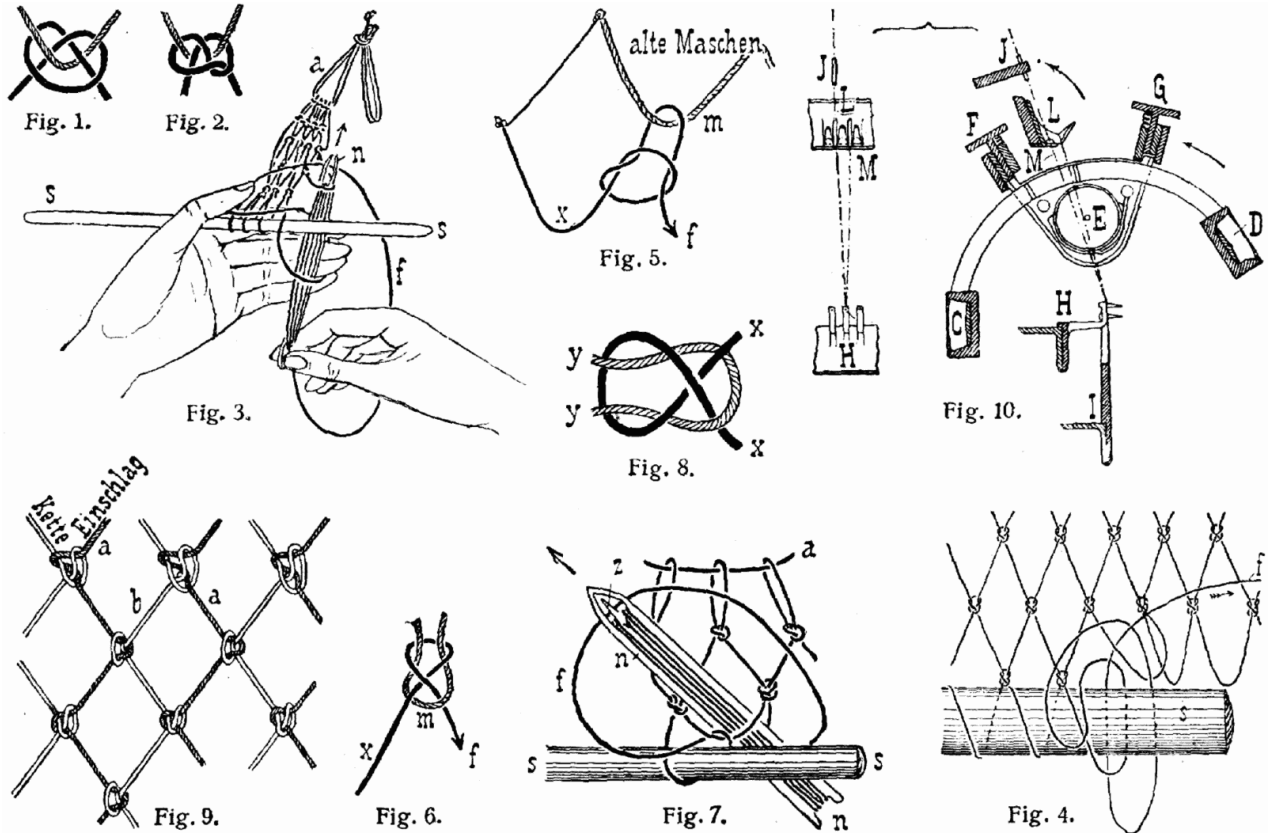
Verteilte Systeme

R. Kaiser, R. Kröger, O. Hahm

(HTTP: http://www.cs.hs-rm.de/~kaiser
E-Mail: robert.kaiser@hs-rm.de)
Kai Beckmann
Sebastian Flothow
Alexander Schönborn

Sommersemester 2021

2. Netzwerkprogrammierung



https://de.wikipedia.org/wiki/Fischernetz#/media/File:L-Netze.png

Inhalt

Hochschule RheinMain

Einführung

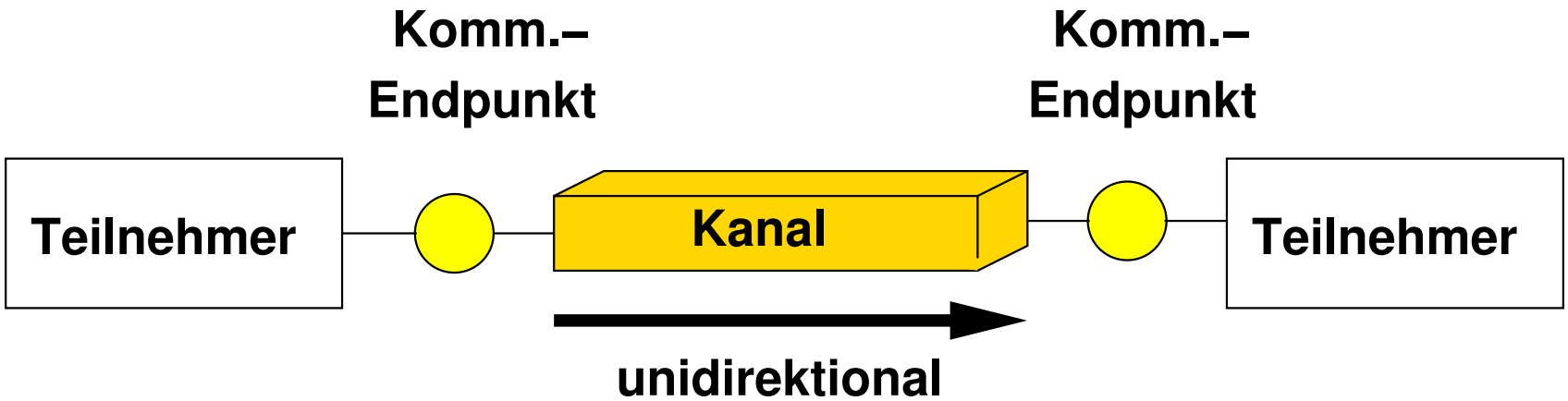
Hochschule RheinMain

Ziele

- Kennenlernen eines allgemeinen nachrichtenorientierten Kommunikationsdienstes hoher praktischer Relevanz (Internet mit TCP/IP-Protokollen) und dessen Programmierschnittstelle (Sockets)
- API zur TCP/IP-Funktionalität der LV „Rechnernetze“ (vgl. 2. Sem.)
- Höhere Kommunikationsdienste (Remote Procedure Calls (RPCs), Web Services) bauen hierauf auf und werden z.T. im weiteren dieser LV behandelt.
- **Merke**: Höhere Kommunikationsdienste oder Middleware-Plattformen offerieren eine abstraktere, auf das jeweilige Kooperations-Paradigma ausgerichtete Schnittstelle, basieren aber intern auf den hier zunächst besprochenen Konzepten eines unterlagerten Kommunikationssystems.

Grundlagen der Kommunikation

- Jegliche Interaktion zwischen Beteiligten verlangt eine zugrunde liegende Kommunikationsmöglichkeit.
- Kommunikationskanal
  - Einrichtung zur Verbindung/Kopplung der Kommunikationspartner heisst Kommunikationskanal oder Kanal.
- Richtung des Nachrichtenflusses eines Kanals
  - Ein Kanal heisst gerichtet oder unidirektional, wenn ein Prozess ausschließlich die Sender-Rolle, der andere ausschließlich die Empfänger-Rolle ausübt, ansonsten heißt er ungerichtet oder bidirektional



Aspekte der Kommunikation

Wichtige Aspekte der Kommunikation

1. Anzahl Kommunikationspartner
2. Adressierung
3. Pufferung
4. Synchronisation
5. Kommunikationsmuster
6. Nachrichtenstruktur

Anzahl der Teilnehmer

Hochschule RheinMain

Anzahl der Kommunikationsteilnehmer eines Kanals

- Genau zwei:
  - einfachster Fall
  - Regelfall
- Mehr als zwei:
  - Für bestimmte Anwendungen können Gruppenkommunikationsformen sinnvoll sein.
  - sogenannter Multicast-Dienst
  - Spezialfall: Broadcast
  - vgl. 2.3.

Adressierung

Direkte Adressierung

- Kommunikationspartner haben eindeutige Adressen.
- Benennung ist **explizit** und **symmetrisch**: ein Sender muss den Empfänger benennen und umgekehrt.

Beispiel:

`SEND ( P, message )` - Sende Nachricht an Prozess P.

`RECEIVE ( Q, message )` - Empfange Nachricht von Prozess Q.

- Asymmetrische Variante (z.B. für Server-Prozesse):
  Nur der Sender benennt den Empfänger, dem Empfänger (Server) wird mit dem Empfang i.d.R. die Identität des Senders bekannt.

Beispiel:

```
SEND ( P, message )
RECEIVE ( sender_id , message )
```

Adressierung (2)

Hochschule RheinMain

Indirekte Adressierung

- Kommunikation erfolgt indirekt über zwischengeschaltete Instanzen.
- Vorteile:
  - Verbesserte Modularität
  - Menge der Partner kann transparent restrukturiert werden, z.B. nach Ausfall eines Beteiligten.
  - Erweiterte Zuordnungsmöglichkeiten von Sendern und Empfängern, wie z.B. m:1, 1:n, m:n.
  - Zwischeninstanz kann
    - nur weiterleiten
    - speichern und weiterleiten (store-and-forward)
    - Nachrichten transformieren

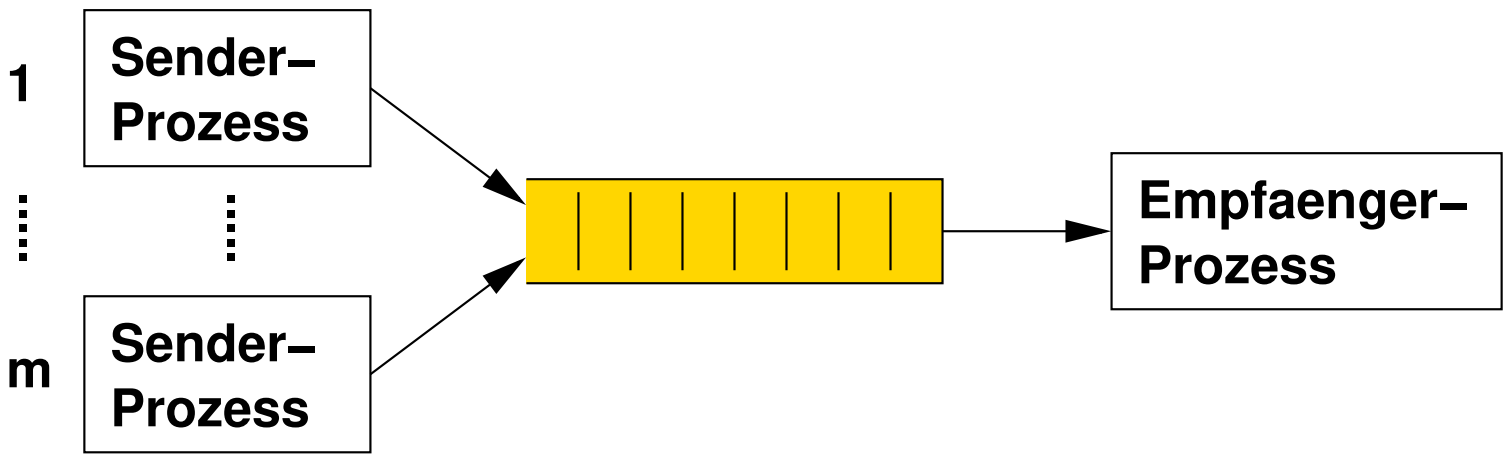
Beispiel indirekte Adressierung

Hochschule RheinMain

Mailbox:

`SEND ( mbox, message )` - Sende eine Nachricht an Mailbox `mbox`.

`RECEIVE ( mbox, message )` - Empfange Nachricht von Mailbox `mbox`.



Pufferung

- Kapazität eines Kanals:
  Anzahl der Nachrichten, die vorübergehend in einem Kanal gespeichert werden können, um Sender und Empfänger zeitlich zu entkoppeln.
- Die Pufferungsfähigkeit eines Kanals wird i.d.R. durch einen Warteraum/Warteschlange erreicht.
- In verteilten Systemen wird ein Warteraum i.d.R. auf der Empfängerseite gehalten (Rendezvous-Seite).
- Pufferung kann ordnungserhaltend sein oder auch die Sendeordnung verändern

Pufferung (2)

Keine Pufferung (Kapazität Null)

- Ungepufferte Kommunikationsform
- Sender und Empfänger werden zeitlich sehr eng koppelt
- auch als *Rendezvous* bezeichnet.
- Rendezvous wird häufig als zu inflexibel angesehen.
- Beispiel:
  - Ein Sender wird blockiert, wenn seine SEND-Operation vor einer entsprechenden `RECEIVE`-Operation stattfindet. Wird dann die entsprechende `RECEIVE`-Operation ausgeführt, wird die Nachricht ohne Zwischenspeicherung unmittelbar vom Sender- zum Empfänger-Prozess kopiert.
  - Findet umgekehrt `RECEIVE` zuerst statt, so wird der Empfänger bis zum Aufruf der `SEND`-Operation blockiert.
- Beispiel: Hoare: Communicating Sequential Processes (CSP).
- Beispiel: Kommunikation zwischen Ada-Tasks.
- Beispiel: Kommunikation zwischen Threads in L4 Mikrokern.

Pufferung (3)

Hochschule RheinMain

Beschränkte Kapazität

- Kanal kann zu einem Zeitpunkt maximal N Nachrichten enthalten (Warteraum der Kapazität N).
- `SEND`-Operation bei nicht-vollem Warteraum:
  - Nachricht wird im Warteraum abgelegt,
  - Sendeprozess fährt in seiner Berechnung fort.
- Warteraum voll (er enthält N gesendete aber noch nicht empfangene Nachrichten):
  - Sender wird blockiert, bis ein freier Warteplatz vorhanden ist, oder die Nachricht wird verworfen.
  - Analog wird ein Empfänger bei Ausführung einer `RECEIVE`- Operation blockiert, wenn der Warteraum leer ist.

Pufferung (4)

Hochschule RheinMain

Konsequenzen

- Gepufferte Kommunikation bewirkt zeitlich lose Kopplung der Kommunikationspartner.
- Übergabe der Nachricht an das Kommunikationssystem bedeutet für den Sender nicht, dass der Empfänger die Nachricht erhalten hat. Er kennt i.d.R. auch keine maximale Zeitdauer, bis dieses Ereignis eintritt.
- Wenn dieses Wissen wesentlich für den Sender ist, muss dazu eine explizite Kommunikation zwischen Sender und Empfänger durchgeführt werden:

Prozess P (Sender):		Prozess Q:
`...`		`...`
`send ( Q, message );`	$\longrightarrow$	`receive ( P, message );`
`receive ( Q, reply);`	$\longleftarrow$	`send ( P, "acknowledgement" );`
`...`		`...`

Synchronisation

Blocking oder Blockierend

- Sender kann beim Aufruf blockiert werden

Non-blocking oder Nicht-blockierend

- Sender kann beim Aufruf nicht blockiert werden
- kann damit unmittelbar weiterarbeiten

Kommunikationsmuster

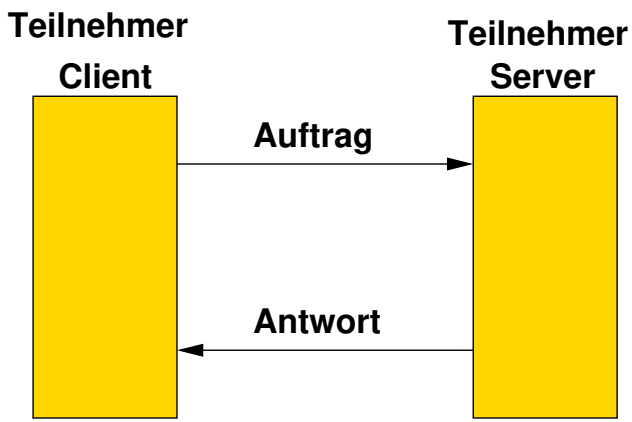
Hochschule RheinMain

One-Way

- Einzelnachricht ohne Antwort oder Quittung

Request/Response oder Auftrag/Antwort

- Client-Rolle (Auftraggeber)
- Server-Rolle (Auftragnehmer)
- häufig blockierend beim Auftraggeber (vgl. Standard-RPC)

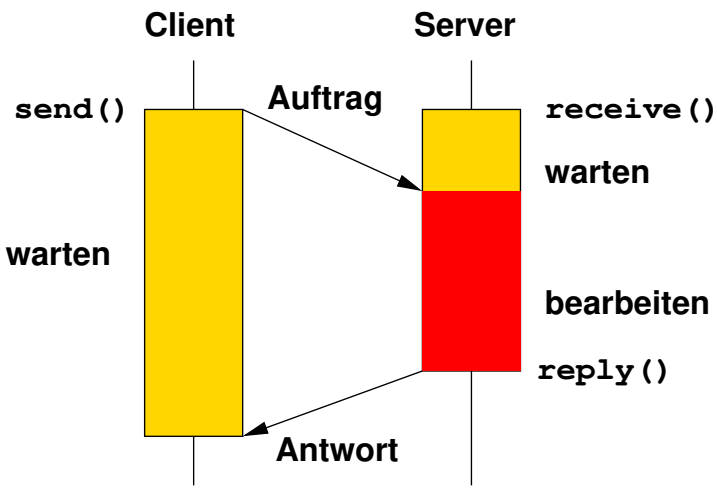


Erweiterung

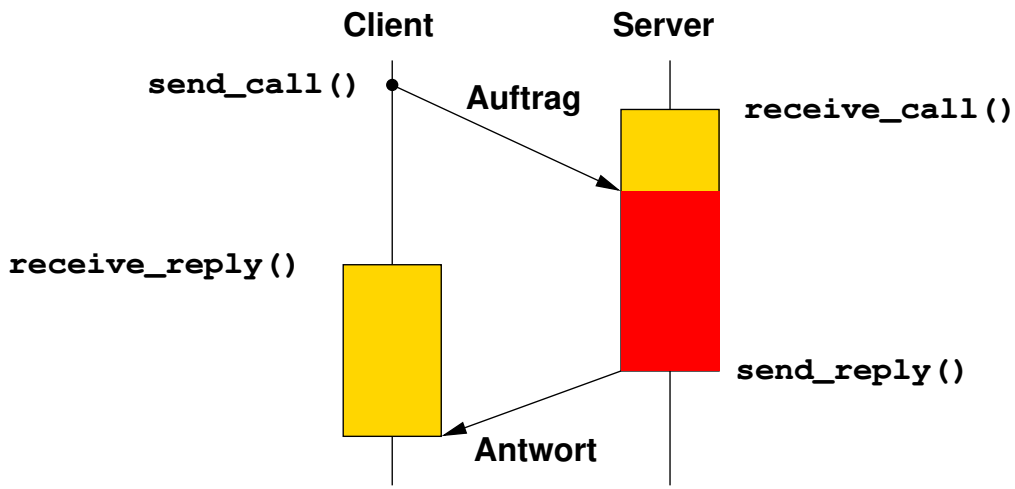
Hochschule RheinMain

Unterschiedliche Synchronität bei Request/Response:

- synchroner Aufruf: Sender blockiert bis zum Abschluss des Kommunikationsvorgangs (hier: Eintreffen der Antwort).
  ⇒ keine Parallelität.
- asynchroner Aufruf: Sender wird nur für die Einleitung des Kommunikationsvorgangs (Übergabe der Nachricht an das Kommunikationssystem) verzögert.



(a) synchron



(b) asynchron

Kommunikationsmuster (2)

Hochschule RheinMain

Publisher / Subscriber-Modell

- Nachrichten klassifiziert in Topics oder Event Channel
- Empfänger abonnieren Topics (Subscriber)
- Sender publizieren Nachrichten oder Events (Publisher)
- Modell erlaubt transparentes Senden einer Nachricht an mehrere Empfänger!
- Beispiele: CORBA Notification Service, OMG DDS, MQTT

Kommunikationsmuster (3)

komplexere Kommunikationsmuster

- nicht besonders üblich mit einfachen Kommunikationssystemen
- Ausnahmebeispiel: Three-Way Handshake zwischen zwei Teilnehmern zum zuverlässigen Verbindungsaufbau
- komplexere Formen entstehen bei Gruppenkommunikation
- sehr verbreitet auf höheren Ebenen
- Beispiel: Geschäftsprozesse

Semantik von Nachrichten

Bytestrom

- Übergebene Nachrichten verschiedener `SEND`-Operationen sind als Einheiten nicht mehr identifizierbar.
  Nachrichtengrenzen gehen verloren.
- Der Empfänger (und das Nachrichtensystem) sehen ausschließlich eine Folge von Zeichen (Bytestrom).
- Beispiel: UNIX pipes

Nachrichtencontainer

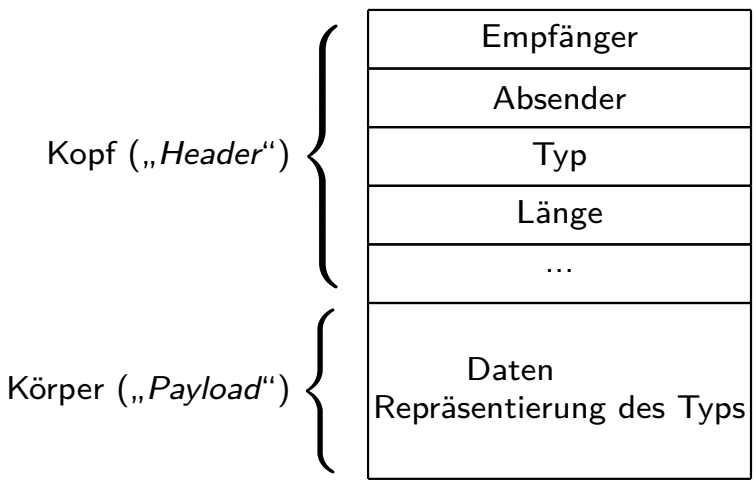
- Nachrichten sind für Sender und Empfänger identifizierbare Einheiten fester oder variabler Länge.
  Nachrichtengrenzen bleiben erhalten.
- Die korrekte Interpretation der internen Struktur einer Nachricht obliegt den Kommunikationspartnern.
- Beispiel: UNIX message queues

Nachrichtenstruktur

Hochschule RheinMain

Typisierte Nachrichten

- Nachrichten haben eine typisierte Struktur.
- Typ ist Sender und Empfänger und z.T. dem Kommunikationssystem bekannt und wird in Operationen verwendet.
- Beispielhafter Aufbau einer Nachricht:



- Nachrichtenkörper kann typisierte Objekte (im Sinne der Objektorientierung) enthalten.

Nachrichtenserialisierung

Beispiel

- Java object serialization überführt Objekt in Bytestrom und zurück (deserialization)
- Header enthält Information über Typ, Layout usw., Körper die Daten
- Java Klasse implementiert Interface java.io.Serializable
- Alle Attribute der Klasse müssen selbst `serializable` sein oder als `transient` markiert sein.
- Operationen `writeObject()`, `readObject()`

Semantik von Nachrichten (2)

Nachrichten mit Dokumentcharakter

- Beispiel: HTML über HTTP
- XML-Dokumente
  - heute stark favorisiert
  - Typbeschreibung durch Schema
- Beispiel: SOAP (*Simple Object Access Protocol*)

```
1  <soap-env:Envelope
2  xmlns:soap-env="http://schemas.xmlsoap.org/soap/envelope/"
3  soap-env:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
4  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
6      <soap-env:Body>
7          <tns:getFlaeche xmlns:tns="urn:tns:beispiel">
8              <tns:seite1 xsi:type="xsd:double">8.0</tns:seite1>
9              <tns:seite2 xsi:type="xsd:double">4.0</tns:seite2>
10         </tns:getFlaeche>
11     </soap-env:Body>
12 </soap-env:Envelope>
```

Internet-Protokolle (Wiederholung 2.Sem)

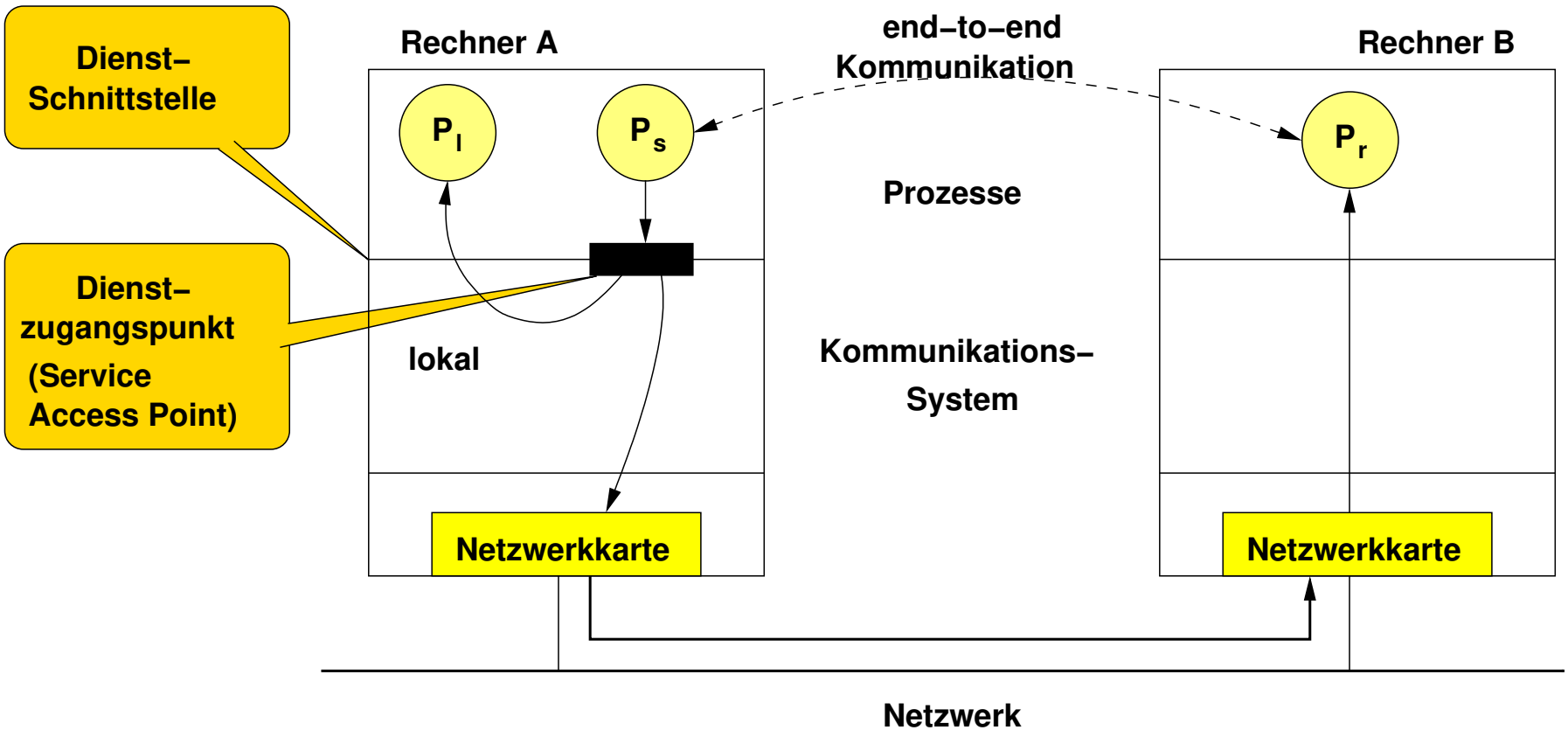
Typische Eigenschaften eines Kommunikationsdienstes

- End-to-End-Kommunikation zwischen Prozessen
- Quality-of-Service (QoS)-Parameter
  - Merkmale bzgl. der Güte der Diensterbringung, z.B.
    - Leistungseigenschaften (Antwortzeit, Durchsatz)
    - Echtzeiteigenschaften
    - Fehlereigenschaften
- Transparenz bzgl. Netzwerktopologie und Übertragungsverfahren
- Wichtige interne Aspekte:
  - Fehlerkontrolle
  - Flusskontrolle
  - Routing (Wegfindung)

Kommunikationsmodell

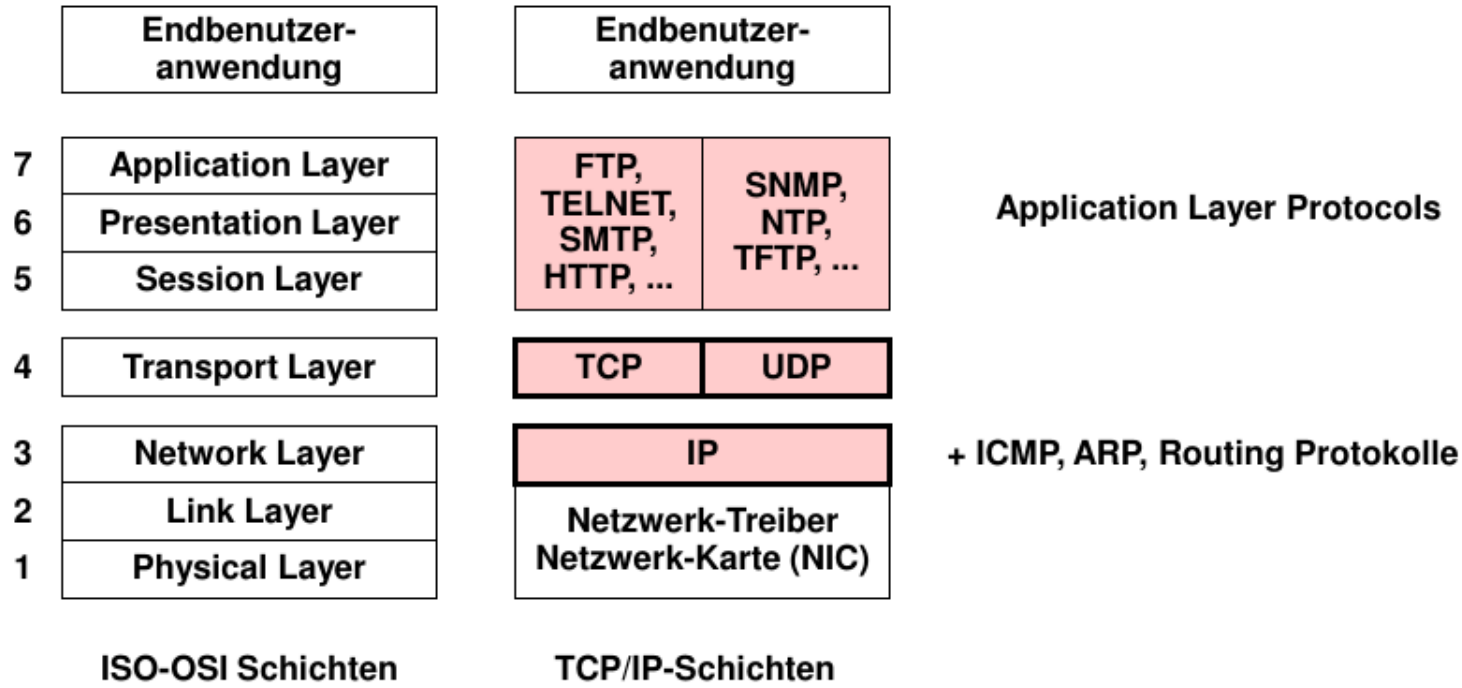
Hochschule RheinMain

Allgemeines Modell der systemweiten Interprozesskommunikation:



Einordnung der Internet-Protokolle

Hochschule RheinMain



IP:	Internet Protocol
TCP:	Transmission Control Protocol
UDP:	User Datagram Protocol
FTP:	File Transfer Protocol
TELNET:	Remote Terminal Protocol
SMTP:	Simple Mail Transfer Protocol
HTTP:	HyperText Transfer Protocol
SNMP:	Simple Network Management Prot.
NTP:	Network Time Protocol
TFTP:	Trivial File Transfer Protocol

Hauptmerkmale der TCP/IP-Architektur

Hochschule RheinMain

- Verbindungsloses Protokoll (IP) auf der Netzwerkebene
- Netzknoten zur Paketvermittlung
- Statisches und dynamisches Routing
- Transportprotokoll (TCP) mit Ende-zu-Ende-Sicherungsfunktionen

Zusammenfassung IP

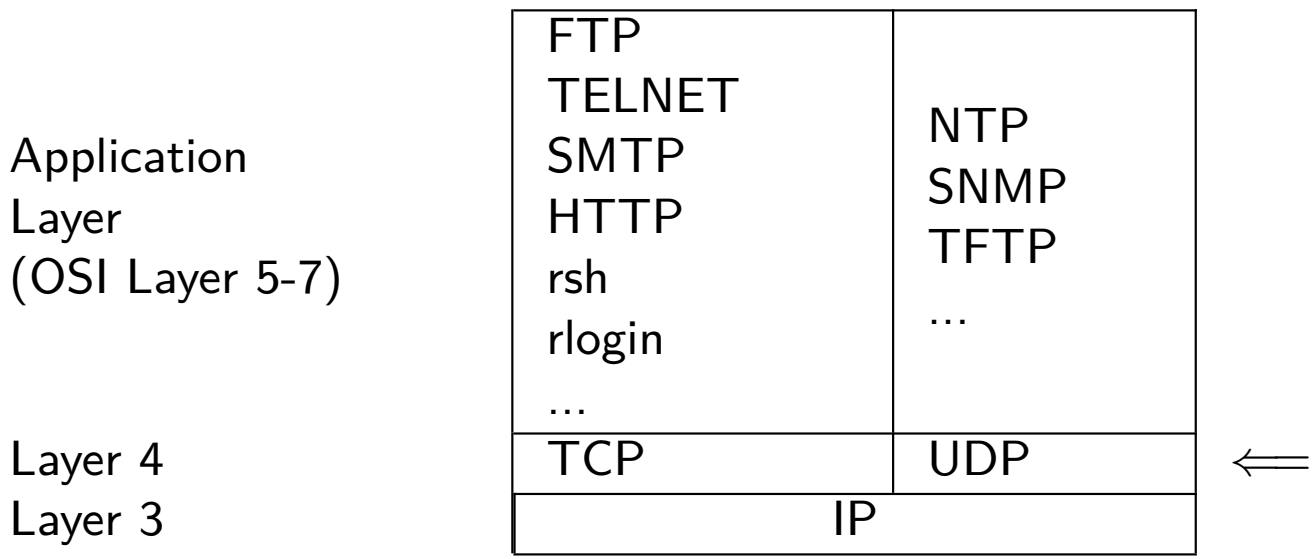
Wesentliche Eigenschaften des Internet-Protokolls (IPv4):

- verbindungsloses Protokoll
- „best-effort“-Beförderung von Einzelnachrichten (Datagram, =Paket)
- Adressierung von Hosts durch 32-bit Internet-Adressen
- Fragmentierung von Paketen bei Bedarf
- Prüfsumme nur für den Kopfteil, nicht über den Dateninhalt
- nicht oft benutzte Protokollfelder sind optional enthalten
- Lebensdauer eines Pakets im Netz begrenzt
- neue Version IPv6

TCP, UDP und die Transportebene

Hochschule RheinMain

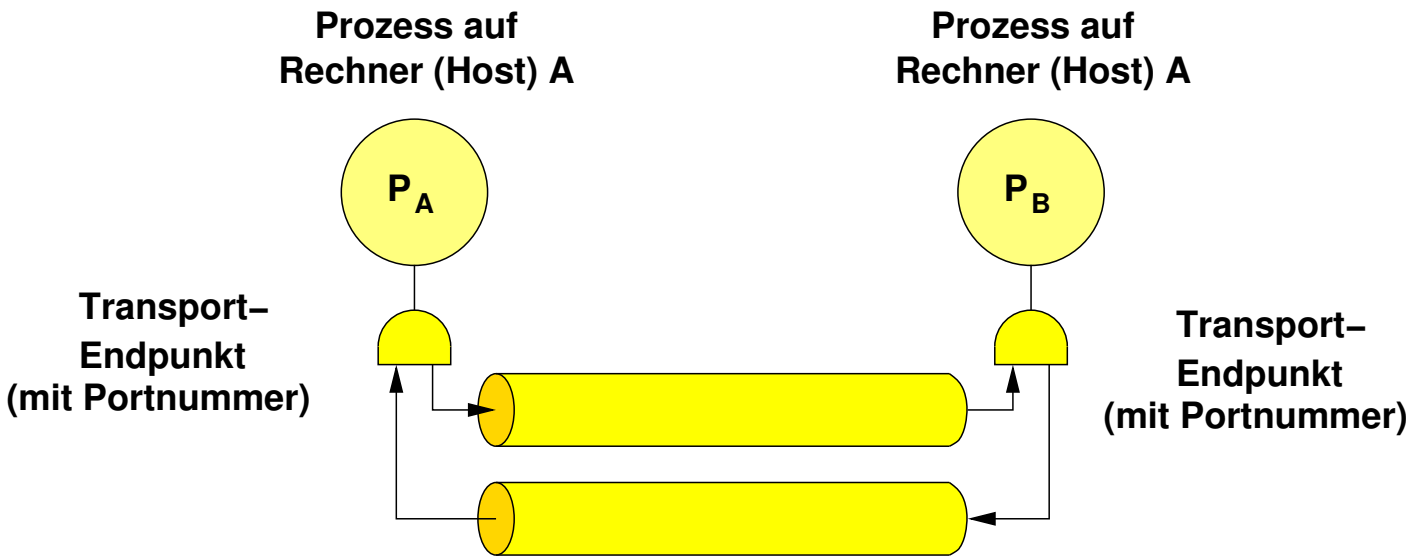
- Aufgabe des Transmission Control Protocols (TCP):
  - zuverlässiger bidirektionaler Punkt-zu-Punkt-Transport eines Bytestroms zwischen Prozessen in Endsystemen
- Aufgabe des User Datagram Protocols (UDP):
  - Best-effort Datagram-Dienst der IP-Netzwerkebene wird Prozessen in Endsystemen zugänglich gemacht
- Einordnung:



TCP-Kommunikationsmodell

Hochschule RheinMain

- verbindungsorientiert
- virtuelle, bidirektionale, voll-duplex-fähige Verbindung zwischen Transport-Endpunkten, die von Prozessen genutzt werden
- Adressierung der Transportdienst-Endpunkte durch 16-bit-Portnummern (in Ergänzung der IP-Adresse des Knotens)
- Bytestrom-orientiert, nicht blockweise. Ein in einem Paket versendeter Abschnitt des Bytestroms wird Segment genannt



Zusammenfassung TCP

Weitere Eigenschaften des TCP-Protokolls:

- Sicherungsfunktion der Übertragung durch
  - Sequenznummern
  - Prüfsummenbildung (Algorithmus wie IP)
  - Empfangsquittungen und Timeouts
  - Wiederholung nach Timeout
- Sliding Window-Prinzip zur Flusskontrolle
- Urgent Data und Push-Funktion für vorrangige Daten

Zusammenfassung UDP

Hochschule RheinMain

Wesentliche Eigenschaften des UDP-Protokolls:

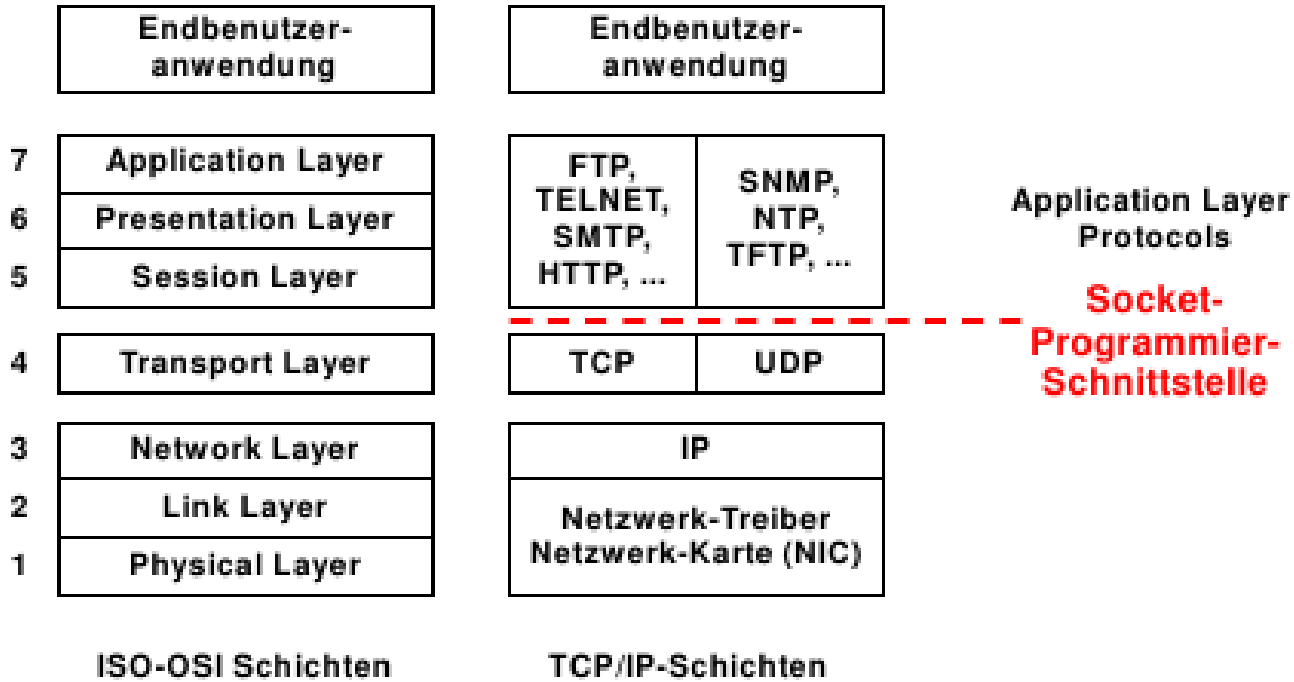
- verbindungsloses Protokoll
- Adressierung der Nutzer durch 16-bit-Portnummern
- „best-effort“-Beförderung von Datagrammen (Einzelnachrichten), i.w. wird der einfache IP-Dienst der Netzwerkebene Anwendungsprozessen zugänglich gemacht (User Datagrams im Unterschied zu IP Datagrams, 1:1-Zuordnung)
- Multicast / Broadcast-fähig (1:n-Kommunikation), direkte Umsetzung bei Multicast-fähigen Netzen wie z.B. Ethernet
- Integritätsprüfung durch Prüfsumme (reale Durchführung kann von der Implementierung abhängen)
- keinerlei Quittungen oder Sicherungsmechanismen, d.h. Datagramme desselben Senders können verloren gehen, in anderer Reihenfolge oder doppelt ankommen
- keine Flusskontrolle, d.h. auf Empfängerseite aufgrund von Puffermangel nicht annehmbare Datagramme werden vom Protokollmodul ohne Mitteilung an den Empfänger verworfen
- gut geeignet zur Implementierung einfacher Auftrag/Antwort-Protokolle

Grundlagen der Socket-Programmierung

Netzwerk-Programmierung mittels Sockets:

- Ziel: Nachrichtenorientierte Interprozesskommunikation zwischen Anwendungsteilen auf verschiedenen Rechnern
- Eingeführt in 4.x BSD UNIX mit C API, heute in nahezu allen Betriebssystemen bereitgestellt (z.B Windows NT, BSe für eingebettete Systeme)
- Heute immer noch die am weitesten verbreitete Schnittstelle für die Entwicklung von Netzwerk-Anwendungen auf der Basis von TCP/IP-Protokollen
- Allen höheren Anwendungsprotokollen (z.B. HTTP) unterlagert
- Sockets als Kommunikationsendpunkte
- Unterstützt Client/Server-Beziehung zwischen Programm-Komponenten:
  - Server: Diensterbringer
  - Client: Dienstnutzer, Kunde, Klient
- Java Sockets bilden BSD Sockets in Menge von Klassen nach

Einordnung

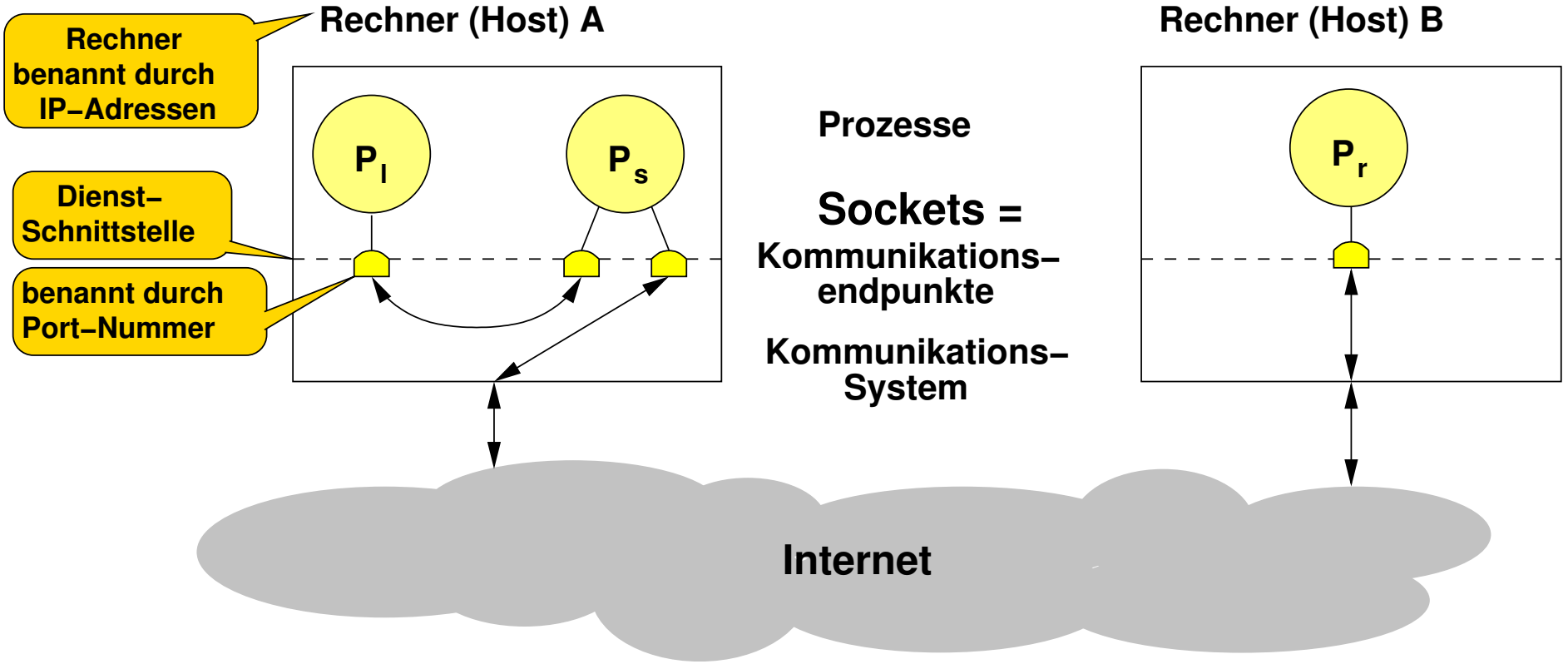


IP:	Internet Protocol	SMTP:	Simple Mail Transfer Protocol
TCP:	Transmission Control Protocol	HTTP:	HyperText Transfer Protocol
UDP:	User Datagram Protocol	SNMP:	Simple Network Management Prot.
FTP:	File Transfer Protocol	NTP:	Network Time Protocol
TELNET:	Remote Terminal Protocol	TFTP:	Trivial File Transfer Protocol

Kommunikationsmodell

Hochschule RheinMain

Socket-basiertes Modell (als Verfeinerung s.o.):



Globaler Kommunikationsendpunkt benannt durch: (Host IP-Adresse, Portnummer)

Internet-Adressen (IPv4)

- Die Repräsentierung einer IP-Adresse geschieht in einem 32-bit-Wort
- In der dezimalen Schreibweise für IP-Adressen (dotted decimal) werden die Inhalte der Bytes einer Adresse in absteigender Wertigkeit, durch Punkte getrennt, als Zeichenkette angegeben:
- Beispiele:
  - `193.175.36.224`
  - `127.0.0.1` (loopback-Netz von localhost)
- Subnetze: CIDR (Classless Inter-Domain Routing): Feste Zuordnung einer IP-Adresse zu Netzklasse (A,B,C) entfällt. Suffix gibt Anzahl der 1-Bits der Netzmaske an. Bsp: /20 mit Netzmaske `255.255.240.0`
- Bzgl. IPv6 und der leicht modifizierten Netzwerkprogrammierung sei z.B. verwiesen auf „Beej's Guide to Network Programming“ `http://beej.us/guide/bgnet/`

Byte Ordering

Hochschule RheinMain

Für die Datendarstellung von Unsigned Integers im Speicher werden unterschieden:

- big-endian Rechner (höherwertige Stelle in Byte mit niederer Adresse)
  Beispiele: Sun SPARC, Motorola, IBM Mainframe
- little-endian Rechner (umgekehrt)
  Beispiele: Intel x86, DEC VAX

Beispiel: IP-Adresse 193.175.36.224

	0	1	2	3	
So ...	193	175	36	224	„Big endian"
..oder so	224	36	175	193	„Little endian"

Netz-Datendarstellung der TCP/IP-Protokoll-Familie ist big-endian

Port-Nummern

Regeln für die Benutzung von Port-Nummern:

- Portnummern werden von der IANA (Internet Assigned Numbers Authority) vergeben.
- Die Verwaltungsprozedur ist beschrieben in RFC 6335
  - 0–1023:
    reservierte System-Ports, well-known Port-Nummern der Standarddienste
  - 1024–49151:
    registrierte oder User-Ports, können von der IANA zugewiesen werden
  - 49152–65535:
    private oder Ephemeral Port-Nummern von benutzerdefinierten Diensten

Beispiele:

- 22: SSH
- 25: SMTP
- 53: DNS
- 80: HTTP

Zuordnungen werden in Datei `/etc/services` angegeben

Netzwerk-Verbindungen

Viele gleichzeitige Verbindungen zwischen Rechnern möglich.
In TCP/IP eindeutig identifizierbar durch:

- Transportschicht-Protokol
- IP-Adresse des lokalen Hosts
- lokale Portnummer
- IP-Adresse des entfernten Hosts
- entfernte Portnummer

Arten von Sockets

Stream Sockets: (`SOCK_STREAM`)

- Verlässliche Kommunikation (i.d.R. eines Byte-Stroms) zwischen zwei Endpunkten
- Verbindungsorientierter Transportdienst
- Im Falle von Internet-domain sockets ist TCP das benutzte Default-Protokoll

Datagram Sockets: (`SOCK_DGRAM`)

- Unzuverlässige Kommunikation von Einzelnachrichten (best-effort delivery)
- Verbindungsloser Datagram-Dienst
- Im Falle von Internet-domain sockets ist UDP das benutzte Default-Protokoll

Raw Sockets: (`SOCK_RAW`)

- erlauben Zugriff auf unterlagerte Protokolle, wie IP, ICMP, ...
- hier nicht weiter betrachtet

BSD Sockets

Socket-Programmierung:

- Eingeführt in 4.x BSD UNIX
- API realisiert durch BSD System Calls
- Einbettung in UNIX I/O-Funktionen
- TCP/IP-Protokolle als Default, aber Nutzung anderer Protokolle prinzipiell möglich
- Schnittstelle auch für die Kommunikation von Prozessen auf *einem* UNIX-Rechner nutzbar (UNIX domain sockets)

Socket-Adressen Datentypen

- Include-Dateien:

```
#include <sys/types.h>
#include <sys/socket.h>
```

- Internet-Adresse:

```
struct in_addr { uint32_t s_addr; };
```

- Socket-Adresse (allg. Typ, in System Calls benutzt):

```
struct sockaddr {
   u_short sa_family;   /* hier AF_xxxx */
   char    sa_data[14]; /* bis 14B typ-spez.Adresse */
};
```

- Socket-Adresse (Internet-Typ):

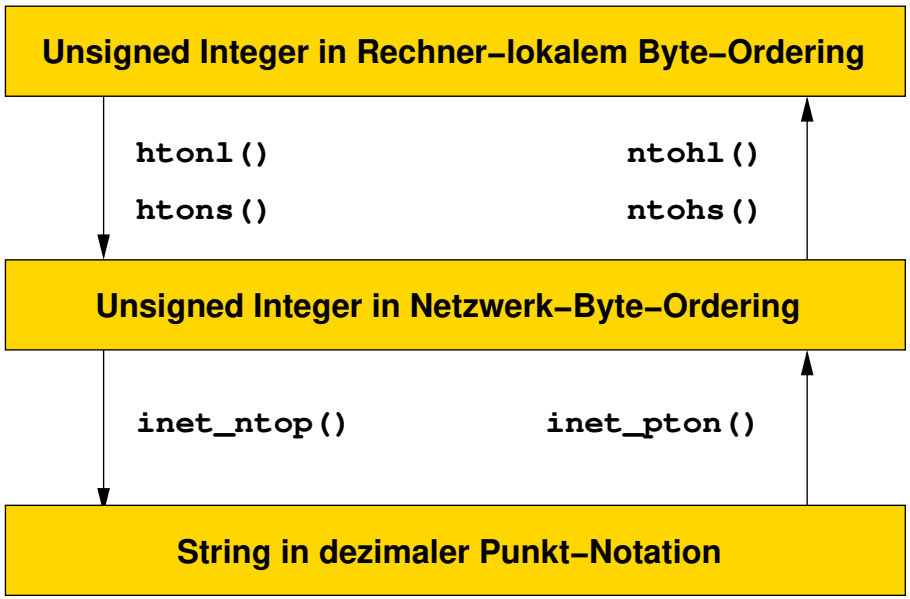
```
struct sockaddr_in {
   u_short sin_family;       /* hier AF_INET, o. AF_UNIX */
   u_short sin_port;         /* Port-Numer in */
                             /* network byte order */
   struct in_addr sin_addr; /* IP-Adresse in */
                             /* network byte order */
   char sin_zero[8];         /* unbenutzt */
};
```

- Cast:

```
struct sockaddr_in my_addr;
... (struct sockaddr*) &my_addr ...
```

Hilfsfunktionen: Adress-Konvertierung

Hochschule RheinMain



.

Funktionen definiert in
`<sys/types.h>`
`<netinet/in.h>`

Funktionen definiert in
`<sys/types.h>`
`<netinet/in.h>`
`<arpa/inet.h>`

`htonl()/htons()`:	host to network long/short
`ntohl()/ntohs()`:	network to host long/short
`inet_ntop()`:	network to presentation/printable
`inet_pton()`:	presentation/printable to network

Hilfsfunktion: Adress-Übersetzung

getaddrinfo()

```
struct addrinfo {
  int              ai_flags;     // AI_PASSIVE, AI_CANONNAME, etc.
  int              ai_family;    // AF_INET, AF_INET6, AF_UNSPEC
  int              ai_socktype;  // SOCK_STREAM, SOCK_DGRAM
  int              ai_protocol;  // use 0 for "any"
  size_t           ai_addrlen;   // size of ai_addr in bytes
  struct sockaddr *ai_addr;      // struct sockaddr_in or _in6
  char            *ai_canonname;// full canonical hostname
  struct addrinfo *ai_next;      // linked list, next node
};
```

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
int getaddrinfo(
  const char *node,
  const char *service,
  const struct addrinfo *hints,
  struct addrinfo **res);
```

„Given node and service, which identify an Internet host and a service, getaddrinfo() returns one or more addrinfo structures, each of which contains an Internet address that can be specified in a call to bind(2) or connect(2).“

(⇒ ersetzt gethostbyname(), getservbyname())

Beispiel

Hochschule RheinMain

```
int main(int argc, char *argv[])
{
    struct addrinfo hints;
    struct addrinfo *result;
    int s;
...
    memset(&hints, 0, sizeof(struct addrinfo));
    hints.ai_family = AF_UNSPEC;     /* Allow IPv4 or IPv6        */
    hints.ai_socktype = SOCK_DGRAM; /* Datagram socket           */
    hints.ai_flags = AI_PASSIVE;     /* For wildcard IP address */
    hints.ai_protocol = 0;           /* Any protocol              */
    hints.ai_canonname = NULL;
    hints.ai_addr = NULL;
    hints.ai_next = NULL;
    s = getaddrinfo(NULL, argv[1], &hints, &result);
    if (s != 0) {
       fprintf(stderr, "getaddrinfo:␣%s\n", gai_strerror(s));
       exit(EXIT_FAILURE);
    }
```

Weitere Hilfsfunktionen

Hochschule RheinMain

`gethostname()`	Ermitteln des eigenen Host-Namens
`gethostid()`	Ermitteln des eigenen unique Host-IDs
`getsockopt()`	Ermitteln der Parameter eines Sockets
`setsockopt()`	Setzen der Parameter eines Sockets

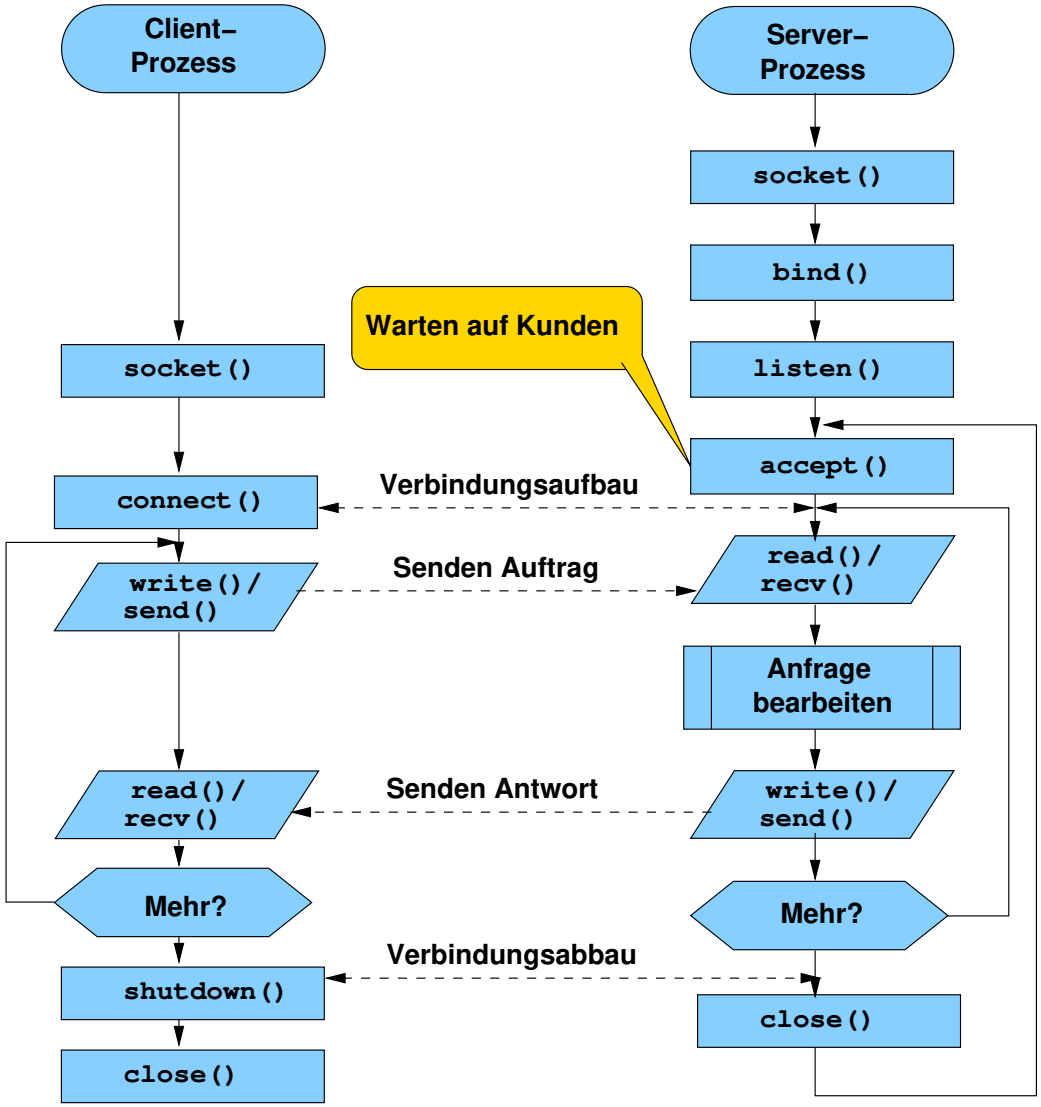
Socket-Funktionen im Überblick

Hochschule RheinMain

`socket()`	Erzeugen
`bind()`	Zuordnung einer Socket-Adresse / eines -Namens
`listen()`	Server: Socket vorbereiten auf Akzeptieren von Klienten
`accept()`	Server: Warten auf Verbindungsanfrage
`connect()`	Client: Verbindung aufbauen
`send()` / `write()`	Senden
`recv()` / `read()`	Empfangen
`shutdown()`	Verbindung schließen
`close()`	Socket zerstören
`sendto()` / `recvfrom()`	Senden/Empfangen UDP
`select()`	Warten auf das Eintreffen eines von mehreren I/O-Ereignissen

Vereinfachtes Zusammenspiel für TCP

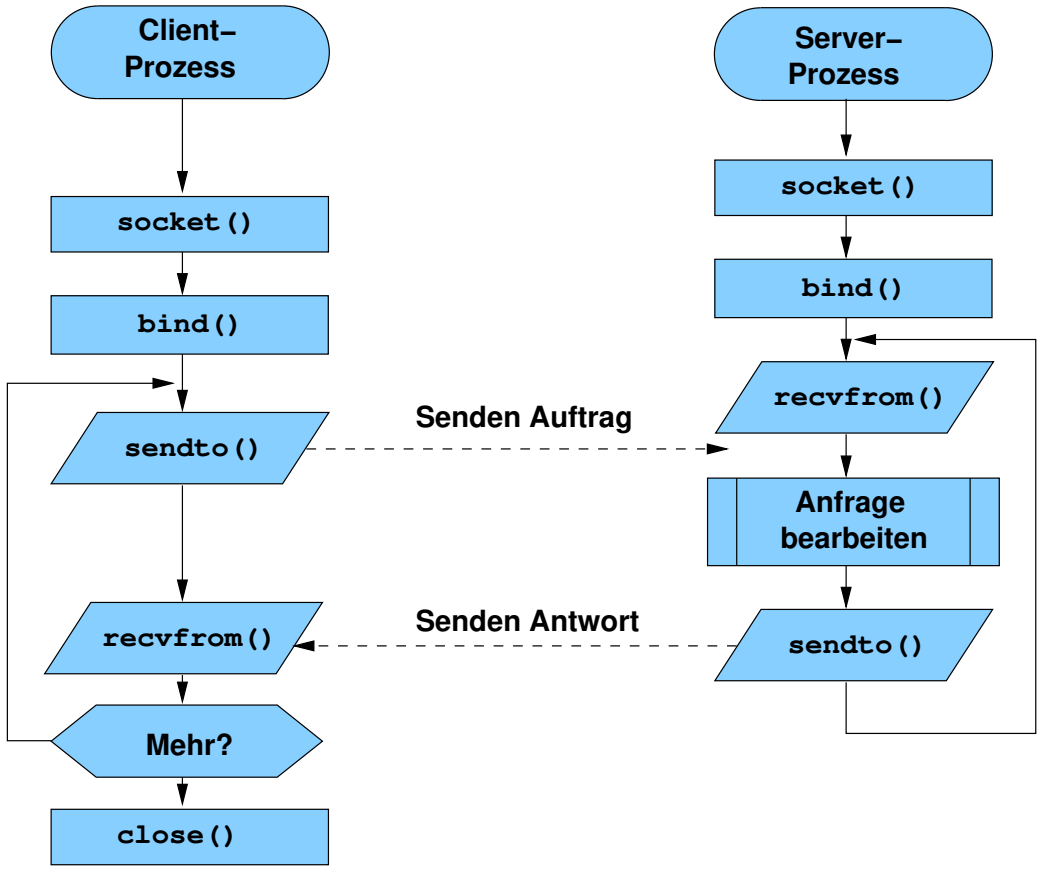
Hochschule RheinMain



Verallgemeinerung:

- Ein Server hält i.a. gleichzeitig mehrere Verbindungen zu Klienten.

Vereinfachtes Zusammenspiel für UDP



socket()

Hochschule RheinMain

Erzeugen eines Sockets

`int socket(int family, int type, int protocol)`

erzeugt einen Socket in der Internet-Domäne (`family=AF_INET`) oder der UNIX-Domäne (`AF_UNIX`) vom Typ Stream-Socket (`type=SOCK_STREAM`), Datagram-Socket (`SOCK_DGRAM`) oder Raw-Socket (`SOCK_RAW`) zur Verwendung mit dem Protokoll protocol und liefert einen Deskriptor für den erzeugten Socket.

Für `protocol` wird i.d.R. der Wert 0 übergeben. Dann wird das Default-Protokoll gewählt, welches in der Internet-Domäne TCP für einen Stream- Socket bzw. UDP für einen Datagram-Socket ist. Es ist noch keine Socket-Adresse zugeordnet, der Socket ist *ungebunden*.

Beispiel:

```
sd1 = socket(AF_INET, SOCK_STREAM, 0)
sd2 = socket(AF_INET, SOCK_DGRAM, 0)
```

bind()

Hochschule RheinMain

Binden einer Socket-Adresse

`int bind(int sd, struct sockaddr *addr, int addrlen)`

bindet den in der `struct sockaddr` übergebenen, von der Domäne des betrachteten Sockets abhängigen Adresse an den Socket. Im Falle der Internet-Domäne entspricht diese Struktur der `struct sockaddr_in`, für einen Socket in der UNIX-Domäne wird im Effekt ein Dateiname übergeben. Der Socket wird im Kommunikationssystem (TCP/IP Protokoll-Modul) registriert.

Für Klienten in verbindungsorientierter Kommunikation nicht notwendig.

Beispiel:

```
rc = bind(sd, (struct sockaddr *) &my_addr, sizeof(my_addr))
```

listen()

Hochschule RheinMain

Socket vorbereiten auf Verbindungsanfragen

`int listen(int sd, int qlength)`

zeigt dem TCP/IP-Modul an, dass TCP-Verbindungen über den Socket `sd` angenommen werden sollen; `qlength` gibt die maximale Anzahl wartender Verbindungswünsche an, für die ein accept aussteht (nicht die Anzahl der insgesamt möglichen Klienten).

Nur für Server-Seite in verbindungsorientierter Kommunikation notwendig.

Beispiel:

```
rc = listen(sd, 5)
```

accept()

Hochschule RheinMain

Warten auf Verbindungsanfragen

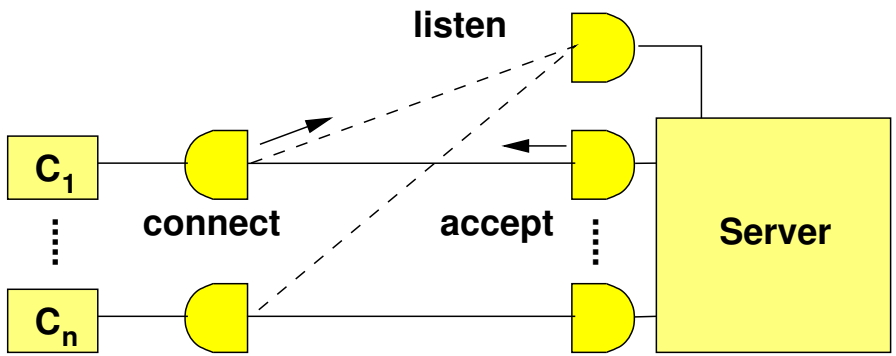
```
int accept(int sd, struct sockaddr *claddr, int *addrlen)
```

blockiert, bis eine Verbindungsanfrage eines Klienten am Socket `sd` vorliegt. Dann wird ein neuer Socket erzeugt, und dessen Deskriptor zurückgegeben. Damit entsteht eine private Verbindung zwischen Klient und Server. Der Socket `sd` steht für weitere Verbindungsanforderungen zur Verfügung. Die Identität des Klienten (entfernte Socket-Adresse) wird in der übergebenen Struktur `claddr` abgelegt und die Längen-Variable `addrlen` entsprechend gesetzt.

Nur für Server-Seite in verbindungsorientierter Kommunikation notwendig.

Beispiel:

```
snew = accept(sd, &clientaddr, &clientaddrlen)
```



connect()

Hochschule RheinMain

Verbindungsanfrage

```
int connect(int sd, struct sockaddr *saddr, int saddrlen)
```

Aktive Verbindungsanfrage eines Klienten über seinen Socket `sd` zu einem Server, dessen Adresse in `saddr` mit der Länge `saddrlen` übergeben wird.

Nur für Client-Seite in verbindungsorientierter Kommunikation notwendig.

Beispiel:

```
rc = connect(sd, &saddr, sizeof(saddr))
```

write()/send() und read()/recv()

Senden

```
int write(int sd, char *buf, int len)
int send(int sd, char *buf, int len, int flag)
```

Der `write`-Aufruf wird wie bei Datei-Deskriptoren verwendet. Der `send`-Aufruf besitzt ein Flag als zusätzliches Argument zur Spezifikation spezieller Optionen.

Empfangen

```
int read(int sd, char *buf, int nbytes)
int recv(int sd, char *buf, int nbytes, int flag)
```

Der `read`-Aufruf wird wie bei Datei-Deskriptoren verwendet. Der `recv`-Aufruf besitzt ein Flag als zusätzliches Argument zur Spezifikation spezieller Optionen.

Beispiele:

```
charcount = write(sd, buf, len)
charcount = send(sd, buf, len, sendflag)
charcount = read(sd, buf, len)
charcount = recv(sd, buf, len, recvflag)
```

shutdown()

Hochschule RheinMain

Schließen einer Verbindung

```
int shutdown(int sd, int how)
```

Geordnetes Schließen einer Verbindung, `how` gibt an, ob sich das TCP/IP-Modul auch nach dem Schließen noch um die Verbindung kümmern soll.

Der Socket-Deskriptor bleibt bestehen und muss, falls gewünscht, mittels `close()` zerstört werden.

Beispiel:

```
rc = shutdown(sd, 2)
```

select()

Warten auf das Eintreffen eines von mehreren I/O-Ereignissen

```
#include <sys/time.h>
int select(int nfds, int *readmask, int *writemask,
int *exceptmask, struct timeval *timeout)
```

bietet die Möglichkeit zum Umgang mit mehreren Socket/File-Deskriptoren in einem Prozess, um so lange zu warten, bis an einem Deskriptor einer vorgebbaren Menge ein Ereignis eintritt (z.B. Socket wird lesbar) oder ein Timeout abgelaufen ist. Die Wartezeit kann zeitlich begrenzt werden oder unbefristet sein.
Die Mengen werden über Bitmasken spezifiziert, wozu z.B. die `FD_SET`- Makros mit Vorteil eingesetzt werden können.
Bei Rückkehr ist `readmask` verändert und enthält die Bitmaske der Deskriptoren, für die Ereignisse eingetreten sind, der Rückgabewert gibt die Anzahl dieser Deskriptoren an.

Beispiel:

```
int sd1, sd2;
fd_set fds;
sd1 = socket(AF_INET,...);
sd2 = socket(AF_INET,...);
...
```

```
FD_ZERO(&fds);
FD_SET(sd1,&fds);
FD_SET(sd2,&fds);
rc=select(FD_SETSIZE,&fds,
NULL,NULL,timeout);
```

Java Sockets

Verkleidung der BSD Sockets hinter mehreren Interfaces/Klassen des Package java.net

Adressierung

- InetAddress mit Subklassen Inet4Address und Inet6Address
- SocketAddress mit Subklasse InetSocketAddress

TCP-Verbindungen

- ServerSocket
- Socket
- Für etablierte Verbindung getInputStream()/getOutputStream()

Datagram-Kommunikation über UDP

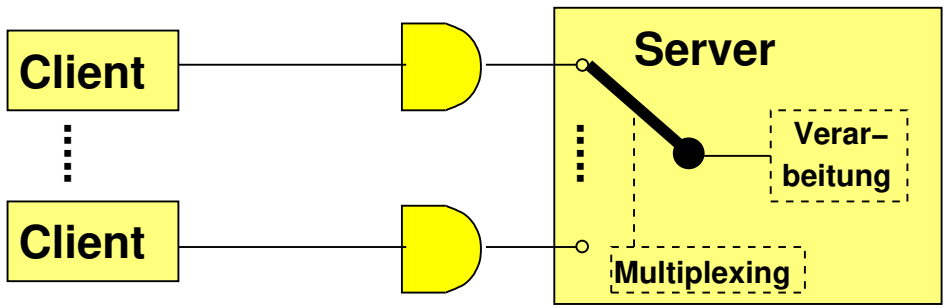
- DatagramPacket
- DatagramSocket – MulticastSocket

Vertiefung im Praktikum

Server-Architektur

Hochschule RheinMain

- **Architektur-Prinzipien für die interne Struktur von Server-Prozessen**
- **Problem:**
  Server muss i.d.R. mit mehreren Klienten kommunizieren können.



Modelle

- Einfacher sequentieller Prozess
- Klassischer sequentieller Server als Zustandsautomat
- Parallele Server-Prozesse
- Multithreaded Server

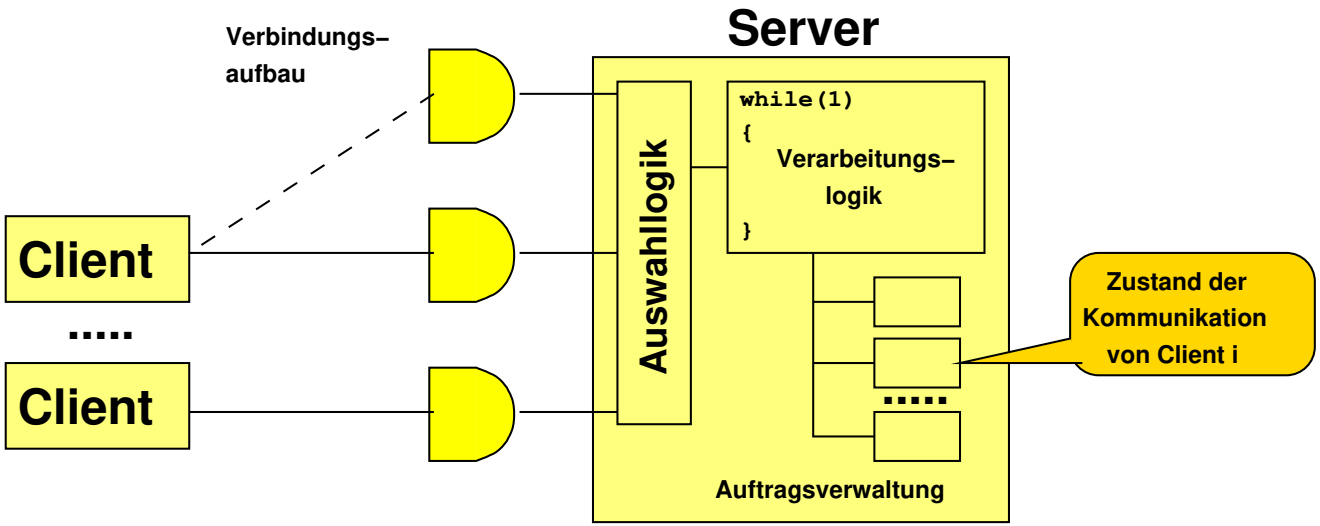
Einfacher sequentieller Prozess

- **Ein** Prozess bearbeitet nacheinander die Anfragen aller Klienten
- Problem, wenn Server während der Bearbeitung selbst zum Client gegenüber einem weiteren Server wird:
  $\Rightarrow$ der gesamte Server ist blockiert!
- Nachteile:
  - keine Nebenläufigkeit im Server
  - keine Nutzung einer unterlagerten Multiprozessor-Architektur durch einen einzelnen Server möglich.
- Ansatz ist in kommerzieller Umgebung selten akzeptabel!

Klassischer sequentieller Server

Hochschule RheinMain

Architektur: Server als Zustandsautomat

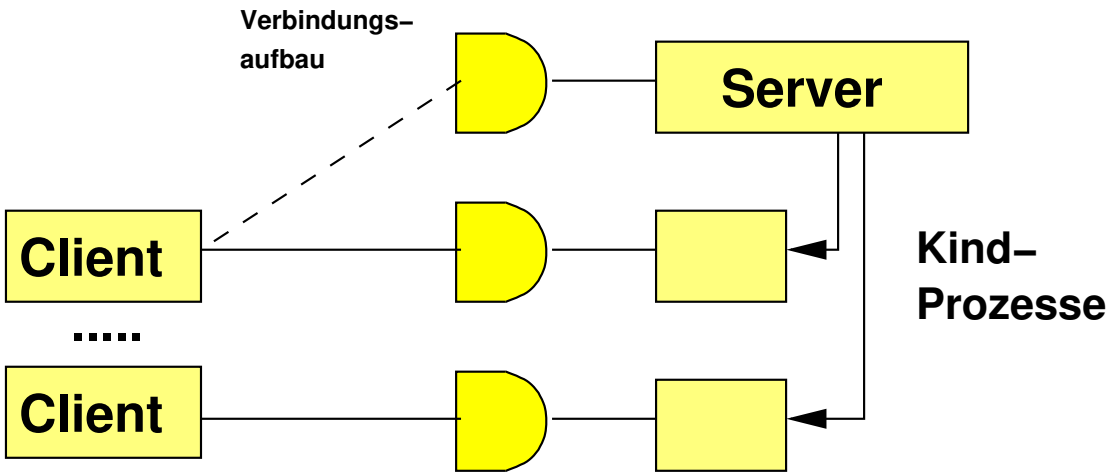


- keine interne Blockierung:
  mehrere Aufträge können überlappend ausgeführt werden
- Multiplexing „von Hand“ $\Rightarrow$ komplex zu programmieren
- Auswahllogik in UNIX:
  - nicht-blockierende Aufrufe (Option `O_NDELAY`) und Polling
  - `select()`

Parallele Server-Prozesse

Hochschule RheinMain

Architektur:



- Kindprozesse enthalten Gedächtnis für den Zustand der Kommunikation mit jeweils einem Client
- Vorteil: Multiprozessor-Architektur kann genutzt werden
- aufwändiges Prozess-Handling aus Performance-Sicht

Multithreaded Server

Hochschule RheinMain

- **Automatisches Lösen des Multiplexing-Problems**
  - jedem Auftrag wird bei Beginn der Verarbeitung ein Thread fest zugeordnet
  - jeder einzelne Thread kann innerhalb des Servers blockieren, aber Nebenläufigkeit bleibt insgesamt erhalten.
  - Thread Pool
- **Anwendbar für alle Paradigmen verteilter Anwendungen (vgl. Kap. 3)**
- **Synchronisation erforderlich**

Multithreaded Server (2)

Hochschule RheinMain

- **Threads sind mittlerweile in allen neueren Betriebssystemen und Runtime-Systemen verfügbar**
- **Implementierung der User-Level Threads durch Kernel Threads oder in Thread Libraries möglich (vgl. LV Betriebsysteme)**
- **Verbreitete Programmierschnittstellen**
  - Pthreads POSIX 1003.4 (C/C++)
  - Boost.Threads (C++)
  - Java Concurrency ab SE 5: java.util.concurrent
- **Wird im Praktikum vertieft**

Zusammenfassung

Hochschule RheinMain

- Netzwerkprogrammierung ist grundlegend für die Umsetzung verteilter Systeme.
- Quasi jedwede Netzwerkprogrammierung basiert letztlich auf dem Konzept der Socket-Schnittstellen.
- BSD-Sockets stellen die ursprüngliche und bis heute wichtigste Socket-API dar.