Aufgabe 4: RPC - verwaiste Fernaufrufe, SUN-RPC

[1] Nelson, B.J.:
Remote procedure Call. Tech. Rep. CLS-81-9, Xerox Palo Alto Research Center, Palo Alto, CA, 1981

[2] Kaiserswerth, M.:
Der Fernaufruf als Betriebssystemdienst. Arbeitsbericht des IMMD, Band 22, Nummer 1 Januar 1989

[3] Tanenbaum, A.
Modern Operating Systems
Prentice Hall Int., 1992

[4] ONC+ Developer's Guide
http://sundocs.rrze.uni-erlangen.de:8888/

a) Was sind Orphans und wie entstehen sie?
b) Wie wird versucht, das Problem der Orphans zu lösen?
c) Welche Funktion leistet der 'Portmapper' bei RPCs?
d) Wie sieht die Implementation einer einfachen Funktion (z.B. die Addition zweier Integer) anhand eines SUN-RPC's aus? Welche Rolle spielt dabei das Programm "rpcgen"?

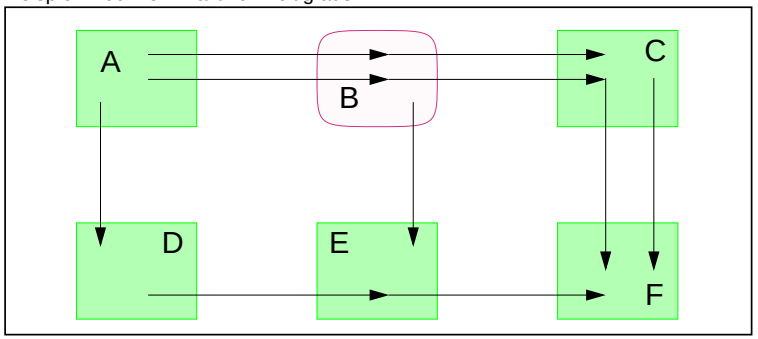
Lösungsvorschlag Aufgabe 4: RPC - verwaiste Fernaufrufe

a) Was sind Orphans und wie entstehen sie?

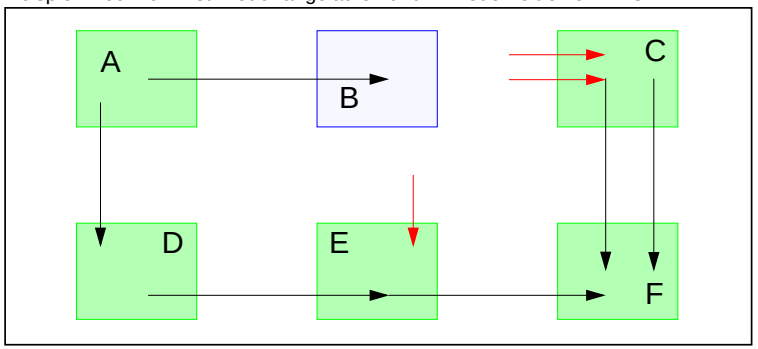
Orphans oder verwaiste Aufrufe sind unerwünschte Ausführungen von Aufrufen und können durch das Ablaufen des letzten Timeouts, durch den Ausfall eines beteiligten Rechners (Clients) oder durch den Ausfall der Kommunikationsverbindung entstehen. Orphans sind die Teile von RPC-Aufrufen auf Servern, die keinen Kontakt mehr zu ihren Clients haben.

Um auch beim RPC die last-one Semantik des lokalen Prozeduraufrufs aufrechtzuerhalten, muß vor dem Neustart oder dem Wiederaufsetzen eines verteilten Programms sichergestellt werden, daß alle Aktivitäten vorhergehender und nun verwaister RPCs abgebrochen wurden [2].

Beispiel: Rechner B fällt kurzzeitig aus.



Beispiel: Rechner B ist wieder angelaufen und A wiederholt einen RPC.



b) Wie wird versucht, das Problem der Orphans zu lösen?

Orphans stellen ein Problem dar, weil sie sinnlos Rechenzeit verbrauchen, wichtige Betriebsmittel belegt haben können und vor allem *Probleme bei der Aufrufsemantik hervorrufen können*. Aus diesem Grund müssen Orphans behandelt werden, um eine last-of-many-Semantik für den transitiven Fall (auch als *last-one-Semantik* bezeichnet) zu erzielen, da Orphans später als der Wiederholungsaufruf ausgeführt werden können, aber das Ergebnis des Wiederholungsaufrufs akzeptiert wurde.
Die wesentlichen Mechanismen, um Orphans zu behandeln, sind:

Extermination

Für eine gezielte Elimination des verwaisten Restes ist es notwendig, sich in einem stabilen Speicher zu merken, auf welchen Rechnern die Unteraufrufe laufen (Ausgangsliste). Nach dem Wiederanlauf des Rechners werden alle in der Liste aufgeführten Rechner benachrichtigt, damit diese die dort verwaisten RPCs abbrechen [1][2].

Reincarnation

Die Zeit wird in Epochen eingeteilt. Der Neustart eines Clients stellt den Beginn einer Epoche für alle Rechner im Netz dar, die durch Nachrichtenaustausch übermittelt wird und zum Abbruch aller Fernaufrufe führt.
Eine Abschwächung dieses drastischen Vorgehens ist die sogenannte **gentle reincarnation**: ein Auftrag wird nach Erkennung einer neuen Epoche nur dann abgebrochen, wenn dessen Auftraggeber nicht mehr existiert. Dadurch werden RPC's, die durch den Clientausfall nicht involviert waren, weitergeführt.

Expiration

Jeder Aufruf hat eine maximale Laufzeit (Timeout). Wenn diese abgelaufen ist beendet er sich selbst. Es sind große Laufzeiten notwendig, um auch bei ungünstiger Rechnerbelastung eine erfolgreiche Ausführung des RPC's zu ermöglichen. Üblich sind Zeitspannen um 30 Sekunden. Voraussetzung sind dabei synchron laufende Systemuhren.
Der Vorteil ist, daß keine Kommunikation zwischen den Rechnern notwendig ist, da der Algorithmus lokal autonom abläuft. Der Nachteil ist, daß die Zeitschranke sehr groß sein muß und dann immer noch nicht sicher ist, daß korrekte RPC-Aufrufe nicht wegen dieser Schranke abgebrochen werden [2]. Ein ausgefallener Rechner muß mind. die Zeitspanne des timeouts bis zum Wiederanlauf warten.

Deadlining With Postponement

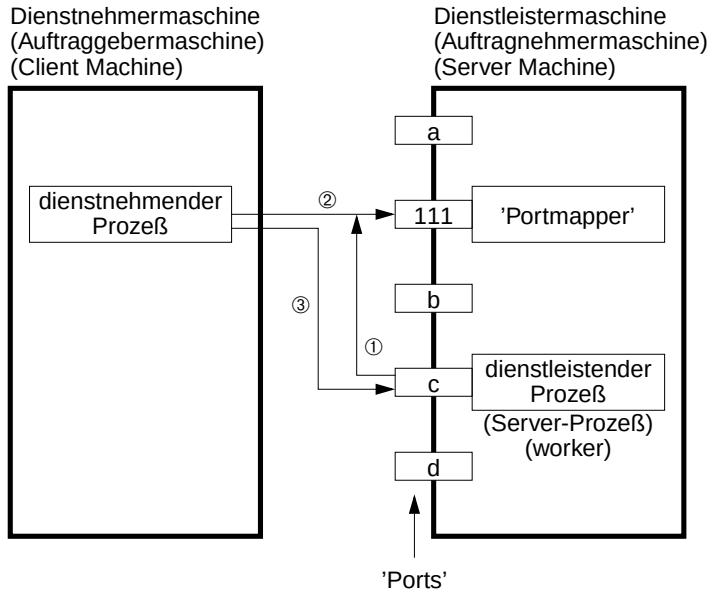
Dem RPC wird eine Zielzeit mitgegeben, bis zu der er spätestens fertig sein muß. Wird diese Zielzeit überschritten, fragt der Server beim Client an, ob dieser noch existiert und ob noch weitergemacht werden soll.

c) Welche Funktion leistet der 'Portmapper' bei RPCs?

Im Allgemeinen ist nicht bekannt, an welchem Port ein Server ansprechbar ist. Die Verbindung wird mit Hilfe eines Vermittlers/Maklers hergestellt, dem sogenannten Portmapper oder Binding Server.

Der Ablauf ist wie folgt:

1 Der Server meldet sich beim Portmapper an. Er teilt ihm mit, welchen Dienst er zur Verfügung stellt und unter welcher Portnummer er Nachrichten entgegennimmt.
2 Der Client wendet sich an den Portmapper, der eine feste Portnummer hat, und erhält die Portnummer des Servers. (Vergleiche dazu auch Telefon-Auskunft – "Hier werden Sie geholfen")
3 Mit dieser Portnummer kann sich der Client direkt an den Server wenden und den Fernaufruf ausführen lassen.

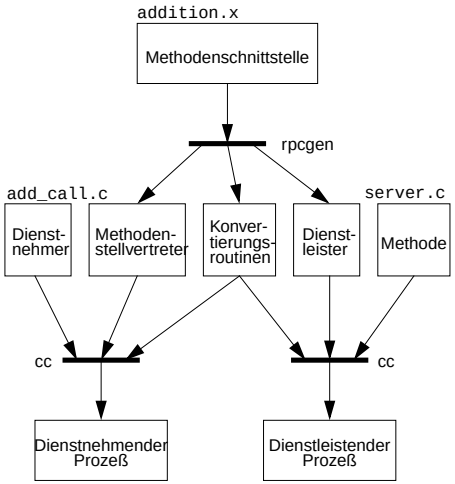


d) Implementation eines SUN-RPC

Überblick SUN-RPC:

- der SUN-RPC basiert auf dem Stub-Mechanismus.
- zur automatischen Generierung der Stubs kann das Utility "rpcgen" eingesetzt werden.
- pro RPC-Aufruf ist lediglich ein Parameter erlaubt. Benötigt der RPC mehrere Parameter, muß eine Hilfsstruktur definiert werden. (Diese Aufgabe kann der rpcgen übernehmen).
- Der Host, an dem der RPC aufgerufen wird, muß beim Aufruf bekannt sein. Übliche Abbildungen via DNS und NIS/NIS+ werden durchgeführt.
- Als Transportschicht kann UDP und TCP verwendet werden.
- Die Aufrufsemantik ist "at least once" im fehlerfreien Fall (Aufruf kommt ohne Fehlermeldung zurück). RPC's müssen demzufolge idempotente Funktionalität besitzen.
- Für das Marshalling wird der XDR-Standard verwendet. Dadurch wird eine gewisse Kompatibilität mit alternativen Implementationen gewährleistet.
- Ein RPC wird durch ein Tripel (Programm-Nummer, Versions-Nummer, Prozedur-Nummer) identifiziert. Die Programm-Nummer repräsentiert dabei eine Gruppe von verwandten RPC's.

Beispiel-Implementation SUN-RPC: Addition zweier Zahlen:
Das folgendes Schema verdeutlicht die Vorgehensweise:



Um den RPC zu implementieren, müssen drei Teilbereiche programmiert werden:

- Die Methodenschnittstelle
- die eigentliche Prozedur
- der Dienstnehmer, um den RPC auszuführen

Diese Codefragmente könnten wie folgt aussehen:

addition.x:

```
program ADDRPC {
        version ADDVERS {
                int ADD (int, int) = 1;
        } = 1;
} = 0x20000002;
```

server.c (die Prozedur):

```
int *add_1 (int a, int b)
{
  static int result;

  result = a + b;
  return (&result);
}
```

add_call.c (der Dienstnehmer):

```
int main(int argc, char *argv[])
{
  char   *machine;
  CLIENT *clnt;
  int    *result;
  int     zahl1, zahl2;

  if (argc != 4)
     {fprintf (stderr, "usage: %s remote-host Zahl1 Zahl2\n", argv[0]);
       exit(1);}
  machine = argv[1];

  clnt = clnt_create(machine, ADDRPC, ADDVERS, "udp");

  zahl1 = atoi(argv[2]);
  zahl2 = atoi(argv[3]);

  result = add_1(zahl1, zahl2, clnt);

  if (result == (int *)NULL)
  { clnt_perror (clnt, argv[1]);
    exit (1); }

  fprintf (stdout, "Ergebnis: %d\n", *result);
return 0;
}
```


Durch den rpcgen wird der Client-Stub, der Server-Stub und die Transformationsroutinen zum Marshalling erzeugt:

addition_clnt.c (der Client-Stub):

```
...
int *add_1(int arg1, int arg2, CLIENT *clnt)
{
        add_1_argument arg;
        static int clnt_res;

        //Ergebnis zunächst auf 0 setzen
        memset((char *)&clnt_res, 0, sizeof (clnt_res));

        //Vorbereitung der Parameterstruktur
        arg.arg1 = arg1;
        arg.arg2 = arg2;

        //Aufruf des RPC-Subsystems, gleichzeitiges Marshalling
        //durch die XDR-Filter.
        //das Ergebnis des Aufrufes wird in clnt_res abgelegt.
        if (clnt_call(clnt, ADD,
                (xdrproc_t) xdr_add_1_argument, (caddr_t) &arg,
                (xdrproc_t) xdr_int, (caddr_t) &clnt_res,
                TIMEOUT) != RPC_SUCCESS) {
                return (NULL);
        }
        return (&clnt_res);
}
```

addition_xdr.c (für das Marshalling):

```
//Marshalling: die Parameter-Struktur add_1_argument wird in
//ihre rudimentären Datentypen zerlegt.
//Für diese existieren bereits vorgefertigte Marshalling-Routinen,
//(im Beispiel xdr_int)
bool_t xdr_add_1_argument(XDR *xdrs, add_1_argument *objp)
{
       if (!xdr_int(xdrs, &objp->arg1))
                return (FALSE);
        if (!xdr_int(xdrs, &objp->arg2))
                return (FALSE);
        return (TRUE);
}
```


addition_svc.c: (der Server-Stub - stark vereinfachte Version)

```
main()
{
  int transpnum;

// Anmeldung des RPC's beim Portmapper. Gleichzeitig Definition
// der Transportform - hier UDP.
  transpnum = svc_create(_add_1, ADDRPC, ADDVERS, "udp");
  if (transpnum == 0) {
       fprintf (stderr, "could not create service\n");
       exit (1);
}
// Starten des Servers.
  svc_run();
  exit(0);
}

//Diese Funktion wird vom Server aufgerufen, wenn ein RPC-Request
//für das Programm ADDRPC ankommt. Bei mehreren Prozeduren pro
//Programm verteilt sie die Aufrufe an die jeweiligen Dienste.
//Im vorliegenden Beispiel wird einfach die Prozedur add_1 aufgerufen.

static void _add_1(struct svc_req *rqstp, SVCXPRT *transp)
{
  char *result;
  add_1_argument add_1_arg;

  switch (rqstp->rq_proc)
  {
       case NULLPROC:
              svc_sendreply (transp, xdr_void, NULL);
              return;
       case ADD:
              //Holen der Argumente für den verlangten RPC-Aufruf
              svc_getargs(transp,xdr_add_1_argument,
                             (caddr_t)&add_1_arg);
              // Ausführung des RPC's. Der letzte Parameter gibt
              // zusätzliche Informationen an und wurde in diesem
              // Beispiel vernachlässigt.
              result = add_1(add_1_arg.arg1, add_1_arg.arg2, rqstp);
              // Zurücksenden des Ergebnisses
              svc_sendreply(..., result, ...);
              break;
       default:
              svcerr_noproc (transp);
  }
  return;
}
```