Systemprogrammierung
Test 2

Jan Fässler

3. Semester (HS 2012)

Inhaltsverzeichnis

1 Sockets

Die Socket Schnittstelle wurde an der University of California entwickelt. Ziel war es eine offene, generische Programmierschnittstelle für die Interprozesskommunikation für UNIX zu realisieren. Sockens erlauben es dem Nutzer sich mit dem Netzwerk zu verbinden. Der Socket ist dabei der Endpunkt einer Kommunikationsverbindung, wie zum Beispiel eine Telefonsteckdose. Die Anforderungen an die Schnittstelle waren:

Transparenz bezüglich Netzwerkverhalten

Effizienz um die Programmierer zu überzeugen

Kompatibilität mit dem UNIX Standard I/O

1.1 Grundkonzepte

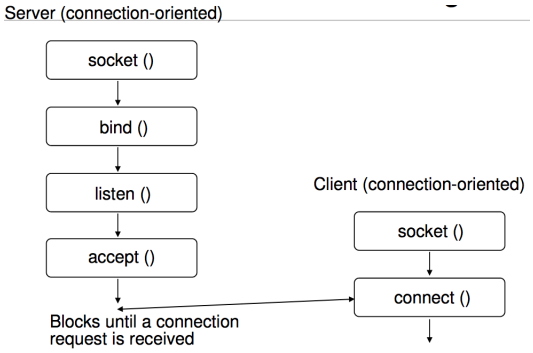
communication domains
Unterstützung verschiedener Netzwerkprotokolle

communication types
Klassifikation von Eigenschaften

name binding
Bennenung und Adressierung von Endpunkten

Sockets
Einheiltiche Abstraktion von Endpunkten

1.2 Erstellen einer Socket Verbindung



1.3 Senden und Empfangen von Daten

Um möglichst viele Domänen (Unix, Internet, ...) und Protokolle (TCP, UDP, ...) zu unterstützen, gibt es verschiedene Paare von System Calls welche aber in den meisten UNIX Varianten auf einer einzigen internen Funktion abgebildet werden.

read/write
Rückwärtskompatibilität zum UNIX Standard I/O, ermöglichen von Pipes mit dup()

send/recv
zusätzliche Funktionalität und Parametrisierung nutzt TCP, Packete geordnet

sendto/recvfrom
erlaubt senden ganzer Datagramme ohne Verbindung (UDP), Packete ungeordnet, keine Fehlerüberprüfung, zwei Buffer nötig (Sender/Empfänger)

sendmsg/recvmsg
nicht Bytestrom orientiert, Appl. wird erst benachrichtig wenn die ganze Nachricht angekommen ist.

1.4 Blockierendes vs. selektives Warten auf Sockets

accept()
Blockiert den aufrufenden PRozess solange, bis ein Verbindungsaufbauwunsch am Socket signalisiert wird

select()
Erlaubt nicht blockierendes Warten auf eine Verbindung

1.5 Beenden einer Verbindung

shutdown()
Signalisiert Verbinungsabbruch der anderen Seite und wartet auf eine Bestätigung. Vor der Bestätigung eintreffende Daten werden noch bestätigt, jedoch nicht an die Appl. weitergegeben. Ein beenden kann einseitig oder beidseitig erfolgen.

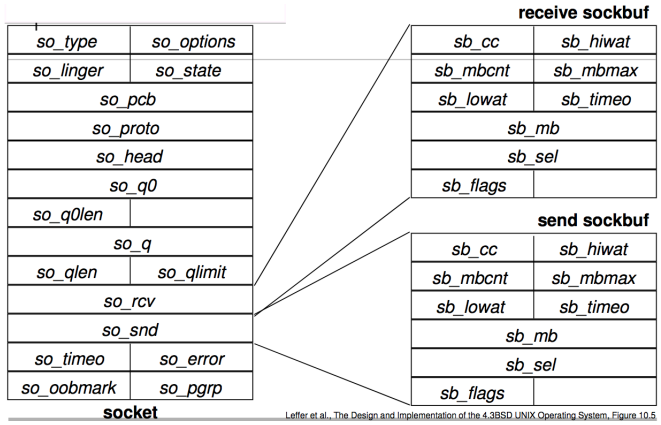
close()
Deallociert die lokale Socket Datenstruktur und die zugerdneten Ressourcen (Puffer)

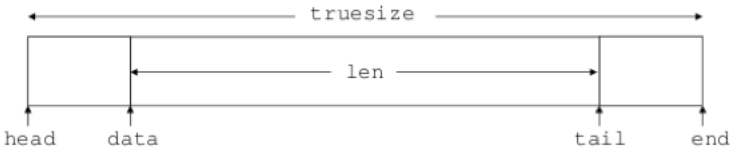
exit()
Terminiert das PRogramm ohne explizites shutdown/close. Es überlässt die Aufräumarbeiten dem OS und der Gegenpartei.

Provider Abort
überlässt das Afuräumen dem OS beider Gegenparteien

1.6 Speicherverwaltung auf der Socket Schicht

Daten auf einem socket werden in sogenannten mbufs gespeichert, welche jeweils auf den folgenden mbuf verweisen (wie eine LinkedList). Alle mbufs zusammen sind der mbuf Pool.

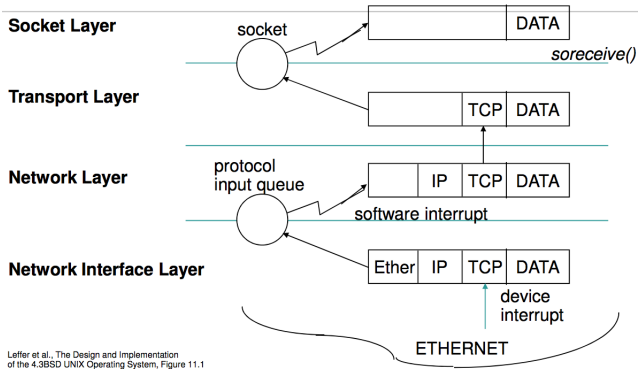





```
struct sk_buff {
      struct sk_buf *next;       /* Next buffer is list */
      struct sock   *sk;            /* Socket we are owned by */
      unsigned long len;            /* Length of actual data */
      unsigned int  truesize;       /* Buffer size */
      unsigned char * head;         /* Head of buffer */
      unsigned char *data;       /* Data head pointer */
      unsigned char *tail;       /* Tail pointer */
      unsigned char *end;           /* End pointer */
}
```

J. O'Gorman, Operating System with Linux, page 316/317

1.7 Datenfluss duch die Schichten



1.8 System Calls

socket(domain, type, prot)
Socket erstellen

bind(sockfd, addr, addlen)
Nach dem Erstellen eines Sockets existiert dieser blos als Name aber er hat noch keine Adresse. Bind verknüpft ihn mit einer Adresse.

listen(sockfd, backlog)
Markiert den Socket als passiven Socket. Das bedeutet als Socket der einkommende Verbindungen mit accept entgegennehmen kann.

accept(socketfd, addr, addlen, flags)
Wird mit Verbindungsbassierten Socket Typen (STREAM, SEQPACKET) verwendet. Entpackt die erste Anfrage au der Queue der zu berarbeitenden Verbindungen für den hörenden Socket. Kreiert einen neuen verbundenen Socket und gibt einen neuen File Descriptor zurück, welcher auf diesen verweist. Der ursprüngliche Socket wird nicht verändert. Der neue ist nicht im listening state.

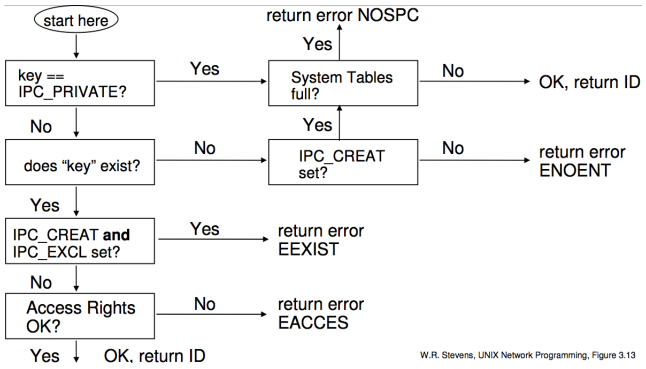
connect(socketfd,addr,addrlen)
Verbindet den Socket zur Adresse.

2 Interprocess Communication (IPC)

2.1 Zentrale Konzepte

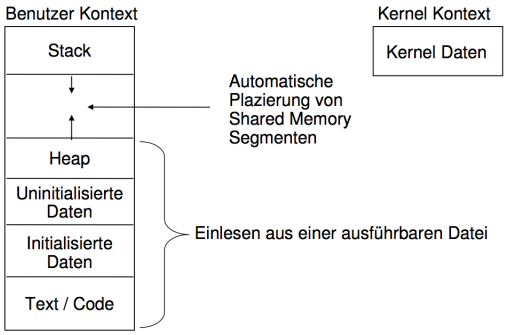
Es gibt im System drei verschiedene IPC MEchanismen, für deren Verwaltung der Kernel je eine separate Tabelle pflegt: **Message Queues**, **Shared Memory** und **Semaphore**.
Für die Verwaltung aller drei Mechanismen werden **dieselben Prozeduren** verwendet. Der Zugriff erfolgt über **einen numerischen Schlüssel**. Es gibt keine Registratur für verwendete/reservierte Schlüssel. **Kollisionen sind also möglic**h. Die IPC Objekte sind **nicht kompatibel mit dem Standard I/O** bassierter Prozesskommunikation wie Dateinen, Sockets, Pipes oder Geräte. Sie sind alle **limitiert auf ein lokales System**.

2.2 Verwaltung der Objekte



2.3 Shared Memory

Shared Memory erlaubt die gemeinsame Nutzung von Hauptspeicher-Seiten zwischen verschidenen (nicht) verwandten Prozessen. Es wird ein dedizierter Segment Typ verwendet, der auf der normalen Speicherverwaltung basiert. Die Synchronisation muss über Semaphore, Signalbits oder allgemeine Signale erfolgen.



2.3.1 Erstellen eines Segmentes (shmget)

- Entfernen einer Region von der Liste freier Regionen
- Einen Typ zuweisen
- Einen inode Pointer zuweisen
- Den reference count der inode um eins erhöhen
- region der Liste von aktiven Regionen hinzufügen
- gesperrete Region zurückgeben.

Listing 1: Erstellen eines Shared Memory Segments

```
1 int size, permflags, shm_id; key_t key;
  shm_id = shmget (key, size, permflags);
```

2.3.2 Einbinen eines Segmentes (shmat)

- check validity of descriptor, permissions
- if (user specified virtual address)
  - round off virtual address, as specified by flags;
  - check legality of virtual address, size of region;
- else - kernel picks virtual address: error if none available;
- attach region to process address space (algorithm attachreg);
- if (region being attached for the first time)
  - allocate page tables, memory for region (algorithm growreg);
- return (virtual address where attached);

Listing 2: Einbindung eines Shared Memory Segments

```
int shm_id, shmflags; char *memptr, *daddr, *shmat();
memptr = shmat (shm_id, daddr, shmflags);
```

2.3.3 Entfernen eines Segmentes (shmdt)

- get auxiliary memory management tables for process,
- release as appropriate;
- decrement process size;
- decrement region reference count;
- if (region count is 0 and region not sticky bit)
  - free region (algorithm freereg);
- else */* either reference count non-0 or region sticky bit on */*
  - free inode lock, if applicable (inode associated with region);
  - free region lock;

Listing 3: Entfernen eines Shared Memory Segments

```
int       retval; char      *memptr;
retval = shmdt (memptr);
```

2.3.4 System Calls

shmget(key, size, shmflag)
Gibt ID des Shared Memory Segments entsprechenden

shmat(shmid, shmaddr, shmflg)
Attach: Fügt das Segment zum Addressspace des Prozesses hinzu.

shmdt(shmaddr)
Detach: Entfernt das Segemnt vom Addressspace des Prozesses

shmctl(shmid,cmd,*buf)
Fürht cmd auf dem Segment aus

2.4 Message Queues

Message Queues dienen zum Versenden von Datenmengen von einem Server an einen Client. Sie ist eine verkettete Liste wobei die Liste vom Listenkopf aus kontrolliert wird. Mit msgsend() werden Elemente an die Schlange angefügt und mit msgrcv() gelesen und wieder entfernt. Es können beliebige strukturierte Daten ausgetauscht werden zwischen verdandten oder nicht verwandten Prozessen.

2.4.1 Erzeugen einer Message Queue

Durch den Aufruf von msget() besorgt sich ein Prozess den Zugriff auf eine message queue id. Mit msgctl() kann diese gelöscht, gelesen, modifiziert, gesperrt oder entsperrt werden.

Listing 4: Erzeugen einer Message Queue

```
int msg_qid, permflags; key_t key;
msg_qid = msgget (key, permflags);
```

2.4.2 Senden einer Nachricht

- check legality of descriptor, permissions;
- while (not enough space to store message)
  - if (flags specify not to wait) return;
  - sleep (until event enough space is available);
- get message header;
- read message text from user space to kernel;
- adjust data structures:
  - enqueue message header,
  - message header points to data,
  - counts,
  - time stamps,
  - process ID;
- wakeup all processes waiting to read message from queue;

Listing 5: Versenden einer Nachricht

```
int msg_qid, size, flags, retval;
struct my_msg {
  long mtype;
  char mtext[SOMEVALUE];
} message;
retval = msgsnd (msg_qid, &message, size, flags);
```

2.4.3 Empfang einer Nachricht

- check permissions;
- **loop;**
- check legality of message descriptor;
- */* find message to return to user */*

- **if (requested message type == 0)**
  - consider first message on queue;
- **else if (requested message type > 0)**
  - consider first message on queue with given type;
- else **/* requested message type < 0 */**
  - consider first of the lowest typed messages on queue, such that its type is <= absolute value of requested type;
- if (there is a message)
  - adjust message size or return error if user size too small;
  - copy message type, text from kernel space to user space;
  - unlink message from queue;
  - return;
- */* no message */*
- if (flags specify not to sleep) return with error;
- sleep (event message arrives on queue);
- **goto loop;**

Listing 6: Empfangen einer Nachricht

```
int msg_qid, size, flags, retval; long msg_type;
struct my_msg {
  long mtype;
  char mtext[SOMEVALUE];
} message;

retval = msgrcv (msg_qid, &message, size, msg_type, flags);
```

2.5 Semaphore

Ein Semaphor ist zwar kein Mechanismus für die Datenübertragung, erkann jedoch hilfreich sein für die Synchronisierung der Prozesse.Vereinfacht gesprochen ist ein Semaphor ein Zähler, dessen Wert entscheidet, ob eine Ressource eklusiv nutzbar ist oder nicht. Er besteht aus einem Zähler und einer Warteschlange für die Aufnahme blockierter Prozesse. Er hat drei Funktionen:

initsem()
: Initialisierungsfunktion

P()
: Prüfen und dekrementieren des Semaphor. Falls er danach noch grösser als null ist, setzt der Prozess seine Arbeit fort, ansonsten wartet er.

V()
: Der Semaphor erhöhen, was eventuell wartende Prozesse wieder freigibt.

2.5.1 Benutzung für gegenseitigen Ausschluss

In einem kritischen Abschnitt verändert Prozess P1 eine Datenstruktur, die der Prozess gemeinsam mit einem Prozess P2 nutzt. Prozess P2 verändert die Datenstruktur in seinem kritischen Abschnitt. Ein Semaphor in Ausschlussfunktion wird eingesetzt, um zu erreichen, dass sich die Prozesse P1 und P2 niemals gleichzeitig in ihren kritischen Abschnitten befinden. Hierzu wird der Semaphor mit 1 initialisiert

```
s: semaphore (1);

P1 : process                          P2 : process
      ...                                   ...
      P(s);                                 P(s);
      ... -- critical section               ... - critical section
      V(s);                                 V(s);
      ...                                   ...
end  process                          end  process
```

2.5.2 Benutzung für Prozess-Synchronisation

Der Prozess P1 beginnt mit einem kritischen Abschnit gleich nach der Initialisierung. Der Prozess P2 kann vorerst nur bis zu der P-Funktion laufen, da der Semaphor mit 0 initialisiert wurde. Erst wenn P1 mit der V-Funktion den Zähler um eins incrementiert kann der Prozess P2 weiterlaufen.

```
s: semaphore (0);

P1 : process                          P2 : process
      ...                                   ...
      V(s); -- signal event                 P(s); -- wait for event
      ...                                   ...
end  process                          end  process
```

2.5.3 Additive Semaphore

```
exclusion : add_semaphore (N);

     type reader = process
     ...
     P (exclusion, 1);
          ... - read within critical section
     V (exclusion, 1);
     end process;

     type writer = process
     ...
     P (exclusion, N);
          ... -- write within critical section
     V (exclusion, N);
     end process;
```

2.5.4 Beispiel

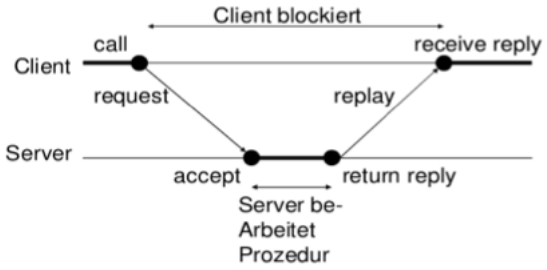
Listing 7: Semaphore Beispiel

```
P (semid) int semid; {
     struct sembuf p_buf;
     p_buf.sem_num = 0;
     p_buf.sem_op = -1;        /* negativer Wert, also Fall 1 = P() */
     p_buf.sem_flg = SEM_UNDO;
     if (semop (semid, &p_buf, 1) == -1) {
          perror (?p(semid) failed?);
          exit (1);
     } else return (0);
}
V (semid) int semid; {
     struct sembuf v_buf;
     v_buf.sem_num = 0;
     v_buf.sem_op = 1;         /* positiver Wert, also Fall 2 = V() */
     v_buf.sem_flg = SEM_UNDO;
     if (semop (semid, &v_buf, 1) == -1) {
          perror (?v(semid) failed?);
          exit (1);
     } else return (0);
}

main () {
     key_t semkey = 0x200;
     if (fork () == 0) handlesem (semkey);
     if (fork () == 0) handlesem (semkey);
     if (fork () == 0) handlesem (semkey);
}
handlesem (skey) key_t skey; {
     int semid, pid = getpid();
     if ((semid = initsem (skey)) > 0) exit (1);
     printf (?\nprocess %d before critical section\n?, pid);
     P (semid);
     printf (?process %d in critical section\n?, pid);
     sleep (2);
     printf (?process %d leaving critical section\n?, pid);
     V (semid);
     printf (?process %d exiting\n?, pid);
     exit (0);
}
```

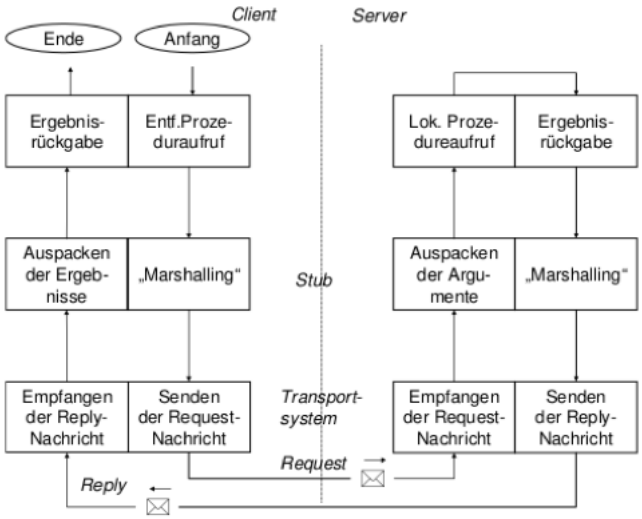
3 Remote Procedure Calls

RPC ist eine Technik zur Realisierung von Interprozesskommunikation zwischen verschidenen Rechnern. Es gibt verschiedene Implementierungen welche meist nicht untereinander kompatiebel sind.



3.1 Funktionsweise

Der Client ruft eine entfernte Prozedur auf einem Server auf. Die zur Bearbeitung benötigten Parameter werden an die Client-Stub Funktionsstelle des RPC Systems geschickt. Die Stub verpackt alle Funktionsparameter in eine komplexe Datenstruktur – dieser Vorgang wird auch "Marshalling"genannt.
Danach beauftragt die Stub das System mit der Übertragung der Nachricht an den Server. Der Client wartet nun auf eine Antwort vom Server und ist blockiert. Bekommt die Client-Stub eine Nachricht vom Server zurück, wird diese dekodiert (ünmarshalling") und an die Applikation zurückgegeben.
Wenn der Server eine Nachricht erhält, wird diese vom Betriebssystem an die Server-Stub weitergeleitet. Dort werden die Parameter entpackt (ünmarshalling") und die entsprechende Prozedur aufgerufen. Das zurücksenden der Ergebnisse ist äquivalent.



3.2 Server / Client Bindung

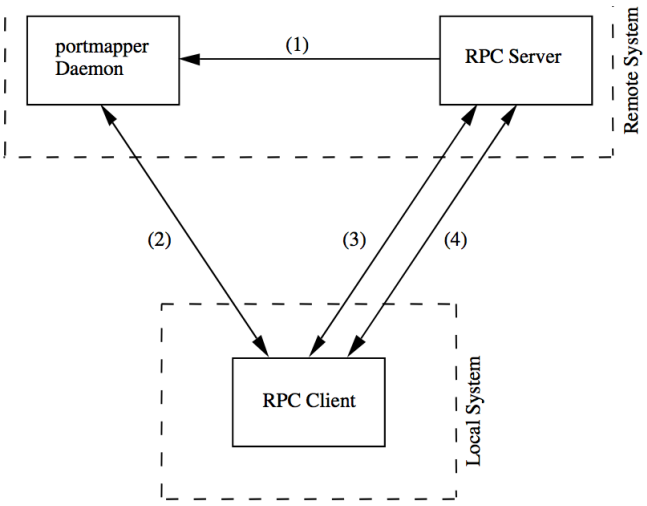
- Finden eines Servers im Netz
- Finden des gewünschten Service auf dem Server im Netz
- Sun RPC benutzt die Standard-Unix-Methode für das Finden von Servern im Internet (DNS).
- Alle Serverprogramme, Programmversionen und angebotenen Prozeduren werden mit eindeutigen Nummern gekennzeichnet.
- Ein Prozess kann eine oder mehrere Prozeduren anbieten.

- Der protmapper Prozess (Linux: rpcbind) auf Porst 111 auf jedem Serversystem dient als zentrale, lokale Registratur für verfügbare RPC-Dienste.

3.3 RPC Portmapping Ablauf

1. Der Server erstellt einen Socket und registriert Programmnummer, Versionsnummer und Portnummer beim Portmapper.
2. Ein Client kontaktiert den Portmapper und fragt nach einer Programm-, Versions- und Prozedurnummer. Falls lokal bekannt, sendet der Portmapper die Portnummer zurück.

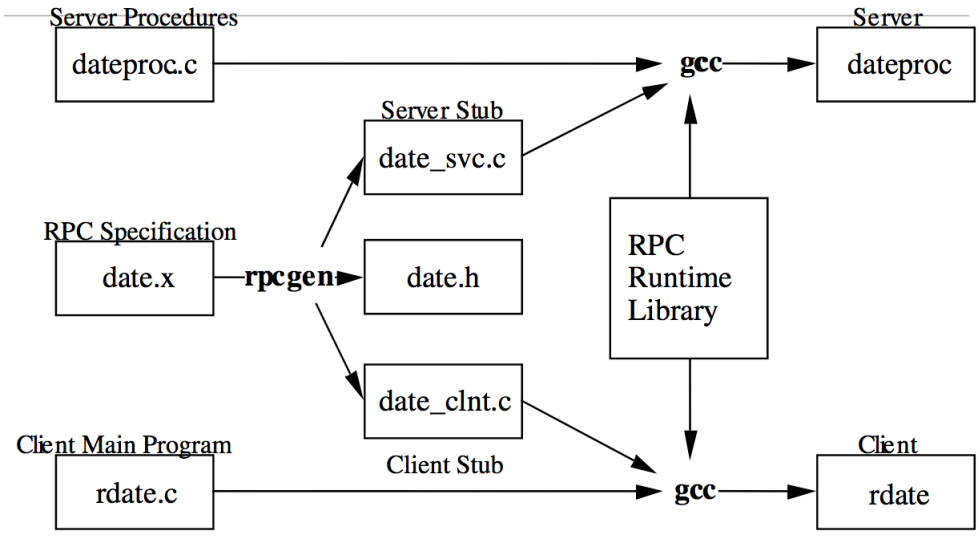
3./4. Der Client kann nun die gewünschte Prozedur direkt beim Server aufrufen.



3.4 Auswahl des Transportprotokolls

- RPC ist unabhängig von spezifischen Transportdiensten oder Protokollen. (Sun RPC unterstützt TCP und UDP)
- Es werden Abbildungen auf die üblichen Transportprotokolle angeboten.

3.5 Programmierbeispiel



3.6 Ausnahmebehandlung

- Zusätzlich zu lokalen Fehlern können in RPC weitere Fehler auf dem Serversystem und bei der Datenübermittlung durch das Netz auftreten.
- Abbruch von bereits übermittelten oder gestarteten Prozeduraufrufen durch den Klienten beim Server.
- Terminieren des Klienten bevor der Server den Ablauf der entfernten Prozdedur beendet hat.
- Sun RPC verwendet das automatische Neusenden von Anfragen im Fall der Benutzung von UDP, und erkennt verlorene Verbindungen in TCP.
- Sun RPC unterstützt keinen separaten Kontrollkanal.

3.7 Aufrufsemantik

- Prozedur wird genau einmal ausgeführt
- Prozedur wird höchstens einmal ausgeführt
- Prozedur wird mindestens einmal ausgeführt
- Jeder Server unterhält einen Cache mit kürzlich erhaltenen Prozeduraufrufen und den zurückgesendeten Resultaten, und sendet das gespeicherte Resultat zurück, wenn ein Duplikat eines PRozeduraufrufs entdeckt wird.

3.8 Benutzung und Beschränkungen

- Die Server sind meist zustandslos:
  - alle Operationen sind unabhängig voneinander
  - Bobustheit gegen Fehler im Klienten, im Server und im Netz
- Performance (lokale vs. entfernte Prozeduren)
- Service-Strategien (ein Server, ein Server pro Klient, ...)
- Verteilungsstrategien (wo im Netz werden Server plaziert.