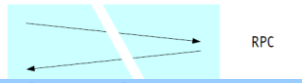
VSY - Remote Procedure Call

Kommunikation

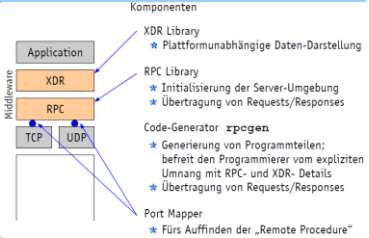
- Verschiedene Paradigmen werden genutzt
  a) Verbindungsorientierter Dienst eines (symmetrischen) Protokolls → Sockets oder TCP
  b) Message Passing → von UDP bis E-Mail
  c) Procedure Call → in allen "remote" Varianten



SUN-RPC - das Original

- Remote Procedure Call
- Ideen und Anfänge ca. 1980 im XEROX PARC
- Echt durchexerziert an der Stanford University (1986) für alles Mögliche (z.B. "distributed shared memory")
- Als RFC gefestigt erst 1995 von Sun Microsystems unter dem Begriff "Open Network Computing"
  - RPC RFC1831
  - XDR RFC1832
  - Binding Protocols RFC1833
- Von Sun Microsystems breit eingesetzt für NFS und NIS
- Entwickelte sich als Standard für alle Unix-Varianten

RPC im OSI-Modell

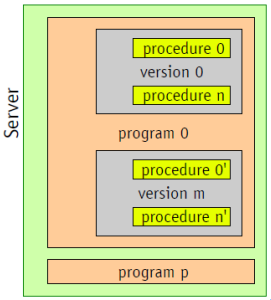


eXternal Data Representation (XDR)

- Einigung auf einen gemeinsamen Nenner
  - Zeichendarstellung (Codes)
  - Byte- oder Bit-Ordnung (little-endian etc.)
  - Ausrichtung auf Wortgrenzen (Alignment)
  - Darstellung von Fliesskomma-Werten
    - ASCII
    - big endian
    - 32-Bit-Architektur (Adressen, Werte)
    - IEEE Floating Point Standard
- Marshalling (marshalling yard = Rangierbahnhof) ???
  - Umwandlung von Strukturen in lineare, übertragbare Bitmuster ("Serialisierung")

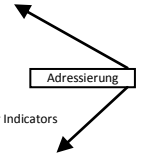
Adressierung von Remote Procedures

- Ein Programm kann mehrere Prozeduren anbieten
- Mehrere Programme können gleichzeitig ihre Prozeduren anbieten
- Implementierungen von Prozeduren sind Änderungen unterworfen
- → benötigen dreistufige **Adressierung**
- Service Identifier
  - program number
    - laufendes Programm (Prozess auf Server)
  - program version number
    - mehrere Versionen können gleichzeitig laufen
  - procedure number
    - Prozedur innerhalb des Prozesses



RPC Protokoll

- Request
  - Transaction Identifier (XID)
  - RPC Version Number
  - **Program Number, Program Version Number, Procedure Number**
  - Authentication Data
  - Argument Data
- Response
  - Transaction Identifier (XID)
  - Authentication Data
  - Outcome
    - "Accepted" Result Data, Error Indicators
    - "Rejected" Reason
- Transport
  - UDP und/oder TCP (IP Adress, Port Number)

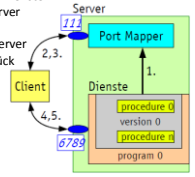


- XID: Zuordnung von Request und Response

Portmapper

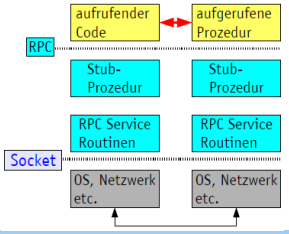
- Dient als Registrar für die Zuordnungen Programm ←→ UDP/TCP Port
  - Linux: **portmapper** Solaric: **rpcbind**
- Lässt sich abfragen über well-known port 111 (TCP/UDP) **rpcinfo**

1. Programm registriert seine Dienste
2. Client sucht Dienste auf Server
3. Portmapper antwortet
4. Client sendet Request an Server
5. Client erhält Response zurück



RPC für den "Programmierer"

- Vertrautes Paradigma (im Gegensatz zu Protokollen wie TCP)
- Stub-Prozeduren besorgen mit Hilfe der RPC-Service-Routinen
  - Adressierung
  - Verbindung
  - Datenformate
  - Überwachung (!)
- Stub-Prozeduren werden (automatisch) für den jeweiligen Anwendungsfall generiert (rpcgen)
- Eine RPC-Transaktion kann höchstens ein Argument übergeben (und natürlich wie üblich genau einen Rückgabewert zurücknehmen)



Prozeduraufruf mit einem Argument

- Der RPC verkraftet genau eine Argument-zu-Parameter-Übergabe

```
#include <stdio.h>
struct argumente {
   int a;
   int b;
};

int addieren_1 ( struct argumente add ) {
   int ergebnis;
   ergebnis = add.a + add.b;
   return ( ergebnis );
}

int main ( int argc, char *argv[] ) {
   struct argumente args;
   int c;
   args.a = 27;
   args.b = 2;
   c = addieren_1 ( args );
   printf ( "%d + %d = %d\n", args.a, args.b, c );
   return 0;
}
```

Datentyp des Arguments
Deklaration der Prozedur
Abfüllen der eigentlichen Argumente
Prozeduraufruf

Spezifikation für den RPC

- "Protokoll"
  - Deklaration der Signatur der aufzurufenden Prozedur
  - Angabe der Kennwerte für die Identifikation

rpc_demo.x — Syntax ist C-angelehnt (nicht C): RPCL

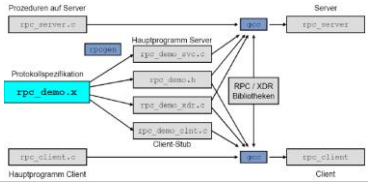
```
struct argumente {
   int a;
   int b;
};

program ADD_PROG {
   version ADD_VERS {
      int ADDIEREN ( struct argumente ) = 1; /* procedure ID */
   } = 1; /* program Version */
} = 0x31271966; /* program ID */
```

Und jetzt: `rpcgen -K rpc_demo.x`

...produziert diverse C-Quelltexte

Generierung einer RPC-Anwendung

- rpcgen produziert vier Dateien
- Daneben braucht es natürlich die Substanz der Server-Prozedur und den Client, welcher dereinst den RPC lancieren möchte



Generierung der diversen Komponenten

- Jetzt wirkt der C-Compiler

```
rpcgen -K rpc_demo.x

gcc -Wall -o rpc_client.o -c rpc_client.c
gcc -Wall -o rpc_server.o -c rpc_server.c

gcc -Wall -o rpc_demo_clnt.o -c rpc_demo_clnt.c
gcc -Wall -o rpc_demo_svc.o -c rpc_demo_svc.c
gcc -Wall -o rpc_demo_xdr.o -c rpc_demo_xdr.c

gcc -Wall -o rpc_client rpc_client.o rpc_demo_clnt.o rpc_demo_xdr.o
gcc -Wall -o rpc_server rpc_server.o rpc_demo_svc.o rpc_demo_xdr.o
```

Übersetzen von Client und Server
Übersetzen der generierten Stücke (Stubs und XDR)
Zusammenpappen von Client und Server

- sowas lässt man sich von der make-Utility besorgen

rpc_demo.h

```
struct argumente {
      int a;
      int b;
};

typedef struct argumente argumente;

#define ADD_PROG 0x31271966
#define ADD_VERS 1
#define ADDIEREN 1

extern int * addieren_1();
extern int add_proc_1_freeresult ();

/* the xdr functions */
extern bool_t xdr_argumente ();
```

Rückgabewert der Prozedur kommt als Referenz (Pointer)
(Argument(e) werden auch als Pointer gehandelt)
Deklaration der XDR-Umwandlung

Beispiel der Pointer-Magie

- Einfacher lokaler Prozeduraufruf mit Speicheradressen (Pointern)

```
#include <stdio.h>
struct argumente {
   int a;
   int b;
};

int *addieren_1 ( struct argumente *add ) {
   static int ergebnis;
   ergebnis = add->a + add->b;
   return ( &ergebnis );
}

int main ( int argc, char *argv[] ) {
   struct argumente args;
   int *c;
   args.a = 27;
   args.b = 2;
   c = addieren_1 ( &args );
   printf ( "%d + %d = %d\n", args.a, args.b, *c );
   return 0;
}
```

Ergebnis ist Pointer auf int
Argument ist Pointer auf struct
return-Wert ist Adresse wo der Wert liegt
Argument ist Pointer
Ergebnis ist Pointer

XDR-Wandlungsroutine rpc_demo_xdr.c

```
#include "rpc_demo.h"

bool_t xdr_argumente (xdrs, objp)
   XDR *xdrs;
   argumente *objp;

{
   register int32_t *buf;
   if (!xdr_int (xdrs, &objp->a))
      return FALSE;
   if (!xdr_int (xdrs, &objp->b))
      return FALSE;
   return TRUE;
}
```

Konvertierung von Datentyp `int` aus C in XDR-interne Darstellung

- XDR-Funktion: ausführlich beschrieben in man xdr_int und Zubehör

Client Stub rpc_demo_clnt.c

- eine lokale Funktion, die lediglich den RPC in Bewegung setzt

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

int * addieren_1(argp, clnt)
   struct argumente *argp;
   CLIENT *clnt;
{
   static int clnt_res;
   memset((char *)&clnt_res, 0, sizeof(clnt_res));

   if (clnt_call (clnt, ADDIEREN,
         (xdrproc_t) xdr_argumente, (caddr_t) argp,
         (xdrproc_t) xdr_int, (caddr_t) &clnt_res,
         TIMEOUT) != RPC_SUCCESS) {
      return (NULL);
   }
   return (&clnt_res);
}
```

Name der lokalen Funktion
Name der Remote-Prozedur
RPC-Funktion
XDR-Umwandlungen

siehe auch: `man clnt_call`

Client-Hauptprogramm

- rpc_client.c

```
#include <stdio.h>
#include <rpc/rpc.h>
#include "rpc_demo.h"

int main ( int argc, char *argv[] ) {
   CLIENT *cl;
   char *server;
   struct argumente args;
   int *c;

   server = argv[1];
   if (( cl = clnt_create(server,ADD_PROG,ADD_VERS,"udp"))==NULL){
      clnt_pcreateerror ( server );
      exit ( -1 );
   }

   args.a = 27;
   args.b = 2;
   c = addieren_1 ( &args, cl );
   printf ( "%d + %d = %d\n", args.a, args.b, *c );
   clnt_destroy ( cl );
   return 0;
}
```

Erzeugen des RPC Socket (Bindung an Server/Prozedur/Protokoll)
Effektiver Aufruf (nichts Spezielles)

Remote Procedure für den Server

- rpc_server.c

```
#include "rpc_demo.h"

int *addieren_1 ( struct argumente *add )
{
   static int ergebnis;
   ergebnis = add->a + add->b;
   return ( &ergebnis );
}
```

Generiertes Server-Hauptprogramm

rpc_demo_svc.c (Auszug Hauptteil)

```
int main (argc, argv)
   int argc;
   char **argv;

{
   ...
   transp = svcudp_create(RPC_ANYSOCK);
   ...
   if (!svc_register(transp,ADD_PROG,ADD_VERS,
         add_proc_1,IPPROTO_UDP)) {
   ....
   }
   transp = svctcp_create(RPC_ANYSOCK, 0, 0);
   ...
   if (!svc_register(transp,ADD_PROG,ADD_VERS,
         add_proc_1,IPPROTO_TCP)) {
   ....
   }

   svc_run ();
   exit (1);
}
```

Dienst registrieren
... für UDP
... für TCP
Start des eigentlichen Servers

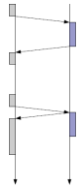
rpc_demo_svc.c (Auszug "registrierte Funktion")

```
static void add_proc_1(rqstp, transp)
   struct svc_req *rqstp;
   register SVCXPRT *transp;
{
   switch (rqstp->rq_proc) {
   case NULLPROC:
      (void) svc_sendreply(transp, (xdrproc_t)xdr_void,
                           (char*)NULL);
      return;
   case ADDIEREN:
      _xdr_argument = xdr_argumente;
      _xdr_result = xdr_int;
      local = (char *(*)()) addieren_1;
      break;
   default:
      svcerr_noproc (transp);
      return;
   }
   return;
}
```

enthält Attribute des Requests, z.B. den „Namen" der Prozedur
Aufruf der eigentlichen „Remote Procedure"

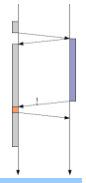
Synchroner und Asynchroner RPC

- Synchron
  - Client wartet auf die Ergebnisse (blocking)
  - Direkte Analogie zum lokalen Prozeduraufruf
- Asynchron
  - Server akzeptiert "Call" und bestätigt dies unmittelbar
  - Client fährt mit seinem Job weiter (non-blocking)
  - Ergebnis der Prozedur ist für den Client nicht von sofortiger Bedeutung
- Variante Einweg-RPC: Client wartet nicht einmal auf die Response "Akzeptiert"



Verzögerter Synchroner RPC

- Client fährt nach Call fort (non-blocking)
- Server macht "Interrupt" beim Client um die Resultate zu melden
- Eigentlich sind es zwei asynchrone RPCs mit vertauschten Rollen zwischen Client und Server
- Callback

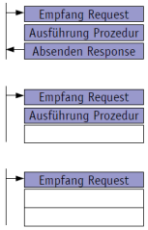


Semantik des RPC

- Auf welche Effekte kann man sich beim RPC verlassen?
- ideal
  - perfektes System, immer genau ein Request/Response-Paar
  - alles so, wie man es vom lokalen Procedure Call gewohnt ist
- jedoch
  - Fehler im Kommunikationskanal
    - UDP: Verlust eines Datagramms
    - TCP: verlorene Verbindung
  - Ausfall des Servers (im falschen Moment)
  - Ausfall des Clients
- Semantik im Fehlerfall
  - maybe
  - at most once
  - at least once
  - exactly once
- benötigt je nachdem zusätzliche Protokollelemente

Problem der Fehlererkennung

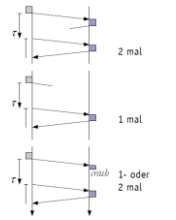
- Client kann **Fehler auf der Serverseite** nicht zuverlässig erkennen
- Kein Absturz, fehlerfreie Kommunikation
- Keine Response; Absturz nach der Ausführung
- Keine Response; Absturz vor der Ausführung
  => wie kann der Client zwischen diesen beiden Fällen unterscheiden?
- UDP: Verlust der Response
- TCP: Verlust der Verbindung



Fehlerbehebung beim RPC

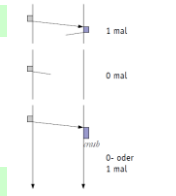
Definierte Ziel-Semantik: at least once

- Implementierung
  - Request wird so oft wiederholt, bis Response eintrifft
  - Benötigt (clientseitig) einen Timer für den Timeout-Mechanismus



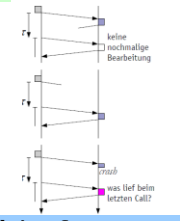
Definierte Ziel-Semantik: at most once

- Implementierung
  - Client sendet Request genau einmal
  - Responses werden ignoriert
  - Es braucht keinen Timer



Definierte Ziel-Semantik: exactly once

- Implementierung
  - Request wird so oft gesendet, bis Response eintrifft
  - Braucht Timer/Timeout
  - Server muss seine Tätigkeit protokollieren ("stateful server")



Reaktionen bei Ausfall eines Systems

- Problem
  - Nach Neustart von Client oder Server liegen allenfalls teilweise bearbeitete Aufträge vor
  - Aus Sicht der Anwendung ist die Situation zwischen Client und Server unklar (Zustand)
- mögliche Reaktion des **Clients**
  - at most once — keine Reaktion
  - at least once — Retransmit nach Timeout
  - exactly once — besondere Massnahmen notwendig
- Reaktionen des **Servers**
  - Nur bei "exactly once" besondere Massnahmen notwendig
  - Commit — Bestätigung bei vollständiger Erledigung
  - Rollback — Abbruch und Zurücknahme der Bearbeit.
  - Checkpoints — Sicherung von definierten Zuständen zur Wiederaufnahme der Bearbeitung

Reaktionen des Clients auf Serverabsturz

- Beispiel
  - Client sendet Text an Print-Server
  - Server sendet stets Empfangsbestätigung
  - Nach Absturz/Wiederaufstart meldet Server "bin wieder da"
  - Die Übertragung läuft fehlerfrei
- Mögliche Strategien für das Senden der Empfangsbestätigung
  - Server gibt Empfangsbestätigung vor Bearbeitung
  - Server gibt Empfangsbestätigung nach erfolgreicher Bearb.
- Mögliche Strategien des **Client** auf die Absturz-Meldung
  - keine Wiederholung einmal gesandter Aufträge
  - immer wiederholen
  - Wiederholung, wenn Empfangsbestätigung vorliegt
  - Wiederholung, wenn keine Empfangsbestätigung vorliegt

Reaktionen des Clients auf Serverabsturz

- Systematische Betrachtung

Beschreibung der Situation auf dem Server	M: „Bestätigung" P: „Auftrag (Printing) erledigt" C: „Crash"
Mögliche Sequenzen für die Strategie M;P	M;P;C M;C(;P) C(;M;P)
Mögliche Sequenzen für die Strategie P;M	P:M;C P;C(;M) C(;P;M)

- Kombinationen von Strategien

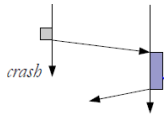
Client	Server					
Strategie für erneutes Senden	Strategie M;P			Strategie P;M		
	M;P;C	M;C(;P)	C(;M;P)	P;M;C	P;C(;M)	C(;P;M)
immer	DUP	OK	OK	DUP	DUP	OK
nie	OK	NULL	NULL	OK	OK	NULL
Nur nach M	DUP	OK	NULL	DUP	OK	NULL
Nur ohne M	OK	NULL	OK	OK	DUP	OK

- Ergebnis
  - für "at least once" und "at most once" existiert eine taugliche Strategie, für "exactly once" nicht
  - auch dann nicht, wenn dem Client die Strategie des Servers bekannt ist

Verhalten des Clients bei ausbleibender Response

- Client kann Request erneut senden.
  - Nur sinnvoll bei "idempotenten" Operationen
- Client muss aber auch bei nicht idempotenten Operationen Request nochmals senden
  - Erfordert Vorkehrungen, dass der Server Retransmits erkennt (XID)
  - Benötigt "stateful" Server
- Clients könnte Retransmit explizit als solchen kennzeichnen
- Server bearbeitet Requests ohne dieses Flag "normal"
- Server muss bei gesetztem Flag die bisherige Bearbeitung auf Teilresultate überprüfen, allenfalls von da aus weiterfahren

Verhalten des Clients nach eigenem Absturz



- Remote Operation verwaist, weil niemand mehr auf ihr Ergebnis wartet

- Orphans führen i.d.R. zu erheblichen Problemen

- Eventuell unerwartete Responses nach Neustart
- Unklarer Zustand zwischen neu aufgestartetem Client und Server

- saubere Beseitigung von Orphans ist nicht trivial!

- **Extermination**
  - Client protokolliert offene Requests
  - Versucht nach Neustart alle Orphans (per RPC!) auf dem Server zu löschen
  - Problem Protokollierungsaufwand
  - Problem bei verschachtelten Operationen
  - Problem des gezielten Rollback der Operationen eines Orphans

- **Reincarnation**
  - Sequentielle Zeitabschnitte sind numeriert (Epochen)
  - Neustart des Client beginnt in neuer Epoche; diese aktuell gültige Epoche wird allen Servern per Broadcast mitgeteilt
  - Server kann Operationen aus vergangenen Epochen löschen

- **Soft Reincarnation = Reincarnation plus…**
  - Server versucht zu Orphan den zugehörigen Client
  - Falls gefunden, kann Orphan in die neue Epoche übergeführt werden

- **Expiration**
  - Jeder Request hat einen Timeout-Wert τ
  - Server fordert Verlängerung an, sobald τ abgelaufen
  - Bleibt die "Bewilligung" vom Client aus, wird Operation gelöscht
  - Client wartet bei Neustart τ, (Ableben aller Orphans)
  - Problem: geeignete Bestimmung des Timeout-Wertes

Zusammenfassend

- Der RPC stammt aus der vor-objektorientierten Zeit
- Heutige Varianten (RMI, CORBA, SOAP, etc.) sind moderner, aber grundsätzlich mit denselben Eigenschaften behaftet

- Die Semantik des herkömmlichen Methodenaufrufes erhält im "Remote"-Fall mehrere neue Dimensionen:
  - das "Eigenleben" von mindestens drei beteiligten Systemen
  - keine gesicherte Information über den Gesamtzustand
  - die Unsicherheit über die von den anderen Systemen angewandten Kommunikations- und Überlebensstrategien und dessen Erfolg

- Dennoch erfreut sich der RPC einer gewissen Beliebtheit, denn es ist keine durchschlagend bessere Alternative in Sicht