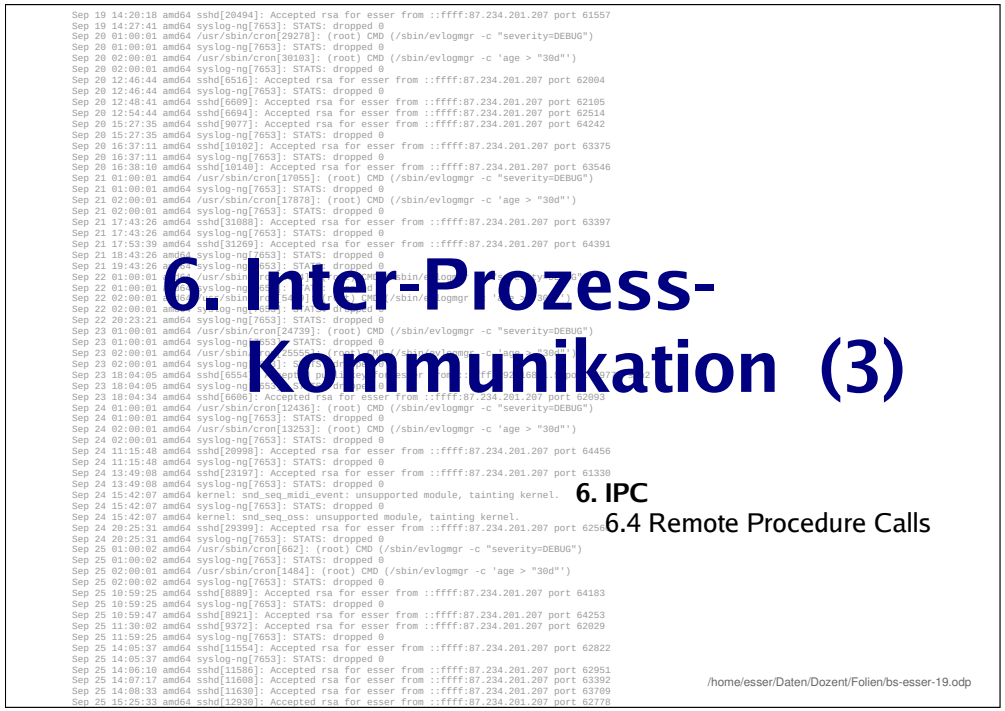
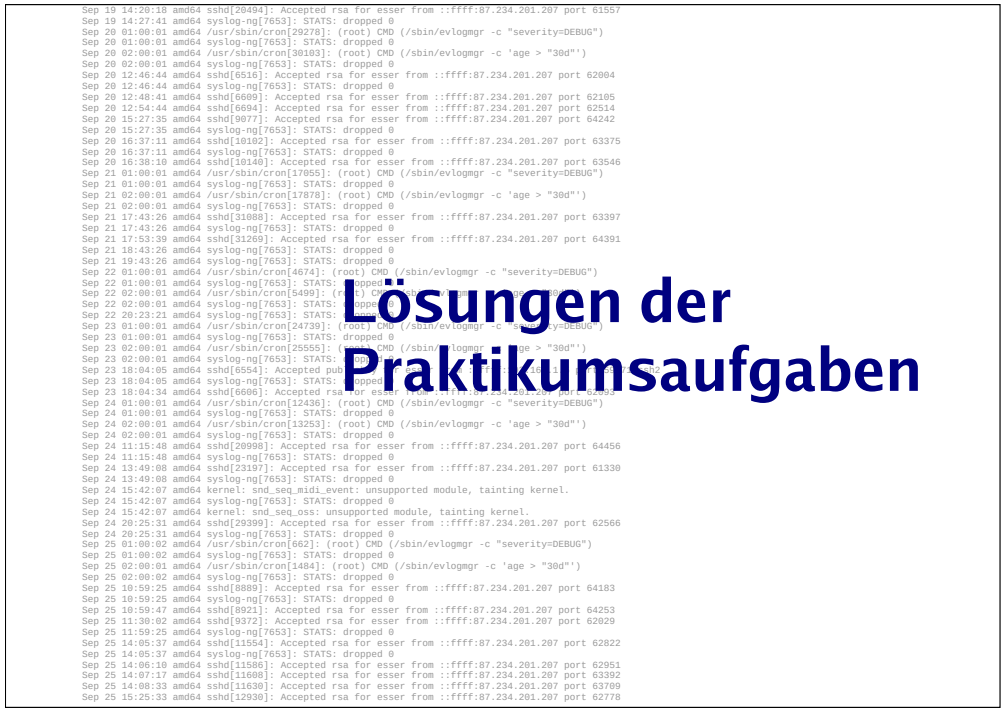


6. Inter-Prozess-Kommunikation (3)

6. IPC

6.4 Remote Procedure Calls



Lösungen der Praktikumsaufgaben

Praktikum: IPC mit TCP-Sockets (1)

5. Inter Process Communication (IPC) mit Sockets

a) Programmieren Sie (in Python oder einer Sprache Ihrer Wahl) zwei Programme, die die folgenden Aufgaben übernehmen: Ein Server (server.py) öffnet einen TCP-Port (z. B. 5555; SOCK_STREAM) und wartet auf Verbindungen. Ein Client (client.py) baut eine Verbindung zum Server auf und schickt ihm einen Dateinamen.

Der Server prüft, ob die Datei existiert – wenn ja, schickt er als Antwort mehrere Zeilen Text mit folgenden Informationen zurück:

- Größe der Datei (wenn es eine reguläre Datei ist)
- erste Zeile der Datei (wenn es eine reguläre Datei ist)
- Anzahl der Einträge im Verzeichnis (wenn es ein Verzeichnis ist)
- wenn es sich weder um eine Datei noch um ein Verzeichnis handelt (aber ein Eintrag unter dem angegebenen Namen existiert), schickt der Server die Meldung „E_NEITHERFILENORDIR“ zurück
- existiert gar kein Eintrag, schickt der Server die Meldung „E_NOSUCHFILE“ zurück.

Der Client wertet die Rückgabe aus.

- Falls eine Datei erkannt wurde, schickt er an den Server die Aufforderung „linetwo“; der Server reagiert darauf, indem er als Antwort die zweite Zeile der Datei schickt.
- Für den Fall, dass ein Verzeichnis erkannt wurde, schickt er an den Server die Aufforderung „listdir“; der Server reagiert darauf, indem er als Antwort eine Liste aller Dateien schickt.
- In den Fällen E_NEITHERFILENORDIR und E_NOSUCHFILE erfolgt keine weitere Kommunikation.

Der Client gibt schließlich die erhaltenen Daten aus. Beide Programme bauen die Verbindung ab.

Praktikum: IPC mit TCP-Sockets (2)

```
#!/usr/bin/python
# client.py
from socket import *      # socket, AF_INET, SOCK_STREAM
from sys import argv,exit
host = 'localhost'        # Hostname, hier alles lokal
port = 5555               # Gleicher Port wie beim Server

# Kommandozeilen-Argumente auswerten
if len (argv) != 2:
  print "Client erwartet genau ein Argument (einen Dateinamen)."
  exit (-1)
filename = argv[1]

print "Argument: ", filename

# Socket erzeugen, Verbindung aufbauen
s = socket (AF_INET, SOCK_STREAM)
s.connect ((host, port))
s.send(filename)
reply = s.recv(1024)
reply2 = False
if reply[0:5] == "Norma":   #  normale Datei
  s.send("linetwo")
  reply2 = s.recv(1024)
if reply[0:5] == "Verze":   #  Verzeichnis
  s.send("listdir")
  reply2 = s.recv(1024)

s.close ()
print reply,
if reply2: print reply2,
```

Praktikum: IPC mit TCP-Sockets (3)

```
# server.py
from socket import *      # socket, AF_INET, SOCK_STREAM
from stat import *        #
from os import stat
from dircache import listdir
host = ''                 # localhost
port = 5555               # Verbindungen auf Port 5555 annehmen
s = socket (AF_INET, SOCK_STREAM)
s.bind ( (host, port) )   # An Port binden
s.listen (1)              # nur eine Verbindung gleichzeitig

while 1:                  # Endlosschleife
  conn, addr = s.accept ()
  print 'Verbindung von', addr

  filename = conn.recv (128); print "Dateiname: >"+filename+"<"

  finished = 0
  try:
    filestatus = stat (filename)
  except:
    reply = "E_NOSUCHFILE"; finished = 1

  if not finished:
    st_mode = filestatus[ST_MODE]
    # Datei?
    if S_ISREG(st_mode):
      reply = "Normale Datei, Größe: "+str(filestatus.st_size)
      try:
        f = file (filename)
        firstline = f.readline (); secondline = f.readline ()
        f.close()
      except:
        firstline = "Konnte Datei nicht lesen\n"
        secondline = "Konnte Datei nicht lesen\n"
      reply = reply + "\n" + firstline
    # Verzeichnis?
    elif S_ISDIR (st_mode):
      reply = "Verzeichnis"

    # weder noch
    else:
      reply = "E_NEITHERFILENORDIR"
      finished = 1  # keine weitere
                    # Kommunikation

  conn.send (reply)

  if not finished:
    secondcmd = conn.recv(128)
    if secondcmd == "linetwo"
    and S_ISREG (st_mode):
      conn.send (secondline)
    if secondcmd == "listdir"
    and S_ISDIR (st_mode):
      # Verzeichnis auslesen
      try:
        dircontent = listdir (filename)
        reply = repr (dircontent)
      except:
        reply = "Kann Verzeichnis nicht lesen"
      conn.send (reply)

  conn.close ()
# Ende der aeusseren While-Schleife
```


Praktikum: IPC mit UDP-Sockets (1)

b) Passen Sie Ihr Programm aus a) so an, dass es mit UDP (statt TCP) arbeitet. Vorsicht: In dem Fall gibt es keine Verbindung, und Sie müssen mit SOCK_DGRAM arbeiten.

6. IPC mit Pipes

Ändern Sie die Programme aus Aufgabe 5 so ab, dass sie zwei benannte Pipes zur Kommunikation verwenden,

- die erste Pipe (*/tmp/pipe-request*) ist für die Übertragung der Anfrage vom Client zum Server gedacht,
- die zweite Pipe (*/tmp/pipe-reply*) für die Übertragung der Antwort in umgekehrter Richtung.

Dabei soll der Server die beiden Pipes erzeugen. (Er startet auch als erstes der beiden Programme.)

Praktikum: IPC mit UDP-Sockets (2)

```
#!/usr/bin/python

# client.py (UDP-Version)

from socket import *      # socket, AF_INET, SOCK_DGRAM
from sys import argv,exit

host = 'localhost'        # Hostname, hier alles lokal
port = 5555               # Gleicher Port wie beim Server

# Kommandozeilen-Argumente auswerten
if len (argv) != 2:
  print "Client erwartet genau ein Argument (einen Dateinamen)."
  exit (-1)
filename = argv[1]

print "Argument: ", filename

# Socket erzeugen, Verbindung aufbauen
s = socket (AF_INET, SOCK_DGRAM)

addr = (host,port)

s.sendto(filename, addr)
reply, zzz = s.recvfrom(1024)
reply2 = False

if reply[0:5] == "Norma":   #  normale Datei
  s.sendto("linetwo", addr)
  reply2, zzz = s.recvfrom(1024)

if reply[0:5] == "Verze":   #  Verzeichnis
  s.sendto("listdir", addr)
  reply2, zzz = s.recvfrom(1024)

s.close ()
print reply,
if reply2: print reply2,
```

```
#!/usr/bin/python

# client.py

from socket import *      # socket, AF_INET, SOCK_STREAM
from sys import argv,exit

host = 'localhost'        # Hostname, hier alles lokal
port = 5555               # Gleicher Port wie beim Server

# Kommandozeilen-Argumente auswerten
if len (argv) != 2:
  print "Client erwartet genau ein Argument (einen Dateinamen)."
  exit (-1)
filename = argv[1]

print "Argument: ", filename

# Socket erzeugen, Verbindung aufbauen
s = socket (AF_INET, SOCK_STREAM)
s.connect ((host, port))

s.send(filename)
reply = s.recv(1024)

reply2 = False

if reply[0:5] == "Norma":   #  normale Datei
  s.send("linetwo")
  reply2 = s.recv(1024)

if reply[0:5] == "Verze":   #  Verzeichnis
  s.send("listdir")
  reply2 = s.recv(1024)

s.close ()
print reply,
if reply2: print reply2,
```


Praktikum: IPC mit UDP-Sockets (3)

```
#!/usr/bin/python

# server.py (UDP-Version)

from socket import *      # socket, AF_INET, SOCK_DGRAM
from stat import *        #
from os import stat
from dircache import listdir

host = ''                 # localhost
port = 5555               # Anfragen auf Port 5555 annehmen
s = socket (AF_INET, SOCK_DGRAM)
s.bind ( (host, port) )   # An Port binden

while 1:                  # Endlosschleife
  filename, addr = s.recvfrom (128)
  print 'Anfrage von', addr

....

  s.sendto (reply, addr)

  if not finished:
    secondcmd, addr = s.recvfrom(128)
    if secondcmd == "linetwo" and S_ISREG (st_mode):
      reply = secondline
    if secondcmd == "listdir" and S_ISDIR (st_mode):
      # Verzeichnis auslesen
      try:
        dircontent = listdir (filename)
        reply = repr (dircontent)
      except:
        reply = "Kann Verzeichnis nicht lesen"

    s.sendto (reply, addr)
```

```
#!/usr/bin/python

# server.py

from socket import *      # socket, AF_INET, SOCK_STREAM
from stat import *        #
from os import stat
from dircache import listdir

host = ''                 # localhost
port = 5555               # Verbindunge auf Port 5555 annehmen
s = socket (AF_INET, SOCK_STREAM)
s.bind ( (host, port) )   # An Port binden
s.listen (1)              # nur eine Verbindung gleichzeitig

while 1:                  # Endlosschleife
  conn, addr = s.accept ()
  print 'Verbindung von', addr

  filename = conn.recv (128)

  conn.send (reply)

  if not finished:
    secondcmd = conn.recv(128)
    if secondcmd == "linetwo" and S_ISREG (st_mode):
      conn.send (secondline)
    if secondcmd == "listdir" and S_ISDIR (st_mode):
      # Verzeichnis auslesen
      try:
        dircontent = listdir (filename)
        reply = repr (dircontent)
      except:
        reply = "Kann Verzeichnis nicht lesen"
      conn.send (reply)

  conn.close ()
```

Praktikum: IPC mit FIFOs (1)

```
#!/usr/bin/python
# server.py (mit Pipes)
...
from os import stat, mkfifo, remove
...
p1name = "/tmp/pipe-request"; p2name = "/tmp/pipe-reply"
try: mkfifo (p1name)    # FIFOs einmalig erzeugen
except: True
try: mkfifo (p2name)
except: True
infile = file (p1name, "r+"); outfile = file (p2name, "w+")

while 1:                            # Endlosschleife
  filename = infile.readline() [:-1]
  print "Dateiname: >"+filename+"<"
  ...
  reply = ...

  outfile.write (reply+"\nEOF\n"); outfile.flush()

  if not finished:
    secondcmd = infile.readline () [:-1]
    if secondcmd == "linetwo" and S_ISREG (st_mode):
      outfile.write (secondline+"\nEOF\n"); outfile.flush()
    if secondcmd == "listdir" and S_ISDIR (st_mode):
      ...
      reply = ...
      outfile.write (reply+"\nEOF\n"); outfile.flush()

outfile.close (); infile.close ()
remove (p1name); remove (p2name)
```


Praktikum: IPC mit FIFOs (2)

```
#!/usr/bin/python
# client.py (mit Pipes)

p1name = "/tmp/pipe-request"; p2name = "/tmp/pipe-reply"
infile = file (p2name, "r+"); outfile = file (p1name, "w+")

def read_rest():      # Rest der Server-Meldung (bis ^EOF\n) lesen
  z = ""
  while z != "EOF\n": z = infile.readline()

[...]
# Anfrage senden
outfile.write(filename+"\n"); outfile.flush()
reply = infile.readline() [:-1]
read_rest ()
reply2 = False

if reply[0:5] == "Norma":   #  normale Datei
  outfile.write("linetwo\n"); outfile.flush()
  reply2 = infile.readline() [:-1]
  read_rest ()

if reply[0:5] == "Verze":   #  Verzeichnis
  outfile.write("listdir\n"); outfile.flush()
  reply2 = infile.readline() [:-1]
  read_rest ()

outfile.close (); infile.close ()
print reply,
if reply2: print reply2,
```


Praktikum: Test and Set (1)

7. Synchronisation: Test and Set

In der Vorlesung haben Sie das folgende (fehlerhafte) Programm *test.py* gesehen:

```
# test.py
from threading import Thread
class testthread(Thread):
   def __init__ (self):
      Thread.__init__(self)
   def run(self):
      global globalcount
      for j in range(0,99999):
         globalcount += 100
         globalcount -= 100
globalcount=0; threads = []
for i in range(0,100): # 100 Threads
  t = testthread(); threads.append(t)
  t.start()
for t in threads: t.join()
print "Ergebnis:",globalcount
```

Ebenfalls aus der Vorlesung kennen Sie das Konzept der `test_and_set`-Funktionen, welche atomar eine Variable ändern und ihren vorherigen Wert zurückgeben. In der Datei *test_and_set.py* finden Sie obiges Programm um eine `test_and_set`-Funktion ergänzt, die innerhalb der *testthread*-Klasse definiert ist:

```
def test_and_set(self,x):
  # x muss 0 oder 1 sein
  global tslvar
  tsl_lock.acquire()
  oldvalue = tslvar
  tslvar = x
  tsl_lock.release()
  return oldvalue
```


Praktikum: Test and Set (2)

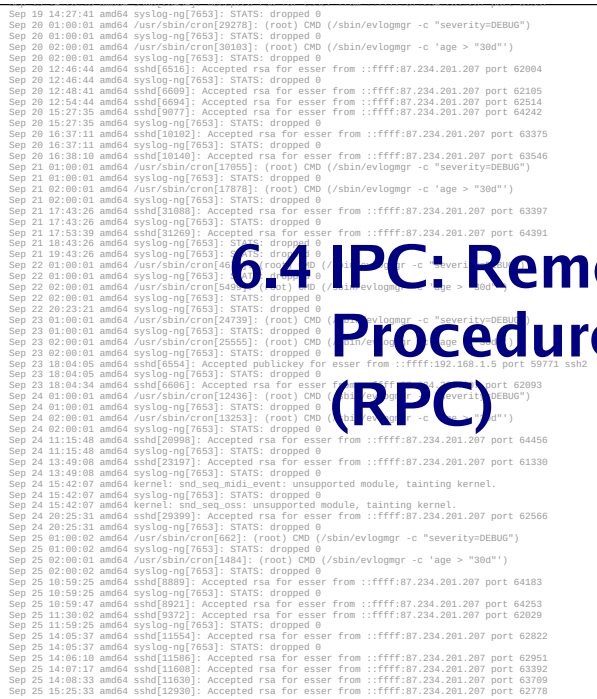
a) Beheben Sie das Problem im Programm mit einem Aufruf von *test_and_set*(). Hinweis: Da die Funktion in der Klasse definiert ist, müssen Sie sie mit *self.test_and_set(self,x)* aufrufen.

b) Vergleichen Sie die Laufzeit mit den Lösungen aus der Vorlesung (die mit Locks oder Semaphoren arbeiten; Sie finden diese Dateien im Aufgabenpaket als *lock.py* und *sem.py*. Warum ist die Variante mit *test_and_set* so langsam?

Praktikum: Test and Set (3)

```
# test_and_set.py

from threading import Thread, Lock
from time import sleep

class testthread(Thread):
  def __init__ (self):
    Thread.__init__(self)
  def test_and_set(self,x):
    # x muss 0 oder 1 sein
    global tslvar
    tsl_lock.acquire()
    oldvalue = tslvar
    tslvar = x
    tsl_lock.release()
    return oldvalue
  def run(self):
    global globalcount, tslvar
    for j in range(0,99999):
      t = 1
      while t:
        t = self.test_and_set (1)
      globalcount += 100
      globalcount -= 100
      tslvar = 0
```

```
globalcount=0       # globalen Counter
                    # initialisieren
tslvar=0            # globale Variable
                    # fuer test_and_set
tsl_lock = Lock()   # Lock für tslvar
threads = []
for i in range(0,100):
  t = testthread()
  threads.append(t)
  t.start()

for t in threads: t.join()

print "Ergebnis:",globalcount
```

6.4 IPC: Remote Procedure Calls (RPC)

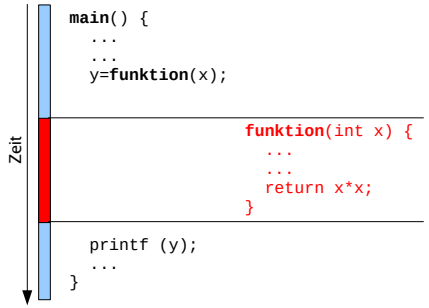
Remote Procedure Calls (1)

- Idee: Funktionsaufrufe, die Prozess- und sogar Rechner-übergreifend sind
- Programmierer muss sich über die nötige Kommunikation keine Gedanken machen
  - Message Passing, I/O etc. werden vor dem Programmierer verborgen
- Funktionsargumente und -ergebnisse mit einem RPC-Protokoll zwischen den beteiligten Prozessen hin und her übertragen
- Client- / Server-Prinzip

Remote Procedure Calls (2)

Normaler Funktionsaufruf:

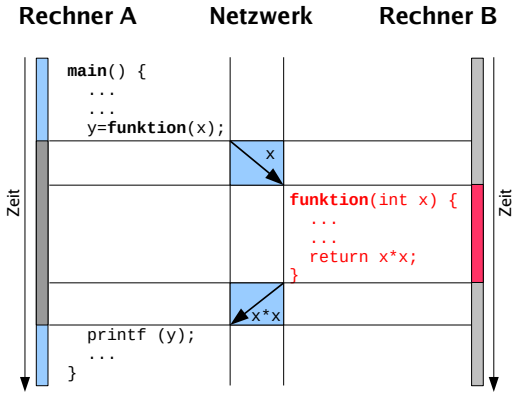
- Programm führt *main*()-Funktion aus.
- Beim Aufruf einer Funktion werden Parameter auf den Stack gelegt, und der Prozessor springt via **jmp** oder **call** zur Startadresse der Funktion.
- Während die Funktion rechnet, ist die *main*()-Funktion unterbrochen (synchroner Auftrag).
- Nach Abarbeitung springt der Prozessor in die *main*()-Funktion zurück, die dann das Ergebnis verwertet.



Remote Procedure Calls (3)

Remote-Funktionsaufruf (anschaulich):

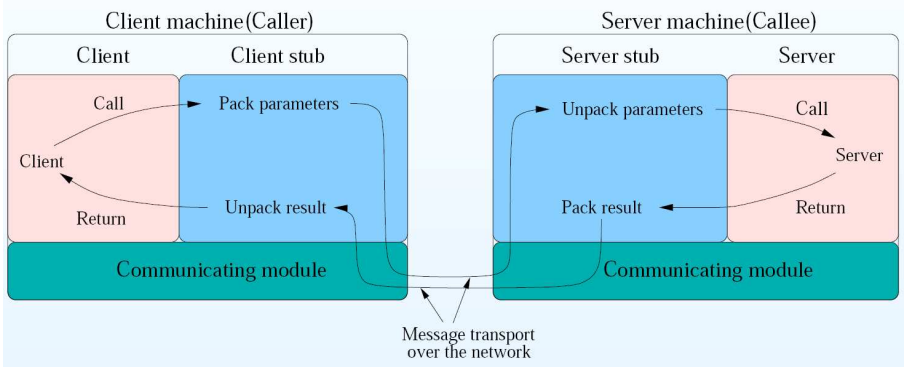
- Programm führt *main*()-Funktion aus.
- Vor dem Start der Funktion wird das Argument (über das Netzwerk) an den Prozess übermittelt, der die Funktion ausführt.
- Dort wird das Ergebnis berechnet und zurück übertragen.
- Während der Berechnung und den beiden Übertragungen ist der aufrufende Prozess untätig (synchroner Auftrag).



Remote Procedure Calls (4)

- Prozess, der Funktion aufruft, enthält **client stub** für diese Funktion. Aufgabe:
  - Funktionsaufruf (über das Netzwerk) an anderen Prozess weiterleiten
  - Antwort empfangen und Ergebnis zurückgeben
- Prozess, der Funktion ausführt, enthält **server stub** für diese Funktion. Aufgabe:
  - Argumente von einem client stub entgegennehmen
  - Funktion ausführen
  - Ergebnis zurück senden

Remote Procedure Calls (5)



Quelle: http://www-wjp.cs.uni-sb.de/lehre/seminar/ss04/reports/A_Shadrin-RPC-folien.pdf

Remote Procedure Calls (6)

Unterschiede zu normalen Funktionen:

- kein **call by reference** möglich
  – aufrufender Code und aufgerufene Funktion haben keinen gemeinsamen Adressraum
- client stub kann aber calls by reference annehmen und dann die Daten für den Transport vorbereiten
- keine **Nebeneffekte** möglich (z. B.: Funktion verändert globale Variable)

Remote Procedure Calls (7)

Marshalling / Serialisierung (1)

- Problem: Client- und Server-Rechner verwenden evtl. unterschiedliche interne Repräsentationen für Datentypen
  - klassisches (und einfachstes) Beispiel: Byte order – big-endian / little-endian 0xCC99 = 0xCC 0x99 oder 0x99 0xCC
  - wie packt man Strings, Floats, komlexere Strukturen (abstrakte Datentypen), Objekte etc. ein?

Remote Procedure Calls (8)

Marshalling / Serialisierung (2)

- System stellt **pack()** und **unpack()** bereit
- Es ist sicher gestellt, dass die Reihenfolge
  - Client: s = **pack** (objekt);
send (srv, s);
  - Server: recv (cl, s);
objekt = **unpack** (s);

  dazu führt, dass „objekt“ auf Client und Server das gleiche Objekt repräsentiert

Remote Procedure Calls (9)

Marshalling / Serialisierung (3)

- Beispiel für Marshalling: Python

```
import pickle
data1 = {'a': [1, 2.0, 3, 4+6j],
         'b': ('string', u'Unicode string'),
         'c': None}
selfref_list = [1, 2, 3]
selfref_list.append(selfref_list)

output = open('data.pkl', 'wb')

# Pickle dictionary using protocol 0.
pickle.dump(data1, output)

# Pickle the list using the highest protocol
# available.
pickle.dump(selfref_list, output, -1)

output.close()
```

```
import pickle

pkl_file = open('data.pkl', 'rb')

data1 = pickle.load(pkl_file)
print data1

data2 = pickle.load(pkl_file)
print data2

pkl_file.close()
```


Remote Procedure Calls (10)

Marshalling / Serialisierung (4)

```
> hexdump -C data.pkl
00000000  28 64 70 30 0a 53 27 61  27 0a 70 31 0a 28 6c 70  |(dp0.S'a'.p1.(lp|
00000010  32 0a 49 31 0a 61 46 32  2e 30 0a 61 49 33 0a 61  |2.I1.aF2.0.aI3.a|
00000020  63 5f 5f 62 75 69 6c 74  69 6e 5f 5f 0a 63 6f 6d  |c__builtin__.com|
00000030  70 6c 65 78 0a 70 33 0a  28 46 34 2e 30 0a 46 36  |plex.p3.(F4.0.F6|
00000040  2e 30 0a 74 70 34 0a 52  70 35 0a 61 73 53 27 63  |.0.tp4.Rp5.asS'c|
00000050  27 0a 70 36 0a 4e 73 53  27 62 27 0a 70 37 0a 28  |'.p6.NsS'b'.p7.(|
00000060  53 27 73 74 72 69 6e 67  27 0a 70 38 0a 56 55 6e  |S'string'.p8.VUn|
00000070  69 63 6f 64 65 20 73 74  72 69 6e 67 0a 70 39 0a  |icode string.p9.|
00000080  74 70 31 30 0a 73 2e 80  02 5d 71 00 28 4b 01 4b  |tp10.s...]q.(K.K|
00000090  02 4b 03 68 00 65 2e                              |.K.h.e.|
00000097
```

RPC-Beispiel (1)

- Ein Server-Programm soll Durchschnitts-berechnungen ausführen
- Ein Client ruft die Server-Prozedur transparent über RPC auf
- Aufgaben:
  - Interface-Definition erstellen und mit **rpcgen** übersetzen (erzeugt u. a. Client- und Server-Stubs)
  - Client und Server programmieren

RPC-Beispiel (2)

rpcgen erzeugt alle nötigen Dateien:

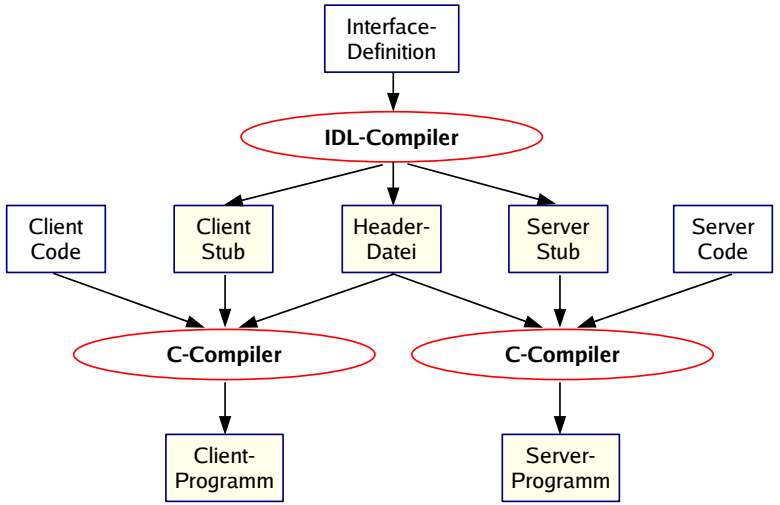
$ **rpcgen avg.x**

- *avg_clnt.c* enthält Client-Code (den Client-Stub)
- *avg_svc.c* enthält Server-Code (den Server-Stub)
- *avg_xdr.c* enthält Typbeschreibungen
- *avg.h* ist die Header-Datei

```
/* avg.x
 *
 * The average procedure receives an array
 * of real numbers and returns the average
 * of their values. This toy service handles
 * a maximum of 200 numbers.
 */
const MAXAVGSIZE  = 200;

struct input_data {
  double input_data<200>;
};

typedef struct input_data input_data;

program AVERAGEPROG {
    version AVERAGEVERS {
        double AVERAGE(input_data) = 1;
    } = 1;
} = 22855;
```

Quelle für dieses Beispiel (Code):
http://www.linuxjournal.com/article/2204

RPC-Beispiel (3)



RPC-Beispiel (4)

von **rpcgen** erzeugte Header-Datei (*avg.h*):

```
/* Please do not edit this file. It was generated using rpcgen. */
#include <rpc/rpc.h>
#define MAXAVGSIZE 200

struct input_data {
        struct {
                u_int input_data_len;
                double *input_data_val;
        } input_data;
};
typedef struct input_data input_data;

#define AVERAGEPROG 22855
#define AVERAGEVERS 1

extern double * average_1(input_data *, CLIENT *);
extern double * average_1_svc(input_data *, struct svc_req *);
extern int averageprog_1_freeresult (SVCXPRT *, xdrproc_t, caddr_t);

/* the xdr functions */

extern bool_t xdr_input_data (XDR *, input_data*);
extern bool_t xdr_input_data (XDR *, input_data*);
```

RPC-Beispiel (5)

von **rpcgen** erzeugter Client-Stub (*avg_clnt.c*):

```
/* Please do not edit this file. It was generated using rpcgen. */

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

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

double * average_1(input_data *argp, CLIENT *clnt) {
        static double clnt_res;

        memset((char *)&clnt_res, 0, sizeof(clnt_res));
        if (clnt_call (clnt, AVERAGE,
                (xdrproc_t) xdr_input_data, (caddr_t) argp,
                (xdrproc_t) xdr_double, (caddr_t) &clnt_res,
                TIMEOUT) != RPC_SUCCESS) {
                return (NULL);
        }
        return (&clnt_res);
}
```


RPC-Beispiel (6)

von **rpcgen** erzeugter Server-Stub (*avg_svc.c*)

```
/*
 * Please do not edit
 * this file. It was gene-
 * rated using rpcgen.
 */

#include "avg.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 averageprog_1(struct svc_req *rqstp, register SVCXPRT *transp) {
   union {
      input_data average_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 AVERAGE:
         _xdr_argument = (xdrproc_t) xdr_input_data;
         _xdr_result = (xdrproc_t) xdr_double;
         local = (char *(*)(char *, struct svc_req *)) average_1_svc;
         break;
      default:
         svcerr_noproc (transp);
         return;
   }
   memset ((char *)&argument, 0, sizeof (argument));
   if (!svc_getargs (transp, (xdrproc_t) _xdr_argument, (caddr_t) &argument)) {
      svcerr_decode (transp);
      return;
   }
   result = (*local)((char *)&argument, rqstp);
   if (result != NULL && !svc_sendreply(transp, (xdrproc_t) _xdr_result, result)) {
      svcerr_systemerr (transp);
   }
   if (!svc_freeargs (transp, (xdrproc_t) _xdr_argument, (caddr_t) &argument)) {
      fprintf (stderr, "%s", "unable to free arguments"); exit (1);
   }
   return;
}
```


RPC-Beispiel (7)

von **rpcgen** erzeugter Server-Stub (*avg_svc.c*) (Fortsetzung)

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

   pmap_unset (AVERAGEPROG, AVERAGEVERS);

   transp = svcudp_create(RPC_ANYSOCK);
   if (transp == NULL) {
      fprintf (stderr, "%s", "cannot create udp service.");
      exit(1);
   }
   if (!svc_register(transp, AVERAGEPROG, AVERAGEVERS, averageprog_1, IPPROTO_UDP)) {
      fprintf (stderr, "%s", "unable to register (AVERAGEPROG, AVERAGEVERS, udp).");
      exit(1);
   }

   transp = svctcp_create(RPC_ANYSOCK, 0, 0);
   if (transp == NULL) {
      fprintf (stderr, "%s", "cannot create tcp service.");
      exit(1);
   }
   if (!svc_register(transp, AVERAGEPROG, AVERAGEVERS, averageprog_1, IPPROTO_TCP)) {
      fprintf (stderr, "%s", "unable to register (AVERAGEPROG, AVERAGEVERS, tcp).");
      exit(1);
   }

   svc_run ();
   fprintf (stderr, "%s", "svc_run returned");
   exit (1);
   /* NOTREACHED */
}
```


RPC-Beispiel (8): Client-Code

```
#include "avg.h"
#include <stdlib.h>

void averageprog_1( char* host, int argc, char *argv[]) {
   CLIENT *clnt;
   double *result_1, *dp, f;
   char *endptr;
   int i;
   input_data average_1_arg;
   average_1_arg.input_data.input_data_val =
     (double*) malloc(MAXAVGSIZE*sizeof(double));
   dp = average_1_arg.input_data.input_data_val;
   average_1_arg.input_data.input_data_len = argc - 2;
   for (i=1;i<=(argc - 2);i++) {
      f = strtod(argv[i+1],&endptr);
      printf("value   = %e\n",f);
      *dp = f;
      dp++;
   }
   clnt = clnt_create(host, AVERAGEPROG,
                      AVERAGEVERS, "udp");
   if (clnt == NULL) {
      clnt_pcreateerror(host);
      exit(1);
   }
   result_1 = average_1(&average_1_arg, clnt);
   if (result_1 == NULL) {
      clnt_perror(clnt, "call failed:");
   }
   clnt_destroy( clnt );
   printf("average = %e\n",*result_1);
}
```

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

   if(argc < 3) {
     printf(
      "usage: %s server_host value ...\n",
      argv[0]);
      exit(1);
   }
   if(argc > MAXAVGSIZE + 2) {
      printf("Two many input values\n");
      exit(2);
   }
   host = argv[1];
   averageprog_1( host, argc, argv);
}
```

RPC-Beispiel (9): Server-Code

```c
#include <rpc/rpc.h>
#include "avg.h"
#include <stdio.h>

static double sum_avg;

double * average_1(input_data *input,
        CLIENT *client) {

   double *dp = input->input_data.input_data_val;
   u_int i;
   sum_avg = 0;
   for(i=1;i<=input->input_data.input_data_len;i++) {
      sum_avg = sum_avg + *dp; dp++;
   }
   sum_avg = sum_avg / input->input_data.input_data_len;
   return(&sum_avg);
}

double * average_1_svc(input_data *input,
        struct svc_req *svc) {
  CLIENT *client;
  return(average_1(input,client));
}
```


RPC-Beispiel (10)

- Kompilieren:

```
$ rpcgen avg.x
$ gcc avgclient.c avg_clnt.c \
  avg_xdr.c -o avgclient -lnsl
$ gcc avgserver.c avg_svc.c \
  avg_xdr.c -lnsl
```

- Server starten und RPC-Eintrag prüfen:

```
$ ./avgserver &
$ /usr/sbin/rpcinfo -p localhost
   Program Vers Proto   Port
    100000    2   tcp    111  portmapper
    100000    2   udp    111  portmapper
    391002    2   tcp    892  sgi_fam
     22855    1   udp   1025
     22855    1   tcp   1025
```

- Client starten:

```
$ ./avgclient localhost 49.32 12.8 1.0
value   = 4.932000e+01
value   = 1.280000e+01
value   = 1.000000e+00
average = 2.104000e+01
```


NFS (1)

- NFS: Network File System
- Sun hat RPC („Sun RPC“) für NFS entwickelt
- NFS-Prozeduren:
  - mkdir, rmdir, readdir
  - create, remove, read, write
  - getattr, setattr
  - link, symlink, readlink
  - statfs

(aus */usr/include/linux/nfs2.h*)

```c
/*
 * Procedure numbers for NFSv2
 */
#define NFSPROC_NULL            0
#define NFSPROC_GETATTR         1
#define NFSPROC_SETATTR         2
#define NFSPROC_ROOT            3
#define NFSPROC_LOOKUP          4
#define NFSPROC_READLINK        5
#define NFSPROC_READ            6
#define NFSPROC_WRITECACHE      7
#define NFSPROC_WRITE           8
#define NFSPROC_CREATE          9
#define NFSPROC_REMOVE          10
#define NFSPROC_RENAME          11
#define NFSPROC_LINK            12
#define NFSPROC_SYMLINK         13
#define NFSPROC_MKDIR           14
#define NFSPROC_RMDIR           15
#define NFSPROC_READDIR         16
#define NFSPROC_STATFS          17
```


NFS (2)

- Beobachten, wie Nachrichten übertragen werden:

```
tcpdump -s 1024 host 192.168.1.2 | grep nfs > /tmp/tcpdump.log
```

- Dann von NFS-Client aus Verzeichnis erzeugen, Datei anlegen, umbenennen, löschen
- NFS-Client führt via RPC die NFS-Prozeduren mkdir, create, write, read, remove aus

NFS (3)

```
cd /mnt/server; mkdir tmpdir
```

```
13:01:42.624 sony.9583 > amd64.nfs: 120 getattr fh Unknown/7B0905003C4B04009B540000EB14000002000000
13:01:42.627 amd64.nfs > sony.9583: reply ok 116 getattr DIR 40700 ids 500/100 sz 36144
13:01:42.627 sony.6799 > amd64.nfs: 124 access fh Unknown/7B0905003C4B04009B540000EB14000002000000 001f
13:01:42.630 amd64.nfs > sony.6799: reply ok 124 access c 001f
13:01:42.630 sony.4015 > amd64.nfs: 132 lookup fh Unknown/7B0905003C4B04009B540000EB14000002000000 "tmpdir"
13:01:42.632 amd64.nfs > sony.4015: reply ok 120 lookup ERROR: No such file or directory
13:01:42.632 sony.1231 > amd64.nfs: 160 mkdir fh Unknown/7B0905003C4B04009B540000EB14000002000000 "tmpdir"
13:01:42.636 amd64.nfs > sony.1231: reply ok 256 mkdir fh Unknown/ABC70B007B090500C33F20007B0905003C4B0400
```

```
echo "Betriebssysteme" > /mnt/server/tmpdir/bs.txt
```

```
13:01:51.308 sony.2879 > amd64.nfs: 120 getattr fh Unknown/7B0905003C4B04009B540000EB14000002000000
13:01:51.313 amd64.nfs > sony.2879: reply ok 116 getattr DIR 40700 ids 500/100 sz 36168
13:01:51.313 sony0095 > amd64.nfs:  124 access fh Unknown/7B0905003C4B04009B540000EB14000002000000 001f
13:01:51.315 amd64.nfs > sony.0095: reply ok 124 access c 001f
13:01:51.315 sony.7311 > amd64.nfs: 132 lookup fh Unknown/7B0905003C4B04009B540000EB14000002000000 "tmpdir"
13:01:51.318 amd64.nfs > sony.7311: reply ok 248 lookup fh Unknown/ABC70B007B090500C33F20007B0905003C4B0400
13:01:51.318 sony.4527 > amd64.nfs: 124 access fh Unknown/ABC70B007B090500C33F20007B0905003C4B0400 001f
13:01:51.322 amd64.nfs > sony.4527: reply ok 124 access c 001f
13:01:51.322 sony.1743 > amd64.nfs: 132 lookup fh Unknown/ABC70B007B090500C33F20007B0905003C4B0400 "bs.txt"
13:01:51.327 amd64.nfs > sony.1743: reply ok 120 lookup ERROR: No such file or directory
13:01:51.327 sony8959 > amd64.nfs:  164 create fh Unknown/ABC70B007B090500C33F20007B0905003C4B0400 "bs.txt"
13:01:51.332 amd64.nfs > sony.8959: reply ok 280 create fh Unknown/C8C70B00ABC70B00C33F2000ABC70B007B090500
13:01:51.332 sony.6175 > amd64.nfs: 124 getattr fh Unknown/ABC70B007B090500C33F20007B0905003C4B0400
13:01:51.335 amd64.nfs > sony.6175: reply ok 188 getattr REG 2 ids 2/500 sz 2147483648000
13:01:51.336 sony.3391 > amd64.nfs: 156 write fh Unknown/C8C70B00ABC70B00C33F2000ABC70B007B090500 16 (16) bytes @0
13:01:51.349 amd64.nfs > sony.3391: reply ok 140 write [|nfs]
```


NFS (4)

```
cat /mnt/server/tmpdir/bs.txt
```

```
13:01:56.256 sony.30607 > amd64.nfs:  120 getattr fh Unknown/ABC70B007B090500C33F20007B0905003C4B0400
13:01:56.259 amd64.nfs > sony.30607:  reply ok 116 getattr DIR 40755 ids 500/500 sz 72
13:01:56.259 sony.7823 > amd64.nfs:   132 lookup fh Unknown/ABC70B007B090500C33F20007B0905003C4B0400 "bs.txt"
13:01:56.262 amd64.nfs > sony.7823:   reply ok 248 lookup fh Unknown/C8C70B00ABC70B00C33F2000ABC70B007B090500
13:01:56.262 sony.5039 > amd64.nfs:   124 access fh Unknown/C8C70B00ABC70B00C33F2000ABC70B007B090500 002d
13:01:56.264 amd64.nfs > sony.5039:   reply ok 124 access c 000d
```

```
mv /mnt/server/tmpdir/bs.txt /mnt/server/tmpdir/bs-alt.txt
```

```
13:02:03.920 sony.2255 > amd64.nfs:   136 lookup fh Unknown/ABC70B007B090500C33F20007B0905003C4B0400 "bs-alt.txt"
13:02:03.922 amd64.nfs > sony.2255:   reply ok 120 lookup ERROR: No such file or directory
13:02:03.922 sony.9471 > amd64.nfs:   120 getattr fh Unknown/C8C70B00ABC70B00C33F2000ABC70B007B090500
13:02:03.925 amd64.nfs > sony.9471:   reply ok 116 getattr REG 100644 ids 500/500 sz 16
13:02:03.925 sony.6687 > amd64.nfs:   188 rename fh Unknown/ABC70B007B090500C33F20007B0905003C4B0400 "bs.txt"
                                      -> fh Unknown/ABC70B007B090500C33F20007B0905003C4B0400 "bs-alt.txt"
13:02:03.929 amd64.nfs > sony.6687:   reply ok 264 rename
```

```
rm /mnt/server/tmpdir/bs-alt.txt
```

```
13:02:08.376 sony.3903 > amd64.nfs:   120 getattr fh Unknown/ABC70B007B090500C33F20007B0905003C4B0400
13:02:08.378 amd64.nfs > sony.3903:   reply ok 116 getattr DIR 40755 ids 500/500 sz 80
13:02:08.378 sony.1119 > amd64.nfs:   136 lookup fh Unknown/ABC70B007B090500C33F20007B0905003C4B0400 "bs-alt.txt"
13:02:08.381 amd64.nfs > sony.1119:   reply ok 248 lookup fh Unknown/C8C70B00ABC70B00C33F2000ABC70B007B090500
13:02:08.381 sony.8335 > amd64.nfs:   124 access fh Unknown/C8C70B00ABC70B00C33F2000ABC70B007B090500 002d
13:02:08.384 amd64.nfs > sony.8335:   reply ok 124 access c 000d
13:02:08.385 sony.5551 > amd64.nfs:   136 remove fh Unknown/ABC70B007B090500C33F20007B0905003C4B0400 "bs-alt.txt"
13:02:08.390 amd64.nfs > sony.5551:   reply ok 124 remove
```


LPC: Local Procedure Call

- Variante von RPC, bei der Client und Server auf demselben Rechner laufen
- Einige „Einsparungen“ möglich:
  - kein Marshalling nötig (Sender und Empfänger verwenden gleiche Datenrepräsentation)
  - *call by value* evtl. möglich, falls Sender- und Empfänger-Prozess gemeinsamen Speicher nutzen
  - Besondere RPC-Fehler (etwa: Funktionsergebnis „Server abgestürzt“) können nicht auftreten
- Microsoft LPC: Windows wechselt automatisch auf ein einfacheres Protokoll, wenn es eine lokale Verbindung erkennt

Vorschau

nächste Vorlesung am 10.01.

Letztes IPC-Thema:
Message Passing mit MPI
(High Performance Computing)

15./17.01.: Kapitel 7 – Deadlocks
22./24.01.: Zusammenfassung