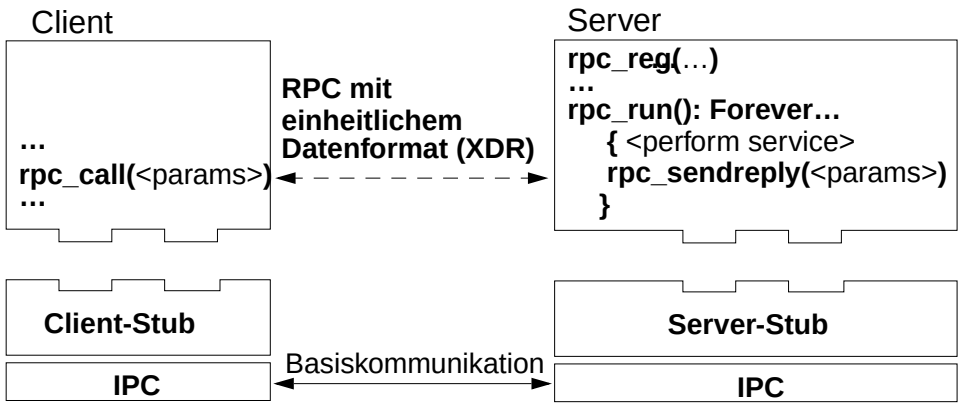
Sun-RPC

- RPC-Paket der Firma Sun, welches unabhängig von der Rechnerarchitektur vielfältig eingesetzt wird
  - hier nur Überblick, Einzelheiten siehe Handbuch und man-pages
- Beobachtung beim RPC: Grundgerüst ist immer gleich
  --> Grossteil des Aufrufrahmens vorkonfektionierbar
  --> automatische Generierung des Gerüstes



- Der Server richtet sich mit je einem *rpc_reg* für jeden Service ein (--> Anmeldung beim Portverwalter)
- Mit *rpc_run* wartet er dann blockierend (mittels select) auf ein Rendezvous mit dem Client
  - und ruft dann die richtige lokale Prozedur auf
- Mit *rpc_call* wendet sich der Client an den Server
  - wird im Fehlerfall innerhalb einiger Sekunden ein paar Mal wiederholt

Sun-RPC: Komponenten

- RPC-Library: Vielzahl aufrufbarer Funktionen ("API")
  - z.B. rpc_reg, rpc_run, rpc_call
  - daneben auch Funktionen einer Low-level-Schnittstelle: z.B. Spezifikation von Timeout-Werten oder eines Authentifizierungsprotokolls
- rpcgen: Stub-Generator
- Portmapper: Zuordnung Dienstnummer <--> Portadresse
- XDR-Library: Datenkonvertierung
  - Repräsentation der Daten in einem einheitlichen Transportformat

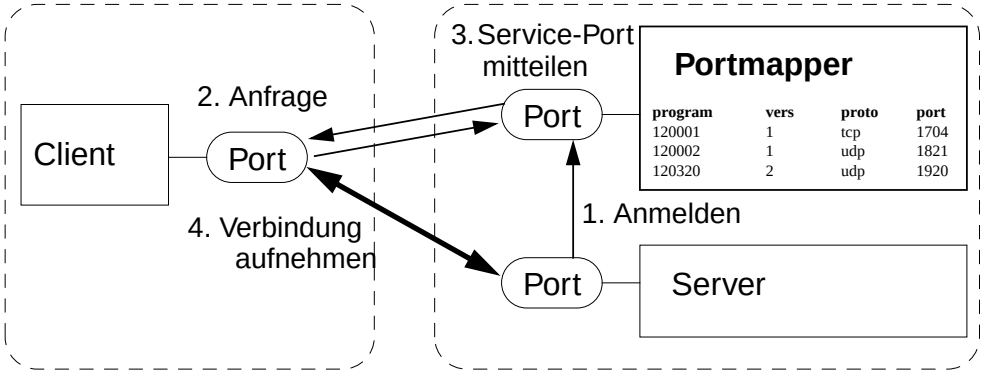
---

- Sicherheitskonzepte
  - z.B. diverse Authentifizierungsvarianten unterschiedlicher "Stärke"
- Semantik: "at least once"
  - jedoch abhängig vom darunter liegenden Kommunikationsprotokoll
- Unterstützt UDP- und TCP-Verbindungen
  - UDP: Datagramme, verbindungslose Kommunikation
  - TCP: Stream, verbindungsorientierte Kommunikation

Der Portmapper

- Bei Kommunikation über TCP oder UDP muss stets eine Portnummer angegeben werden
  - Portnummer ist zusammen mit der IP-Adresse Teil jedes UNIX-Sockets

- Jeder Dienst meldet sich beim lokalen Portmapper mit Programm-, Versions- und Portnummer an
  - Programmnummer ist primäre Kennzeichnung des Dienstes
  - ein Dienst kann in mehreren verschiedenen Versionen (“Releases”) gleichzeitig vorliegen (Koexistenz von Versionen in der Praxis wichtig)



- Portmapper ist ein Service, der die Zuordnung zwischen Programmnummern und Portnummern verwaltet

- Client kontaktiert vor einem RPC zunächst den Portmapper der Servermaschine, um den Port herauszufinden, wohin die Nachricht gesendet werden soll
  - Portmapper hat immer den well-known Port 111
  - BUGS: If portmap crashes, all servers must be restarted

Portmapper (2)

- Interaktive Anfrage beim Portmapper (UNIX / LINUX)
  - shell > rpcinfo -p

```
   program vers proto   port  service
    100000    2   tcp    111  portmapper
    100004    2   udp    743  ypserv
    100004    1   udp    743  ypserv
    100004    1   tcp    744  ypserv
    100001    2   udp  32830  rstatd
    100029    1   udp    657  keyserv
    100003    2   udp   2049  nfs
     ...
536870928     1   tcp   4441
536870912     1   udp   2140
536870912     1   tcp   4611
     ...
```

Dynamisch generierte Port- und Programmnummern

- Bsp.: Auf Port 2049 “horcht” Programm 100003; es handelt sich um das verteilte Dateisystem NFS (Network File Service)

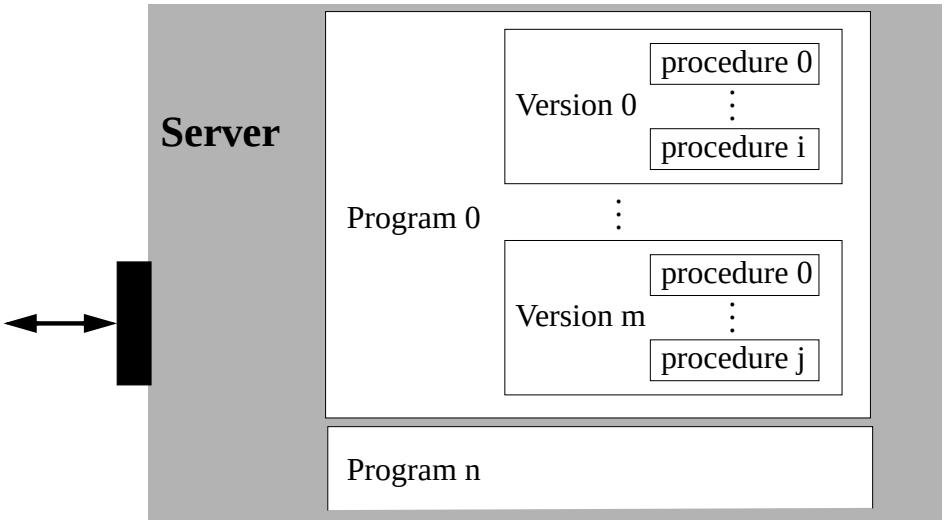
---

rpcinfo makes an RPC call to an RPC server and reports what it finds. ... rpcinfo lists all the registered RPC services with rpcbind on host.... ... makes an RPC call to procedure 0 of prognum and versnum on the specified host and reports whether a response was received.... If a versnum is specified, rpcinfo attempts to call that version of the specified prognum. Otherwise, rpcinfo attempts to find all the registered version numbers for the specified prognum by calling version 0.

-b Make an RPC broadcast to procedure 0 of the specified prognum and versnum and report all hosts that respond.

Service-Identifikation

- Eine entfernte Prozedur wird identifiziert durch das Tripel (prognum, versnum, procnum)



- Jede Prozedur eines Dienstes realisiert eine Teilfunktionalität (z.B. open, read, write... bei einem Dateiserver)

- Prozedur Nummer 0 ist vereinbarungsgemäss für die "Nullprozedur" reserviert

  - keine Argumente, kein Resultat, sofortiger Rückkehr ("ping-Test")

- Mit der Nullprozedur kann ein Client feststellen, ob ein Dienst in einer bestimmten Version existiert:

  - falls Aufruf von Version 4 des Dienstes XYZ nicht klappt, dann versuche, Version 3 aufzurufen...

Service-Registrierung

int *rpc_reg*(prognum, versnum, procnum, procname, inproc, outproc)

Register procedure *procname* with the RPC service package. If a request arrives for program *prognum*, version *versnum*, and procedure *procnum*, *procname* is called with a pointer to its parameter; procname must be a procedure that returns a pointer to its static result; *inproc* is used to decode the parameters while *outproc* is used to encode the results.

- Welche Programmnummer bekommt ein Service?

  --> Einige Programmnummern für *Standarddienste* sind bereits konfiguriert und stehen in /etc/rpc:

portmapper	100000	portmap
rstatd	100001	rup
rusersd	100002	rusers
nfs	100003	nfsprog
ypserv	100004	ypprog
mountd	100005	mount
...	...	
keyserv	100029	keyserver

Linke Spalte: Servicename

Zuordnung mittels *getrpcbyname*() und *getrpcbynumber*() möglich

Rechte Spalte: Kommentar

  --> Ansonsten freie Nummer wählen:

- Mit *pmap_set*(prognum, versnum, protocol, port) bekommt man den Returncode FALSE, falls prognum bereits (dynamisch) vergeben; ansonsten wird dem Service die Portnummer 'port' zugeordnet

neu und "enhanced": "rpcb_set"

TCP oder UDP

Service-Aufruf

int *rpc_call*(host, prognum, versnum, procnum, inproc, in, outproc, out)

Call the remote procedure associated with *prognum*, *versnum*, and *procnum* on the machine, *host*. The parameter *in* is the address of the procedure's argument, and *out* is the address of where to place the result; *inproc* is an XDR function used to encode the procedure's parameters, and *outproc* is an XDR function used to decode the procedure's results.

Warning: You do not have control of timeouts or authentication using this routine.

- Es gibt auch eine entsprechende Broadcast-Variante:

rpc_broadcast(prognum, versnum, procnum, inproc, in, outproc, out, eachresult)

Like rpc_call(), except the call message is broadcast... Each time it receives a response, this routine calls eachresult(). If eachresult() returns 0, rpc_broadcast() waits for more replies.

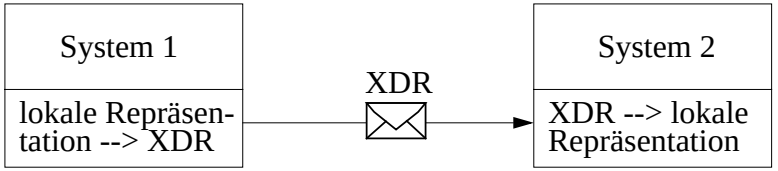
XDR (eXternal Data Representation)

- Sun-Standard zur Beschreibung von Daten in einem hardwareunabhängigen Format

- Formale Sprache zur *Datentyp-Beschreibung*
  - ähnlich zu Typdeklarationen von Pascal, C, etc. bzw. ASN.1

- Definition der *Repräsentation* der Daten, d.h. Kodierungskonventionen. z.B.:
  - Position des höherwertigen Bytes bei Integer
  - Format von Gleitpunktzahlen
  - Länge / Ende von Strings
  - Ausrichtung auf Wortgrenzen bei Verbundtypen
  - Zeichendarstellung: EBCDIC, ASCII usw.

- Prinzip der XDR-Datenkonversion:

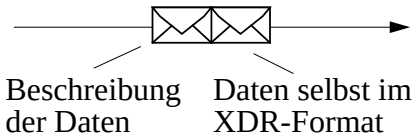


- Beachte: Jeweils zwei Konvertierungen erforderlich; für jeden Systemtyp jeweils Kodierungs- und Dekodierungsroutinen vorsehen

- Alternative ("receiver makes it right"): Kennung der lokalen Repräsentation mitsenden --> Umwandlung entfällt bei gleichen Systemtypen --> ggf. aber insgesamt mehr Umwandlungsroutinen!

XDR (2)

- Weitere Anwendungsmöglichkeit: “Selbstbeschreibende Daten” durch Mitsenden der XDR-Beschreibung:



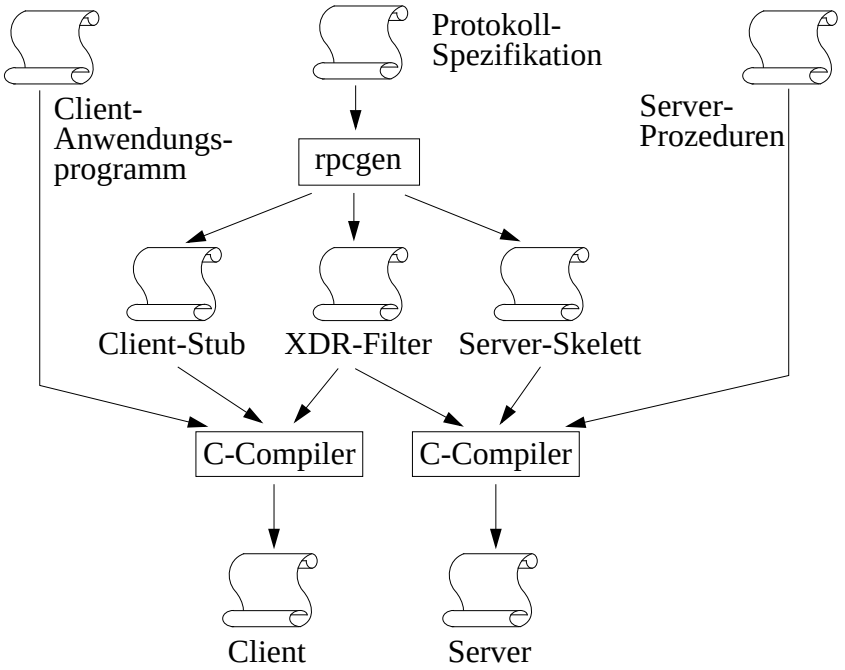
  - *Vorteil*: Empfänger kann Format auf Richtigkeit prüfen
  - *Nachteil*: Grösserer Aufwand (Empfänger sollte eigentlich wissen, was für Daten er erwartet)

- XDR-Library: Menge von C-Funktionen (“XDR-Filter”), die Kodierung / Dekodierung vornehmen

- Aus gegebenen XDR-Filtern für einfache Datentypen lassen sich eigene XDR-Filter (“custom filter”) für komplexe Datentypen (z.B. Strukturen) bauen

Stub- und Filtergenerierung

- *rpcgen-Compiler*: Generiert aus einer Protokollspezifikation (= Programmname, Versionsnummern, Name von Prozeduren sowie Parameterbeschreibung) die Stubs und XDR-Filter



Beispiel zu rpcgen

Die Ausgangsdatei add.x mit der *Protokollspezifikation*:

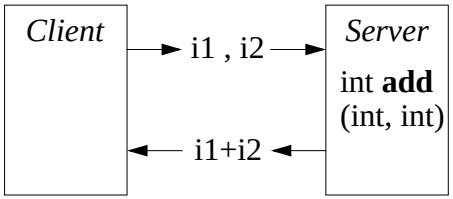
```
struct i_result
{ int x; };

struct i_param
{ int i1;
  int i2; };

program ADD_PROG
{ version ADD_VERS
  { i_result ADDINT
    (i_param) = 1;
  } = 1;
} = 222111;
```

Bem.: Dies ist kein vollständiges Beispiel; es soll nur grob zeigen, was im Prinzip generiert wird.

Beispiel: ein “Additionsserver”:



Der generierte *Headerfile* add.h (Auszug):

```
struct i_result {
        int x;
};
typedef struct i_result i_result;

struct i_param {
        int i1;
        int i2;
};
typedef struct i_param i_param;

#define ADD_PROG ((unsigned long)(222111))
#define ADD_VERS ((unsigned long)(1))
#define ADDINT ((unsigned long)(1))
```

Diese Datei ist zugegebenermassen nicht besonders spannend: i.w. eine “Paraphrase” von add.x

Generierter Client-Code (Auszug)

```
i_result * addint_1(argp, clnt) i_param *argp; CLIENT *clnt;
{       static i_result clnt_res;

        clnt_call(clnt, ADDINT,
                (xdrproc_t) xdr_i_param, (caddr_t) argp,
                (xdrproc_t) xdr_i_result, (caddr_t) &clnt_res, TIMEOUT)
        return (&clnt_res);
}

void add_prog_1
{       char *host;
       CLIENT *clnt;
        i_result  *result_1;
        i_param  addint_1_arg;

        clnt = clnt_create(host, ADD_PROG, ADD_VERS, “netpath”);

        result_1 = addint_1(&addint_1_arg, clnt);
...
}
```

im handle “clnt” stecken die weiteren Angaben

die beiden Routinen xdr_i_param und xdr_i_result werden ebenfalls von rpcgen generiert (hier nicht gezeigt)

hier Server (“host”) lokalisieren!

hier Parameter setzen!

eigentlicher Prozeduraufruf

RPC library routines: ... First a CLIENT handle is created and then the client calls a procedure to send a request to the server.

CLIENT *clnt_create(const char *host, const u_long prognum, const u_long versnum, const char *nettype);

Generic client creation routine for program prognum and version versnum. nettype indicates the class of transport protocol to use.

enum clnt_stat clnt_call(CLIENT *clnt, const u_long procnum, const xdrproc_t inproc, const caddr_t in, const xdrproc_t outproc, caddr_t out, const struct timeval tout);

A function macro that calls the remote procedure procnum associated with the client handle, clnt. The parameter inproc is the XDR function used to encode the procedure’s parameters, and outproc is the XDR function used to decode the procedure’s results; in is the address of the procedure’s argument(s), and out is the address of where to place the result(s). tout is the time allowed for results to be returned.

Generierter Server-Code (Auszug)

```
if (!svc_reg(transp, ADD_PROG, ADD_VERS, add_prog_1, 0))
{_msgout("unable to register (ADD_PROG, ADD_VERS).");

svc_run();
```

svc_reg funktioniert analog zu rpc_reg

```
i_result * addint_1(argp, rqstp)
        i_param *argp;
        struct svc_req *rqstp;
{       static i_result  result;
        /* insert server code here */
        return (&result);
}

static void add_prog_1(rqstp, transp)
{  switch (rqstp->rq_proc) {
   case NULLPROC:
        (void) svc_sendreply(transp, xdr_void, (char *)NULL);
        return;
   case ADDINT:
        _xdr_argument = xdr_i_param;
        _xdr_result = xdr_i_result;
        local = (char *(*)()) addint_1;
        break;
   default:
        svcerr_noproc(transp);
   }

   svc_getargs(transp, _xdr_argument, (caddr_t) &argument)
   result = (*local)(&argument, rqstp);
   ... svc_sendreply(transp, _xdr_result, result) ...
}
```

Bem.: Server-Code ist über 200 Zeilen lang

result.x =argp->i1 + argp->i2

bool_t svc_sendreply(const SVCXPRT *xprt, const xdrproc_t outproc, const caddr_t out);

Called by an RPC service's dispatch routine to send the results of a remote procedure call. The parameter xprt is the request's associated transport handle; outproc is the XDR routine which is used to encode the results; and out is the address of the results.

Generierte XDR-Konversionsroutinen

```
…
bool_t xdr_i_result(xdrs, objp)
 XDR *xdrs;
 i_result *objp;

{
 if (!xdr_int(xdrs, &objp->x)) return (FALSE);
 return (TRUE);
}


bool_t xdr_p_result(xdrs, objp)
 XDR *xdrs;
 i_param *objp;

{
 if (!xdr_int(xdrs, &objp->i1)) return (FALSE);
 if (!xdr_int(xdrs, &objp->i2)) return (FALSE);
 return (TRUE);
}
```

Sicherheitskonzept des Sun-RPC

- Nur Unterstützung zur Authentifizierung; Autorisierung (= Zugriffskontrolle) muss der Server selbst realisieren!

- Authentifizierung basiert auf zwei Angaben, die i.a. bei einem RPC-Aufruf mitgeschickt werden:

  - *Credential*: Identifiziert einen Client oder Server
    (Vgl. Angaben auf einem Reisepass)

  - *Verifier*: Soll Echtheit des Credential garantieren
    (Vgl. Passfoto)

- Feld im Header einer RPC-Nachricht spezifiziert eines der möglichen Authentifizierungsprotokollen ("flavors"):

  - *NONE:* keine Authentifizierung
    - Client kann oder will sich nicht identifizieren
    - Server interessiert sich nicht für die Client-Identität
    - Credential und Verifier sind beide NULL

  - *SYS:* Authentifizierung im UNIX-Stil

  - *DES:* echte Authentifizierung ("Secure RPC")

  - *KERB:* Authentifizierung mit Kerberos
    - Kerberos-Server muss dann natürlich installiert sein

SYS-"Flavor" bei Sun-RPC

- Sinnvoll, wenn im Sinne der UNIX-Sicherheitsphilosophie der Zugang zu gewissen Diensten auf bestimmte Benutzer / Benutzergruppen beschränkt werden soll

- Es wird mit dem RPC-Request folgende Struktur als Credential versandt (kein Verifier!):

```
{unsigned int stamp;
 string machinename (255);
 unsigned int uid;
 unsigned int gid;
 unsigned int gids (...);
};
```

Effektive user-id des Client

Effektive Gruppen-id

Weitere Gruppen, in denen der Client Mitglied ist

- Server kann die Angaben verwenden, um den Auftrag ggf. abzulehnen

- Server kann zusammen mit der Antwort eine *Kurzkennung* an den Client zurückliefern
  - Client kann bei zukünftigen Aufrufen die Kurzkennung verwenden
  - Server hält sich eine Zuordnungstabelle

- Probleme...
  - gleiche Benutzer müssen auf verschiedenen Systemen die gleiche (numerische) uid-Kennung haben
  - ungesichert gegenüber Manipulationen
  - nur in verteilten UNIX-Systemen sinnvoll anwendbar

Secure RPC mit DES

- Im Unterschied zum UNIX-Flavor: Weltweit eindeutige Benutzernamen (“netname”) als String (= Credential)
  - in UNIX z.B. mittels user2netname() generiert aus Betriebssystem, user-id und eindeutigem domain-Namen, z.B.: unix.37@fix.cs.uni-xy.eu

- Client und Server vereinbaren einen DES-Session-key K nach dem Diffie-Hellman-Prinzip

- Mit jeder Request-Nachricht wird ein mit K kodierter Zeitstempel mitgesandt (= Verifier)

- Die erste Request-Nachricht enthält ausserdem verschlüsselt die Window-Grösse W als zeitliches Toleranzintervall sowie (verschlüsselt) W-1
  - “zufälliges” Generieren einer ersten Nachricht nahezu unmöglich!
  - replay (bei kleinem W) ebenfalls erfolglos!

- Server überprüft jeweils, ob:
  (a) Zeitstempel grösser als letzter Zeitstempel
  (b) Zeitstempel innerhalb des Zeitfensters

- Die Antwort des Servers enthält (verschlüsselt) den letzten erhaltenen Zeitstempel-1 (--> Authentifizierung!)

- Gelegentliche Uhrenresynchronisation nötig (RPC-Aufruf kann hierzu optional die Adresse eines “remote time services” enthalten)