

- **Protokollmechanismen der Transportschicht**

- ▶ Prozessadressierung, strombasierte vs. paketbasierte Kommunikation, verbindungsorientiert/verbindungslos

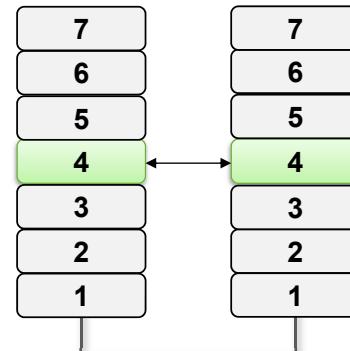
- **Die Transportschicht im Internet**

- ▶ TCP (Transmission Control Protocol)

- Adressierung, Sockets

- TCP-Verbindung
  - Flusskontrolle
  - TCP-Header
  - Staukontrolle (Congestion Control)

- ▶ UDP (User Datagram Protocol)



## TCP: Eigenschaften und Dienste (1)

- **TCP: gesicherte Übertragung eines Bytestroms zwischen zwei Anwendungen über das unzuverlässige IP**

- ▶ **Verbindungsverwaltung**

- Verbindungsaufbau zwischen zwei *Sockets*
    - Datentransfer über eine (virtuelle) Verbindung
    - Gesicherter Verbindungsabbau (alle übertragenen Daten müssen quittiert sein)

- ▶ **Multiplexen**

- Mehrere Verbindungen der Transportschicht können auf eine „Verbindung“ (eher Strecke) der Vermittlungsschicht abgebildet werden



Eine TCP-Verbindung dient zur Übertragung eines Bytestromes von Anwendung A zu Anwendung B. Es werden also einzelne Bytes ausgetauscht, keine Nachrichten.

TCP ist ein robustes Protokoll, welches sich dynamisch an die Eigenschaften und die aktuelle Verfügbarkeit des darunter liegenden Netzes anpassen kann. Das ist auch notwendig, weil es sich beim Internet um eine äußerst heterogene Topologie von unzähligen Netzwerken und verschiedenen Komponenten (Hosts, Router, Switches) handelt, die unterschiedlich leistungsfähig sind und unterschiedliche Eigenschaften besitzen.

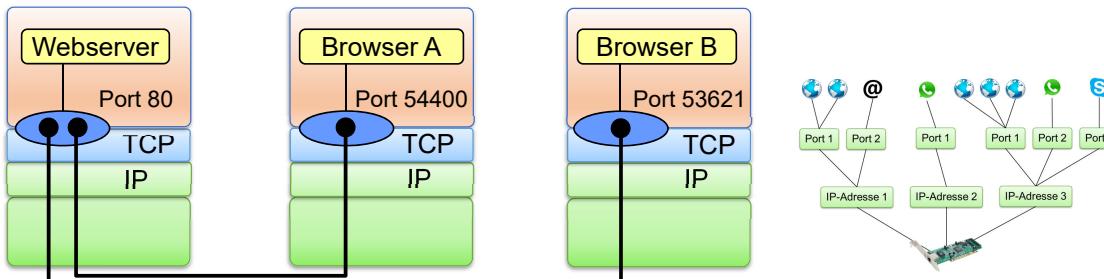
Auch bei TCP gibt es die klassischen Phasen bei der Verbindungsverwaltung:

*Verbindungsaufbau, Datentransfer und Verbindungsabbau.* Um der Zuverlässigkeit der Datenübertragung zu genügen, ist der Verbindungsabbau zusätzlich gesichert, d.h. es müssen sämtliche übertragenen Daten vom Empfänger quittiert sein, bevor ein Verbindungsabbau eingeleitet werden kann.

TCP setzt das Konzept des Multiplexens um, indem es Ports einführt. Ein Rechner kann über die (verbindungslose) IP-Schicht mehrere Ende-zu-Ende-Verbindungen zu beliebigen anderen Rechnern aufbauen. Diese werden anhand der Portnummern identifiziert.

## TCP: Adressierung

- Identifikation von Kommunikationsendpunkten geschieht über *Ports*
  - ▶ 16-Bit-Nummer
  - ▶ Portnummern bis 1023 für Standarddienste reserviert (z.B. 80 für HTTP, 25 für SMTP, 22 für SSH)
  - ▶ **Socket:** umfasst IP-Adresse eines Rechners und einen Port
    - Notation: (IP-Adresse:Portnummer) → internetweit eindeutig



Protokolle der Anwendungsschicht können die von TCP bzw. UDP bereitgestellten Dienste an *Ports* in Anspruch nehmen. Ports erlauben die Adressierung mehrerer Kommunikationsendpunkte im selben System. Falls zwischen zwei Rechnern gleichzeitig mehrere Verbindungen bestehen, können die Pakete anhand der Portnummern dem richtigen Kommunikationsfluss zugeordnet werden. Protokolle der Anwendungsschicht, über die weitverbreitete Dienste realisiert werden (wie beispielsweise FTP als Protokoll für die Übertragung von Dateien), haben eigene Ports reserviert. Dadurch ist es jedem möglich, einen Webserver (auf Port 80) anzubieten, ohne dafür sorgen zu müssen, dass alle Clients die verwendete Portnummer erfahren. Möchte man nicht, dass jeder den Dienst nutzen kann, kann man aber auch einen anderen Port wählen. Die Ports bis 1023 sind standardisiert und sind für bestimmte Dienste vorgesehen. Oft können Anwendungen diese Ports nur mit Root-Rechten verwenden. Die Ports von 1024 – 49151 sind registrierte Ports – sie werden auch üblicherweise für bestimmte Anwendungen verwendet, aber auch von Standardbenutzern verwendbar. Die höheren Ports können dynamisch verwendet werden.

Bitte beachten: Standardports werden nur auf der Serverseite verwendet, siehe Abbildung auf der Folie. Auf Clientseite wird kein Standardport ausgewählt – die TCP-Implementierung im Betriebssystem wird bei Initiierung eines Verbindungsaufbaus einfach einen freien Port außerhalb der reservierten Nummern wählen und von dieser Adresse aus einen Verbindungsaufbau vornehmen. Der standardisierte Port ist nur auf Seiten des Servers notwendig, der kontaktiert werden muss.

Es ist auch möglich, über TCP direkt zu kommunizieren, ohne ein Anwendungsprotokoll zu verwenden. Instanzen der Anwendungsschichtprotokolle werden einen sogenannten Socket erzeugen, um darüber zu kommunizieren. Diese Sockets sind eine Datenstruktur mit Buffer, die an eine IP-Adresse und einen Port gebunden werden und damit einen Kommunikationsendpunkt darstellen. Ein Socket kann eine TCP-Verbindung zu einem anderen Socket herstellen. Die Protokolle der Anwendungsschicht verwenden dann eine Variable des Datentyps, um Daten über den Socket zu versenden bzw. zu empfangen.

Implementiert man seine eigene Anwendung, benötigt man nicht notwendigerweise ein Anwendungsschichtprotokoll, man kann auch direkt auf Sockets zugreifen und über diese kommunizieren

## Standardisierte Port-Nummern (well-known ports)

- Viele Anwendungen nutzen TCP als Protokoll

- ▶ Allerdings muss der richtige Port gewählt werden, um auf der Gegenseite mit der richtigen Anwendung zu kommunizieren

- 13: daytime
- 20/21: FTP  
(File Transfer Protocol)
- 25: SMTP  
(Simple Mail Transfer Protocol)
- 53: DNS  
(Domain Name System)
- 80: HTTP  
(HyperText Transfer Protocol)
- ...

```
> telnet relay.rwth-aachen.de 25
Trying 134.130.3.1...
Connected to sokrates .
Escape character is '^]'.
220 sokrates ESMTP Sendmail 8.8.5/8.8.5;
Mon, 4 Aug 2007 17:02:51 +0200
HELP
214-This is Sendmail version 8.8.5
214-Topics:
214- HELO EHLO MAIL RCPT DATA
214- RSET NOOP QUIT HELP VRFY
214- EXPN VERB ETRN DSN
214-For more info use "HELP <topic>".
...
214 End of HELP info
```



## TCP: Verbindungsorientierung

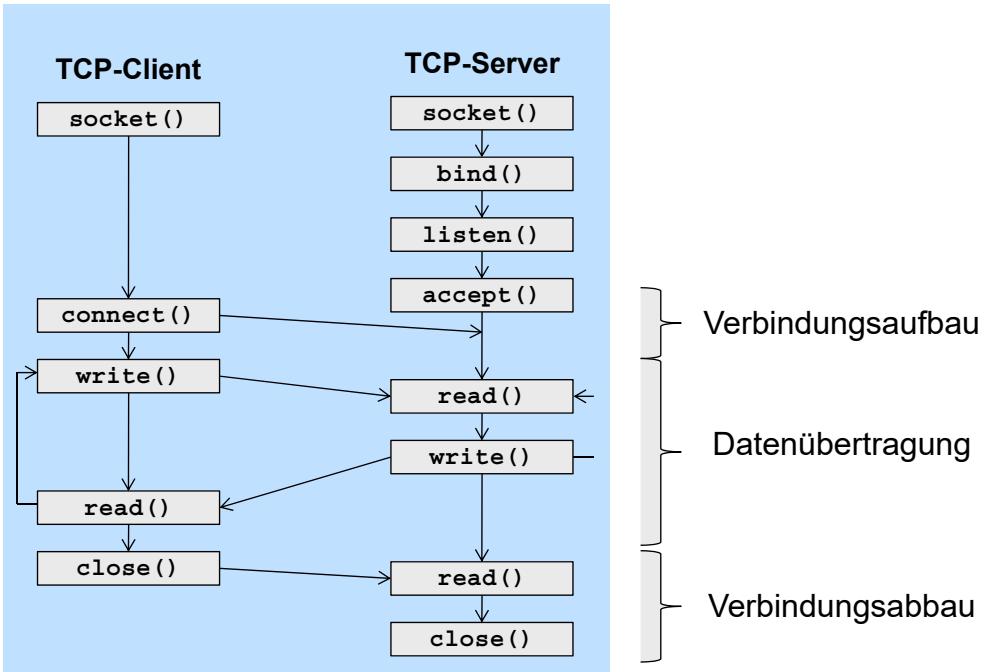
- Sockets können *aktiv* oder *passiv* erstellt werden

- ▶ Aktiver Modus: *Initiator* fordert auf einem Socket eine TCP-Verbindung an
  - Meist verwendet durch einen *Client*
- ▶ Passiver Modus: *Responder* informiert TCP, dass er auf eine eingehende Verbindung wartet
  - Typischerweise verwendet durch einen *Server*
  - Spezifikation eines speziellen Sockets (Clients), von dem eine eingehende Verbindung erwartet wird (*fully specified passive open*)
  - Alle Verbindungen annehmen (*unspecified passive open*)
  - Geht ein Verbindungsaufbauwunsch ein, wird ein neuer Socket erzeugt, der als Verbindungsendpunkt dient



Wird ein Socket erstellt, kann er in den aktiven oder passiven Modus versetzt werden. Aktiv bedeutet, dass über den Socket eine Verbindung zu einem spezifizierten Kommunikationspartner (IP-Adresse und Port, also ein konkreter Socket) aufgebaut wird. Passiv heißtt, dass die TCP-Instanz auf eingehende Verbindungsaufbauwünsche wartet. Bei üblichen Client/Server-Anwendungen verwendet der Client einen aktiven, der Server einen passiven Socket.

# Ablauf bei der Kommunikation über TCP-Sockets



Bei der Programmierung von netzwerkfähigen Anwendungen mittels Sockets werden einem Programmierer Primitive bereitgestellt, die für Auf- und Abbau der Verbindung sowie für die Datenübertragung verwendet werden können und somit die Schnittstelle einer Anwendung zu TCP darstellen.

## Socket-Primitive in TCP

- Kommunikation via TCP durch einen Satz von Primitiven, die ein Anwendungsprogrammierer verwenden kann:

Primitiv	Bedeutung
SOCKET	Erstellung eines neuen Netzzugangspunkts
BIND	Verknüpfe eine lokale Adresse mit dem Socket
LISTEN	Warte auf ankommende Verbindungswünsche
ACCEPT	Nimm einen Verbindungswunsch an
CONNECT	Initiierung eines Verbindungsaufbaus
WRITE (SEND)	Sende Daten über die Verbindung
READ (RECEIVE)	Empfange Daten auf der Verbindung
CLOSE	Freigabe der Verbindung

Je nach Programmiersprache sind die Primitive unterschiedlich benannt; eventuell werden sogar mehrere Primitive in einer einzigen Funktion zusammengefasst (z.B. LISTEN und ACCEPT in Java).

## Server

```
int main()
{
10    int listenfd, connfd;
    socklen_t len;
    struct sockaddr_in servaddr, cliaddr;
    char buffer[1024];
    time_t ticks;

15    listenfd = socket(AF_INET, SOCK_STREAM, 0);
    memset(&servaddr, 0, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
    servaddr.sin_port = htons(8013);
20    bind(listenfd, (struct sockaddr*)&servaddr, sizeof(servaddr));
    listen(listenfd, 5);
    while(1){
        len = sizeof(cliaddr);
        connfd=accept(listenfd,(struct sockaddr*)&cliaddr, &len);
25        printf("connection from %s, port %d\n",
            inet_ntop(AF_INET,&cliaddr.sin_addr,buffer,sizeof(buffer)),
            ntohs(cliaddr.sin_port));
        ticks=time(NULL);
        sprintf(buffer,sizeof(buffer),"%.24s\r\n",ctime(&ticks));
30        write(connfd,buffer,strlen(buffer));
        close(connfd);
    }
    return 0;
}
```

**1-8:** declarations of the used library functions

**10-14:** declaration of the used variables: File descriptors, `sock_addr` structures...

**15:** instantiation of a IPv4 (`AF_INET`) / TCP (`SOCK_STREAM`) socket

**16-19:** fill in the `sockaddr_in` structure: IP-address (`INADDR_ANY`) and port (8013), converting into network byte order

**20:** configuration of the sockets with `bind(...)`. It will listen to port 8013 and accepts connections on all IP addresses (in accordance with `sockadd_in`)

**21:** switch socket to passive mode (`listen`)

**23:** tell `accept`, how large our `sockadd_in` structure is

**24:** `accept` waits for incoming connections and returns a file descriptor `connfd` of the connection

**26-28:** prints the connecting parameters

**29-30:** generate a string with data in `buffer` and sends the answer (`write`)

**31:** close the active connection and wait for the next one

# Client

```
int main(int argc, char **argv)
{
10    int sockfd;
    struct sockaddr_in destaddr;
    char buffer[1024];

    if(argc!=3){
        printf("Syntax: simplex-client address port\n");
15        return 0;
    }
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    memset(&destaddr,0,sizeof(destaddr));
    destaddr.sin_family = AF_INET;
20    destaddr.sin_port = htons(atoi(argv[2]));
    inet_pton(AF_INET,argv[1],&destaddr.sin_addr);
    connect(sockfd,(struct sockaddr*)&destaddr,sizeof(destaddr));
    while(1){ /* endless loop */
        int r;
25        r = read(sockfd, buffer, 1024);
        if(r<=0)break; /* no data to read or connection closed */
        printf("%s",buffer);
    }
    close(sockfd);
30    return 0; /* never reached */
}
```

**1-7:** declarations of the used library functions

**10-12:** declaration of the used variables:

File descriptors, `sock_addr` structures...

**13-16:** test command line parameters

**17:** initializing the socket

**18-21:** fill in the `sockaddr_in` structure: destination address and port with command line parameters, converting the port number into network byte order

**22:** initiate connection. Waits until connection works

**23-31:** infinite loop to read data, abort with `break`

**25:** read data (maximum 1024 byte) into `buffer`

**26:** abort, if connection failed (`read(...)` returns -1)

**27:** print the received data

**29:** close socket

# Kapitel 5: Transportschicht

- **Protokollmechanismen der Transportschicht**

- ▶ Prozessadressierung, strombasierte vs. paketbasierte Kommunikation, verbindungsorientiert/verbindungslos

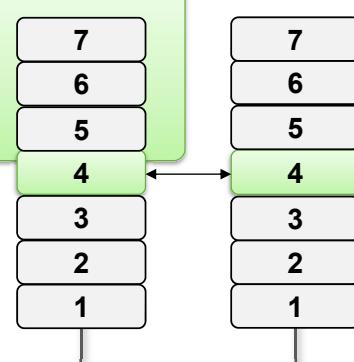
- **Die Transportschicht im Internet**

- ▶ TCP (Transmission Control Protocol)

- Adressierung, Sockets

- TCP-Verbindung
  - Flusskontrolle
  - TCP-Header
  - Staukontrolle (Congestion Control)

- ▶ UDP (User Datagram Protocol)



## TCP: Eigenschaften und Dienste (2)

- **TCP: gesicherte Übertragung eines Bytestroms zwischen zwei Anwendungen über das unzuverlässige IP**
  - ▶ *Segmentierung* von Byteströmen zur Übertragung in IP-Paketen
  - ▶ Datenübertragung:
    - *Vollduplex* und *Punkt-zu-Punkt*
    - *Fehlerkontrolle* durch Folgenummern (Sequenznummern), Prüfsumme, Quittierung, Übertragungswiederholung im Fehlerfall
      - Verwendung von Timern / *Timeouts*
    - *Flusskontrolle* (durch Fenstermechanismus) und *Staukontrolle*
  - ▶ *Fehleranzeige*
    - Ist die Auslieferung der Daten in einer bestimmten Zeit nicht möglich, wird der Dienstnutzer darüber in Kenntnis gesetzt
  - ▶ *Dynamische Anpassung* an Eigenschaften des Internets
    - z.B. heterogene Topologien, schwankende Bandbreiten



Die Gewährung der Zuverlässigkeit erfordert u.a. eine reihenfolgetreue Auslieferung der einzelnen Pakete, welche durch IP auf Vermittlungsebene nicht gewährleistet ist. Dies wird bei TCP durch die Verwendung von Folgenummern erreicht. Jedes Paket wird mit einer Sequenznummer versehen, die im TCP-Header vermerkt ist. Der Empfänger hat somit die Möglichkeit, die ankommenden Pakete wieder in die richtige Reihenfolge zu bringen und kann gegebenenfalls Pakete erneut anfordern, falls diese unterwegs verloren gingen oder beschädigt wurden. Zur Erkennung einer Fehlersituation ist es oft notwendig, das Ausbleiben einer Quittung zu erkennen, wozu Timeouts eingesetzt werden.

Die Flusskontrolle basiert bei TCP auf einem Fenstermechanismus, wie er bereits in Kapitel 3 (Sicherungsschicht) ausführlich beschrieben wurde. Die Staukontrolle hat eine ähnliche Aufgabe wie die Flusskontrolle – die Flusskontrolle soll eine Überlastung des Empfängers vermeiden, die Staukontrolle eine Überlastung des Netzes. Dies ist keine einfache Aufgabe, da TCP nach Design unabhängig vom Netz ist. Auf die Fluss- und vor allem die Staukontrolle bei TCP wird im weiteren Verlauf dieses Kapitels noch detaillierter eingegangen, da sie zentrale Mechanismen des Transportprotokolls darstellen.

## TCP: Verbindungsorientierung zur Zuverlässigkeit

- **TCP zerlegt einen Bytestrom in *Segmente***
    - ▶ Datensegmente
    - ▶ Kontrollsegmente für Verbindungsaufbau, -verwaltung, -abbau
  - **Verbindungsorientierung und Zuverlässigkeit bei TCP**
    - ▶ Verbindungsaufbau durch Kontrollsegmente, z.B. Reservierung von Speicher (Buffer) auf beiden Seiten
    - ▶ Zerteilung der Daten in Segmente, die jeweils mit einer Sequenznummern versehen werden (im TCP-Header)
    - ▶ Empfänger schickt Quittungen für korrekt empfangene Segmente (Wiederholung unquittierter Segmente, Reihenfolgewiederherstellung)
    - ▶ Verbindungsabbau durch Kontrollsegmente, z.B. Freigabe von Buffer
- ***virtuelle Verbindung über verbindungsloses IP***

TCP nennt sich zwar ein verbindungsorientiertes Protokoll, was aber im Gegensatz zu den tieferen Schichten nicht bedeutet, dass ein fester Pfad durch das Netz aufgebaut wird. Bei einer Verbindungsorientierung auf Schicht 3 würde genau dies geschehen – der beste Weg für Dateneinheiten über ein komplexes Netz hinweg würde ermittelt und alle Dateneinheiten entlang der gleichen Route geschickt. Im Internet ist diese Schicht allerdings verbindungslos (IP). TCP baut auf IP auf und muss dessen Dienste nutzen, so dass keine feste Verbindung erzwungen werden kann. Stattdessen wird der nächsthöheren Schicht vorgegaukelt, dass TCP verbindungsorientiert ist, indem es die Probleme von IP verbirgt (Paketverlust, Reihenfolgeumstellung) und den Anschein erweckt, die Daten wären verbindungsorientiert und zuverlässig übertragen worden.

# Verbindungsauftbau

## • Three-Way-Handshake

- Netz kann Pakete verlieren, verspätet ausliefern, duplizieren
- Abhilfe: Sequenznummern und dreiteiliger Handshake



## • Kompromiss zwischen Komplexität und garantierter Einigung auf eine Verbindung

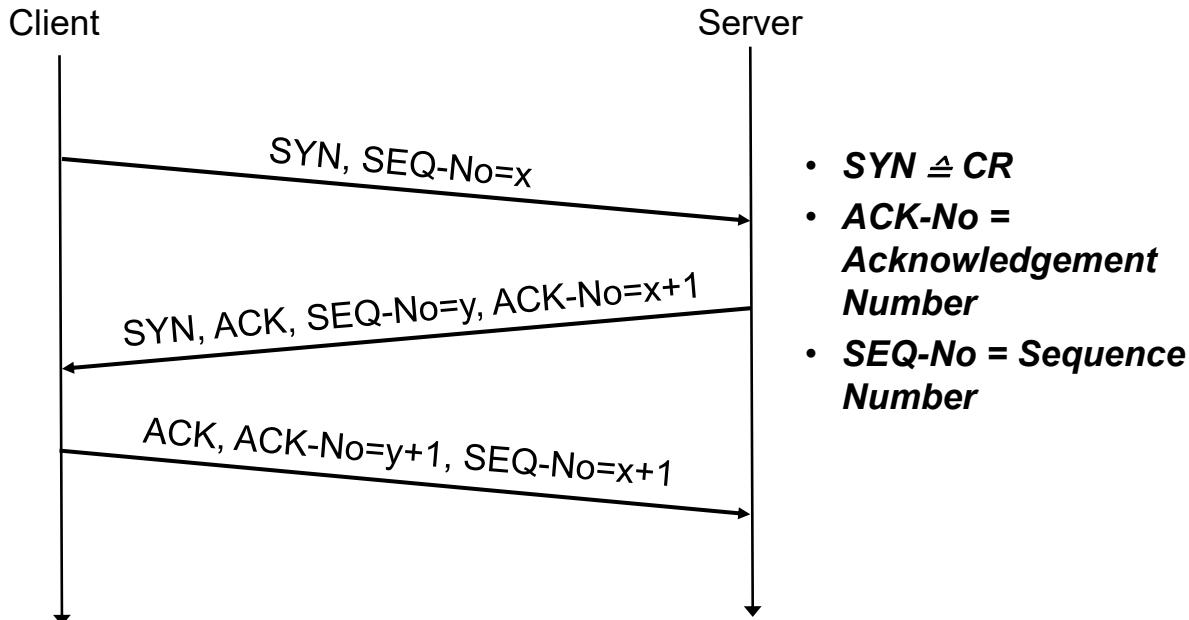
Zweck des Verbindungsauftbaus ist es, die Kommunikationsbereitschaft der Gegenstelle zu testen und initiale Parameter auszuhandeln (z.B. Buffergrößen, Sequenznummern).

Alle Pakete können bei Verwendung von IP verloren gehen, daher ist nicht gewährleistet, dass ein Verbindungsauftbauwunsch (Connection Request, CR) beim Kommunikationspartner ankommt. Eine Quittierung der Verbindungsauftbaunauftrag (Acknowledgement, ACK) ist also nicht nur notwendig, um dem Initiator der Verbindung die Kommunikationsbereitschaft zu signalisieren und Parameter mitzuteilen, sondern auch, damit der Initiator weiß, dass sein CR nicht verloren gegangen ist. Aber auch diese Quittung könnte verloren gehen, so dass der Empfänger in Bereitschaft ist, aber nie Daten folgen. Daher quittiert auch der Initiator den Erhalt der Quittung noch einmal. Zwar kann auch diese Quittung verloren gehen, so dass ihr Erhalt auch wieder bestätigt werden müsste, aber dieses Spielchen ließe sich endlos fortsetzen. Bei Verlust der letzten Quittung wird dies als nicht tragisch angesehen, da der Initiator im Normalfall auch direkt mit der Datenübertragungsphase beginnen wird, durch die implizit die verlorene Quittung ersetzt wird.

TCP arbeitet hier durchgängig mit Timern – trifft eine Quittung nicht innerhalb einer bestimmten Zeit ein, geht TCP von einem Datenverlust aus und wiederholt das Segment.

Während dieses Handshakes werden bereits Folgenummern verwendet, um gegen Fehler bei der Übertragung zu schützen. Es könnte z.B. sein, dass ein Duplikat der CR-Nachricht beim Empfänger eintrifft, wodurch dieser annehmen würde, dass ein neuer Verbindungsauftbauwunsch vorliegt. Trifft die zugehörige Quittung beim Initiator ein, kann er identifizieren, dass sich diese auf einen nicht mehr aktuellen Verbindungsauftbauwunsch bezieht, und wird eine negative Quittung zurückschicken, damit der Empfänger nicht unnötig Buffer für diese Verbindung reserviert.

## TCP-Verbindungsmanagement: 1. Verbindungsaufbau

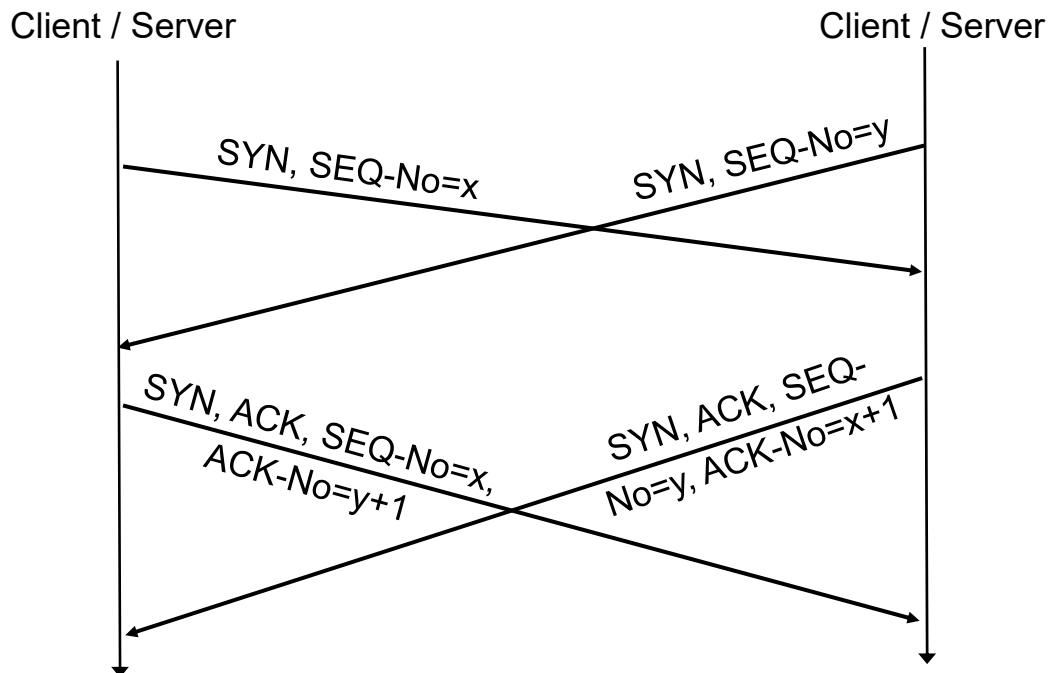


Wird ein Socket im aktiven Modus erstellt, wird ein Connection Request an den adressierten passiven Socket gesendet und nach dem Three-Way-Handshake quittiert. Dass es sich bei den ausgetauschten Segmenten um Nachrichten eines Verbindungsaufbaus handelt, wird durch Setzen eines bestimmten Flags im TCP-Header kenntlich gemacht. TCP definiert nur eine PDU-Struktur und erlaubt durch eine Menge von Flags, kenntlich zu machen, welchem Zweck das aktuelle Segment dient (Verbindungsaufbau (SYN), Daten (kein gesetztes Flag), Quittung (ACK), Verbindungsabbau (FIN), Zurücksetzen der Verbindung (RST), ...). Im Connection-Request-Segment sendet der Client eine initiale Sequenznummer (x). Ebenso sendet der Server in seiner Quittung eine Sequenznummer (y) mit zurück. Hierbei werden das ACK-Flag und das SYN-Flag gesetzt: die Nachricht ist gleichzeitig eine Quittung über den Erhalt der CR-Nachricht (ACK-Flag – hierdurch wird signalisiert, dass die Nachricht eine Quittungsnummer enthält) als auch eine Annahme des Verbindungsaufbauwunsches (SYN-Flag). Die letzte Nachricht des Clients schließt den Three-Way-Handshake ab, die Verbindung ist aufgebaut.

Die Sequenznummern x und y werden auf beiden Seiten zufällig ausgewählt (sie müssen nicht bei 0 beginnen). Der Hauptzweck der Sequenznummern ist – wie auf Schicht 2 – die Einführung eines Quittungsmechanismus und einer Flusskontrolle. Während des Verbindungsaufbaus teilen sich die beiden Kommunikationspartner dadurch mit, mit welchen Sequenznummern sie im Folgenden beginnen werden, die auszutauschenden Daten durchzunummerieren.

Gleichzeitig wird beim Verbindungsaufbau auf beiden Seiten Bufferspeicher reserviert, in dem empfangene Daten zwischengespeichert werden können, bis sie in richtiger Reihenfolge an die Anwendung weitergereicht werden können.

## Alternativ: ungeregelter Verbindungsauftbau

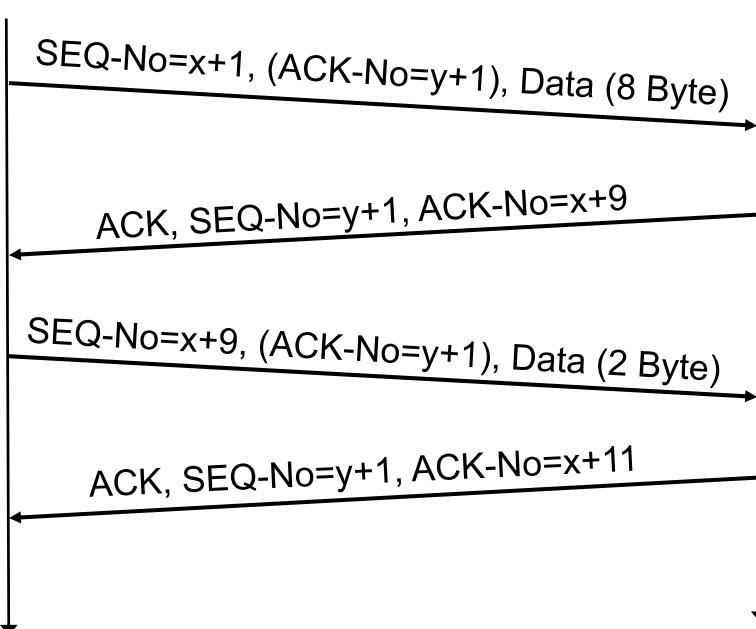


Gibt es keine festgelegten Client- und Serverrollen, kann es eventuell passieren, dass beide Seiten simultan versuchen, eine Verbindung mit dem jeweils Anderen aufzubauen. Dies ist überflüssig, da TCP-Verbindungen vollduplex sind – und darüber hinaus kann zwischen zwei Kommunikationsendpunkten sowieso nur genau eine Verbindung existieren, da jede Verbindung eindeutig durch IP-Adressen und Ports der beiden Sockets identifiziert wird. Daher wird in diesem Fall die hier dargestellte Variante des Verbindungsauftbaus verwendet: jede Seite hat bereits eine SYN-Nachricht des Kommunikationspartners erhalten und sendet auf diese eine Quittung zurück. Danach ist die Verbindung aufgebaut.

## TCP-Verbindungsmanagement: 2. Datenübertragung

Client

Server



- **Übliche Segmentgröße: 1460 Byte – vermeiden von IP-Fragmentierung**

- **Verwendung von Timern (Timeouts) zum Schutz gegen Paketverlust**

Nach dem Verbindungsauftakt besteht kein Unterschied mehr zwischen den Kommunikationspartnern. Die Verbindung ist bidirektional, Daten und Quittungen können in beide Richtungen übertragen werden. Im Beispiel auf der Folie sendet nur der Client Daten, der Server sendet Quittungen zurück. Der Client sendet einen Bytestrom, den er segmentiert – eine übliche Größe für den Payload, den ein Segment enthält, ist 1460 Byte – zuzüglich der Standard-Header von TCP und IP (je 20 Byte) entsteht so eine SDU von 1500 Byte Größe auf der Sicherungsschicht, was genau die Größe ist, die maximal in einem Ethernet-Rahmen übertragen werden kann, so dass IP-Fragmentierung in üblichen LANs vermieden wird. In der Praxis sind also die Schichten doch nicht völlig voneinander getrennt sondern so konfiguriert, dass sie optimal zusammenspielen.

Jedes Datensegment wird mit einer Sequenznummer versehen. Diese bezeichnet – basierend auf der beim Verbindungsauftakt ausgetauschten Sequenznummer ( $x+1$ ) – die Position des ersten im Segment enthaltenen Bytes innerhalb des gesamten Bytestroms.

Die Quittungsnummern des Empfängers haben die gleiche Bedeutung wie bei der Flusskontrolle, die auf Schicht 2 behandelt wurde: die Quittungsnummer entspricht demjenigen Byte, welches als nächstes erwartet wird, alle vorherigen Bytes wurden korrekt empfangen. Der Unterschied zur Implementierung auf Schicht 2 ist also die Einheit der Nummerierung – Rahmen vs. Bytes, so dass bei TCP die Sequenznummern zweier aufeinanderfolgender Segmente im Normalfall keine aufeinanderfolgenden Nummern sein werden.

Da durch IP Pakete verloren gehen oder in falscher Reihenfolge ausgeliefert werden können, wird nur eine positive Quittung zurückgeschickt, falls das nächste empfangene Segment auch die Sequenznummer des nächsten erwarteten Bytes enthält. Der Sender startet beim Senden jedes Segments einen Timer; bei einem Timeout wird von einem Paketverlust ausgegangen und das Segment erneut übertragen.

Die in Klammern gesetzten Quittungsnummern bei den Segmenten vom Client an den Server sollen andeuten, dass es sich um keine notwendige Angabe handelt. Da keine Daten in Serverrichtung quittiert werden müssen, braucht der Client das ACK-Flag nicht zu setzen und daher auch keinen gültigen Wert als ACK-No einzutragen. In der Praxis werden diese Angaben aber trotzdem gemacht – dadurch werden schlicht und einfach vorherige Quittungen wiederholt, was dem Ablauf von TCP nichts schadet, aber gegen negative Auswirkungen des Verlusts vorheriger Quittungen schützt. Beispielsweise könnte die dritte Nachricht des Three-Way-Handshakes von Folie 27 verlorengehen, so dass der Server irgendwann einen Timeout hat und seine Quittung wiederholt. Setzt der Client hier auf dieser Folie bei jedem seiner Segmente die Quittungsnummer (und das zugehörige Flag), wiederholt er die verlorene Quittung mit jedem Datensegment, so dass der Timer des Server stoppen würde und die Neuübertragung seiner vorherigen Quittung nicht vorgenommen wird.

Bitte nicht täuschen lassen: diese Folie scheint den Eindruck zu erwecken, dass die Fehlerbehebungsstrategie von TCP Stop-and-Wait ist – allerdings ist hier nur zur Vereinfachung diese Darstellung gewählt worden. Auch bei TCP können Strategien wie Go-Back-N und Selective Repeat eingesetzt werden.

## TCP: Neuübertragung von Segmenten

- Mehrere Verfahren zur Reaktion auf Fehler (fehlende Segmente, Segmente in falscher Reihenfolge)

- ▶ Ursprüngliches TCP: *Go-Back-N*
  - Quittungen werden nur für Segmente mit erwarteter Nummer gesendet
  - Quittungsnummer  $n$  bestätigt Empfang aller Bytes bis  $n-1$
  - Nicht erwartete Segmente werden verworfen
  - Übertragungswiederholung aufgrund Ablauf von Retransmission Timer des Senders
- ▶ Oder Aushandlung bei Verbindungsaufbau:
  - *Selective-Repeat* (RFC 1106)
    - Durch NAK kann fehlerhaftes Segment explizit angefordert werden
  - *Selective Acknowledge (SACK)* (RFC 2018)
    - Erlaubt Listen von positiven und negativen Quittungen für mehrere Segmente

Im Gegensatz zu Selective Repeat, das nie zum vollwertigen Internet-Standard wurde, gilt Selective Acknowledge heutzutage als Standard.

## Reaktion des Empfängers

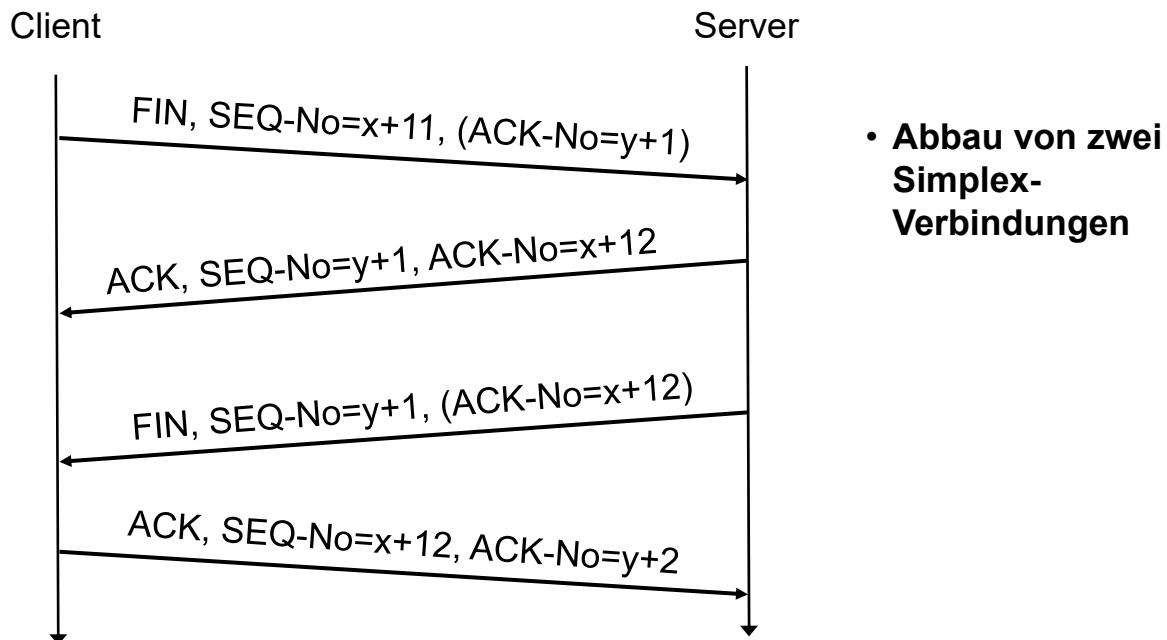
Ereignis beim Empfänger	Aktion beim Empfänger
Ankunft eines Segments mit der erwarteten Sequenznummer. Alle Daten bis dahin sind schon bestätigt worden.	Delayed ACK. Warte bis zu 500ms auf das nächste Segment. Wenn dieses nicht empfangen wird, verschicke die Bestätigung.
Ankunft eines Segments mit der erwarteten Sequenznummer. Allerdings wurde noch keine Bestätigung des vorigen Segments verschickt.	Sofortiges Verschicken einer kumulativen Quittung, die beide Segmente bestätigt.
Ankunft eines Segments hinter der erwarteten Segmentnummer. Es wird eine Lücke entdeckt.	Sofortiges Verschicken eines Duplicate ACK (DUP-ACK). Dieses gibt die erwartete Segmentnummer an.
Ankunft eines Segments, das eine Lücke teilweise oder vollständig füllt.	Sofortiges Verschicken einer Quittung für den gesamten nun vollständigen Teil des Bytestroms.

TCP kann Quittungen auch absichtlich verzögern, um nicht zu viel Overhead durch Quittungsnachrichten zu erzeugen. Es wird nicht auf jedes korrekte Segment mit einem ACK reagiert, sondern nur auf jedes zweite (oder noch seltener, je nach Konfiguration). Hierbei muss man allerdings drauf achten, keinen versehentlichen Timeout zu erzeugen.

Nur in Fehlersituationen oder bei Behebung einer Fehlersituation erfolgt eine sofortige Reaktion.

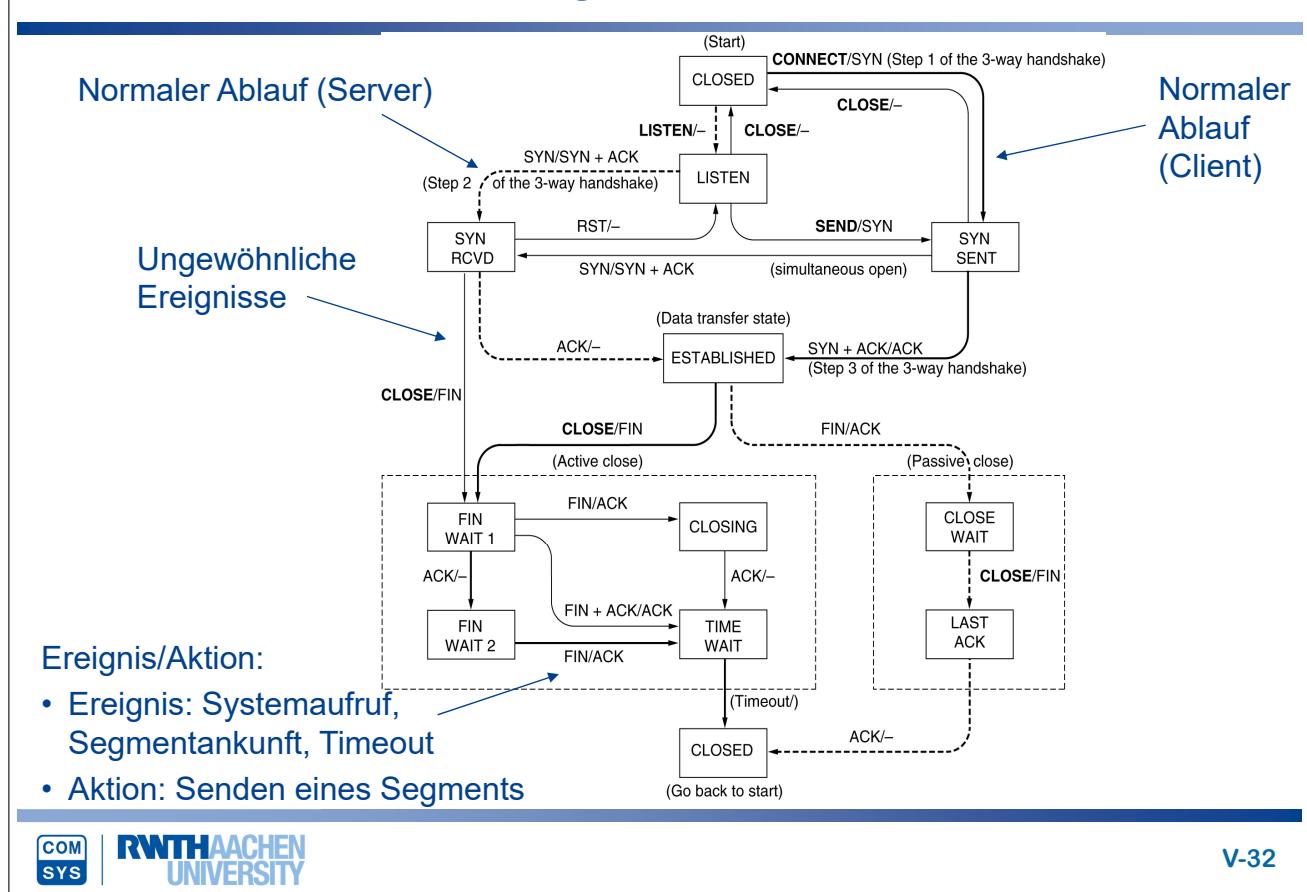
DUP-ACKs werden im Zusammenhang mit der Staukontrolle benötigt, siehe weiter unten.

## TCP-Verbindungsmanagement: 3. Verbindungsende



Wie der Verbindungsaufbau, wird auch der Verbindungsabbau durch das Setzen eines Flags (FIN) markiert. Der Abbau der beiden Übertragungsrichtungen kann hierbei unabhängig voneinander erfolgen – im obigen Beispiel kann der Client nach Austausch der ersten beiden Nachrichten keine Daten mehr an den Server versenden, der Server könnte aber noch – falls nötig – Daten an den Client senden. Wiederum werden Timer und Neuübertragung zum Schutz gegen Paketverlust verwendet.

# Gesamte TCP-Verbindung



Anhand von TCP kann man gut einen Bogen zurück zum ersten Kapitel schlagen – die Instanzen, die den Dienst der zuverlässigen, verbindungsorientierten Datenübertragung von TCP erbringen, implementieren einen Automaten, anhand dessen der Protokollablauf modelliert ist. Hierbei gibt es einen einzigen Automaten, der das Verhalten einer TCP-Instanz beschreibt; je nachdem, ob eine Instanz sich aktiv oder passiv verhält, werden unterschiedliche Übergänge des Automaten verwendet. Während wir in Kapitel 1 Dienstprimitive verwendet haben, um die Übergänge zu beschriften, wird hier nur dargestellt, welche Ereignisse eintreten, nicht welche Primitive dazu genau verwendet werden.

(Automaten werden auch zur Modellierung der anderen behandelten Protokolle verwendet – allerdings wurde bis hier gewartet, um noch einmal Bezug darauf zu nehmen, da TCP über die größte Funktionalität verfügt und daher den interessantesten Automaten hat. Bitte beachten: hinter dem Zustand „Established“ verbirgt sich ein komplexer Teilautomat, der die im Folgenden noch dargestellten Funktionalitäten der Datenübertragungsphase spezifiziert.)

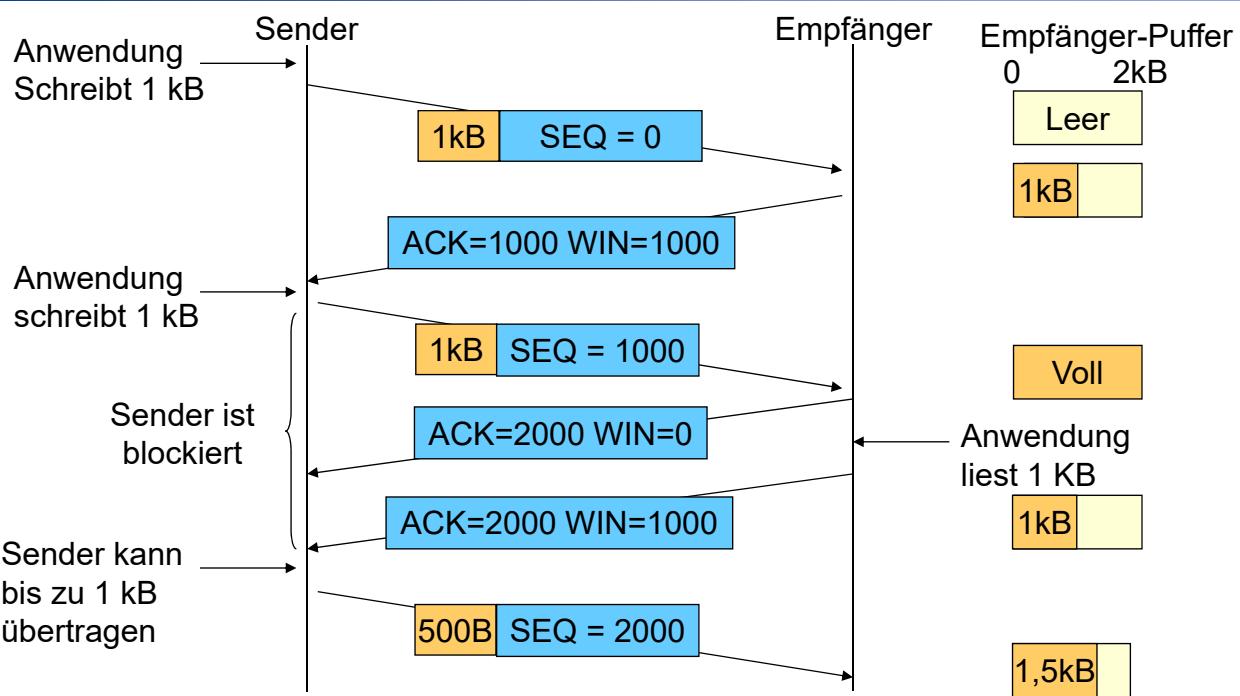
Etwas seltsam erscheinen mag der TIME-WAIT-Zustand – die Verbindung ist abgebaut und damit sind auch alle Daten ausgetauscht. Hier findet allerdings gewollt eine verzögerte Freigabe des Sockets statt um zu vermeiden, dass die gleiche IP-Adressen/Port-Kombination direkt von einer neuen Anwendung verwendet werden kann. Die Idee dahinter ist zu vermeiden, dass eventuell verspätet noch Paket-Duplikate der alten Verbindung eintreffen, die als Elemente der neuen Verbindung betrachtet werden und im schlimmsten Fall zu einem Rücksetzen der neuen Verbindung führen.

(Für die Praxis heißt dies: nicht wundern, wenn beim Testen von TCP-basierten Anwendungen nach Beendigung einer Verbindung nicht direkt eine neue Verbindung aufgebaut werden kann. Einfach etwas warten...)

## Zustände der TCP-Verbindung

State	Description
CLOSED	Keine aktive Kommunikation
LISTEN	Passiver Modus – warten auf Connection Requests
SYN RCVD	Warten auf die dritte Nachricht des Verbindungsaufbaus
SYN SENT	Warten auf die zweite Nachricht des Verbindungsaufbaus
ESTABLISHED	Verbindung aufgebaut, Datenaustausch
FIN WAIT 1	Anwendung hat Verbindungsabbau gestartet
FIN WAIT 2	Einseitiger Verbindungsabbau quittiert
TIME WAIT	Warten auf Duplikate
CLOSING	Simultaner Verbindungsabbau; warte auf Quittung der anderen Seite
CLOSE WAIT	Verbindung einseitig abgebaut
LAST ACK	Warten auf Quittung für den Abbau der zweiten Richtung

## Datenübertragung: Sliding Window mit variablem Fenster



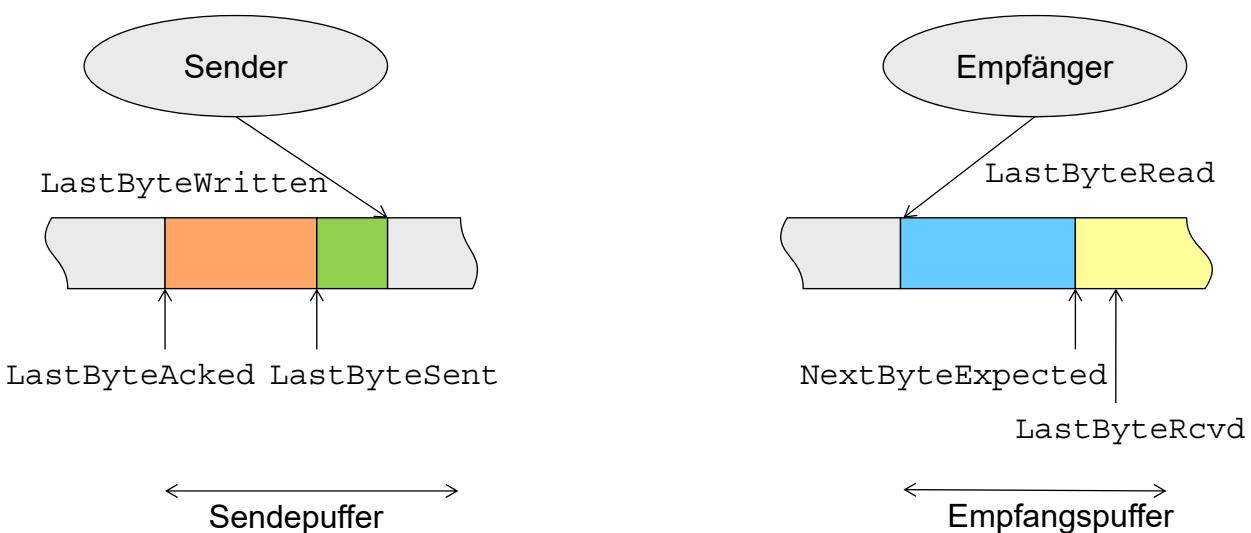
Im Window-Feld des Headers (WIN) kann der Empfänger die Fenstergröße verändern

Während der Phase der Datenübertragung wird wie auf Schicht 2 das Sliding-Window-Verfahren zur Flusskontrolle verwendet. Es gibt allerdings einen wesentlichen Unterschied zu dem Verfahren auf Schicht 2: TCP muss mehrere Verbindungen simultan verwalten können. Der insgesamt zur Verfügung stehende Bufferspeicher muss zwischen allen Verbindungen aufgeteilt werden. Darüber hinaus werden Daten auf Schicht 2 schnellstmöglich an Schicht 3 (IP) weitergegeben, um dort verarbeitet zu werden. Bei TCP verbleiben die Daten im Buffer, bis die Anwendung (bzw. das Protokoll der Anwendungsschicht) die Daten aus dem Socket ausliest. Dies heißt insgesamt: eventuell muss der maximal zur Verfügung stehende Bufferspeicher einer TCP-Verbindung im Laufe der Zeit an die Zahl der Verbindungen angepasst werden, und es hängt von der Lesegeschwindigkeit und -frequenz der Anwendung ab, wie stark sich die Größe des freien Bufferspeichers im Laufe der Zeit verändert. Der Sender muss also permanent über die Größe des aktuell zur Verfügung stehenden freien Bufferspeichers informiert werden.

Während des Verbindungsaufbaus teilt der Empfänger dem Sender die Größe des zur Verfügung stehenden Bufferspeichers mit, der exklusiv für diese Verbindung reserviert wurde. Den entsprechenden Wert nimmt der Sender als Fenstergröße des Sliding-Window-Verfahrens. In jeder Quittung teilt der Empfänger dem Sender mit, wieviel Platz aktuell noch vorhanden ist, um weitere Daten zu empfangen. Im obigen Beispiel dargestellt ist die Situation, dass der Buffer auf Empfängerseite voll läuft; in dem Fall kann sogar eine neue Fenstergröße von Null vorgegeben werden, der Sender wird gebremst. (Analog zum Beispiel der HALT-/WEITER-Kommandos im Foliensatz zu Schicht 2.)

Es besteht allerdings die Gefahr, dass die spätere Meldung des Empfängers, dass wieder Speicher vorhanden ist, verloren geht – um zu vermeiden, dass der Empfänger auf neue Daten wartet, während der Sender auf eine (verloren gegangene) Sendefreigabe wartet, wird wieder ein Timer eingesetzt (Persistence Timer). Dieser wird beim Sender gestartet, sobald ein Segment eingeht, das eine Fenstergröße von Null vorgibt. Nach Ablauf des Timers wird der Sender das nächste Segment abschicken und auf die Antwort des Empfängers warten, in der wieder die aktuelle Fenstergröße enthalten ist.

## Verwaltung des Fensters



Sockets stellen eine Datenstruktur dar, die über einen Bufferspeicher verfügt. Dieser wird in Sende- und Empfangsbufferspeicher unterteilt. Eine Anwendung schreibt Daten in den Sendebuffer und liest Daten aus dem Empfangsbufferspeicher. Daten verbleiben im Sendebuffer, bis sie erfolgreich übertragen wurden, d.h. bis sie von der empfangenden TCP-Instanz quittiert und im dortigen Empfangsbufferspeicher abgespeichert wurde (bis die empfangende Anwendung sie entnimmt).

Zur Verwaltung des Bufferspeichers werden Zeiger eingesetzt, die auf die zur Verwaltung wichtigen Bufferpositionen zeigen und nach Ankunft eines Datensegments bzw. einer Quittung sowie bei einer Schreib-/Leseoperation einer Anwendung verrückt werden:

### Senderseite

- LastByteWritten zeigt auf Position des Sendebuffers, bis zu der die Anwendung Daten geschrieben hat.
- LastByteSent hält mit, bis zu welcher Position im Bytestrom bereits Daten versendet wurden, LastByteAcked hält mit, bis zu welcher Position der Bytestrom bereits quittiert wurde. Alle vorherigen Bytes wurden erfolgreich zugestellt und aus dem Sendebuffer gelöscht. Dieser Bufferspeicher kann also von der Anwendung neu beschrieben werden (Ringbuffer).
- Die Daten zwischen LastByteAcked und LastByteWritten sind bereits gesendet, aber noch nicht quittiert. Die Menge dieser unquittierten Daten darf die Buffergröße des Empfängers nicht überschreiten.

### Empfängerseite

- NextByteExpected gibt an, bis zu welcher Position der Bytestrom bereits komplett empfangen wurde.

- LastByteRead gibt an, bis zu welcher Position die empfangende Anwendung die Daten bereits ausgelesen hat.
- Die Menge der Daten zwischen LastByteRead und NextByteExpected belegt noch den Empfangsbuffer, die Differenz zur Gesamtgröße des Buffers entspricht dem noch freien Fenster.
- Falls Go-Back-N zur Fehlerbehebung verwendet wird, werden Segmente, deren Sequenznummer nicht "NextByteExpected" entspricht, verworfen. Es ist keine weitere Verwaltung notwendig. Wird stattdessen Selective Repeat eingesetzt, werden auch Segmente außerhalb der erwarteten Reihenfolge gespeichert und es sind zwei weitere Zeiger notwendig (hier nicht dargestellt), um Anfang und Ende solch eines Teilbytestroms festzuhalten. Eventuell sind sogar mehrere solche Zeigerpaare notwendig, falls vereinzelte Segmente empfangen werden.
- Mittels LastByteRcvd kann festgehalten werden, bis zu welcher Position des Bytestroms bereits Daten empfangen wurden (Ende des letzten Segments, welches außerhalb der erwarteten Reihenfolge empfangen wurde). Dies ist nur dann notwendig, wenn das "Selective ACK (SACK)" -Verfahren benutzt wird.

## Verwaltung des Fensters

- Verwendung der Zeiger bei der Flusskontrolle:

- ▶ Senderseite:

- Größe des Senderbuffers: **MaxSendBuffer**
    - **LastByteSent - LastByteAcked ≤ AdvertisedWindow**
    - **EffectiveWindow = AdvertisedWindow - (LastByteSent - LastByteAcked)**
    - **LastByteWritten - LastByteAcked ≤ MaxSendBuffer**
    - Blockiere den Versuch der Anwendung,  $y$  Byte zu schreiben, falls  $(LastByteWritten - LastByteAcked) + y > \text{MaxSenderBuffer}$

- ▶ Empfängerseite:

- Größe des Empfängerbuffers: **MaxRcvBuffer**
    - **LastByteRcvd - LastByteRead ≤ MaxRcvBuffer**
    - **AdvertisedWindow =**
      - **MaxRcvBuffer - ((NextByteExpected - 1) - LastByteRead)** [bei Go-Back-N, Selective Repeat]
      - **MaxRcvBuffer - (LastByteRcvd - LastByteRead)** [bei SACK]

AdvertisedWindow ist die Angabe, die im WIN-Feld jeder Quittung an den Sender übertragen wird.

EffectiveWindow ist der dem Sender noch zur Verfügung stehende freie Teil des Fensters; maximal so viele Byte dürfen noch übertragen werden, ohne dass eine weitere Quittung eingeht.

# Silly-Window-Syndrom

## • *Silly Window*

- ▶ Sender wird durch ACK mit WIN=0 gestoppt
- ▶ Empfänger liest 1 Byte aus Buffer
  - Wiederholung des vorherigen ACKs mit WIN=1
- ▶ Sender schickt 1 Byte (plus 20 Byte TCP-Header + 20 Byte IP-Header)
  - ACK mit WIN=0
- ▶ ...
- ▶ Unnötige Netzbelastung!

## • Lösungen

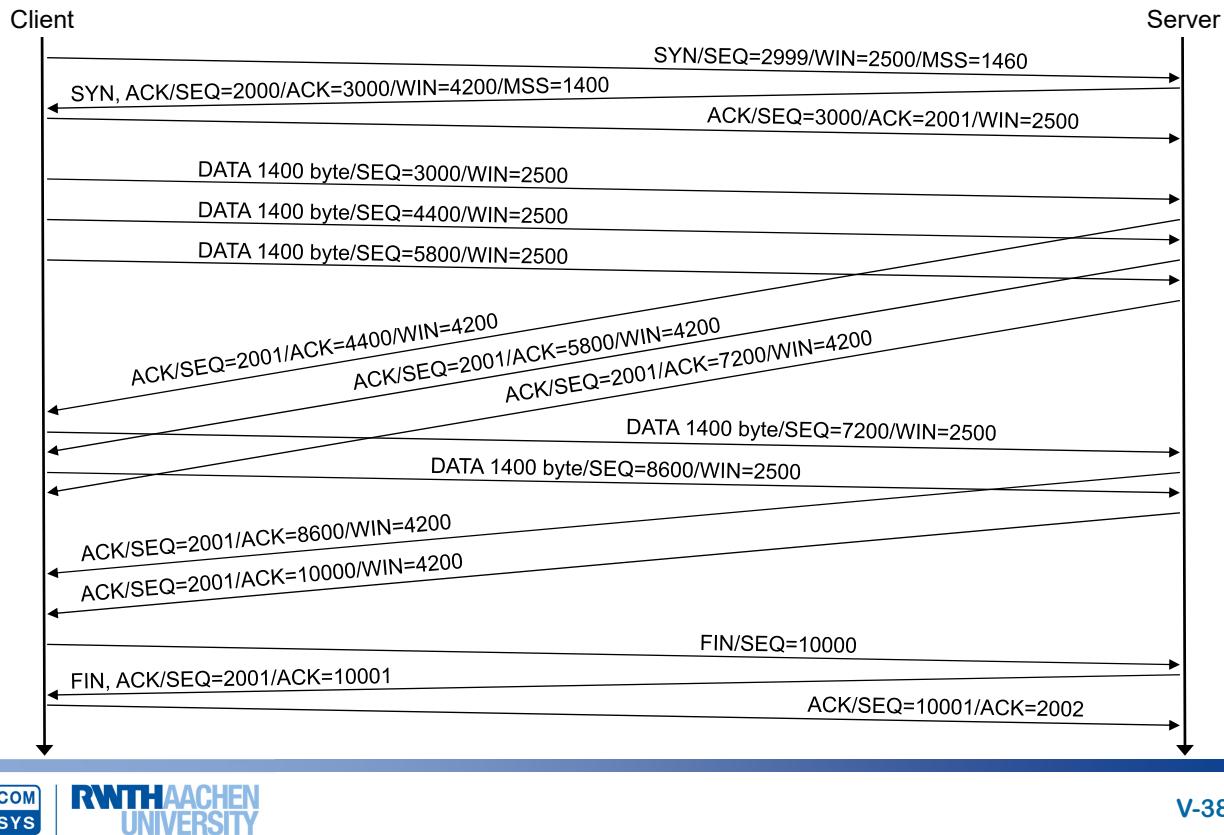
- ▶ *Nagle's Algorithmus*: Sender wartet, bis Segment maximaler Größe gesendet werden kann
- ▶ *Clark's Solution*: Empfänger verzögert ACK, bis Buffer halb leer ist oder Platz für ein maximales Segment ist

Bei stupider Anwendung des Fenstermechanismus‘ kann es zum Silly-Window-Syndrom kommen: der Empfängerbuffer wird komplett gefüllt, da die empfangende Anwendung keine Daten entnimmt. Laut Protokoll wird der Sender durch ein ACK mit WIN=0 gestoppt. Entnimmt die Anwendung nur langsam Daten, kann es nun sein, dass nur wenige Byte aus dem Buffer gelesen werden. Der Sender wird entsprechend entsperrt und kann ein sehr kleines Segment schicken, so dass durch die TCP- und IP-Header sehr viel Overhead erzeugt wird. Danach ist der Empfängerbuffer wieder voll. Wiederholt sich dieses Spiel, da die Anwendung kurz hintereinander immer wieder kleine Mengen an Daten entnimmt, wird die Übertragung ineffizient.

Für Abhilfe sorgen z.B. die Lösungen von Nagle (Senderseite) und Clarke (Empfängerseite). Nagle’s Algorithmus hat sich durchgesetzt und soll laut Standard von jeder TCP-Implementierung per default aktiviert sein.

Bei Nagle’s Algorithmus versendet TCP die von einer Anwendung übergebenen Daten nur dann sofort, wenn momentan keine Daten unbestätigt unterwegs sind oder wenn ein Segment maximaler Größe (oft 1460 Byte) versendet werden kann. Liegen weniger Daten vor und es stehen noch Bestätigungen des Empfängers aus, verzögert TCP die Versendung. Dadurch können Daten eventuell sehr stark verzögert werden. Implementiert man eine Anwendung, die geringe Mengen an Daten sofort versenden soll, muss man darauf achten, den Buffer zu flushen, also TCP mitzuteilen, Nagle’s Algorithmus zu ignorieren und trotzdem ein kleines Segment zu senden.

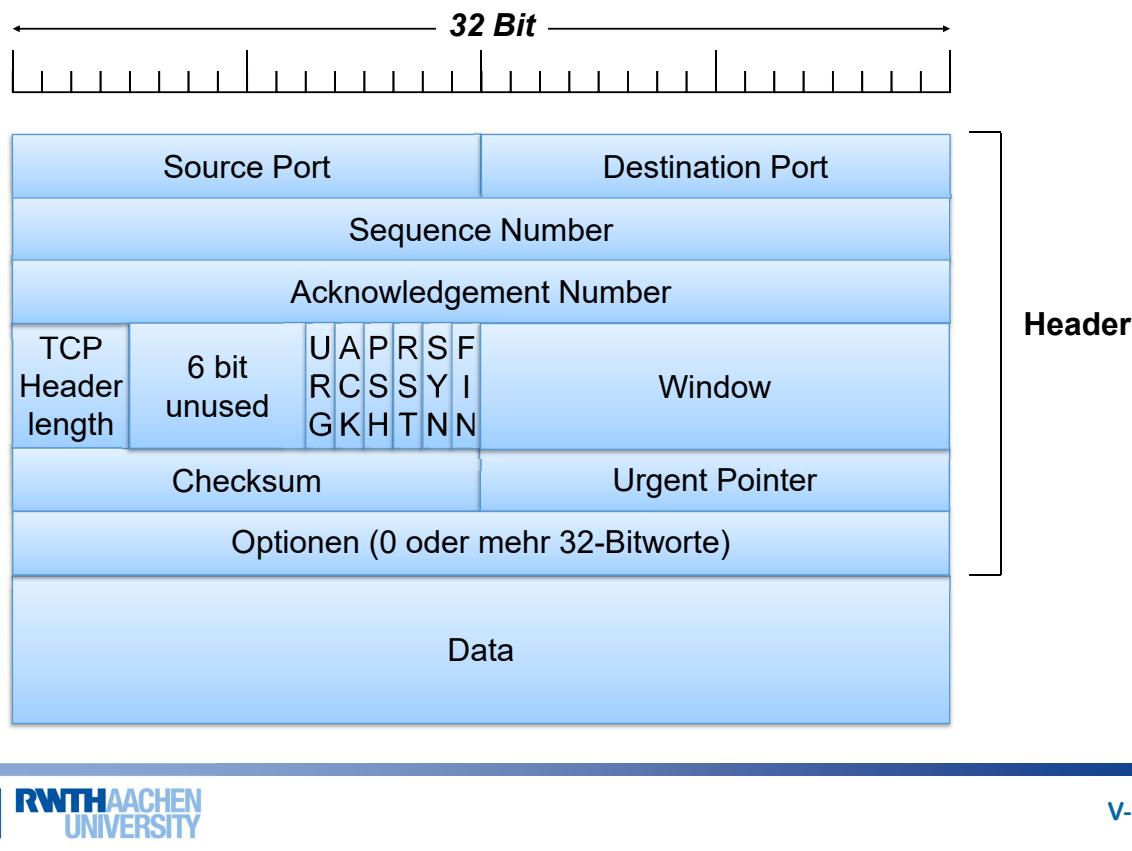
## Die gesamte TCP-Verbindung...



Beispiel für eine fehlerfreie TCP-Übertragung, bei der die Daten auf Empfängerseite immer direkt verarbeitet werden und das Fenster damit immer die maximale Größe hat.

Mit enthalten in den ersten beiden Nachrichten des Verbindungsaufbaus ist hier auch der Parameter MSS (Maximum Segment Size) – die beiden Kommunikationspartner einigen sich hier auf die maximale Größe des Payloads eines Segments während der Datenaustauschphase. Die MSS wird auf jeder Seite so gewählt, dass die resultierenden IP-Pakete nie die Größe der lokal verwendeten MTU überschreiten, um IP-Fragmentierung zu vermeiden. Das Minimum der beiden vorgeschlagenen Werte wird verwendet.

## TCP-Segment: Aufbau

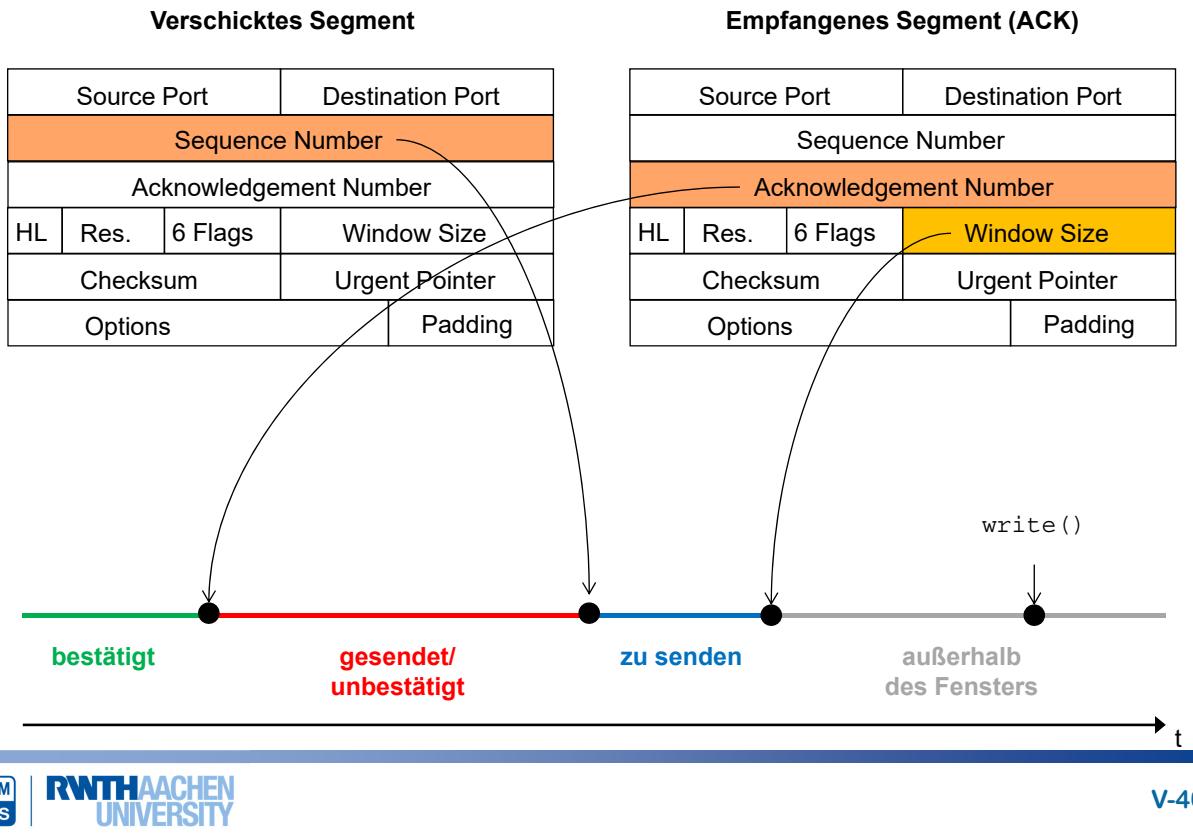


Bedeutung der Felder des TCP-Headers:

- **Source- und Destination-Port:** Portnummer von Sender bzw. Empfänger
- **Sequence Number/Acknowledgement Number:** 32-Bit-Sequenz- und Quittungsnummer für die Flusskontrolle
  - Sequenz- und Quittungsnummer zählen einzelne Bytes
  - Die Quittungsnummer gibt das nächste erwartete Byte an
  - Sequenznummern fangen nicht notwendigerweise bei 0 an, der Anfangswert wird beim Verbindungsauftakt festgelegt
  - Piggybacking: eine Quittung kann in einem Datensegment mitgeschickt werden
- **HL:** Header Length: der TCP-Header hat variable Länge: es können Optionen angegeben werden. Eine Angabe der Länge ist nötig (in 32-Bit-Worten), damit der Empfänger weiß, was noch Optionen sind und wo im Segment der Datenteil beginnt.
- **Res** = Reserved. Für spätere Verwendung freigehaltene Bits. Mittlerweile sind zwei dieser Bits als weitere Flags standardisiert; dies wird später noch erläutert.
- Sechs Flags:
  - **URG:** URGENT – gesetzt, wenn das Feld Urgent Pointer beachtet werden soll, um „wichtige“ Daten außerhalb der Flusskontrolle zu senden
  - **ACK:** gesetzt, wenn eine Quittung mitgesendet wird. Nur bei gesetztem ACK-Flag wird der Empfänger des Segments den in Acknowledgement Number befindlichen Wert beachten.
  - **PSH:** PUSH – direkte Weiterleitung der Daten, kein Warten auf weitere Daten (d.h. Versenden auf Senderseite, Verarbeitung auf Empfängerseite)
  - **RST:** RESET – Rücksetzen einer Verbindung, z.B. falls eine Verbindungsabfrage nicht akzeptiert werden kann
  - **SYN:** gesetzt beim Aufbau einer Verbindung
  - **FIN:** gesetzt, um die Verbindung zu schließen

- *Window Size (WIN)*: Größe des Bufferspeichers für die Verbindung - gibt an, wie viele Bytes der Empfänger aktuell noch puffern kann (AdvertisedWindow)
- *Checksum*: Prüfsumme über Header und Daten. Dient u.a. zur Verifikation, dass das Paket an das richtige Gerät ausgeliefert worden ist.
- *Urgent Pointer*: in dringenden Fällen können Daten ohne Beachtung der Flusskontrolle gesendet werden. Dieses Feld gibt an, wo die „dringenden Daten“ enden (Byteversatz von der aktuellen Folgenummer)

# Flusskontrolle: Senderseite



In Ergänzung zu der vorherigen Folie zum Sende- und Empfangsbuffer sind hier der Bytestrom als Ganzes und der Bezug der Header-Angaben zu diesem Bytestrom dargestellt.

# TCP-Header: Protokollevolution

## • TCP-Optionen

- ▶ Angaben zur Kontrolle/Verwaltung einer Verbindung
  - Erlaubt Weiterentwicklung des „alten“ TCP
    - Stark genutzt, da nur Endsysteme geändert werden müssen
- ▶ Wichtige Optionen:
  - **Maximum Segment Size (MSS)**: Vereinbarung einer maximalen Segmentgröße beim Verbindungsauftbau
  - **Window Scale**: Skalierung der Fenstergröße durch Angabe eines Faktors beim Verbindungsauftbau, mit dem der Wert in WIN multipliziert wird
    - Logarithmische Angabe, maximaler Wert: 14
      - maximaler WIN-Wert: 65535, maximaler Skalierungsfaktor: 14
      - Damit:  $65535 \cdot 2^{14} = 1.073.725.440$  Byte max. Fenstergröße
  - Vereinbarung des **Fehlerbehebungsmechanismus** für eine Verbindung
    - Go-Back-N, Selective Repeat, SACK, ...

TCP verfügt über kein Versionsfeld zur Weiterentwicklung des Protokolls; stattdessen werden Änderungen durch Optionen umgesetzt. Dies ist einfacher als eine Änderung auf IP-Ebene, da zur Änderungen von TCP nur die Betriebssysteme von Endgeräten aktualisiert werden müssen, nicht auch alle Router. Die Optionen bei TCP werden heute stark genutzt, um veraltete Einstellungen und Mechanismen des ursprünglichen TCP abzuändern.

Über Optionen können beim Verbindungsauftbau Parameter der nachfolgenden Verbindung ausgehandelt werden, z.B.:

- Welcher Mechanismus soll zur Übertragungswiederholung verwendet werden? Go-Back-N, SACK, ...
- Welche Maximum Segment Size soll verwendet werden?
- Window Scaling Factor – leider ist die Angabe einer Fenstergröße bei TCP durch 16 Bit beschränkt – ein größeres Fenster als 64 kByte kann nicht eingetragen werden, was bei den Datenraten heutiger Netze unbefriedigend ist. Daher können die Kommunikationspartner einen Skalierungsfaktor vereinbaren, mittels dessen der Wert des WIN-Feldes vervielfacht werden kann.

Aber Optionen können auch während der Datenaustauschphase verwendet werden. Zu beachten ist jedoch, dass ein IP-Paket maximal 1500 Byte umfassen soll, so dass mit steigender Zahl an Optionen der TCP-Payload geringer wird.

## Berechnung der Prüfsumme

- **Prüfsumme: Validierung der Korrektheit der empfangenen Daten**
  - ▶ Berechnung über Pseudo-Header + TCP-Header + Daten
  - ▶ Einerkomplement der Summe aller 16-Bit-Worte
- *Pseudo-Header:*

Source address (IP)		
Destination address (IP)		
00000000	Protocol = 6	Länge des TCP-Segments

- ▶ Prüfsumme berücksichtigt auch Korrektheit der IP-Adressen

Während IP nur den Header auf Korrektheit prüft, führt TCP auch eine (einfache) Sicherung des Payloads durch. Zudem geht ein sogenannter Pseudo-Header mit in die Berechnung der Prüfsumme ein, durch den auch die IP-Adressen noch einmal mit eingeschlossen werden. Hierdurch soll vermieden werden, dass durch unentdeckte Fehler auf tieferer Ebene (Verfälschung der IP-Adressen) ein Paket an den falschen Host ausgeliefert wird.

# Kapitel 5: Transportschicht

- **Protokollmechanismen der Transportschicht**

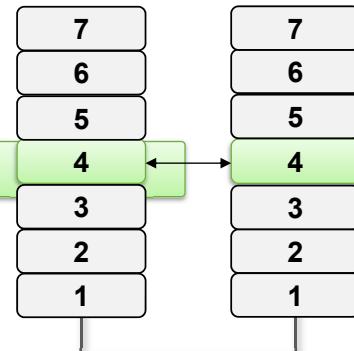
- ▶ Prozessadressierung, strombasierte vs. paketbasierte Kommunikation, verbindungsorientiert/verbindungslos

- **Die Transportschicht im Internet**

- ▶ TCP (Transmission Control Protocol)

- Adressierung, Sockets
- TCP-Verbindung
- Flusskontrolle
- TCP-Header
- **Staukontrolle (Congestion Control)**

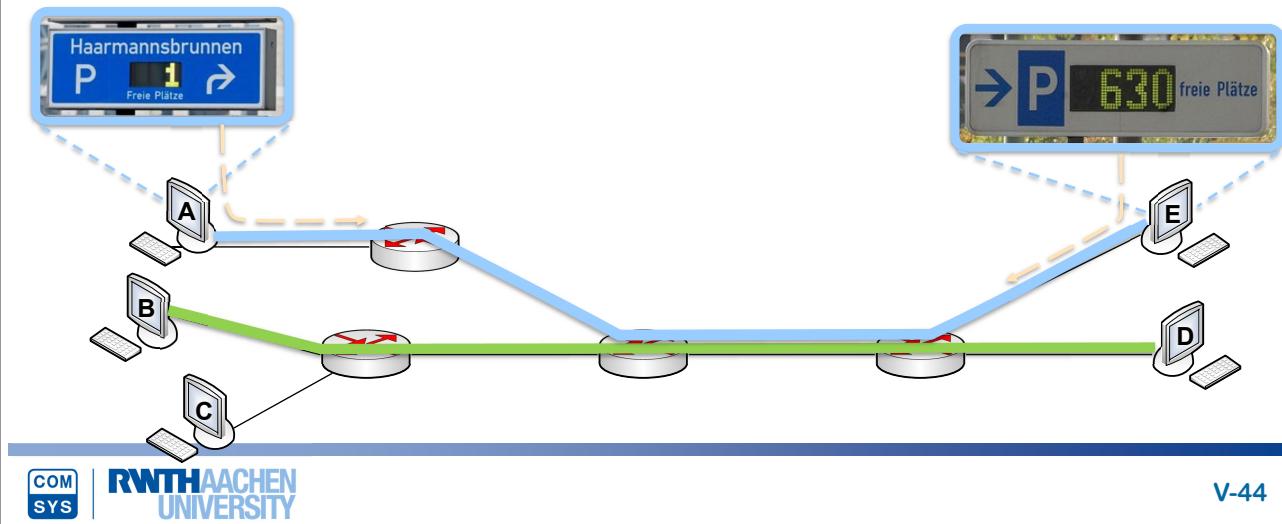
- ▶ UDP (User Datagram Protocol)



# Flusskontrolle

- **Wiederholung:**

- ▶ Flusskontrolle regelt die Datenmenge zwischen den Partnern einer TCP-Verbindung
- ▶ Ziel: *Keine Überlastung des TCP-Partners* innerhalb einer Verbindung
- ▶ Aber: Das verhindert keine *Überlastung des Netzes!*



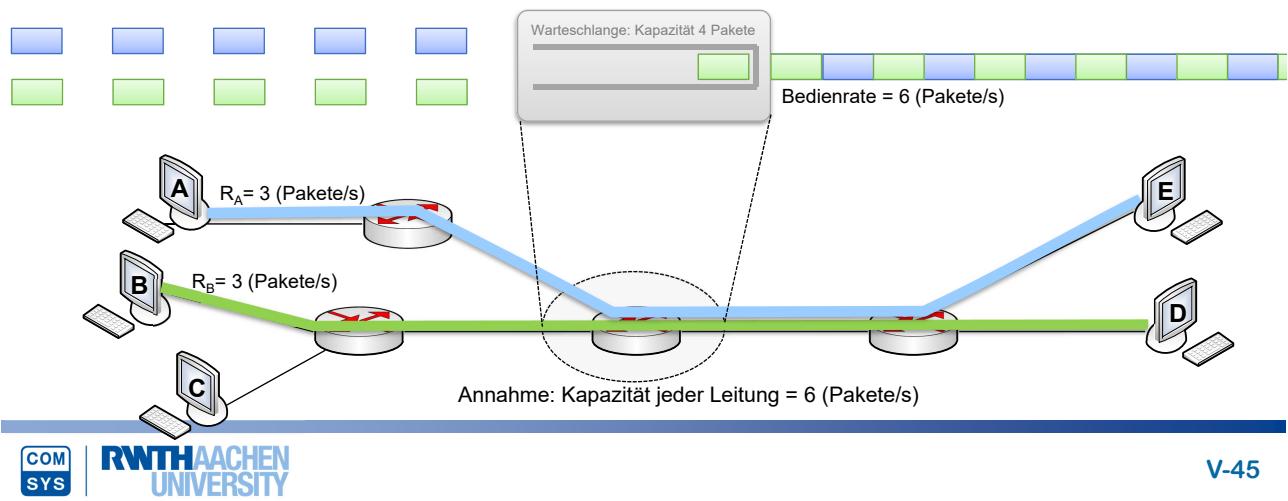
Die bislang vorgestellte Funktionalität von TCP entspricht im Wesentlichen der Funktionalität von Schicht 2. Allerdings gibt es einen bedeutenden Unterschied zwischen TCP und der Schicht 2: auf Schicht 2 wird nur die Übertragung über eine Leitung betrachtet. TCP setzt auf IP auf, durch welches all diese Teilstrecken zu einem komplexen Netz verbunden werden. TCP hat zwar logisch gesehen auch seinen Kanal (die Verbindung), kennt allerdings nicht die zugrundeliegende Netzstruktur. Auf IP-Ebene kann es zum Datenstau kommen (engl. Congestion), falls ein Router über all seine Leitungen in Summe mehr Daten empfängt, als er verarbeiten (d.h. weiterleiten) kann. Als Resultat wird der entsprechende Router eingehende IP-Pakete schlicht verwerten.

Auf TCP hat dies die Auswirkung, dass keine Quittungen mehr eingehen und nach einem Timeout eine Überragungswiederholung initiiert wird. Dadurch wird die Überlastsituation im Netz noch verschlimmert.

Als (etwas hinkende) Analogie kann man hier die Medienzugriffskontrolle auf Schicht 2 herziehen: ist das Netz stark belastet, so dass es zu vielen Kollisionen kommt, werden die Wartezeiten der einzelnen Stationen so erhöht (randomisiert), dass das Kollisionsrisiko verringert wird; aber auch die erzielbare Datenrate der einzelnen sendenden Stationen wird dadurch reduziert.

Ein ähnlicher Mechanismus ist auch bei TCP notwendig, nur dass hier Paketverlust der Indikator eines Problems ist, keine Kollisionen.

# Stausituationen im Internet



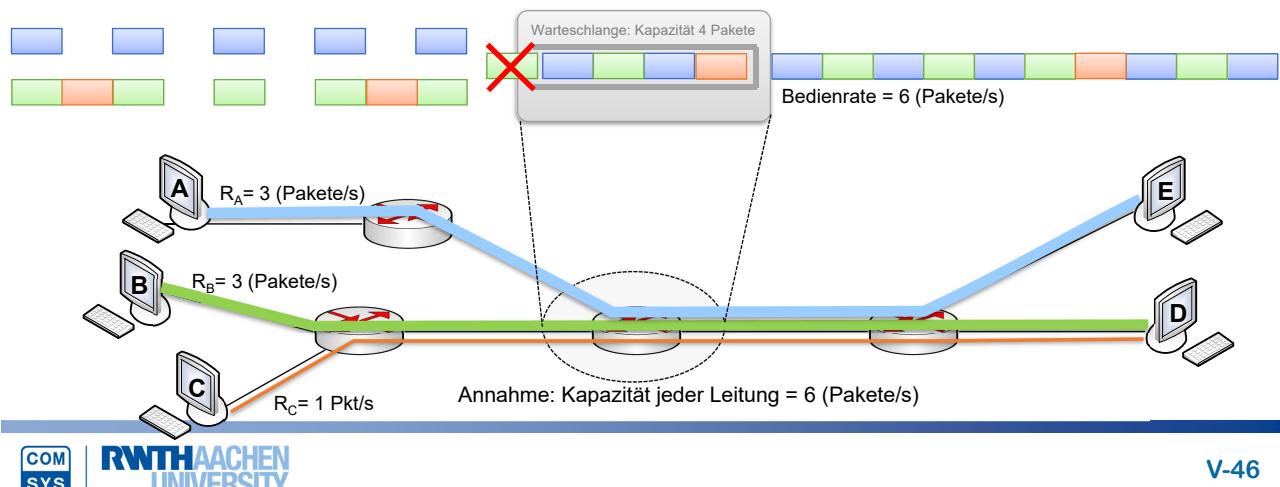
# Stausituationen im Internet – Ursache und Definition

- Überlastung des Kernnetzes: Stau im Internet

- ▶ Pakete können aufgrund von Überlastsituationen nicht mehr gepuffert werden → Paketverlust

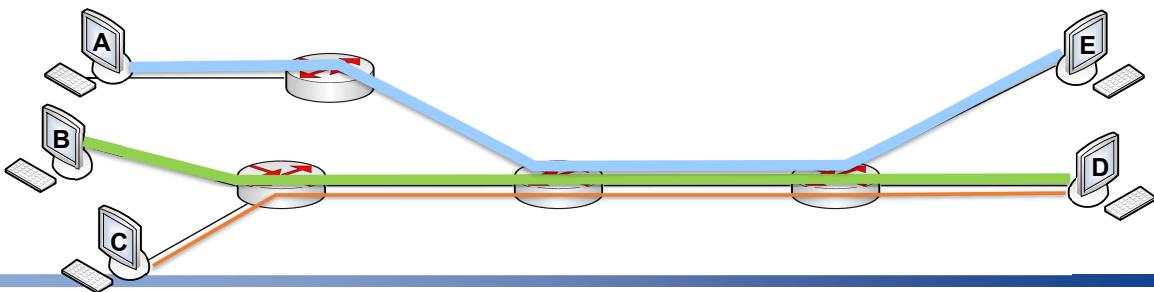
- Wann treten Stausituationen auf?

- ▶ Wenn Summe der Datenraten (Ankunftsichten  $R_j^{in}$ ) auf ausgehendem Port  $i$  größer als die Kapazität (Bedienrate  $R_i^{out}$ ) des Link  $i$  ( $\sum_j R_j^{in} > R_i^{out}$ ) ist und die Warteschlange voll ist



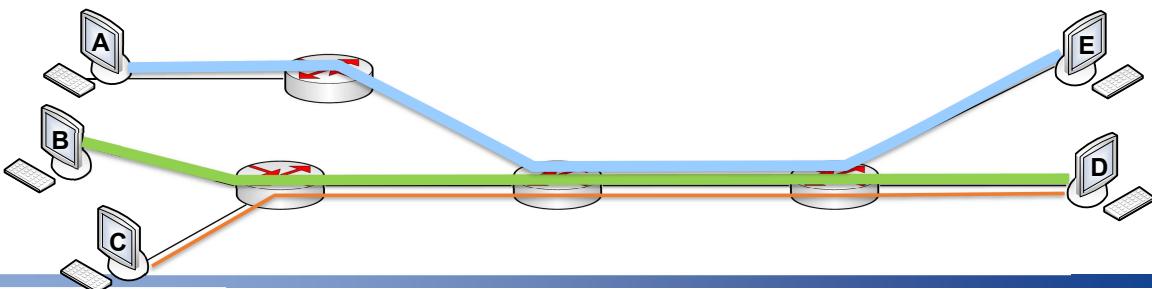
# Stausituationen im Internet – Ursache und Definition

- Überlastung des Kernnetzes: Stau im Internet
  - ▶ Pakete können aufgrund von Überlastsituationen nicht mehr gepuffert werden → Paketverlust
- Wann treten Stausituationen auf?
  - ▶ Wenn Summe der Datenraten (Ankunftsichten  $R_j^{in}$ ) auf ausgehendem Port  $i$  größer als die Kapazität (Bedienrate  $R_i^{out}$ ) des Link  $i$  ( $\sum_j R_j^{in} > R_i^{out}$ ) ist und die Warteschlange voll ist
- Wie erkennt eine TCP-Instanz eine Stausituation?
  - ▶ Sender bemerkt, dass Pakete nicht mehr bestätigt werden (vgl. Kapitel 3, Link Layer)
    - Ausbleiben einer Bestätigung → *Time-Out*
    - Erneute (kumulative) Bestätigung eines Pakets → *Duplicate ACK (DupACK)*



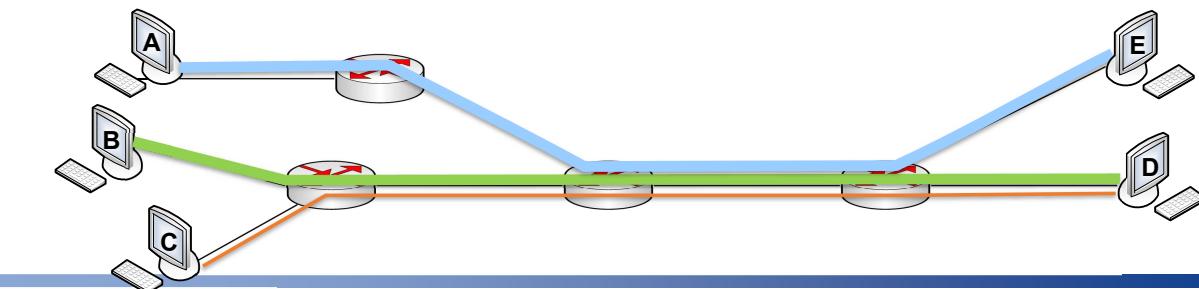
## Reaktion auf Stausituation (Staubehandlung)

- Wie soll ein TCP-Sender auf einen Stau reagieren?
  - ▶ Überlast ist Ursache → *Reduktion der Last*
  - ▶ Welcher Sender reduziert? → *Die Sender, deren Pakete verloren gehen*
  - ▶ Kein Wissen über Netz-Kapazitäten, über Bottleneck-Links, über aktuelle Belastung, über andere Ströme

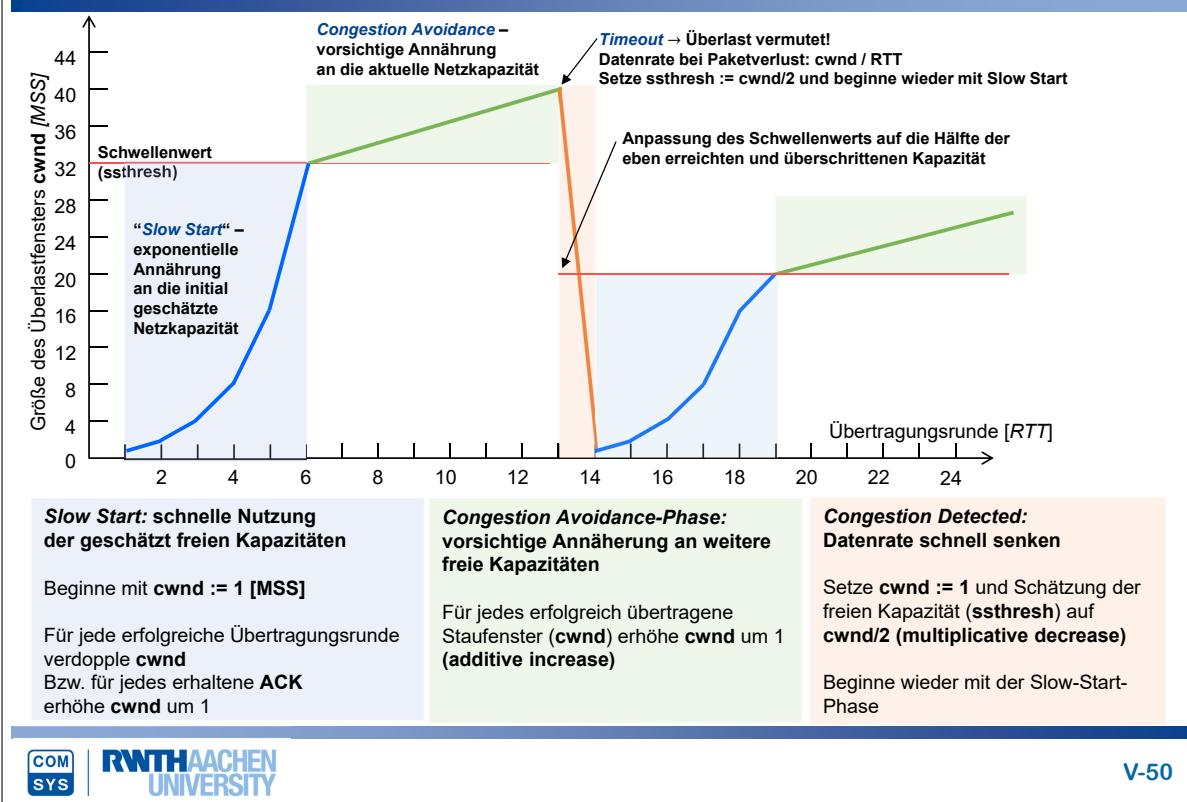


## Reaktion auf Stausituation (Staubehandlung)

- TCP-Staukontrolle (Congestion Control):
  - ▶ Basiert auf grober Schätzung der freien Netzkapazität
  - ▶ Überlastfenster (Congestion Window,  $cwnd$ ) limitiert (neben Sendefenster) die Anzahl der Segmente, die pro Runde ( $RTT$ ) gesendet werden dürfen
  - ▶ Angenommene „sichere“ Kapazität (=  $ssthresh$  /  $RTT$ )
    - ▶ Initial sehr hoch, bzw. auf maximal Größe des Initial Window (Flusskontrolle, angegeben im TCP-Header, auch als  $rwnd$  bezeichnet)



## Beispielauf von Slow Start / Congestion Avoidance



Bemerkungen:

- Die initiale Größe des Congestion-Windows wurde im Laufe der Zeit angepasst, es sind mittlerweile auch größere Werte zulässig. Zur Vereinfachung werden wir in Vorlesung und Übung allerdings immer von einem Startwert von 1 ausgehen.
- Die Benennung „Slow Start“ ist irreführend – man fängt zwar langsam mit nur einem Segment an zu senden, um das Netz nicht direkt zu überlasten, steigert dann aber die Senderate exponentiell, um die freie Netzkapazität schnell ausnutzen zu können.
- Die Bestimmung des Congestion Window erfolgt allein durch den Sender.
- Bei Erreichen des Thresholds wird die Senderate nur noch linear gesteigert, um zu verhindern, dass man durch zu schnelle Steigerung plötzlich das Netz überlastet (Stauvermeidung, Congestion Avoidance).
- Die Verkleinerung des Thresholds auf die Hälfte röhrt daher, dass angenommen wird, dass unter der vorherigen Konfiguration durch exponentielles Wachstum die Kapazität des Netzes stark überschritten wurde. Daher soll beim weiteren Senden ein vorsichtigeres Herantasten bis an die freie Kapazität erfolgen.

Der Überlastkontrollmechanismus wird auch *Additive Increase/Multiplicative Decrease (AIMD)* genannt.

Das Überlastfenster  $cwnd$  wird am Anfang auf ein Segment gesetzt. Nach der erfolgreichen Übertragung eines Segments dürfen zwei weitere Segmente übertragen werden – bei erfolgreicher Übertragung eines vollen Fensters hat sich das Überlastfenster somit verdoppelt. Eine solche Übertragung eines vollen Fensters ist auf dieser Folie als „Übertragungsrunde“ bezeichnet. Die „Übertragungsrunde“ ist allerdings nur eine vereinfachte Darstellung zur Erläuterung des Algorithmus – in der Realität wird der

Wert von cwnd mit jeder eintreffenden Quittung um die Größe der MSS erhöht. Dies entspricht insgesamt in einer gedachten Übertragungsrunde einer Verdopplung der Fenstergröße. Bei fehlerfreier Übertragung steigt die Größe des Überlastfensters also exponentiell an.

Hat das Überlastfenster einen Schwellenwert erreicht, so wird nur noch bei Übertragung eines vollen Sendefensters die Größe des Überlastfensters um eine MSS erhöht (linearer Anstieg). Man tastet sich sozusagen vorsichtig an den maximalen Wert der Netzkapazität heran. Dies ist der „Additive Increase“. Die vorherige Slow-Start-Phase soll nur verhindern, dass der „Additive Increase“ bei 1 beginnen muss – der Start soll beschleunigt werden. Im Prinzip verhindert der Slow-Start-Algorithmus also einen Slow-Start.

Tritt ein Segmentverlust auf, so wird das Überlastfenster wieder auf den Initialwert zurückgesetzt und der Schwellenwert auf die Hälfte des vorher erreichten Überlastungsfensters gesetzt. Diese Halbierung ist der „Multiplicative Decrease“.

Der Mechanismus produziert also immer einen sägezahnartigen Verlauf. Für längere Verbindungen ist er geeignet, um Überlastsituationen zu vermeiden, aber bei kleinen Datenmengen verschenkt man anfangs sehr viel Netzkapazität. Darüber hinaus führt ein Paketverlust zu einem deutlichen Einbruch in der Übertragungsrate und die Verbindung erholt sich nur langsam davon. Heutzutage kommen Varianten des Slow-Start-Mechanismus‘ zum Einsatz, die ebenfalls effektiv bei der Verhinderung von Netzüberlastungen sind, sich aber schneller von Paketverlusten erholen – Ziel ist es, das „Multiplicative Decrease“ nicht nur auf den Schwellwert, sondern auch auf das Überlastungsfenster anzuwenden, also zu verhindern, dass das Fenster auf 1 zurückfällt.

# Stau oder kein Stau?

- **Paketverlust = Stau?**

- **Massive Überlast**

- Viele Pakete werden verworfen
      - Große Reduktion der Last erforderlich
- Indiz: Der Sender erhält keine ACKs, aber es erfolgt Timeout, da keine Datenpakete ankommen

- **Leichte Überlast**

- Einzelne Pakete werden verworfen
      - Große Reduktion der Rate wäre übertrieben
- Indiz: Der Sender erhält Duplicated-ACKs, da weitere Pakete durchkommen und bestätigt werden

- **Übertragungsfehler (z.B. in drahtlosen Netzen)**

- Einzelne Pakete sind defekt
      - Reduktion der Rate nicht erforderlich
- Indiz: Der Sender erhält Duplicated-ACKs

Timeout → Slow Start

TCP Tahoe  
(erste Version TCP):  
Triple DupACK → Slow Start

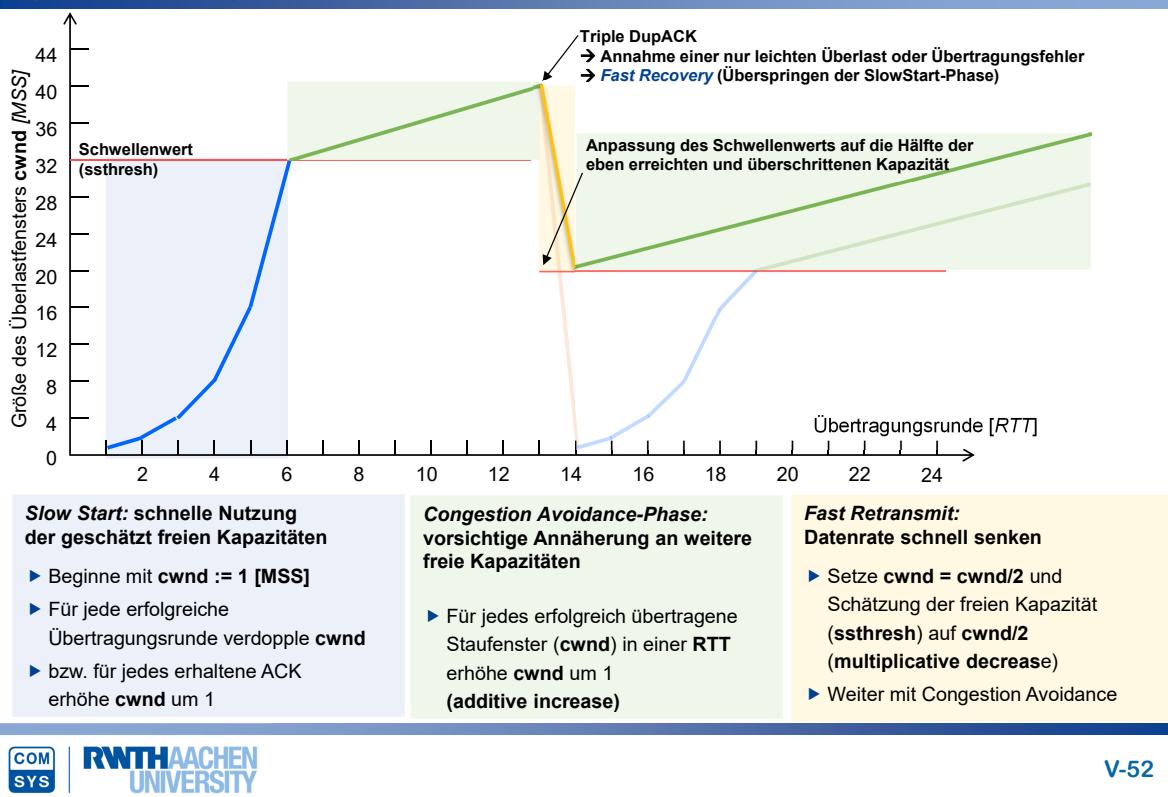
TCP Reno  
(zweite TCP Version)  
Triple DupACK → Fast Recovery

Gehen nur einzelne Segmente verloren, ohne dass eine wirkliche Überlastsituation vorliegt, reagiert der Slow-Start-Algorithmus falsch – eine drastische Reduktion der Datenrate ist nicht nötig. Um zu vermeiden, dass der Slow-Start ausgelöst wird, schickt der Empfänger bei jedem erhaltenen Segment eine Quittung zurück; kann er keine positive Quittung schicken, da ein Segment fehlt, sendet er die zuletzt gesendete Quittung noch einmal. Der Sender erhält dadurch Feedback: es liegt keine vollständige Überlast vor, da immer noch einige seiner Segmente beim Empfänger ankommen. Daher kann er eine direkte Wiederholung des fehlenden Segments initiieren.

Der Sender reagiert aber erst beim dritten DUP-ACK, welches er empfängt. Da IP keine reihenfolgetreue Auslieferung der Daten garantiert, kann es sein, dass ein DUP-ACK bedeutungslos ist, da kurz nach seiner Versendung die angemahnten Daten doch eintreffen. Dies würde zu einer unnötigen Wiederholung und damit Belastung des Netzes führen. Daher reagiert der Sender erst nach dem Eintreffen mehrerer Duplikate einer Quittung. Er wiederholt das anscheinend fehlende Segment in der Hoffnung, dass dadurch noch vor dem Timeout für das verlorengegangene Segment ein ACK zurückkommt und Slow Start vermieden wird. Dieser Mechanismus wird „Fast Retransmit“ genannt.

Unterschiedliche TCP-Varianten unterscheiden sich vorrangig im verwendeten Congestion-Control-Algorithmus. Basierend auf dem implementierten Algorithmus haben die TCP-Varianten Namen, traditionell nach Städten oder Regionen benannt.

## Slow Start/Congestion Avoidance + Fast Recovery (TCP Reno)



Bei Eintreffen des dritten DUP-ACK wird nicht nur eine sofortige Neuübertragung vorgenommen, die Verbindung wird auch eingefroren. Der Schwellwert wird um die Hälfte reduziert und das Überlastfenster auf die Größe des Schwellwertes gesetzt, um die Datenrate zu reduzieren. Mit jedem weiteren einkommenden DUP-ACK wird eine Überlastsituation unwahrscheinlicher, da immer mehr Segmente korrekt ankommen. Daher wird das Überlastfenster an die Zahl ankommender Pakete angepasst und eventuell noch weiteres Senden erlaubt. Trifft eine Quittung für das fehlende Segment ein (und vermutlich direkt auch für viele folgende Segmente), setze das Überlastfenster zurück auf den Threshold, um die Datenrate wieder vorsichtig weiter steigern zu können.

Durch dieses Vorgehen wird ein drastischer Abfall der Datenrate wie durch Slow Start vermieden; man setzt nach einer potentiellen Überlastsituation wieder mit einer höheren Datenrate (durch den Threshold vorgegeben) auf und kann auch während der Fehlerbehandlungsperiode weitersenden.

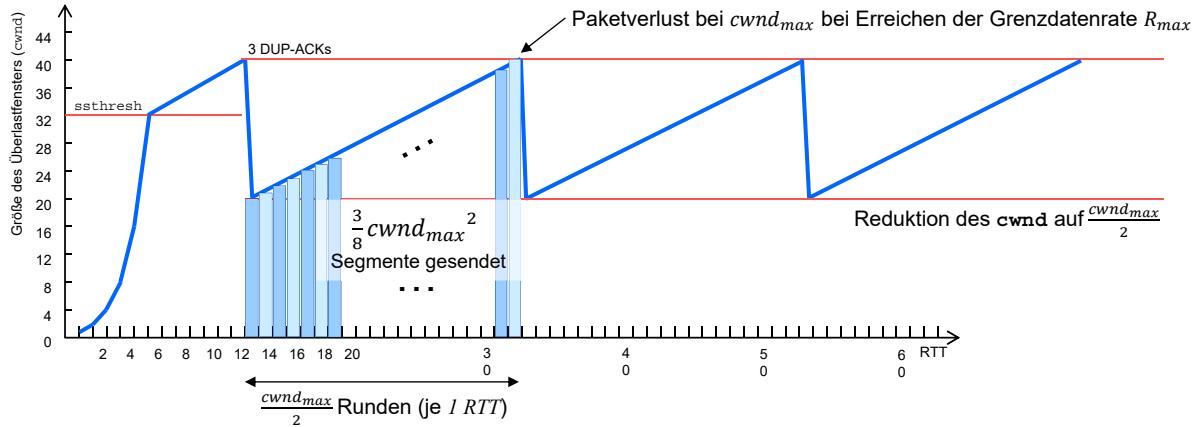
Insgesamt ist nun das AIMD-Prinzip besser umgesetzt: die Steigerung der Datenrate erfolgt linear (nach anfänglicher Slow-Start-Phase), die Verringerung multiplikativ (Halbierung der Datenrate bei Segmentverlust).

(Bitte beachten: das Vorgehen bei Fast Retransmit / Fast Recovery lässt sich schlechter in das Prinzip der Übertragungsrunden einpassen als Slowstart. Auf Folie 55 ist eine detailliertere Darstellung dessen, was passiert.)

Eine Frage ist noch, wie der initiale Threshold gewählt werden sollte. Die RFCs sagen dazu, dass er möglichst groß angesetzt werden solle. Z.B. könne er am Fenster des Empfängers orientiert werden (Window Size im TCP-Header, auch rwnd genannt), um wenn möglich schnell die durch den Empfänger erlaubte Kapazität erreichen zu können (Initial sshthresh = advertised window).

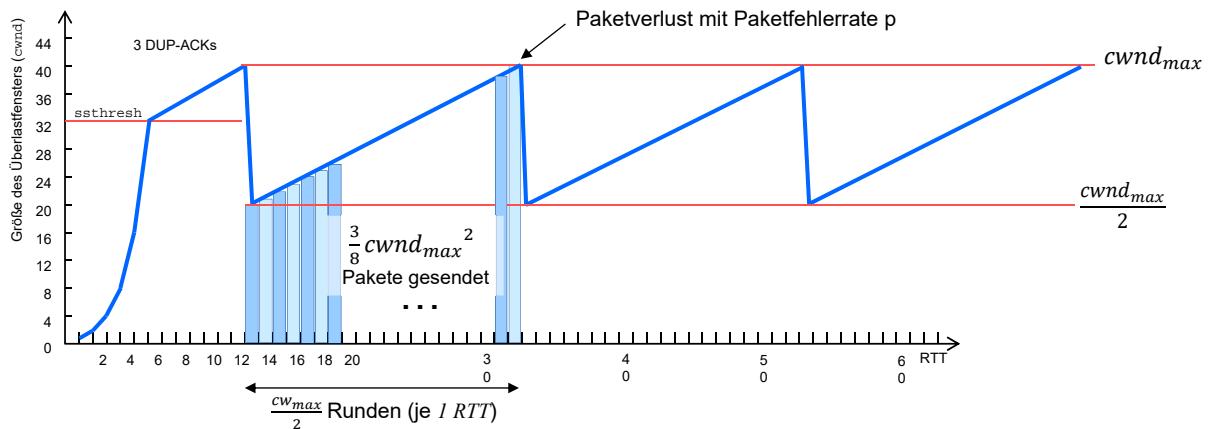
Bestimmte Betriebssysteme verwenden eventuell unterschiedliche Werte. Laut Wikipedia war in Microsoft Windows 9x, Me und NT das Empfangsfenster standardmäßig 8 Kilobyte groß. Windows 2000 und XP reservierten 16 Kilobyte. Laut der englischen Version des Artikels hatten Windows Vista und Windows 7 ein Empfangsfenster von 64 kB, hochskalierbar bis 16 MB (Autotuning).

# TCP Goodput (Throughput made good) bei garantierter Datenrate



- Annahme: Konstanter Durchsatz  $R_{max}$  ist für die TCP-Verbindung garantiert
  - Welche Datenrate erreicht die TCP-Verbindung im Durchschnitt?
    - Datenrate entspricht der durchschnittlichen Rate pro Fast-Recovery-Abschnitt (1 Sägezahn)
    - Gesendete Pakete pro „Sägezahn“: #Pakete =  $\frac{cwnd_{max}}{2} \cdot \left( \frac{\frac{cwnd_{max}}{2} + cwnd_{max}}{2} \right) / 2 = \frac{3}{8} cwnd_{max}^2$
    - Durchsatz pro „Sägezahn“:  $R_{avg} = \frac{\#Pakete \cdot MSS}{\#Runden \cdot RTT} = \frac{\frac{3}{8} cwnd_{max}^2 \cdot MSS}{\frac{cwnd_{max}}{2} \cdot RTT} = \frac{3 \cdot cwnd_{max} \cdot MSS}{4 \cdot RTT} = \frac{3}{4} R_{max}$

## TCP Goodput bei Paketverlustrate $p$



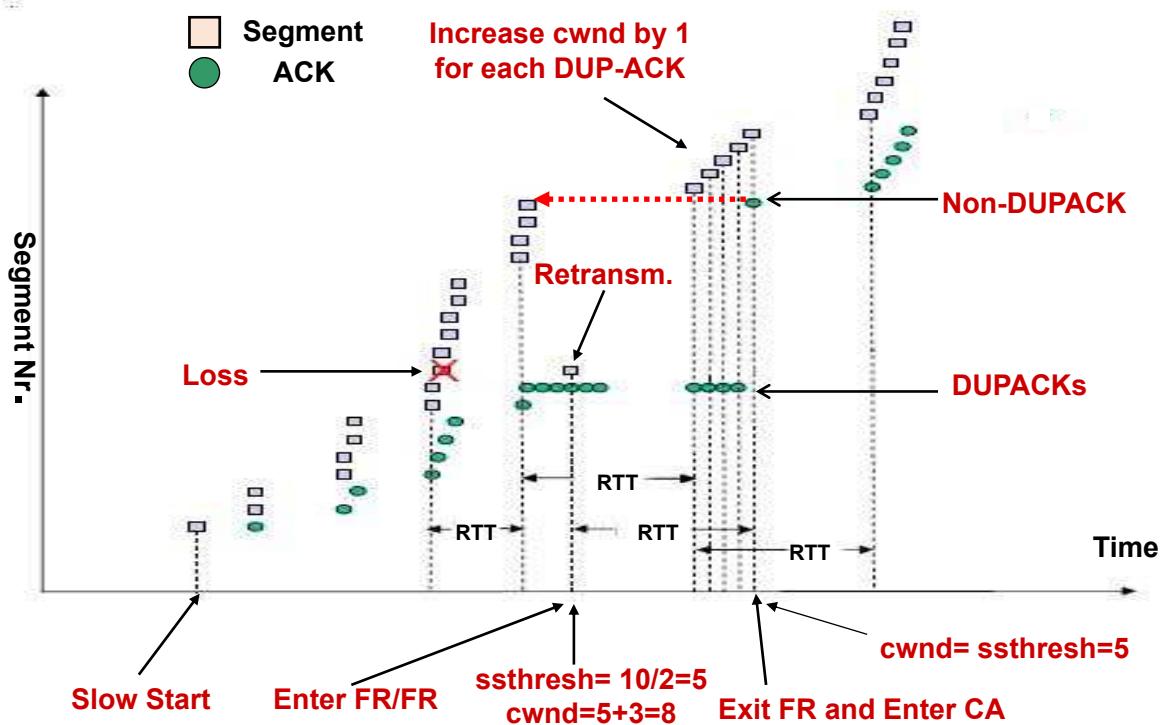
- Neue Annahme: unbeschränkte Kapazität, aber Paketverlustrate  $p$

► Gesendete Pakete bis Verlust:  $\# \text{Pakete} = \frac{cwnd_{max}}{2} \cdot \left( \frac{cwnd_{max} + cwnd_{max}}{2} \right) / 2 = \frac{3}{8} cwnd_{max}^2$

►  $p = \frac{1}{\# \text{Pakete}} = \frac{1}{\frac{3}{8} cwnd_{max}^2} \Rightarrow cwnd_{max} = \sqrt{\frac{8}{3p}}$

• Durchsatz:  $R_{avg} = \frac{\# \text{packets} \cdot MSS}{\# \text{rounds} \cdot RTT} = \frac{\frac{3}{8} cwnd_{max}^2 \cdot MSS}{\frac{cwnd_{max} \cdot RTT}{2}} = \frac{3}{4} cwnd_{max} \frac{MSS}{RTT} = \frac{3}{4} \sqrt{\frac{8}{3p} \frac{MSS}{RTT}} = \sqrt{\frac{3}{2} \frac{MSS}{RTT \cdot \sqrt{p}}}$

## Beispiel FR/FR: Paketverlust



Die hier gezeigte Art der Darstellung wird in der TCP-Welt meist gewählt. Da sie allerdings deutlich komplexer ist als die vereinfachte Darstellung unter Zuhilfenahme von Übertragungsrunden, werden wir meist bei der einfacher verständlichen Übertragungsrundendarstellung bleiben.

## Weitere Verbesserungen der Staukontrolle

- Heute: Vielzahl von Erweiterung:

- ▶ **TCP SACK**: Empfänger verschickt Listen von positiven und negativen Quittungen (im Optionsfeld des Headers)
- ▶ Binary Increase Congestion Control (**BIC**) für Hochgeschwindigkeitsnetze mit großer Latenz
- ▶ **CUBIC** als Variante von BIC in Linux (2.6.19 – 3.1): Überlastfenster ist kubische Funktion der Zeit seit der letzten Überlastsituation
  - Fenster wächst zunächst konkav (schnelle Annäherung an Fenstergröße vor der letzten Überlastsituation), danach konvex (vorsichtiges Testen auf höhere Datenraten, schnelle Steigerung)
  - Entkopplung vom Empfang von Quittungen
  - Abgelöst durch TCP *Proportional Rate Reduction* in Linux 3.2
- ▶ **Compound TCP** als TCP Reno mit Delay-Window
- ▶ ...

Die ursprünglichen Algorithmen sind bei vielen Betriebssystemen weiter verbessert worden, beispielsweise durch Selective Acknowledgements. Linux und Windows als prominente Kernel implementieren ihre eigenen Varianten.

CUBIC: die Fenstergröße ist eine kubische Funktion der Zeit seitdem das letzte Mal eine Stausituation aufgetreten war; der Empfang von ACKs ist nicht nötig, um das Fenster zu vergrößern. Dadurch wird zum einen eine zu schnelle Reaktion auf fehlende ACKs vermieden, zum anderen ist eine bessere Fairness zwischen unterschiedlichen TCP-Strömen gegeben, da bei traditionellen TCP-Algorithmen das Fenster schneller wächst, wenn die RTT zum Empfänger sehr gering ist. Dadurch kann es passieren, dass TCP-Ströme zwischen relativ dicht beieinander befindlichen Stationen gegenüber denen, die weite Strecken zurücklegen, bevorzugt werden und eine höhere mittlere Datenrate bekommen. Mittlerweile abgelöst durch TCP Proportional Rate Reduction, welches die Fenstergröße nach einem Paketverlust nahe am Threshold hält.

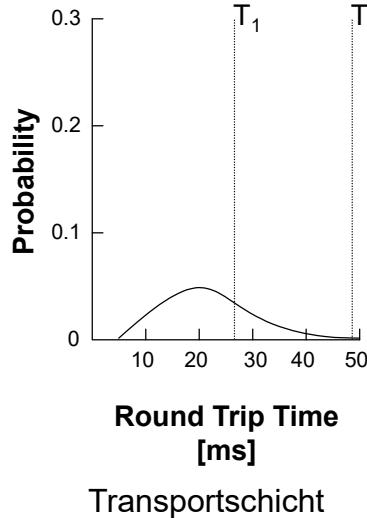
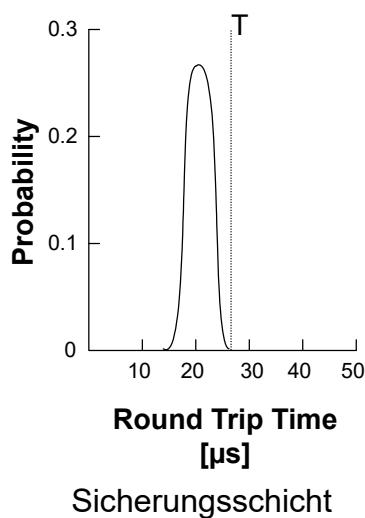
Compound TCP: optimiert für Verbindungen mit hohem Bandbreiten-Verzögerungs-Produkt. Es basiert auf TCP Reno, ergänzt das Fenster allerdings um eine Delay-basierte Komponente. Dieser Teil des Fenster wächst bei kleiner Verzögerung zwischen den Kommunikationspartner sehr schnell an, um die zur Verfügung stehende Kapazität des Netzes schnell auszunutzen; später wird das Fenster verkleinert, um die Datenrate insgesamt relativ konstant und in der Nähe des Bandbreiten-Verzögerungs-Produktes zu halten.

Einen (unvollständigen) Überblick über TCP-Varianten bietet  
[http://en.wikipedia.org/wiki/TCP\\_congestion-avoidance\\_algorithm](http://en.wikipedia.org/wiki/TCP_congestion-avoidance_algorithm).

## Retransmission Timer

- Wie sollte der Retransmission Timer gewählt werden?

- ▶  $T_1$ : zu klein, zu viele Neuübertragungen
- ▶  $T_2$ : zu groß, zu lange Wartezeiten bei tatsächlichem Verlust



Ein Problem bei der praktischen Umsetzung der Algorithmen ist: wann sollte ein Timeout ausgelöst werden, wie groß sollte der Retransmission Timer gewählt werden?

Während man auf der Sicherungsschicht immer nur einen einzelnen Link betrachtet und daher gute Schätzungen abgeben kann, wann eine Quittung eintreffen sollte, betrachtet man auf der Transportschicht ein Netzwerk aus Routern, die indeterministische Verzögerungen erzeugen können. Die Laufzeit eines Segments durchs Netz kann nur geschätzt werden und sich auch von Segment zu Segment ändern. Der Retransmission Timer kann daher nicht statisch festgelegt werden.

## Berechnung des Retransmission Timers

- **Algorithmus zur Berechnung (Jacobson, 1988)**

- ▶ Stoppe Round-Trip-Time  $t$  für jedes Segment
- ▶ Führe Variable  $RTT$  als Schätzung der aktuellen Round-Trip-Time
- ▶ Initialisiere  $RTT$  mit einem Startwert
  - Z.B. 2 Sekunden in Linux, 3 Sekunden in Windows
- ▶ Falls eine Quittung vor Ablauf des Retransmission Timers eintrifft, aktualisiere  $RTT$ :
  - $RTT = \alpha \cdot RTT + (1 - \alpha) \cdot t_{\text{letzte } RTT}$
  - $\alpha$  ist Glättungsfaktor, oft 0.875
- ▶ Wahl der Retransmission Timers  $RTO$ 
  - Erster Ansatz: wähle Retransmission Timer als  $\beta \cdot RTT$
  - Wie sollte  $\beta$  gewählt werden?

## Berechnung des Retransmission Timers

- **Beispielalgorithmus (Jacobson, 1988)**

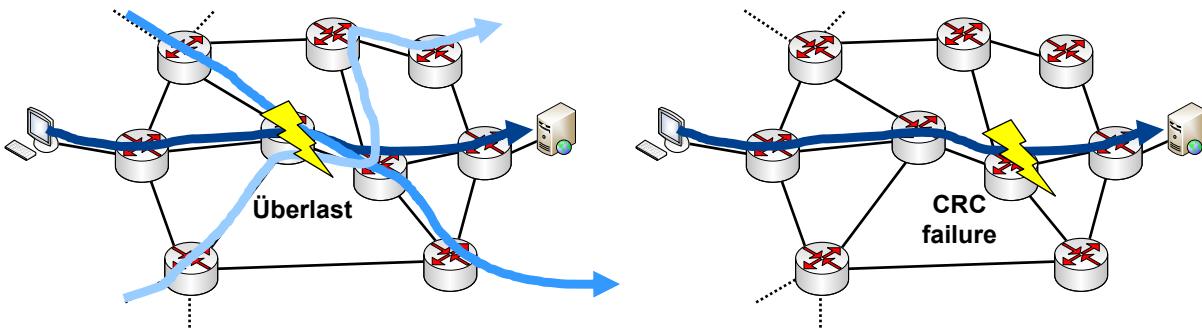
- ▶ Wahl von  $\beta$ : orientiert an Standardabweichung der Round-Trip-Zeiten
  - $\beta = \alpha' \cdot \beta + (1 - \alpha') \cdot |RTT - t_{\text{letzte RTT}}|$
- ▶ Dann:  $RTO = RTT + 4 \cdot \beta$ 
  - „4“ wurde mehr oder weniger willkürlich gewählt (effiziente Implementierung der Multiplikation)
  - Ist  $RTO$  kleiner 1 Sekunde, wird auf 1 Sekunde aufgerundet

- **Zusätzlich:**

- ▶ Mit jedem Timeout nach einer Neuübertragung verdopple den Wert des Retransmission Timers
  - Wiederholter Verlust scheint auf dauerhafte Überlastsituation hinzudeuten
- ▶ Kann bis auf mehrere Minuten steigen!

## Neuere Entwicklung: Explicit Congestion Notification

- Router verwirft bei Überlast Pakete
  - ▶ Sender erhält DUP-ACKs oder hat Timeout
- Sender kann nicht entscheiden:
  - ▶ Liegt eine Überlastung eines Routers vor?
  - ▶ Ist ein Paket aufgrund eines Bitfehlers verworfen worden?

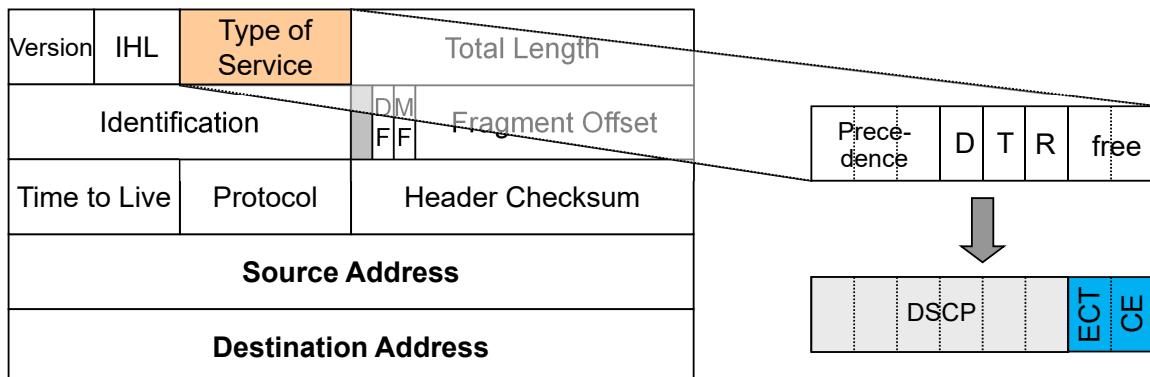


- ▶ Wissen nur auf Schicht 3 vorhanden!

# Neuere Entwicklung: Explicit Congestion Notification

## • Überarbeitung des IP-Headers:

- ▶ Neudefinition des „Type-of-Service“-Felds
  - DSCP – Differentiated Service Code Point (QoS-Unterstützung)
  - *ECT – Explicit Congestion Notification (ECN) Capable Transport*
  - *CE – Congestion Experienced*



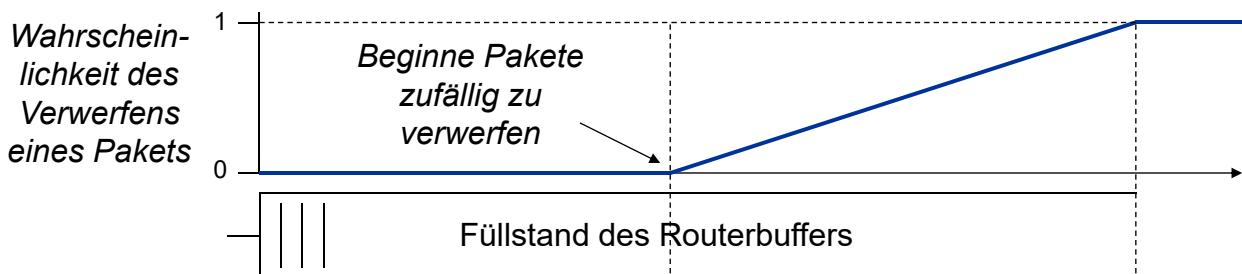
Bereits vor mehreren Jahren wurde das ToS-Feld des IP-Headers neu definiert. Da Routerhersteller die Informationen sowieso nicht berücksichtigten, wurden die sechs standardisierten Bits umdefiniert zu einem „Differentiated Service Code Point“. Dieser wird im Rahmen einer QoS-Erweiterung von IP benötigt, die hier nicht behandelt werden soll.

Die beiden letzten Bits des Feldes wurden zunächst unberührt gelassen: sie waren für spätere Verwendung freigehalten worden und im Laufe der Zeit von manchen Anwendungen für proprietäre Signalisierungszwecke missbraucht worden. Mittlerweile sind aber auch diese Bits neu definiert, um mit der Staukontrolle von TCP zusammenzuarbeiten.

## Interaktion von TCP und IP

- IP kann Überlastsituationen erkennen:

- ▶ ECT-Flag wird vom Sender gesetzt, wenn er die Stauerkennungs-erweiterung implementiert
- ▶ Router können das CE-Flag setzen, wenn eine Überlastsituation vorliegt
  - *Random Early Detection* (RED)



- ▶ Der Empfänger erkennt eine Überlastsituation anhand des CE-Flags

In einer einfachen Router-Implementierung werden empfangene IP-Pakete in einen Buffer eingereiht und der Reihe nach entnommen, auf ihre Zieladresse hin untersucht und weitergeleitet. Ist der Buffer voll, werden weitere Pakete verworfen.

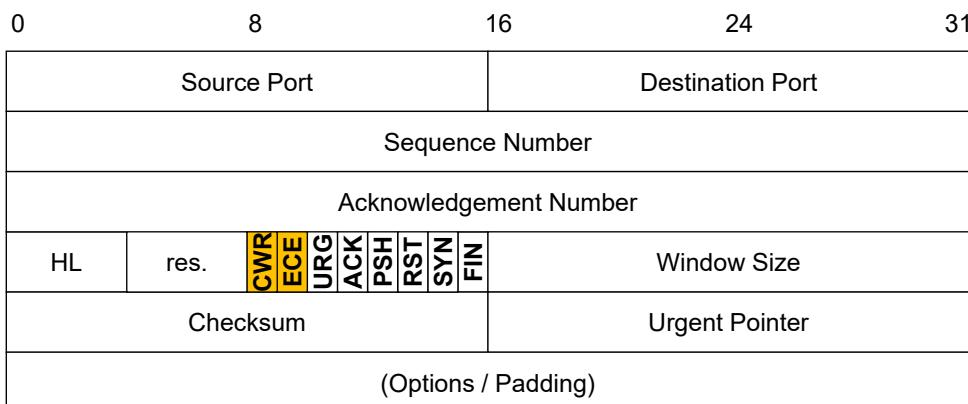
Heutige Router implementieren allerdings „Random Early Detection“. Um Überlast zu vermeiden, beginnt ein Router bereits ab einem definierten Füllstand seines Buffers, zufällig Pakete zu verwirfen. Die Verwurfswahrscheinlichkeit steigt dabei mit zunehmendem Bufferfüllstand. Die Idee dahinter ist, mit einem Verwerfen von Paketen nicht zu warten, bis Überlast eingetreten ist, sondern Pakete schon vorher zu verwirfen, damit der Sender die Datenrate reduziert. Dadurch pendeln sich TCP-Verbindungen bei einer Datenrate ein, die das Netz weitgehend auslastet, aber nicht überlastet.

Kritisieren kann man nun allerdings, dass selbst ohne akute Überlast Pakete verworfen werden. Dies soll die ECN-Erweiterung vermeiden: bei Verwendung von ECN kann ein Router das CE-Flag im IP-Header setzen, statt Pakete zu verwirfen. Dadurch erfährt die empfangende IP-Instanz, dass im Netz eine Überlast droht. Diese Information muss nun an die sendende TCP-Instanz weitergegeben werden.

## Modifikation von TCP

- **TCP und ECN: Verwendung zweier neuer Flags**

- ▶ **ECN Echo (ECE)**: Empfänger setzt dieses Bit in den Quittungen für Pakete mit gesetztem CE-Flag
- ▶ Sender passt **cwnd** an und signalisiert dies dem Empfänger über das **CWR-Flag (Congestion Window Reduced)**



Dies erfolgt über den TCP-Header, durch Standardisierung zweier weiterer Flags. In der Quittung auf das Segment, welches ohne ECN aufgrund von RED verworfen worden wäre, setzt der Empfänger das ECE-Flag. Der Sender verhält sich daraufhin so, als wäre ein Paketverlust aufgetreten (Verringerung von cwnd nach FR/FR), ohne dass ein Timeout droht.

# Steigerung der TCP-Performance

- Auch außerhalb der Staukontrolle Verbesserungen

- ▶ *Fast Open*

- Beschleunigung des Datenaustauschs
    - Daten bereits in Handshake-Nachrichten
    - Im Linux-Kernel ab Version 3.13 Standard

- ▶ *Multipath-TCP*

- Gleichzeitige Nutzung mehrerer Pfade
      - Erhöht Redundanz und Durchsatz
      - Z.B. Server mit Multihoming mehr Stabilität bei Ausfall eines Pfades
    - Mobilitätsunterstützung
      - Nutzung von mehreren Netzwerktechnologien gleichzeitig
      - Wechsel z.B. zwischen WiFi und Mobilfunk wird vereinfacht

- ▶ ... mehr dazu z.B. in “Communication Systems Engineering”

## Kapitel 5: Transportschicht

- **Protokollmechanismen der Transportschicht**

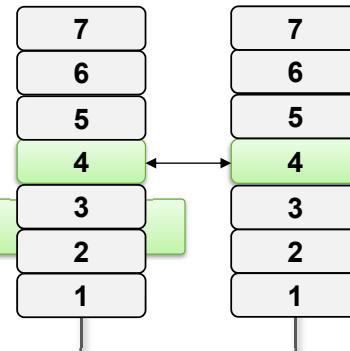
- ▶ Prozessadressierung, strombasierte vs. paketbasierte Kommunikation, verbindungsorientiert/verbindungslos

- **Die Transportschicht im Internet**

- ▶ TCP (Transmission Control Protocol)

- Adressierung, Sockets
- TCP-Verbindung
- Flusskontrolle
- TCP-Header
- Staukontrolle (Congestion Control)

- ▶ UDP (User Datagram Protocol)



## UDP: „Keep it simple“

### • UDP: verbindungslose Kommunikation

- ▶ *Unzuverlässige* Datenübertragung
  - Paketverlust, Reihenfolgevertauschung und Duplikate werden nicht erkannt und nicht behandelt (keine Quittungen)
  - Fehlerhaft empfangene Pakete können zwar durch eine (optionale) Checksumme erkannt werden, werden aber einfach verworfen
- ▶ Geringe Zuverlässigkeit, aber *kein Overhead durch Verbindungsverwaltung!*
  - Dadurch schnellere Übertragung kleiner Datenmengen
  - Möglich: Quittungen/Neuübertragungen in Anwendungen implementiert
- ▶ Hauptzweck von UDP: Angabe von Sender- und Empfänger-Port auf dieser Ebene notwendig
  - Und: Nutzung für Multicast (nicht möglich bei TCP)
- ▶ Geringer Funktionsumfang: nur 8 Byte Header

### UDP - User Datagramm Protocol

UDP stellt einen einfachen, unbestätigten Datagrammdienst auf der Transportschicht zur Verfügung. UDP versendet Datagramme bis zu einer Größe von 64 kByte. Der Empfang wird nicht bestätigt, und es muss beachtet werden, dass Datagramme verloren gehen können.

Die einzigen Protokollfunktionen, die UDP ausführt, sind lokale Fehlerbehandlung (fehlerhafte PDUs werden verworfen) und Multiplexen (wie bei TCP mit Ports).

UDP wird häufig benutzt, um kleine Informationsmengen, die in keinem unmittelbaren Zusammenhang stehen, auszutauschen. Man möchte dabei auf den Verbindungsoverhead von TCP verzichten. UDP wird oft als „schnell“ bezeichnet, da es weder durch den Verbindungsoverhead noch durch Staukontrolle ausgebremst wird und man theoretisch mit höchsten Datenraten übertragen kann, ohne ausgebremst zu werden.

Außerdem kann es von Nutzen sein, dass man keine Fehlerbehebungsmechanismen ausführt, beispielsweise bei einer Videoübertragung. Wenn man 0 als Prüfsumme einträgt, so wird die Fehlerkontrolle ganz ausgeschaltet und es können auch fehlerhafte PDUs übergeben werden.

Verglichen mit dem TCP-Segment ist das UDP-Datagramm daher sehr einfach aufgebaut.

Die Adressierung der Verbindungsendpunkte verläuft analog zu TCP, wobei auch für UDP sog. Well-known ports vereinbart wurden, z.B.:

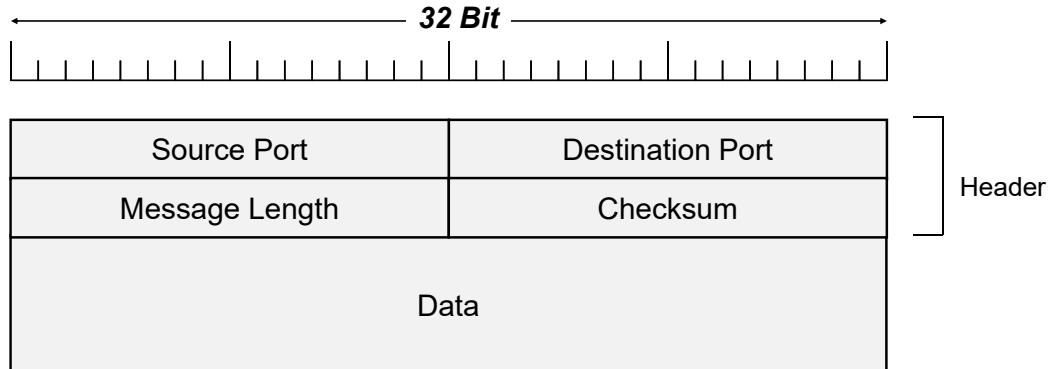
- 13: Daytime
- 53: Domain Name System
- 123: Network Time Protocol

Der Header von UDP umfasst nur zwei 32-Bit-Worte (Sender- und Empfängerport, ein Prüfsummenfeld (Mechanismus wie bei TCP) und ein 16-Bit-Längenfeld.

## UDP: User Datagram Protocol

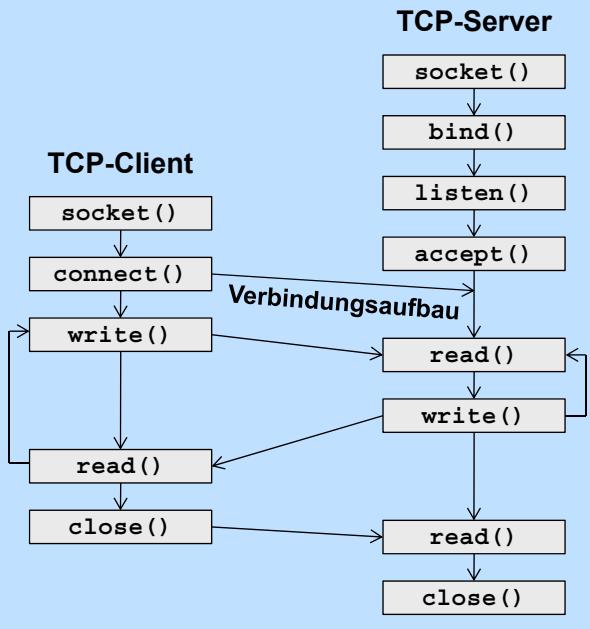
- Kompaktes Header-Format

- ▶ Im Wesentlichen Portnummern
- ▶ Checksumme ist optional

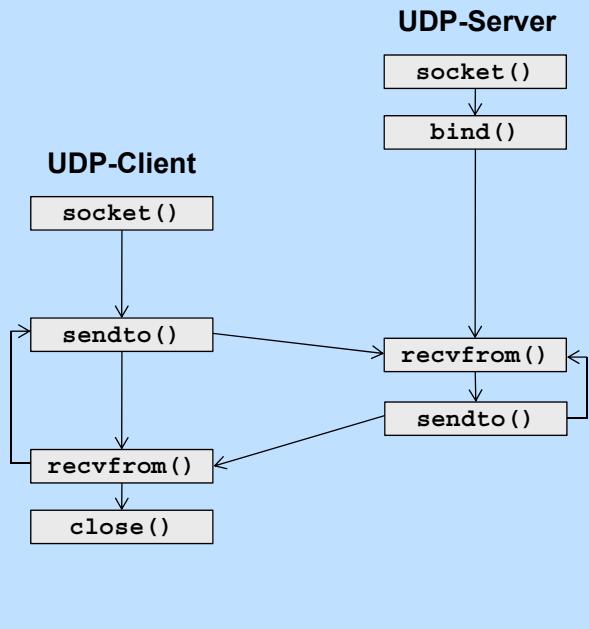


# Ablauf bei der Kommunikation über Sockets

- TCP-basierte Kommunikation



- UDP-basierte Kommunikation



UDP-Sockets bieten einen geringeren Umfang an Primitiven als TCP-Sockets, da sie keinen Verbindungsau- und –abbau verwenden.

## Beispiel: UDP Echo Server

```
int main()
{
10    int sockfd;
    struct sockaddr_in servaddr, cliaddr;

    sockfd=socket(AF_INET,SOCK_DGRAM,0);
    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family=AF_INET;
15    servaddr.sin_addr.s_addr=htonl(INADDR_ANY);
    servaddr.sin_port=htons(8012);
    bind(sockfd, (struct sockaddr*) &servaddr, sizeof(servaddr));
    for (;;) { /* infinite loop */
        int n, socklen_t len;
20        char data[256];
        n=recvfrom(sockfd, data, 256, 0,(struct
            sockaddr*)&cliaddr,&len);
        sendto(sockfd, data, n, 0, (struct sockaddr*)
            &cliaddr, len);
    }
    return 0;
25 }
```

### Description:

**0-7:** the usual #include instructions

**10,11:** required variables

**12:** socket( ) call IPv4 (AF\_INET) and UDP (SOCK\_DGRAM)

**13-16:** initialization of the sockaddr structure with the server address and port

**17:** bind( )only required if fixed port

**18-23:** infinite loop

**21:** receive a packet (max. 256 Byte).

The received address information is stored in cliaddr (max. length len).

**22:** send back the received data exactly to the address and the port from which the data was sent

# Client

```
int main(int argc, char **argv)
{
10    int sockfd; struct sockaddr_in servaddr;
    if(argc!=3){
        printf("usage: %s <adresse> <port>\n", argv[0]);
        exit(0);
    }
    memset(&servaddr, 0, sizeof(servaddr));
15    servaddr.sin_family=AF_INET;
    servaddr.sin_port=htons(atoi(argv[2]));
    inet_pton(AF_INET, argv[1], &servaddr.sin_addr);
    sockfd=socket(AF_INET, SOCK_DGRAM, 0); /* IPv4 & UDP */
    for (;;) {
20        /* an int and some buffer space */
        int n; char send_data[1024], recv_data[1024];
        /* read a line from stdin */
        if( fgets(send_data, 1024, stdin) == NULL ) break;
        sendto(sockfd, send_data, strlen(send_data), 0,
               (struct sockaddr*)&servaddr, sizeof(servaddr));
25        n=recvfrom(sockfd, recv_data, 1024, 0, NULL, NULL);
        recv_data[n]=0; /* make sure the string ends !! */
        /* write the received string to stdout */
        fputs(recv_data,stdout);
    };
30    close(sockfd);
    return 0;
}
```

## Description:

**0-7:** the usual `#include` instructions

**10:** required variables

**13:** examining the command line parameters

**14-17:** initialization of the `sockaddr` structure with the destination address and port

**18:** Initialize `socket()`: IPv4 (`AF_INET`) and UDP (`SOCK_DGRAM`)

**19-29:** infinite loop

**21:** some variables for the loop block

**23:** read from `stdin`, with abort condition

**24-25:** send input and receive the result

**27,28:** add the string end and write the received string to `stdout`

## Zusammenfassung

- **Zwei prominente Protokolle zur Kommunikation zwischen Prozessen:**
  - ▶ TCP: zuverlässige Datenübertragung durch Quittungsmechanismus und Flusskontrolle (virtuelle Verbindungsorientierung)
  - ▶ UDP: schnelle Übertragung, da es keine Kontrollmechanismen hat; füge einem IP-Paket lediglich Ports hinzu
- **Im Laufe der Jahre: Entwicklung weiterer Transportprotokolle**
  - ▶ SCTP (Stream Control Transmission Protocol)
  - ▶ DCCP (Datagram Congestion Control Protocol)
  - ▶ QUIC (Quick UDP Internet Connections)
  - ▶ ...

TCP wird laufend weiterentwickelt – nicht nur die Verfahren zur Congestion Control werden ständig erweitert, auch in anderer Hinsicht erfolgen Erweiterungen. So gibt es mittlerweile z.B. eine Spezifikation zu „TCP Fast Open“, bei dem bereits im ersten SYN-Segment Daten mitgeschickt werden können, um den Start der Datenübertragungsphase zu beschleunigen. Eine andere wesentliche Neuerung ist Multipath-TCP, bei dem simultan Daten über mehrere IP-Adressen/Interfaces versendet werden können, die alle hinter einer TCP-Verbindung verborgen sind.

## Zusatz-Vorlesung zu Transport-Evolution (Video, nicht klausurrelevant)

- Ursachen und Probleme von TCP

- ▶ Evolution des Internets
  - Alter von TCP, Wandlung der Internetnutzung, Technologie-Entwicklung
- ▶ Beispiel: Parallelismus und TCP (Connection Racing)

- Evolution von TCP

- ▶ Erweiterbarkeit von TCP und Limitierungen
  - WS, SACK, TS, ECN, IW10 und Pacing, TFO, MPTCP, ...
- ▶ Middleboxes

- QUIC

- ▶ Sinn und Ziele
- ▶ Aufbau
- ▶ Fundamentale Unterschiede zu TCP

- Congestion Control

## Lessons Learned

- **Transportschicht dient zur Adressierung von Prozessen**
  - ▶ Verwendung von Ports
- **Unterschiedliche Anwendungscharakteristika**
  - ▶ Strombasierte Kommunikation (TCP)
    - Zuverlässigkeit durch ARQ
    - Flusskontrolle durch Sliding Window mit dynamischer Fenstergröße
    - Staukontrolle zur Vermeidung der Netzüberlastung
      - Wird immer noch weiterentwickelt
  - ▶ Paketbasierte Kommunikation (UDP)
    - Keine Kontrollmechanismen, daher gleiche Probleme wie IP
    - Aber kein Overhead durch Verbindungsaufbau und –verwaltung
      - Geringere Verzögerung (Latenz) beim Senden kleinerer Datenmengen
  - ▶ Aktuellere Entwicklung: QUIC – adressiert Zuverlässigkeit und Latenz