

Verteilte Systeme

Teil 2: Netzwerkkommunikation (Sockets)

Inhalte der Vorlesung

- ▶ Einführung
- ▶ Netzwerkkommunikation
- ▶ Middleware
- ▶ Algorithmen und Verfahren
 - ▶ Synchronisation
 - ▶ Skalierbarkeit
 - ▶ Konsistenz / Replikation
 - ▶ Fehlerbehandlung
- ▶ Dienstbeispiele
 - ▶ Verteilte Dateisysteme
- ▶ Sicherheit

Inhalte der Vorlesung

- ▶ Einführung
- ▶ **Netzwerkcommunication**
- ▶ Middleware
- ▶ Algorithmen und Verfahren
 - ▶ Synchronisation
 - ▶ Skalierbarkeit
 - ▶ Konsistenz / Replikation
 - ▶ Fehlerbehandlung
- ▶ Dienstbeispiele
 - ▶ Verteilte Dateisysteme
- ▶ Sicherheit

- ▶ Kommunikationsformen
 - ▶ Verbindungslos/-orientiert
 - ▶ Synchron, asynchron
- ▶ Ein paar Netzwerkgrundlagen
- ▶ Programmierung: Beispiele
 - ▶ Berkeley Socket API
 - ▶ Java Sockets

Arten der Kommunikation

- ▶ Gibt es eine Verbindung?
 - ▶ Verbindungsorientiert
 - ▶ Verbindungslos
- ▶ Muss auf eine Antwort gewartet werden?
 - ▶ Ja: synchrone Kommunikation
 - ▶ Nein: asynchrone Kommunikation

Arten der Kommunikation

1. Verbindungsorientiert

- ▶ Verbindung zwischen Sender und Empfänger wird explizit aufgebaut (*handshake*) und abgebaut
- ▶ Schnittstelle für den Austausch von Nachrichten stellt sich der Anwendung als ein Datenstrom mit gesicherter Übertragung (keine Verluste von Nachrichten) dar
- ▶ Entwickler sind nicht mit Problemen wie Reihenfolgetreue oder Paketverlusten befasst (Hinweis: das stimmt nicht ganz...warum nicht?)
- ▶ Kein Broad- / Multicast

▶ Beispiel?

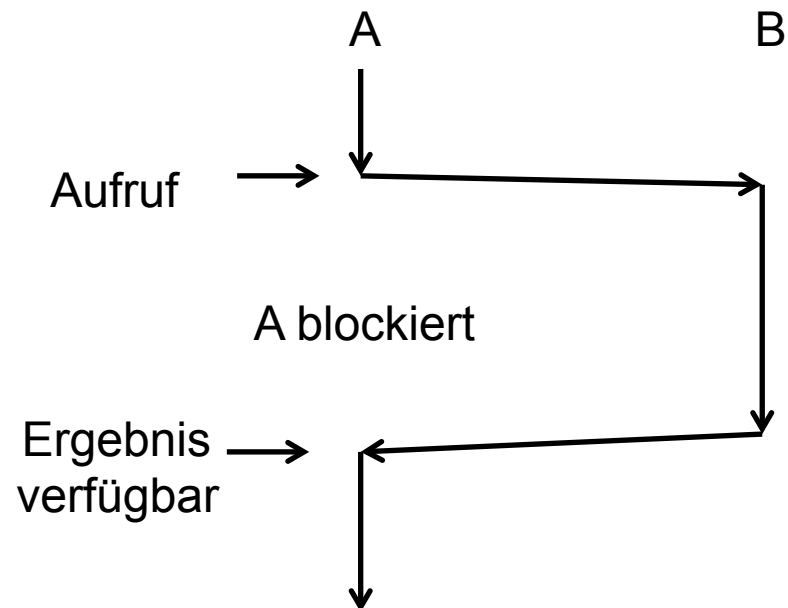
2. Verbindungslos

- ▶ keine Verbindung
- ▶ Nachrichten werden einzeln durch das Netzwerk geschickt
- ▶ Verluste von Paketen möglich
- ▶ Reihenfolge von Paketen muss in der Anwendung geprüft werden

▶ Beispiel?

Synchrone Kommunikation

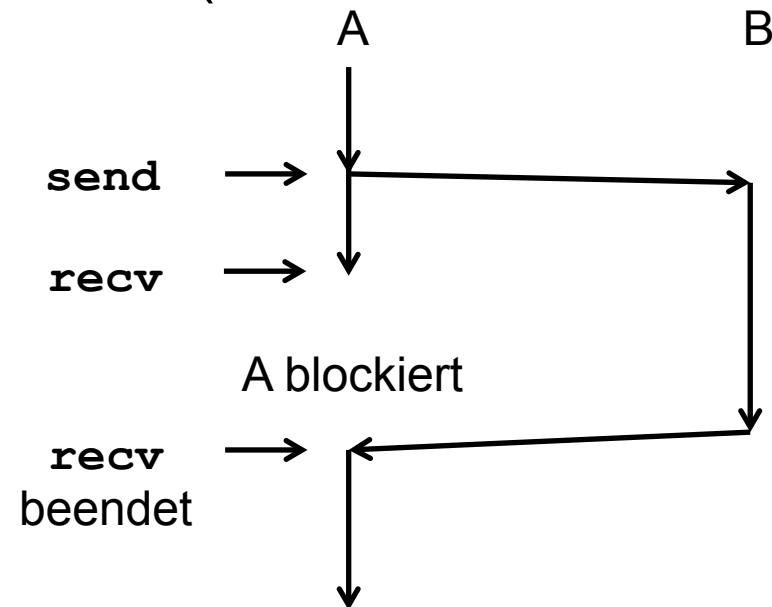
- ▶ Sender wird blockiert, bis Nachricht vom Empfänger bestätigt bzw. beantwortet wurde
- ▶ Enge Kopplung zwischen Sender und Empfänger



Synchrone Kommunikation

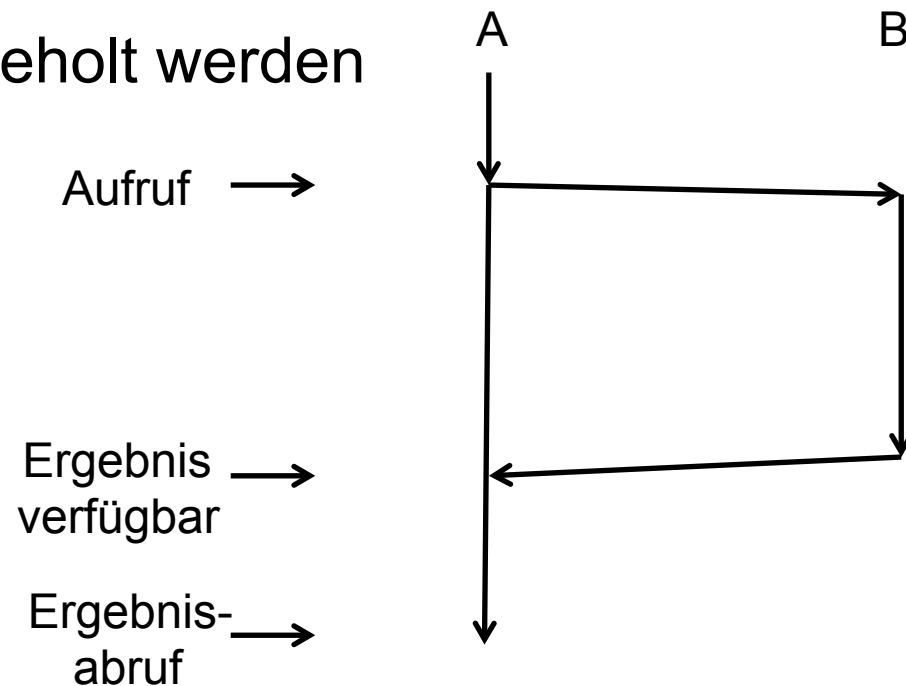
► Sockets

- Standard: synchron
- Empfangsoperationen (z.B. **recv** bzw. **recvfrom**) blockieren bis etwas empfangen wird (oder ein Fehler auftritt)
- Sendeoperationen können auch blockieren, wenn nichts gesendet werden kann (z.B. bei TCP Problemen)



Asynchrone Kommunikation

- ▶ Sender wird nicht blockiert
- ▶ Empfänger muss Empfangsbestätigung (z.B. das Ergebnis einer Anfrage) explizit schicken
 - ▶ Es geht auch ohne Empfangsbestätigung
 - ▶ Ergebnis kann gepuffert werden
 - ▶ Ergebnis muss nicht abgeholt werden
- ▶ Führt zu loserer Koppelung von Komponenten im verteilten System



Kommunikationsformen

- ▶ Frage: Welche Kommunikationsform (verbindungslos/-orientiert, synchron/asynchron) ist am besten geeignet in einem verteilten System?
 - ▶ Hat das einen Einfluss auf die Qualität des Systems ?
- ▶ Beispiel: Fehler
 - ▶ Synchron kann problematisch im Fehlerfall sein
 - ▶ Nachrichtenverlust
 - ▶ Ausfall des Empfängers
 - ▶ potentiell unbegrenzte Wartezeit

- ▶ Qualitätsmerkmale von Netzwerken
 - ▶ Latenz: Minimale Verzögerung zwischen Sender und Empfänger ("Zeit um leere Nachricht zu versenden")
 - ▶ Datentransferrate: Anzahl Bits, die maximal zwischen zwei Prozessen übertragen werden können
 - ▶ Nachrichtentransferzeit: Tatsächliche Verzögerung zwischen Sender und Empfänger für eine Nachricht
 $\text{Latenzzeit} + \text{Nachrichtenlänge} / \text{Datentransferrate}$
 - ▶ Durchsatz: Menge übertragener Bits pro Zeiteinheit
 - ▶ Bandbreite: Maximal mögliche physikalische Übertragungsrate (Grenze des Netzwerks)

Qualitätsparameter

- ▶ Frage: Warum ist Latenz nicht gleich Nachrichtentransferzeit?
- ▶ Frage: Bei welchen Anwendungen ist Latenz, wo ist Bandbreite wichtiger?
- ▶ Frage: Warum kann die Bandbreite i.d.R. von einer Anwendung nicht voll ausgeschöpft werden?

- ▶ Warum ist das wichtig für verteilte Systeme?
 - ▶ Nachrichtenaustausch zwischen Komponenten benötigt Zeit => wichtig z.B. für Synchronisation
 - ▶ Nachrichtenaustausch hat Grenzen: es ist nicht möglich, beliebig viele Daten in einem gegebenen Zeitintervall zu übertragen
 - ▶ So etwas macht sich häufig erst ab einer gewissen Größe eines verteilten Systems bemerkbar

▶ Fehler

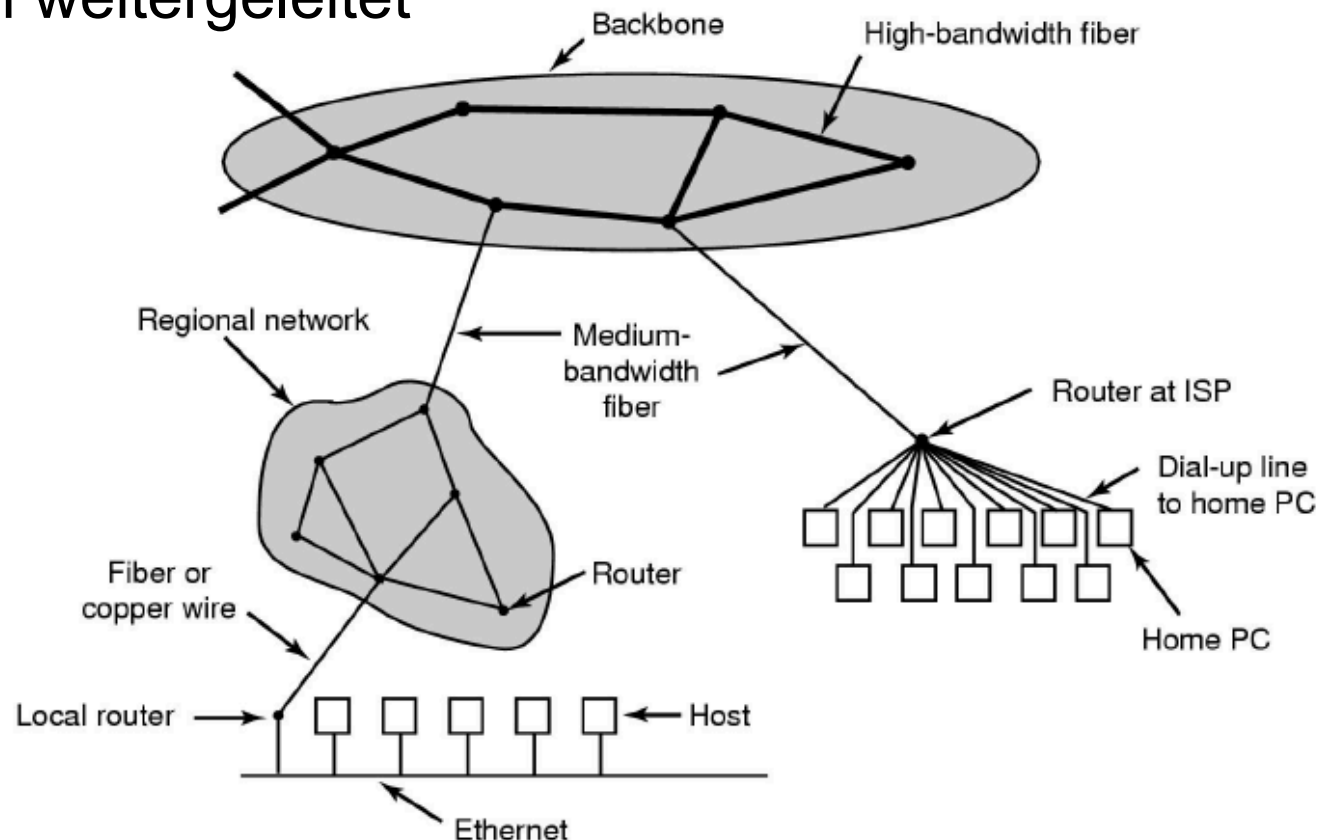
- ▶ Verlust von Nachrichten im Netz möglich
- ▶ Reihenfolge von Nachrichten kann vertauscht werden

▶ Routing

- ▶ I.d.R. gibt es mehrere Wege zwischen Sender und Empfänger
- ▶ Routing kann Übertragungsdauer von Nachrichten beeinflussen
- ▶ Mehrere Empfänger für dieselbe Nachricht möglich (Broadcast: an alle, Multicast: an manche)

Internet: Aufbau

- ▶ Unabhängige Netzwerke (heterogen)
- ▶ Verbunden durch Router
- ▶ Pakete werden durch das Netzwerk verschickt und von Routern weitergeleitet



Internet: Aufbau

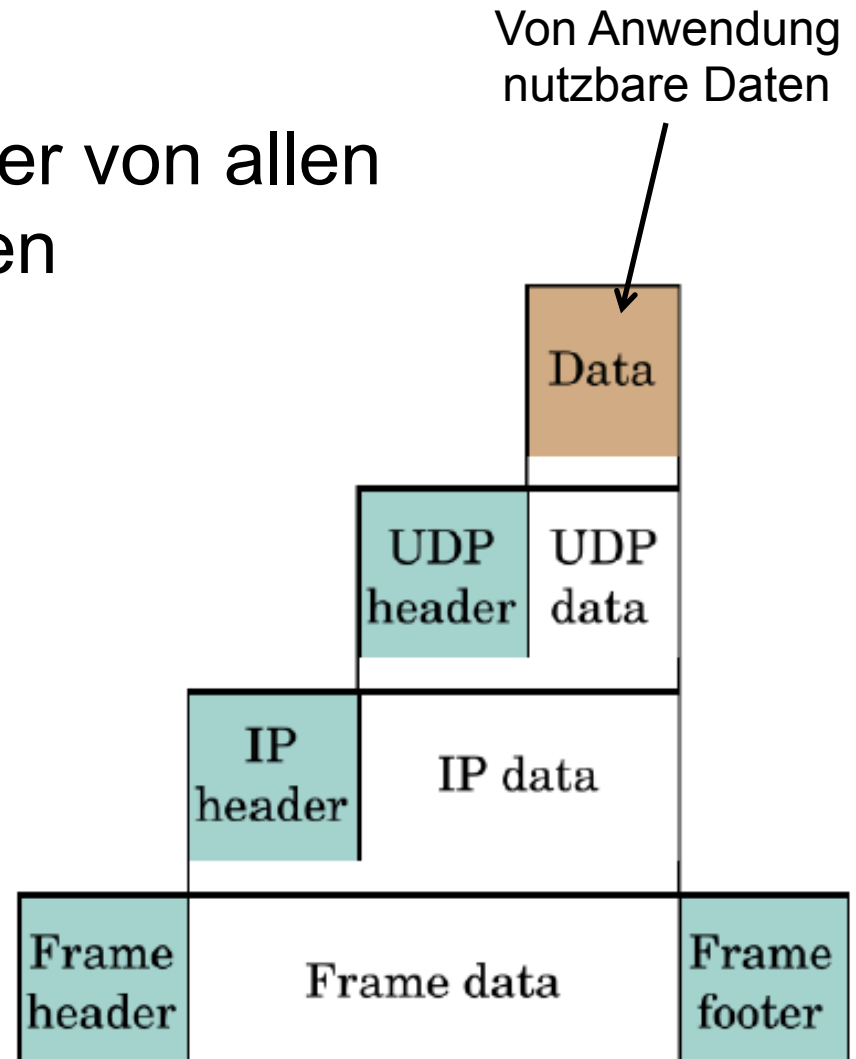
- ▶ Nachrichten im Internet geschichtet
- ▶ Auf jeder Schicht existieren Protokolle, die Nachrichten beschreiben
- ▶ 7 Schichten (OSI Schichtenmodell)

in dieser Vorlesung behandelt

OSI-Schicht	TCP/IP	Verbindung	Beispiel	Einheiten	
7 Anwendung (Application)	Anwendung	Ende zu Ende (Multihop)	HTTP FTP HTTPS SMTP LDAP NCP	Daten	
6 Darstellung (Presentation)					
5 Sitzung (Session)					
4 Transport (Transport)	Transport	Punkt zu Punkt	TCP UDP SCTP SPX	Segmente	
3 Vermittlung (Network)	Vermittlung		ICMP IGMP IP IPX	Pakete	
2 Sicherung (Data Link)	Netzzugang		Ethernet Token Ring FDDI ARCNET	Rahmen (Frames)	
1 Bitübertragung (Physical)				Bits	

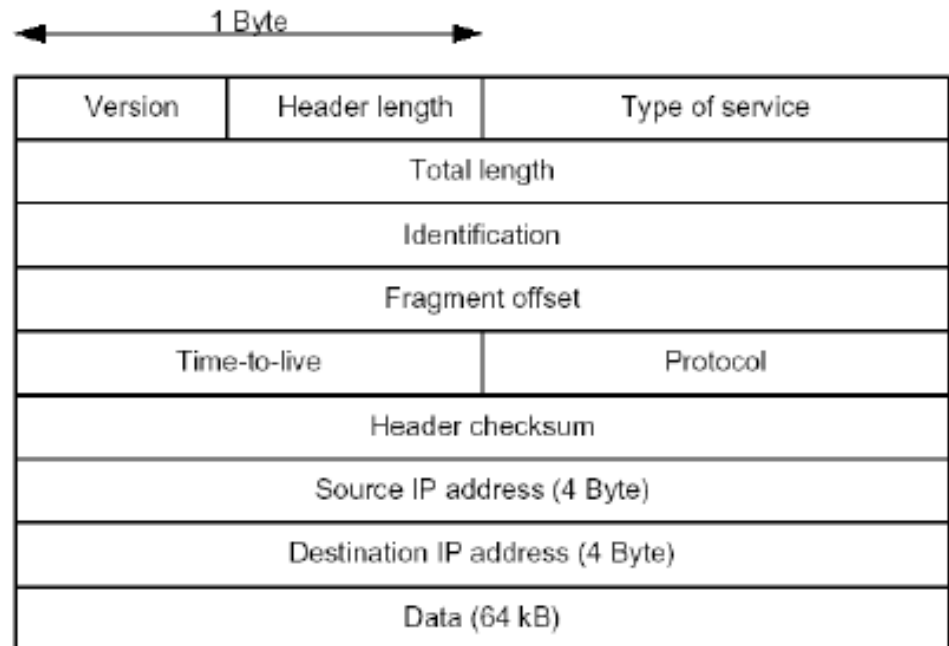
Protokolle

- ▶ Jedes Protokoll hat eine bestimmte Funktionalität
- ▶ Pakete enthalten Header von allen verwendeten Protokollen
- ▶ Hier interessant:
 - ▶ IP Schicht 3
 - ▶ UDP Schicht 4
 - ▶ TCP Schicht 4



Protokolle: IP

- ▶ Überträgt Nachrichten zwischen Rechnern
- ▶ Enthält IP Adresse (verwendet zum Routing)
- ▶ "best-effort": Paketverluste möglich
- ▶ Verbindungslos
- ▶ Duplikate möglich
- ▶ Reihenfolge nicht garantiert



IPv4 Datagrammformat (vgl. IPv6)

Protokolle: UDP

- ▶ Schicht 4: Innerhalb eines IP Pakets
- ▶ Verbindungslos
- ▶ Enthält Port als "Adresszusatz"
- ▶ "best-effort": Paketverluste möglich
- ▶ Duplikate möglich
- ▶ Reihenfolge nicht garantiert
- ▶ Multicast-fähig

2 Byte	2 Byte	2 Byte	2 Byte	variabel
Source port	Destination port	Length	Checksum	Data

UDP Datagrammformat

Protokolle: TCP

- ▶ Schicht 4 (die gleiche wie UDP)
- ▶ Verbindungsorientiert
- ▶ Enthält Port als "Adresszusatz"
- ▶ Automatische Fehlerkorrektur, d.h.
 - ▶ keine Paketverluste, d.h. Pakete werden erneut verschickt
 - ▶ keine Duplikate, d.h. doppelte Pakete werden verworfen
 - ▶ Reihenfolge garantiert
- ▶ Multicast / Broadcast nicht möglich

2 Byte	2 Byte	4 Byte	4 Byte	2 Byte
Source port	Destination port	Sequence number	Acknowledge number	Flags

2 Byte	2 Byte	2 Byte	2 Byte	variabel
Window size	Checksum	Urgent pointer	Options	Data

TCP Paketformat

UDP vs. TCP

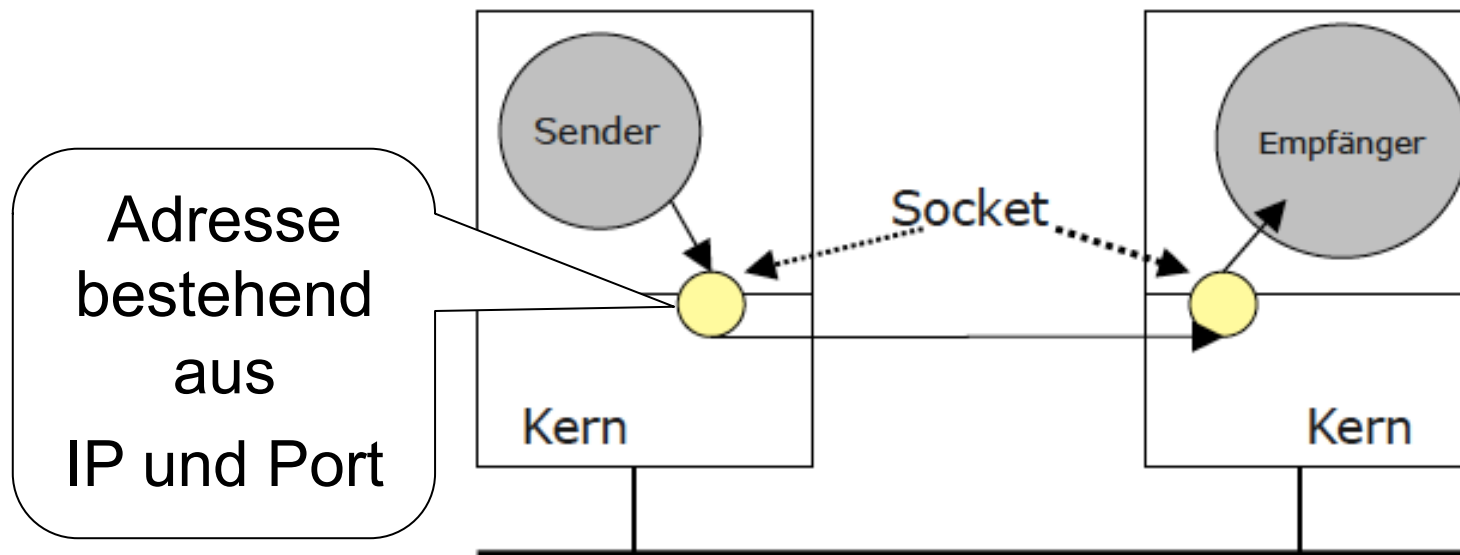
- ▶ Frage: Welches Protokoll ist für welche verteilte Anwendung am besten geeignet?
 - ▶ Videokonferenzen
 - ▶ Bank Transaktionen
 - ▶ Web Server
 - ▶ DNS

- ▶ Wichtig: Pakete bestehen aus
 - ▶ Protokolldaten: Adressen, Ports, TTL
 - ▶ Nutzdaten: was tatsächlich von der Anwendung übertragen werden soll
- ▶ Frage: Warum muss man das wissen?
- ▶ Anders formuliert: Für welche Qualitätsparameter ist das wichtig?

- ▶ Sockets
 - ▶ Ursprung: Berkeley Sockets
 - ▶ Erfunden in den 80er Jahren (Berkeley Sockets)
- ▶ Verfügbar in
 - ▶ C, C++, C#, Java, ...
 - ▶ ... sehr, sehr vielen Programmiersprachen

UDP / TCP Kommunikation mit Sockets

- ▶ Was ist ein Socket?
 - ▶ Ein Kommunikationsendpunkt (bi-direktional), für z.B. UDP / TCP Kommunikation, repräsentiert als Objekt im Speicher



UDP: Programmierung mit Sockets

- ▶ Jedes einzelne Datenpaket (bis 64k Größe) muss empfangen und gesendet werden (ach!)
- ▶ Entwickler sind verantwortlich für
 - ▶ Reihenfolgetreue
 - ▶ Fehlerkorrektur
 - ▶ Segmentierung großer Datenvolumen
- ▶ Ein Socket kann Datenpakete mit beliebig vielen verschiedenen Adressen austauschen (!)
- ▶ Entwickler können Multicast als Kommunikationsform vorsehen
 - ▶ Mehrere Empfänger
 - ▶ Spezielle IP Adressen

UDP: Programmierung

▶ API (Auszug): C (POSIX)

- | | |
|---------------------|--|
| ▶ socket | Socket erzeugen |
| ▶ bind | Socket an Adresse binden |
| ▶ sendto | Daten senden |
| ▶ recvfrom | Daten empfangen |
| ▶ close | Socket schließen |
| ▶ setsockopt | optional: Optionen (z.B.
Puffergröße, REUSE_ADDR) |
| ▶ ... | |

UDP: Programmierung

- ▶ API (Auszug): Java

- ▶ **DatagramSocket** Klasse für UDP Socket
IP und Port als Parameter
des Konstruktors (Funktionalität
wie bei **bind**)

- ▶ Methoden

- ▶ **send** senden
 - ▶ **receive** empfangen

- ▶ **setReceiveBufferSize** Puffergröße setzen
(optional)

- ▶ ...

UDP: Programmierung

► Ablauf (Schema: Server)

1. Socket erzeugen
2. Socket an Netzwerkadresse binden (IP und Port)
3. Daten senden und/oder empfangen
4. ggfs. weitere Operationen (mehr Daten senden, mehr Daten empfangen)
5. Socket schließen

was passiert,
wenn man das
nicht macht?

UDP: Programmierung

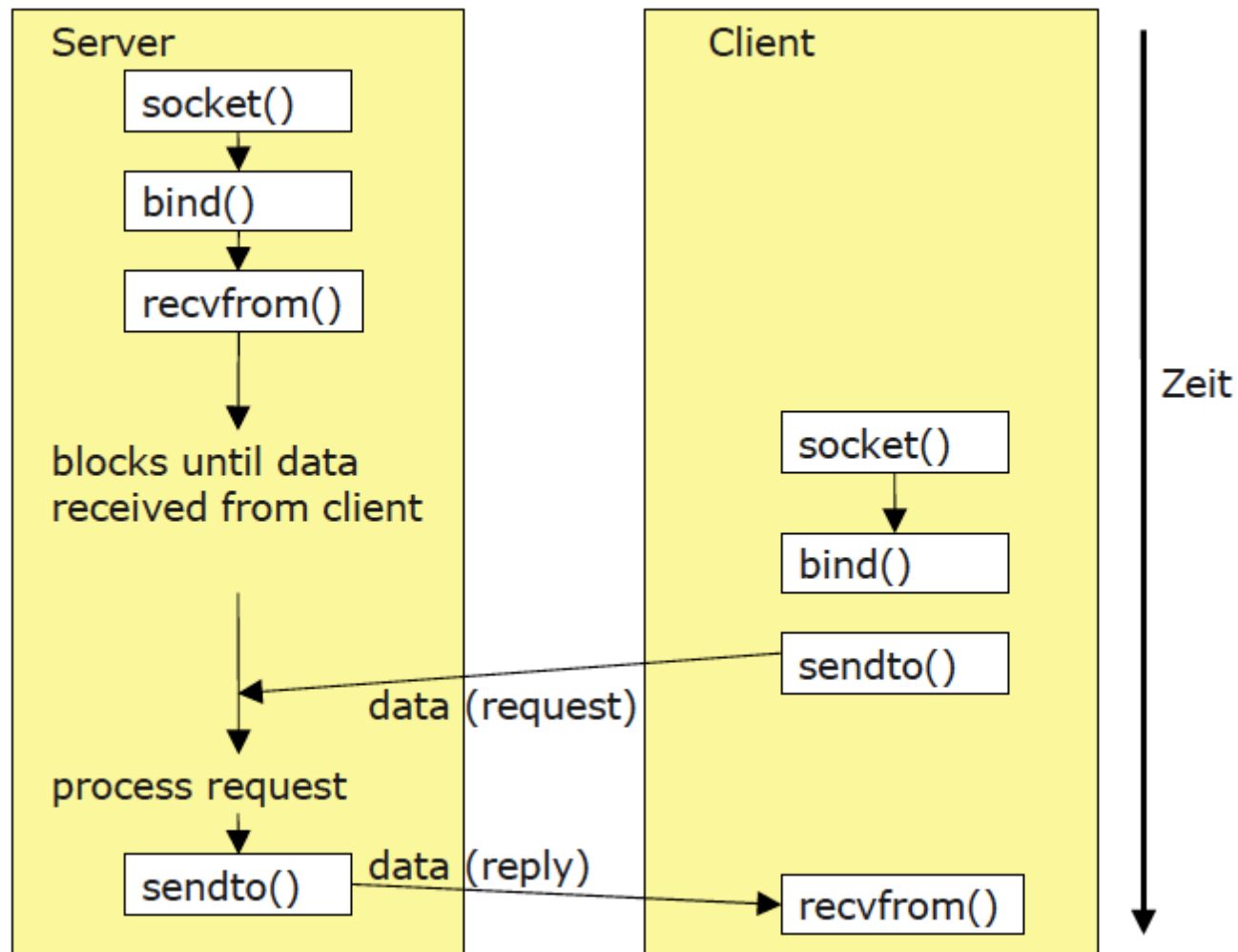
► Ablauf (Schema: Client)

1. Socket erzeugen
2. optional: Socket an Netzwerkadresse binden (IP und Port)
3. Daten senden und/oder empfangen
4. ggfs. weitere Operationen (mehr Daten senden, mehr Daten empfangen)
5. Socket schließen

nicht
notwendig

Port von Client
und Server
müssen nicht
identisch sein

UDP: Programmierung



- ▶ Probleme mit UDP
 - ▶ Paketverluste
 - ▶ Duplikate
 - ▶ Reihenfolge
 - ▶ Das bedeutet hier: Entwickler müssen sich um diese Probleme selbst kümmern

Beispiel: UDP Socket Server in C 1/2

```
int main()
{
    int my_socket;
    int result = -1;
    char buf[10];

    my_socket = socket(AF_INET, SOCK_DGRAM, 0);
    if (my_socket < 0) {
        perror("failed to create UDP socket\n");
        return -1;
    }
    // Adresse des Servers
    struct sockaddr_in server_addr;
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(54321);
    server_addr.sin_addr.s_addr = INADDR_ANY;

    result = bind(my_socket, (struct sockaddr*)&server_addr,
                  sizeof(struct sockaddr_in));

    if (result < 0) {
        return -1;
    }
}
```

Beispiel: UDP Socket Server in C 2/2

```
struct sockaddr_in client_addr;
int addr_len = sizeof(struct sockaddr_in);

result = recvfrom(my_socket,buf,20,0,(struct sockaddr*)&client_addr,
                  (socklen_t*)&addr_len);

if (result<0) {
    perror("Empfang fehlgeschlagen");
    return -1;
}
printf("Empfangen: '%s' (%d Bytes)\n",buf,result);
result = sendto(my_socket,buf,1,0,
                (struct sockaddr*)&client_addr,sizeof(struct sockaddr_in));
if (result<0) {
    perror("Senden fehlgeschlagen\n");
    return -1;
}
return 0;
}
```

TCP Programmierung mit Sockets

- ▶ Verbindungsorientiert
- ▶ Verbindung muss im Programm aufgebaut werden
- ▶ Rollenverteilung
 - ▶ Client: baut Verbindung auf
 - ▶ Server: nimmt Verbindung an
- ▶ Fehler (verlorene Pakete, falsche Reihenfolge) werden automatisch vom Socket korrigiert
- ▶ Datenstrom statt einzelner Pakete
 - ▶ Aufteilung der Daten in Pakete erfolgt automatisch
 - ▶ Beispiel: 2x5 Byte werden mit **send** verschickt, 1x10 Byte werden mit **recv** empfangen
 - ▶ Empfangsoperation liefert Daten unabhängig von der versendeten Größe

TCP: Programmierung

▶ API (Auszug): C/C++ (POSIX)

- | | |
|---------------------|--|
| ▶ socket | Socket erzeugen |
| ▶ bind | Socket an Adresse binden |
| ▶ listen | Verbindungsbereitschaft (Server) |
| ▶ accept | Verbindung annehmen (Server) |
| ▶ connect | Verbindung aufbauen (Client) |
| ▶ send | Daten senden |
| ▶ recv | Daten empfangen |
| ▶ setsockopt | Optionen (z.B. Puffergröße)
definieren (optional) |
| ▶ ... | |

TCP: Programmierung

- ▶ API (Auszug): Java

- ▶ **Socket** Klasse für TCP Client
Socket

- ▶ **ServerSocket** Klasse für TCP Server Socket
IP und Port als Parameter
des Konstruktors, **listen** wird
automatisch ausgeführt

- ▶ Methoden

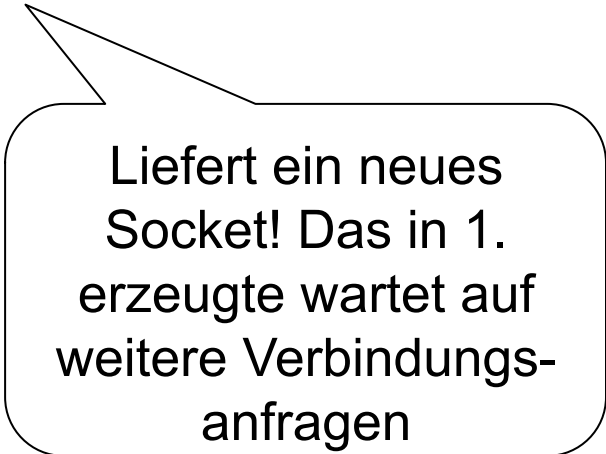
- ▶ **accept**

- ▶ **getOutputStream** Schreiben über separate
Klasse

- ▶ **getInputStream** Lesen über separate
Klasse

► Ablauf (Schema: Server)

1. Socket erzeugen
2. Socket an Netzwerkadresse binden (IP und Port)
3. Status auf Status *listen* setzen (danach können Verbindungsanfragen angenommen werden)
4. Verbindungsanfrage akzeptieren
5. Daten senden/empfangen
6. Socket schließen

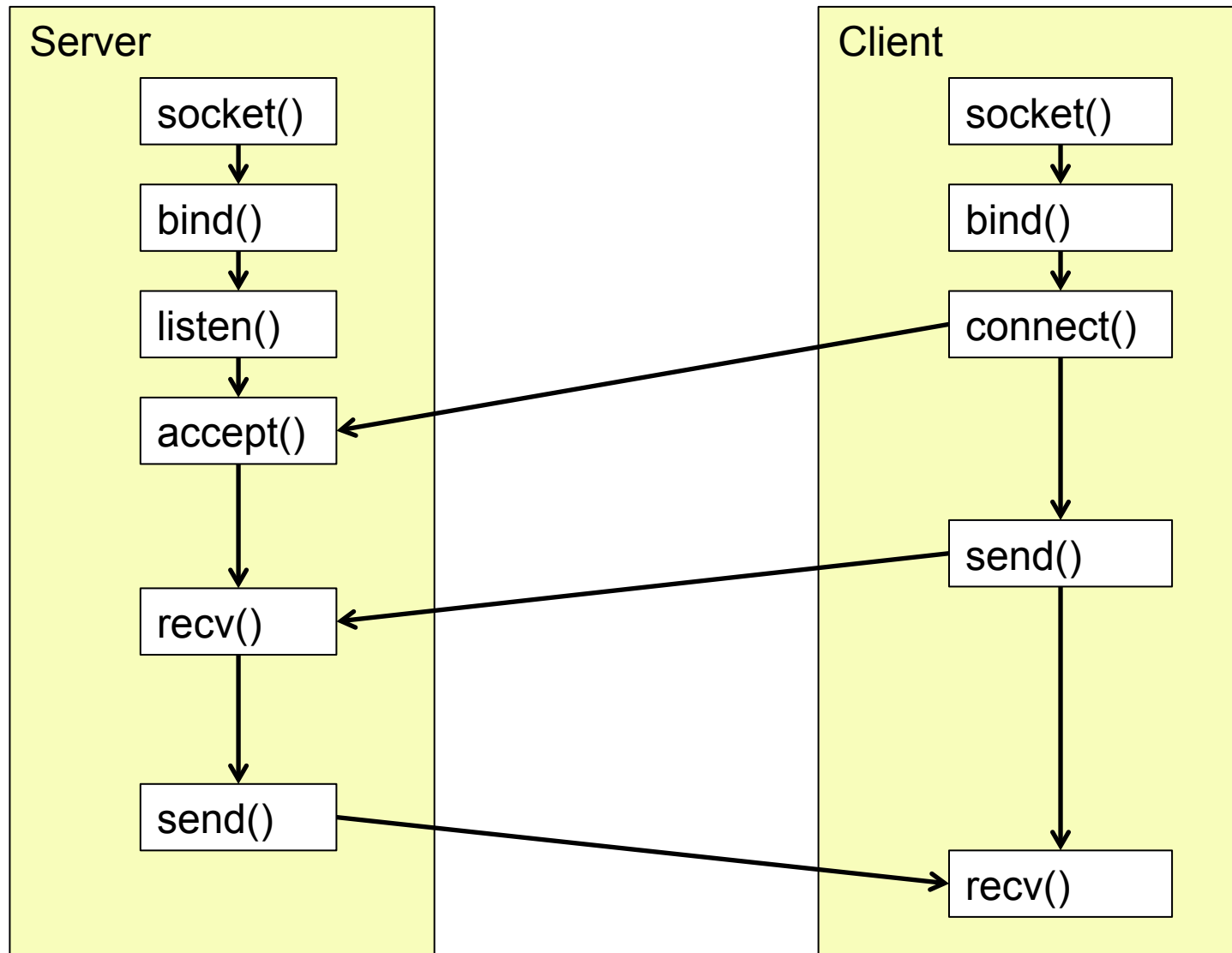


Liefert ein neues Socket! Das in 1. erzeugte wartet auf weitere Verbindungsanfragen

► Ablauf (Schema: Client)

1. Socket erzeugen
2. optional: Socket an Netzwerkadresse binden (IP und Port)
3. Verbindungsanfrage an Server senden
4. Auf Verbindungsbestätigung warten
5. Daten senden/empfangen
6. Socket schließen

TCP: Programmierung



- ▶ Probleme mit TCP (?)
 - ▶ Paketverluste? Nein
 - ▶ Duplikate? Nein
 - ▶ Reihenfolge? Nein
 - ▶ Automatische Sortierung und erneutes Senden
 - ▶ Entwickler müssen sich darum nicht kümmern
- ▶ Frage:
 - ▶ Was passiert in der Anwendung wenn Pakete verloren gehen oder vertauscht werden?

▶ Antwort:

- ▶ Der Datenstrom wird verzögert oder angehalten
- ▶ Empfangen (und Senden) u.U. nicht mehr möglich bzw. nur noch mit großer zeitlicher Verzögerung
- ▶ Im schlimmsten Fall: Verbindung bricht ab (`recv` liefert Fehlercode)
- ▶ s. TCP Sliding Window Protocol

Beispiel: TCP Sockets in C (Server) 1/2

```
int main(int argc, char **argv)
{
    int          sockfd, newsockfd, clilen, childpid;
    struct sockaddr_in  cli_addr, serv_addr;

    // Open a TCP socket (an Internet stream socket).
    if ( (sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0)
        perror("server: can't open stream socket");

    // Bind our local address so that the client can send to us.

    bzero((char *) &serv_addr, sizeof(serv_addr));
    serv_addr.sin_family      = AF_INET;
    serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    serv_addr.sin_port       = htons(SERV_TCP_PORT);

    if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0)
        perror("server: can't bind local address");

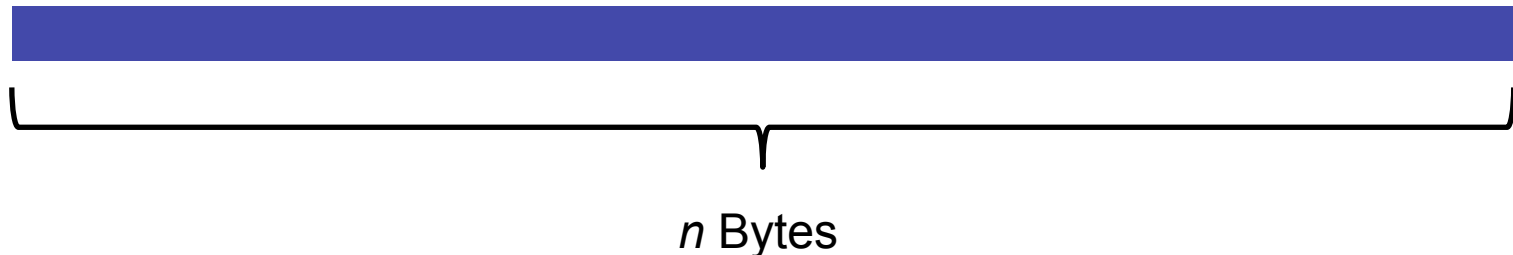
    listen(sockfd, 5);
```

Beispiel: TCP Sockets in C (Server) 2/2

```
for ( ; ; ) {  
    // Wait for a connection from a client process.  
    // This is an example of a concurrent server.  
  
    clilen = sizeof(cli_addr);  
    newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr,  
                        (socklen_t *) &clilen);  
  
    if (newsockfd < 0)  
        perror("server: accept error");  
  
    if ( (childpid = fork()) < 0)  
        perror("server: fork error");  
  
    else if (childpid == 0) {    // child process  
        close(sockfd);          // close original socket  
        str_echo(newsockfd);    // process the request  
        exit(0);  
    }  
    close(newsockfd);           // parent process  
}  
}
```

TCP Datenstrom

- ▶ Auswirkung: Datenstrom statt einzelner Pakete
- ▶ Szenario
 - ▶ n Bytes sollen verschickt werden



TCP Datenstrom

- ▶ Frage: Wie viele **send()** Aufrufe benötigt man für die n Byte?
- ▶ Antwort
 - ▶ 1 Aufruf von **send()** wird benötigt
- ▶ Frage
 - ▶ Woher weiß man das?
- ▶ Antwort: man page !!!
 - ▶ "If no messages space is available at the socket to hold the message to be transmitted, then **send()** normally blocks"

TCP Datenstrom

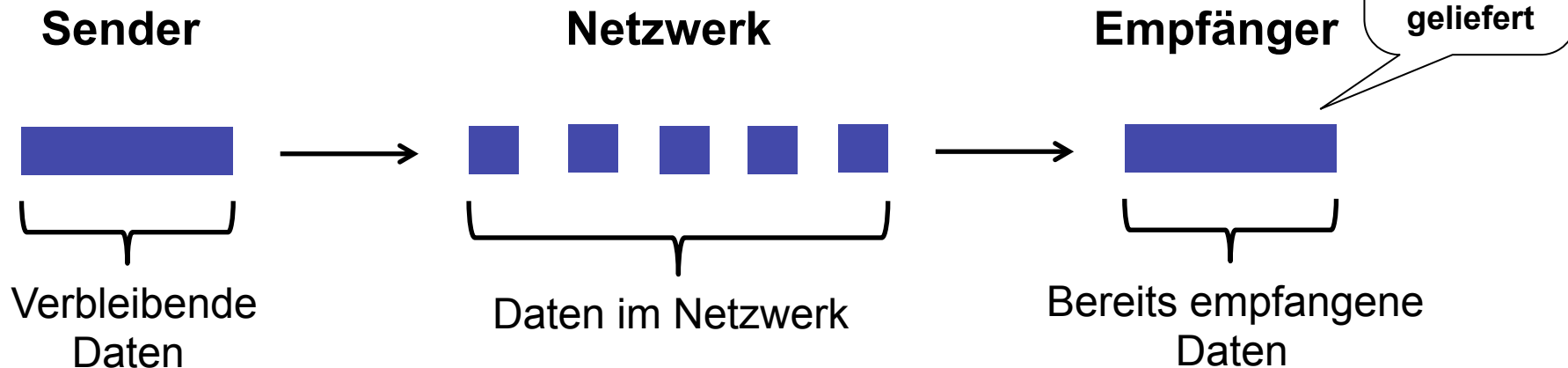
- ▶ Frage: Wie viele `recv()` Aufrufe benötigt man, um die gesendeten n Bytes zu empfangen?

1. 1 Aufruf
2. 10 Aufrufe
3. n Aufrufe
4. Nicht zu beantworten

- ▶ Antwort: 4. Das kann man nicht beantworten
- ▶ Frage: Warum kann man das nicht beantworten (immerhin garantiert TCP das alle Daten fehlerfrei übertragen werden) ?

TCP Datenstrom

- ▶ Szenario: Versenden von n Bytes
- ▶ Momentaufnahme:

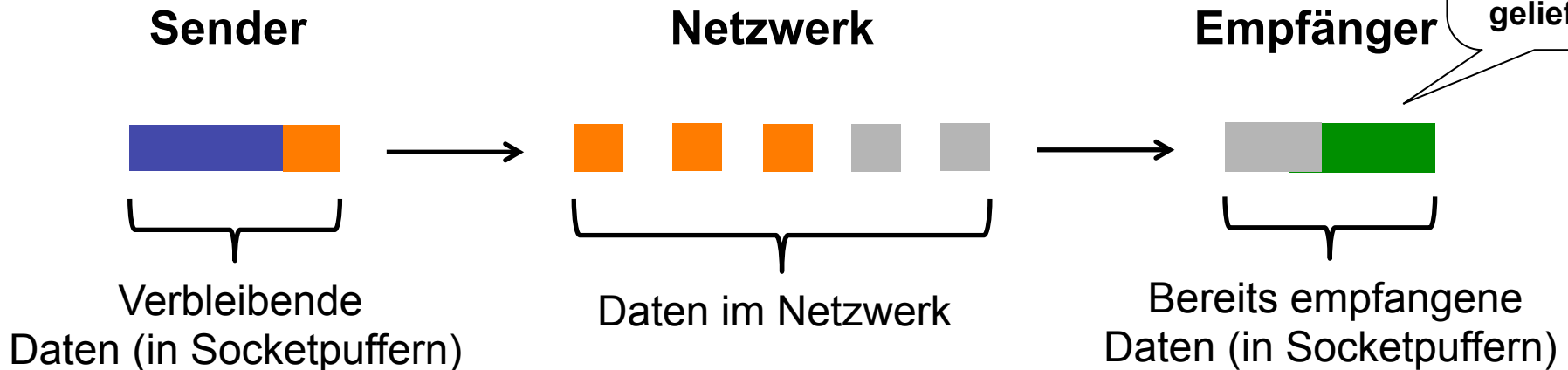


- ▶ Zu einem Zeitpunkt können sich Daten sowohl auf Senderseite, im Netzwerk als auch auf Empfängerseite befinden.
- ▶ Wenn man in der o.a. Situation im Empfänger `recv()` aufruft, erhält man nur die bereits empfangenen Daten

TCP Datenstrom

- ▶ Szenario: Versenden von mehreren Nachrichten
- ▶ Momentaufnahme (nach `send()` Aufruf(en)):

mit dem
nächsten
`recv()`
Aufruf
geliefert



- ▶ Zu einem Zeitpunkt können sich Daten **mehrerer** Nachrichten sowohl auf Senderseite, im Netzwerk als auch auch Empfängerseite befinden.
- ▶ Wenn man in der o.a. Situation im Empfänger `recv()` aufruft, erhält man evtl. Daten von mehreren Nachrichten

TCP Sockets

- ▶ Man kann in einem `recv()` Aufruf manchmal Flags setzen, die erzwingen, dass `recv()` erst zurückkehrt wenn die gewünschte Menge Bytes zurückgeliefert werden kann
- ▶ Aber: das hilft NICHT bei Nachrichten variabler Länge, d.h. wenn man nicht weiß, wie viele Bytes empfangen werden müssen (Bsp. HTTP)
- ▶ Generell: IMMER den Rückgabewert von `recv()` prüfen

Nachrichtenaustausch mit UDP / TCP

▶ UDP

- ▶ Man definiert eine Nachricht (<64k) und verschickt diese mittels **sendto**
- ▶ Das Socket auf Empfängerseite empfängt mit einem (!) Aufruf von **recvfrom** entweder
 - ▶ die ganze Nachricht oder
 - ▶ gar nichts
 - ▶ (oder einen Teil der Nachricht)
- ▶ Man muss also die empfangenen Nachrichten in die richtige Reihenfolge zu bringen und sich um verlorene Nachrichten kümmern
- ▶ Frage: Woher weiß das Socket wie lang die Nachricht ist?

Selber schuld: dann war der Puffer nicht ausreichend groß!

▶ UDP

- ▶ Frage: Wie verschickt man Nachrichten >64k ?
- ▶ Antwort: Aufteilen. Um Fehlerkorrektur, Reihenfolge, etc. muss man sich dann selbst kümmern

Nachrichtenaustausch mit UDP / TCP

▶ TCP

- ▶ Man definiert eine Nachricht und verschickt diese mittels **send**
- ▶ Das Socket auf Empfängerseite empfängt mit einem (!) Aufruf von **recv** entweder
 - ▶ die ganze Nachricht oder
 - ▶ gar nichts oder
 - ▶ Teile von einer oder mehreren Nachrichten
- ▶ Man muss sich auch darum kümmern, dass eine Nachricht vollständig empfangen wurde
- ▶ Frage: Woher weiß das Socket wie lang eine Nachricht ist?
- ▶ Antwort: s. Praktikum / HTTP



egal wie groß
der Puffer ist!!!

Asynchrone Kommunikation

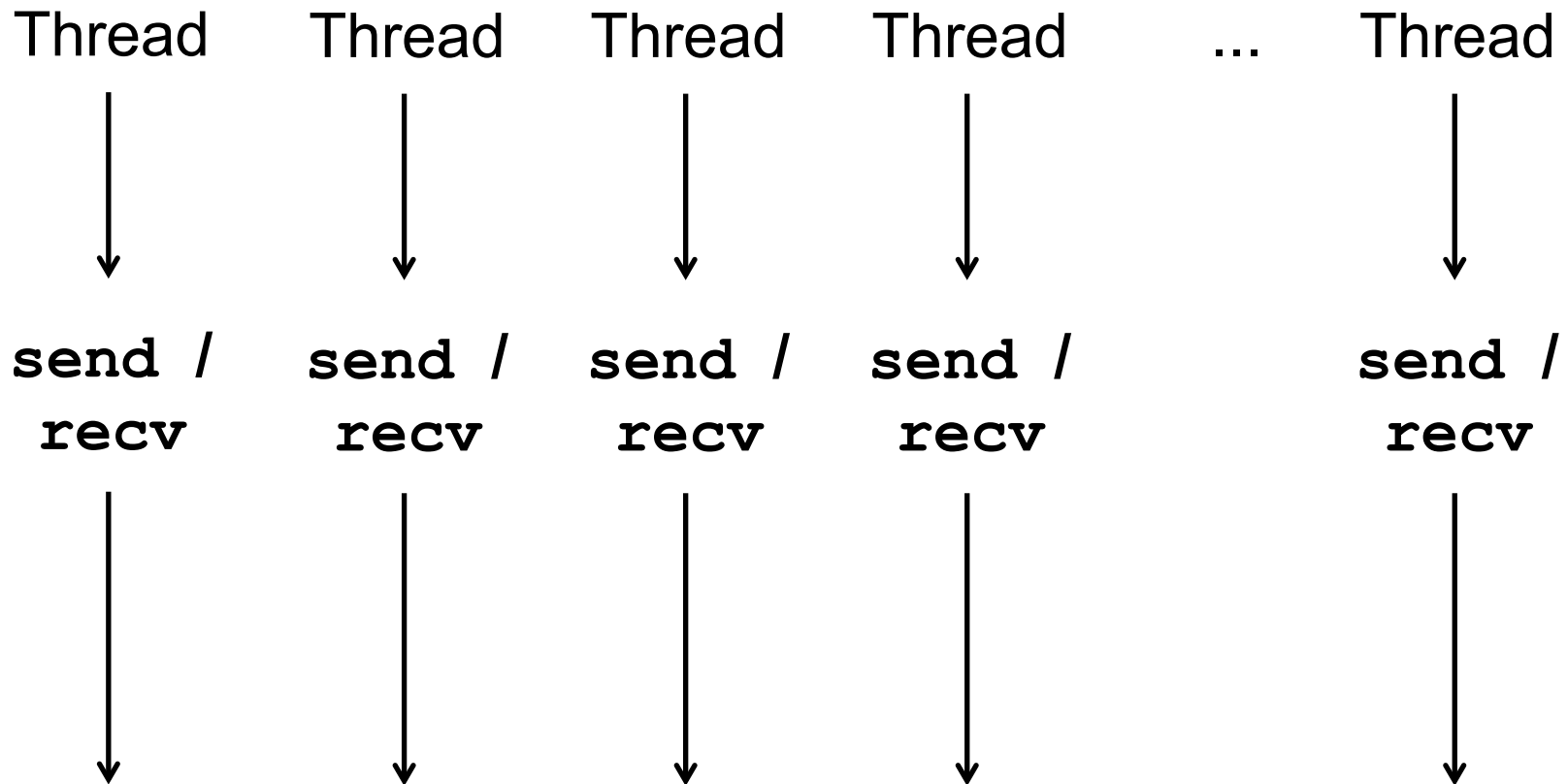
- ▶ Sockets können auch in Modus *nicht-blockierend* gesetzt werden
 - ▶ z.B. in C beim Erzeugen mittels **socket** (nur Linux)
 - ▶ über **fcntl**
 - ▶ **send** / **recv** Aufrufe kehren sofort zurück, Rückgabewert zeigt Erfolg an
- ▶ Frage: Woher weiß man, dass etwas empfangen werden kann?
 - ▶ Idee: Man überprüft (durch wiederholte **recv** Aufrufe) so lange bis etwas empfangen werden kann (*polling*)
 - ▶ Zwischen **recv** Aufrufen kann man noch andere Aufgaben erledigen
 - ▶ Unschön (warum?), aber möglich
 - ▶ Blockieren wird vermieden, Prozess bleibt aktiv

Sockets: Asynchrones Verhalten

- ▶ Frage: Wie kann man asynchrones Verhalten noch erreichen?
 1. n Prozesse/Threads arbeiten mit n Sockets
 2. Abhängig von BS / Umgebung: spezielle Benachrichtigungsfunktionalität (z.B. UNIX Signale)
 3. Spezielle Funktion **select** (in C, Java hat ähnlichen Mechanismus) überwacht eine Menge von Sockets auf Aktivität

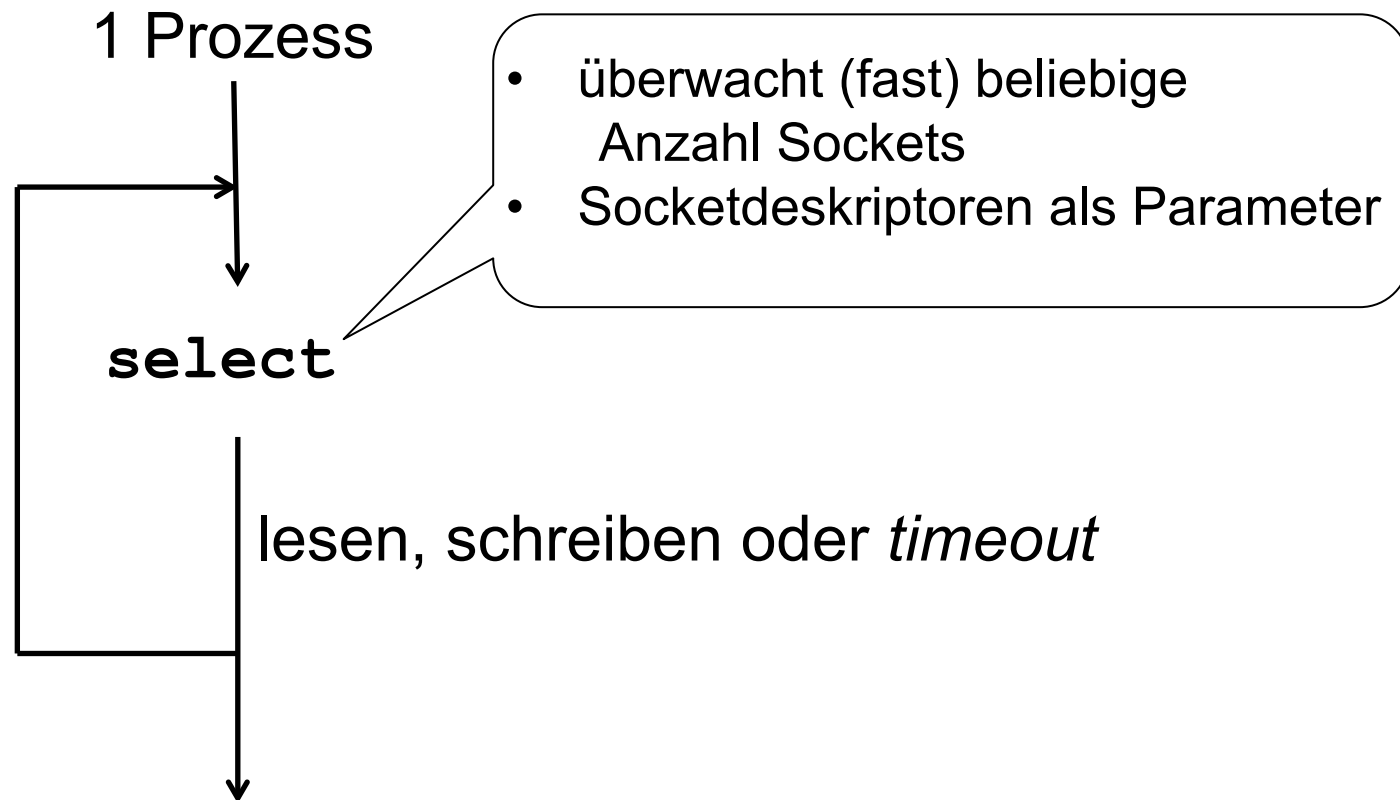
select

Statt vieler blockierender **send/recv** Aufrufe ...



select

... 1 Prozess der für **n** Sockets (alle gleichzeitig!) auf Aktivität warten kann



select

- ▶ **select** blockiert (!), allerdings kann man einen *timeout* festlegen
- ▶ **select** kann zu einem Socket verschiedene Ereignisse melden
 - ▶ Empfangen möglich (d.h. **recv** liefert Daten)
 - ▶ Senden möglich (d.h. **send** sendet Daten)
 - ▶ Fehler (z.B. Verbindung abgebrochen)

select

- ▶ **select** kann für UDP und TCP verwendet werden
- ▶ Hinweis: **select** ist nur für begrenzte Anzahl von Sockets geeignet, bei mehr als 1000 Sockets gibt es ähnliche aber bessere Mechanismen (**epoll**, ...)
 - ▶ Beispiel: <http://www.lighttpd.net/> (Webserver)
- ▶ Alternative:
 - ▶ Signale/Events mit callback Funktionen bei manchen Betriebssystemen (z.B. UNIX, Windows)
 - ▶ Abhängig vom Betriebssystem / Umgebung

Codebeispiel: select

1 Prozess, mehrere Sockets

Code:

Datentyp f.
Menge von
Sockets

max.
Socket ID
+1

```
int main() {
    [...]
    fd_set rdfs;
    int sock[MAX_SOCKETS], i, active_sock;
    char recv_message[BUFSIZE];

    while (1) {
        FD_ZERO(&rdfs);
        for (i=0; i<MAX_SOCKETS; i++)
            FD_SET(sock[i], &rdfs);

        if (select(MAX(rdfs)+1, &rdfs, NULL, NULL, NULL) > 0) {
            // get socket with activity using FD_ISSET
            recv(active_sock, recv_message, BUFSIZE, 0);
            // do something
        }
    }
    [...]
}
```

Parameter für
timeouts

select

- ▶ Frage: Warum ist das asynchron? **select** selbst blockiert doch auch...
- ▶ Antwort: **select** kann in separatem Thread laufen, dieser Thread übernimmt die Benachrichtigung
 - => nur ein einziger Thread blockiert (besser als pro Socket einen Thread zu blockieren)

Was kann man mit Sockets anfangen?

- ▶ Versenden / Empfangen von Daten aus Anwendungen
- ▶ Frage: Was braucht man sonst noch zum Datenaustausch, also um ein funktionierendes verteiltes System aufzubauen?
- ▶ Antwort: ein gemeinsames Datenformat
 - ▶ Wie sehen Datenpakete aus und welche Bedeutung haben diese?

- ▶ Frage: Was passiert, wenn man einen Integer (4 Byte) über ein Socket zu einem anderen Rechner schickt und dort ausgibt ?

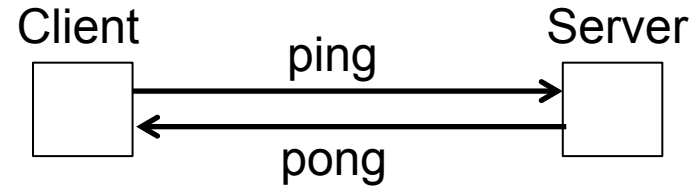
Big Endian / Little Endian

- ▶ Umwandlung von Zahlenformaten
- ▶ API (POSIX, C/C++)
 - ▶ `ntohs` *network-to-host short*
 - ▶ `ntohl` *network-to-host long*
 - ▶ `htons` *host-to-network short*
 - ▶ `htonl` *host-to-network long*
- ▶ Network byte order: big endian
- ▶ Z.B. nötig um den Port korrekt in die Adressstruktur von `bind`, `sendto`, etc. einzutragen:

```
server_addr.sin_port = htons(12345);
```


Sockets: Beispiel Echo Server 1/2

► Server



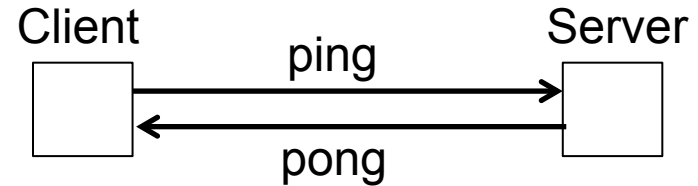
```
#include <socket.h>          // ... und noch mehr header, s. Praktikum

void main()
{
    int mySock = socket(AF_INET, SOCK_DGRAM, 0);

    // Adresse des Servers
    struct sockaddr_in server_addr, client_addr;
    server_addr.sin_family = AF_INET;
    server_addr.sin_port = htons(54321); // Port des Servers
    server_addr.sin_addr.s_addr = INADDR_ANY; // alle auf dem Rechner
                                           // verfügbaren IP Adressen
    result = bind(my_socket, (struct sockaddr*)&server_addr,
                  sizeof(struct sockaddr_in));
    if (result < 0) {
        perror("bind fehlgeschlagen");
        return -1;
    }
    [...]
}
```

Sockets: Beispiel Echo Server 2/2

► Server



```
[...]
// bind erfolgreich

// Anfrage empfangen
int addr_len = sizeof(struct sockaddr_in);
result = recvfrom(my_socket, buf, 10, 0, (struct sockaddr*)&client_addr,
                  (socklen_t*)&addr_len);

if (result < 0) {
    perror("Empfang fehlgeschlagen");
    return -1;
}
[...]
```

Immer korrekt initialisieren!

```
result = sendto(my_socket, buf, 10, 0, (struct sockaddr*)&client_addr,
                sizeof(client_addr));

if (result < 0) {
    perror("Senden fehlgeschlagen\n");
    return -1;
}
[...]
```

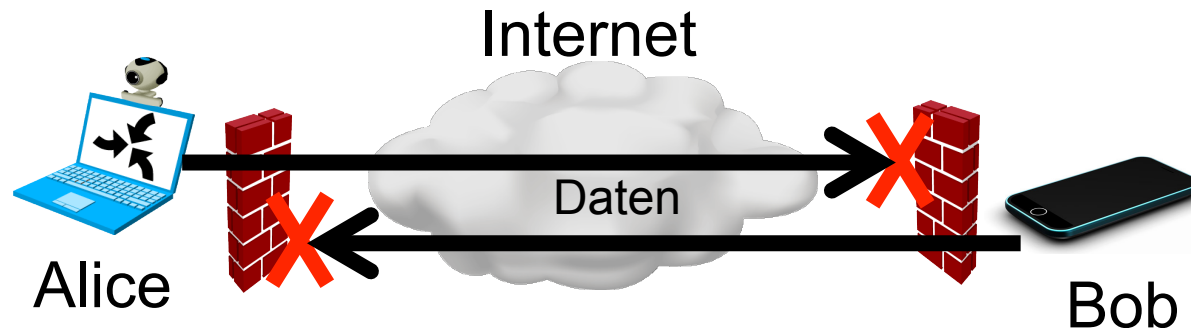
Nachrichtenaustausch mit Sockets

- ▶ In einem realen verteilten System mit Sockets müssen Entwickler selbst ...
 - ▶ das Nachrichtenformat festlegen, d.h. Inhalt und Länge von Nachrichten definieren
 - ▶ Fehlerbehandlung durchführen (TCP oder UDP)
 - ▶ Nachrichtenverarbeitung implementieren (Parsing, Typkonvertierung, ggfs. **select**, ...)

Probleme mit TCP/UDP

i.d.R. keine
(einfache)
Lösung

- ▶ Beispiel: Sicherheitsmechanismen
 - ▶ Firewalls / NATs
 - ▶ Sicherheitsmechanismen auf Rechnern, z.B. Personal Firewalls, SELinux
 - ▶ Blockieren Zugang zu Rechnern und Anwendungen
 - ▶ Bekanntes Problem bei Peer-to-Peer Verbindungen zwischen 2 Rechnern



Fazit: Sockets

- ▶ Frage: Was bringen Sockets im Hinblick auf die Transparenzeigenschaften (Folie 1-19) ?
 - ▶ Zugriffstransparenz?
 - ▶ Ortstransparenz?
 - ▶ Migrationstransparenz?
 - ▶ Nebenläufigkeitstransparenz?
 - ▶ Fehlertransparenz?

- ▶ Frage: was bringen Sockets im Hinblick auf die anderen Ziele eines verteilten Systems (Folie 1-17) ?
 - ▶ Offenheit?
 - ▶ Skalierbarkeit?
 - ▶ Zuverlässigkeit?
 - ▶ Einfacher Zugriff auf Ressourcen?

Fazit: Sockets

- ▶ Damit kann man arbeiten, aber...
- ▶ ... Entwickler müssen sehr viele (häufig wiederkehrende Aufgaben) selbst erledigen
 - ▶ Sockets erzeugen und an Adresse binden
 - ▶ Senden, Empfangen aufrufen
 - ▶ Datenformat festlegen
 - ▶ Fehler behandeln
 - ▶ ...
- ▶ Es wäre schöner, wenn es etwas gäbe das solche Standardaufgaben übernimmt