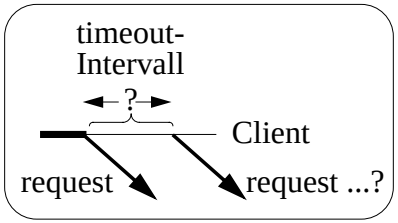
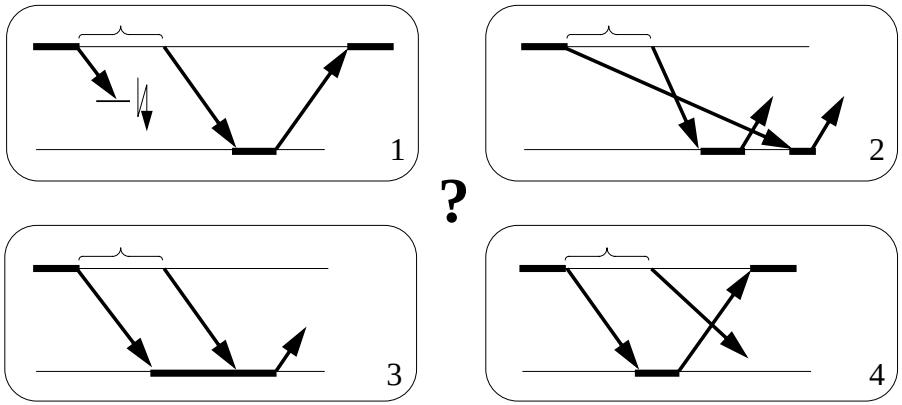
Typische Fehlerursachen: I. Verlorene Request-Nachricht

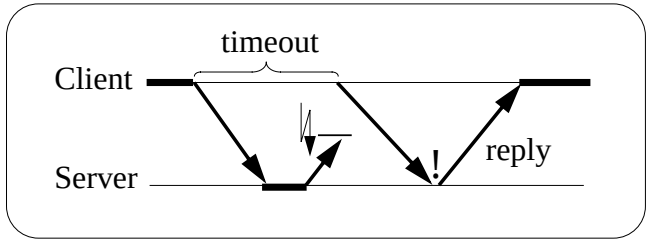
- *Gegenmassnahme*:
  - Nach Ablauf eines Timers ohne Reply die Request-Nachricht erneut senden



- *Probleme*:
  - Wieviele Wiederholungsversuche maximal?
  - Wie gross soll der Timeout sein?
  - Falls die Request-Nachricht gar nicht verloren war, sondern Nachricht oder Server untypisch langsam:
    - Doppelte Request-Nachricht! (Gefährlich bei nicht-idempotenten Operationen!)
    - Server sollte solche Duplikate erkennen. (Wie? Benötigt er dafür einen Zustand? Genügt es, wenn der Client Duplikate als solche kennzeichnet? Genügen Sequenznummern? Zeitmarken?)
    - Würde das Quittieren der Request-Nachricht etwas bringen?



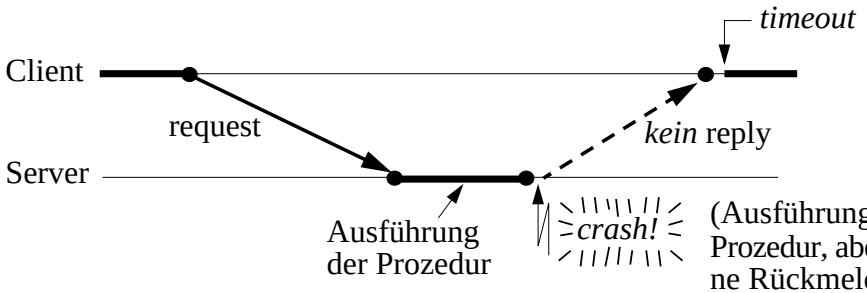
II. Verlorene Reply-Nachricht

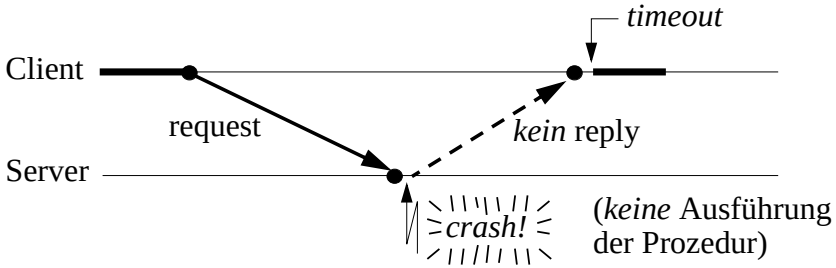


- *Gegenmassnahme 1:* analog zu verlorener Request-Nachricht
  - Also: Anfrage nach Ablauf des Timeouts wiederholen
- *Probleme:*
  - Vielleicht ging aber tatsächlich der Request verloren?
  - Oder der Server war nur langsam und arbeitet noch?
  - Ist aus Sicht des Clients nicht unterscheidbar!

- *Gegenmassnahme 2:*
  - Server könnte eine “Historie” der versendeten Replies halten
    - Falls Server Duplikate erkennt und den Auftrag bereits ausgeführt hat: letztes Reply erneut senden, ohne das Resultat nochmals zu berechnen!
    - Pro Client muss nur das neueste Reply gespeichert werden.
    - Bei vielen Clients u.U. dennoch Speicherprobleme:
      --> Historie nach “einiger” Zeit löschen.
      (Ist in diesem Zusammenhang ein ack eines Reply sinnvoll?)
      Und wenn man ein gelöschtes Reply später dennoch braucht?

III. Server-Crash

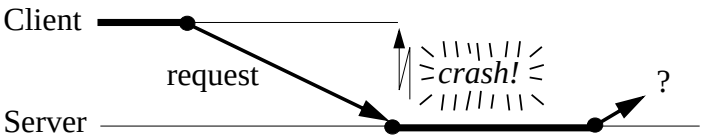




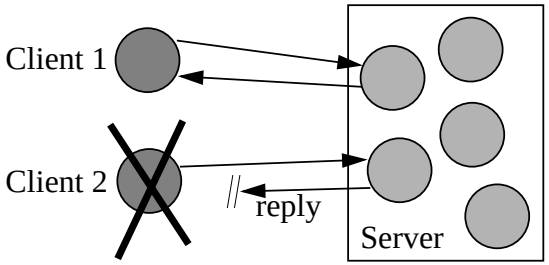
Probleme:

- Wie soll der Client dies unterscheiden?
  - ebenso: Unterschied zu verlorenem request bzw. reply?
  - Sinnhaftigkeit von Gegenmassnahmen hängt ggf. davon ab
  - Client *meint* u.U. zu Unrecht, dass ein Auftrag nicht ausgeführt wurde (--> falsche Sicht des Zustandes!)
- Evtl. Probleme nach einem Server-Restart
  - z.B. “Locks”, die noch bestehen (Gegenmassnahmen?) bzw. allgemein: “verschmutzer” Zustand durch frühere Inkarnation
  - typischerweise ungenügend Information (“Server Amnesie”), um in alte Kommunikationszustände problemlos wieder einzusteigen

IV. Client-Crash



- Reply des Servers wird nicht abgenommen
  - Server wartet z.B. vergeblich auf eine Bestätigung (wie unterscheidet der Server dies von langsamen Clients oder langsamen Nachrichten?)
  - blockiert i.a. Ressourcen beim Server!
- “Orphans” (Waisenkinder) beim Server
  - Prozesse, deren Auftraggeber nicht mehr existiert



Beispiel: Pro RPC-Auftrag eines Client gründet der Server einen eigenen Prozess

- Nach Neustart des Client dürfen alte Replies nicht stören
  - “Antworten aus dem Nichts” (Gegenmassnahme: Epochen-Zähler)
- Nach Restart könnte ein Client versuchen, Orphans zu killen (z.B. durch Benachrichtigung der Server)
  - dadurch bleiben aber u.U. locks etc. bestehen
  - Orphans könnten bereits andere RPCs abgesetzt haben, weitere Prozesse gegründet haben...
- Pessimistischer Ansatz: Server fragt bei laufenden Aufträgen von Zeit zu Zeit und vor wichtigen Operationen beim Client zurück (ob dieser noch existiert)

RPC-Fehlersemantik

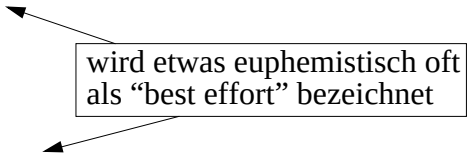
Operationale Sichtweise:

- Wie wird auf (vermeintlich?) nicht eintreffende Requests oder Replies nach einem Timeout und auf wiederholte Requests reagiert?
- Und wie auf gecrashte Server / Clients?

---

1) Maybe-Semantik:

- Keine Wiederholung von Requests
- *Einfach und effizient*
- Keinerlei Erfolgsgarantien --> oft nicht anwendbar
  Mögliche Anwendungsklasse: Auskunftsdienste (noch einmal probieren, wenn keine Antwort kommt)



2) At-least-once-Semantik:

- Hartnäckige Wiederholung von Requests
- Keine Duplikatserkennung (*zustandsloses Protokoll* auf Serverseite)
- Akzeptabel bei idempotenten Operationen (z.B. Lesen einer Datei)

RPC-Fehlersemantik (2)

3) At-most-once-Semantik:

- Erkennen von Duplikaten (Sequenznummern, log-Datei etc.)
- Keine wiederholte Ausführung der Prozedur, sondern ggf. erneutes Senden des Reply
- Geeignet auch für *nicht-idempotente* Operationen
- Kein Ergebnis bei abgestürztem Server

4) Exactly-once-Semantik:

- Wunschtraum?
- Oder geht es zumindest unter der *Voraussetzung*, dass der Server nicht crasht und ein reply letztlich auch durchkommt? (Z.B. durch hartnäckige Wiederholung von Requests?)
- Was ist mit verteilten Transaktionen? (--> Datenbanken! Stichworte: Checkpoint; persistente Datenspeicherung; Recovery...)

- Nochmals: Fehlertransparenz bei RPC?
  - Problem: Client / Server haben u.U. (temporär?) eine inkonsistente Sicht
  - Einige Fehler sind bei gewöhnlichen Prozeduraufrufen nicht möglich
  - Timeout beim Client kann *verschiedene* Ursachen haben (verlorener Request, verlorenes Reply, langsamer Request bzw. Reply, langsamer Server, abgestürzter Server...) --> Fehlermaskierung schwierig
  - Vollständige Transparenz ist kaum erreichbar
  - Hohe Fehlertransparenz = hoher Aufwand

Wirkung der RPC-Fehlersemantik

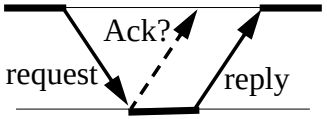
	Fehlerfreier Ablauf	Nachrichten-verluste	Ausfall des Servers
Maybe	Ausführung: 1 Ergebnis: 1	Ausführung: 0/1 Ergebnis: 0	Ausführung: 0/1 Ergebnis: 0
At-least-once	Ausführung: 1 Ergebnis: 1	Ausführung: ≥1 Ergebnis: ≥1	Ausführung: ≥0 Ergebnis: ≥0
At-most-once	Ausführung: 1 Ergebnis: 1	Ausführung: 1 Ergebnis: 1	Ausführung: 0/1 Ergebnis: 0
Exactly-once	Ausführung: 1 Ergebnis: 1	Ausführung: 1 Ergebnis: 1	Ausführung: 1 Ergebnis: 1

May-be ---> At-least-once ---> At-most-once ---> ...
ist zunehmend aufwendiger zu realisieren!

- man begnügt sich daher, falls es der Anwendungsfall gestattet, oft mit einer billigeren aber weniger perfekten Fehlersemantik
- Motto: so billig wie möglich, so „perfekt“ wie nötig

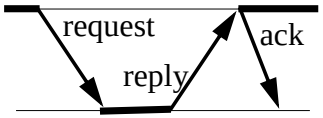
RPC-Protokolle

- *RR-Protokoll* (“Request-Reply”):



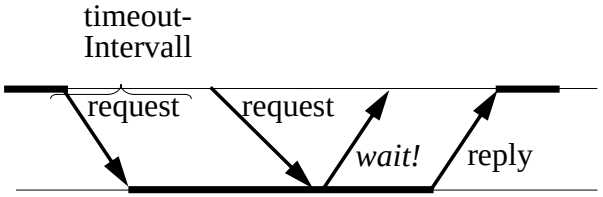
- Reply ist implizite Quittung für Request
- lohnt sich ggf. eine unmittelbare Bestätigung des Request?

- *RRA-Protokoll* (“Request-Reply-Acknowledge”):



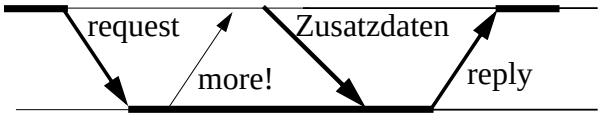
- “pessimistischer” als das RR-Protokoll
- Vorteil: Server kann evtl. gespeicherte Replies frühzeitig löschen (und natürlich Replies bei Ausbleiben des ack wiederholen)

- *Sinnvoll bei langen Aktionen / überlasteten Servern:*



“wait” = Bestätigung eines erkannten Duplikats

- *Parameter-Übertragung „on demand“*



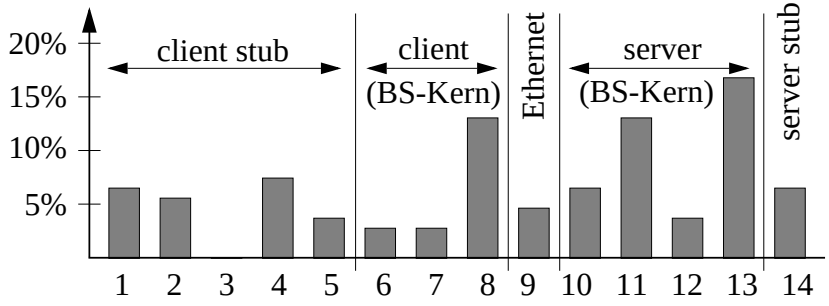
- spart Pufferkapazität
- bessere Flusssteuerung
- Zusatzdaten abhängig vom konkreten Ablauf

- *Weitere RPC-Protokollaspekte:*
  - effiziente Implementierung einer geeigneten (=?) Fehlersemantik
  - geeignete Nutzung des zugrundeliegenden Protokolls (ggf. aus Effizienzgründen eigene Paketisierung der Daten, Flusssteuerung, selektive Wiederholung einzelner Nachrichtenpakete bei Fehlern, eigene Fehlererkennung / Prüfsummen, kryptogr. Verschlüsselung...)

RPC: Effizienz

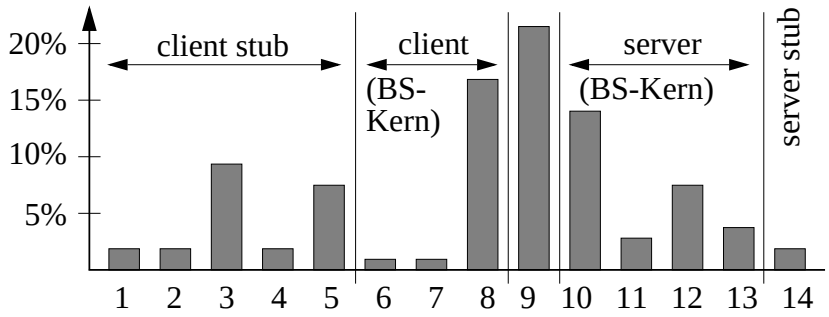
Analyse eines RPC-Protokolls durch Schroeder
(zitiert nach A. Tanenbaum)

a) Null-RPC (Nutznachricht der Länge 0, kein Auftragsbearbeitung):



1. Call stub
2. Get message buffer
3. Marshal parameters
4. Fill in headers
5. Compute UDP checksum
6. Trap to kernel
7. Queue packet for transmission
8. Move packet to controller over the bus
9. Ethernet transmission time
10. Get packet from controller
11. Interrupt service routine
12. Compute UDP checksum
13. Context switch to user space
14. Server stub code

b) 1440 Byte Nutznachricht (ebenfalls kein Auftragsbearbeitung):

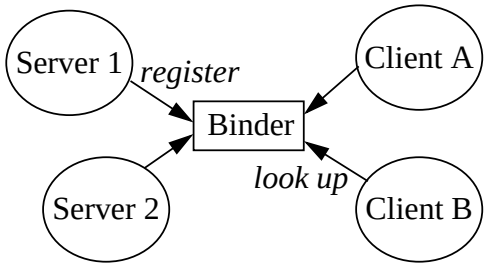


- Eigentliche Übertragung kostet relativ wenig
- Rechenoverhead (Prüfsummen, Header etc.) keineswegs vernachlässigbar
- Bei kurzen Nachrichten: Kontextwechsel zw. Anwendung und BS wichtig
- Mehrfaches Kopieren kostet viel

RPC: Binding

- Problem: Wie werden Client und Server “gematcht”?
  - Verschiedene Rechner und i.a. verschiedene Lebenszyklen --> kein gemeinsames Übersetzen / statisches Binden (fehlende gem. Umgebung)

Dynamisches Binden mittels eines “Binders”



- Server (-stub) gibt den Namen etc. seines Services (RPC-Routine) dem Binder bekannt
  - “*register*”; “exportieren” der RPC-Schnittstelle (Typen der Parameter...)
  - ggf. auch wieder abmelden
- Client erfragt beim Binder die Adresse eines geeigneten Servers
  - “*look up*”; “importieren” der RPC-Schnittstelle

oft auch “registry” oder “look-up service” genannt

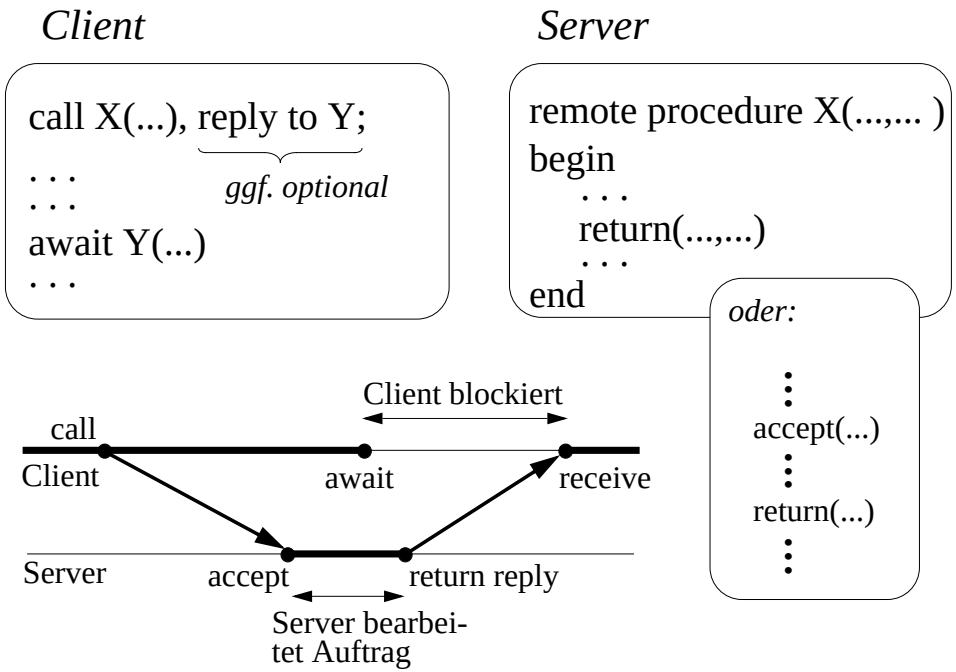
- *Vorteile*: im Prinzip kann Binder
  - mehrere Server für den gleichen Service registrieren (--> Fehlertoleranz; Lastausgleich)
  - Autorisierung etc. überprüfen
  - durch Polling der Server die Existenz eines Services testen
  - verschiedene Versionen eines Dienstes verwalten

dann eher “Trader” oder “Broker”

- *Probleme*:
  - zentraler Binder ist ein potentieller Engpass (Binding-Service geeignet verteilen? Konsistenz!)
  - dynamisches Binden kostet Ausführungszeit

Asynchroner RPC: “Remote Service Invocation”

- auftragsorientiert --> Antwortverpflichtung



- Parallelverarbeitung von Client und Server möglich, solange Client noch nicht auf Resultat angewiesen

Future-Variablen

- Zuordnung Auftrag / Ergebnisempfang bei der asynchron-auftragsorientierten Kommunikation?
  - unterschiedliche Ausprägung auf Sprachebene möglich
  - “await” könnte z.B. einen bei “call” zurückgelieferten “handle” als Parameter erhalten (also z.B. Y = call X(...); ... await (Y);)
  - ggf. könnte die Antwort auch asynchron in einem eigens dafür vorgesehenen Anweisungsblock (vgl. Interrupt-Routine) empfangen werden

- Spracheinbettung evtl. auch durch “Future-Variablen”
  - Future-Variable = handle, der wie ein Funktionsergebnis in Ausdrücke eingesetzt werden kann
  - Auswertung der Future-Variable erst, wenn unbedingt nötig
  - Blockade nur dann, falls Inhalt bei Auswertung noch nicht feststeht
  - Beispiel:

```
FUTURE future: integer;
some_value: integer;
…
future = RSI_call(…);
…
some_value = 4711;
print(some_value + future);
```

Die Socket-Programmierschnittstelle

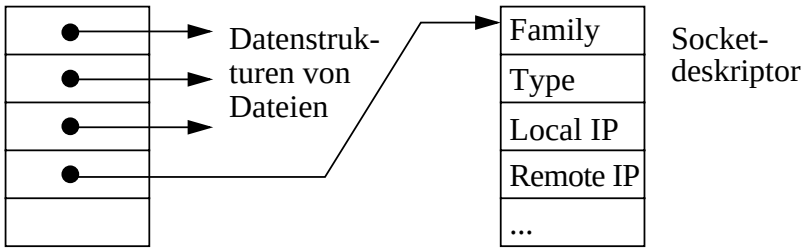
- Zu TCP (bzw. UDP) gibt es keine festgelegten “APIs”
- Bei UNIX ist dafür entstanden: “sockets” als Zugangspunkte zum Transportsystem
  - etwas modernere Alternative: TLI (Transport Layer Interface)
- Semantik eines sockets: analog zu Datei-Ein/Ausgabe
  - ist insbesondere bidirektional (“schreiben” und “lesen”)
  - ein socket kann aber auch mit mehreren Prozessen verbunden sein
- Programmiersprachliche Einbindung (typw. in C)
  - sockets werden wie Variablen behandelt (können Namen bekommen)
  - Beispiel in C (Erzeugen eines sockets):

```
int s;
s = socket(int PF_INET, int SOCK_STREAM, 0);
```

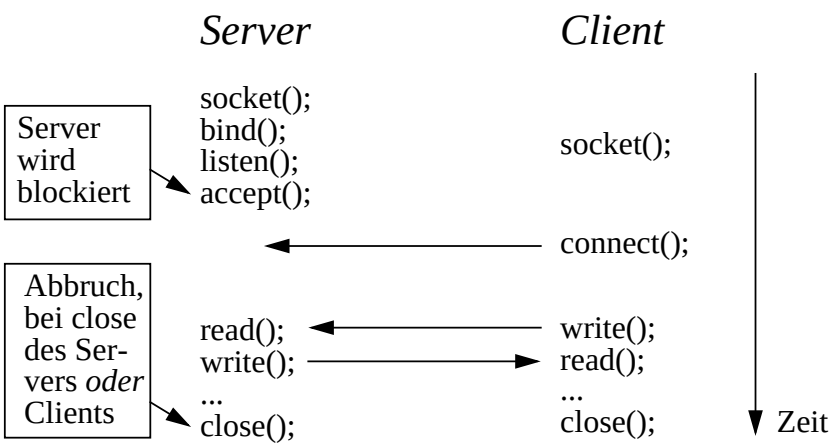
“Family”: Internet oder nur lokale Domäne

“Type”:Angabe, ob TCP verwendet (“stream”); oder UDP (“datagram”)

- Bibliotheksfunktion “socket” erzeugt einen Deskriptor
  - wird innerhalb der Filedeskriptortabelle des Prozesses angelegt
  - Datenstruktur wird allerdings erst mit einem nachfolgenden “bind”-Aufruf mit Werten gefüllt (binden der Adressinformation aus Host-Adresse und einer “bekannten” lokaler Portnummer an den socket)



Client-Server mit Sockets (Prinzip)



- Voraussetzung: Client “kennt” die IP-Adresse des Servers sowie die Portnummer (des Dienstes)
  - muss beim connect angegeben werden
- Mit “listen” richtet der Server eine Warteschlange für Client-connect-Anforderungen ein
  - Auszug aus der Beschreibung: *“If a connection request arrives with the queue full, tcp will retry the connection. If the backlog is not cleared by the time the tcp times out, the connect will fail”*
- Accept / connect implementieren ein “Rendezvous”
  - mittels des 3-fach-Handshake von TCP
  - bei “connect” muss der Server bereits listen / accept ausgeführt haber
- Rückgabewerte von read bzw. write: Anzahl der tatsächlich gesendeten / empfangenen Bytes
- Varianten: Es gibt ein select, ein nicht-blockierendes accept etc., vgl. dazu die UNIX-Bibliothek (“man”...)

Ein Socket-Beispiel in C

- Verwendung von sockets in C erfordert u.a.
  - Header-Dateien mit C-Strukturen, Konstanten etc.
  - Programmcode zum Anlegen, Füllen etc. von Strukturen
  - Fehlerabfrage und Behandlung

- Socket-Programmierung ist ziemlich “low level”
  - umständlich, fehleranfällig bei der Programmierung
  - aber dicht am Netz und dadurch ggf. manchmal von Vorteil (vgl. Assembler-Programmierung)

- Zunächst der Quellcode für den Client:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

#define PORT 4711
#define BUF_SIZE 1024

main(argc,argv)
int   argc;
char *argv[];
{
    int                 sock, run;
    char                buf[BUF_SIZE];
    struct sockaddr_in  server;
    struct hostent     *hp;
    if(argc != 2)
    {
        fprintf(stderr,”usage: client
                       <hostname>\n”);
        exit(2);
    }
```


Socket-Beispiel: Client

```
    /* create socket */
    sock = socket(AF_INET,SOCK_STREAM,0);
    if(sock < 0)
    {
        perror(“open stream socket”);
        exit(1);
    }
    server.sin_family = AF_INET;
    /* get internet address of host specified by command line */
    hp = gethostbyname(argv[1]);
    if(hp == NULL)
    {
       fprintf(stderr,”%s unknown host.\n”,argv[1]);
       exit(2);
    }
    /* copies the internet address to server address */
    bcopy(hp->h_addr, &server.sin_addr, hp->h_length);
    /* set port */
    server.sin_port = PORT;
    /* open connection */
    if(connect(sock,&server,sizeof(struct sockaddr_in)) < 0)
    {
       perror(“connecting stream socket”);
       exit(1);
    }
    /* read input from stdin */
    while(run=read(0,buf,BUF_SIZE))
    {
       if(run<0)
       {
          perror(“error reading from stdin”);
          exit(1);
       }
       /* write buffer to stream socket */
       if(write(sock,buf,run) < 0)
       {
          perror(“writing on stream socket”);
          exit(1);
       }
    }
    close(sock);
}
```

Socket-Beispiel: Server

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

#define PORT 4711                    /* random port number */
#define MAX_QUEUE 1
#define BUF_SIZE 1024

main()
{
    int sock_1,sock_2;     /* file descriptors for sockets */
    int rec_value, length;
    char buf[BUF_SIZE];
    struct sockaddr_in server;

    /* create stream socket in internet domain*/
    sock_1 = socket(AF_INET,SOCK_STREAM,0);
    if (sock_1 < 0)
    {
       perror("open stream socket");
       exit(1);
    }
    /* build address in internet domain */
    server.sin_family = AF_INET;
    /* everyone is allowed to connet to server */
    server.sin_addr.s_addr = INADDR_ANY;
    server.sin_port = PORT;
    /* bind socket */
    if(bind(sock_1,&server,sizeof(struct sockaddr_in)))
    {
       perror("bind socket to server_addr");
       exit(1);
    }
```


Socket-Beispiel: Server (2)

```
    listen(sock_1,MAX_QUEUE);
    /* start accepting connection */
    sock_2 = accept(sock_1,0,0);
    if(sock_2 < 0)
    {
       perror("accept");
       exit(1);
    }
    /* read from sock_2 */
    while (rec_value=read(sock_2,buf,BUF_SIZE))
    {
       if(rec_value<0)
       {
          perror("reading stream message");
          exit(1);
       }
       else
          write(1,buf,rec_value);
    }
    printf("Ending connection.\n");
    close(sock_1); close(sock_2);
}
```

- Sinnvolle praktische Übungen:

1) Beispiel genau studieren; Semantik der socket-Operationen etc. nachlesen (Online-Dokumentation von UNIX oder Bücher)

2) Varianten und andere Beispiele implementieren, z.B.:

- Server, der zwei Zahlen addiert und Ergebnis zurücksendet
- Produzent / Konsument mit dazwischenliegendem Pufferprozess (unter Vermeidung von Blockaden bei vollem Puffer)
- Server, der mehrere Clients gleichzeitig bedienen kann
- Chat- bzw. Konferenzserver analog zu IRC
- Trader, der geeignete Clients und Server zusammenbringt
- Messung des Durchsatzes im LAN; Nachrichtenlängen in mehreren Experimenten jeweils verdoppeln