Struktur von Kapitel II - Kommunikation

I Einleitung

II Grundlegende Kommunikationsdienste

- A Kommunikationsformen
- B Netzwerk Grundlagen
- C Berkeley Sockets
- D Java Sockets
- E Remote Procedure Calls (RPC)
- F XML-RPC
- G Java RMI

III Middleware

IV Architekturen & Algorithmen

- A Synchronisierung
- B Konsistenz und Replication
- C Fehlertoleranz

V Beispiele bzw. Dienste

- A Verteilte Dateisysteme
- B Namensdienste

VI Sicherheit & Sicherheitsdienste

VII Zusammenfassung

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbi
FACHBEREICH INFORMATIK

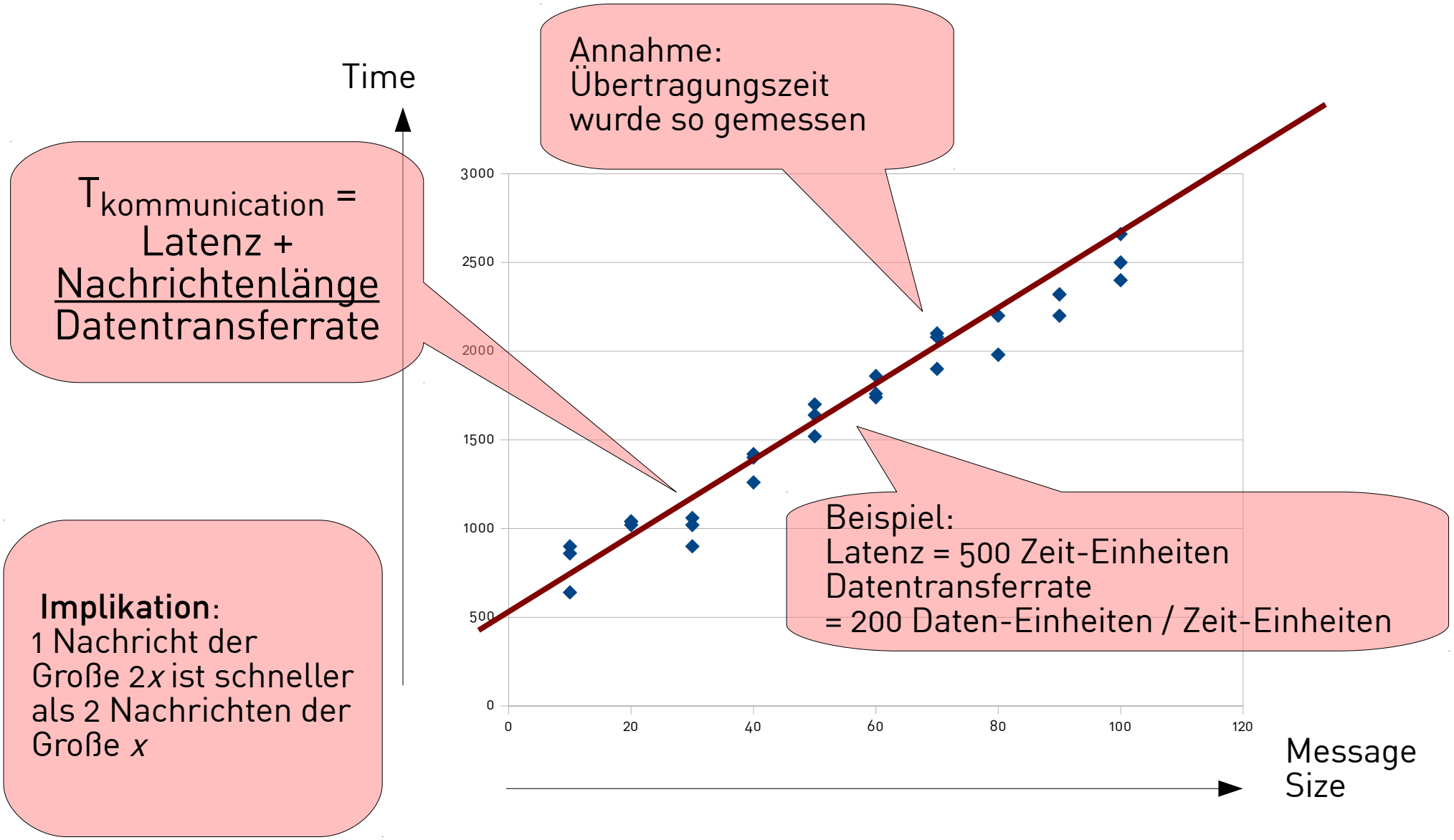
Qualitätsparameter

- **Latenzzeit** [s]
  die **minimale** Zeit, die eine Nachricht bis zum Empfangsprozess braucht,
- **Bandbreite [bit/s]**
  Die physikalische obere Schranke des Durchsatzes (nur Hardware),
- **Datentransferrate** [bit/s]
  die **maximal** Anzahl Bits, die zwischen zwei Prozessen pro Sekunde übertragen werden kann (Software inkl.).
- **Nachrichtentransferzeit** [s]
  = Latenzzeit + $\frac{\text{Nachrichtenlänge}}{\text{Datentransferrate}}$
- **Durchsatz** [bit/s]
  Anzahl übertragener Bits pro Sekunde über eine gewisse zeitliche Dauer – tatsächlich beobachtet.

Also:
Bandbreite ≥ Datentransferrate &
Datentransferrate ≥ Durchsatz

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbi
FACHBEREICH INFORMATIK

Latenz - nochmals



h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbi
FACHBEREICH INFORMATIK

Qualitätsparameter (2)

- **Latenzzeit** [s]
  die **minimale** Zeit, die eine Nachricht bis zum Empfangsprozess braucht.
- **Bandbreite [bit/s]**
  Die physikalische obere Schranke des Durchsatzes.
- **Datentransferrate** [bit/s]
  die **maximal** Anzahl Bits, die zwischen zwei Prozessen pro Sekunde übertragen werden kann
- **Nachrichtentransferzeit** [s]
  = Latenzzeit + $\frac{\text{Nachrichtenlänge}}{\text{Datentransferrate}}$
- **Durchsatz** [bit/s]
  Anzahl übertragener Bits pro Sekunde über eine gewisse zeitliche Dauer – tatsächlich beobachtet.

  Bandbreite ≥ Datentransferrate und Datentransferrate ≥ Durchsatz

- **Fehlerkontrolle**
  Kann das Netz einen Fehler erkennen? Die Fehler durch redundante Datenübertragung möglicherweise beheben?
  = Fehlererkennung & Fehlerkorrektur.
- **Routing**
  Bei verschiedenen möglichen Wegen von Prozess A zu Prozess B entscheidet die Wegeführung (engl.: Routing) über den Zustellweg.

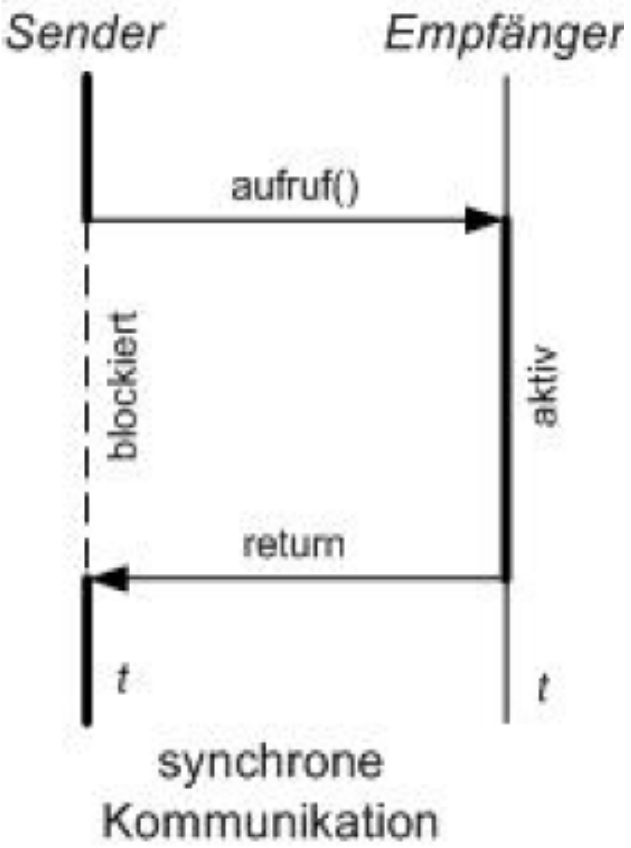
  Manchmal unterstützt das Routing mehrere Empfänger:

  - **Broadcasting** (an alle) bzw.
  - **Multicasting** (an manche)

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbi
FACHBEREICH INFORMATIK

Synchrone Kommunikation

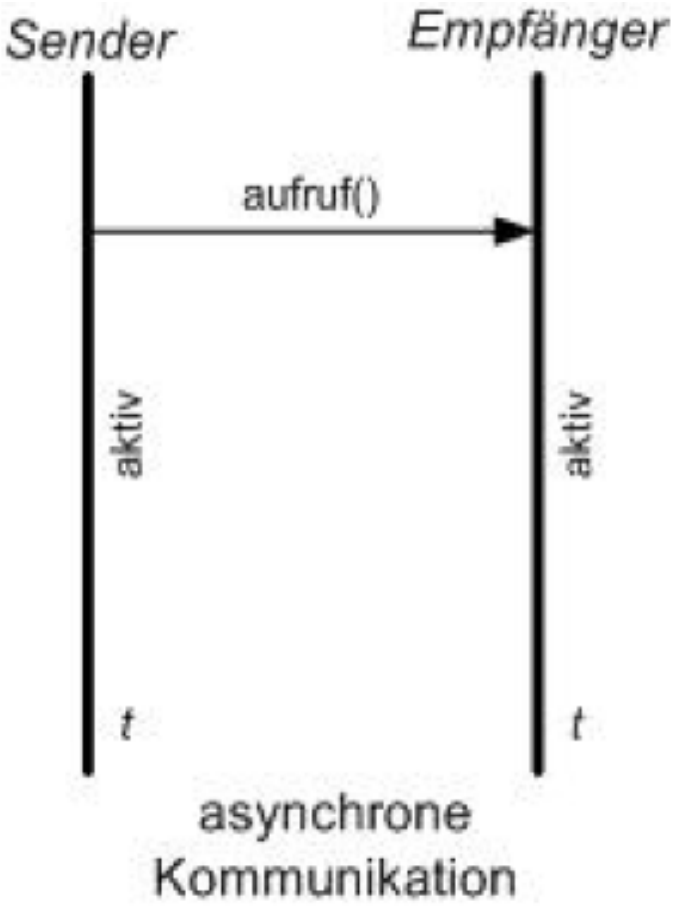
- Sender und Empfänger **blockieren** beim Ausführen der ‚Senden' bzw. ‚Empfangen'.
- Eigenschaften:
  - Enge Kopplung zwischen Sender und Empfänger mit allen Vor- und Nachteilen.
  - Hohe Abhängigkeit insbesondere im Fehlerfall.
- Voraussetzung:
  - sichere und schnelle Netzverbindungen sind verfügbar.
  - empfangender Prozess ist verfügbar.



synchrone Kommunikation

Asynchrone Kommunikation

- Sender wird nicht blockiert; Prozess kann parallel zur Übertragung der Nachricht mit der Ausführung fortfahren.
- Antworten sind optional:
  - Der Sender erhält bei Gelegenheit das Ergebnis asynchron
  - Der Sender holt sich bei Gelegenheit aktiv das Ergebnis
- Pufferung auf der Sende- und Empfangsseite.
- Eigenschaften:
  - Lose Koppelung von Prozessen.
  - Geringere Fehlerabhängigkeit.
  - Empfänger muss nicht empfangsbereit sein.



Verbindungs(lose) Kommunikation

- Für eine **verbindungsorientierte** Kommunikation wird eine Sitzung aufgebaut (*Hand-Shaking).*
- Diese wird auch explizit beendet.
- Die ausgetauschten Daten werden von den Anwendungen als kontinuierlicher Strom wahrgenommen.
- Ein Anwendungsprotokoll beschreibt die Strukturierung des Datenstroms zwischen Client und Server.
- Keine Multicasting / Broadcast

- Eine **verbindungslose** Kommunikation wird durch das Senden eines Datenpakets (Datagram) beendet.
- Es ist daher kein Verfahren zum Verbindungsabbau notwendig.
- Keine Beziehung zwischen zwei Datenpackete darf angenommen werden – inkl. Reihenfolge.
- Multicasting / Broadcast möglich

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbi
FACHBEREICH INFORMATIK

Struktur von Kapitel II - Kommunikation

I Einleitung

II Grundlegende Kommunikationsdienste

- A Kommunikationsformen
- B Netzwerk Grundlagen
- C Berkeley Sockets
- D Java Sockets
- E Remote Procedure Calls (RPC)
- F XML-RPC
- G Java RMI

III Middleware

IV Architekturen & Algorithmen

- A Synchronisierung
- B Konsistenz und Replication
- C Fehlertoleranz

V Beispiele bzw. Dienste

- A Verteilte Dateisysteme
- B Namensdienste

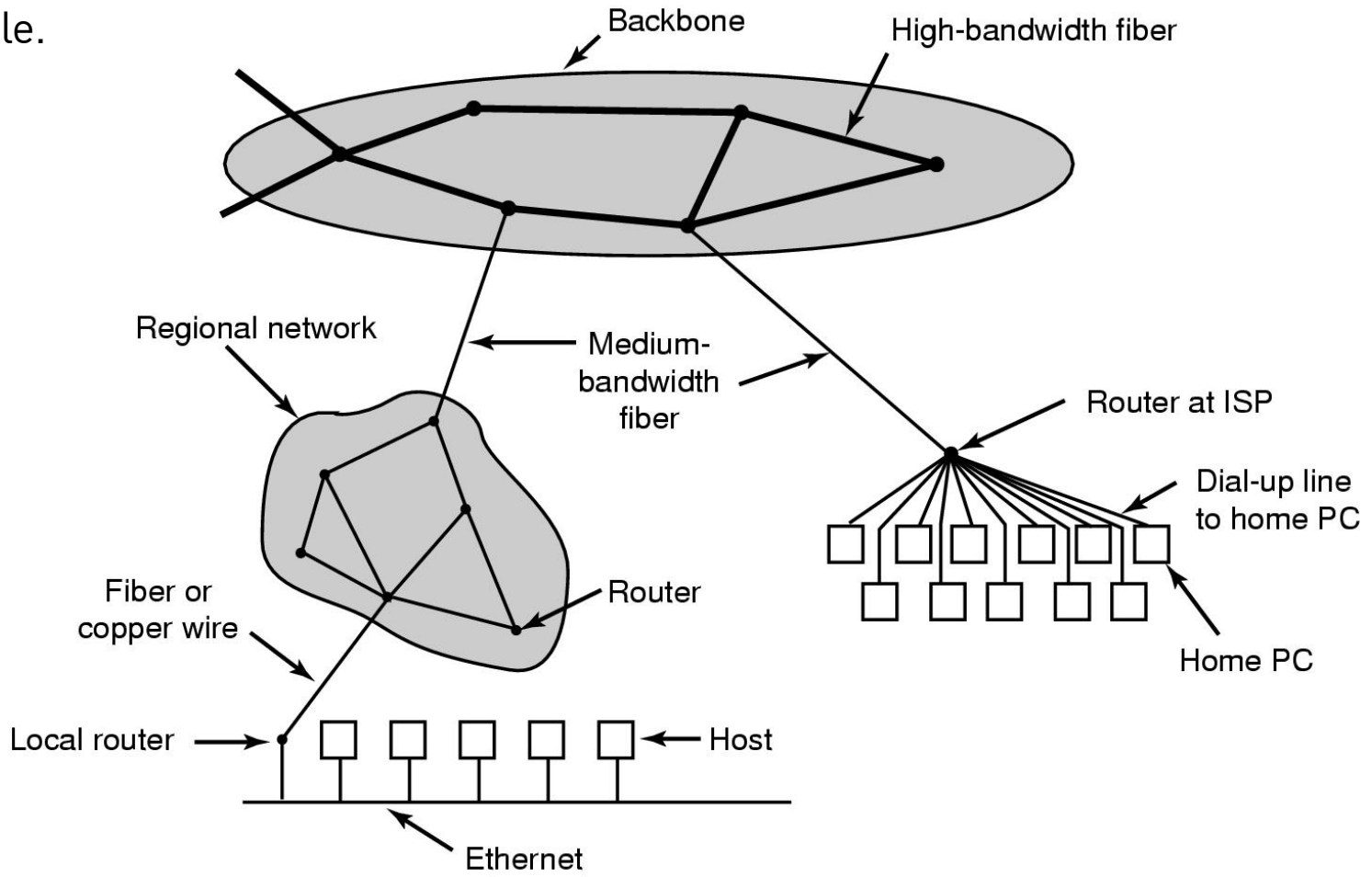
VI Sicherheit & Sicherheitsdienste

VII Zusammenfassung

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbi
FACHBEREICH INFORMATIK

Das Internet = Heterogenität

Das Internet besteht aus heterogene lokale Netzwerke, zusammen- gehalten mit Router und Netzwerkprotokolle.



Das Internet = Protokolle

- **Protokolle:** bieten vershiedene QoS (Quality of Service) an.
- **Protocol-Stacks:** Jedes Protokoll baut auf einem anderen : Z.B. TCP auf IP...

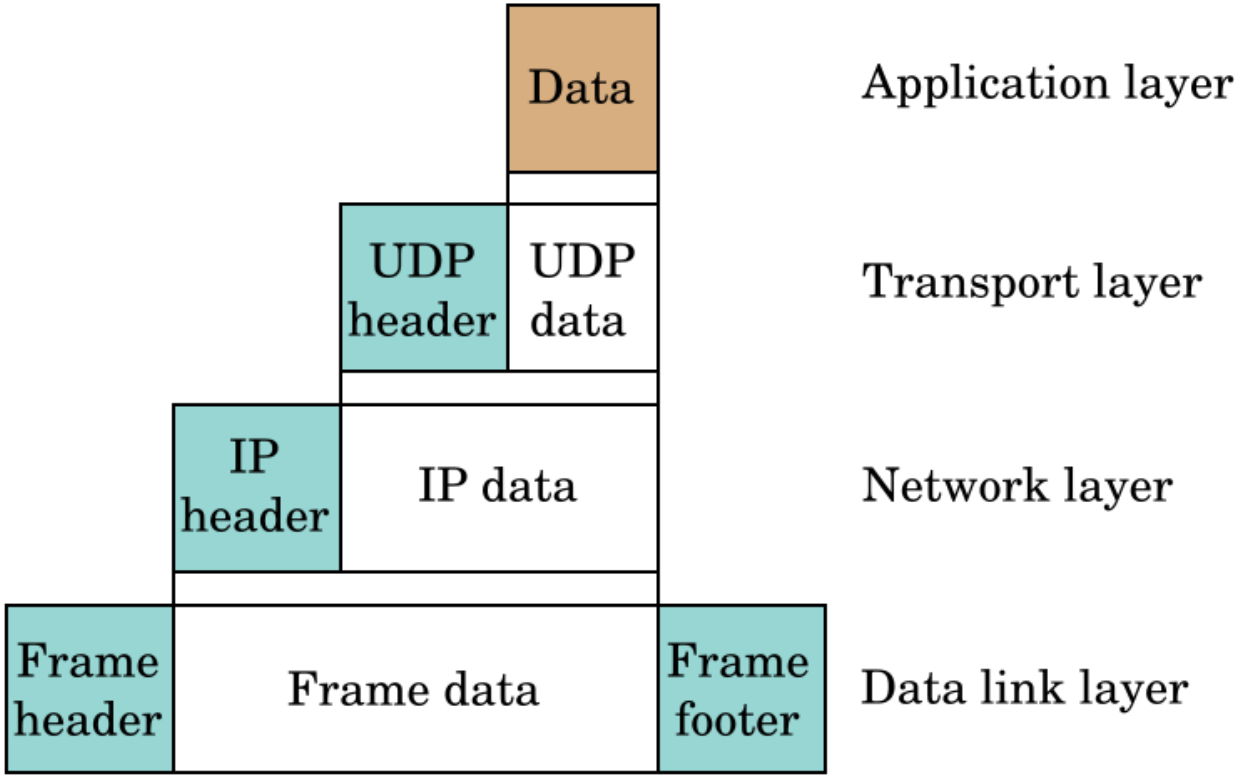
	OSI-Schicht	TCP/IP	Verbindung	Beispiel	Einheiten
7	Anwendung (Application)	Anwendung	Ende zu Ende (Multihop)	HTTP FTP HTTPS SMTP LDAP NCP	Daten
6	Darstellung (Presentation)				
5	Sitzung (Session)				
4	Transport (Transport)	Transport		TCP UDP SCTP SPX	Segmente
3	Vermittlung (Network)	Vermittlung	Punkt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

Diese Vorlesung behandelt die **Anwendungsschicht** (OSI 5-7)

...und daher auch den Wahl eines **Transportschicht**-Protokolls: Verbindungsorient oder nicht?

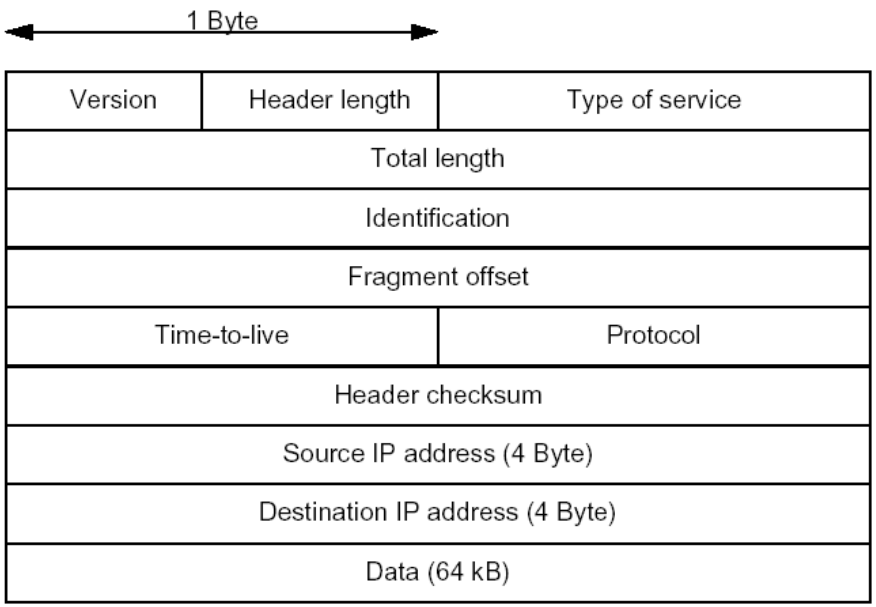
Das Internet = Packets & Headers

- **Protokolle:** bieten vershiedene QoS (Quality of Service) an.
- **Protocol-Stacks:** Jedes Protokoll baut auf einem anderen : Z.B. TCP auf IP...
- **Resultat:** Anhäufung der Packet-Header



IP-Funktionalität

- IP überträgt Nachrichten (Datagramme) verbindungslos
- Es ist eine Host-zu-Host Übertragung
- „Best Effort“, d.h. unzuverlässige Zustellung
- Probleme: Paketverluste, Duplikate, vertauschte Reihenfolge
  - **Routing der Pakete**
    - Address Resolution Protocol, ARP, zur Abbildung von IPAdressen auf Ethernetadressen.
    - Im lokalen Netzsegment reicht diese Information, um Zielrechner direkt zu adressieren.
    - Liegt das Ziel außerhalb, wird die Adresse eines Routers zurückgeliefert.



IPv4 Datagrammformat (vgl. IPv6)

UDP-Funktionalität

- ein verbindungsloser Datagrammdienst,
- Es ist eine Port-zu-Port Übertragung
- nur unzuverlässig Verbindung möglich,
- keine Garantie auf die Einhaltung der Reihenfolge der Datagramme
- keine Garantie auf die Vermeidung doppelter Nachrichtenzustellung
- Nachrichtengrenzen werden immer erhalten d.h. eine korrekt empfangene und gelesene Nachricht entspricht exakt der gesendeten
- Multicastadressen und Broadcastadressen sind möglich

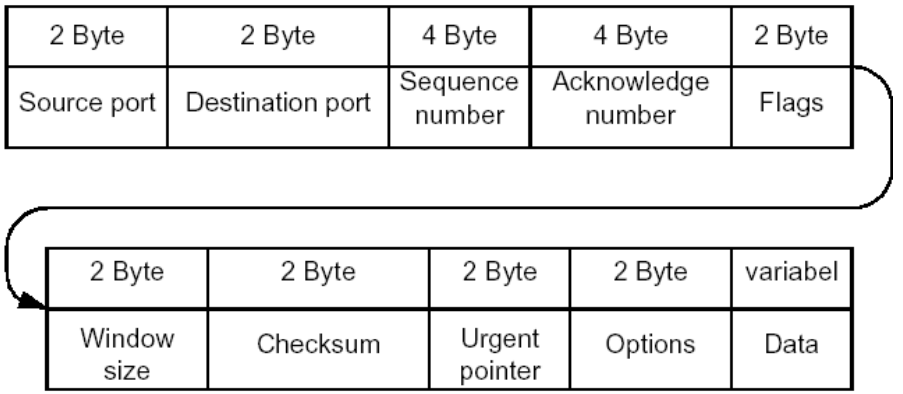
2 Byte	2 Byte	2 Byte	2 Byte	variabel
Source port	Destination port	Length	Checksum	Data

UDP Datagrammformat

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbi
FACHBEREICH INFORMATIK

TCP-Funktionalität

- **verbindungsorientierter** Transportdienst
- Zuverlässigkeit ist garantiert
- Nachrichten werden fehlerfrei übergeben
- keine Nachrichtenverluste oder Duplikate
- selbständige Fehlerkorrektur
- verlorengegangene Pakete werden erneut geschickt, Duplikate verworfen
- Multicast und Broadcast sind **nicht** möglich



TCP Paketformat

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbi
FACHBEREICH INFORMATIK

Struktur von Kapitel II - Kommunikation

I Einleitung

II Grundlegende Kommunikationsdienste

- A Kommunikationsformen
- B Netzwerk Grundlagen
- C Berkeley Sockets
- D Java Sockets
- E Remote Procedure Calls (RPC)
- F XML-RPC
- G Java RMI

III Middleware

IV Architekturen & Algorithmen

- A Synchronisierung
- B Konsistenz und Replication
- C Fehlertoleranz

V Beispiele bzw. Dienste

- A Verteilte Dateisysteme
- B Namensdienste

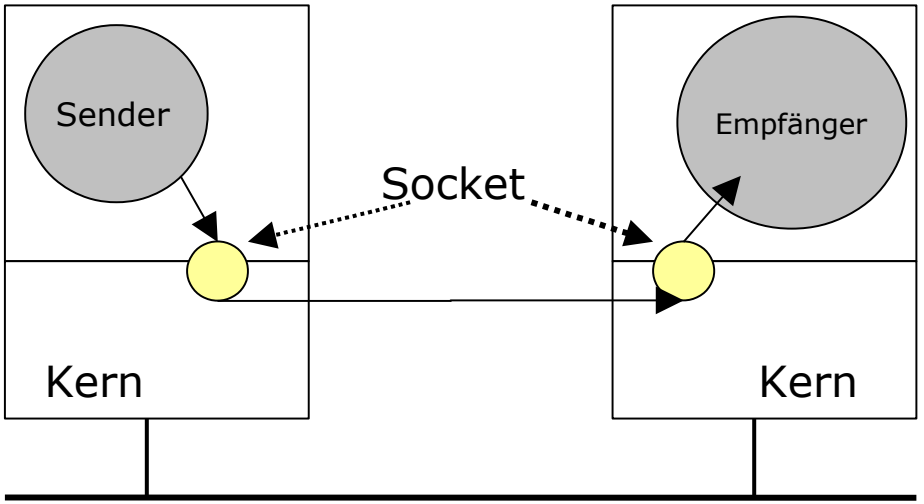
VI Sicherheit & Sicherheitsdienste

VII Zusammenfassung

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbi
FACHBEREICH INFORMATIK

Konzept

- Berkeley Sockets stammt von BSD Unix (1983 bzw 1989).
- Basis für verschiedene andere Sockets: Winsocks, Unix Sockets...
- Ein **Port** ist die Abstraktion eines physischen Ortes, über den Client und Server kommunizieren.
- Sockets sind Kommunikations-endpunkte, die dem Programmierer eine Schnittstelle zu einem Netzwerk zur Verfügung stellen



- Sockets können dynamisch erzeugt und zerstört werden.
- Eine Adresse wird an ein Socket gebunden, meist eine IP Adresse.
- Nach der Erzeugung wird ein Dateideskriptor zurück geliefert, der für den Aufbau einer Verbindung, das Lesen bzw. Schreiben der Daten und den Verbindungsabbau verwendet wird.
- Die eigentliche Kommunikation ist dann wie eine E/A Operation mit *read* und *write* möglich.

Arten von Sockets

Jeder Socket unterstützt eine spezielle Art der Vernetzung, die bei der Erzeugung des Socket anzugeben sind:

- **zuverlässiger verbindungs-orientierer Bytestrom,**
- **Zuverlässiger verbindungs- orientierter Paketstrom**
- **unzuverlässige Paketübertragung** zum elementaren Zugriff auf das Netzwerk (für Echtzeitanwendungen)

Bei der Erzeugung eines Socket muss das zu verwendende Protokoll angegeben werden:

- **TCP/IP** für zuverlässige Byte- und Packetübertragung
- **UDP** für unzuverlässige Übertragung.

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbi
FACHBEREICH INFORMATIK

Verbindungslose Sockets

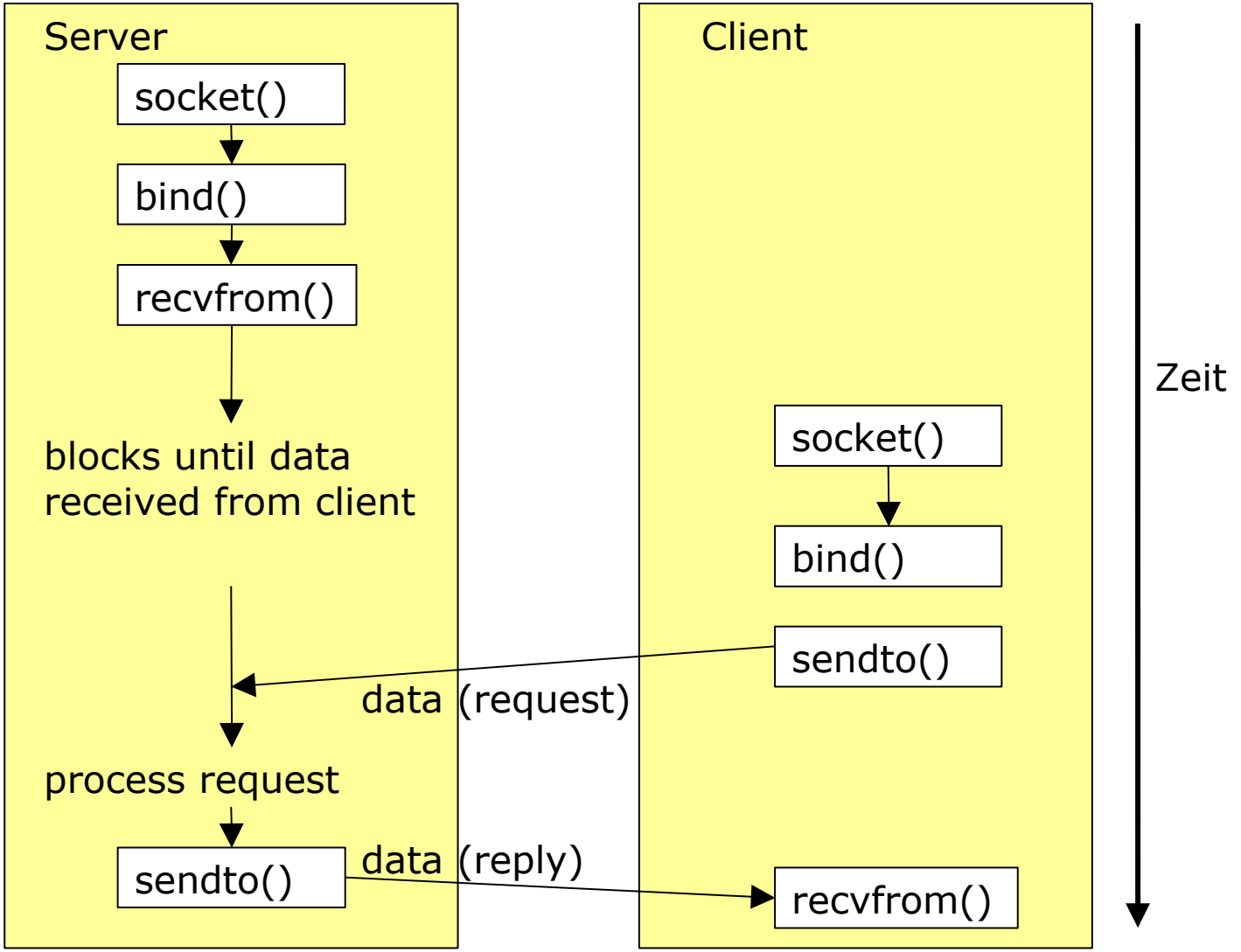
- Paketorientierte Kommunikation wird **sichtbar**.
- Der Anwendungsentwickler ist selbst für **Fehlerüberwachung** und -**korrektur**, Gewährleistung der **reihenfolge**-treuen Übertragung und der **Segmentierung** großer Datenvolumen zuständig.
- Datagramm-Sockets werden bei der Erzeugung nicht mit der Adresse des Zielprozesses initialisiert. Sie können mit jedem neu zu versendenden Datagramm eine andere **Zieladresse** erhalten.
- **Verschiedene Prozesse** können mit einem DatagramSocket nacheinander kontaktiert werden.
- Auch Datagramm-Sockets reservieren sich über das Betriebssystem eine **Portnummer**. Der Anwendungsentwickler braucht sich darum nicht zu kümmern.
- Allein ein Server muss immer einen festen Port wählen.
- Mit der Klasse DatagramSocket lassen sich auch **Multicast-Pakete** versenden.

  Als Zieladresse wird eine Multicast-Adresse angegeben. Zum Empfangen von Multicastpaketen wird MulticastSocket benutzt.

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbi
FACHBEREICH INFORMATIK

Verbindungsloser Socket-Ablauf

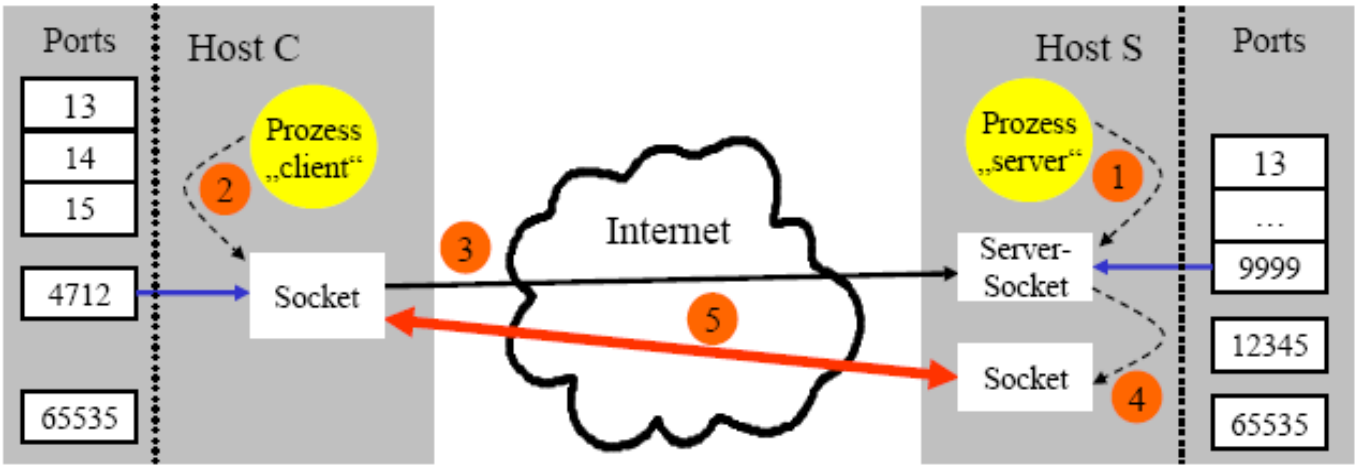
Bei verbindungslose Kommunikation:



Verbindungsorientierte Socket-Ablauf (1)

Bei verbindungsorientierter Kommunikation:

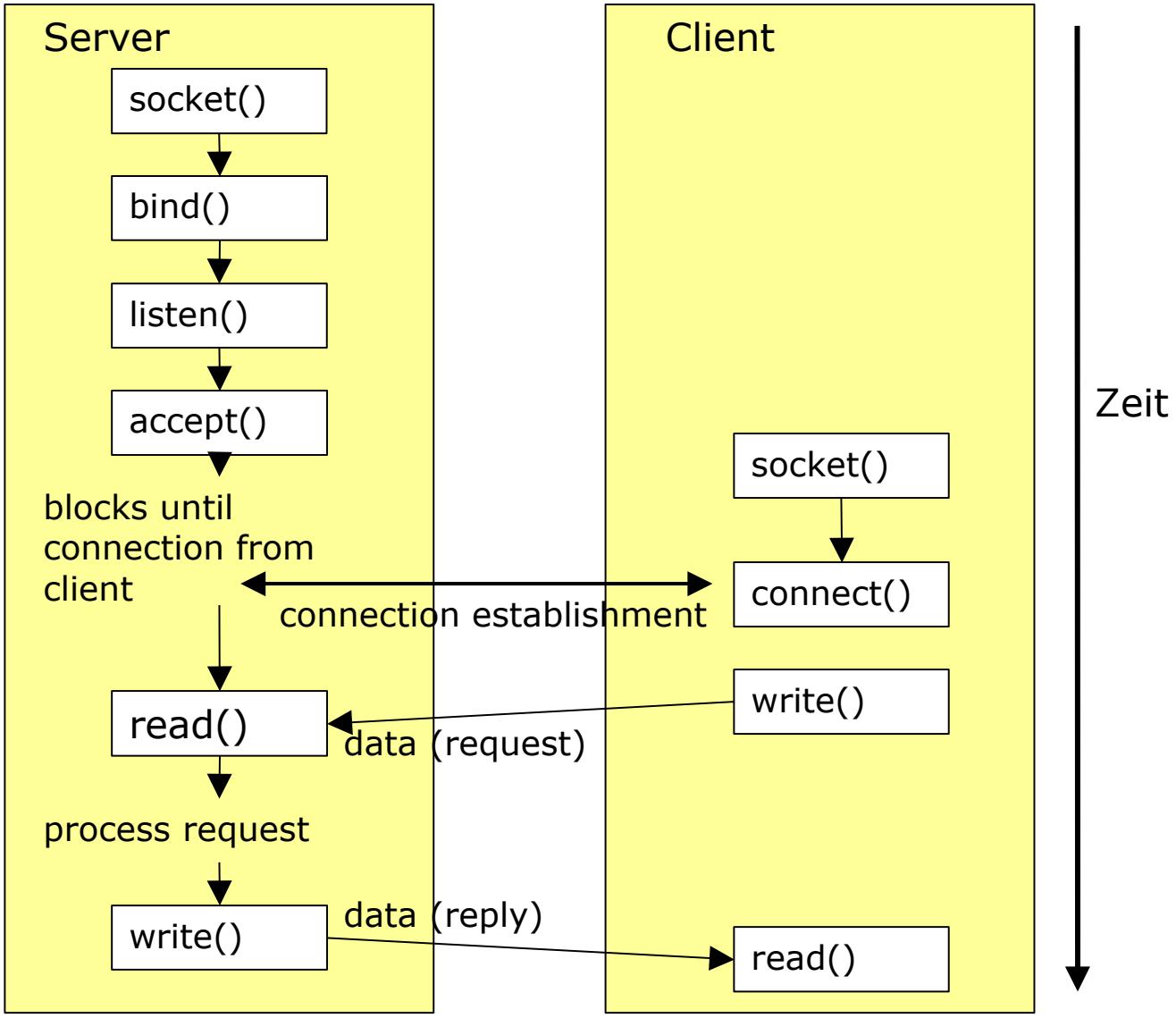
Accept erstellt einen neuen Server-Socket, sodass der erste Server-Socket weiter verwendet werden kann:
das setzt allerdings *multithreading* voraus.



1. Prozess „server“ in Host S erstellt einen (permanenten) Server-Socket.
2. Prozess „client“ in Host C erstellt einen Verbindungsendpunkt (Socket) mit einer vom Betriebssystem gelieferten Portnummer.
4. Ein Verbindungswunsch wird an den Kontaktport des Servers im Host S adressiert.
5. Der Prozess „server“ akzeptiert die Verbindung *und erzeugt neuen Socket.*
6. Die Verbindung wird übergeben und der Kontaktsocket ist frei für neue Anfragen.

Verbindungsorientierte Socket-Ablauf (2)

Bei verbindungsorientierter Kommunikation:

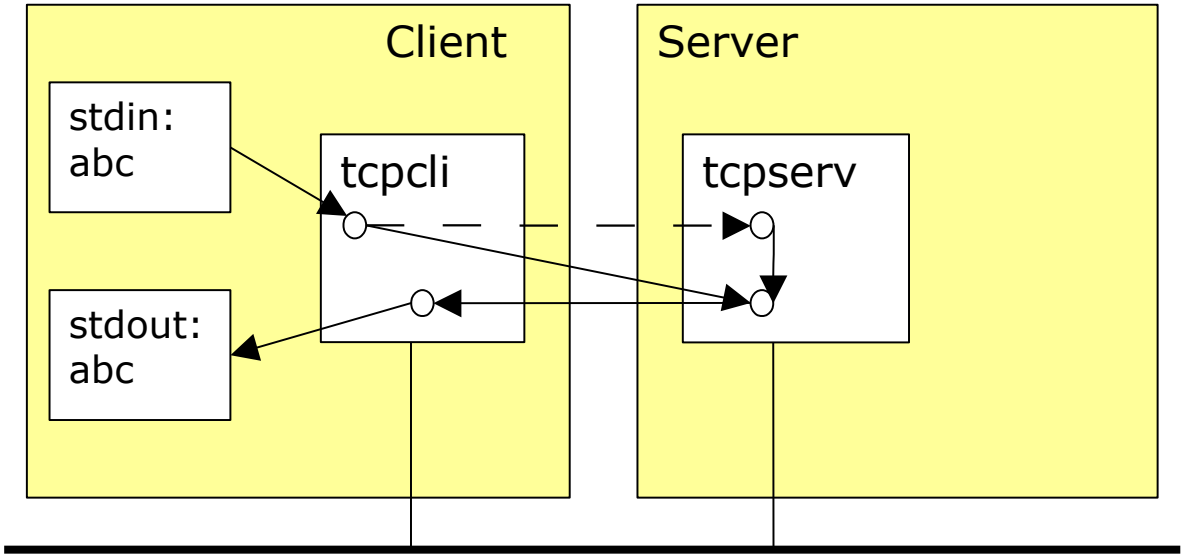


Beispiel – Echo Server

1. Der Klient liest eine Zeile von seiner Standard-eingabe und sendet sie zum Server.

2. Der Server liest eine Zeile vom Netzwerk und schreibt sie zurück zum Klienten.

3. Der Klient liest eine Zeile vom Netzwerk und gibt sie auf seiner Standard-ausgabe aus.



Der Server muss zuerst gestartet werden, ansonsten meldet der Client:

„tcpcli: client: can't connect to server (Connection refused)"

Beispiel – Echo Server – Quellen (1)

```
/*****************************************************************
 * main function, server for the TCP/IP echo server
 */
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);

     ...
```

socket() erstellt einen Socket. Der Socket ist noch nicht an einem Port gebunden.

bind() bindet den socket an einer Internet Adresse (z.B. host+port).

Beispiel – Echo Server – Quellen (2)

```
/*****************************************************************
 * main function, server for the TCP/IP echo serve
 */
int main(int argc, char **argv)
{
     int                 sockfd, newsockfd, clilen
     struct sockaddr_in  cli_addr, serv_addr;

     // Open a TCP socket (an Internet stream so
     if ( (sockfd = socket(AF_INET, SOCK_STREAM
          perror("server: can't open stream so

      // Bind our local address so that the
     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_PO

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

     listen(sockfd, 5);

     ...
```

```
BYTEORDER(3) Linux Programmer's Manual BYTEORDER(3)



NAME
       htonl, htons, ntohl, ntohs - convert values
between host and network byte order

SYNOPSIS
       #include <arpa/inet.h>

       uint32_t htonl(uint32_t hostlong);

       uint16_t htons(uint16_t hostshort);

       uint32_t ntohl(uint32_t netlong);

       uint16_t ntohs(uint16_t netshort);

DESCRIPTION
       The  htonl()  function converts the unsigned
integer hostlong from host byte order to network
byte order.

       The htons() function converts the unsigned
short integer  hostshort  from  host  byte order to
network byte order....
```

Beispiel – Echo Server – Quellen (3)

```
/*****************************************************************
 * main function, server for the TCP/IP echo server
 */
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 *) &se
               sizeof(serv_addr)) < 0)
          perror("server: can't bind

     listen(sockfd, 5);

     ...
```

listen() blockiert nicht, sondern trifft notwendige Vorbereitungen (Pufferspeicher, usw.) für späterere Verbindungsaufbau.

In dieser Beispiel, können bis zu 5 Verbindungen im Wartezustand gesetzt werden, bevor eine neue Verbindung verweigert wird.

Ab jetzt kann der Server-Socket Verbindungen von Clienten annehmen.

Beispiel – Echo Server – Quellen (4)

```
    ...

    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 req
               exit(0);
          }
          close(newsockfd);        // parent process
    }
}
/*******                                          */
```

accept() blockiert bis ein Client eine Verbindung aufbaut. Rückgabewert vom **accept** ist der neuer socket.

fork() erstellen einen Clone vom Server-Prozess. Nur der Rückgabewert – childpid – unterscheidet **Vater** vom **Kind**.

Das Kind (Clone) schließt seine Verbindung zum ersten Socket (sodass der Vater ihn weiter verwenden kann) und verwendet den neuen Socket.

Der Vater schließt schließt seine Verbindung zum neuen Socket (sodass das Kind ihn weiter verwenden kann) und verwendet den ursprunglichen Socket - d.h. springt nach oben und ruft nochmals **accept()** auf.

str_echo() ist kein normale Sockets-Funktion, sonden Teil des Beispiels...

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbi
FACHBEREICH INFORMATIK

Beispiel – Echo Server – Quellen (5)

```
/*****************************************************************
 *
 * Read a stream socket one line at a time, and write each line back
 * to the sender.
 *
 * Return when the connection is terminated.
 */
void str_echo(int sockfd)
{
     const int MAXLINE=255;
     int  n;
     char line[MAXLINE];

     for ( ; ; ) {
          n = readline(sockfd, line, MAXLINE);
          if (n == 0)
               return;        /* connection terminated */
          else if (n < 0) {
               perror("str_echo: readline error");
          }

          if (writen(sockfd, line, n) != n) {
               perror("str_echo: writen error");
          }
     }
}
/*****************************************************************
```

str_echo() wurde vom Kind (Clone) Server aufgerufen, und ruft **readline()** und **writen()** auf.

readline() ließt eine Zeile vom sockfd, als ob sockfd eine Datei wäre. Die Funktion wird auf die nächste Folie definiert.

writen() ist ähnlich – der schreibt *n* Bytes zum sockfd.

Bemerkung: sockfd wird für sowohl Lesen als auch Schreiben verwendet!

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbi
FACHBEREICH INFORMATIK

Beispiel – Echo Server – Quellen (6)

```
/*****************************************************************
 * Read a line from a descriptor.  Read the line one byte at a time,
 * looking for the newline.  We store the newline in the buffer,
 * then follow it with a null (the same as fgets(3)).
 * We return the number of characters up to, but not including,
 * the null (the same as strlen(3)).
 */
int readline(int fd,char* ptr, int maxlen)
{
     int  n, rc;
     char c;

     for (n = 1; n < maxlen; n++) {
          if ( (rc = read(fd, &c, 1)) == 1) {
               *ptr++ = c;
               if (c == '\n')
                    break;
          } else if (rc == 0) {
               if (n == 1)
                    return(0); /* EOF, no data read */
               else
                    break;     /* EOF, some data was read
          } else
               return(-1);     /* error */
     }

     *ptr = 0;
     return(n);
}
/*****************************************************************/
```

readline() ließt eine Zeile vom sockfd, als ob sockfd eine Datei wäre.

read() ist einer grundlegender Systemaufruf, der in Unix-Systemen verwendet wird, um mit Dateien (und Devices usw.) zu kommunisieren. Der Rückgabewert ist der Anzahl tatsächlich (erfolgreich) gelesene Bytes.

Hier wird **read()** verwendet, um genau ein Byte zu lesen – und es zu prüfen, bevor wir weiter lesen.

Beispiel – Echo Client – Quellen (1)

```
/*****************************************************************
 * main function, client for TCP/IP echo server
 */
int main(int argc, char** argv)
{
     int                 sockfd;
     struct sockaddr_in  serv_addr;

     // Fill in the structure "serv_addr" with the address of the
     // server that we want to connect with.
     bzero((char *) &serv_addr, sizeof(serv_addr));
     serv_addr.sin_family      = AF_INET;
     serv_addr.sin_addr.s_addr = inet_addr(SERV_HOST_ADDR);
     serv_addr.sin_port        = htons(SERV_TCP_PORT);

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

     // Connect to the server.
     if (connect( sockfd,
                  (struct sockaddr *) &serv_addr,
                  (serv_addr)) < 0)
          perror("client: can't connect to server");

     str_cli(stdin, sockfd);        /* do it all */

     close(sockfd);
     exit(0);
}
/*****************************************************************/
```

socket() erstellt einen Socket. Der Socket wird nicht an einem Port gebunden.

connect() baut eine Verbindung zum Server auf. Das setzt voraus, dass der Server existiert, und **listen()** und **accept()** aufgerufen hat.

str_cli() wird auf die nächste Folie definiert

Beispiel – Echo Client – Quellen (2)

```
/*****************************************************************
 * Read the contents of the FILE *fp, write each line to the stream
 * socket then read a line back from the socket and write it to the
 * standard output.
 */
void str_cli(FILE *fp, int sockfd)
{
     const int MAXLINE=255;
     int  n;
     char *retval = NULL;
     char sendline[MAXLINE], recvline[MAXLINE + 1];

     while ( true ) {
          retval = fgets(sendline, MAXLINE, fp);
          if ( NULL == retval ) break; // leave loop

          n = strlen(sendline);
          if (writen(sockfd, sendline, n) != n)
               perror("str_cli: writen error on socket");

          // Now read a line from the socket and write
          // it to our standard output.
          n = readline(sockfd, recvline, MAXLINE);
          if (n < 0)
               perror("str_cli: readline error");

          recvline[n] = 0;    /* null terminate */
          fputs( "Server replies:", stdout );
          fputs(recvline, stdout);
     }

     if (ferror(fp))
          perror("str_cli: error reading file");
}
/*****************************************************************
```

str_cli() wurde vom Client aufgerufen, und ruft **readline()** und **writen()** auf.
Vgl. **str_echo()** oben!

writen() wurde oben definiert – es ist genau dieselbe Funktion!

readline() ließt eine Zeile vom sockfd, als ob sockfd eine Datei wäre.
Die 2 Funktionen basieren auf die System-aufrufen **read()** bzw. **write()**

Wichtig: Es wird eine Zeile pro Operation ausgetauscht *nur weil wir es so programmiert haben!* **Es muss nicht so sein...**

Es muss auch nicht immer Text sein! *Any bytes will do!*
Auch binäre Daten können übertragen werden!

Struktur von Kapitel II - Kommunikation

I Einleitung

II Grundlegende Kommunikationsdienste

- A Kommunikationsformen
- B Netzwerk Grundlagen
- C Berkeley Sockets
- D Java Sockets
- E Remote Procedure Calls (RPC)
- F XML-RPC
- G Java RMI

III Middleware

IV Architekturen & Algorithmen

- A Synchronisierung
- B Konsistenz und Replication
- C Fehlertoleranz

V Beispiele bzw. Dienste

- A Verteilte Dateisysteme
- B Namensdienste

VI Sicherheit & Sicherheitsdienste

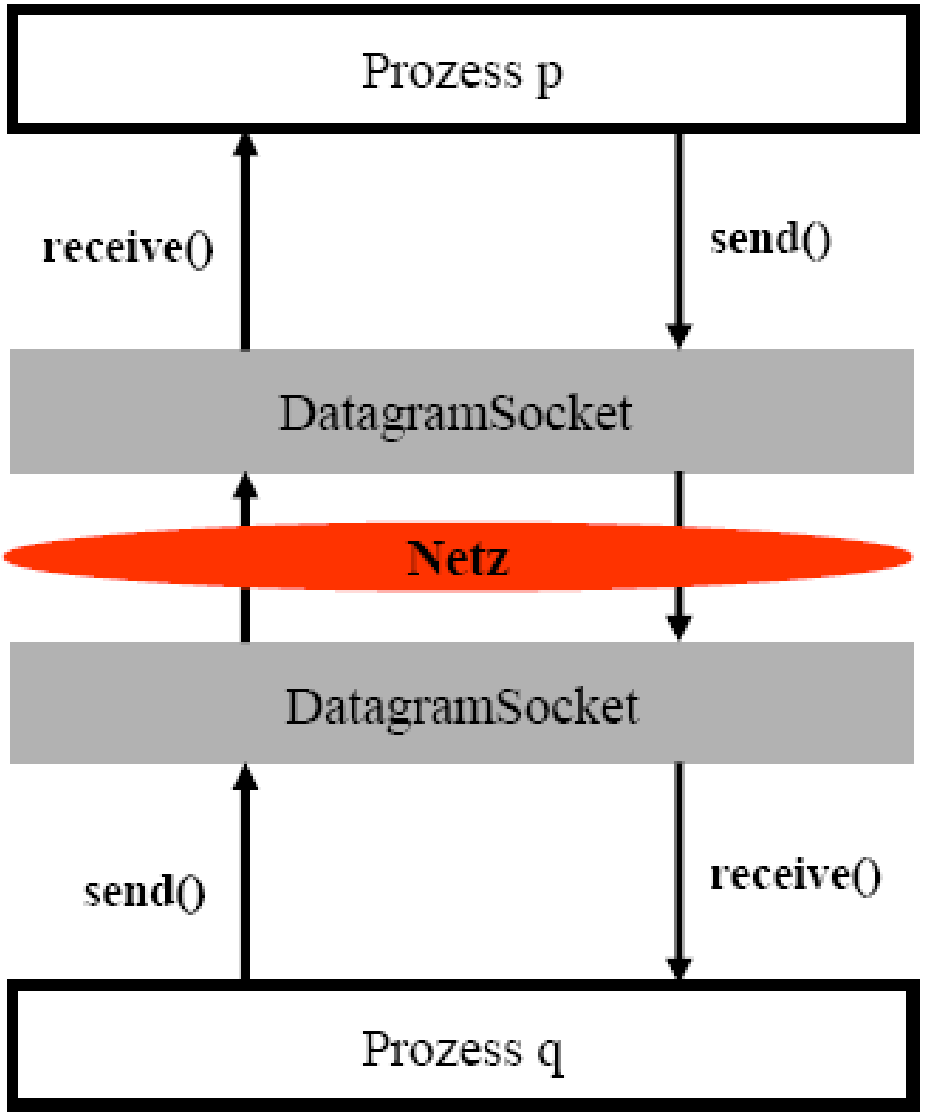
VII Zusammenfassung

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbi
FACHBEREICH INFORMATIK

Verbindungslose Sockets in Java

Eine Java-Anwendung greift mit den Methoden **send()** und **receive()** direkt auf einen DatagramSocket zu.

Die Klasse DatagramPacket modelliert ein Datagramm.
Darin ist ein Verweis auf den Puffer mit der Nachricht enthalten.



Beispiel – Time Server in Java

```
import java.net.*;
import java.util.*;

public class UDPTimeServer {
     public static void main(String[] args) {
          try
              {
                byte data[] = new byte[1024];
                DatagramPacket packet;
                DatagramSocket socket = new DatagramSocket(13131);
                System.out.println("TimeServer startet auf Port 13131");
                while ( true )
                {
                  // Auf Anfrage warten
                  packet = new DatagramPacket(data, data.length);
                  socket.receive(packet);
                  // Empfänger auslesen, aber Paketinhalt ignorieren
                  InetAddress address = packet.getAddress();
                  int         port    = packet.getPort();
                  // Paket für Empfänger zusammenbauen
                  String s = "Antwort auf Anfrage von "+address+" am Port "
                  +port+": "+new Date().toString() + "\n";
                  data = s.getBytes();
                  packet = new DatagramPacket(data,data.length,address,port);
                  socket.send(packet);
                }
              }
              catch (Exception e)
              { System.out.println(e); }
     }
}
```

Normalerweise Port 13 (wenn man darf...).

Eingabe wird nur verwendet, um Client zu identifizieren

Ausgabe beinhaltet die aktuelle Uhrzeit...

Beispiel – Time Client in Java

```
import java.io.*;
import java.net.*;

public class UDPTimeClient {

//  static String host = new String("192.168.178.20");
    static String host = new String("localhost");

    static int port = 13131;

    public static void main(String[] args) throws IOException {
        System.out.println("TimeClient startet");
        DatagramSocket socket = new DatagramSocket();
        byte msg[] = new byte[256];
        InetAddress address = InetAddress.getByName(host);

        DatagramPacket packet = new DatagramPacket(msg, msg.length,
                    address, port);
        socket.send(packet);
        packet = new DatagramPacket(msg, msg.length);
        socket.receive(packet);
        System.out.println("Die Zeit wird angefragt bei "+host
                           +" Port "+port);
        System.out.println(new String(packet.getData()));
        socket.close();
    }
}
```

Normalerweise Port 13 (wenn man darf...).

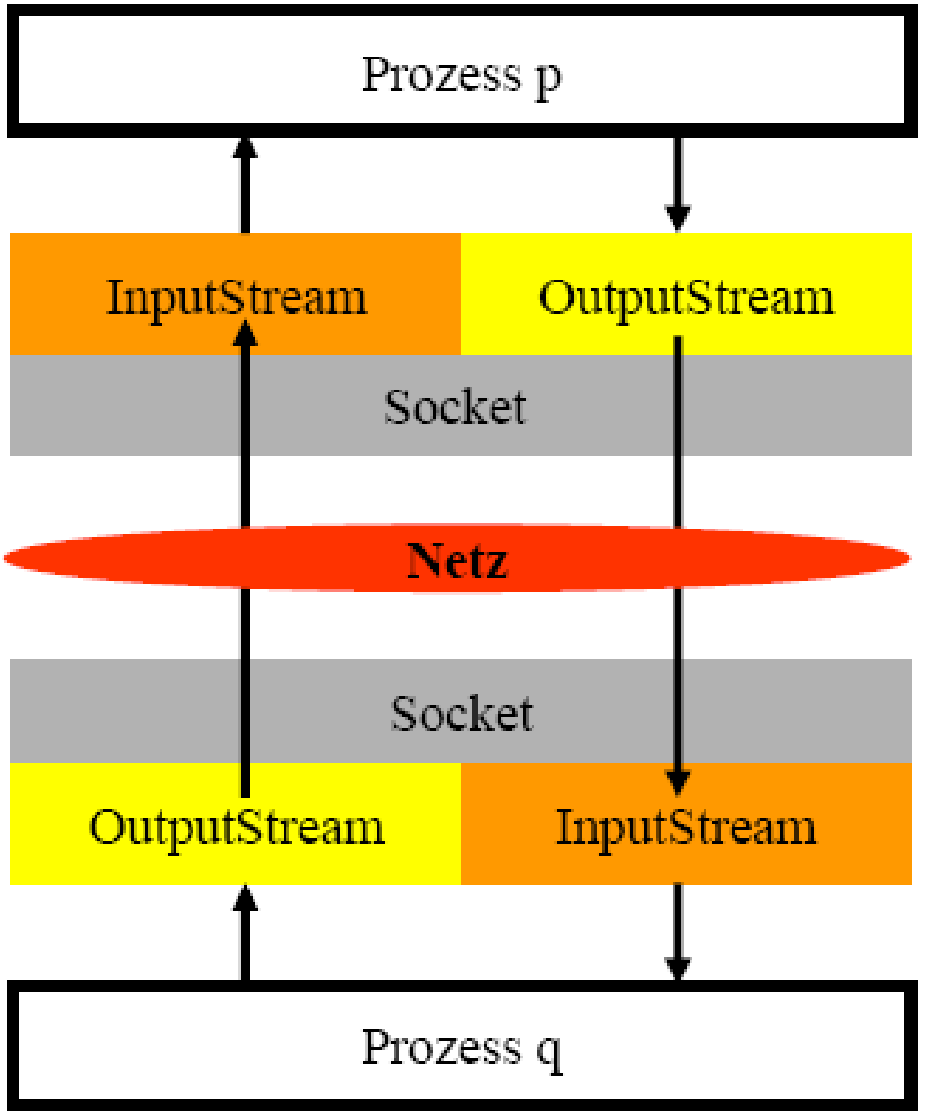
Ausgabe fast leer – identifiziert den Client trotzdem.

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbi
FACHBEREICH INFORMATIK

Verbindungsorienterte Sockets in Java

Eine Java-Anwendung sieht allein Stream-Objekte.

Ein Stream-Objekt wird auf einem Socket erzeugt mit **getInputStream**() oder **getOutputStream**().



Beispiel – Echo Server in Java (1)

```
import java.io.*;
import java.net.*;
public class TCPServer {
    public static void main(String[] args) throws Exception{
        String line; boolean verbunden;
        ServerSocket listenSocket = new ServerSocket(9999);
        while (true){
            Socket cliSocket = listenSocket.accept();
            verbunden = true;
            BufferedReader fromClient =
                    new BufferedReader(
                        new InputStreamReader(
                                cliSocket.getInputStream()));
            DataOutputStream toClient =
                        new DataOutputStream(
                            cliSocket.getOutputStream());
            while(verbunden){
                line = fromClient.readLine();
                System.out.println("Empfangen: " + line);
                if (line.equals("stop")) {
                    verbunden = false;
                    System.out.println("Stop empfangen");
                } else
                    toClient.writeBytes(line.toUpperCase()
                                        +'\n');
            } // end while verbunden
            toClient.close(); cliSocket.close();
        } // end repeat forever
    } // end method main
} // end class TCPServer
```

ServerSocket constructor ersetzt Aufruf zu **server()** *und* **bind()** *und* **listen()**.

Aus dem ClientSocket werden zwei **Streams** erstellt: Ein für Eingabe, ein für Ausgabe

ReadLine() ist ein Method vom Class **BufferedReader**, **writeBytes()** ist ein Method vom Class **DataOutputStream**.

Beispiel – Echo Client in Java

```
import java.io.*;
import java.net.*;

public class TCPClient {
    public static void main(String[] args) throws Exception{
        String line;  boolean verbunden = true;
        Socket cliSocket = new Socket("localhost",9999);
        BufferedReader stdIn = new BufferedReader(
                        new InputStreamReader(System.in));
        DataOutputStream toServer =
            new DataOutputStream(cliSocket.getOutputStream());
        BufferedReader fromServer =
           new BufferedReader(
              new InputStreamReader(
                 cliSocket.getInputStream()));
        while (verbunden){
            System.out.print("Nachricht für Server: ");
            line = stdIn.readLine();
            toServer.writeBytes(line+'\n');
            if (line.equals("stop")) verbunden = false;
            else
                System.out.println("Server antwortet: "+
                new String(fromServer.readLine().getBytes()));

        } // end while verbunden
        fromServer.close();
        cliSocket.close();
    } // end method main
} // end class TCPClient
```

Socket constructor ersetzt Aufruf zu **socket()** *und* **connet()**.

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbi
FACHBEREICH INFORMATIK

Struktur von Kapitel II - Kommunikation

I Einleitung

II Grundlegende Kommunikationsdienste

- A Kommunikationsformen
- B Netzwerk Grundlagen
- C Berkeley Sockets
- D Java Sockets
- E C++17(?)/Boost Sockets
- F Remote Procedure Calls (RPC)
- G XML-RPC
- H Java RMI

III Middleware

IV Architekturen & Algorithmen

- A Synchronisierung
- B Konsistenz und Replication
- C Fehlertoleranz

V Beispiele bzw. Dienste

- A Verteilte Dateisysteme
- B Namensdienste

VI Sicherheit & Sicherheitsdienste

VII Zusammenfassung

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbi
FACHBEREICH INFORMATIK

Beispiel – TCP Date/Time Client in C++/Boost

```
#include <iostream>
#include <boost/array.hpp>
#include <boost/asio.hpp>

using boost::asio::ip::tcp;

int main(int argc, char* argv[]) {
  try {
    if (argc != 3) {
      std::cerr << "Usage: blocking_tcp_echo_client <host> <port>\n";
      return 1;
    }

    boost::asio::io_service io_service;

    tcp::resolver resolver(io_service);
    tcp::resolver::query query(tcp::v4(), argv[1], argv[2]);
    tcp::resolver::iterator endpoint_iterator = resolver.resolve(query);

    tcp::socket socket(io_service);
    boost::asio::connect(socket, endpoint_iterator);

    for (;;) {
      boost::array<char, 128> buf;
      boost::system::error_code error;

      size_t len = socket.read_some(boost::asio::buffer(buf), error);

      if (error == boost::asio::error::eof)
        break; // Connection closed cleanly by peer.
      else if (error)
        throw boost::system::system_error(error); // Some other error.

      std::cout.write(buf.data(), len);
    }
  }
  catch (std::exception& e) {
    std::cerr << e.what() << std::endl;
  }

  return 0;
}
```

Quellen:
http://www.boost.org/doc/libs/1_49_0/doc/html/boost_asio/tutorial/tutdaytime1/src.html
http://www.boost.org/doc/libs/1_49_0/doc/html/boost_asio/tutorial/tutdaytime1.html

Servername von der Kommandozeile

io_service Object ist Pflicht bei der asio Bibliothek

Servername wird beim DNS nachgeschlagen...

Socket erzeugt und mit Server verbunden – mit *einem* der Ergebnissen des DNS-Lookups(!)

Text wird gelesen, ohne zuerst etwas zu schicken. Solange die Verbindung steht, wird weiter gelesen.

Beispiel – TCP Date/Time Server in C++/Boost (1)

```
#include <ctime>
#include <iostream>
#include <string>
#include <boost/asio.hpp>

using boost::asio::ip::tcp;

std::string make_daytime_string() {
  using namespace std; // For time_t, time and ctime;
  time_t now = time(0);
  return ctime(&now);
}

int main() {
  try   {
    boost::asio::io_service io_service;

    tcp::acceptor acceptor(io_service, tcp::endpoint(tcp::v4(), 13));

    for (;;) {
      tcp::socket socket(io_service);
      acceptor.accept(socket);

      std::string message = make_daytime_string();

      boost::system::error_code ignored_error;
      boost::asio::write(socket, boost::asio::buffer(message),
                         ignored_error);
    }
  }
  catch (std::exception& e) {
    std::cerr << e.what() << std::endl;
  }

  return 0;
}
```

Quellen:
http://www.boost.org/doc/libs/1_49_0/doc/html/boost_asio/tutorial/tutdaytime2/src.html
http://www.boost.org/doc/libs/1_49_0/doc/html/boost_asio/tutorial/tutdaytime2.html

„Geschäftslogik“

io_service Object ist Pflicht bei der asio Bibliothek

Port 13, hard coded (besser – aus argv[1])

Dieser Server verarbeitet eine Verbindung nach der andere. Dafür brauchern wir ein Socket und einen Acceptor…

Ohne zu warten, wird ein String gebaut und an den Klient geschrieben. Das war‘s.

Struktur von Kapitel II - Kommunikation

I Einleitung

II Grundlegende Kommunikationsdienste

- A Kommunikationsformen
- B Netzwerk Grundlagen
- C Berkeley Sockets
- D Java Sockets
- E C++17(?)/Boost Sockets
- F Remote Procedure Calls (RPC)
- G XML-RPC
- H Java RMI

III Middleware

IV Architekturen & Algorithmen

- A Synchronisierung
- B Konsistenz und Replication
- C Fehlertoleranz

V Beispiele bzw. Dienste

- A Verteilte Dateisysteme
- B Namensdienste

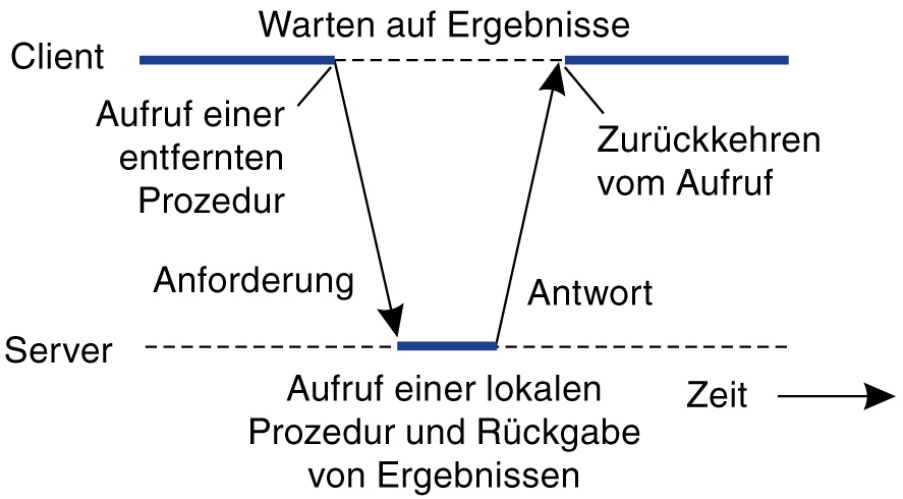
VI Sicherheit & Sicherheitsdienste

VII Zusammenfassung

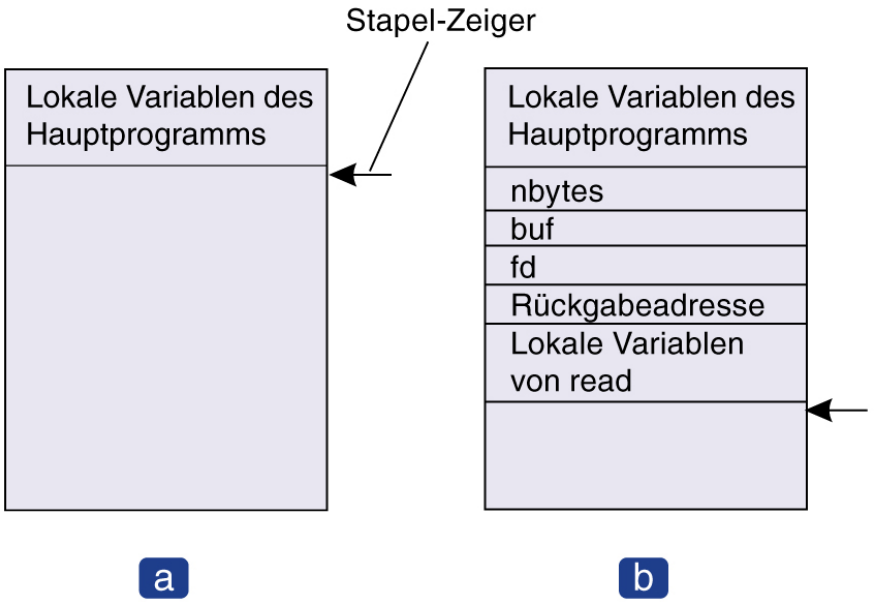
h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbi
FACHBEREICH INFORMATIK

Remote Procedure Call

- **Herausforderung:** Wie können wir die Programmierung leichter (produktiver, abstrakter) machen?
- **Idee 0:** Stellen wir die Kommunikationskanale wie Dateien vor → Sockets.
- **Idee 1:** Stellen wir die Server (Dienstanbieter) wie Funktionen vor, die wir aufrufen können → RPC = Remote Procedure Calls.

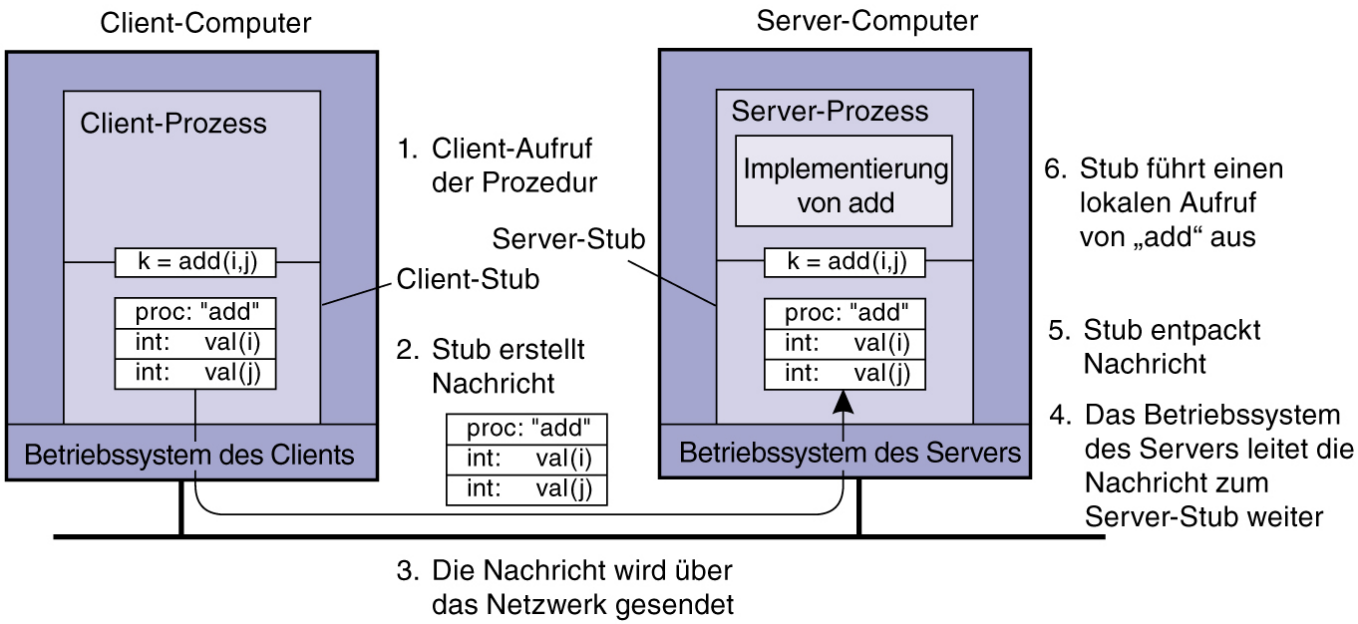


- **Problem:** Wie übertragen wir
  1) Parameter (zu Remote Procedures)?
  2) Rückgabewerte (von Remote Procedures)?
- **Vergleich:** Local Procedure Call mit *Stack:*



Remote Procedure Call (2)

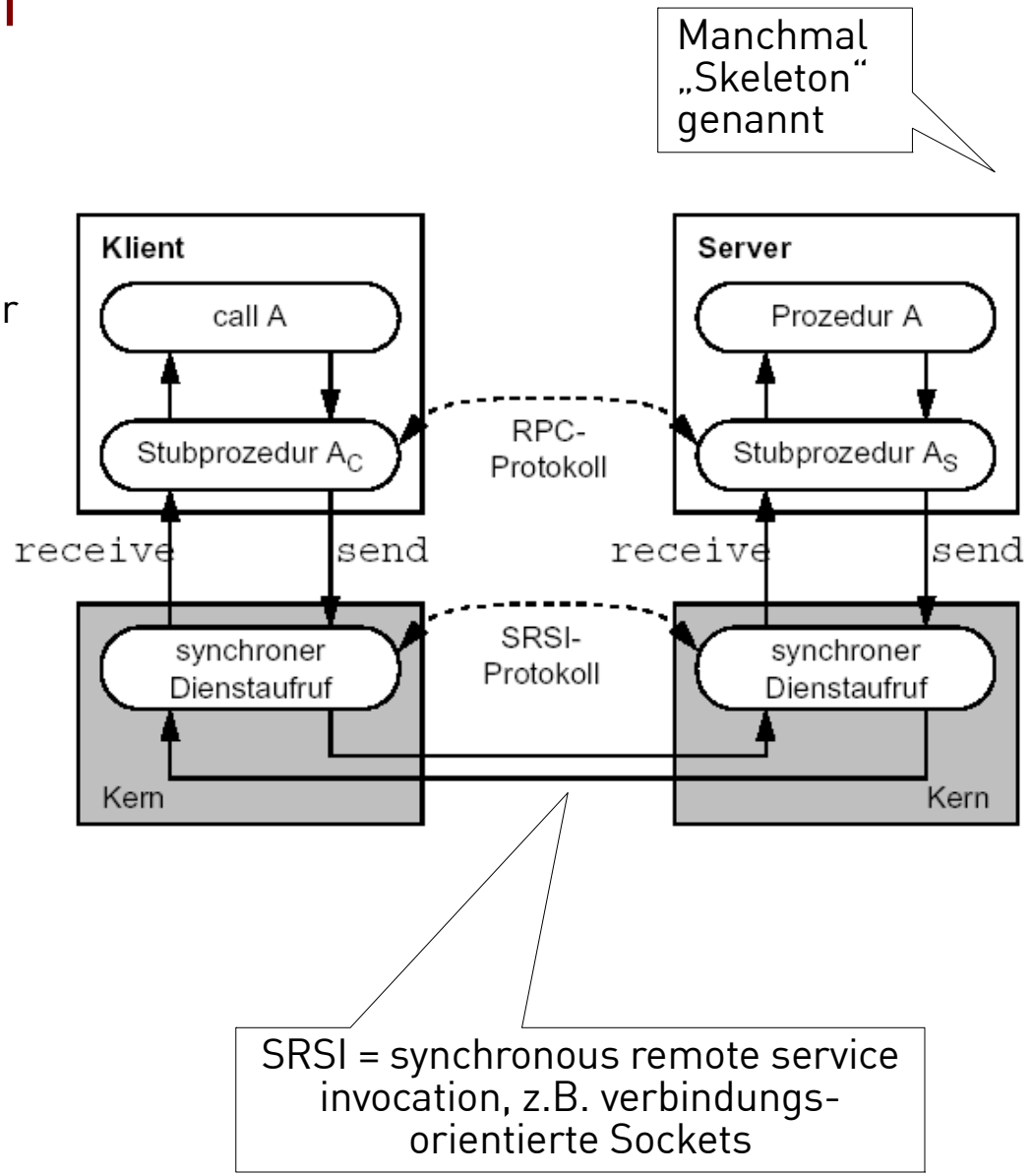
- **Problem:** Wie übertragen wir:
  1) Parameter?
  2) Rückgabewerte?
- **Lösungsansatz:** *Datenstrukturen* stellen die Parameter bzw. Rückgabewerte dar = ***Stubs***



RPC Ablauf

Aufgaben der Stubprozedur A_{Client}

1) Parameter zu einer Nachricht zusammensetzen
2) Serialisierungs-prozeß = **Marshalling** (Verpacken der Parameter)



Aufgaben der Stubprozedur A_{Server}

1) Übergabeparameter rekonstruieren
2) Deserialisierung = **Demarshalling** (Entpacken der Parameter)
3) Aktivierung der Prozedur A

Die Übertragung der Rückgabeparameter erfolgt analog.

RPC Einschränkungen (1)

- Da auf zwei verschiedenen Rechnern gearbeitet wird, gibt es **keinen gemeinsamen Adressraum:**
  - **Kein Zeiger!**
  - **Kein Call-by-reference!**
  - Stattdessen: Call-by-Copy und Call-by-Restore (auch call-by-value/result)
- Durch den **Kommunikations-Overhead** ist die Aufrufdauer um Größenordnungen länger als bei lokalen Aufrufen.
  - **Also – wann lohnt es?**
  - **Antwort:** Wenn der Server Informationen bzw. Fähigkeiten hat, die der Client *nicht* hat.

Beispiel: NFS = Network File System.
Server bietet Zugang zu einer Festplatte an.
ONC (Sun) RPC wurde für NFS gebaut!

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbi
FACHBEREICH INFORMATIK

RPC Einschränkungen (2)

- Unterschiedliche Rechner stellen **Zahlen unterschiedlich** dar (1er Komplement, 2er Komplement, Big-Endian vs. Little-Endian)...

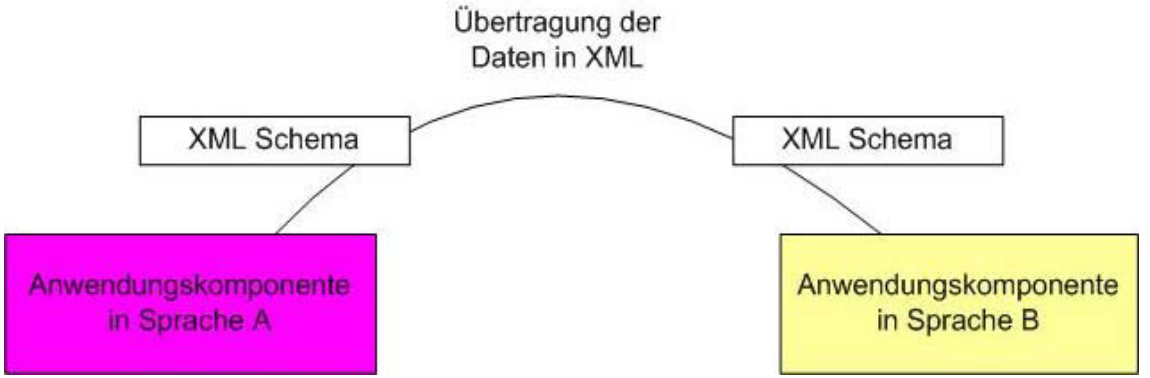
  Lösungen:

  - **Heterogenität von Hardware bzw. Betriebssysteme** wird transparent für die Anwendung durch die **Middleware** behandelt.
  - **Heterogenität der Programmiersprache:** Einführung eines einheitlichen Datenformats:
    - **Externes** Datenformat oder
    - **Plattformspezifisches** Datenformat

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbi
FACHBEREICH INFORMATIK

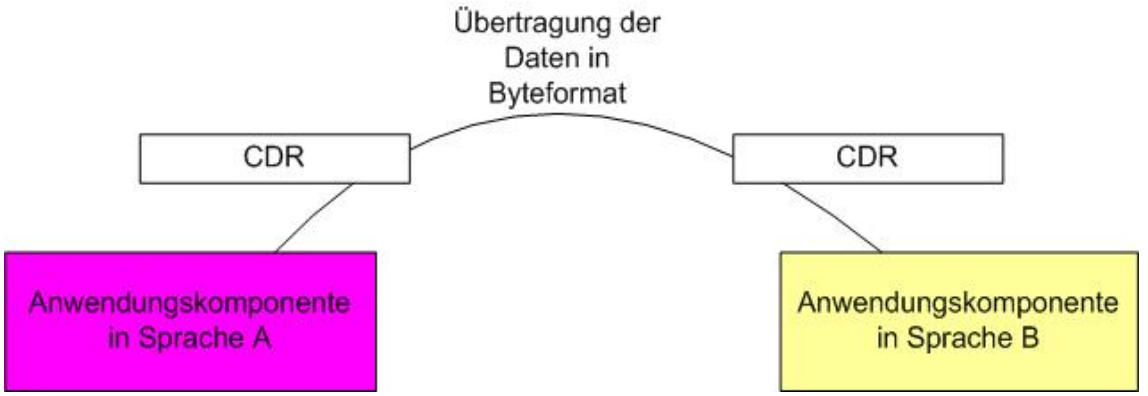
Externes Datenformat = XML

- Als externes Datenformat hat sich weitgehend XML und XML Schema durchgesetzt.
- Ein gemeinsames Datenformat ist als XML Schema definiert.
- Die zu übertragenden Daten werden anhand des Schemas in XML übersetzt und z.B. über HTTP übertragen.
- Beispiel: Web Services mit XML, XML-RPC (s.u.).



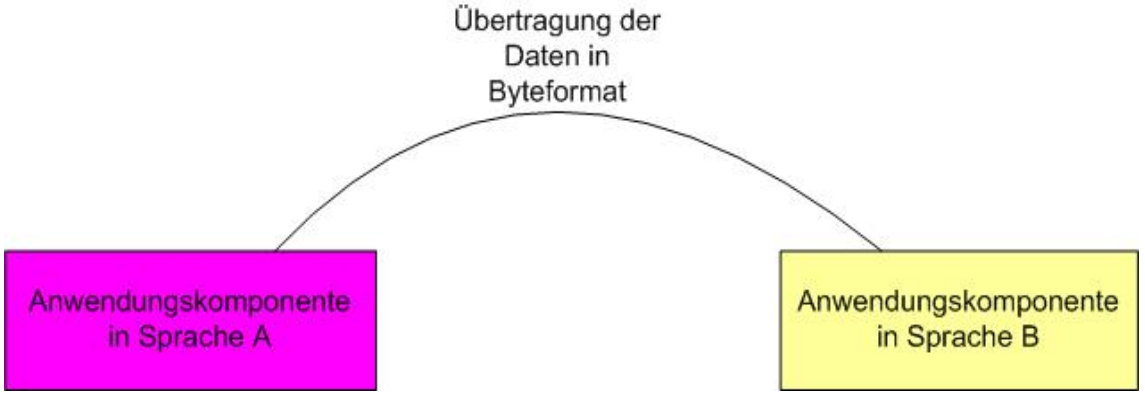
Plattformspezifisches Datenformat: CORBA

- Common Data Representation (CDR) als gemeinsames Datenformat
- Daten werden entsprechend der Vorgaben der CDR in Byteformat umgewandelt.

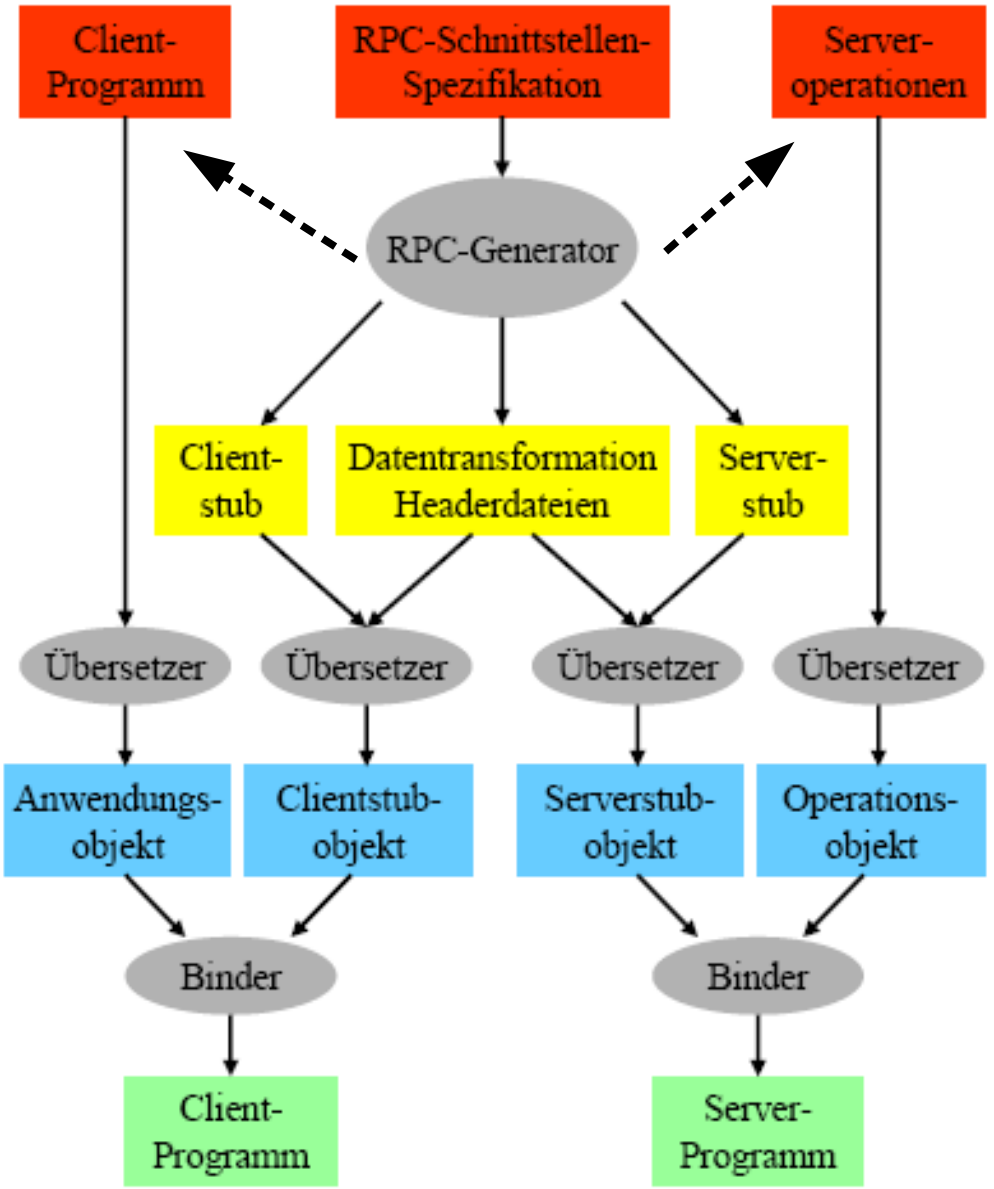


Plattformspezifisches Datenformat: Java

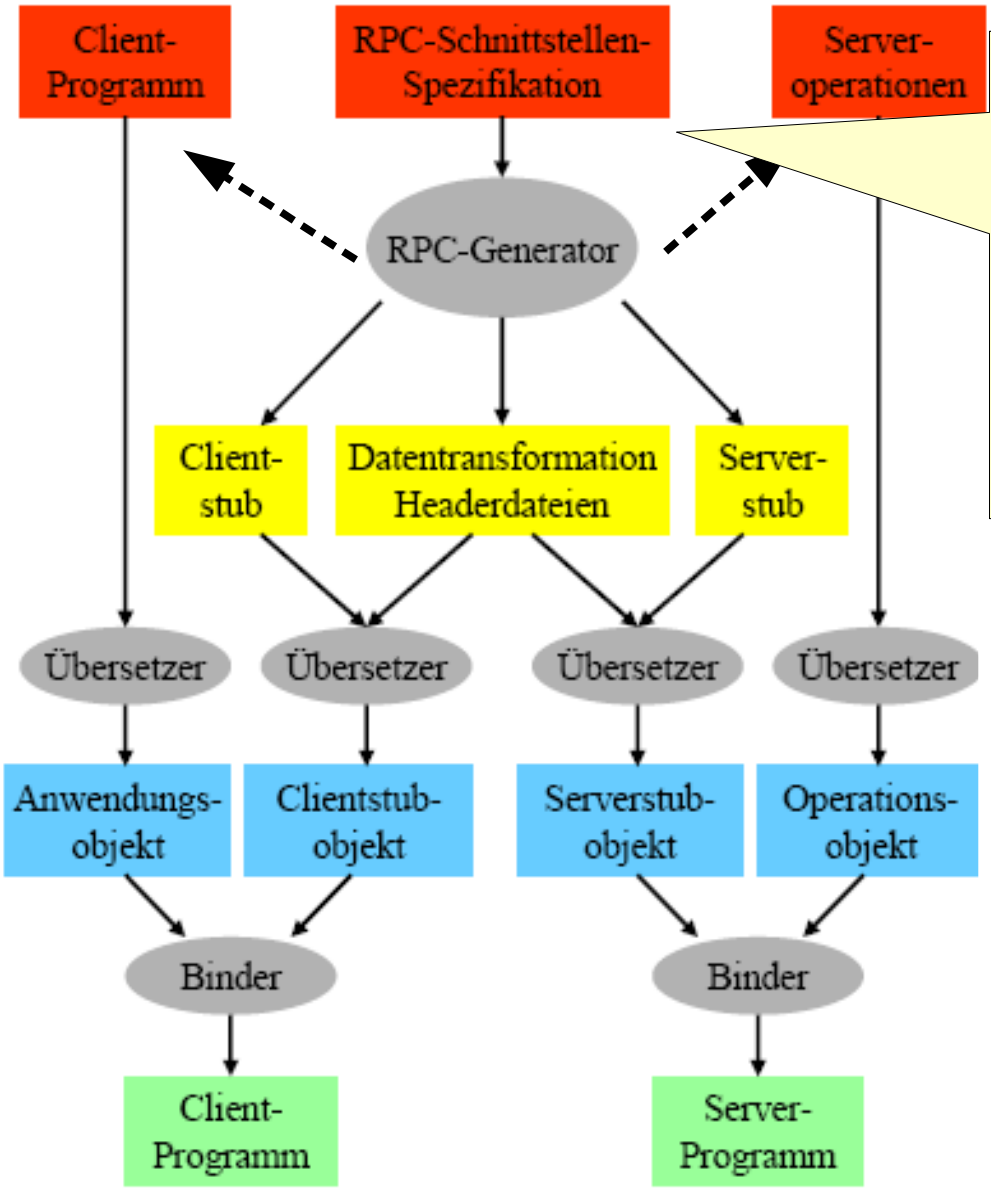
- Java geht von einer einheitlichen Sprache bei Client und Server aus. Das plattformspezifische Format ist Java selbst.
- Die Datentransformation erfolgt durch Objekt-serialisierung direkt in Byteformat.



SUN bzw. ONC rpcgen (1)



SUN bzw. ONC rpcgen (2)



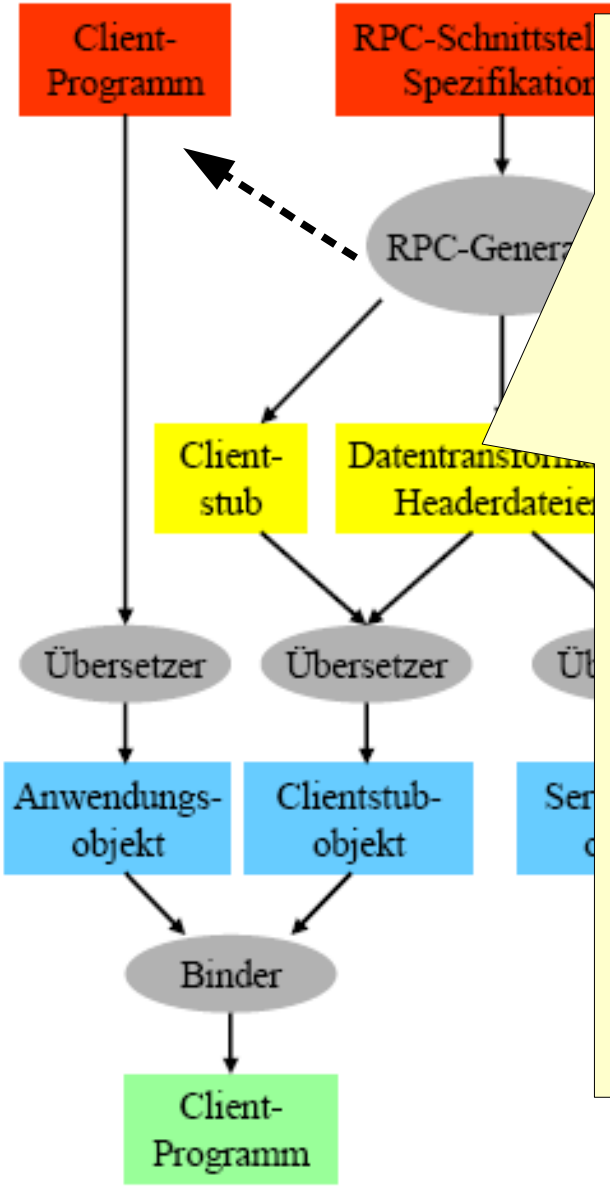

```
$ cat addiere.x
struct add_struct {
  int p1;
  int p2;
};

program ADDIERE_TEST {
    version ONE {
        int addiere(add_struct p) = 1;
    } = 1;
} = 1234567;
```

Dieses IDL-Interface enthält die Routinen-Spezifikation, die Definition der Rückgabe und Formalparameter der Funktion sowie die Deklaration etwaiger verwendeter Strukturen.

Die IDL heißt „XDR“ = „eXternal Data Representation“. Erlaubte Datentypen sind (unsigned) int, (unsigned) float, bool, string, struct und union.

Programnummer und Versionnummer werden zur Laufzeit beim **Binder** verwendet und Client und Server zu verbinden.

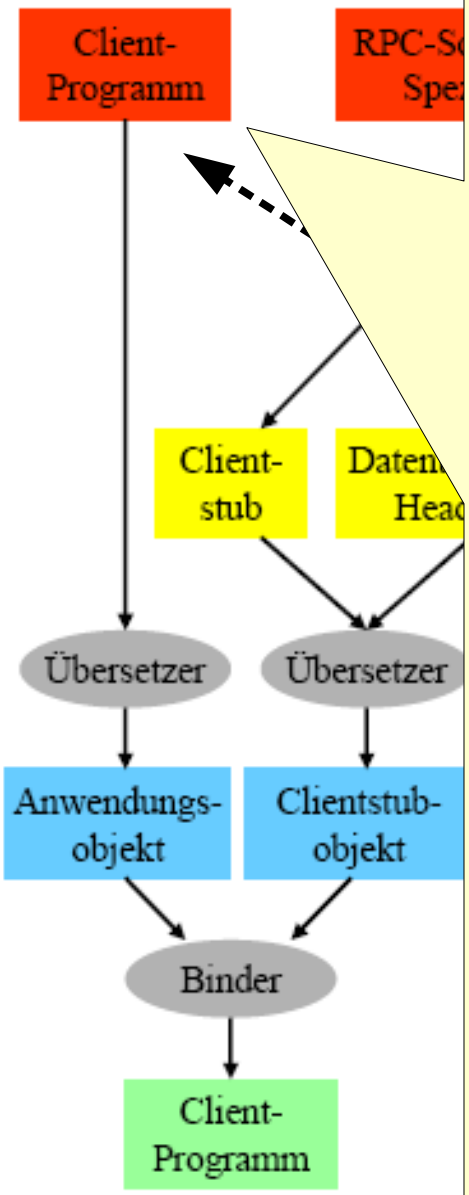
SUN bzw. ONC rpcgen (3)




```
$ cat addiere.h
/*Please do not edit this file. It was generated using
rpcgen. */

#ifndef _ADDIERE_H_RPCGEN
#define _ADDIERE_H_RPCGEN

#include <rpc/rpc.h>

// etwas vereinfacht...
struct add_struct {
        int p1; int p2;
};
typedef struct add_struct add_struct;

#define ADDIERE_TEST 1234567
#define ONE 1

#define addiere 1
extern  int * addiere_1(add_struct *, CLIENT *);
extern  int * addiere_1_svc(add_struct *, struct svc_req *);
extern int addiere_test_1_freeresult (SVCXPRT *, xdrproc_t,
caddr_t);

/* the xdr functions */
extern  bool_t xdr_add_struct (XDR *, add_struct*);
#endif /* !_ADDIERE_H_RPCGEN */
```

Wichtig für den Programmier sind hier die Übersetzungen der Datenstrukturen

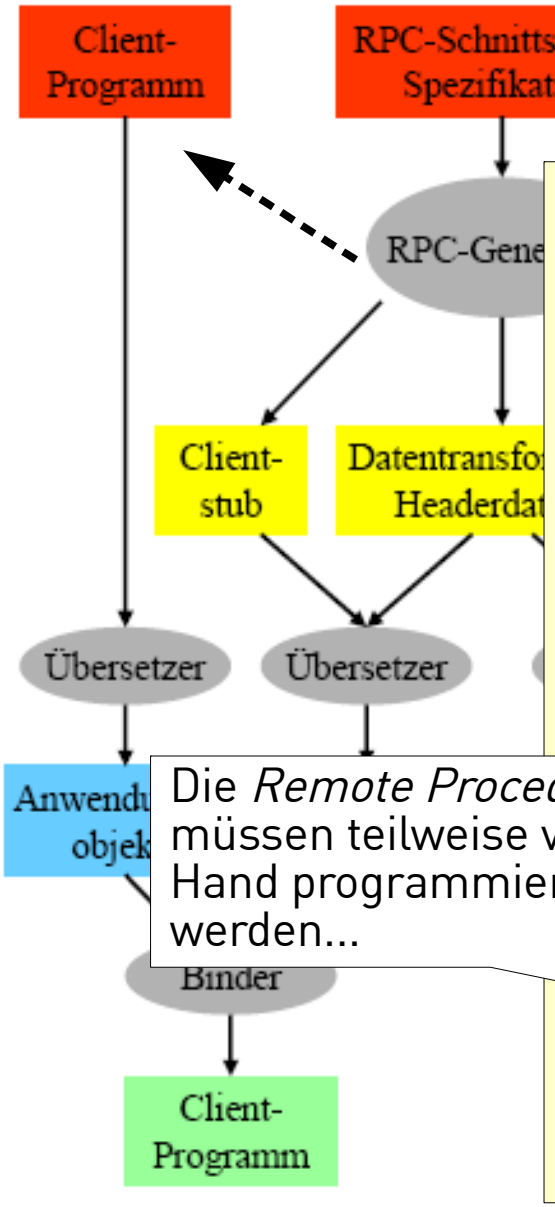
SUN bzw. ONC rpcgen (4)




```
$ cat addiere_client.c # NACH_Ergaenzung
/* This is sample code generated by rpcgen. These are only templates and
you can use them as a guideline for developing your own functions. */

#include "addiere.h"

void addiere_test_1(char *host)
{
        CLIENT *clnt;
        int  *result_1;
        add_struct  addiere_1_arg;

        clnt = clnt_create (host, ADDIERE_TEST, ONE, "udp");
        if (clnt == NULL) {
                clnt_pcreateerror (host);
                exit (1);
        }

        /* my code START */
        addiere_1_arg.p1 = 24; addiere_1_arg.p2 = 18;
        /* my code END */

        result_1 = addiere_1(&addiere_1_arg, clnt);
        if (result_1 == (int *) NULL)
                clnt_perror (clnt, "call failed");
        /* my code START */
        else
                printf("addiere( 24, 18 ) liefert %d\n",
                        *result_1);
        /* my code END */

        clnt_destroy (clnt);
}

/* main ruft addiere_test_1( argv[1] auf ...  */
```

Der Client muss *teilweise* von Hand programmiert werden...

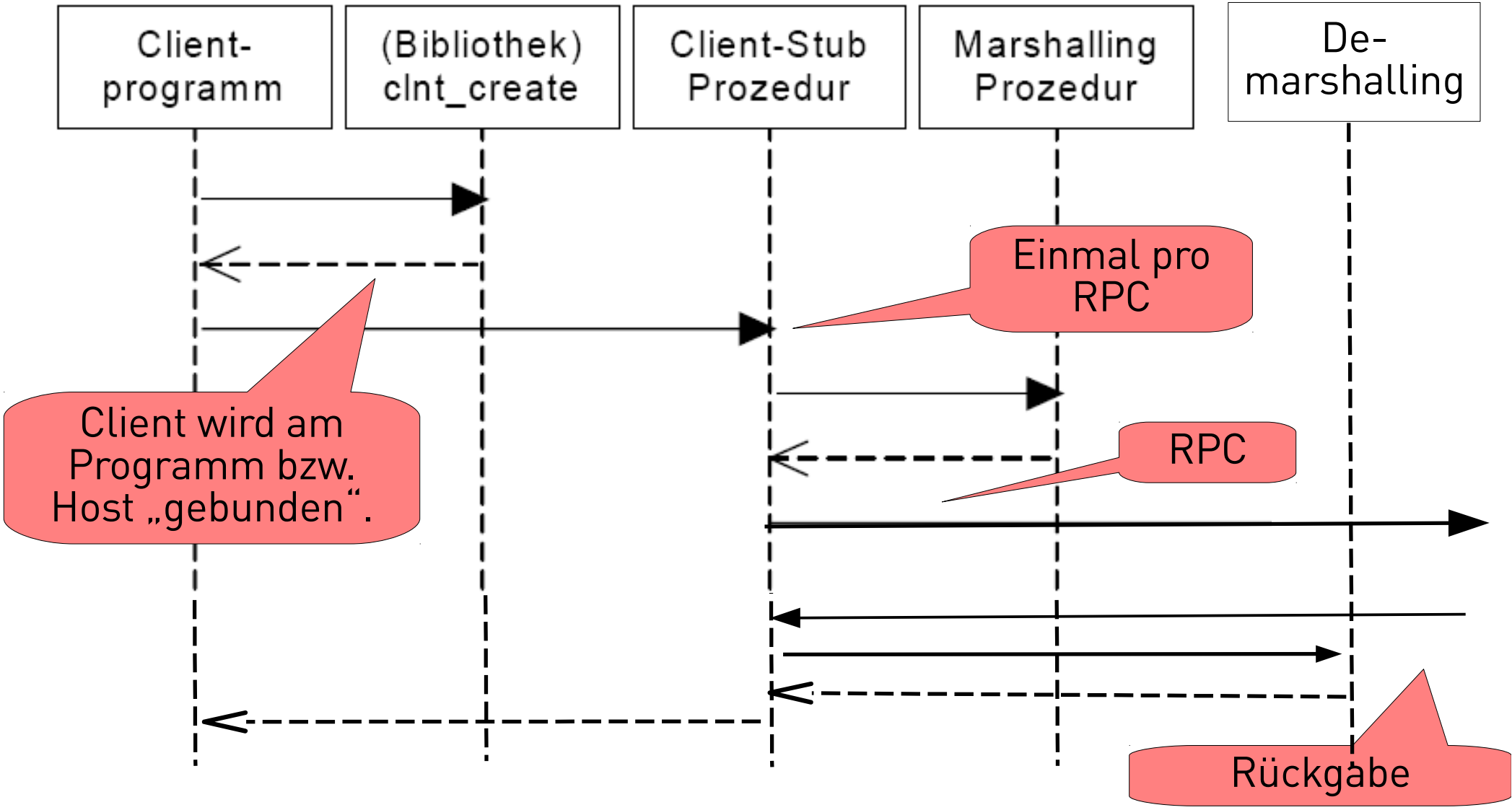
SUN bzw. ONC rpcgen (5)



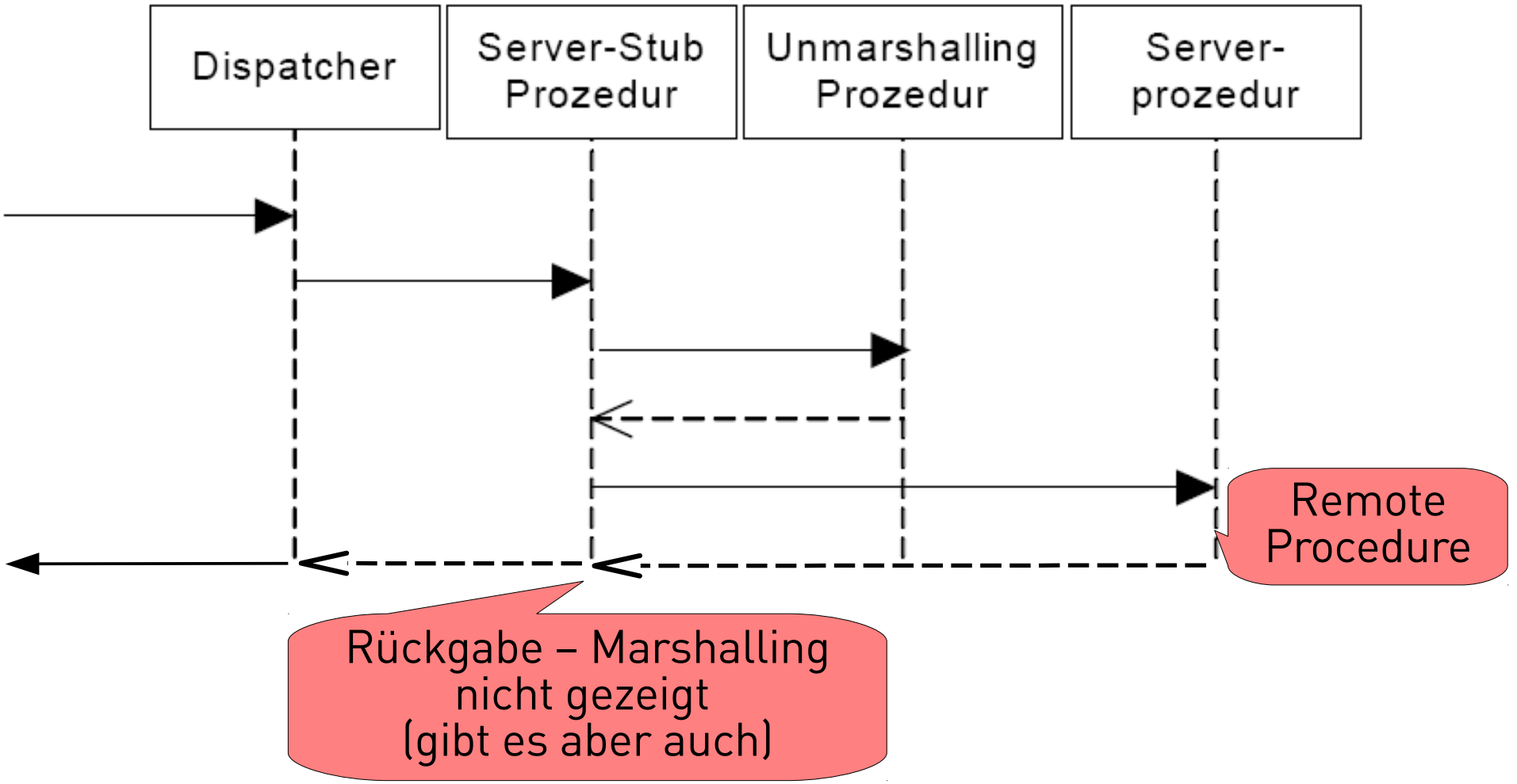
Die *Remote Procedures* müssen teilweise von Hand programmiert werden...

```
$ cat addiere_server.c # NACH_Ergaenzung
/*
 * This is sample code generated by rpcgen.
 * These are only templates and you can use them
 * as a guideline for developing your own functions.
 */

#include "addiere.h"

int *
addiere_1_svc(add_struct *argp, struct svc_req *rqstp)
{
        static int  result;

        /*
         * insert server code here
         */

        /* my Code START */
        result = argp->p1 + argp->p2;
        /* my Code END */

        return &result;
}
```

SUN bzw. ONC RPC zur Laufzeit (Client)



SUN bzw. ONC RPC zur Laufzeit (Server)



Struktur von Kapitel II - Kommunikation

I Einleitung

II Grundlegende Kommunikationsdienste

III Middleware

IV Architekturen & Algorithmen

- A Synchronisierung
- B Konsistenz und Replication
- C Fehlertoleranz

V Beispiele bzw. Dienste

- A Verteilte Dateisysteme
- B Namensdienste

VI Sicherheit & Sicherheitsdienste

VII Zusammenfassung

- A Kommunikationsformen
- B Netzwerk Grundlagen
- C Berkeley Sockets
- D Java Sockets
- E Remote Procedure Calls (RPC)
- F XML-RPC
- G Java RMI

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbi
FACHBEREICH INFORMATIK

XML RPC Grundlagen

- **XML** stellt eines kanonische Austauschformat dar.
- **HTTP** stellt eines überall vorhandene, akzeptierte Transportprotokoll dar.
  - Request = HTTP POST mit XML in „Payload“
  - Response = XML Dokumnt
- Die **Spezifikation** umfasst weniger als 1800 Worte, ist “leichtgewichtig” und einfach zu verstehen sowie mit vielen Beispielen versehen.

Wenige Probleme mit Firewalls

http://www.xmlrpc.com/spec

- Client und Server können verschiedene (heterogene) Programmiersprachen, Betriebs-systeme und Hardware verwenden.
- **Unterstützte Sprachen** (u.a.): C/C++, Java, Perl, Python, Lisp, PHP, Microsoft .NET, Rebol, Tcl...
- **Unterstützten Datenstrukturen:**
  - integer,
  - boolean,
  - string,
  - double,
  - date & time,
  - base64 Binaries,
  - structs (assoziative Arrays),
  - Arrays (Vektoren)

MIME binary-to-text encoding

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbi
FACHBEREICH INFORMATIK

XML - Beispiel

```
<?xml version="1.0" encoding="UTF-8" ?>
<bank>
    <kunde id=1>
        <name>Meier</name>
        <vorname>Emil</vorname>
        <telefon>1234</telefon>
    </kunde>
    <kunde id=2>
        .....
    </kunde>
</bank>
```

XML Kopf, mit XML Version & Zeichensatz

Tags für Datenstrukturierung

Tags mit Attributen

Elemente treten paarweise, geschachtelt auf

Problem: Die Tags sind noch nicht definiert bzw. beschrieben.

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbi
FACHBEREICH INFORMATIK

Beispiel Anfrage

```
POST /RPC2 HTTP/1.0
User-Agent: Frontier/5.1.2 (WinNT)
Host: betty.userland.com
Content-Type: text/xml
Content-length: 181
```

Header

User-Agent, Host, Content-Type & -Length sind Pflicht

```
<?xml version="1.0"?>
<methodCall>
   <methodName>
      examples.getStateName
   </methodName>
   <params>
      <param>
         <value>
            <i4>
               41
            </i4>
         </value>
      </param>
   </params>
</methodCall>
```

Payload

<methodCall> & <methodName> sind Pflicht. Jeweils genau ein mal.

Es kann 0 oder 1 <params> Tag geben. Es kann 0 oder mehr <param> Tags geben.

Beispiel Antwort (1)

```
HTTP/1.1 200 OK
Connection: close
Content-Length: 158
Content-Type: text/xml
Date: Fri, 17 Jul 1998 19:55:08 GMT
Server: UserLand Frontier/5.1.2-WinNT
```

Header

Immer 200, außer bei Fehler *unterhalb* von XML-RPC (für Fehler *in* XMLRPC, s.u.).

```
<?xml version="1.0"?>
<methodResponse>
   <params>
      <param>
        <value>
           <string>
              South Dakota
           </string>
        </value>
      </param>
   </params>
</methodResponse>
```

Payload

<methodResponse> beinhaltet 1 <params> Tag oder 1 <fault> Tag (s.u.).

<params> muss 1 <param> Tag beinhalten, das 1 <value> Tag beinhaltet.

Beispiel Antwort (2) Faults

```
HTTP/1.1 200 OK
Connection: close
Content-Length: 426
Content-Type: text/xml
Date: Fri, 17 Jul 1998 19:55:02 GMT
Server: UserLand Frontier/5.1.2-WinNT
```

Header

```
<?xml version="1.0"?>
<methodResponse>
   <fault>
      <value>
         <struct>
            <member>
               <name>faultCode</name>
               <value><int>4</int></value>
            </member>
            <member>
               <name>faultString</name>
               <value>
                  <string>Too many parameters.</string>
               </value>
            </member>
         </struct>
      </value>
   </fault>
</methodResponse>
```

Payload

fault = **value**
value = ein Paar:
1) **faultCode**
= **int** | **i4**
(eine Zahl)
2) **faultString**
= **string**
(eine Erklärung)

value = skalarer Wert | **struct** | **array**
struct = beliebig viele **member**s
member = **name** **value** ← *Rekursiv!*

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbi
FACHBEREICH INFORMATIK

Apache XML-RPC für Java: Data

Java Type	*XML Tag Name*	*Description*
Integer	*i4, or int*	A 32-bit, signed, and non-null, integer value.
Boolean	*boolean*	A non-null, boolean value (0, or 1).
String	*string*	A string, non-null.
Double	*double*	A signed, non-null, double precision, floating point number. (64 bit)
java.util.Date	*dateTime.iso8601*	A pseudo ISO8601 timestamp, like 19980717T14:08:55. However, compared to a true ISO8601 value, milliseconds, and time zone informations are missing.
byte[]	*base64*	A base64 encoded byte array.
java.util.Map	*struct*	A key value pair. The keys are strings. The values may be any valid data type, including another map.
Object[] java.util.List	*array*	An array of objects. The array elements may be any valid data type, including another array.

Quelle: http://ws.apache.org/xmlrpc/types.html

Apache XML-RPC für Java: Client

```
import java.net.URL;

import org.apache.xmlrpc.client.XmlRpcClient;
import org.apache.xmlrpc.client.XmlRpcClientConfigImpl;

public class CalcClient {
    public static void main(String[] args) throws Exception {

    XmlRpcClientConfigImpl config = new XmlRpcClientConfigImpl();

    config.setServerURL(new URL("http://127.0.0.1:8080/xmlrpc"));
    XmlRpcClient client = new XmlRpcClient();
    client.setConfig(config);

    Object[] params = new Object[]{new Integer(33), new Integer(9)};
    System.out.println("About to get results...(params[0] = " + params[0]
                       + ", params[1] = " + params[1] + ")." );

    Integer result = (Integer) client.execute("Calculator.add", params);
    System.out.println("Add Result = " + result );

    // ...

    }
}
```

S. https://ws.apache.org/xmlrpc/ und https://archive.apache.org/dist/ws/xmlrpc/ bzw. https://svn.apache.org/viewvc/webservices/archive/xmlrpc/trunk/

Hier wird die Adresse des Servers bestimmt.

Hier werden die Parameter vorbereitet

Hier wird das *Remote Procedure* aufgerufen – als *Handler.Method*, wo „Calculator" der *Handler* ist (s.u.).

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbi
FACHBEREICH INFORMATIK

Apache XML-RPC für Java: Server (1)

```
import org.apache.xmlrpc.server.PropertyHandlerMapping;
import org.apache.xmlrpc.server.XmlRpcServer;
import org.apache.xmlrpc.server.XmlRpcServerConfigImpl;
import org.apache.xmlrpc.webserver.WebServer;

public class CalcServer {

  public Integer add(int x, int y) {
    return new Integer(x+y);
  }
    // ...


public static void main (String [] args) {
    try {

      WebServer webServer = new WebServer(8080);

    // ...
```

Die Package org.apache.xmlrpc.webserver enthält die Klasse WebServer, um einen einfachen XML-RPC Server zu bauen.

Die *Remote Procedures* müssen (sollen) nicht in diesem Class sein – es vereinfacht nur die Darstellung.

Hier wird eine WebServer mit Port 8080 erzeugt (aber noch nicht zum Laufen gebracht).

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbi
FACHBEREICH INFORMATIK

Apache XML-RPC für Java: Server (2)

```
    // ...

      XmlRpcServer xmlRpcServer = webServer.getXmlRpcServer();
      PropertyHandlerMapping phm = new PropertyHandlerMapping();

      phm.addHandler( "Calculator", CalcServer.class);
      xmlRpcServer.setHandlerMapping(phm);

     XmlRpcServerConfigImpl serverConfig =
              (XmlRpcServerConfigImpl) xmlRpcServer.getConfig();

      webServer.start();

    } catch (Exception exception) {
       System.err.println("JavaServer: " + exception);
    }
  }
}
```

Hier wird die Server an der *Handler* („Calculator“) gebunden.

Es gibt verschiedene Wege, die *Handler* anzumelden...

Stellt die Default-Konfiguration ein – Danach kann man auch verschiedene *serverConfig.set...* Methoden aufrufen.

Der Server wartet danach auf Port 8080 auf Anfragen. Der läuft als eigenes Thread (main *könnte* noch was anderes machen).

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbi
FACHBEREICH INFORMATIK

Struktur von Kapitel II - Kommunikation

I Einleitung

II Grundlegende Kommunikationsdienste

- A Kommunikationsformen
- B Netzwerk Grundlagen
- C Berkeley Sockets
- D Java Sockets
- E Remote Procedure Calls (RPC)
- F XML-RPC
- G Java RMI

III Middleware

IV Architekturen & Algorithmen

- A Synchronisierung
- B Konsistenz und Replication
- C Fehlertoleranz

V Beispiele bzw. Dienste

- A Verteilte Dateisysteme
- B Namensdienste

VI Sicherheit & Sicherheitsdienste

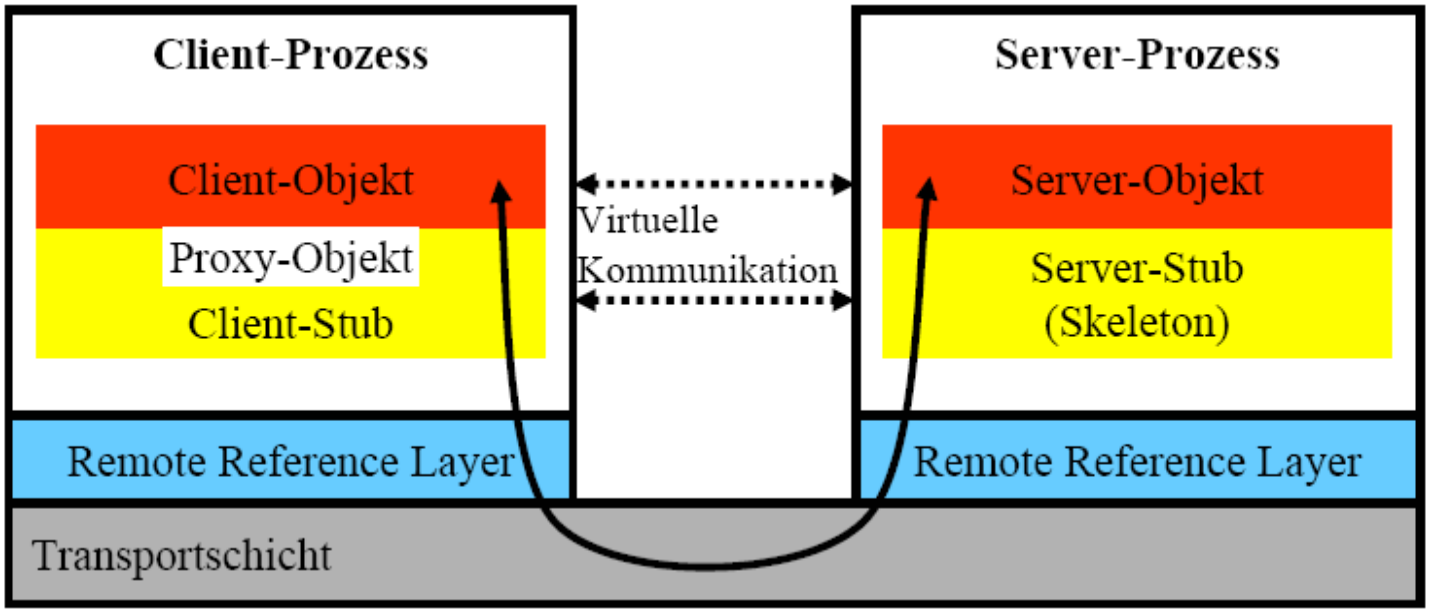
VII Zusammenfassung

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbi
FACHBEREICH INFORMATIK

RMI Grundlagen

Allgemein – nicht nur *Java*-RMI!

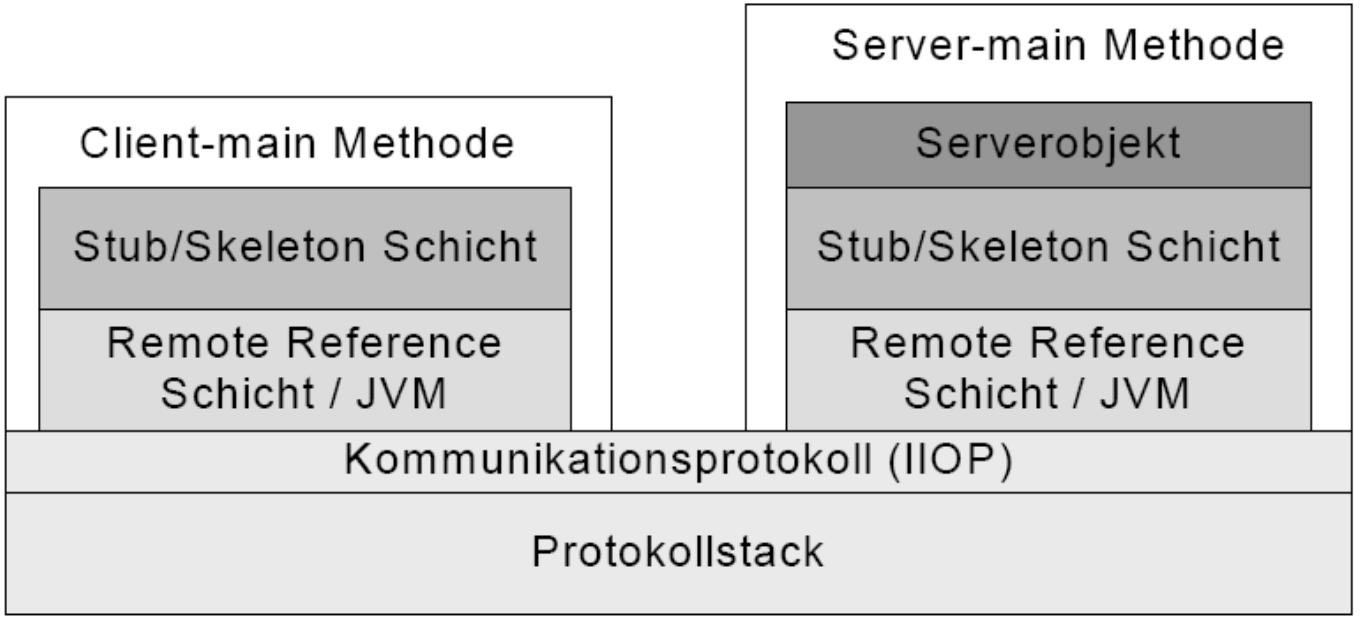
- Können wir es *noch* bequimer für die Programmier machen?



- Nach *Procedures* kam doch *Objekte, Classes, Methoden…*
- Also: RMI = *Remote Method Invocation.*
- Ermöglicht lokalem Objekt, Methoden eines fernen Objekts aufzurufen.
- Das Modell setzt auf dem *Proxy Pattern* auf:
- Auf Clientseite wird ein Stellvertreter-objekt des eigentlichen Serverobjekts eingeführt (Proxy).
- Proxy und Serverobjekt implementieren die gleiche Schnittstelle.

Java RMI

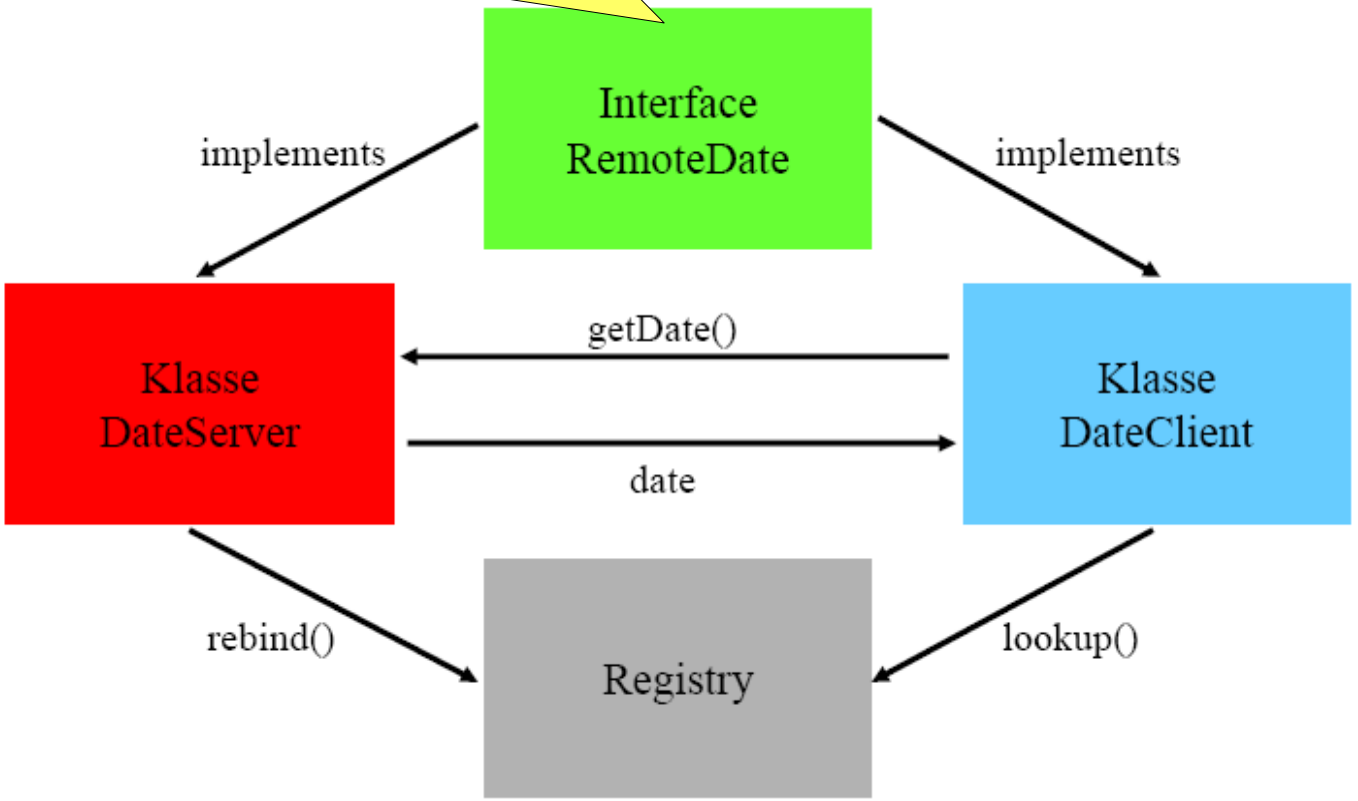
- Java-basierte Implementierung des RMI Kommunikationsmodells.
- Integraler Bestandteil der Java Plattformen.
- Client- und Serverobjekte grundsätzlich in Java formuliert.



- Datentransformation: Objektserialisierung.
- Integrierter Namendienst: RMI Registry (andere Namendienste können ebenfalls verwendet werden).
- Verwendet Sicherheitsmodell der Java Plattform (s.u.).

Beispiel – Date/Time Schnittstelle

```
import java.rmi.*;
import java.util.*;

public interface RemoteDate extends Remote {
  public Date getDate()
                        throws RemoteException;
}
```



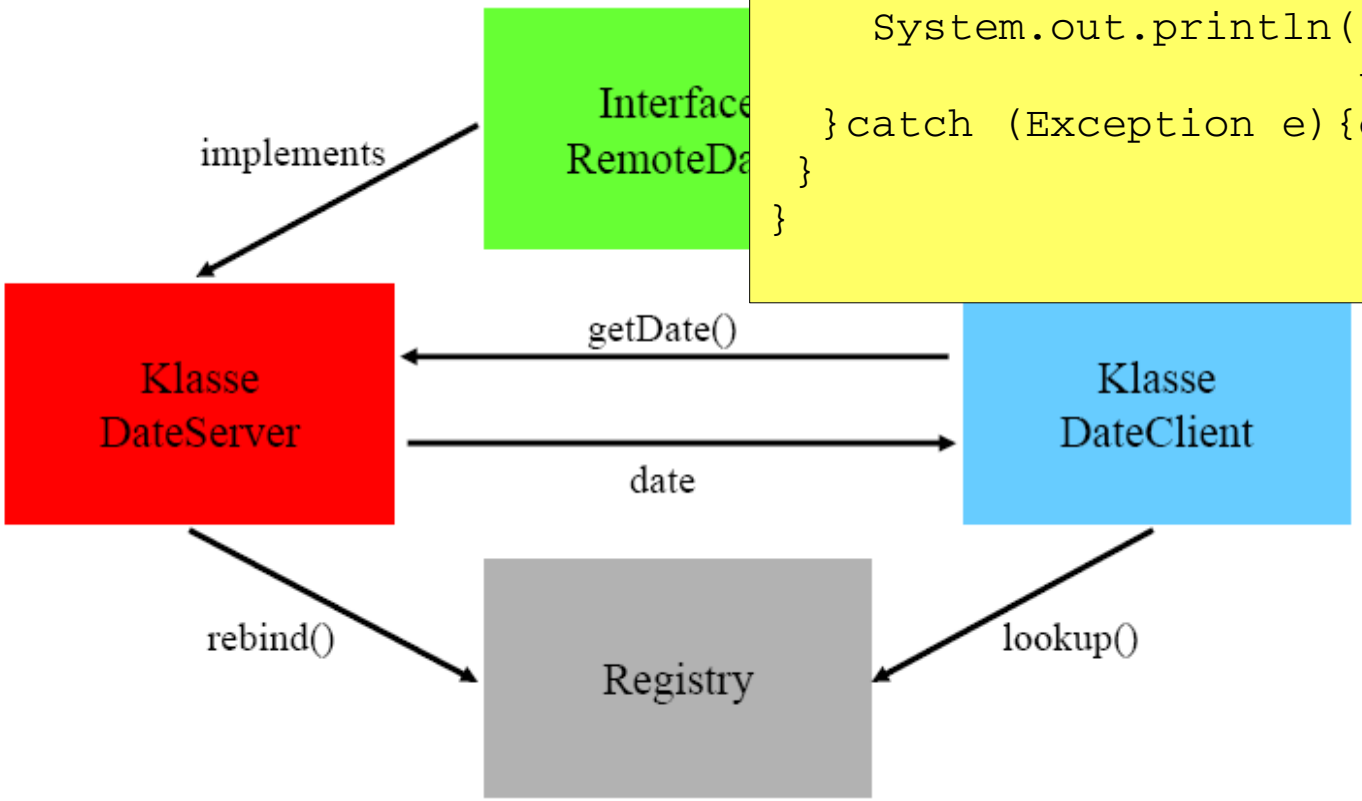
Java RMI - Schnittstellen

- Schnittstellen werden definiert
  - mit Hilfe des Java `interface` Konstrukts und
  - erben die Eigenschaften des `Remote` Interface.
  - Für jede Methode wird eine `RemoteException` definiert.
- Das Interface **`Remote`**
  - zeigt an, dass es sich um ferne Aufrufe handelt,
  - das Interface selbst verfügt über keine Methoden.

- In der Schnittstelle können alle Java-Typen als Parameter verwendet werden.
- Neue Klassen können definiert und ebenfalls als Parameter von Methoden verwendet werden.
- **Wichtig:** Alle Java Typen, die übertragen werden, müssen das `Serializable` Interface implementieren.

Beispiel – Date/Time Client

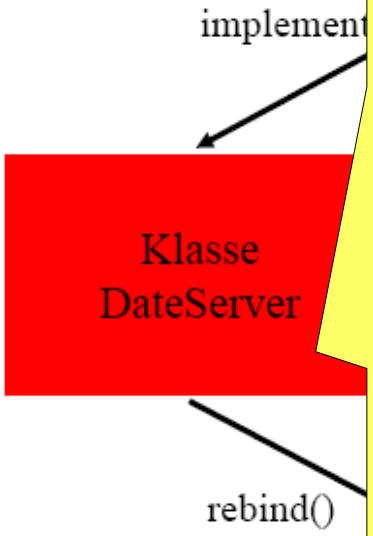
```
import java.rmi.*;
import java.util.*;

public class DateClient {
 public static void main(String[] args) {
  try {
    RemoteDate dateService =
      (RemoteDate)Naming.lookup("DateServer");
    Date date = dateService.getDate();
    System.out.println("Server suggests:"
                       + date);
  }catch (Exception e){e.printStackTrace();}
 }
}
```



Java RMI - Client

- Verschafft sich über die statische Methode `lookup()` der Klasse `Naming` bei der RMI-Registry die Objektreferenz vom Namendienst zur Kontaktierung des Serverobjekts.
- Die Referenz ist Grundlage für die Generierung eines Proxy Objekts (RMI Proxy Pattern!).
- Über den Proxy können alle Methoden des Serverobjekts angesprochen werden.

Beispiel – Date/Time Server




```
import java.util.Date;
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;

public class DateServer extends UnicastRemoteObject
                                    implements RemoteDate {
  private static final long serialVersionUID=4712L;
  public DateServer() throws RemoteException{ super();}
  public static final String REGISTRY_NAME =
                                      DateServer.class.getName();

  public Date getDate(){ return new Date();}

  public static void main(String[] args) {
    int registryPortNumber = 1099;
    try{
      LocateRegistry.createRegistry(registryPortNumber);
      Naming.rebind(REGISTRY_NAME, new DateServer());
      System.out.println("waiting for requests ...");
    }catch(Exception e){e.printStackTrace();}
  }
}
```

Java RMI - Server

- Die Serveranwendung erweitert die Klasse `UnicastRemoteObject`
  - Vererbung der Fähigkeiten zur Kommunikation mit fernen Objekten.
  - Unterstützt die Kommunikation über TCP.
  - Für andere Transportprotokolle muss die Klasse ersetzt werden.
- Die statische Klasse `Naming`
  - Stellt dem Server Hilfsmethoden zur Verfügung wie zum Anmelden von Serverobjekten beim Namendienst, der RMI Registry.
  - Zur Anmeldung wird eine eindeutige Objektreferenz des Serverobjekts im Namendienst bekanntgegeben.
  - Die Objektreferenz enthält die Adresse des Objekts.
- Die `main`() Methode initialisiert das Serviceobjekt und meldet es am Namendienst an.

Java RMI - Registry

- Aufgabe der RMI-Registry ist die Bereitstellung von Referenzen auf Serviceobjekte.
- Die RMI Umgebung generiert bei der Anmeldung eines Serviceobjekts an der RMI Registry eine Objektreferenz.
- Diese enthält die Ortbeschreibung zur Lokalisierung des Serviceobjekts:
  - IP-Adresse
  - Port
  - ObjektID
- Die Objektreferenzen werden unter einem eindeutigen Namen von der Registry registriert.
- Clients holen sich über diesen Namen die Objektreferenz von der Registry und können damit auf das Serviceobjekt zugreifen.

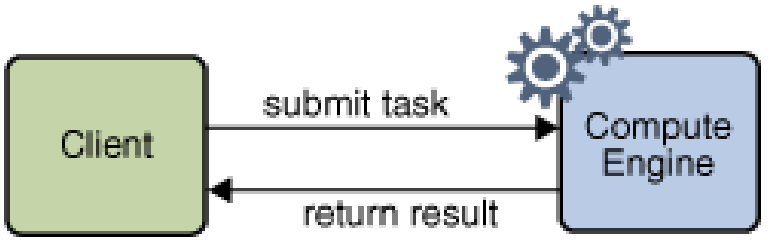
RMI ≠ RPC?!?

- **Beobachtung:** Wir haben noch nichts gemacht, dass wir nicht auch mit RPC hätten machen können...
- **Frage:** Oder?!?

- **Vermutung:** Wir haben ein Methode eines Objektes aufgerufen, also `obj->f( x )`, statt einfach `f(x)`!
- **Frage:** Wie ist das anders als `f( obj_ID, x )`?

- **Vermutung:** Wir haben einen Server aufgerufen, ohne seine IP-Nummer zu kennen!
- **Frage:** Wie ist das anders als RPC mit DNS? Und wir mussten die IP-Nummer des Registry's kennen (bzw ausfinden).

- Das nächste Beispiel zeigt:
  - Wie man (fast) alles *zur Laufzeit* definieren kann – nur eine kleine Basis-Interface wird vorab definiert.
  - Wie *ausführbare Byte-Code über das Netzwerk migrieren* kann.
- Das Beispiel stammt von: http://java.sun.com/docs/books/tutorial/rmi/
- **Aufgabe:**
  Ein *generischer* Compute-Server.



Compute-Beispiel: Interfaces

compute/Compute.java

```
package compute;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Compute
                  extends Remote
{
    <T> T executeTask(Task<T> t)
         throws RemoteException;
}
```

compute/Task.java

```
package compute;

public interface Task<T> {
    T execute();
}
```

Java Generics:
T = Return-Type

Compute Service

Task<T> muss auch serializable sein

Compute-Beispiel: Server

engine/ComputeEngine.java

```
package engine;

import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
import compute.Compute;
import compute.Task;

public class ComputeEngine
             implements Compute {

    public ComputeEngine() {
        super();
    }

    public <T> T executeTask(
                     Task<T> t) {
        return t.execute();
    }

    ...
```

```
...

public static void main(String[] args) {
    if (System.getSecurityManager() == null) {
        System.setSecurityManager(new SecurityManager());
    }
    try {
        String name = "Compute";
        Compute engine = new ComputeEngine();
        Compute stub =
          (Compute) UnicastRemoteObject.exportObject(engine, 0);
        Registry registry = LocateRegistry.getRegistry();
        registry.rebind(name, stub);
        System.out.println("ComputeEngine bound");
    } catch (Exception e) {
        System.err.println("ComputeEngine exception:");
        e.printStackTrace();
    }
  } // end main
} // end class
```

Security Manager muss dabei sein!

Der Server läuft weiter nachdem main() fertig ist...

Etwas andere Anmeldung... 0 = anonymous port

Compute-Beispiel: Client

client/ComputePi.java

```
package client;

import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.math.BigDecimal;
import compute.Compute;

public class ComputePi {
    public static void main(String args[]) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new SecurityManager());
        }
        try {
            String name = "Compute";
            Registry registry = LocateRegistry.getRegistry(args[0]);
            Compute comp = (Compute) registry.lookup(name);
            Pi task = new Pi(Integer.parseInt(args[1]));
            BigDecimal pi = comp.executeTask(task);
            System.out.println(pi);
        } catch (Exception e) {
            System.err.println("ComputePi exception:");
            e.printStackTrace();
        }
    }
}
```

Security Manager muss dabei sein!

args[0]= Name des Servers (!!!)

Code für Pi kommt gleich... (eine Klasse, die Task erweitert)

Compute-Beispiel: Tasks

client/Pi.java

```
package client;

import compute.Task;
import java.io.Serializable;
import java.math.BigDecimal;

public class Pi implements Task<BigDecimal>, Serializable {

    private static final long serialVersionUID = 227L;

    // ...

    /**
     * Calculate pi.
     */
    public BigDecimal execute() {
        return computePi(digits);
    }

    // Compute the value of pi to the specified number of
    // digits after the decimal point.  ...
    public static BigDecimal computePi(int digits) {
        int scale = digits + 5;
        // ... viele Berechnungen...
    }
    // ... noch mehr Berechnungen ...
}
```

Wichtig & notwendig

Version Nummer

Vgl. http://java.sun.com/docs/books/tutorial/rmi/client.html

Compute-Beispiel: Security Policy

security.policy

```
grant codeBase "file:/home/ron/.../src/" {
    permission java.security.AllPermission;
};
```

Local Code = Trusted
Remote Code = Untrusted

Wird wie folgt verwendet:

Zuerst das Registry...

```
> rmiregistry &
> java -Djava.security.policy=security.policy engine/ComputeEngine
ComputeEngine bound

> # in einem anderen Shell...
> java -Djava.security.policy=policy.policy client/ComputePi localhost 42
3.14159265358979323846264338327950288419716 9
```

...dann den Server anfangen...

...dann den Client anfangen.

Vgl. http://java.sun.com/docs/books/tutorial/rmi/running.html

h_da
HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES
fbi
FACHBEREICH INFORMATIK