Kommunikationsmodelle

In diesem Abschnitt werden Programme und Prozesse, die Informationen austauschen betrachtet, nicht mehr Rechner bzw. Netzwerkkarten.

Wir werden Begriffe, wie

- Sender und Empfänger,
- Erzeuger und Verbraucher,
- Klient (client) und Server,
- Gleichberechtigte Partner (peer)
- FIFOS,
- Sockets und
- RPC
- Multicasts

kennen lernen.

Inhalt

1.Das Client-Server Modell....6
2.Adressierung....9
2.1.Direkte Adressierung....9
2.2.Indirekte Adressierung....10
2.2.1.Mailboxen....10
2.2.2.Ports....22
3.Blockierung....26
3.1.Synchrone Kommunikation....26
3.2.Asynchrone Kommunikation....28
4.Pufferung....30
4.1.Ungepufferte Kommunikation....30
4.2.Gepufferte Kommunikation....31
5.Kommunikationsformen....32

5.1.Meldungsorientierte Kommunikation....................32
5.2.Auftragsorientierte Kommunikation....................34
6.Zuverlässigkeit auftragsorientierter Kommunikation....................36
7.Socket Kommunikation....................45
7.1.Konzept....................46
7.2.Fallbeispiele....................62
7.2.1.Echoserver....................62
7.2.2. Zeitserver....................73
7.2.3.Portscanner....................75
7.3.Datagramm Kommunikation....................77
8.Remote Procedure Call....................81
8.1.Funktionsweise des RPC:....................84
8.2.Parameterübergabe....................86
8.2.1.Call-by-Value Parameter....................87
8.2.2.Call-By-Reference Parameter....................87
8.2.3.Transfersyntax....................88

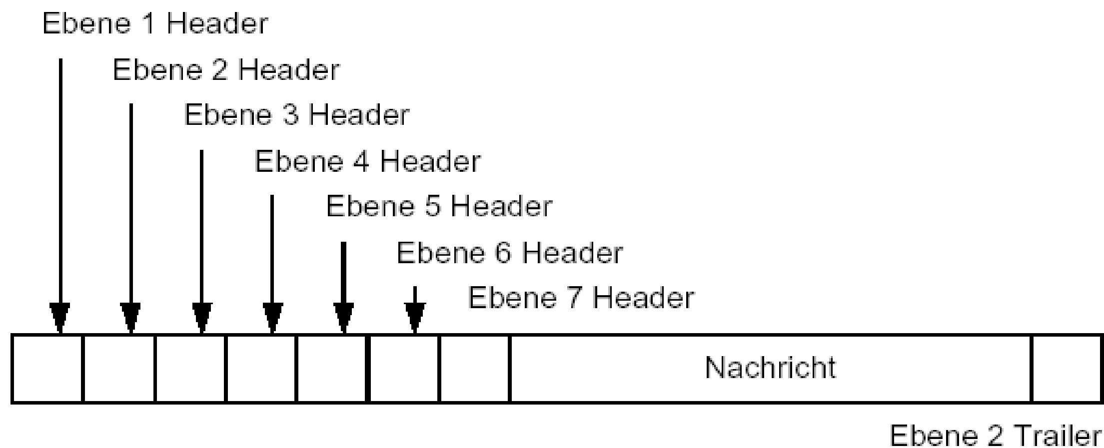
8.3.Beschreibung der RPC-Schnittstellen....89
8.4.Der Bindevorgang beim RPC....91
8.5.Implementierungsaspekte....93
8.5.1.Kritischer Pfad des RPC....95
8.5.2.Prozessverwaltung beim Server....96
8.6.Fallbeispiel RPC (Beispiel des SUN RPC)....98
8.6.1.Grundsätzliche Vorgehensweise....98
8.6.2.Beispiel addiere(a,b)....100
9.Gruppenkommunikation....123
9.1.Entwurfskriterien....125
9.1.1.Wahl der Gruppenform....125
9.1.2.Primitive für die Gruppenverwaltung....125
9.2.Semantik der Gruppenkommunikation....126
9.2.1.Zuverlässigkeitsgrade....126
9.2.2.Ordnungsgrade....127
9.3.Implementierung von Gruppenkommunikation....130

9.3.1.Implementierungsaspekte zur Effizienz....131
9.3.2.Implementierungsaspekte zur Zuverlässigkeit....131
9.4.Fallbeispiel – Chat 135
10.Messaging Systeme....141
10.1.Kommunikationsbeziehungen....142
10.2.Kommunikationsmodelle....142
10.3.Beispiele....143

1. Das Client-Server Modell

Verteilte Systeme können auf der Basis des OSI Schichtenmodells implementiert werden.

Dabei muss **jede Nachricht** vom **Sender pro Schicht verpackt** und vom **Empfänger pro Schicht entpackt** werden.



In verteilten Anwendungen, die über ein Weitverkehrsnetz ablaufen, fällt dieser Verwaltungsaufwand nicht ins Gewicht.

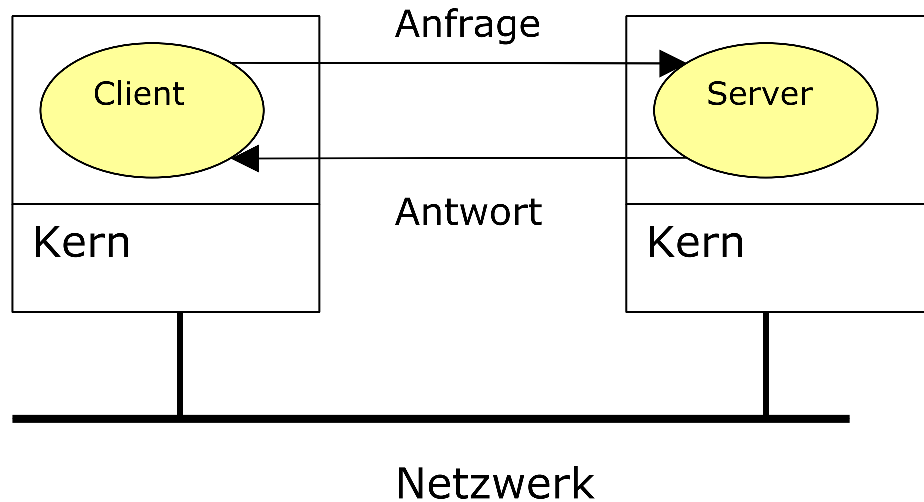
Für **verteilte Systeme**, die innerhalb eines **LAN** oder eines Hochgeschwindigkeitsnetzes (Cluster-Computing) ablaufen, ist dies **nicht akzeptierbar**.

Hierfür wurde das **Client-Server Modell** entwickelt.

> Ein verteiltes System ist hier eine Menge kooperierender Prozesse (Server), die den Benutzern (Clients) Dienste bereitstellen.

In Client-Server Systemen wird meist ein **Anfrage/Antwortprotokoll** verwendet:

> Ein Client-Prozess sendet eine Anfragenachricht, in der er einen bestimmter Dienst nachfragt an einen Server. Der Server erfüllt den Dienst, in dem er die nachgefragten Daten (oder eine Fehlermeldung) zurück zum Client sendet.



Diese Art der Kommunikation ist effektiver, da nur die Protokollschichten 1, 2 und 5 erforderlich sind. Da kein Routing nötig ist und keine Sessionverwaltung gebraucht wird ergibt sich folgendes Bild:

Schicht	
7 (Anwendung)	
6 (Darstellung)	
5 (Sitzung)	Anfrage/Antwort
4 (Transport)	
3 (Vermittlung)	
2 (Sicherung)	Verbindungsschicht
1 (Übertragung)	Bitübertragung

Wegen der einfachen Struktur können die Kommunikationsdienste, die der Kern anbieten muss auf zwei **elementare Operationen** reduziert werden:

- **send**(dest, &mptr)
- **receive**(addr, &mptr)
  dest identifiziert den Empfänger
  mptr ist die Nachricht
  addr ist Adresse, die Empfänger auf Nachrichten überprüft

Im Folgenden werden **unterschiedliche Varianten** der Realisierung von Client-Server Systemen diskutiert.

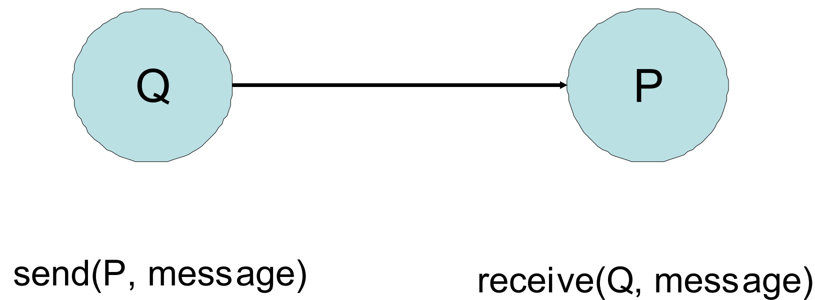
2. Adressierung

Adressierung bezeichnet die Art des Zugriffs eines Senders auf einen Empfänger.

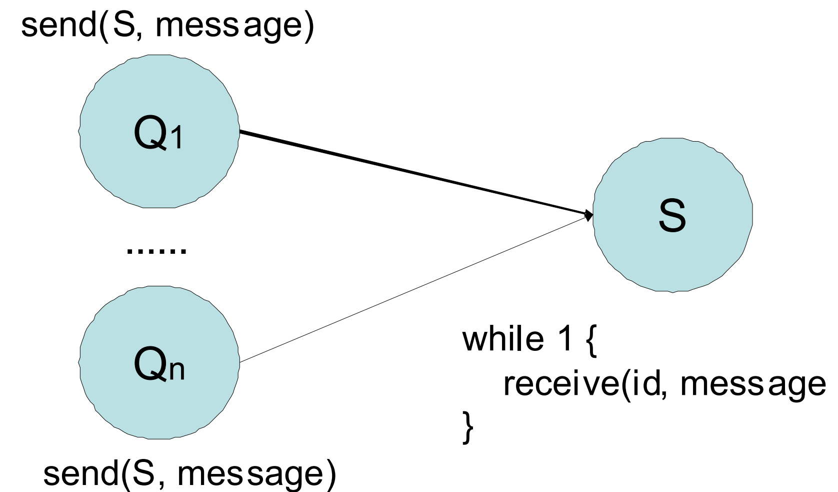
2.1. Direkte Adressierung

Der Sender spricht den Empfänger **direkt** und im Allgemeinen fest an. Man unterscheidet:

symmetrisch



asymmetrisch



Der Server kann so mehrere Klienten bedienen.

2.2.Indirekte Adressierung

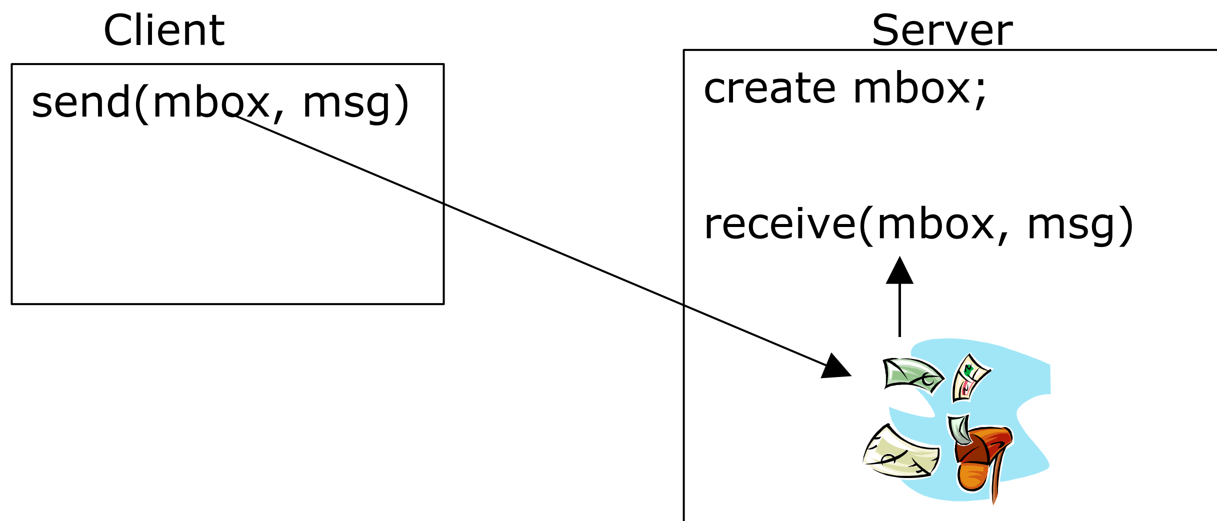
Hier spricht der Sender seinen Empfänger nicht direkt, sondern über eine **Empfangsstelle** an. Unterschieden werden können folgende Arten.

2.2.1.Mailboxen

Durch global bekannte FIFO-Puffer kann eine Mailbox realisiert werden, bei der ein Empfänger eine Nachricht in einen Puffer schreibt, der Empfänger liest in der Reihenfolge des Einstellen.

Es gibt auch Mailboxen für mehrere Sender bzw. Empfänger gleichzeitig.

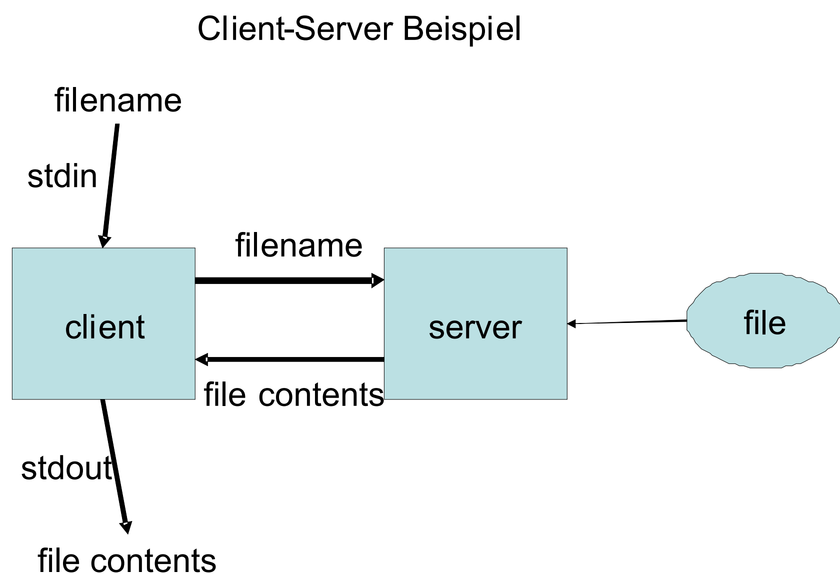
Das folgende Schaubild verdeutlicht die Mailbox-Kommunikation:



Fallbeispiel: FIFOs in Unix

An einem einfachen Beispiel (**Client-Server cat**) wird die Mailbox-Kommunikation verdeutlicht:

> eine Client-Server Anwendung, bei der ein Client einen Dateinamen von stdin einliest und den Namen auf einen Kommunikationskanal schreibt. Der Server liest den Dateinamen vom Kommunikationskanal, öffnet die Datei, liest den Inhalt und schreibt ihn über einen Kommunikationskanal zurück. Der Client erwartet den Inhalt der Datei vom Kommunikationskanal (FIFO) und gibt ihn über stdout aus.



FIFO ist die Abkürzung für First In, First Out. Somit ist die Abarbeitungsreihenfolge von Daten so wie bei Pipes - ein Einwegaustausch, bei dem das erste Byte, das geschrieben wird, auch gelesen wird.

Anders als bei Pipes gibt es bei FIFOs **Namen**, die die Struktur kennzeichnen und somit unabhängigen Prozessen zur Verfügung stehen. Deshalb nennt man FIFOs auch "**named pipes**". Dieser FIFO ist der „Briefkasten, über den die Kommunikation läuft. Im Beispiel wird es verwendet, um in 2 Richtungen zu kommunizieren.

Ein FIFO wird erzeugt durch den Systemaufruf:

```
int mknod(char *pathname, int mode, int dev);
```

Dabei ist ***pathname***, ein normaler **Dateiname** von Unix, der Name des FIFO. Das Argument ***mode*** definiert den **Zugriff** (read, write mit Rechten für owner, group, others). mode wird logisch OR verknüpft mit dem Flag `S_IFIFO` (aus `<sys/stat.h>`), um auszudrücken, dass *mknode* einen FIFO erzeugen soll. *dev* wird für FIFOs ignoriert (relevant für andere Objekte).

FIFOs können auch durch das Unix Kommando *mknod* erzeugt werden:

```
/etc/mknod name p
```

Wenn ein FIFO erzeugt ist, muss es **explizite** zum Lesen oder Schreiben **geöffnet** werden (`open, read, fopen, fread`).

Für FIFOs (und Pipes) gelten folgenden Regeln für Lese- und Schreiboperationen:

- Wenn durch `read` weniger Daten angefordert werden, als im FIFO vorhanden sind, liefert `read` nur die angeforderte Datenmenge zurück. Der restliche Inhalt des FIFO kann durch wiederholte read Kommandos ausgelesen werden.
- Wenn ein Prozess bei einer `read` Operation mehr Daten anfragt, als im FIFO vorhanden sind, so wird nur die vorhandene Datenmenge zurück geliefert. Der Leseprozess muss berücksichtigen, dass u.U. weniger Daten geliefert werden, als angefordert wurden.
- Wenn der **FIFO leer** ist und Daten angefordert werden (`read`) und zudem **kein Prozess** den FIFO zum Schreiben geöffnet hat, wird 0 zurückgeliefert; dies bedeutet dann **EOF**.
- Wenn ein Prozess weniger Daten in den FIFO schreiben (`write`) will, als der FIFO an Kapazität zur Verfügung stellt, so ist die write Operation **atomar**. D.h. Schreiben zwei Prozesse gleichzeitig in einen FIFO, so werden zuerst die Daten des ersten Prozesses, dann die Daten des zweiten Prozesses (oder umgekehrt) geschrieben; ein **Mischen** der Daten beider Prozesse ist somit **verhindert**.
  Wenn jedoch nicht genug Platz im FIFO ist, kann Mischen nicht ausgeschlossen werden!
- Schreibt ein Prozess in einen FIFO, aber es existiert kein Prozess, der den FIFO zum Lesen geöffnet hat, dann wird das Signal `SIGPIPE` erzeugt und `write` gibt 0 zurück. Ist kein Signalhandler aktiv (Normalfall) dann terminiert der Schreibprozess.

Das Verhalten von `open` und `read` bei FIFOs (und Pipes) ist durch das Flag O_NDELAY (=no delay) bestimmt.

Bedingung	normal	O_NDELAY gesetzt (durch fcntl)
open FIFO, read-only, kein Prozess hat FIFO zum Schreiben offen	warten bis Prozess FIFO zum Schreiben geöffnet hat	return ohne Warten, kein Fehler
open FIFO, write-only, kein Prozess hat FIFO zum Lesen offen	warten bis Prozess FIFO zum Lesen geöffnet hat	return ohne Warten, Fehler (errno=ENXIO)
read FIFO, keine Daten	warten bis Daten in FIFO oder kein Prozess mehr vorhanden, der FIFO zum Schreiben offen hat (return 0); ansonsten return gelesene Datenmenge	return ohne Warten mit Wert 0, kein Fehler
write FIFO, FIFO voll	warten bis ausreichend Platz, dann Schreiben der Daten in FIFO	return ohne Warten mit Wert 0, kein Fehler

Betrachten wir einen Dämon (z.B. print spooler), der auf Client Anfragen durch Lesen einer Pipe wartet. Wenn ein Client geschrieben hat und kein anderer Client die Pipe geöffnet hat, so müsste der Dämon von EOF ausgehen, die Pipe schließen und erneut öffnen, um weiterhin Anfragen entgegen nehmen zu können. Dies kann verhindert werden, indem der Dämon die Pipe nicht nur zum Lesen, sondern selbst zum Schreiben öffnet. Die Schreibseite wird aber nie benutzt.

Das Beispielproblem (CS-cat) wird nun mit FIFOs gelöst.

Der eigentliche Vorteil von FIFOs (unabhängige Prozesse können Daten austauschen) wird in der folgenden Realisierung des CS-cat Problems verdeutlicht. Es sind zwei Programme, eines für Server, eins für den Client.

Anmerkung:

Das Programm ist Basis für das Praktikum 1, wenn nicht mit Sockets gearbeitet wird.

```
$ cat fifo.h
#include        <sys/types.h>
#include        <sys/stat.h>
#include        <sys/errno.h>
extern int      errno;

#define FIFO1   "/tmp/fifo.1"    // ggf an dieser Stelle Anlegen
#define FIFO2   "/tmp/fifo.2"

#define PERMS   0666
$
```

client.cpp

```
/* Client */
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>
```

```
#include <fcntl.h>
#include <unistd.h>
#include <stdarg.h>
#include <string>
#include <iostream>
using namespace std;

#include    "fifo.h"
#include    "error.cpp"

#define MAXBUFF      1024

void client(int readfd, int writefd){
    charbuff[MAXBUFF];
    int n;
    /*
     * Read the filename from standard input,
     * write it to the IPC descriptor.
     */
```

```
    if (fgets(buff, MAXBUFF, stdin) == NULL)
        err_sys("client: filename read error");

    n = strlen(buff);
    if (buff[n-1] == '\n')
        n--;        /* ignore newline from fgets() */

    if (write(writefd, buff, n) != n)
        err_sys("client: filename write error");;
    /*
     * Read the data from the IPC descriptor and write
     * to standard output.
     */
    while ( (n = read(readfd, buff, MAXBUFF)) > 0)
        if (write(1, buff, n) != n)   /* fd 1 = stdout */
            err_sys("client: data write error");
    if (n < 0)
        err_sys("client: data read error");
}
```

```
int main(int argc, char** argv)
{
    int readfd, writefd;

    /*
     * Open the FIFOs.  We assume the server has already created them.
     */
    if ( (writefd = open(FIFO1, 1)) < 0)
        err_sys("client: can't open write fifo: FIFO1");
    if ( (readfd = open(FIFO2, 0)) < 0)
        err_sys("client: can't open read fifo: FIFO2");

    client(readfd, writefd);

    close(readfd);
    close(writefd);

    /*
     * Delete the FIFOs, now that we're finished.
     */
    if (unlink(FIFO1) < 0)
        err_sys("client: can't unlink FIFO1");
    if (unlink(FIFO2) < 0)
        err_sys("client: can't unlink FIFO2");
    exit(0);
}
```

server.c

```
/* server */
#define SERVER 1
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

#include <stdarg.h>

#include <string>
#include <iostream>
using namespace std;

#include    "fifo.h"
#include    "error.cpp"

#define MAXBUFF      1024

void server(int readfd, int writefd){
    char    buff[MAXBUFF];
    char    errmesg[256];
    int     n, fd;
    extern int   errno;
```

```
    /*
     * Read the filename from the IPC descriptor.
     */

    if ( (n = read(readfd, buff, MAXBUFF)) <= 0)
        err_sys("server: filename read error");
    buff[n] = '\0';      /* null terminate filename */

    if ( (fd = open(buff, 0)) < 0) {
        err_ret(": can't open");

    if (write(writefd, buff, n) != n)
            err_sys("server: errmesg write error");
    } else {
        /*
         * Read the data from the file and write to
         * the IPC descriptor.
         */
        while ( (n = read(fd, buff, MAXBUFF)) > 0)
            if (write(writefd, buff, n) != n)
                err_sys("server: data write error");
        if (n < 0)
            err_sys("server: read error");
    }
}
```

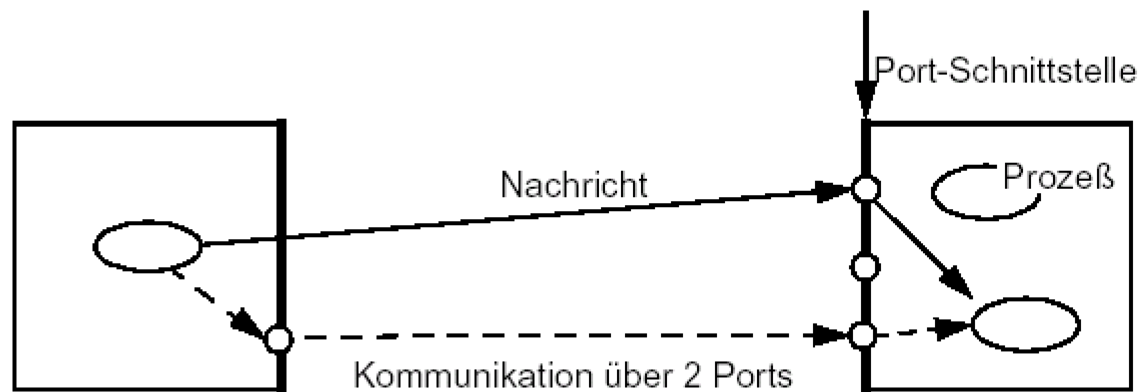
```
int main(int argc, char* argv) {
    int readfd, writefd;
    /*
     * Create the FIFOs, then open them - one for reading and one
     * for writing.
     */
    cout << "Server Started" << endl;
    if ( (mknod(FIFO1, S_IFIFO | PERMS, 0) < 0) && (errno != EEXIST)) {
        cout << "calling err_sys" << endl;
        err_sys("can't create fifo: FIFO1");
    }
    else
        cout << "FIFO1 - created" << endl;
    if ( (mknod(FIFO2, S_IFIFO | PERMS, 0) < 0) && (errno != EEXIST)) {
        unlink(FIFO1);
        cout << "calling err_sys" << endl;
        err_sys("can't create fifo: FIFO2");
    }

    if ( (readfd = open(FIFO1, 0)) < 0)
        err_sys("server: can't open read fifo: FIFO1");
    if ( (writefd = open(FIFO2, 1)) < 0)
        err_sys("server: can't open write fifo: FIFO2");

    server(readfd, writefd);

    close(readfd);
    close(writefd);
    exit(0);
}
```

Die Programme sind etwa wie folgt zu starten (Server im Hintergrund, Client interaktiv):

```
$ server &
$ client
Dateiname
Dateiinhalt
.......
$
```

2.2.2.Ports

Ports sind spezielle Sende- und Empfangspunkte. Sie werden vom Betriebssystem zur Verfügung gestellt.



Ports besitzen folgende **Eigenschaften**:

- Sie definieren die **Kommunikationsendpunkte** und kapseln dadurch die interne Objekt- oder Prozessstruktur ab. Der Port ist somit unabhängig von der Implementierung des dahinter liegenden Prozesses.
- Ports haben eine **global eindeutige Identifikation**, die sich meist aus der **Rechneradresse** und einer **Portnummer** zusammensetzt.
- Durch Ports wird ein selektiver Nachrichtenempfang unterstützt. Ein Prozess sendet oder empfängt durchaus auf mehreren Ports.

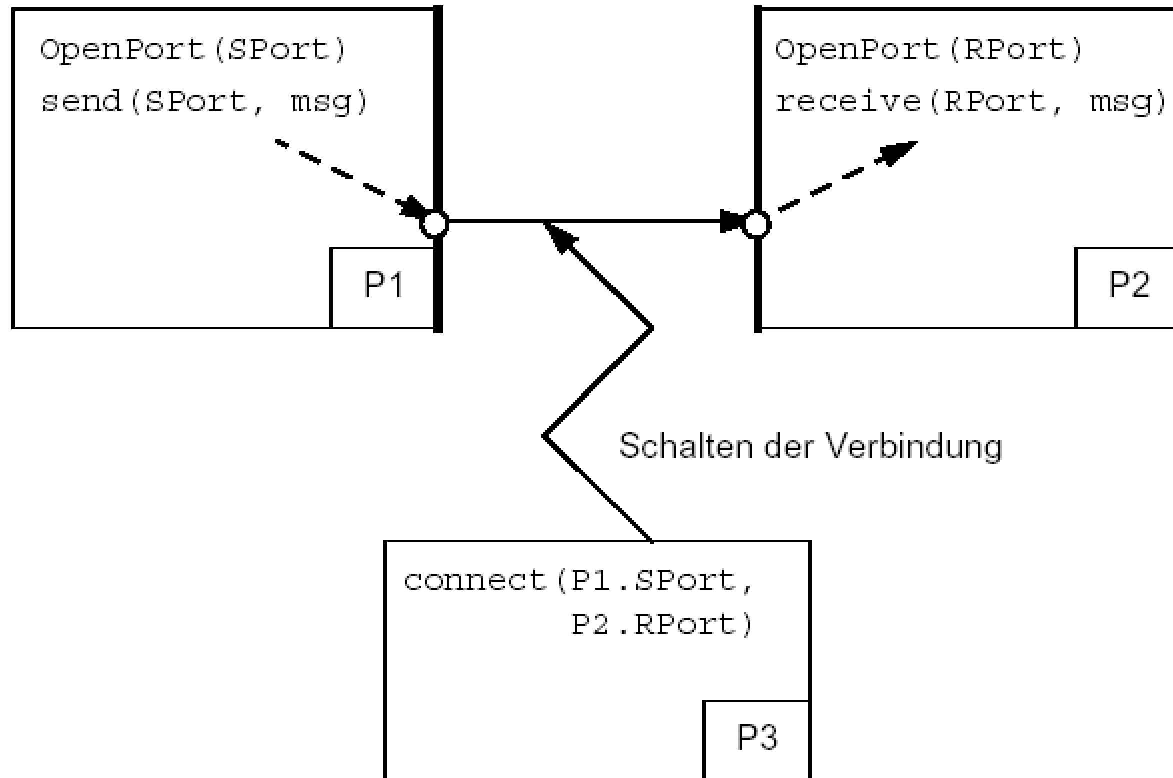
Typische **Port-Operationen** sind:

```
PortId = CreatePort();        -- erzeugt Port und liefert PortId
DeletePort(PortId);           -- löscht einen Port
OpenPort(PortId);             -- öffnet Port für diesen Prozeß
ClosePort(PortId);            -- schließt einen Port
send(PortId, message);        -- schickt Nachricht über den Port
receive(PortId, message);     -- empfängt Nachricht vom Port
```

Der **Bindevorgang** wird über einen externen Vermittlungsdienst abgewickelt, der

- als separater Prozess
- im Betriebssystem verankert ist

Die Port-Kommunikation kann wie folgt veranschaulicht werden:



Anstelle eines externen Binde-Prozesses gibt es folgende Adressierungsvarianten:

- Adressen sind fest in den Programmcode integriert
  - einfach zu programmieren
  - sehr unflexibel
  - Adressänderung führt zur Editierung und Rekompilierung
- Adressanfrage als Broadcast ins Netz
  - Zugehöriger Server meldet sich
  - Lokations-Transparenz
- Adressanfrage an Namensdienst
  - Nachschauen in Adress-Datenbasis
  - Klient wendet sich direkt an den Server

3. Blockierung

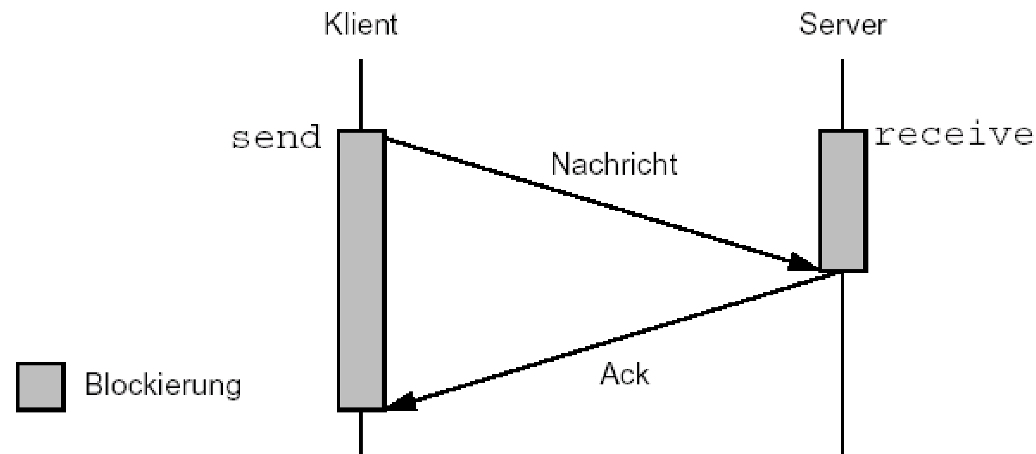
Blockierung bezeichnet die Art, wie ein Kommunikationspartner auf Sende-, bzw. Empfangsoperationen reagiert, unterschieden wird synchrone und asynchrone Kommunikation.

3.1. Synchrone Kommunikation

Blockierende Kommunikation führt automatisch zu einer **Synchronisation** der Teilnehmerprozesse.

Ein Sender blockiert bis die Routine `send` zurückkehrt:

- d.h. er wartet bis die Nachricht über das Netz losgeschickt worden ist und eventuell eine Empfangsbestätigung vom Empfänger zurückgekommen ist.
- Ein korrespondierendes `receive` beim Empfänger blockiert bis eine Nachricht angekommen ist.



Vorteile synchroner Kommunikation:

- Es sind **keine zusätzlichen** Mechanismen zur zeitlichen **Synchronisation** notwendig.
- Da auf jede eingehende Nachricht explizit gewartet wird, ist keine Pufferung von Nachrichten notwendig. Das spart sowohl Speicherplatz, als auch Verwaltungsaufwand.

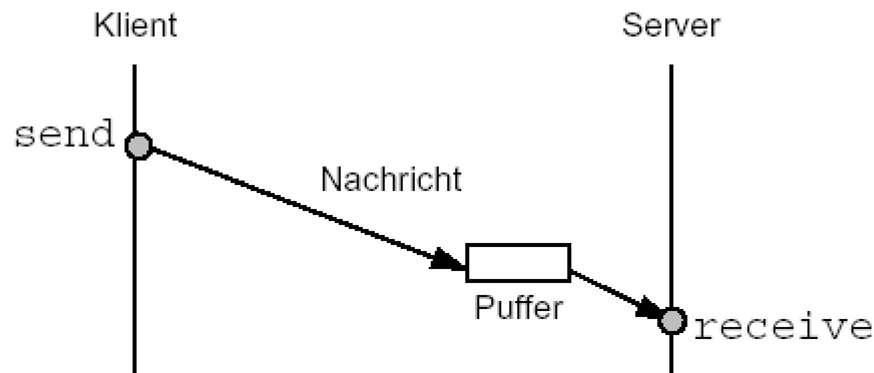
Nachteile synchroner Kommunikation:

- Durch die Blockierung ist die **Parallelität** von Sender und Empfänger stark **eingeschränkt**.
- Bei blockierendem `receive` kann der Empfänger nur auf eine Art von Nachricht warten.
- Ist der Sender bzw. Empfänger nicht bereit, so kann der entsprechende Empfänger bzw. Sender für immer blockieren. Um dies zu vermeiden, muss man mit einem Timeout arbeiten.

3.2.Asynchrone Kommunikation

Asynchrone Kommunikation ist gekennzeichnet durch:

- Bei Kommunikation blockiert der Sender ebenfalls bis die Routine `send` zurückkehrt.
- `send` kehrt aber bereits zurück, wenn die Nachricht im **Transportpuffer** des Kommunikationssystems abgelegt ist.
- Ob und wann sie auf dem Netz erscheint, weiß der Sender damit nicht.
- Das korrespondierende `receive` liest in einem bereitgehaltenen Empfangspuffer.
- Ist eine Nachricht enthalten, wird sie gelesen, ansonsten kehrt `receive` in das rufende Programm zurück.



Oft ist es ausreichend, dass nur der Sender asynchron läuft; dann kann der Empfänger im `receive` auch blockieren.

Vorteile asynchroner Kommunikation:

- zeitliche Entkopplung von Sender und Empfänger.
- Parallelarbeit möglich.

Nachteile asynchroner Kommunikation:

- Pufferung auf der Sende- und Empfangsseite.
- Der Sender weiß nicht, ob und wann die Puffer frei sind.
- Sind die Puffer voll, so muss blockiert werden, was dem asynchronen Konzept widerspricht.

Einsatzgebiete: Echtzeit- oder ereignisgesteuerte Anwendungen

4. Pufferung

4.1. Ungepufferte Kommunikation

Kommunikation ohne Pufferung ist gekennzeichnet durch:

- `receive` stellt im **Programm-Adressraum** eine einzelne **Datenstruktur** (i.a. als Array) für eine Nachricht zur Verfügung.
- Beim Eingang einer Nachricht kopiert das Kommunikationssystem im Betriebssystemkern die Nachricht in diese Datenstruktur und setzt den Empfangsprozess auf „ready".

Vorteile:

- kein Speicher- und Verwaltungsaufwand

Nachteile:

- Wurde `receive` noch nicht aufgerufen, kann das Kommunikationssystem eine eingehende Nachricht nicht zustellen und muss sie verwerfen.
- Rufen mehrere Klienten den Server, so ist dieser eventuell nicht empfangsbereit, da er noch eine alte Nachricht bearbeitet.

4.2. Gepufferte Kommunikation

Kommunikation mit Pufferung ist gekennzeichnet durch:

- Der **Betriebssystemkern** hält **Puffer** vor, um Nachrichten zwischenzuspeichern, die vom Empfänger oder dem Netzwerk momentan nicht abgenommen werden können.
- Falls diese Puffer voll laufen, wird ein Wegwerfen von Nachrichten bzw. eine Blockierung des Senders notwendig.

5. Kommunikationsformen

Kommunikationsformen können durch folgende Tabelle charakterisiert werden:

	asynchron	**synchron**
meldungsorientiert	Datagramm	Rendezvous
auftragsorientiert	asynchroner entfernter Dienstaufruf	synchroner entfernter Dienstaufruf

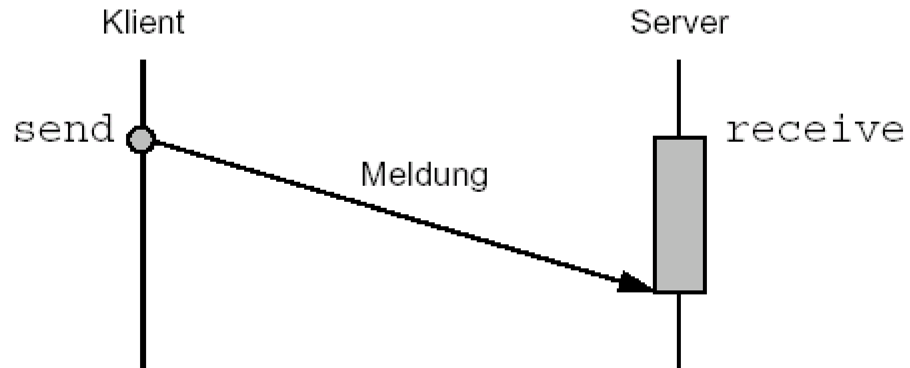
5.1. Meldungsorientierte Kommunikation

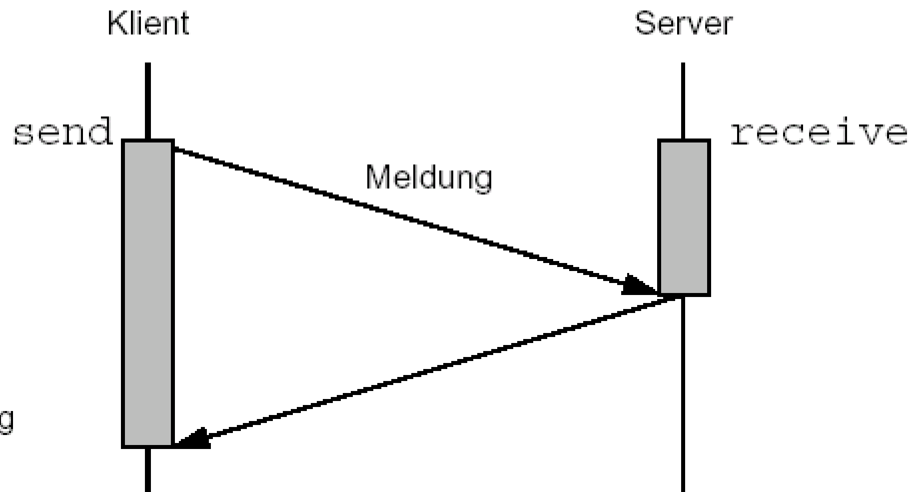
Bei der Kommunikation werden **einfach Nachrichten versendet**, ohne dass eine Antwort erwartet wird:

- Es sind also immer **Einwegnachrichten**,
- d.h. der Sender schickt eine Nachricht und erwartet höchstens eine Empfangsbestätigung
- Eine echte Antwort ist nicht direkt mit der Nachricht verbunden

Asynchrone meldungsorientierte Kommunikation bezeichnet man als **datagramm-basierte** Kommunikation.

Synchrone meldungsorientierte Kommunikation bezeichnet man als **Rendezvous**.





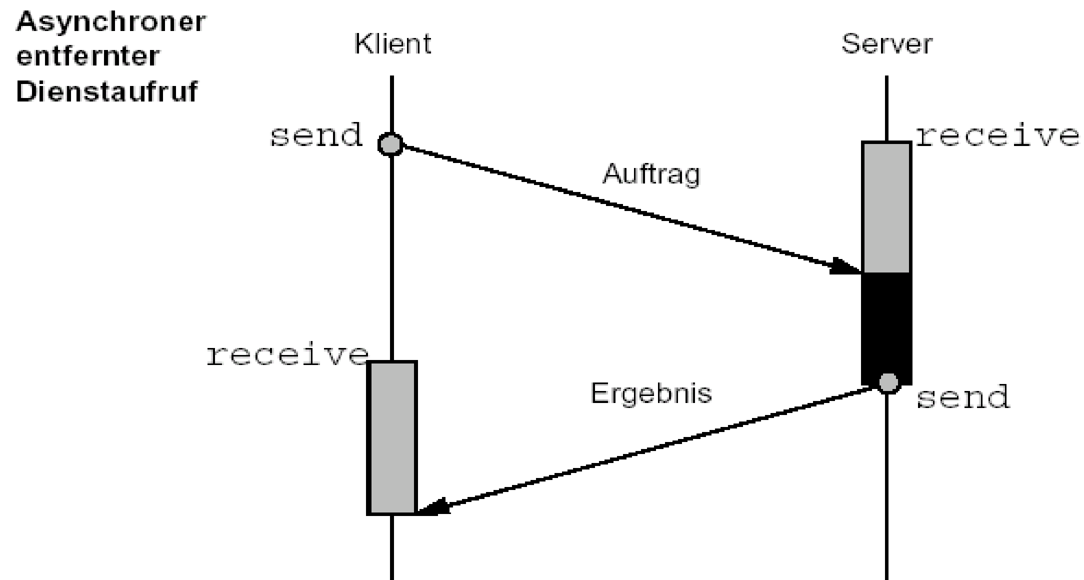
5.2.Auftragsorientierte Kommunikation

Diese Kommunikationsform ist charakterisiert durch:

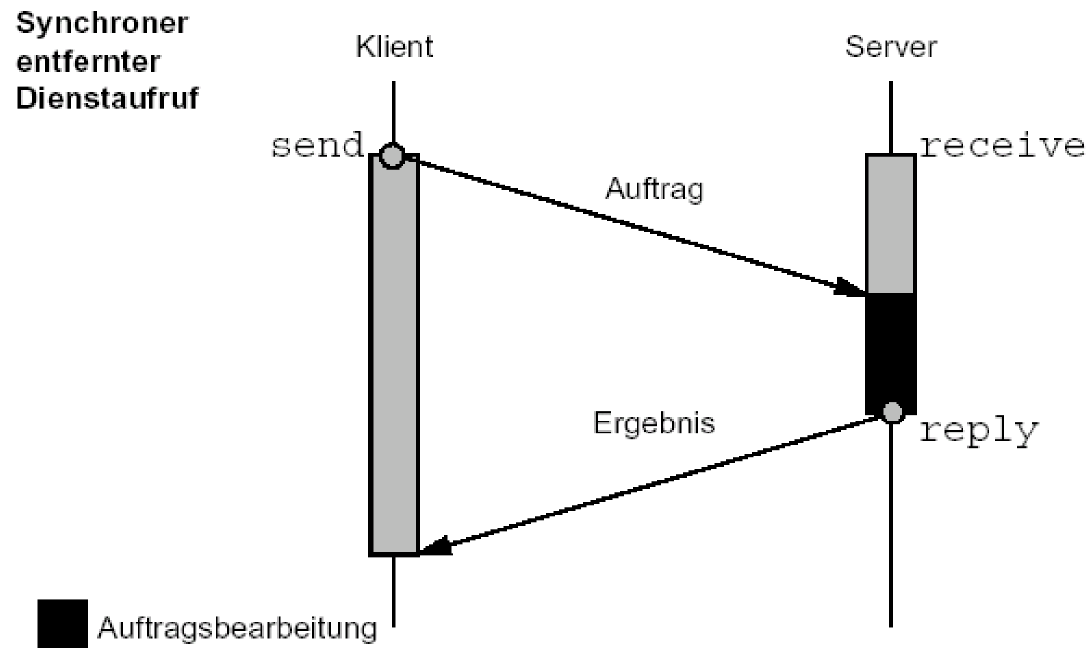
- Es gibt Hin- und Rücknachrichten
- d.h. der Sender schickt eine Auftragsnachricht und erwartet die Rücksendung eines Ergebnisses.

Die Kommunikationsformen bezeichnet man als

- asynchronen entfernten Dienstaufruf (engl.: asynchronous remote service invocation, ARSI), bzw.



- synchronen entfernten Dienstaufruf (engl.: synchronous remote service invocation, SRSI).



6. Zuverlässigkeit auftragsorientierter Kommunikation

Der **fehlerfreie Ablauf eines Prozesses** innerhalb eines Verteilte Systems ist abhängig von:

- der **eigenen** Umgebung,
- vom Verhalten des **anderen** Prozesses und
- vom Verhalten des **Kommunikationssystems**.

Folgende **Fehlerursachen** sind unterscheidbar:

- Der **Zielknoten** oder -rechner wird **nicht erreicht**:
    - Adressierungsproblem
    - Netzwerkfehler
- Die beteiligten Rechner oder **Prozesse** sind **abgestürzt**:
    - dies kann zu jedem Zeitpunkt passieren,
    - sogar „mitten in der Nachricht“;
    - Folge:
        - der Klient wartet z.B. endlos, falls er einen blockierenden Aufruf verwendet hat und der Server ist abgestürzt.

  - verwaiste Aufrufbearbeitungen, so genannte **Orphans**, wenn der Server bereits begonnen hat, einen Auftrag zu bearbeiten, der zugehörige Klient jedoch vor Ergebnisabnahme abgestürzt ist.

- Der angesprochener Server ist nicht mehr vorhanden
- Die gewünschte Servicefunktion wird nicht mehr angeboten

Verteilte Systeme reagieren auf die o.a. Fehlerszenarien mit unterschiedlicher Vorgehensweise. Man unterscheidet folgende **Fehlersemantik**:

Maybe

- es erfolgen **keine Fehlerbehandlungsmaßnahmen**
- der Dienst wird gar nicht oder höchstens einmal durchgeführt
- im Fehlerfall hat man keine Hinweise, ob der Dienst tatsächlich ausgeführt wurde oder nicht

Die **Maybe-Semantik** ist u.U. ausreichend für **Auskunftsdienste**. Wenn nach einer gewissen Zeit keine Auskunft erteilt wurde, so probiert man es eben noch einmal.

At-Least-Once

- der Auftrag wird **mindestens einmal** ausgeführt
- bei Nachrichtenverlusten wird der Auftrag **bis zum Erfolg wiederholt**
- da keine Filterung von Duplikaten stattfindet, kann es vorkommen, dass ein Auftrag mehrfach ausgeführt wird

Bei **idempotenten Operationen** (d.h. mehrfache Ausführung verändert das Ergebnis nicht) ist diese Semantik ausreichend. Das wiederholte Lesen einer unveränderten Datei ist beispielsweise eine idempotente Operation.

At-Most-Once

- die Fehlerbehandlung erfolgt durch Wiederholen des Auftrags
- zusätzlich werden alle Duplikate ausgefiltert
- auch bei vermeintlichen Nachrichtenverlusten wird somit der **Auftrag höchstens einmal** ausgeführt
- sollte der Server jedoch abgestürzt sein, so wird kein Ergebnis mehr erwartet

Diese Fehlersemantik wird verwendet, bei **Atomarität der entfernten Operation**. Ein Auftrag wird also komplett durchgeführt, oder ein Fehler an den Klienten gemeldet, wobei dieser dafür Sorge tragen muss, mit dem nicht ausgeführten Auftrag umzugehen.
In **RPC-Implementierungen**, wie **Corba**, **DCOM**, **Java RMI** wird diese Fehlersemantik verwendet.

Exactly-Once

- schließt den Absturz und den Wideranlauf von Komponenten ein
- Konsistente Rücksetzungsmaßnahmen garantieren, dass die Operation **genau einmal** durchgeführt wird
- persistente Datenhaltung und verteilte Transaktionsmechanismen sind notwendig

Die Exactly-Once-Semantik ist durch den großen Aufwand nur bei Anwendungen mit **hohen Sicherheitsanforderungen** sinnvoll.

Beispielszenarien für Fehlersituationen

Annahme:

alle zu einem Auftrag gehörenden Protokollnachrichten sind identifizierbar, z.B. durch eine **fortlaufende Nummer**.

Ausgangslage:

Der Klient erhält nach dem Abschicken des Auftrags längere Zeit keine Antwort vom Server und reagiert mit einem Timeout.

mögliche Ursachen:

- Die Auftragsnachricht ging verloren.

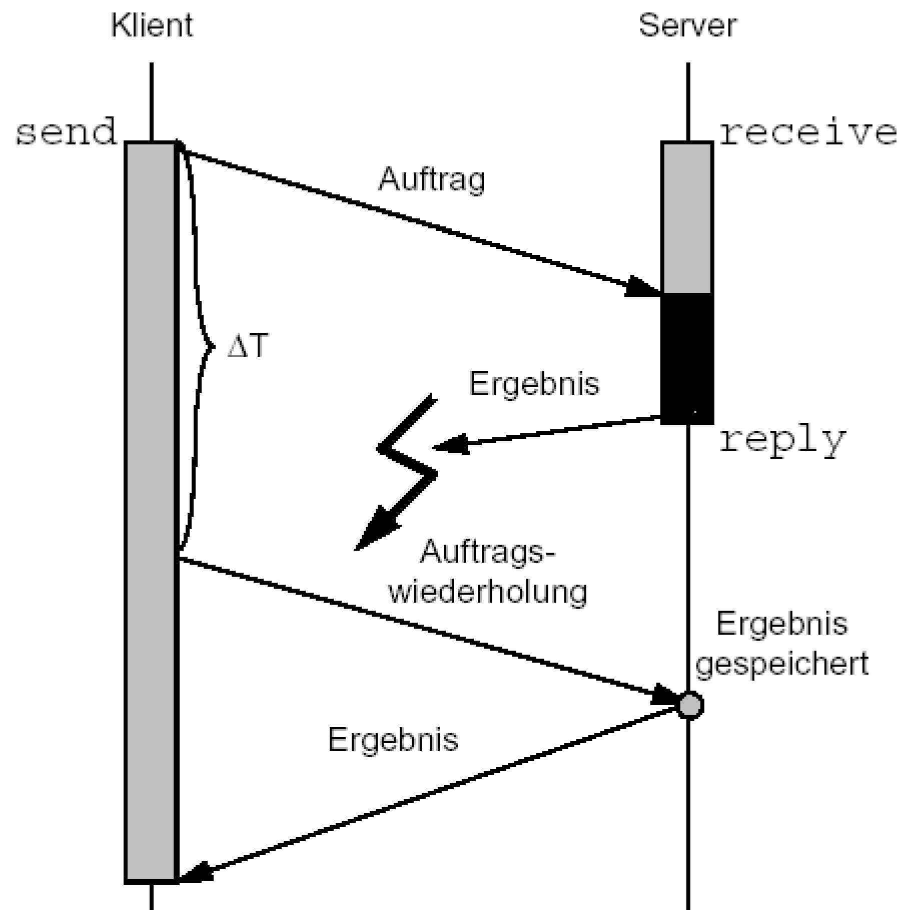
Reaktion:

Den Auftrag wiederholen.
Diese Maßnahme ist in allen anderen noch folgenden Fällen ebenfalls sinnvoll.

- Die Antwort des Servers ging verloren oder ist noch nicht angekommen.

Reaktion:

Neben der Auftragswiederholung kann man auf Serverseite die Antwort-Nachrichten **zwischenspeichern** und bei erneuter Anforderung nochmals ohne wiederholte Berechnung schicken.


Klient
Server
send
receive
Auftrag
ΔT
Ergebnis
reply
Auftrags-
wiederholung
Ergebnis
gespeichert
Ergebnis

- Der Server ist abgestürzt.
  Reaktion:
  - Ist nach der Auftragswiederholung der Server immer noch abgestürzt, so hat der Klient „Pech gehabt“ und wird erneut mit dem Timeout reagieren.
  - Wurde der Server erneut gestartet, so hat er normalerweise keine Kenntnis über bereits bearbeitete Aufträge seines Vorgängers (Server-Amnesie).
  - Deswegen muss er beim Klienten eventuell Rückfrage halten, ob sein Vorgänger diesen Auftrag bereits bearbeitet hat.

  Nach einem erneuten Start des Servers gibt es zwei Varianten bezüglich des Zustands der Kommunikationsverbindungen zwischen Klient und Server:

  - Die Kommunikationsbeziehungen sind ungültig:

    auf der Klientenseite muss ein Protokoll anlaufen, das die Verbindung mit neuem Server aufnimmt. Danach können neue Aufträge erteilt werden.

  - Die alten Beziehungen sind weiterhin intakt.

    Server wartet n * T bis er wieder Aufträge annimmt,
    wobei n = Wiederholzahl der Aufträge, T = maximale Lebensdauer einer Nachricht

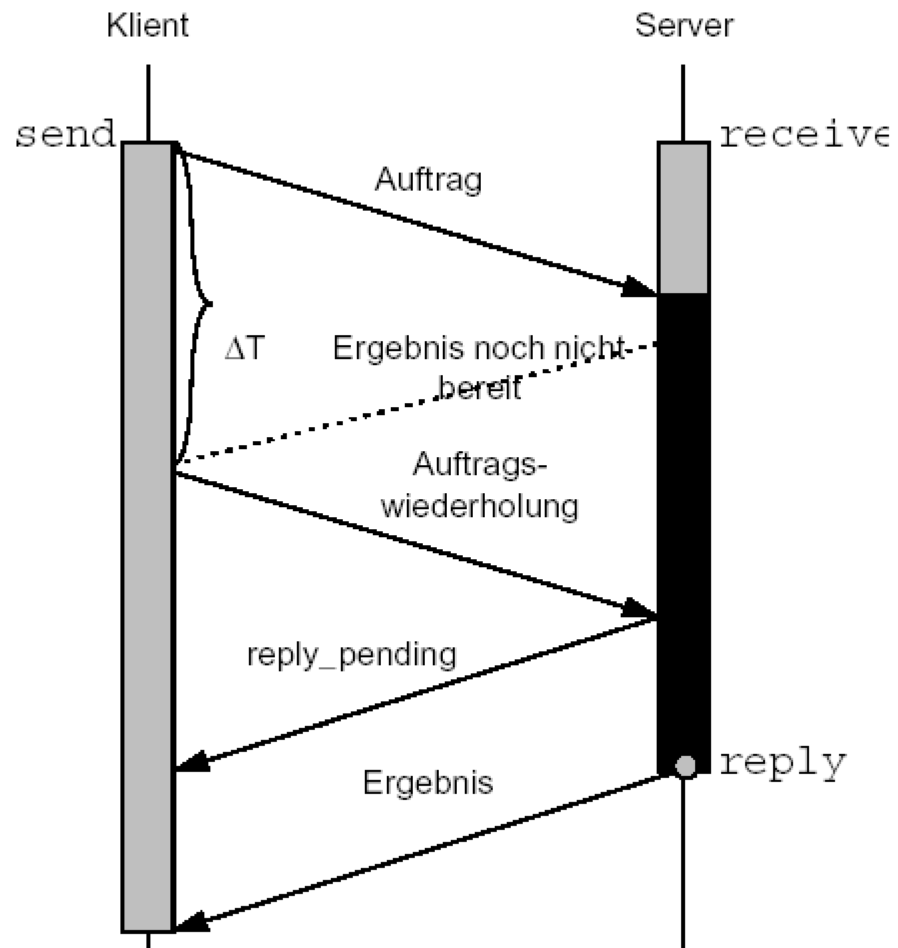
    Die Klienten haben bis dahin die alten Aufträge verworfen.

- Der Server ist gerade vor dem Versenden der Antwort abgestürzt
  Reaktion:

  Ohne Exactly-Once-Semantik hat der Klient keine Möglichkeit zu überprüfen, ob der Auftrag ausgeführt wurde oder nicht.

- Der Server braucht lange Zeit für die Auftragsausführung.
  Reaktion:

  Erreicht den Server während der Auftragsbearbeitung ein Wiederholungsauftrag, so kann er `reply_pending` schicken. Hiermit wird dem Klienten signalisiert, dass der Server noch mit der Bearbeitung seines Auftrags beschäftigt ist.


Klient
Server
send
receive
Auftrag
ΔT
Ergebnis noch nicht bereit
Auftrags-wiederholung
reply_pending
Ergebnis
reply

7. Socket Kommunikation

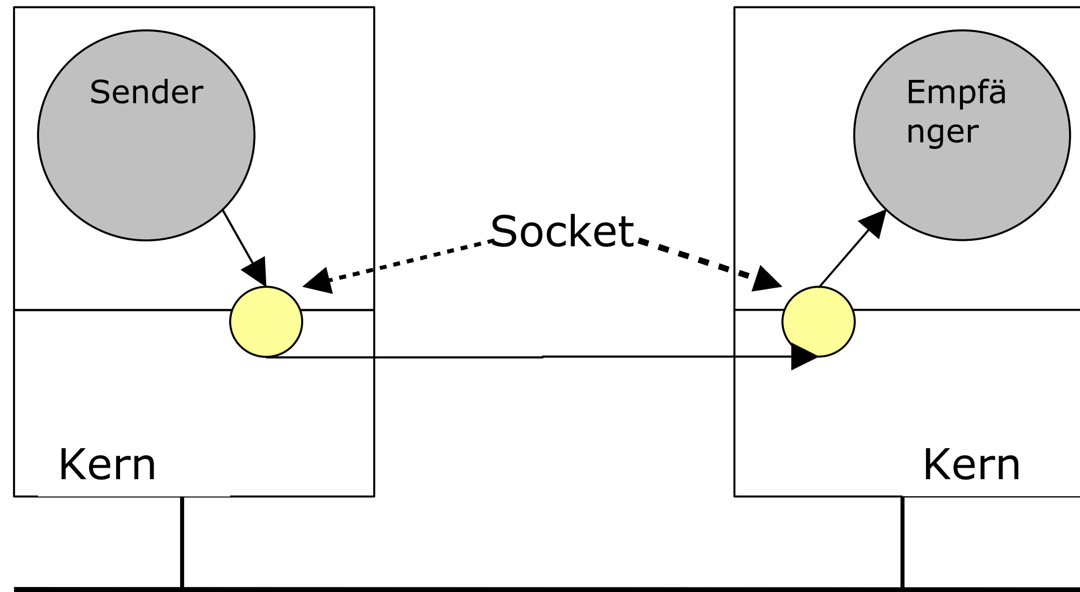
Die Kommunikation zwischen zwei Rechnern in einem Netzwerk wurde stark beeinflusst vom Konzept der **Sockets**, das im **Berkley** Unix vorgestellt wurde.

Ein **Port** ist die **Abstraktion eines physischen Ortes**, über den Client und Server kommunizieren. Der Server stellt den Port zur Verfügung, mit dem sich der Client verbindet. In Betriebssystemen wird jedem Port ein Prozess zugeordnet, die die Kommunikation über den Port verwaltet. Z.B. liefert Unix die Uhrzeit über Verbindungen mit Port 13 durch ein Programm, dass die Zeit vom System erfragt.

Hier wollen wir solche **Server** mittels Sockets programmieren.

```
-> $ telnet localhost 13“, /etc/services
```

7.1.Konzept



Sockets sind **Kommunikationsendpunkte**, die dem Programmierer eine Schnittstelle zu einem Netzwerk zur Verfügung stellen (so wie ein Telefon eine Schnittstelle ist zum Telefonsystem), so wie in den o.a. Abbildung verdeutlicht.

Sockets sind quasi **Abstraktionen des Netzes**.

Sockets können dynamisch erzeugt und zerstört werden.

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.

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

- zuverlässiger verbindungsorientierer **Byte**strom
  Dadurch ist eine **Pipekommunikation über Rechnergrenzen** hinweg möglich: die Bytes kommen so an wie sie gesendet wurden.
- zuverlässiger verbindungsorientierter **Paket**strom
  hier ist das Verhalten wie bei 1, aber Paketgrenzen werden eingehalten.
- unzuverlässige Paketübertragung zum elementaren Zugriff auf das Netzwerk (für Echtzeitanwendungen)

Ein weiterer Parameter, der bei der Erzeugung eines Socket angegeben werden muss, ist das zu verwendende Protokoll:

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

Bevor ein Socket zur Kommunikation verwendet werden kann, muss eine Adresse an ihn gebunden werden, meist eine IP Adresse.

Nachdem die Sockets auf Quell- und Zielrechner erzeugt sind, kann eine Verbindung aufgebaut werden (bei Typ 1 und 2).

Auf der **Empfängerseite** wird (durch LISTEN) **ein Puffer aufgebaut** und der Empfänger blockiert bis zum Eintreffen von Nachrichten.

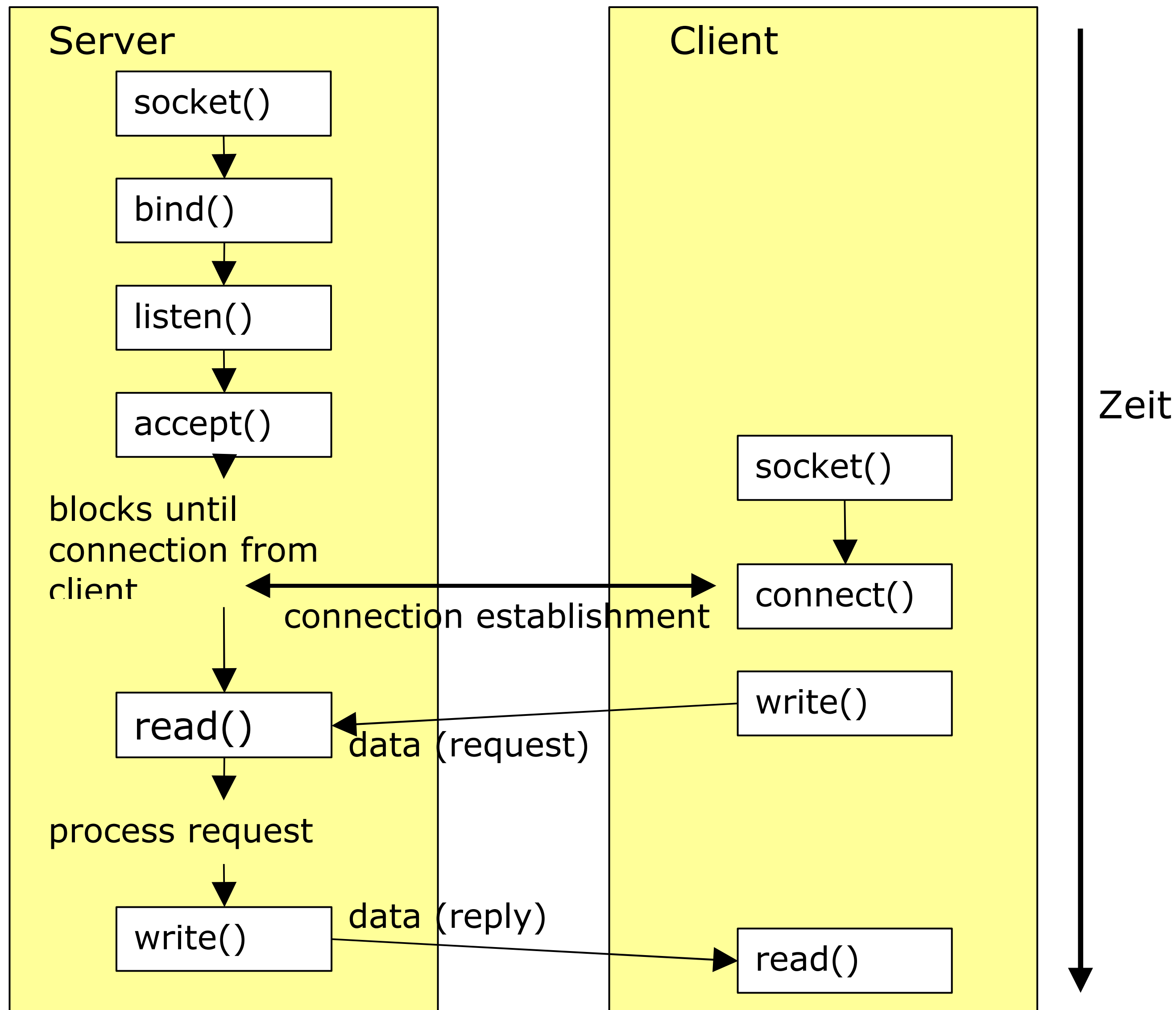
Auf der **Sendeseite** wird (durch CONNECT) eine Datenstruktur aufgebaut, die lokales und entferntes **Socket adressiert**. Dann wird die Verbindung hergestellt.

Nach Verbindungsaufbau erfolgt die **Kommunikation** analog zu **Pipes**, allerdings über das Netzwerk.

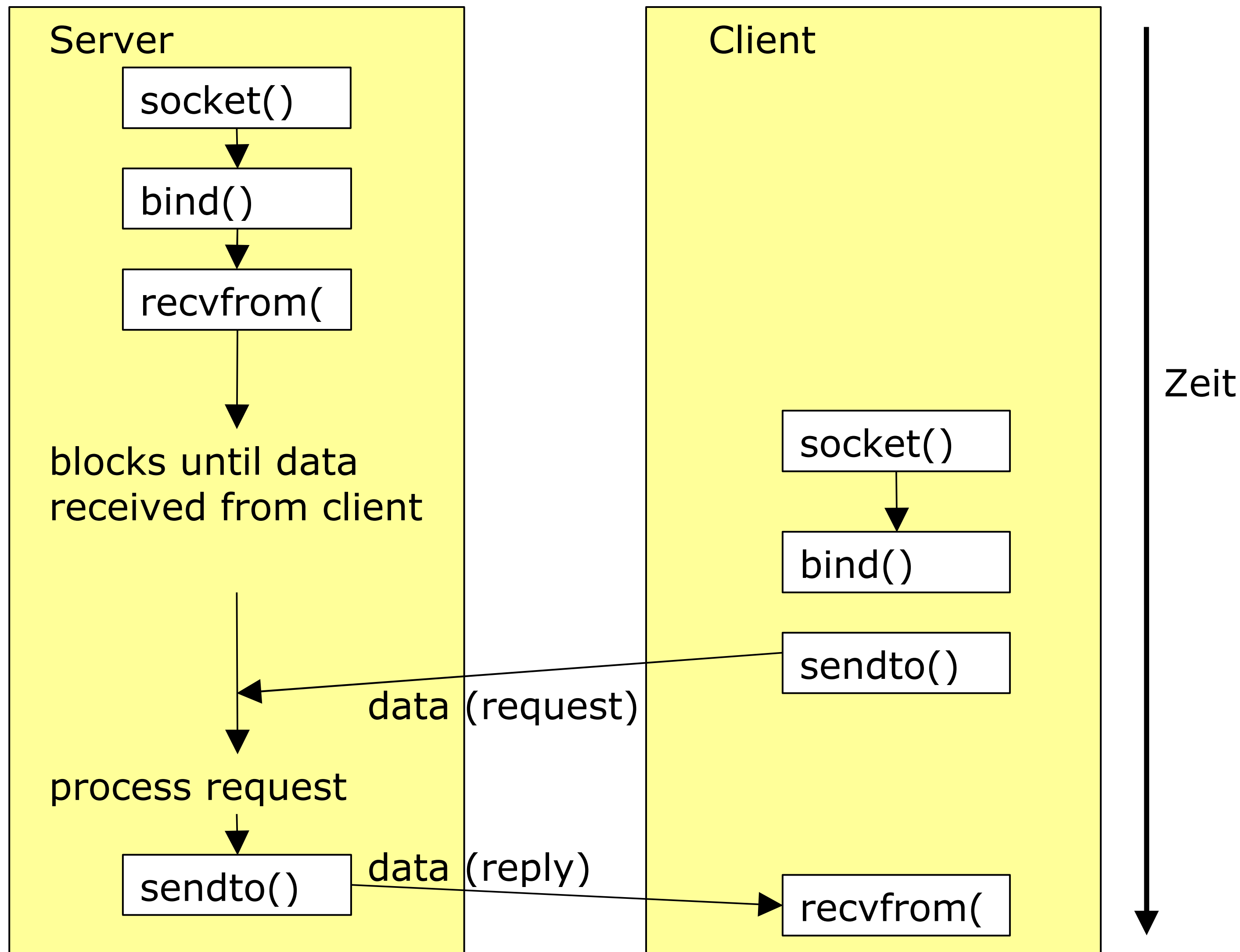
Zum Kommunizieren werden Bibliotheksfunktionen (z.B. in C oder Java-Klassen) verwendet.

Aufruf	verbindungsorientiert	Verbindungslos
socket()	Festlegen der Kommunikationsart und Erzeugen des Kommunikationsendpunktes	
bind()	Zuweisen eines Namens zum bis dahin namenlosen Socket	
listen()	Server ist zur Kommunikation bereit	
accept()	Server akzeptiert Requests	
read(), write()	Lesen, Schreiben	
connect()	Verbindungsaufbau	
sendto(), recvfrom()		Lesen, Schreiben

Das Zusammenspiel der Systemaufrufe bei **verbindungsorientierter** Kommunikation erfolgt wie folgt:



Das Zusammenspiel der Systemaufrufe bei **verbindungsloser** Kommunikation erfolgt wie folgt:



Die wesentlichen Systemaufrufe sind wie folgt:

```
BEZEICHNUNG
       socket - erzeuge einen Kommunikationsendpunkt

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

       int socket(int domain, int type, int protocol);

BESCHREIBUNG
       Socket  erzeugt  einen Kommunikationsendpunkt und gibt den
       zugehörigen Deskriptor zurück.

       Der Parameter domain  spezifiziert  die  Kommunikationsdo
       main,  in der die Kommunikation stattfinden soll, also die
       Protokollfamilie, die benutzt werden soll.  Diese Familien
       sind in der Include-Datei <sys/socket.h> definiert.
```

Zur Zeit werden folgende Domains unterstützt:

AF_UNIX (UNIX interne Protokolle)

AF_INET (ARPA Internet Protokolle)

AF_ISO (ISO Protokolle)

AF_NS (Xerox Network Systems Protokolle)

AF_IMPLINK
(IMP "host at IMP" Link Ebene)

Der Socket hat den in type angegebenen Typ, der die Art der Kommunikation bestimmt. Zur Zeit sind folgende Arten definiert:

SOCK_STREAM
SOCK_DGRAM
SOCK_RAW
SOCK_SEQPACKET
SOCK_RDM

Ein Socket vom Typ SOCK_STREAM stellt einen sequenziellen, verläßlichen, zwei-weg-basierten Byte-Stream zur Verfügung. Ein "out-of-band" Übertragungsmechanismus kann unterstützt werden. Ein Socket vom Typ SOCK_DGRAM bietet Datagramme (verbindungslos, unverläßliche Nachricht einer festen (meist kleinen) maximalen Länge). Ein SOCK_SEQ PACKET Socket kann einen sequenziellen, verläßlichen,

zwei-weg-basierten Übertragungspfad für Datagramme einer festen maximalen Länge bieten. Möglicherweise wird vom lokalen Endpunkt erwartet, daß er ein komplettes Paket mit jedem read System-Call liest. Diese Art ist protokoll spezifisch und zur Zeit nur für AF_NS implementiert. SOCK_RAW Sockets stellen Zugriff auf interne Netzwerkpro tokolle und Schnittstellen zur Verfügung. Die Typen SOCK_RAW, der nur für den Superuser (=root) zugänglich ist, und SOCK_RDM, der geplant aber noch nicht implemen tiert ist, werden hier nicht beschrieben.

Das Protocol bezeichnet ein spezielles Protokoll, das auf diesem Socket benutzt wird. **Normalerweise gibt es nur ein einziges Protokoll,** das von einem speziellen Socket einer Protokollfamilie unterstützt wird. Nichtsdestotrotz ist es möglich, daß mehrere Protokolle existieren. In diesem Fall muß das zu verwendende auf diese Art angegeben wer den. Die Protokollnummer ist individuell für eine bes timmte "Kommunikationsdomain". Siehe dazu auch proto cols(5).

Sockets des Typs **SOCK_STREAM** sind **voll-duplex-orientierte Byte-Streams, ähnlich wie Pipes.** Ein Stream-Socket muß sich in einem connected Stadium befinden bevor mit ihm irgendwelche Daten gesendet oder empfangen werden können. Eine Verbindung zu einem anderen Socket wird mit con nect(2) hergestellt. Einmal verbunden können Daten mit read(2) und write(2) übertragen werden bzw. mit Varianten von send(2) oder recv(2). Wenn eine Verbindung abgebaut

```
werden  soll, wird close(2) ausgeführt.  Out-of-band Daten
können, wie in send(2) beschrieben, gesendet und,  wie  in
recv(2) beschrieben, empfangen werden.

Die Kommunikationsprotokolle, die verwendet werden, um ein
SOCK_STREAM zu implementieren, stellen sicher,  daß  Daten
weder  verloren  gehen  noch  verdoppelt werden.  Wenn ein
Datum, für das das Partnerprotokoll ausreichend Puffer zur
Verfügung hat, in einem angemessenen Zeitraum nicht erfol
greich übertragen werden kann, wird  angenommen,  daß  die
Verbindung  kaputt ("broken") ist und Aufrufe zeigen einen
Fehler an, indem sie  -1  zurückgeben  und  ETIMEDOUT  als
entsprchenden  Wert in der globalen Variable errno setzen.
Die Protokolle halten den  Socket  unter  Umständen  warm,
indem sie ca. jede Minute eine Übertragung erzwingen, wenn
keine anderen  Aktivitäten  vorliegen.   Ein  Fehler  wird
angezeigt,  wenn  keine  Antwort  auf  einer sonst stillen
Verbindung  in  einer  erweiterten  Zeitspanne   (z.B.   5
Minuten)  erzielt  werden  kann.   Ein SIGPIPE-Signal wird
erzeugt, wenn ein Prozeß auf einen kaputten Stream sendet;
das  verursacht  bei  naive Prozesse, die das Signal nicht
verarbeiten, ein Programmende.

SOCK_SEQPACKET-Sockets setzen  dieselben  Systemcalls  ein
wie  SOCK_STREAM-Sockets.  Der einzige Unterschied besteht
darin, daß read(2) nur die  angeforderte  Menge  an  Daten
zurückliefert und alle restlichen verwirft.

SOCK_DGRAM-  und  SOCK_RAW-Sockets erlauben das Senden von
```

```
        Datagrammen zu Empfängern, die im send(2)  Aufruf  genannt
        werden.   Datagramme  werden grundsätzlich mit recvfrom(2)
        empfangen, das das  nächste  Datagramm  zusammen  mit  der
        Absenderadresse zurückliefert.

        Ein   fcntl(2)   Aufruf   kann   benutzt  werden,  um  ein
        Prozeßgruppe zu spezifizieren, die ein SIGURG-Signal  emp
        fangen  soll,  wenn  out-of-band  Daten ankommen.  So kann
        ebenfalls eingestellt werden,  daß  non-blocking  I/O  und
        asynchrone Benachrichtigung von I/O Ereignissen  via SIGIO
        vorgenommen werden.

        Die Arbeitsweise von Sockets wird von  Socket-Level-Optio
        nen   gesteuert.   Diese   sind   in   der  Include-Datei
        <sys/socket.h> definiert.  Setsockopt(2) und getsockopt(2)
        werden  verwendet,  um  diese  Optionen  zu setzen bzw. zu
        lesen.

RÜCKGABEWERTE
        -1 wird zurückgegeben, wenn ein Fehler auftritt, ansonsten
        wird  die  Nummer  des  Deskriptors zurückgegeben, der den
        Socket referenziert.
```

```
BEZEICHNUNG
       bind - verbindet einen Namen mit einem Socket.

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

       int   bind(int   sockfd,  struct  sockaddr  *my_addr,  int
       addrlen);

BESCHREIBUNG
       bind weist dem Socket sockfd die  lokale  Adresse  my_addr
       zu.  my_addr ist addrlen Bytes lang.  Traditionsgemäß wird
       dies "einem Socket einen Namen zuweisen" genannt (wenn ein
       Socket  mit  socket(2) erzeugt wird, existiert er in einer
       Adreßfamilie (Namespace), hat aber keinen eigenen Namen.
```

```
BEZEICHNUNG
       listen - Horche auf einem Socket auf Verbindungen

ÜBERSICHT
       #include <sys/socket.h>
       int listen(int s, int backlog);

BESCHREIBUNG
       Um  eine  Verbindung anzunehmen, muß ein Socket zuerst mit
       socket(2)  erzeugt   werden.   Der   Wunsch,   eingehende
       Verbindungen anzunehmen sowie ein Limit an nicht bearbeit-
       eten Verbindungen, die  in  einer  Warteschleife  gehalten
       werden,  wird mit listen() angezeigt.  Anschließend werden
       Verbindungen mit accept(2)  angenommen.   Der  Aufruf  von
       listen  ist  für Sockets vom Typ SOCK_STREAM und SOCK_SEQ
       PACKET gültig.

       Das Argument backlog spezifiziert die maximale  Länge  der
       Warteschlange,  die  noch  nicht  angenommene Verbindungen
       aufnimmt.  Wenn eine Verbindungsanfrage  ankommt  und  die
       Warteschlange  ist  bereits  voll,  dann erhält der Client
       einen Fehler mit der Angabe ECONNREFUSED oder die  Anfrage
       wird   ignoriert,   wenn  das  zugrundeliegende  Protokoll
       "Retransmission"  unterstützt,  damit   weitere   Versuche
       erfolgreich sind.

RÜCKGABEWERT
       Bei  Erfolg  wird  null zurückgegeben, bei einem Fehler -1
       und errno wird entsprechend gesetzt.
```

BEZEICHNUNG

accept - **nimmt** eine **Verbindung** auf einem Socket **an**

BEZEICHNUNG

```
#include <sys/types.h>
#include <sys/socket.h>
int accept(int s, struct sockaddr *addr, int *addrlen);
```

BESCHREIBUNG

Der Parameter s ist ein Socket, der mit socket(2), erzeugt wurde, mit bind(2) an eine Adresse gebunden wird und mit listen(2) auf Verbindungen wartet. Die Funktion accept extrahiert den ersten Verbindungswunsch aus der Warteschlange der ankommenden Verbindungen, erzeugt einen neuen Socket mit den gleichen Eigenschaften von s und alloziert einen neue Deskriptor für den Socket. Wenn keine wartende Verbindung vorhanden ist und der Socket nicht als nicht-blockierend markiert ist, blockiert accept den aufrufenden Prozeß bis eine Verbindung vorhanden ist. Wenn der Socket als nicht-blockierend markiert ist und keine wartenden Verbindungen vorhanden sind, gibt accept eine Fehlermeldung, wie sie unten beschrieben ist, zurück. Der akzeptierte Socket kann nicht mehr für weitere Verbindungen benutzt werden. Der Originalsocket s bleibt offen.

Das Argument addr ist ein Rückgabeparameter, das mit der Adresse der verbindenden Einheit gefüllt wird, wie sei der Kommunikationsschicht bekannt ist. Das exakte Format des addr Parameters wird von der Domain festgelegt, in der die Kommunikation stattfindet. Die Variable addrlen ist ein

```
       Rückgabeparameter,  sie  sollte  anfangs  die Anzahl Bytes
       enthalten auf die addr zeigt; bei der Rückgabe enthält  es
       die  aktuelle Länge der Adresse (in Bytes).  Dieser Aufruf
       wird bei verbindungsbasierten Sockettypen benutzt,  momen
       tan in Verbindung mit SOCK_STREAM.
       Es ist möglich, einen Socket mit select(2) aufzumachen, um
       ihn mit einem accept zum Lesen zu benutzen.
       Bei bestimmten Protokollen, die explizite Bestätigung ver
       langen,  wie ISO oder DATAKIT, kann davon ausgegangen wer
       den,  daß  accept  nur  die  nächste  Verbindung  aus  der
       Warteschlange  holt  ohne  sie  automatisch zu bestätigen.
       Die Bestätigung kann ein normaler Lese-  oder  Schreibvor
       gang auf dem neuen Deskriptor mit sich bringen, eine Able
       hung kann impliziert werden durch ein Schließen des  neuen
       Sockets.
       Man  kann  die  Daten  einer  Verbindungsanforderung  ohne
       Bestätigung erhalten, indem man  einen  recvmsg(2)  Aufruf
       absetzt  mit  einer auf null gesetzten msg_iovlen un einem
       msg_controllen ungleich null oder durch  Aufruf  von  get
       sockopt(2).   Analog  dazu  kann  man  die Ablehnung einer
       Benutzerverbindung erzeugen, indem man sendmsg(2) nur  mit
       den  Kontrollinformationen  aufruft  oder  durch  setsock
       opt(2).

RUECKGABEWERTE
       Die Funktion gibt bei Fehlern -1 zurück. Wenn  der  Aufruf
       erfolgreich  war,  gibt  sie  einen  positiven Integerwert
       zurück, der der Deskriptor für  den  aktzeptierten  Socket
       ist.
```

```
BEZEICHNUNG
       connect - Verbindungsaufbau zu einem "Socket"

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

       int  connect(int  sockfd, struct sockaddr *serv_addr, int
       addrlen );

BESCHREIBUNG
       Der Parameter sockfd ist ein "socket".  Wenn  es  vom  Typ
       SOCK_DGRAM  ist,  dann  beschreibt  dieser Parameter den
       "Verbindungspartner"  mit/über  welchen  der  "socket"  in
       Verbindung  gebracht  wird.  Diese Adresse ist die, zu der
       "datagrams" gesendet werden, und  die  einzigste  Adresse,
       von  der  "datagrams"  empfangen  werden können.  Wenn das
       "socket" vom Typ SOCK_STREAM ist, wird dieser Aufruf  ver
       suchen,   eine   Verbindung   zu  einem  anderen  "socket"
       aufzubauen.  Das andere  "socket"  wird  durch  serv_addr,
       welches  die  Adresse innerhalb des Kommunikations Adress
       raumes des "socket" bezeichnet,  festgelegt.  Jeder  Komm.
       Addressraum  interpretiert  serv_addr  Parameter auf seine
       eigene Weise.

       Generell  sollten  "stream  sockets"  nur  einen   connect
       benutzen.

       "Datagram sockets" können mit aller Warscheinlichkeit con
       nect  öfter  verwenden.  "Datagram  sockets"  werden  beim
```

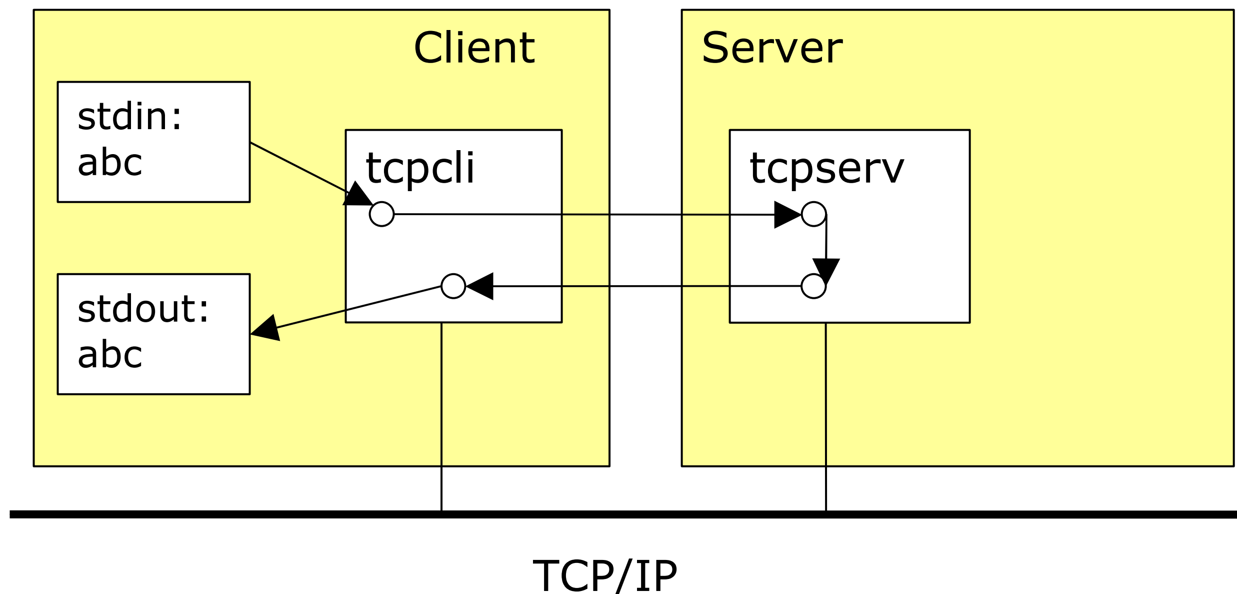
```
        Verbinden  mit unzulässigen Adressen mit aller Wahrschein
        lichkeit  eine  "null   addresse"   zur   Fehlerbehandlung
        zurückgeben.

RÜCKGABEWERT
        Wird die Verbindung zufriedenstellend Hergestellt, so wird
        eine  0  (zero)  zurückgegeben,  im  Fehlerfall  wird   -1
        zurückgegeben.   Zusätzlich  wird  errno auf den passenden
        Wert gesetzt.
```

7.2. Fallbeispiele

7.2.1. Echoserver

Die Socket-Kommunikation wird am Beispiel einer CS Anwendung verdeutlicht, bei der ein **Echo-server** realisiert ist:

1. Der Klient liest eine Zeile von seiner Standardeingabe 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 Standardausgabe aus.



Zu entwickeln sind also **zwei Programme**. Einigung muss dabei über die Kommunikation bestehen.

Der **Server** soll nicht nur die Verbindung zu einem Client erlauben, er **soll mehrere Klienten befriedigen** können, dazu führt er bei jeder Verbindungsanfrage ein **fork** durch – das Kind befriedigt die Anfrage.

Im **Beispiel ist eine verbindungsorientierte Kommunikation über TCP/IP** erläutert. Verwendet wird der Port 9001. Die Serveradresse ist als IP Adresse angegeben. Diese Definitionen sind in eine Header Datei verlagert, die sowohl Client als auch Server verwenden:

```
$ cat c++/inet.h
/**********************************************************************
 * Definitions for TCP and UDP client/server programs.
 */

#include    <stdio.h>
#include    <fcntl.h>
#include    <sys/types.h>
#include    <sys/socket.h>
#include    <arpa/inet.h>
#include    <netinet/in.h>

#define SERV_TCP_PORT    9019
#define SERV_HOST_ADDR   "127.0.0.1"      /* Localhost */
$
```

Hilfsfunktionen zum Lesen und Schreiben und zur Fehlerbehandlung sind in „util.cpp“ und „error.cpp“ abgelegt.

Der Server ist folgendermaßen kodiert:

```
$ cat tcpserv.cpp
/*********************************************************************
 *
 * Example of server using TCP protocol.
 *
 *********************************************************************/

#include    <sys/stat.h>                // open()
#include    "inet.h"
#include    "util.cpp"

#include <iostream>
using namespace std;

/*********************************************************************
 *
 * 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];

    while ( true ){
        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");
        }
    }
}
/************************************************************************/

/************************************************************************
 * 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);

    while (true){
         // Wait for a connection from a client process.
         // This is an example of a concurrent server.

        clilen = sizeof(cli_addr);
        newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr,
                                  (socklen_t *) &clilen);
        if (newsockfd < 0)
            perror("server: accept error");

        if ( (childpid = fork()) < 0)
            perror("server: fork error");
        else if (childpid == 0) {// child process
            close(sockfd);        // close original socket
            str_echo(newsockfd); // process the request
            exit(0);
        }
        close(newsockfd);         // parent process
    }
}
/**********************************************************************/
$
```

Der Client ist folgendermaßen kodiert:

```
$ cat tcpcli.cpp
/***********************************************************************
 *
 * Example of client using TCP protocol.
 *
 ***********************************************************************/
#include    "inet.h"
#include    "util.cpp"
#include    <string.h>

#include <iostream>
using namespace std;
/***********************************************************************
 * Read the contents of the FILE *fp, write each line to the
 * stream socket (to the server process), then read a line back from
 * the socket and write it to the standard output.
 *
 * Return to caller when an EOF is encountered on the input file.
 */
void str_cli(FILE *fp, int sockfd) {
  const int MAXLINE=255;
  int   n;
  char  sendline[MAXLINE], recvline[MAXLINE + 1];

  while (fgets(sendline, MAXLINE, fp) != NULL) {
    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 */
    cout << recvline;
  }

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

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

  if (argc != 3) {
    cout << "usage: " << argv[0] << " host port" << endl;
    exit(1);
  }

  port = atoi(argv[2]);
  host = argv[1];
```

```
  // 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(host.c_str());
  serv_addr.sin_port        = htons(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,
              sizeof(serv_addr)) < 0)
    perror("client: can't connect to server");

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

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

```
$ cat util.cpp
/*************************************************************************
 * Utility functions to use with sockets
 *************************************************************************

/*************************************************************************
 * Read "n" bytes from a descriptor.
 * Use in place of read() when fd is a stream socket.
 */
int readn(register int fd, register char *ptr, register int nbytes){
    int nleft, nread;

    nleft = nbytes;
    while (nleft > 0) {
        nread = read(fd, ptr, nleft);
        if (nread < 0)
            return(nread);      /* error, return < 0 */
        else if (nread == 0)
            break;          /* EOF */

        nleft -= nread;
        ptr   += nread;
    }
    return(nbytes - nleft);     /* return >= 0 */
}
```

```
/*********************************************************************
 *
 * 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(register int fd, register char *ptr, register 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);
}
/*********************************************************************/
```

```
/*********************************************************************
 *
 * Write "n" bytes to a descriptor.
 * Use in place of write() when fd is a stream socket.
 */
int writen(register int fd, register char *ptr, register int nbytes){
    int nleft, nwritten;

    nleft = nbytes;
    while (nleft > 0) {
        nwritten = write(fd, ptr, nleft);
        if (nwritten <= 0)
            return(nwritten);       /* error */

        nleft -= nwritten;
        ptr   += nwritten;
    }
    return(nbytes - nleft);
}
/*********************************************************************/
$
```

Die Programme sind auf den beiden Rechnern zu übersetzen und zu starten. Zuerst muss der Server gestartet werden, ansonsten meldet der Client:

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

7.2.2. Zeitserver

Die Realisierung ist nicht nur in C/C++ möglich. Heute werden mehr und mehr **Netzprogramme in Java** kodiert.

Als Beispiel wird im Folgenden eine Socket-Kommunikation in Java gezeigt, bei der ein **Unix Zeitserver** nach der aktuellen Zeit gefragt wird.

Unix Zeitserver antworten auf Port 13. Der verwendete Server und der Port sind fest im Java Programm kodiert.

Ein Aufruf des Java Programms kontaktiert den Server, liest von einem Socket die Antwort und gibt sie auf dem Bildschirm aus:

```
$ java inettime ulab 13
Accessing ulab/13

Sat Jan 12 13:32:20 2002

$
```

```
$ cat inettime.java
import java.io.*;
import java.net.*;
class inettime {
    public static void main(String[] args) {
        String host;
        int port;
        if (args.length > 0)
            host = args[0];
        else
            host = "ulab";
        if (args.length > 1)
            port = Integer.parseInt(args[1]);
        else
            port = 13; // date port
        new inettime(host, port);
    }
```

```
    inettime(String host, int port) {
        System.out.println("Accessing " + host + "/" + port + "\n");

        try {
            Socket s = new Socket(host, port);  //create socket

            InputStreamReader in = new InputStreamReader(s.getInputStream());
            int c;
            do {
                c = in.read();
                if (c>0)
                    System.out.print((char) c);
            } while (c>0);
            System.out.print('\n');
        }
        catch(IOException e) {
            System.out.println("Error" + e);
        }
    }
}
$
```

Im Praktikum wird der Zeitserver per C-Programm aufzurufen sein. Eine Implementierung ist als Muster verfügbar.

7.2.3.Portscanner

Wird eine Socket-Verbindung mit einem Rechner auf einem **Port** versucht, für den es **keinen Service** gibt, der auf dem Port lauscht, wirft der **Konstruktor** der Klasse Socket einen **Ausnahme**. Das kann man ausnutzen, um einen elementaren **Port-Scanner** zu implementieren:

PortScannerTCP.java

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

public class PortScannerTCP {  // scan ports on remote host

  public static void main(String[] args) {
    String host = (args.length == 0) ? "localhost" : args[0];
    try {                    // get remote address
      InetAddress address = InetAddress.getByName(host);
      for (int port = 0; port < 65535; port++) {
        try {
          Socket s = new Socket(address, port);
          System.out.println("Service detected listening: " +
                             host + "/" + port);
          s.close();
        } catch (IOException e) {
          // remote host is not listening on this port
        }
      } // for each port
    } catch (UnknownHostException e) {
      System.err.println(host + " not a valid host name!");
    }
  }  // main
}
```

Der Portscanner ermittelt die Ports, die auf dem überprüften Rechner offen sind, d.h. bei denen ein Verbindungswunsch beantwortet wurde:

```
$ java PortScannerTCP hal
Service detected listening: hal/80
Service detected listening: hal/515
Service detected listening: hal/631
...
$
```

7.3.Datagramm Kommunikation

In einigen Anwendungsfeldern ist eine sichere **Pipeline-artige Kommunikation nicht erforderlich** oder gar nicht gewünscht. Vielmehr soll die Kommunikation so erfolgen, dass ein Sender eine Menge von Informationspaketen an einen Empfänger sendet, wobei es auch vorkommen darf, dass eines der **Pakete** auf dem Übertragungsweg **verloren gehen** kann. Wird beispielsweise Musik von einem Server geladen und sofort von einem Player abgespielt, ist das gelegentliche Verlieren eines kleinen Pakets tolerierbar; das Nachfordern des verlorenen Paketes, so wie es beim TCP Protokoll geschieht, wäre dem Live-Musikgenuss nicht dienlich.

Diese Art der Kommunikation ermöglicht ein **Datagramm**. Ein Datagramm ist eine über ein Netz versendete Nachricht, die unabhängig von anderen Nachrichten ist und deren Ankunft und Ankunftszeit nicht garantiert werden kann. In Java stehen dazu die Klasse `DatagramPacket` zur Verfügung, mittels der man unter zu Hilfenahme der Klasse `DatagramSocket` unzuverlässigen Paketverkehr realisieren kann.

Wir verdeutlichen die Datagram-Kommunikation wieder an einem Zeitserver, der Anfragen von Clients entgegen nimmt. Zunächst wird der Server erläutert.

TimeServer.java

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

public class TimeServer { // UDP
  private static int port =  9005;
  public static void main(String args[]) throws
    IOException {
    DatagramSocket socket = new DatagramSocket(port);
    System.out.println("Timesever listening on port " + port);
    while (true) {
      byte message[] = new byte[256];
      DatagramPacket packet = new DatagramPacket(message, message.length);
      socket.receive(packet);                  // receive request
      // Get response address/port for client from packet
      InetAddress address = packet.getAddress();
      int port = packet.getPort();
      String date = new Date().toString(); // build answer
      message = date.getBytes();
      packet = new DatagramPacket(message, message.length, address, port);
      socket.send(packet);                     // send answer
    }
  }
}
```

Zunächst wird ein DatagramSocket auf einem Port erzeugt, auf dem der Server dann „lauscht“. Ein DatagramPacket wird verwendet, und der Methode receive des Sockets als Parameter mit gegeben. Vom empfangenen Paket werden die Internet-Adresse und der Port (des Client) extrahiert, damit der Server weiß, wem er die Antwort senden soll. Die aktuelle Zeit wird ermittelt, in

ein Bytearray gepackt. Dieses Bytearray zusammen mit der Client-Adresse und dem Client-Port werden in einem DatagramPacket gebündelt und über den Socket an den Client gesendet.

Die Realisierung des Klienten erfolgt analog.

TimeClient.java

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

public class TimeClient { // UDP
  static String host = new String();
  static int port = 9005;
  static void usageCheck(String[] args) {
   ...
  } // usage check

  public static void main(String args[]) throws IOException {
    usageCheck(args);
    DatagramSocket socket = new DatagramSocket();
    byte message[] = new byte[256];
    InetAddress address = InetAddress.getByName(host);
    DatagramPacket packet = new DatagramPacket(message,
                                               message.length,
                                               address, port);
    socket.send(packet);     // send request
    packet = new DatagramPacket(message, message.length);
    socket.receive(packet);  // receive answer
    String time = new String(packet.getData());
    System.out.println("The time at " + host + " is: " + time);
    socket.close();
  }
}
```

Weitere mögliche Themen: Secure Sockets, Multicasts

8. Remote Procedure Call

Die **Kommunikation** ursprünglich über **Ein-/Ausgabe-Primitive** wie `send` oder `receive` ist relativ umständlich und **aufwendig zu programmieren**. Das hat das letzte Beispiel gezeigt.

Gewünscht ist oft eine virtuelle Monoprozessorsicht, bei der eine Funktion oder Prozedur des Servers vom Klienten einfach aufgerufen werden kann, so als ob es eine lokale Funktion wäre.

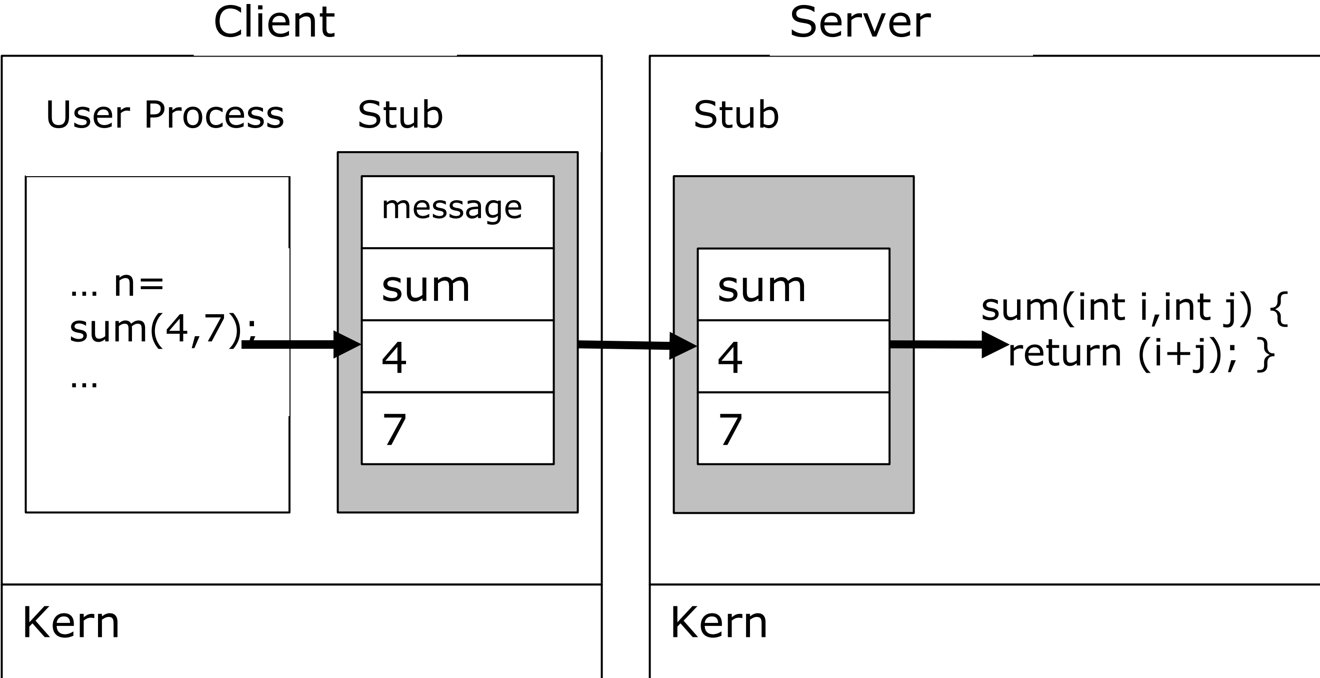
Ein **RPC-System** erlaubt den **Aufruf von Prozeduren auf anderen Rechnern**, inklusive der Übergabe von Parametern, basierend auf den Mechanismen eines **synchronen entfernten Dienstaufrufs**.

Der **Sender blockiert** bis der Aufruf komplett abgeschlossen ist, wie bei einer lokalen Prozedur.

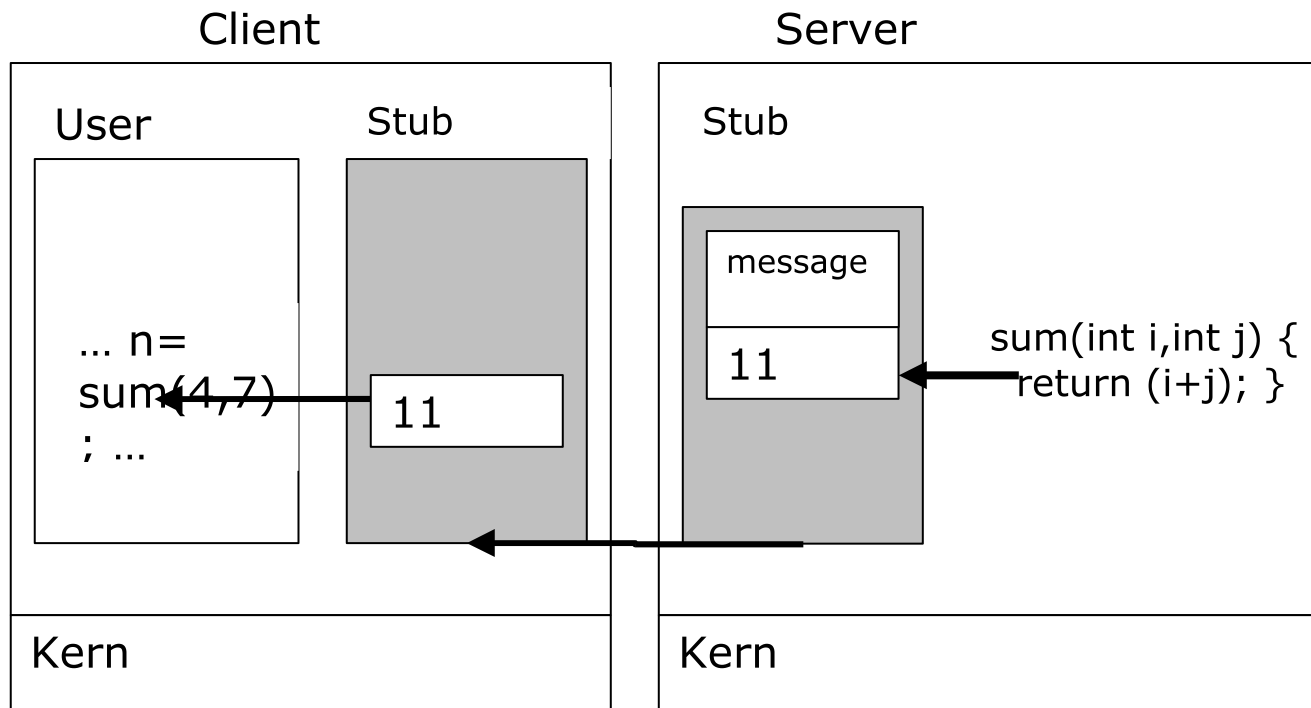
Ein Programmierer kann also nach dem **gleichen Paradigma** programmieren wie er es in einem **lokalen Programm** verwendet.

Die Idee des PRC ist an einem Beispiel folgend verdeutlicht:

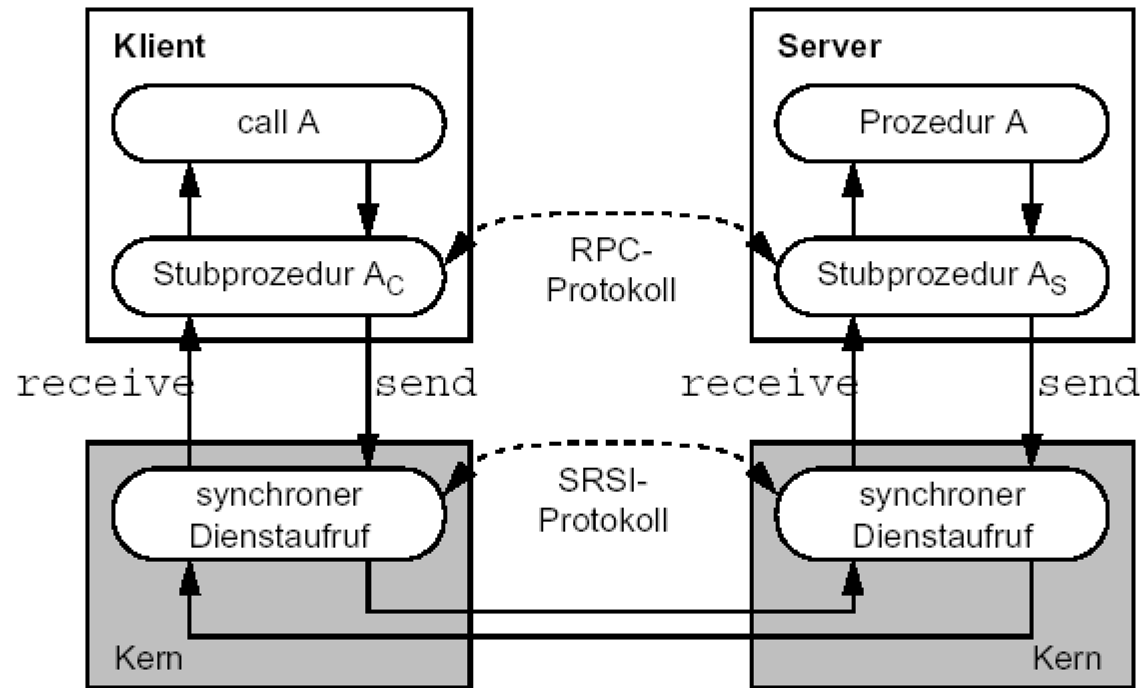
1. Funktionsaufruf



2. Ergebnis erhalten



8.1.Funktionsweise des RPC:



Aufgaben der Stubprozedur A_C

- Parameter der aufzurufenden Prozedur zu einer Nachricht zusammensetzen
- Serialisierungsprozeß = **Marshalling** (Verpacken der Parameter)

Aufgaben der Stubprozedur A_S

- Übergabeparameter rekonstruieren
- Deserialisierung = **Demarshalling** (Entpacken der Parameter)
- Akitiverung der Prozedur A
- Die Übertragung der Rückgabeparameter erfolgt analog.

RPC Besonderheiten und Einschränkungen:

- Da ein Kommunikationssystem involviert ist, ergeben sich zusätzliche Fehlerquellen.
- Da auf zwei verschiedenen Rechnern gearbeitet wird, gibt es **keinen gemeinsamen Adressraum**. Dies hat großen Einfluss auf die Parameterübergabe.
- Die beteiligten Prozesse haben separate Lebenszyklen. Es ist nicht sicher, ob die entfernte Prozedur immer ansprechbar ist.
- Durch den **Kommunikations-Overhead** ist die Aufrufdauer um Größenordnungen länger als bei lokalen Aufrufen.
- Unterschiedliche Rechner stellen **Zahlen unterschiedlich** dar (1er Komplement, 2er Komplement)
- Rechner arbeiten teilweise mit **unterschiedlichen Zeichenkodes** (EBCDIC, ASCII)
- Bytes werden je nach Rechner unterschiedlich nummeriert (links nach rechts oder umgekehrt)

Eine Stub Prozedur hat folgenden grundlegenden Aufbau:

```
PROCEDURE AC (Parameterliste)
BEGIN
    Zusammensetzen der Auftragsnachricht AN aus den Parametern;
    send (server_address, AN);
    receive (EN);
    Ergebnisnachricht EN zerlegen;
    Return (Rückgabeparameter);
END;

PROCEDURE AS
BEGIN
    receive (AN);
    Nachricht AN zerlegen;
    Call A (Parameterliste);
    Zusammensetzen der Ergebnisnachricht EN aus Pesultatsparametern;
    send (client_address, EN);
END;
```

8.2.Parameterübergabe

Folgende Probleme treten bei der Parameterübergabe auf:

- Die **Parameter** können **nicht** auf dem **lokalen Stack** abgelegt werden.
- Da getrennte Adressräume vorliegen, müssen die lokalen Parameter in den entfernten Adressraum überführt werden.

8.2.1.Call-by-Value Parameter

- Im lokalen Fall wird in der gerufenen Prozedur eine Hilfsvariable angelegt, die mit dem übergebenen Wert geladen wird.
- Bei RPC genügt es, den Wert des betreffenden Parameters zu senden.
- **Achtung**:

  Bei z.B. einem Array nicht nur den Startzeiger übergeben, sondern die gesamte Datenstruktur.

8.2.2.Call-By-Reference Parameter

Bei der Call-By-Reference Übergabe treten zusätzlich Probleme auf:

- Durch den getrennten Adressraum kann **keine einfache Dereferenzierung** stattfinden.
- **Dynamische Datenstrukturen**, wie Bäume, müssen **serialisiert und komplett übertragen werden**.
- Die Semantik ändert sich dadurch zu Call-by-Copy und Call-by-Restore. (auch call-by-value/result)
- Es gibt auch die Möglichkeit nur einen Zeiger zu übergeben und jede Dereferenzierung gesondert per Nachricht anzufordern. Dies wird üblicherweise nicht realisiert.

8.2.3. Transfersyntax

Die **Parameter** müssen in einer **vereinbarten Syntax** ausgetauscht werden:

- Byteordnung,
- Zeichen- und Zahlenkodierung
- unterschiedliche Programmiersprachen mit unterschiedlichen Typen der Parameter

Bezüglich der Syntax existieren zwei Varianten:

- *Symmetrisches Verfahren*
  - Alle Klienten und Server verwenden ein kanonisches Format.
  - Alle Parameter werden in dieses Format übersetzt bevor sie übertragen werden.
  - Damit benötigt jeder Teilnehmer eigene Konvertierungsroutinen.
- *Asymmetrisches Verfahren*
  - Klienten schicken ihre Parameter in ihrem eigenen Format.
  - Sie benötigen deshalb keine Konvertierungsroutinen.
  - Server müssen allerdings dann bei n verschiedenartigen Klienten über n Konvertierungsroutinen verfügen.
  - Diese Asymmetrie kann auch zwischen Klienten und Servern vertauscht sein.

8.3.Beschreibung der RPC-Schnittstellen

Um den in eine Nachricht verpackten Prozeduraufruf korrekt **interpretieren** und behandeln zu können, muss der Aufruf exakt beschrieben werden.

Diese Beschreibung heißt **Signatur** und umfasst

- Name der Prozedur
- Typen der Ein- und Ausgangsparameter
- eventuell Ausnahmebehandlungsvorschriften für Fehlerfälle.

Die Signatur muss sowohl für den Klienten, als auch für den Server verfügbar sein.

Um die **Signaturen zu beschreiben**, wurden spezielle **Schnittstellenbeschreibungsspra-chen** entwickelt.

- Diese Sprachen dienen nicht nur dazu, die Signatur zu spezifizieren, sondern beinhalten zusätzliche Deklarationen, die eine **automatische Generierung der Stubprozeduren** ermöglichen.
- Diese Generatoren nennt man **RPC-Compiler**.

Beispiele einiger Schnittstellenbeschreibungssprachen sind:

- von der ISO: ASN.1 (Abstract Syntax Notation)
- bei **Suns ONC-RPC**: **XDR** (eXternal Data Representation)

- beim OSF-RPC (DCE): IDL (Interface Definition Language)
- bei **CORBA**: **IDL** (Interface Definition Language)
- bei Mach: Matchmaker

Beispiel:

```
result = add (request); mit request = a, b.
```

Die Schnittstellenbeschreibung in **XDR** (Dateibezeichnung add.x):

```
struct result { int x; };

struct request { int a;
                 int b;
};

program ADD_PROG {
                    version ADD_VERS {
                        result ADD (request) = 1;      Prozedurnummer
                    } = 1;                             Versionsnummer
} = 20000000;                                          Programmnummer
```

Der RPC Compiler "**rpcgen**" generiert daraus eine Header-Datei (add.h):

```
struct result {
          int x;
};

typedef struct result result;
bool_t xdr_result ();

struct request {
          int a;
          int b;
};

typedef struct request request;
bool_t xdr_request ();
#define ADD_PROG ((u_long) 20000000)
#define ADD_VERS ((u_long) 1)
#define ADD ((u_long) 1)

extern result *add_1();
```

8.4.Der Bindevorgang beim RPC

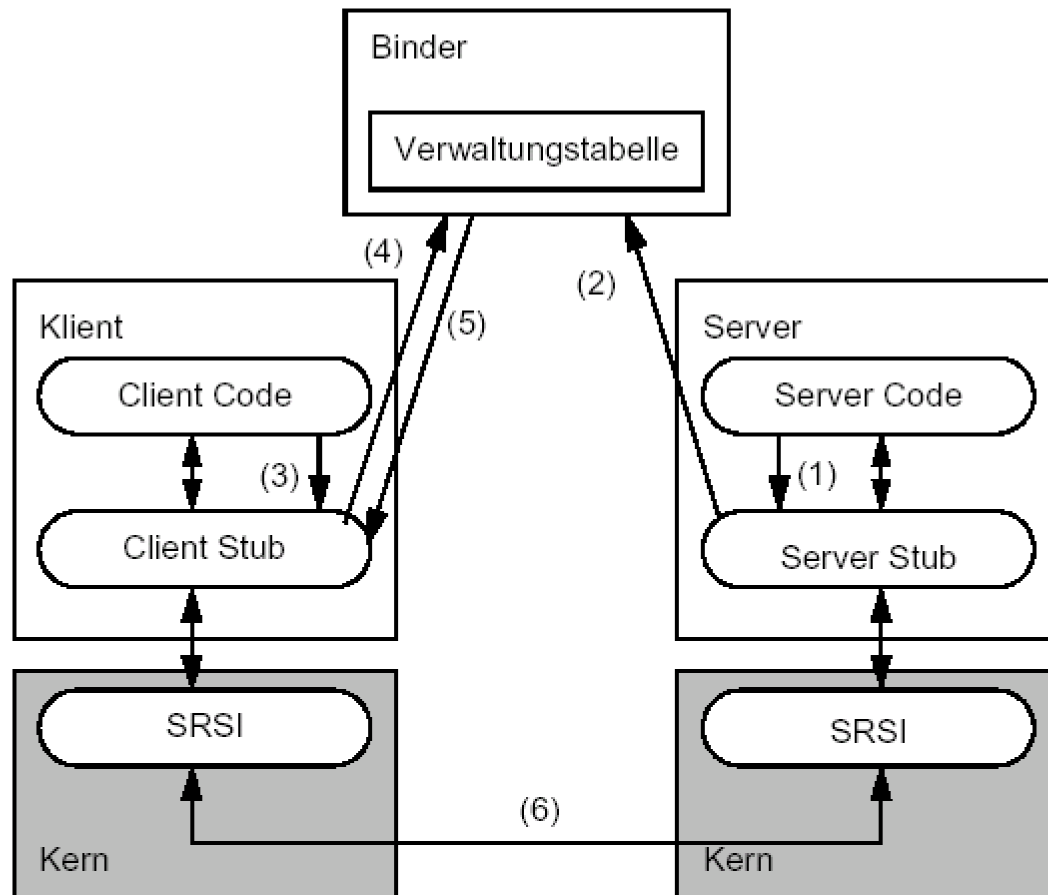
Bevor ein Klient einen Aufruf absetzt, muss festgelegt werden, welcher Server dafür in Frage kommt. Unterschieden werden

- *Fest programmierte Serveradresse*
    - Die vollständige Adresse des Servers ist fest ins rufende Programm kodiert.

- Dies führt zu statischer Bindung zur Compilezeit des Klienten.

- *Verwendung eines Binders*
  - Ein Server registriert seine anzubietenden Prozeduren explizit bei einem Binder
  - Der Binder verwaltet alle exportierten Prozeduren mit ihren Servern.
  - Ein Server kann einen Dienst auch wieder deregistrieren.
  - Die gespeicherte Information besteht z.B. aus
    `<Servername, Programm, Version, Handle>`
  - Ein Klient lässt sich einen Dienst vom Binder vermitteln
    (`lookup`).
  - Dazu erhält der Klient vom Binder einen Handle, mit dem er die Verbindung zum Server aufbauen kann.
  - Die Lokalisierung selbst kann
    - zur Initialisierung einer Anwendung (d.h. dynamisches Binden zur Initialisierungszeit) oder
    - direkt vor dem eigentlichen Aufruf (d.h. dynamisches Binden zur Laufzeit) geschehen.
  - Der Binder und ihre Verzeichnisse sind unter Umständen verteilt ausgelegt und/oder repliziert implementiert.

Der Bindevorgang lässt sich wie folgt verdeutlichen:



8.5.Implementierungsaspekte

Folgende Aspekte sind bei der Implementierung von RPC-Systemen zu beachten:

Aspekte des Kommunikationssubsystems

- *Wahl des Basisprotokolls*
    - Verbindungsorientierte Kommunikation übernimmt weitreichende Fehlerbehandlung außerhalb des RPC-Systems
    - Verbindungslose Kommunikation übernimmt weniger Fehlerbenhandlung. Das RPC-Laufzeitsystem muss eine weitreichendere eigene Fehlersemantik auf das Kommunikationssystem aufsetzen
- *Verwendung eines Standard- oder Spezialprotokolls*
    - Ein Standardprotokoll ist verbreitet und garantiert somit eine hohe Kompatibilität.
    - Es spart Entwicklungskosten
    - Ein Spezialprotokoll, das auf den Einsatz von RPCs zugeschnitten ist, arbeitet deutlich effizienter, ist aber nicht mehr kompatibel zu anderen Netzwerkprotokollen.
- Verwendung von Quittungen
    - Muss eine RPC-Nachricht fragmentiert werden, kann man jedes übermittelte Teilpaket quittieren lassen. Dies nennt man auch **Stop-and-Wait**-Protokoll
    - Um Kommunikationsaufwand zu sparen, kann man auch erst bei Erhalt des kompletten Paketes den Empfang bestätigen. Dies nennt man **Blast**-Protokoll.
    - Um erkennen zu können, wann ein Paket vollständig ist, muss zusätzlich eine Endekennung angefügt werden.

- Sind einzelne Teilpakete fehlerhaft, so können sie mittels selektiver Wiederholung angefordert werden.

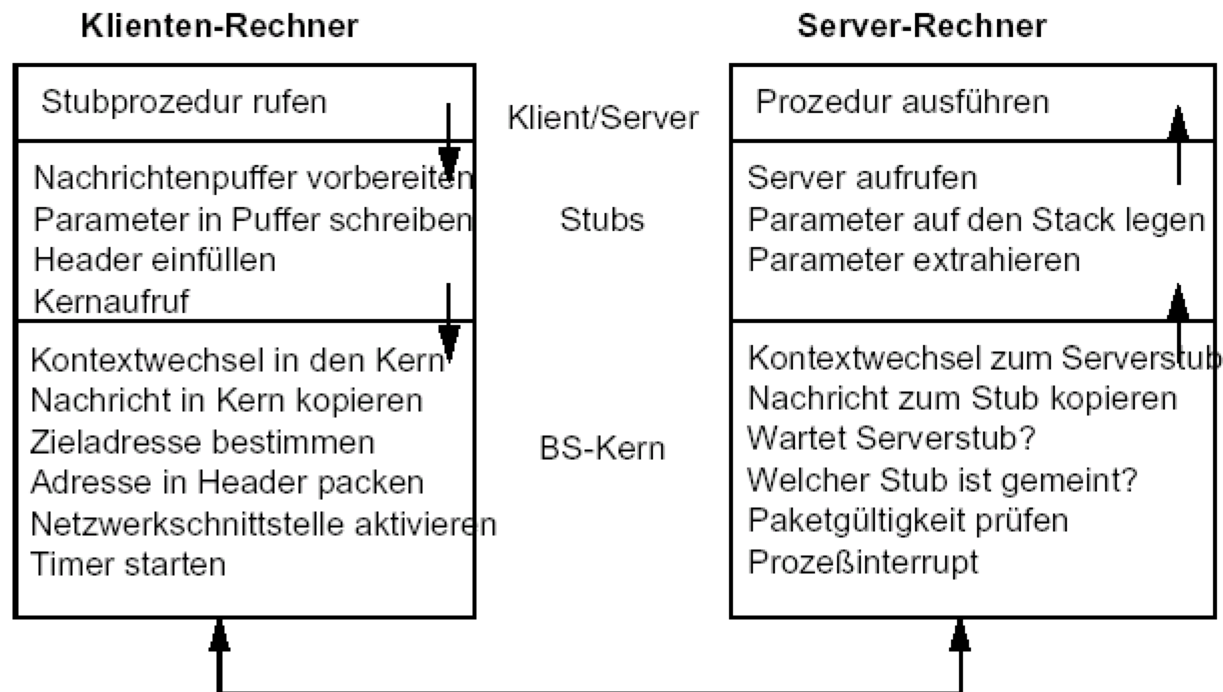
8.5.1. Kritischer Pfad des RPC

Der **zusätzliche Kommunikationsaufwand** bei RPCs ist im Vergleich zu lokalen Prozeduraufrufen der **zeitkritische Anteil**.

Man benötigt Zeit für

- das Marshalling und Demarshalling (Verpacken und Entpacken der Parameter),
- das Generieren von Paketen und Berechnen von Checksummen,
- den Wechsel zwischen dem Adressraum
  - der Anwendung zu dem
  - des Betriebssystems und dem
  - der Netzwerkhardware.
- Auf der Empfangsseite zusätzlich noch Demultiplexen zum Auffinden der richtigen Prozedur

Der kritische Pfad beim RPC kann wie folgt veranschaulicht werden:

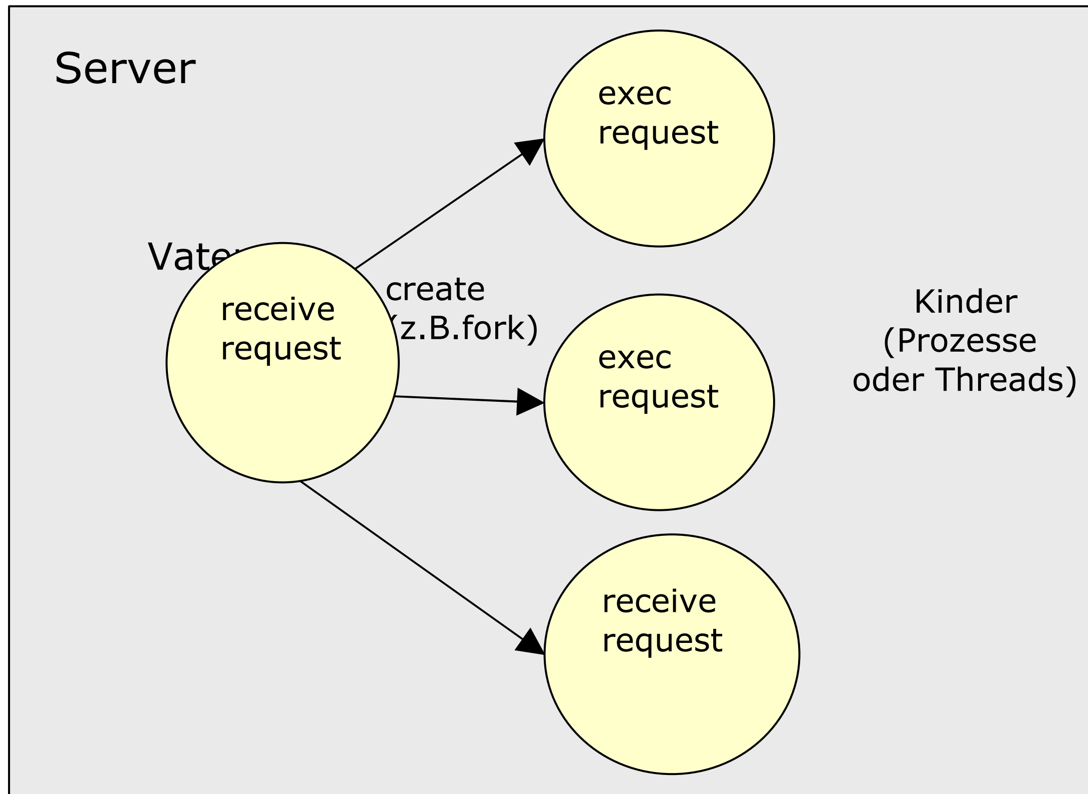


8.5.2. Prozessverwaltung beim Server

Um die Verfügbarkeit des Servers zu erhöhen existieren mehrere Ansätze (nicht gegenseitig ausschließende):

- Der Server kreiert für jeden eingehenden Aufruf einen neuen Prozess oder Thread (wie bei dem echoServer).

- Der Server entnimmt aus einem Pool einen schon bereitgestellten Prozess oder Thread.
- der Server wird schnell wieder empfangsbereit
- Solche Server nennt man multi-threaded Server oder Multiprozess-Server.



8.6. Fallbeispiel RPC (Beispiel des SUN RPC)

8.6.1. Grundsätzliche Vorgehensweise

Jede per RPC zu rufende **Prozedur** wird eindeutig durch eine **Programm**- und **Prozedurnummer identifiziert**.

Gruppen von zusammengehörigen **Prozeduren** sind in einem **Programm** zusammengefasst.

Jedes **Programm** hat eine **Versionsnummer**, um mehrere Releases oder Updates erfassen zu können.

Programmnummern:

```
0x0            - 0x 1fff ffff    # definiert von Sun
0x2000 0000    - 0x 3fff ffff    # Nutzerdefinierbar
0x4000 0000    - 0x 5fff ffff    # Transient
0x6000 0000    - 0x ffff ffff    # reserviert
```

Das Kommando **`rpcinfo`** zeigt die aktuell vorhandenen RPC-Programmnummern.

Der Binder heißt bei Sun-RPC **portmapper** und „hört" auf Port 111.

Als **Transportprotokoll** unterstützt Sun-RPC UDP und TCP.

XDR dient als Datenkodierungsvorschrift.

Der Protokoll-Compiler **`rpcgen`** generiert aus einer Datei mit

- Programmnummer,
- Versionsnummer,
- Prozedurnummern und
- XDR-Parameter- und Datenstrukturbeschreibungen

folgende Dateien:

- eine Header-Datei
- die Datenkonvertierungsroutinen,
- den Clientstub,
- den Serverstub und
- die Server-Hauptschleife.

Selbst zu programmieren bleiben

- die **Server-Prozeduren** und
- das **Klienten-Hauptprogramm**.

8.6.2.Beispiel addiere(a,b)

In diesem Beispiel wird ein RPC-Programm entwickelt, dass eine Berechnung auf einem entfernten Server-System vornimmt. Dabei soll die Routine `int addiere(int p1, intp2)` als RPC-Server implementiert werden. Addiere bildet die Summe der beiden Ganzzahlen p1 und p2 und liefert diese als Ergebnis zurück. Das Beispiel wurde auf MacOS entwickelt und auf Linux und Solaris getestet, sollte aber auch auf anderen Betriebssystemen mit RPC-Implementierung laufen.

Addiere als normale Prozedur

Zunächst zur Illustration der Aufgabe die Implementierung in einem Programm, d.h. addiere wird als normale Prozedur ausgeführt.

Der Quellcode ist nachfolgend gelistet und kann mit dem Befehl `cc -o addiere addiere.cpp` übersetzt werden.

```
$ cat addiere.cpp
# include <iostream>
using namespace std;
int addiere(int p1, int p2)
{
  return p1+p2;
}

int main(int argc, char **argv)
{
    cout << „addiere(5,6) liefert: „ << addiere(5,6) << endl;
    return 0;
}
$
```

Ein Aufruf von addiere auf einem Rechner liefert:

```
$ addiere
addiere(5,6) liefert 11
$
```

addiere als RPC

Um addiere als RPC-Routine zu implementieren, wird zunächst ein IDL-Interface (in **XDR**) angefertigt.

```
$ 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.

Dabei ist zu beachten, dass RPC die Übertragung **mehrerer Parameter** (`addiere` verwendet `p1` und `p2`) nur durch packen dieser Parameter in eine **Struktur** möglich ist `(add_struct)`. Variable Parameterlisten sind demnach nur mittels Unions möglich.

Der RPC-Compiler wird wie folgt gestartet:

```
$ rpcgen –a addiere.x
$
```

Durch Aufruf des RPC-Compilers werden die folgenden Dateien erzeugt:

addiere.h - Definitionen der Routinen und der RPC-ID's

addiere_clnt.c - Client-Stub

addiere_svc.c - Server-Stub

addiere_xdr.c - XDR-Konvertierungsroutinen für den Parameter-Struct

addiere_client.c - Ein Beispielprogramm für einen Client - mit leerem Aufruf

addiere_server.c - Ein Beispielprogramm für den Server - mit leerer Routine

Makefile.addiere - Ein Makefile zum Übersetzen

```
$ 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>


#ifdef __cplusplus
extern "C" {
#endif


struct add_struct {
        int p1;
        int p2;
};
typedef struct add_struct add_struct;

#define ADDIERE_TEST 1234567
#define ONE 1

#if defined(__STDC__) || defined(__cplusplus)
#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);
```

```
#else /* K&R C */
#define addiere 1
extern  int * addiere_1();
extern  int * addiere_1_svc();
extern int addiere_test_1_freeresult ();
#endif /* K&R C */

/* the xdr functions */

#if defined(__STDC__) || defined(__cplusplus)
extern  bool_t xdr_add_struct (XDR *, add_struct*);

#else /* K&R C */
extern bool_t xdr_add_struct ();

#endif /* K&R C */

#ifdef __cplusplus
}
#endif

#endif /* !_ADDIERE_H_RPCGEN */
$
```

```
$ cat addiere_clnt.c    Client Stub
/*
 * Please do not edit this file.
 * It was generated using rpcgen.
 */

#include <memory.h> /* for memset */
#include "addiere.h"

/* Default timeout can be changed using clnt_control() */
static struct timeval TIMEOUT = { 25, 0 };

int *
addiere_1(add_struct *argp, CLIENT *clnt)
{
        static int clnt_res;

        memset((char *)&clnt_res, 0, sizeof(clnt_res));
        if (clnt_call (clnt, addiere,
                (xdrproc_t) xdr_add_struct, (caddr_t) argp,
                (xdrproc_t) xdr_int, (caddr_t) &clnt_res,
                TIMEOUT) != RPC_SUCCESS) {
                return (NULL);
        }
        return (&clnt_res);
}
$
```

```
$ cat addiere_svc.c    Server Stub
/*
 * Please do not edit this file.
 * It was generated using rpcgen.
 */

#include "addiere.h"
#include <stdio.h>
#include <stdlib.h>
#include <rpc/pmap_clnt.h>
#include <string.h>
#include <memory.h>
#include <sys/socket.h>
#include <netinet/in.h>

#ifndef SIG_PF
#define SIG_PF void(*)(int)
#endif
```

```
static void
addiere_test_1(struct svc_req *rqstp, register SVCXPRT *transp)
{
  union {
          add_struct addiere_1_arg;
  } argument;
  char *result;
  xdrproc_t _xdr_argument, _xdr_result;
  char *(*local)(char *, struct svc_req *);

  switch (rqstp->rq_proc) {
    case NULLPROC:
     (void) svc_sendreply (transp,( xdrproc_t)xdr_void, (char *)NULL);
     return;

    case addiere:
     _xdr_argument = (xdrproc_t) xdr_add_struct;
     _xdr_result = (xdrproc_t) xdr_int;
     local = (char *(*)(char *, struct svc_req *)) addiere_1_svc;
     break;
    default:
      svcerr_noproc (transp);
      return;
  }
  memset ((char *)&argument, 0, sizeof (argument));
  if (!svc_getargs (transp, _xdr_argument, (caddr_t) &argument)) {
              svcerr_decode (transp);
              return;
  }
  result = (*local)((char *)&argument, rqstp);
  if (result != NULL && !svc_sendreply(transp, _xdr_result, result)) {
```

```
		svcerr_systemerr (transp);
	}
	if (!svc_freeargs (transp, _xdr_argument, (caddr_t) &argument)) {
		fprintf (stderr, "unable to free arguments");
		exit (1);
	}
	return;
}
```

```
int
main (int argc, char **argv)
{
	register SVCXPRT *transp;

	pmap_unset (ADDIERE_TEST, ONE);

	transp = svcudp_create(RPC_ANYSOCK);
	if (transp == NULL) {
		fprintf (stderr, "cannot create udp service.");
		exit(1);
	}
	if (!svc_register(transp, ADDIERE_TEST,
			ONE, addiere_test_1, IPPROTO_UDP)) {
		fprintf (stderr, "unable to register (ADDIERE_TEST, ONE, udp).");
		exit(1);
	}

	transp = svctcp_create(RPC_ANYSOCK, 0, 0);
	if (transp == NULL) {
		fprintf (stderr, "cannot create tcp service.");
		exit(1);
	}
	if (!svc_register(transp, ADDIERE_TEST,
			ONE, addiere_test_1, IPPROTO_TCP)) {
		fprintf (stderr, "unable to register (ADDIERE_TEST, ONE, tcp).");
		exit(1);
	}

	svc_run ();
	fprintf (stderr, "svc_run returned");
```

```
        exit (1);
        /* NOTREACHED */
}

$
```

```
$ cat addiere_xdr.c  XDR-Konvertierungsroutinen für Parameter-Struct
*
 * Please do not edit this file.
 * It was generated using rpcgen.
 */

#include "addiere.h"

bool_t
xdr_add_struct (XDR *xdrs, add_struct *objp)
{
        register int32_t *buf;

         if (!xdr_int (xdrs, &objp->p1))
                 return FALSE;
         if (!xdr_int (xdrs, &objp->p2))
                 return FALSE;
        return TRUE;
}
$
```

```
$ cat addiere_client.c
/*
 * 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;

	result_1 = addiere_1(&addiere_1_arg, clnt);
	if (result_1 == (int *) NULL) {
		clnt_perror (clnt, "call failed");
	}
}

int
main (int argc, char *argv[])
{
	char *host;

	if (argc < 2) {
		printf ("usage: %s server_host\n", argv[0]);
		exit (1);
```

```
    }
    host = argv[1];
    addiere_test_1 (host);
    exit (0);
}
$
```

```
$ cat addiere_server.c
/*
 * 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
         */

        return &result;
}
$
```

Jetzt müssen noch die beiden Server - und Clientbeispielprogramme ergänzt werden, das Programm soll die Summe von 5 und 6 errechnen.

Dazu werden die Parameter im Client-Programm gesetzt und die Ausgabe hinzugefügt sowie der Server um die Addition ergänzt. Damit ergeben sich die neuen Dateien:

```
$ cat addiere_client.c
void
addiere_test_1(char *host)
{
        CLIENT *clnt;
        int  *result_1;
        add_struct  addiere_1_arg;

        /* my code START */
        addiere_1_arg.p1 = 5;
        addiere_1_arg.p2 = 6;
        /* 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(5,6) liefert %d\n", *result_1);
        }
        /* my code END */
}

int
main (int argc, char *argv[])
{
        char *host;
```

```
        if (argc < 2) {
                printf ("usage: %s server_host\n", argv[0]);
                exit (1);
        }
        host = argv[1];
        addiere_test_1 (host);
exit (0);
}
$
```

```
$ cat addiere_server.c
/*
 * 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;
}
$
```

Nun wird das Programm übersetzt:

```
$ make -f makefile.addiere
…
$
```

Dabei werden die Programme `addiere_server` und `addiere_client` erzeugt. Daraufhin kann der Server durch den Befehl "`addiere_server`" gestartet werden, der Client (auch auf einem anderen Rechner) über "`addiere_client hostname`", z.B.

```
$ addiere_server
$ addiere_client localhost
addiere(5,6) lieferte 11
$
```

Der Server bleibt ca. 2min nach dem ersten Aufruf aktiv (das Timeout kann durch `rpcgen`-Parameter geändert werden, siehe `man rpcgen`.

Ist der Server nicht gestartet, komm die Fehlermeldung:

```
$ addiere_client localhost
call failed: RPC: Unable to receive; errno = Connection refused
$
```

Den kritischen Pfad des RPC sollte man sich verdeutlichen, indem man die lokale Prozedur und die RPC-Version gegenüberstellt. Verwendet werden kann das Kommando time.

```
$ time addiere
addiere(5,6) liefert 11

real    0m0.004s
user    0m0.000s
sys     0m0.000s
$
$ time addiere_client localhost
addiere(5,6) liefert 11

real    0m0.006s
user    0m0.000s
sys     0m0.000s
$
```

Der Unterschied wird noch größer, wenn man den Server auf einem Remote Rechner aufruft.

Die folgenden Skripten können verwendet werden, um das Verhalten zu testen:

```
$ cat addiere.run
anz=$1
if [ $# -ne 1 ]
then
        echo "usage $0 <Anzahl Lauefe>"
        exit 1
fi

while [ $anz -gt 0 ]
do
        addiere > /dev/null
        anz=`expr $anz - 1`
done
$
$ cat addiere_client.run
host=$1
anz=$2
if [ $# -ne 2 ]
then
        echo "usage $0 <host> <Anzahl Lauefe>"
        exit 1
fi

while [ $anz -gt 0 ]
do
        addiere_client $host > /dev/null
        anz=`expr $anz - 1`
done
$
```

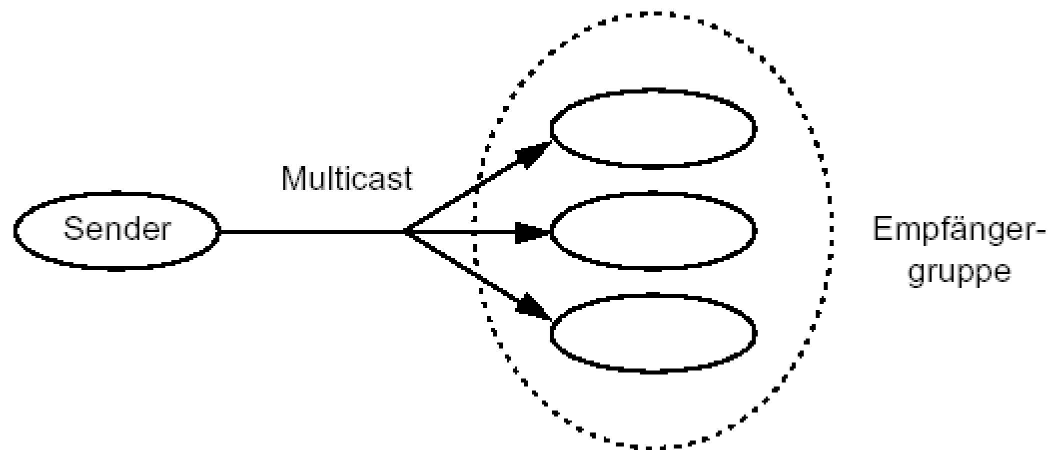
```
$ time addiere.run 100

real    0m0.773s
user    0m0.380s
sys     0m0.400s
$
$ time addiere_client.run localhost 100

real    0m1.113s
user    0m0.540s
sys     0m0.530s
$
```

9. Gruppenkommunikation

Die bisher betrachteten Kommunikationsmodelle waren Punkt-zu-Punkt-Szenarien.

Die Kommunikationsform, bei der **ein Sender mehrere Empfänger** erreichen kann, nennt man Gruppenkommunikation (1:m Kommunikation).

Die Versendung einer Nachricht an die Gruppe erfolgt durch einen Multicast.



Typische Anwendungen für Gruppenkommunikation sind:

- **Fehlertoleranz** wird durch replizierte Dienste gewährleistet, wobei alle Server identische Aufträge erhalten.
- Um die **Lokation** von Objekten zu bestimmen, schickt man eine Nachricht an eine Gruppe potentieller Server.

- Bei **Datenreplikation** werden die Änderungen über Gruppenkommunikation an die replizierten Objekte bekannt gegeben.
- In **Newsdiensten** werden die Daten über Gruppenkommunikation verteilt.
- **Konferenzsysteme** und Gruppenanwendungen (z.B. IRC) bedienen sich einer Gruppenkommunikations-Semantik.
- In **Händlerfrontsystemen** in Banken werden Kurse an Händlergruppen verteilt (->Tafel)

Folgende **Realisierungsvarianten** können unterschieden werden:

- Realisierung über **multicastfähige Netzwerke** (z.B.: Ethernet) ist relativ einfach; jede Gruppe erhält eine Multicastadresse
- Eine 1:m Kommunikation kann auch mittels **Broadcast** erreicht werden, aber
  - keine Möglichkeit einer dynamischen Gruppenadressierung.
  - alle Rechner im Netzwerk erhalten die Nachricht
  - unbeteiligte Empfänger müssen die Nachricht verwerfen
- Steht im Netzwerk weder Multicast noch Broadcast zur Verfügung kann man Multicast durch **Folge von Unicasts** nachbilden.

9.1. Entwurfskriterien

Folgende Kriterien sind für Gruppenkommunikation beim Entwurf zu betrachten.

9.1.1. Wahl der Gruppenform

Geschlossene Gruppen:

Nur Mitglieder der Gruppe dürfen Gruppennachrichten versenden.

Offene Gruppen

Hier können auch Sender, die nicht der Gruppe angehören, Nachrichten an die Gruppe senden.

9.1.2. Primitive für die Gruppenverwaltung

Bei der Gruppenkommunikation lassen sich folgende **Basisprimitive** unterscheiden:

```
GroupId = CreateGroup (ListOfPortIds);
JoinGroup (IndividualPortId, GroupId);
LeaveGroup (IndividualPortId, GroupId);
DeleteGroup (GroupId);
```

Die Verwaltung der Gruppen erfolgt

- dedizierter Gruppen-Server oder
- verteilt unter den Mitgliedern

Kommunikationsprimitive sind:

```
SendtoGroup (GroupId, Message);
ReceivefromGroup (GroupId, Message);
```

9.2. Semantik der Gruppenkommunikation

9.2.1. Zuverlässigkeitsgrade

Der Zuverlässigkeitsgrad gibt an, wie viele Empfänger aus der Gruppe mit Nachrichten bedient werden.

- ***keine Zuverlässigkeit***
  Es gibt keine Garantie darüber, wie viele Empfänger und welche Empfänger eine Gruppennachricht tatsächlich erhalten.

- ***k-zuverlässig***
  Es wird garantiert, dass **mindestens k Mitglieder** der Empfängergruppe eine Gruppennachricht erhalten.

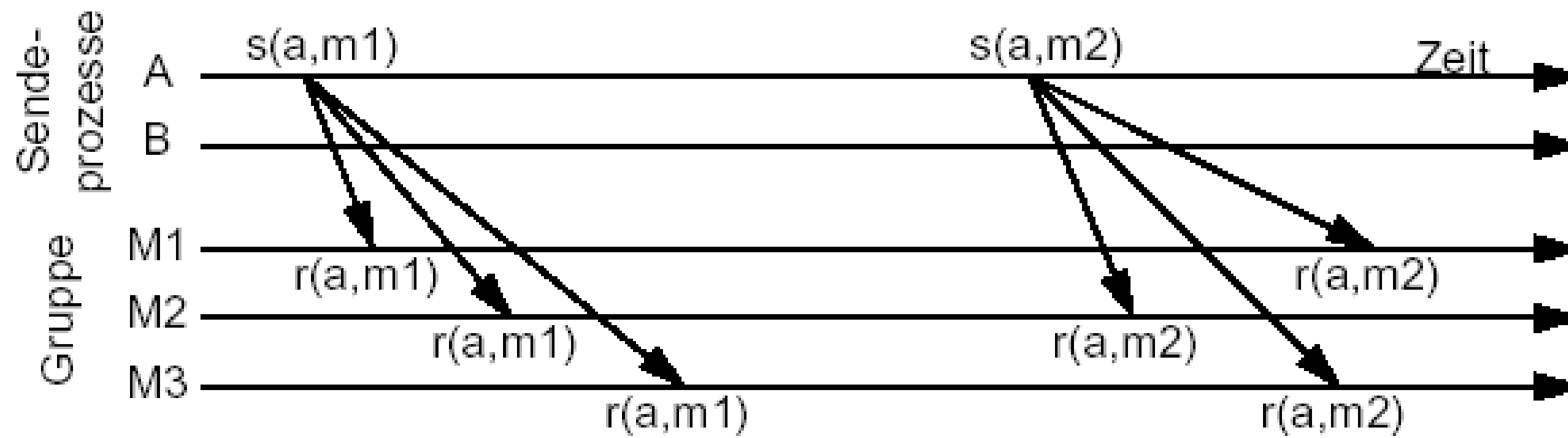
- ***atomar***
  Es wird sichergestellt, dass entweder **alle Mitglieder** der Empfängergruppe eine Gruppennachricht erhalten **oder keines**.

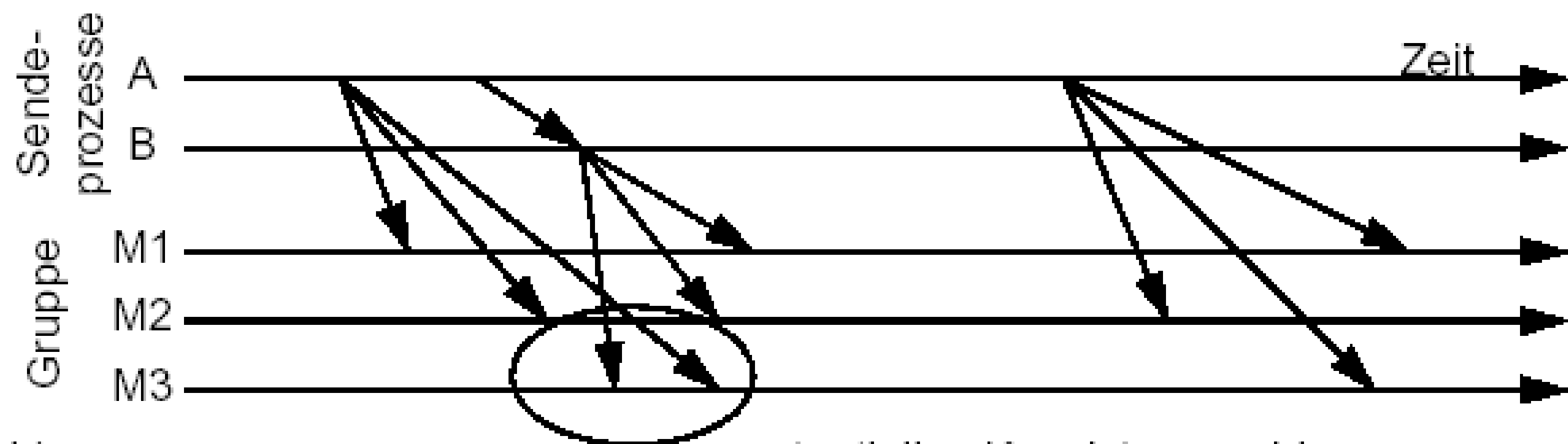
9.2.2.Ordnungsgrade

Der Ordnungsgrad **beschreibt** die **Reihenfolge**, in der Gruppennachrichten von den Gruppenmitgliedern **empfangen** werden.

- ***ungeordnet***
  Es besteht **keine festgeschriebene Empfangsreihenfolge** der Nachrichten. D.h. jede Teilnachricht einer Gruppennachricht kann jede andere Teilnachricht des gleichen Senders oder anderer Sender überholen.

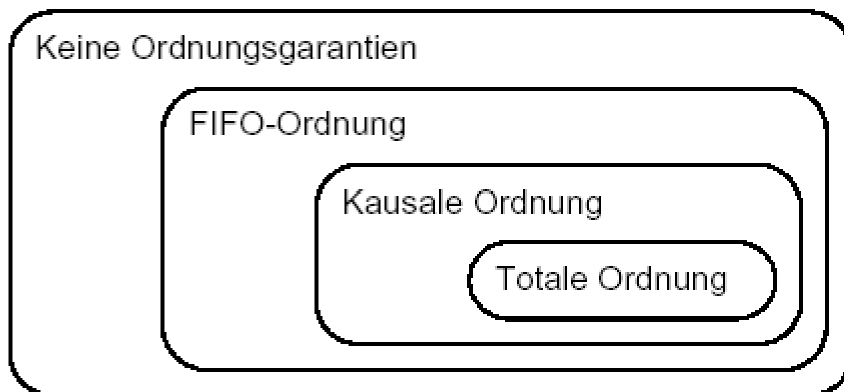
- ***FIFO-geordnet***
  Alle Gruppennachrichten ein und desselben Senders an eine Gruppe kommen bei allen Mitgliedern der Gruppe in der gesendeten Reihenfolge an.
  Das folgende Schaubild verdeutlicht FIFO basierte Gruppenkommunikation und auftretende Probleme:


Sende-
prozesse
Gruppe
A
B
M1
M2
M3
s(a,m1)
s(a,m2)
Zeit
r(a,m1)
r(a,m1)
r(a,m1)
r(a,m2)
r(a,m2)
r(a,m2)
a)


Sende-
prozesse
Gruppe
A
B
M1
M2
M3
Zeit
b)
⇒ potentielles Konsistenzproblem

- ***kausal geordnet***
  Falls eine Gruppennachricht m_b **kausal abhängig** von m_a ist ($m_a < m_b$), so erhalten alle Mitglieder der Gruppe die Nachricht m_b erst nach dem Empfang von m_a.
  Kausal unabhängige Gruppennachrichten sind weiterhin ungeordnet.
- ***total geordnet***
  Zusätzlich zur kausalen Ordnung werden alle nicht kausal abhängigen Nachrichten in einer beliebigen, aber für alle Mitglieder der Gruppe gleichen Reihenfolge empfangen.

Dies führt zur folgenden Übersicht der Ordnungsgrade:



9.3.Implementierung von Gruppenkommunikation

Die Implementierung von Gruppenkommunikation **hängt** stark von der **Unterstützung** der verwendeten **Netzwerke ab**.

Die Simulation eines Multicasts durch einfache Programmschleife hat etwa folgendes Aussehen:

```
PROCEDURE multicast (dest: ARRAY OF PortId; m: Message);
VAR i:INTEGER;
BEGIN
    FOR i:= 0 TO MAX(dest) DO
        send (dest[i], m)
    END;
END;
```

Nachteile der o.a. Lösung:

- keine Garantie über die Ankunft der verschickten Nachrichten
- keine Ordnungskriterien
- ineffiziente Implementierung

9.3.1.Implementierungsaspekte zur Effizienz

Multicastfähigkeiten auf den **unteren Ebenen** des OSI-Modells (Schicht 2 und 3)

- bereits die **Hardware** kann viele der Verwaltungs- und Verteilungsaufgaben übernehmen.

In **lokalen Netzen** mit Broadcastfähigkeit

- jeder Rechner ist mit nur einer einzigen Nachricht erreichbar
- auf jedem Rechner müssen die gewünschten Gruppenteilnehmer herausgefiltert werden
- über ein Netzwerksegment hinaus muss ein Unicast-Tunnel eingerichtet werden

Ist im Netz nur Punkt-zu-Punkt-Adressierung möglich

- Implementierung durch eine Schleife

9.3.2.Implementierungsaspekte zur Zuverlässigkeit

Folgende Fehlerfälle können auftreten:

- Nachrichten können verloren gehen
- Der Gruppenserver kann abstürzen

Dies hat zur Folge:

- Ein Teil der Empfänger erhält die Nachricht nicht.
- Weder der Sender, noch die Empfänger haben eine Kontrolle darüber, wer alles eine Nachricht erhalten hat.
- Bei geordneten oder gar **atomaren Multicasts** ist eine **Überwachung**, ein so genanntes **Monitoring**, **unerlässlich**

Ein oftmals angewendetes **Monitoring-Verfahren** sieht wie folgt aus:

- Der **Sender prüft** ständig, ob **alle Empfänger** noch **aktiv** sind, um Information über die Vollständigkeit der Gruppe zu haben.
- Meldet sich ein Empfänger eine bestimmte Zeit lang nicht, so wird davon ausgegangen, dass dieser nicht mehr vorhanden ist.
- Er wird aus der **Gruppe genommen**.
- Zukünftige Nachrichten, die das ausgeschlossene Gruppenmitglied dennoch sendet, müssen ab dem Zeitpunkt des Ausschlusses ignoriert werden.

Zuverlässiger, atomarer Multicast

Eine mögliche **Implementierung** für zuverlässige, atomare Multicasts ist:

- Der **Sender verschickt** einen **Multicast** und **sammelt** danach von allen Empfängern eine **Empfangsbestätigung** ein.
- Sind alle **Quittungen eingegangen**, schickt der Sender **erneut** einen **Multicast**, um mitzuteilen, dass der vorherige Multicast abgeschlossen ist.
- Ein **Monitor entfernt** dabei **Mitglieder**, die sich fehlerhaft verhalten, aus der Gruppe.
- Um den Fall eines **Senderabsturzes** zu berücksichtigen, **prüfen** die **Empfänger**, welche bereits die Nachricht erhalten haben, den Sender.
- Sollte der **Sender ausfallen**, so **übernimmt** einer der **Empfänger** den Rest des noch ausstehenden Multicastprotokolls.

Den **Multicast** als **abgeschlossen** zu melden, bedeutet erneut einen Multicast zu verschicken. Diese rekursive Semantik muss künstlich terminiert werden.

Folgende verbesserte Techniken wurden realisiert:

- ***Hold-back***
  - Das Multicastsystem wird im Betriebssystem verankert.
  - Die **Zustellung** der Nachricht an den **Anwendungsprozess** wird solange **zurückgehalten**, bis die Ordnungs- bzw. Zuverlässigkeitskriterien gewährleistet sind.

- Vorteil:
  das Betriebssystem hat bessere Privilegien und optimierte Zugriffsmöglichkeiten. Dies erhöht die Effizienz.

- ***Negative Acknowledgement***
  - Jeder **Sender nummeriert** seine Multicastnachrichten mit einer **Sequenznummer**.
  - Die **Empfänger** merken sich zu jedem Sender die eingehenden **Sequenznummern**.
  - Weist bei einem Empfänger die Reihe der Sequenznummern eine **Lücke** auf, so **fordert** er die **fehlende Nachricht** mit einer Negativquittung **nach**.
  - Dazu muss jeder Sender eine Historie seiner gesendeten Nachrichten speichern.
  - Für den Übernahmefall, falls ein Sender ausfällt, müssen die Empfänger ihre empfangenen Nachrichten ebenfalls speichern.
  - Das Problem ist, dass die Historienspeicher nicht zu groß werden sollen. Deshalb werden ab und zu positive Quittungen verschickt, welche die höchste Sequenznummer, die alle Empfänger gesehen haben, enthält.
  - Um zusätzliche Nachrichten zu vermeiden, werden diese Quittungen zu den üblichen Multicasts dazugepackt (engl.: piggybacking).

Total geordneter Multicast

Um ein **Ordnungskriterium** für eine totale Ordnung zu haben, benötigt man **eindeutige Sequenznummern** über alle Nachrichten.

Das **Multicastsystem puffert** die eingehenden Nachrichten in FIFO-Semantik und **leitet** diese erst an ihre **Anwendung** weiter, falls alle Nachrichten eine fortlaufende Sequenznummer ohne Lücke aufweisen.

Dadurch werden alle Nachrichten bei allen Empfängern in der gleichen Reihenfolge an die Anwendungen gegeben.

Generierung eindeutiger Sequenznummern

Folgende **Methode** zur **Sequenzerzeugung** findet man.

- Es werden **Zeitstempel** einer eindeutigen, gemeinsamen logischen oder physikalischen Uhr verwendet.
- Ein **zentraler Sequenzer**, an den alle Nachrichten geschickt werden, hängt die eindeutige Sequenznummer an und verschickt den Multicast. (siehe Amoeba)
- Es läuft ein **Protokoll** zwischen den Mitgliedern, um die Sequenzen zu generieren.

9.4.Fallbeispiel – Chat

Multicast-Kommunikation erfolgt über speziell dafür **reservierte IP-Adressen**, die so genannten **Klasse D** Adressen im Bereich 224.0.0.0 bis 239.255.255.255. Zur (internen) Übertragung von Routing-Informationen ist dabei der Anfangsbereich von 224.0.0.0 bis 224.0.0.255 reserviert. Jeder Multicast-Nachricht beinhaltet eine Art **Lebensdauer**, die Time-to-Live Information (**TTL**), die ausdrückt, wie viele Router die Multicast-Nachricht passieren kann. Jeder Router dekrementiert den TTL-Wert einer empfangenen Multicast-Nachricht, bevor sie weitersendet. Dadurch kann die Ausbreitung der Nachricht begrenzt werden, durch den Standardwert 1 von TTL wird erreicht, dass Multicasts nur im lokalen Netzsegment sichtbar sind.

In Java wird eine Gruppe durch einen **`MulticastSocket`** aus dem Multicast-Adressbereich und einem beliebigen Port definiert. Um eine Gruppe anzulegen, wird zunächst ein `MulticastSocket` instanziiert. **Mitglied** in einer Gruppe kann man durch Aufruf der Methode **`joinGroup`**`()` der Klasse `MulticastSocket` werden, indem die Gruppe durch einen als Parameter mitzugebenden `MulticastSocket` angegeben wird. Die Mitgliedschaft wird beendet durch `leaveGroup()`. **Sende- und Empfang-Operationen** werden in gewohnten Form über **`DatagramPacket`** Objekte über den die Gruppe definierenden Socket realisiert. Ein erstes Beispiel eines Programms erzeugt eine Gruppe, wird Mitglied der Gruppe, sendet eine Nachricht an die Gruppe und hört dann alle Gruppennachrichten ab und gibt sie aus.

Chat.java:

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

public class Chat{
  static String mcAddr = "236.0.0.1";   // multicast address
  static int port = 9006;
  static int bufLen = 512;
  public  static void main(String[] args) throws IOException {
    String id = (args.length == 0) ? "" : args[0];
    try {
      InetAddress group = InetAddress.getByName(mcAddr);
      MulticastSocket socket = new MulticastSocket(port);
      socket.joinGroup(group);
      new Receiver(group, port, socket, bufLen).start();
      new Sender(id, group, port, socket, bufLen).start();
    } catch(IOException e) {
      e.printStackTrace();
      System.exit(1);
    }
  }
} // Chat
```

```
class Receiver extends Thread {
  InetAddress group = null;
  MulticastSocket socket = null;
  int port = 0;
  int bufLen = 0;
  Receiver(InetAddress group, int port, MulticastSocket socket, int bufLen) {
    this.group = group;
    this.port = port;
    this.socket = socket;
    this.bufLen = bufLen;
  }
  public void run() {
    try {
      while (true) {         // get group messages
        byte[] buf = new byte[bufLen];
        DatagramPacket recv = new DatagramPacket(buf, buf.length);
        socket.receive(recv);// receive group message
        byte[] data = recv.getData();
        System.out.println("                            |" +
                           new String(data)); // show message
        System.out.print("> ");       // generate user prompt
      } // while
    } catch(IOException e) {
      e.printStackTrace();
      System.exit(1);
    }
  } // run
} // Receiver
```

```
class Sender extends Thread {
  InetAddress group = null;
  MulticastSocket socket = null;
  int port = 0;
  int bufLen = 0;
  String id;
  Sender(String id, InetAddress group, int port,
         MulticastSocket socket, int bufLen) {
    this.group = group;
    this.port = port;
    this.socket = socket;
    this.bufLen = bufLen;
    this.id = id;
  }
```

```
 public void run() {
    try {
      BufferedReader stdIn = new BufferedReader(new InputStreamReader(System.in));
      String userInput;
      byte[] buf = new byte[bufLen];
      DatagramPacket packet = null;
      System.out.print("> ");
      while (true) {
        if ((userInput = stdIn.readLine()) == null) {
          System.exit(2);
        }
        StringBuffer message = new StringBuffer("(");
        message = message.append(id).append(") ").append(userInput);
        buf = message.toString().getBytes();
        packet = new DatagramPacket(buf, buf.length, group, port); // packet for group
        socket.send(packet);          // send packet to group
        System.out.print("  ");
      } // main loop
    } catch(IOException e) {
      e.printStackTrace();
      System.exit(1);
    }
  } // run
} // Receiver
```

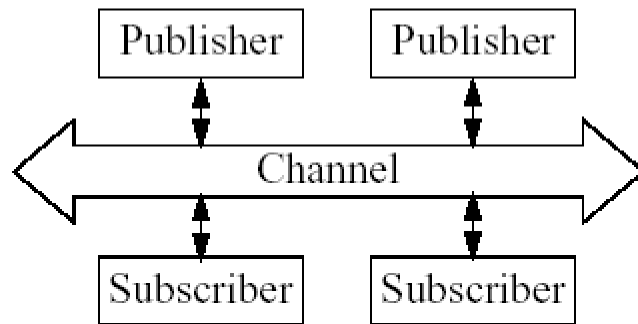
10. Messaging Systeme

Messaging Systeme sind folgendermaßen charakterisiert:

- mehrere Klienten sind an der gleichen Information interessiert
- Klienten „kommen und gehen“
- Objekte können nicht suspendiert werden, während Kommunikation abläuft
- Peer-to-Peer Kommunikationsmuster
- Messaging Agents unterstützen Erzeugung, Verschicken und Empfang von Nachrichten

10.1. Kommunikationsbeziehungen

- Point-to-Point
  - Nachrichten werden an Warteschlangen geschickt und von dort empfangen
- Publish/Subscribe
  - Klientenrollen: Publisher bzw. Subscriber
  - Channel o.ä. als Abstraktion



10.2. Kommunikationsmodelle

- Push
  asynchrones Zustellen bei den Empfängern
- Pull
  Empfänger holen Nachrichten synchron

10.3.Beispiele

- Java Message Service
- Java InfoBus
- CORBA Event Service
- CORBA Notification Channel