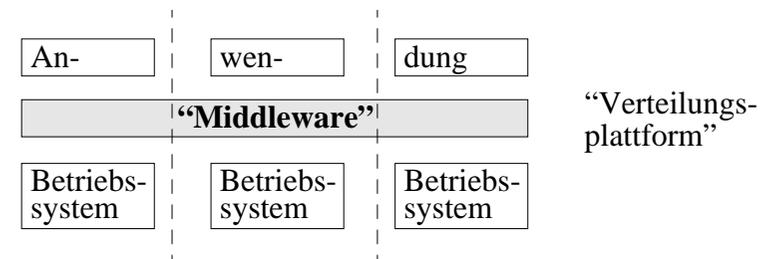


# Middleware

## Middleware

- Kann man durch eine geeignete Softwareinfrastruktur die Realisierung verteilter Anwendungen vereinfachen?
  - wieso ist das überhaupt so schwierig?
  - kann man für viele Anwendungen gemeinsame Aspekte herausfaktorisieren?

- Lösung: Zauberwort “Middleware”



- Aufgabe:
  - Verteilung (für die Anwendung) möglichst transparent machen (z.B. umspannender Namensraum, globale Zugreifbarkeit, Ortstransparenz)
  - zumindest aber die Verteilung einfach handhabbar machen
- Soll insbesondere Kommunikation und Kooperation zwischen Anwendungsprogrammen unterstützen
  - Verbergen von Heterogenität von Rechnern und Betriebssystemen (z.B. durch einheitliche Datenformate)
  - einheitliche „Umgangsformen“: Schnittstellen, Protokolle
- Sollte gewisse Basismechanismen für verteiltes Programmieren anbieten, z.B.
  - Verzeichnis- und Suchdienste (Nameservice, Tradingservices...)
  - automatische Schnittstellenanpassung (Schnittstellenbeschreibungssprache, Stub-Compiler...)

# Der Weg zum „Netzwerkrechner“

## 1. RPC-Pakete: z.B. Sun-RPC

- Client-Server-Paradigma, RPC-Kommunikation
- Schnittstellen-Beschreibungssprache, Datenformatkonversion, Stubgeneratoren
- Sicherheitskonzepte (Authentifizierung, Autorisierung, Verschlüsselung)

## 2. Client-Server-Verteilungsplattformen: z.B. DCE

- Zeitdienst, Verzeichnis- und Suchdienst
- globaler Namensraum, globales Dateisystem
- Programmierhilfen: Synchronisation, Multithreading ...

## 3. Objektbasierte Verteilungsplattformen: z.B. CORBA

- Kooperation zwischen gleichberechtigten („peer-to-peer“-) Objekten
- objektorientierte Schnittstellenbeschreibungssprache, Vererbung
- Objekt Request Broker

## 4. Infrastruktur für spontane Kooperation (z.B. Jini)

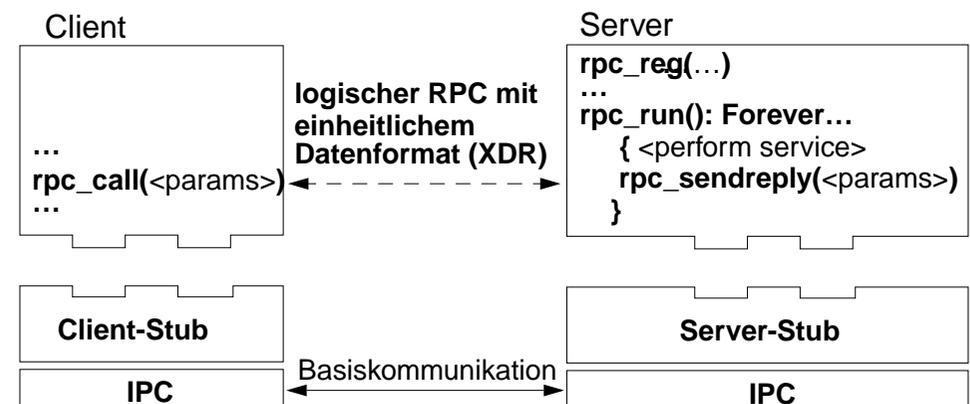
- unterstützt Dienstorientierung, Mobilität, Dynamik

Beachte: Der Begriff „Middleware“ ist leider im Laufe der Zeit zunehmend verwässert worden

- oft weniger gebraucht im technischen Sinne als Verteilungsplattform und Kommunikations- und Dienstinfrastruktur
- sondern „alles“ was nicht gerade Anwendung oder Betriebssystem ist, also auch Datenbanken, Workflow,...

# 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 Gerüsts



- 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

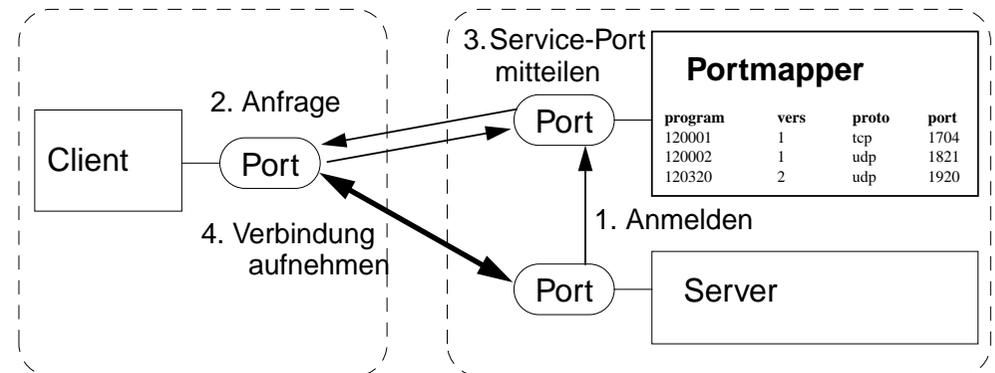
# Sun-RPC: Komponenten

- 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 Sun-OS)

- 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

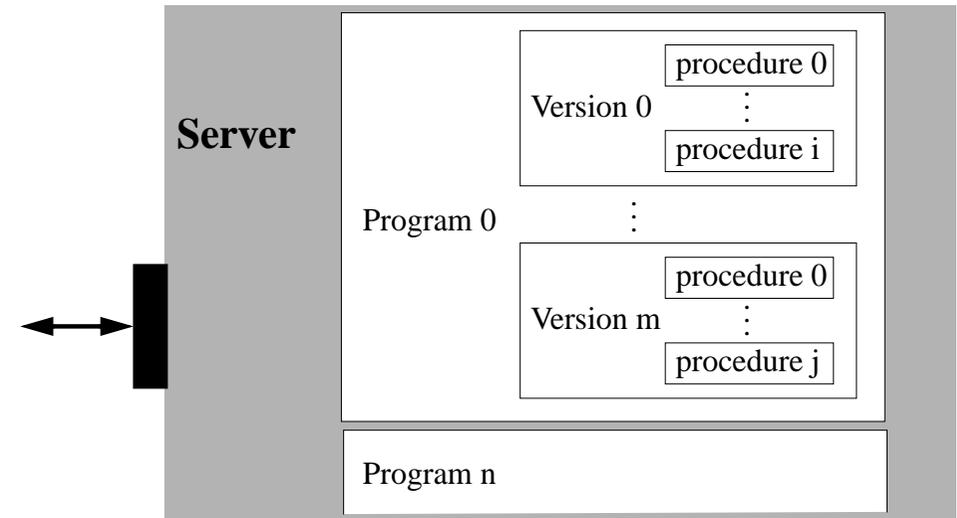
- Bsp.: Auf Port 2049 "horcht" Programm 100003; es handelt sich um das verteilte Dateisystem NFS (Network File Service)

**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 program 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 program. Otherwise, rpcinfo attempts to find all the registered version numbers for the specified program by calling version 0.

- b Make an RPC broadcast to procedure 0 of the specified program 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
...	...	...
keyserver	100029	keyserver

Linke Spalte:  
Servicename

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

Rechte Spalte:  
Kommentar

--> Ansonsten freie Nummer wählen:

neu und "enhanced": "rpcb\_set"

TCP oder UDP

- 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

# 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.

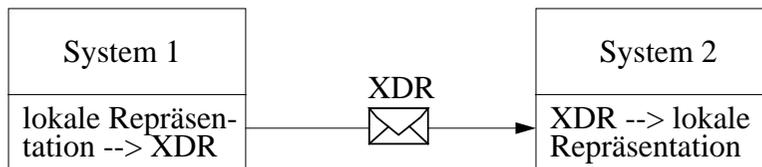
- Es gibt auch eine entsprechende Broadcast-Variante:

```
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)

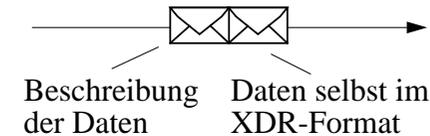
- Sun-Standard zur Beschreibung von Daten in einem hardwareunabhängigen Format
- Formale Sprache zur *Datentyp-Beschreibung*
  - ähnlich zu Typdeklarationen von Pascal, C, etc. bzw. ASN.1
- Definition der *Repräsentation* der Daten, d.h. Kodierkonventionen. z.B.:
  - Position des höherwertigen Bytes 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 Dekodierungen 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:

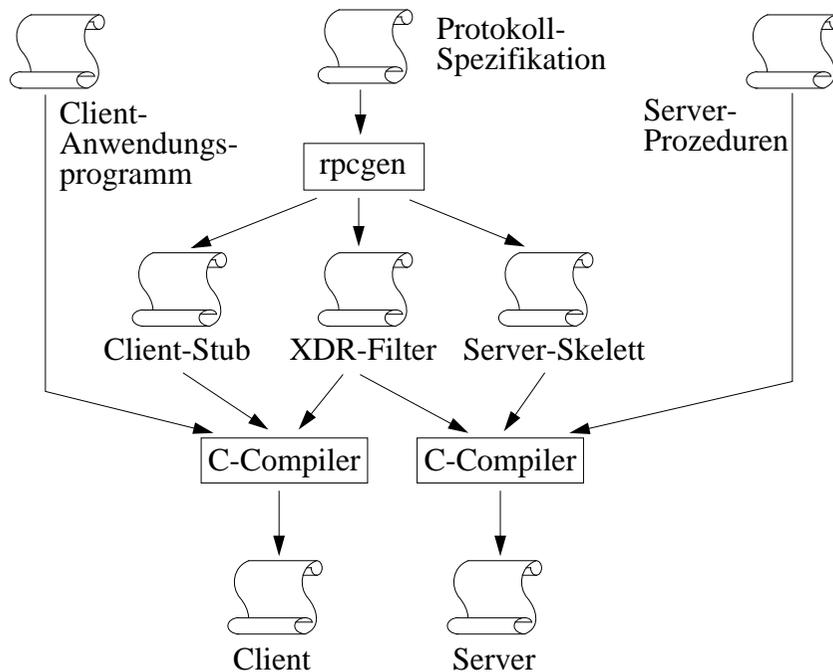


- *Vorteil*: Empfänger kann Format auf Richtigkeit prüfen
- *Nachteil*: Größerer Aufwand (Empfänger sollte eigentlich wissen, was für Daten er erwartet)

- 
- 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*:

```

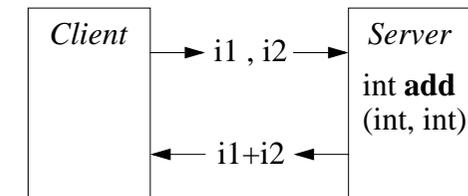
struct i_result
{ int x; };

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);
    ...
}

```

Annotations for Client Code:

- im handle "clnt" stecken die weiteren Angaben (points to `clnt` in `clnt_call`)
- die beiden Routinen `xdr_i_param` und `xdr_i_result` werden ebenfalls von `rpcgen` generiert (hier nicht gezeigt) (points to `xdr_i_param` and `xdr_i_result` in `clnt_call`)
- hier Server ("host") lokalisieren! (points to `host` in `clnt_create`)
- hier Parameter setzen! (points to `&addint_1_arg` in `addint_1`)
- eigentlicher Prozeduraufruf (points to `addint_1` in `addint_1`)

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();

```

Annotation: `svc_reg` funktioniert analog zu `rpc_reg` (points to `svc_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)(amp;argument, rqstp);
... svc_sendreply(transp, _xdr_result, result) ...
}

```

Annotations for Server Code:

- Bem.: Server-Code ist über 200 Zeilen lang (points to the `addint_1` function)
- result.x = argp->i1 + argp->i2 (points to the `/* insert server code here */` comment)

**bool\_t svc\_sendreply(const SVCXPRT \*xpvt, 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 `xpvt` 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 *Kurz-kennung* an den Client zurückliefern
  - Client kann bei zukünftigen Aufrufen die Kurz-kennung verwenden
  - Server hält sich eine Zuordnungstabelle

## - Probleme...

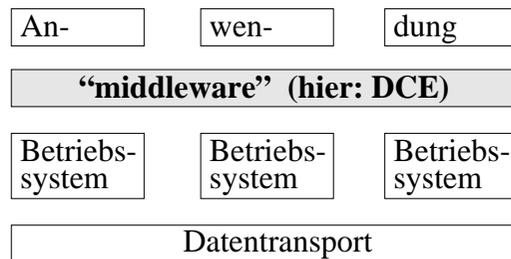
- gleiche Benutzer müssen auf verschiedenen Systemen die gleiche (numerische) uid-Kennung haben
- ungesichert gegenüber Manipulationen
- nur in verteilten UNIX-Systemen sinnvoll anwendbar

# 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)

# DCE - Distributed Computing Environment

- Entwickelt von einem herstellerübergreifenden Konsortium ("OSF" - Open Software Foundation)
  - Anfang der 90er Jahre, u.a. DEC, IBM, Siemens, HP...
  - trotz CORBA noch vielfältig eingesetzt in grossen Organisationen
- System aus zusammenwirkenden Softwarekomponenten (Werkzeuge, Dienste, Laufzeitmechanismen) zur Realisierung verteilter Anwendungen in offenen heterogenen Umgebungen



- Ziel: Schaffung eines Industriestandards für verteilte Verarbeitung
- Vorgehensweise pragmatisch: Soweit möglich, Nutzung geeigneter existierender Technologiekomponenten
- Realisierung auf verschiedenen Plattformen
  - Hardware: IBM, Sun, ...
  - Software: UNIX-basiert, z.B. HP-UX, Solaris, AIX; aber auch Windows, OS/390, Macintosh,...

# Offene Systeme

- Offenlegung von Schnittstellen und Spezifikationen
- Einfache Portierungsmöglichkeit auf viele Systeme
- Interoperabel mit anderen Systemen
- Wechsel von Benutzern zwischen Systemen verschiedener Hersteller einfach

## Vorteile für den Nutzer:

- Herstellerunabhängigkeit
- Kompatibilität verschiedener Systeme
- Kohärenz bzgl. der Bedienbarkeit
- Investitionsschutz

## Standardisierung daher notwendig; z.B. Schnittstellen

- zur Systemumgebung
  - zwischen (unabhängigen) Systemkomponenten
  - zum Benutzer
- > Konsortien: Empfehlungen, Richtlinien...
- > internationale Standardisierungsorganisationen (z.B. ISO; ITU): Normen, Standards

## Konsortien (für offene Systeme)

- Non-profit-Vereinigungen von Herstellern
  - ggf. auch unter Einbeziehung von Nutzern
- Sicherstellung von Plattformunabhängigkeit, Portabilität und Interoperabilität von Systemen
  - Normen, Testszenarien, Zertifizierungsgremien...
  - Arbeitsgruppen zu technischen Fragen und Standards
  - Richtlinien für unabhängige Softwareentwickler
  - Verpflichtung einzelner Mitglieder, gemeinsame Entscheidungen zu unterstützen
- Auswahl, Entwicklung, Anpassung, Zusammenstellung von "Technologiekomponenten"
  - z.B. Betriebssystem, Protokolle, Benutzeroberflächen, Programmbibliotheken, Management-Tools, Entwicklungssysteme...
- Etablierung oft aus marktstrategischen Gründen
  - Investitionsschutz (Entwicklungskosten, Lizenzen...)
  - Vermeidung von Mehrfachentwicklung
  - strategisch / politische Erwägungen (z.B. EU-Richtlinien)
  - Durchsetzen von Normen
  - beschleunigte Produktentwicklung durch strategische Allianzen
  - Gegenallianzen zu Monopolisten

## Konsortien...

Neben der OSF gibt es noch weitere Konsortien, z.B.:

OMG (Object Management Group), bekannt durch das CORBA-Modell

Common Object Request Broker Architecture

- Mehr als 500 Mitglieder (HP, ATT, Sun...)
- Ziel: Bereitstellung von Konzepten für die Entwicklung verteilter Anwendungen mit objektorientierten Modellen
- OMA (Object Management Architecture) und OSA (Object Services Architecture): Dienste zur Verwaltung von Objekten in verteilten heterogenen Systemen (z.B. Trading, verteilte Transaktionen, Replikation, Speicherung, Namensverwaltung, Persistenz, Migration, Security...)

# DCE - Der Hoffnungsträger (1993)

## DCE mit InterFace

**IF**

InterFace Computer GmbH  
Garnsieder Straße 4  
D-50009 Wülfrath 2  
Tel.: 0049/510 35-0  
Fax: 0049/51000-20

All together

Mit DCE (Distributed Computing Environment) existiert endlich ein internationaler, von der OSF entwickelter Standard für die intelligente Integration unterschiedlicher Computerplattformen – vom PC über UNIX-Netze bis hin zu Mainframes oder externen Rechenzentren. An die Stelle eines Nebeneinanders tritt das produktive Zusammenwirken proprietärer Systeme. Eine optimale Nutzung vorhandener Ressourcen, die volle Rechenleistung an jedem Punkt des Netzes und ein unternehmensweiter Zugriff auf Datenbestände werden realisierbar.

Die InterFace Computer GmbH hat sich schon während der Entwicklungsphase intensiv mit der DCE-Technologie beschäftigt. Früher als andere sind wir deshalb in der Lage, den neuen Standard in praktische Lösungen umzusetzen – im Hinblick auf Systemberatung, Integration bestehen-

der oder Entwicklung neuer Systeme ebenso wie durch eigene entwickelte Schulungsangebote. So stellen Ihnen von Anfang an alle durch DCE ermöglichten Vorteile zur Verfügung: Höhere Effizienz durch verteilte Programme; die automatische Zuteilung freier Rechnerkapazitäten; das globale Zusammenspiel aller Hard- und Softwarekomponenten in einem offenen System. Die Komplexität des Netzes bleibt dabei für Anwender unsichtbar; der integrierte Security Service gewährleistet Datenschutz und Datensicherheit auf allen Ebenen und in allen Betriebszuständen.

Kooperation ist eben auch in der EDV die bessere Lösung als isoliertes Spezialistentum. Sprechen Sie mit uns.

now.



# Werbesprüche...

Mit DCE (Distributed Computing Environment) existiert endlich ein internationaler, von der OSF entwickelter Standard für die intelligente Integration unterschiedlicher Computerplattformen – vom PC über UNIX-Netze bis hin zu Mainframes oder externen Rechenzentren. An die Stelle eines Nebeneinanders tritt das produktive Zusammenwirken proprietärer Systeme. Eine optimale Nutzung vorhandener Ressourcen, die volle Rechenleistung an jedem Punkt des Netzes und ein unternehmensweiter Zugriff auf Datenbestände werden realisierbar.

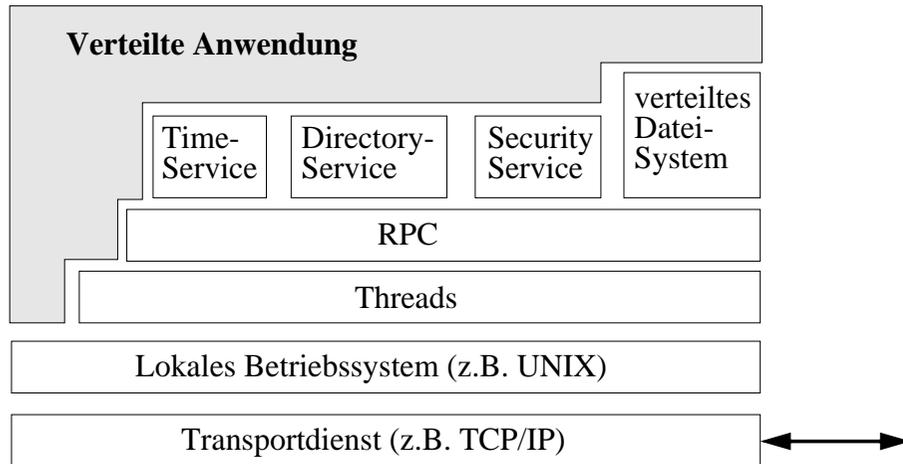
Die InterFace Computer GmbH hat sich schon während der Entwicklungsphase intensiv mit der DCE-Technologie beschäftigt. Früher als andere sind wir deshalb in der Lage, den neuen Standard in praktische Lösungen umzusetzen – im Hinblick auf Systemberatung, Integration bestehen-

der oder Entwicklung neuer Systeme ebenso wie durch eigene entwickelte Schulungsangebote. So stehen Ihnen von Anfang an alle durch DCE ermöglichten Vorteile zur Verfügung: Höhere Effizienz durch verteilte Programme; die automatische Zuteilung freier Rechnerkapazitäten; das globale Zusammenspiel aller Hard- und Softwarekomponenten in einem offenen System. Die Komplexität des Netzes bleibt dabei für Anwender unsichtbar; der integrierte Security Service gewährleistet Datenschutz und Datensicherheit auf allen Ebenen und in allen Betriebszuständen.

Kooperation ist eben auch in der EDV die bessere Lösung als isoliertes Spezialistentum. Sprechen Sie mit uns.

now.

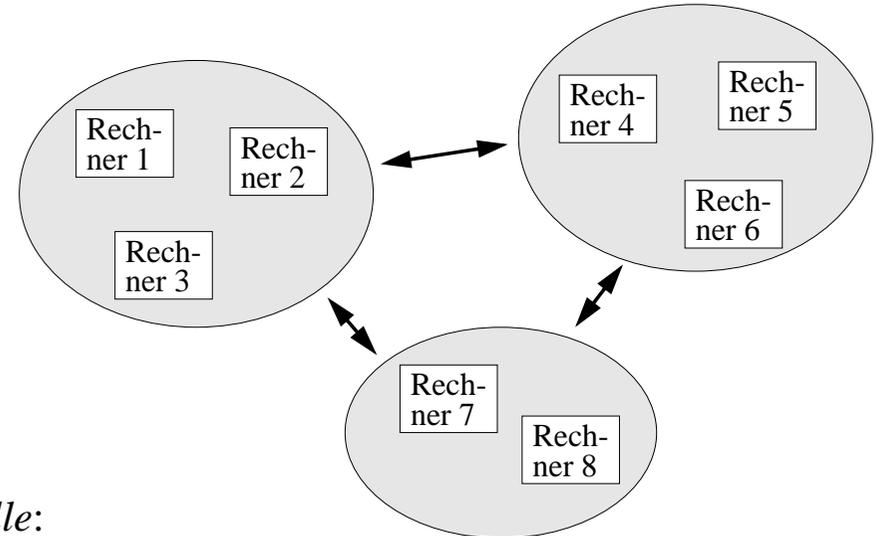
# Hauptkomponenten des DCE



- Baut auf lokalem Betriebssystem und existierendem Transportdienst (z.B. TCP/IP) auf
- Threads und RPC sind Basisdienste, die von anderen Diensten (aber auch von Anwendungen) benutzt werden
- Höhere ("verteilte") Dienste: u.a. Dateisystem, Verzeichnis- und Namensdienst, Sicherheitsdienst
- Eine verteilte Anwendung nutzt die Dienste i.a. über Programmierschnittstellen (API: Application Programming Interface), die für die Sprache C ausgelegt sind
- Es gibt ferner Tools für Stub-Generierung von RPCs, Systemmanagement etc.

# Globale DCE-Architektur: Zellen

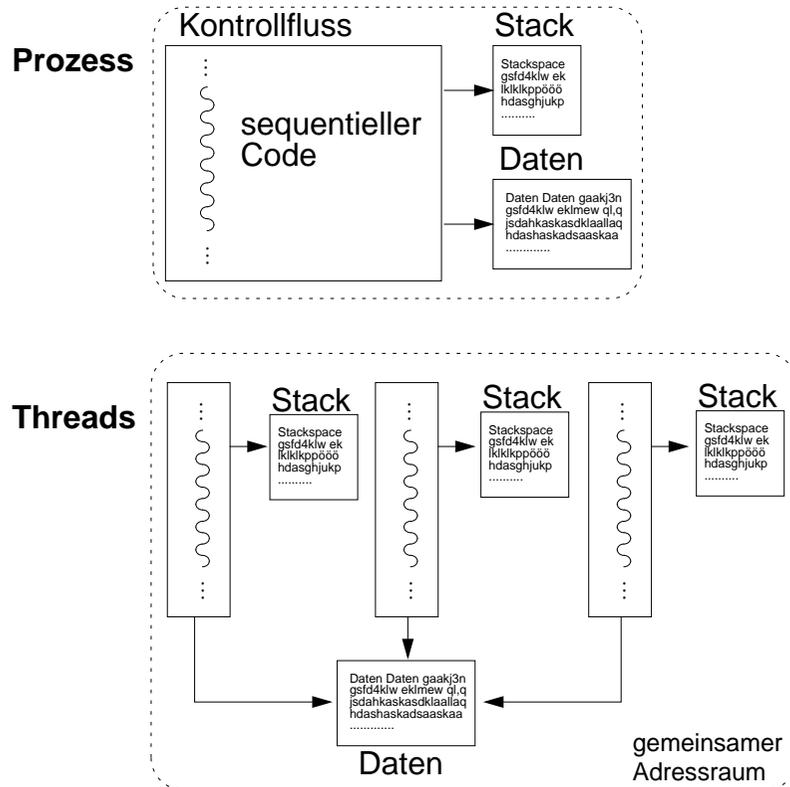
- Partitionierung der Rechner in sogen. *Zellen*
- Subsysteme machen grosse Systeme handhabbarer



- *Zelle:*
  - Ist eine abgeschlossene organisatorische Einheit aus Rechnern, Ressourcen, Benutzern
    - z.B. Abteilung einer Firma
    - i.a. jeweils verantwortlicher Systemverwalter notwendig
    - bildet jeweils eine eigene Schutzzone bzgl. Sicherheitsaspekte
  - Hat *Cell Directory Service (CDS)*, *Security Service* und *Time Service* eingerichtet
    - realisiert durch dauerhafte Prozesse ("Dämonen")
    - ggf. weitere Dienste, z.B. Distributed File System (DFS)
- Prozesse können per RPC zellübergreifend kommunizieren (bei Kenntnis entfernter Adressen)
- *Zellübergreifende Services* (z.B. Zeitservice, Namensverwaltung...) mittels dedizierter Protokolle

# DCE: Threads

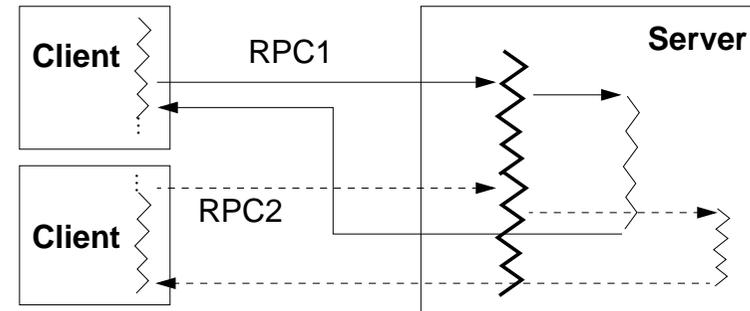
- Threads = leichtgewichtige Prozesse mit gemeinsamem Adressraum



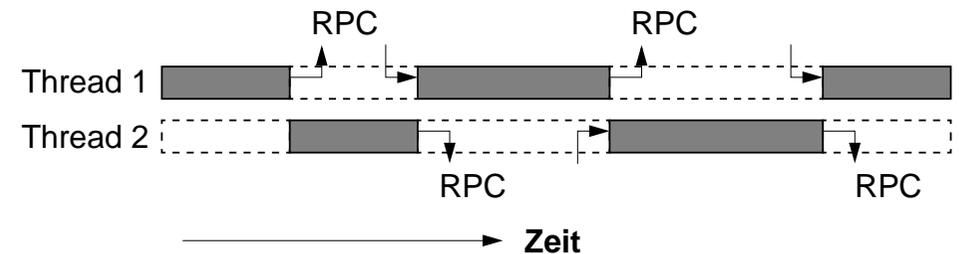
- Einfache Kommunikation zwischen Kontrollflüssen
  - aber: kein gegenseitiger Schutz; ggf. Synchronisation bzgl. Speicher
- Thread hat weniger Zustandsinformation als ein Prozess
- Kontextwechsel i.a. wesentlich schneller
  - kein Umschalten des Adressraumkontexts
  - Cache und Translation Look Aside Buffer (TLB) bleiben "warm"
  - ggf. Umschaltung ohne Wechsel in privilegierten Modus (aufwendig!)

# Wozu Multithreading bei Client-Server-Middleware?

- *Server*: quasiparallele Bearbeitung von RPC-Aufträgen
  - Server bleibt ständig empfangsbereit



- *Client*: Möglichkeit zum „asynchronen RPC“
  - Hauptkontrollfluss delegiert RPCs an nebenläufige Threads
  - keine Blockade durch Aufrufe im Hauptfluss
  - echte Parallelität von Client (Hauptkontrollfluss) und Server



# DCE-Threadkonzept

- Thread-Konzept basiert auf *POSIX-Standard* 1003.4a
- Grössere Zahl von *C-Bibliotheksfunktionen*
  - Erzeugen, Löschen von Threads
  - Synchronisation durch globale Sperren, Semaphore, Bedingungsvariablen
  - warten eines Threads auf ein Ereignis eines anderen Threads
  - wechselseitiger Ausschluss mehrerer Threads (“mutex”)
  - nebenläufige Signalverarbeitung und Ausnahmebehandlung
- Pro Adressraum existiert ein eigener *Thread-Scheduler* mit wählbarer Strategie
  - verschiedene Schedulingstrategien wählbar (z.B. FIFO, Round Robin)
  - wahlweise Verwendung von Zeitscheiben (“präemptiv”)
  - wahlweise Berücksichtigung von Prioritäten

# Problematik von DCE-Threads

- Aufrufe des Betriebssystem-Kerns sind i.a. problematisch
  - a) *nicht ablaufinvariante* (“non-reentrant”) Systemroutinen
    - interne Statusinformation, die ausserhalb des Stacks der Routine gehalten wird, kann bei paralleler Verwendung überschrieben werden
    - z.B. printf: ruft intern Speichergenerierungsroutine auf; diese benutzt prozesslokale Freispeicherliste, deren “gleichzeitige” nicht-atomare Manipulation zu Fehlverhalten führt
    - “Lösung”: Verwendung von “Jacket-Routinen” (wrapper), die gefährdete Routinen kapseln und Aufrufe wechselseitig ausschliessen
  - b) *blockierende* (“synchrone”) Systemroutinen
    - z.B. synchrone E/A, die alle Threads des Prozesses blockieren würde statt nur den aufrufenden Thread
    - “Lösung”: Verwendung asynchrone Operationen zum Test auf mögliche Blockaden

---

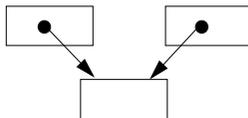
## - Prinzipielle Probleme der Thread-Verwendung:

- fehlender gegenseitiger Adressraumschutz --> schwierige Fehler
- Stackgrösse muss bei Gründung i.a. statisch festgelegt werden
  - > unkalkulierbares Verhalten bei Überschreitung
- von asynchrone Meldungen (“Signale”, “Interrupts”) an den Prozess soll i.a. nur ein einziger (der “richtige”) Thread betroffen werden
- knifflige Synchronisation --> Deadlockgefahr

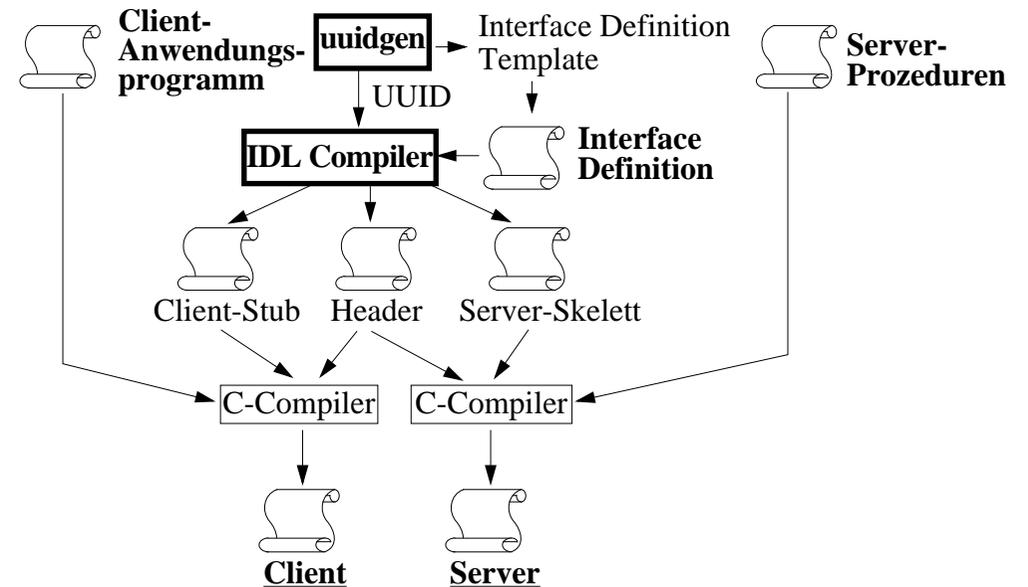
# DCE-RPC

- Weder kompatibel zu Sun-RPC noch zur OSI-Norm
- Ein- und Ausgabeparameter: out, in, in/out
- Nahezu beliebige Parametertypen
  - alle C-Datentypen ausser Prozeduradressen
  - auch verzeigerte Strukturen, dynamische arrays
  - Zeiger werden automatisch dereferenziert und als Wert übergeben; jedoch Vorsicht bei Aliaszeigern!
- Automatische Formatkonvertierung zwischen heterogenen Rechnern
  - Prinzip: "Receiver makes it right"
- Beschreibung der Schnittstelle durch deklarative Sprache IDL ("Interface Description Language")
  - analog, aber nicht identisch zu Sun-RPC
  - IDL-Compiler (entspricht etwa rpcgen bei Sun-RPC) erzeugt Stubs für Client und Server, in denen u.a. die Konvertierung erfolgt

müsste "remote" interpretiert werden



# DCE: Erzeugen von Client- und Server-Programmen



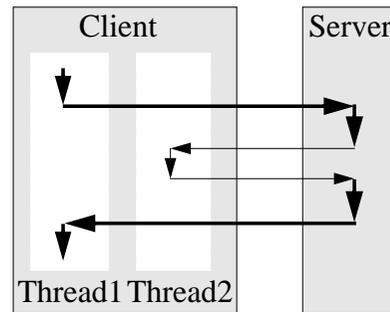
- UUID (Universal Unique Identifier) ist eine aus Uhrzeit und Rechnerkennung generierte systemweit eindeutige Kennung der Schnittstelle

# DCE-RPC: Besonderheiten

- *Asynchrone Aufrufe* durch explizite parallele Threads
  - Kritik: umständlich, Threads sind potentiell fehleranfällig

- *Rückrufe* (“call back RPC”)

- temporärer Rollentausch von Client und Server
- um evtl. bei langen Aktionen Zwischenresultate zurückzumelden
- um evtl. weitere Daten vom Client anzufordern
- Client muss Rückrufadresse übergeben



- *Pipes* als spezielle Parametertypen

- sind selbst keine Daten, sondern ermöglichen es, Daten stückweise zu empfangen (“pull”-Operation) oder zu senden (“push”)
- evtl. sinnvoll bei der Übergabe grosser Datenmengen
- evtl. sinnvoll, wenn Datenmenge erst dynamisch bekannt wird (z.B. Server, der sich Daten aus einer Datenbank besorgt)

- *Context-handles* zur aufrufglobalen Zustandsverwaltung

- werden vom Server dynamisch erzeugt und an Client zurückgegeben
- Client kann diese beim nächsten Aufruf unverändert wieder mitsenden
- Kontextinformation zur Verwaltung von Zustandsinformation über mehrere Aufrufe hinweg z.B. bei Dateiserver (read; read) sinnvoll
- Vorteil: Server arbeitet “zustandslos“

# DCE-RPC: Probleme

Zum Beispiel:

*My server gets a stack error when sending large objects. How can I avoid this?*

Each thread in a process is assigned a fixed area for its procedure-call stack. The stubs normally marshal and unmarshal parameters in space allocated on the thread’s stack. If the parameters are large, the stack size may be exceeded. In most thread implementations, the stack size cannot be increased after the thread is created. For threads created explicitly by your application, you can adjust the size of the thread stack by setting an attribute before calling `pthread_create()`. However, server threads are created automatically, so that method won’t work; instead, call `rpc_mgmt_set_server_stack_size()` before starting the threads with `rpc_server_listen()`.

Another possibility is to use the `[heap]` attribute to have some parameter types marshalled on the heap instead of the stack.

You should know that current implementations of the IDL compiler generate recursive code to marshal linked lists. Therefore, passing a long linked list may cause stack overflow due to all the recursive calls.

# DCE-RPC: Anmeldung von Diensten

- Ein Dienst muss mittels mehrerer Systemaufrufe an drei Stellen bekannt gemacht werden
  - dazu gehört stets auch die Bekanntgabe der vom IDL-Compiler erzeugten und registrierten Dienstschnittstelle
- 1) “Exportieren” des Dienstes durch Anmeldung beim Directory Service der eigenen Zelle
  - Bekanntgabe der Adresse der Server-Maschine
  - ermöglicht es Clients, den Server zu lokalisieren
- 2) Adresse des Dienst-Prozesses (“endpoint”) in eine “endpoint-map” der Server-Maschine eintragen
  - Endpoints entsprechen Ports bei TCP/IP
  - Map wird auf jedem Rechner von einem RPC-Dämon verwaltet
- 3) Registrieren beim lokalen RPC-Laufzeitsystem
  - damit können eintreffende Aufrufe an den zuständigen Dienstprozess weitergeleitet werden (“dispatching”)
  - Angabe, welches Protokoll verwendet werden soll
  - Angabe, wie viele Aufrufe serverseitig gepuffert werden sollen
- Schliesslich teilt der Dienst dem RPC-Laufzeitsystem mit, dass er bereit ist, Aufrufe entgegenzunehmen (“listen”)
  - Angabe, wieviele Aufrufe maximal gleichzeitig bearbeitet werden können --> automatisches Erzeugen von Threads

# Bindevorgang beim DCE-RPC

- Binden = (dyn.) Zuordnung von Client und Server
- Bindevorgang wird eingeleitet durch RPC-Aufruf:

- 1) RPC-Laufzeitsystem des Client stellt fest, dass Prozedur nicht lokal verfügbar ist
- 2) Befragung des Cell Directory Services (CDS)
- 3) CDS liefert Netzadresse der Server-Maschine
- 4) Client wendet sich an den RPC-Dämon der Server-Maschine
- 5) Client erhält dortigen Endpoint des Dienstes

- *zweiphasiger Ablauf* vorteilhaft, da Netzadressen von Services i.a. stabil sind, während sich Endpoints i.a. nach Neustart eines Rechners ändern

- 
- Statt des o.g. *automatischen Bindens*, das für den Client transparent abläuft, ist auch *explizites Binden* möglich:

- umständlicher, aber flexibler
- z.B. programmierte Auswahl eines Backup-Servers, wenn Bindevorgang mit Primärserver unmöglich
- z.B. explizite Auswahl eines Servers einer Gruppe (Lastausgleich etc.)

- 
- Dienste haben eine *Hauptversion* und eine *Unterversion*

- wird beim IDL-Compilieren angegeben, z.B. “3.2”
- beim Binden wird automatisch überprüft:
  - Hauptversion.Client = Hauptversion.Server ?
  - Unterversion.Client ≤ Unterversion.Server (Aufwärtskompatibilität!) ?

# DCE-RPC: Semantik

- Semantik für den *Fehlerfall* ist wählbar:

(a) *at most once*

- bei temporär gestörter Kommunikation wird Aufruf automatisch wiederholt; eventuelle Aufrufduplikate werden gelöscht
- Fehlermeldung an Client bei permanentem Fehler
- ist default

(b) *idempotent*

- keine automatische Unterdrückung von Aufrufduplikaten
- Aufruf wird ein-, kein-, oder mehrmals ausgeführt
- effizienter als (a), aber nur für wiederholbare Dienste geeignet

(c) *maybe*

- wie (b), aber ohne Rückmeldung über Erfolg oder Fehlschlag
- noch effizienter, aber nur in speziellen Fällen anwendbar

- Optionale *Broadcast*-Semantik

- Nachricht wird in einem LAN an mehrere Server geschickt
- RPC ist beendet mit der ersten empfangenen Antwort