Verteilte Systeme II
WS 1998/99
Version 1.0.0
Achtung: unvollständig
Fehler im Skript bitte per Email melden:
Michael.Jaeger@mni.fh-giessen.de

Michael Jäger

28. Februar 1999

Inhaltsverzeichnis

1 Grundlegende Konzepte und Begriffe **4**

1.1 Client/Server-Architektur 4

1.1.1 Das *Concurrent Server*- Prinzip 4

1.1.2 Server-Adressierung und Ortstransparenz 4

1.1.3 Server-Suche bei ortstransparenten Netzadressen 5

1.1.4 Pakettypen für Client-Server-Kommunikation 6

1.1.5 Client-Server-Kommunikation in heterogenen Rechnerverbunden 6

1.2 RPC – Remote Procedure Call 6

1.2.1 RPC-Aufrufmechanismus 7

1.2.2 Probleme bei RPC 7

1.2.1 RPC-Semantik bei Fehlern 8

1.2.2 Verfügbare RPC-Systeme 9

1.3 Transaktionskonzept 9

1.4 Primärserver-Backupserver 11

2 Verteilte Dateisysteme **12**

2.1 Architektur verteilter Dateisysteme 12

2.1.1 Dateidienste 12

2.1.2 Katalogdienste 12

2.1.3 Semantik gemeinsamer Dateibenutzung 13

2.2 Implementierung verteilter Dateisysteme 13

2.2.1 Dateinutzungs-Charakteristika . . . 13
2.2.2 Systemstruktur . . . 14
2.2.3 Caching . . . 15
2.2.4 Replikation . . . 16

3 ONC - Open Network Computing 18
3.1 ONC-RPC . . . 18
3.1.1 Gegenüberstellung der RPC-Systeme ONC – NCA . . . 18
3.1.2 ONC-RPC-Merkmale . . . 19
3.1.3 Der Portmapper . . . 19
3.1.4 XDR – External Data Representation . . . 21
3.1.5 ONC-RPC-Beispiel . . . 23
3.1.6 ONC-RPC Nachrichtenformat . . . 29
3.2 NIS - Network Information System . . . 31
3.2.1 NIS-Maps . . . 32
3.2.2 NIS-Dämon-Prozesse . . . 32
3.2.3 Einige NIS-Kommandos . . . 32
3.2.4 NIS+ . . . 33
3.2.5 Der hierarchische Namensraum . . . 34
3.2.6 NIS+-Maps . . . 36
3.2.7 Update-Protokoll für Slave-Server . . . 37
3.2.8 Server-Suche: Verzeichnis-Cache und Kaltstartdatei . . . 38
3.2.9 Beispiele für die NIS+-Nutzung . . . 39
3.2.10 Befehlsübersicht . . . 39
3.2.11 nisswitch – Konfiguration der Dienste . . . 40

4 DCE - Distributed Computing Environment 41
4.1 DCE-Architektur . . . 41
4.2 Zellen . . . 42
4.3 DCE-Threads . . . 43
4.3.1 Thread-Scheduling . . . 43
4.3.2 Synchronisation von Threads . . . 43
4.3.3 Thread-Funktionen . . . 43
4.4 DCE - Remote Procedure Call . . . 44
4.4.1 Client-Server-Bindung . . . 45
4.4.2 DCE-RPC-Beispiel . . . 45
4.5 Zeit-Service . . . 49

4.5.1 Directory Service . . . 50
4.5.2 CDS – Cell Directory Service . . . 50
4.5.3 Zell-Beispiel . . . 52
4.5.4 X.500 . . . 52
4.5.5 Programmierschnittstelle zu X.500 . . . 53
4.5.6 Auflösung nichtlokaler Namen . . . 54
4.6 Security Service . . . 54
4.6.1 Kerberos . . . 54
4.6.2 Rollenspiele und Begriffe . . . 55
4.6.3 Tickets . . . 55
4.6.4 Weitere Merkmale . . . 55
4.7 DFS – Distributed File Service . . . 56
4.7.1 Directories, Filesets und Aggregate . . . 56
4.7.2 Episode . . . 56
4.7.3 Client-Caching . . . 56
4.8 DFS-Server . . . 57

5 SQL **58**
5.1 ISO-Spezifikationen für SQL . . . 58
5.1.1 SQL-89 . . . 58
5.1.2 SQL-92 . . . 58
5.1.3 SQL 3 . . . 59
5.2 SQL-Server . . . 59
5.3 Stored Procedures . . . 60
5.3.1 Triggers, Rules . . . 60
5.4 SQL-Middleware . . . 60
5.4.1 X/Open SAG CLI . . . 61
5.4.2 MS ODBC . . . 61
5.4.3 IBI-EDA SQL . . . 62
5.4.4 ISO/SAG RDA . . . 62
5.4.5 IBM DRDA . . . 62

6 CORBA **63**

1 Grundlegende Konzepte und Begriffe

1.1 Client/Server-Architektur

Kommunikation in verteilten Systemen basiert meist auf einer *Client-Server-Architektur*. Client und Server sind miteinander kommunizierende Prozesse. Im allgemeinen können Client und Server auf unterschiedlichen Rechnern ablaufen und über ein Netzwerk kommunizieren.

Die Begriffe definieren nur eine Rollenaufteilung im Hinblick auf die Kommunikation: Der Client sendet an den Server einen *Auftrag*, der Server bearbeitet diesen und schickt *Resultate* der Bearbeitung zurück. Ein Prozess kann selbstverständlich zu unterschiedlichen Zeitpunkten die Rolle eines Client und die eines Servers übernehmen.

Manchmal sind mit „Client“ und „Server“ die Rechner gemeint, auf denen die kommunizierenden Prozesse ablaufen.

1.1.1 Das *Concurrent Server*- Prinzip

Typischerweise wird der Server als paralleler (*concurrent*) Server agieren: Ein Thread des Servers, nennen wir ihn „Master-Thread“, wartet auf Aufträge. Diese werden über einen den Clients bekannten „Auftragseingangs“-Kommunikationsendpunkt übermittelt.

Bei Eingang eines Auftrags erzeugt der Master-Thread einen neuen „Slave-Thread“, der den Auftrag bearbeitet. Die Kommunikation zwischen „Slave-Thread“ und Client erfolgt über einen dynamisch erzeugten separaten Kommunikationskanal. Währenddessen kümmert sich der Master-Thread schon um den nächsten Auftrag.

1.1.2 Server-Adressierung und Ortstransparenz

Server-Prozesse werden in der Client-Applikation oft explizit adressiert. Dies betrifft die Adressierung des Server-Rechners genauso wie die Adressierung des Serverprozesses auf diesem Rechner.

Ein Beispiel für explizite Adressierung eines Rechners ist die Angabe seiner IP-Adresse, oder eines Namens, der eindeutig auf eine IP-Adresse abgebildet wird.

Ein Beispiel für explizite Adressierung eines Serverprozesses auf dem Serverrechner ist die Verwendung einer festen TCP-Portnummer auf der Anwendungsebene.

Solche Zieladressen widersprechen der Forderung nach **Ortstransparenz** der Dienste: *Ein Client-Programm muß nicht (und soll nicht) wissen, welcher Rechner und welcher Prozeß als Server für einen Dienst verfügbar ist.*

Explizite Adressen erschweren insbesondere

- die Migration (Verlegung) von Diensten

  (z.B. bei Überlastung eines Dateiservers die Übertragung eines Teils der exportierten Verzeichnisse auf einen zweiten Dateiserver)

- die Server-Replikation

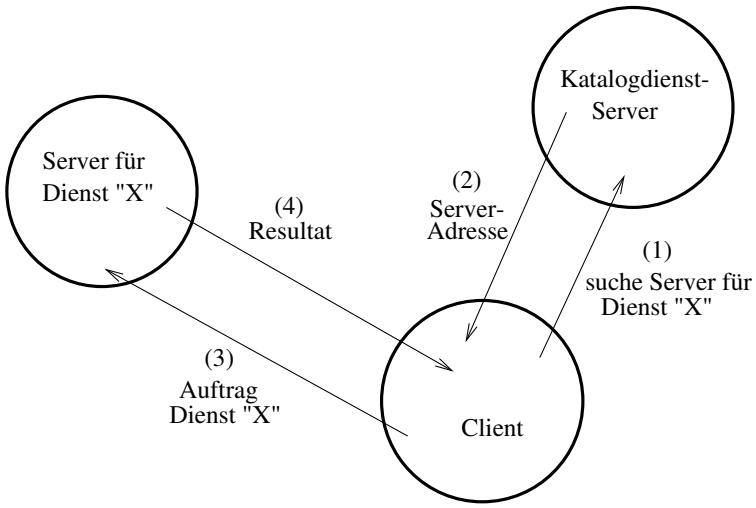
  (Bereitstellung alternativer Server für den gleichen Dienst, um eine Lastverteilung und/oder Ausfallsicherheit des Systems zu erreichen)

1.1.3 Server-Suche bei ortstransparenten Netzadressen

Bei Verzicht auf explizite Serveradressen verwendet ein Client folgende Techniken (meist kombiniert):

- Broadcast-Adressen zur Serversuche, z.B.
  - Ethernet-Broadcasts eines plattenlosen ARP-Client zur Bestimmung seiner IP-Adresse
  - IP-Broadcasts eines NIS-Client zur Serverbestimmung
  - Netbios-Broadcasts eines Windows-Client zur Bestimmung eines Browse-Servers
- Verzeichnisdienste, z.B. DCE, DNS, X.500 oder LDAP

Serververmittlung durch Katalogdienst



- Dynamische Registrierung von Kommunikationsendpunkten über „Portmapper“-Dienste (bei RPC)
- Caching bekannter Adressen auf Client-Seite

1.1.4 Pakettypen für Client-Server-Kommunikation

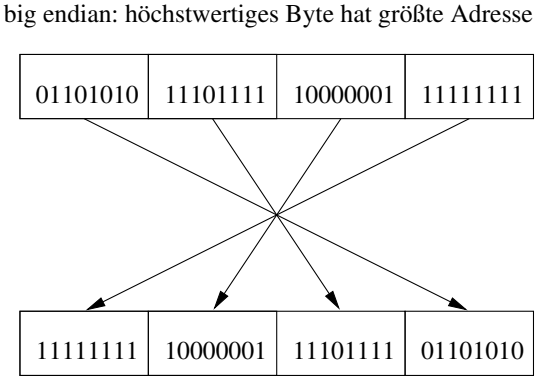
Betrachtet man die Client-Server-Kommunikation auf der Nachrichtenübertragungsebene, so kann man in den verwendeten Protokollen meist folgende Pakettypen wiederfinden:

Code	Pakettyp	Absender	Empfänger
REQ	Anforderung	Client	Server
REP	Antwort	Server	Client
ACK	Empfangsbestätigung	beliebig	beliebig
AYA	Are You Alive ?	Client	Server
IAA	I am Alive	Server	Client
TA	Try Again	Server	Client
AU	Address Unknown	Server	Client

1.1.5 Client-Server-Kommunikation in heterogenen Rechnerverbunden

In heterogenen Umgebungen ist ein erheblicher Aufwand nötig, um reibungslose Kommunikation auf der Applikationsebene zu ermöglichen.

Ein typisches Problem sind z.B. maschinenbedingte Darstellungsunterschiede für Integer-Datentypen („little endian“ / „big endian“). Ein „little endianServer (PC) wird einen vom „big endian“-Client (z.B. Sun SPARC) ohne weitere Vorkehrungen gesendeten 4-Byte-Integerwert falsch interpretieren, da die Byte-Reihenfolge umgekehrt ist. Konversionen sind notwendig.



Dabei ist es natürlich wünschenswert, die bei der Datenübertragung notwendigen Konversionen möglichst transparent für die Anwendungsebene in eine separate Kommunikationsschicht gekapselt werden. Sowohl die verbreiteten RPC-Systeme als auch die objektorientierten Verteilungsmechanismen (z.B. CORBA, DCOM, Tooltalk) enthalten solche Darstellungs-Anpassungsmechanismen.

1.2 RPC – Remote Procedure Call

„Remote Procedure Call“ oder „entfernter Funktionsaufruf“ ist ein Kommunikationsmechanismus, der auf Applikationsebene von der Nachrichtenübertragung abstrahiert. Stattdessen dient das

wohlbekannte Konzept des Unterprogrammaufrufs als Verteilungsmechanismus.

Grundprinzip: Ein Unterprogrammaufruf wird von einem entfernten Server bearbeitet, der die Argumente entgegennimmt, Resultate berechnet und zurückliefert.

Ziel: Die Verteilung soll transparent sein, so daß der Aufruf aussieht wie ein lokaler Unterprogramm-Aufruf.

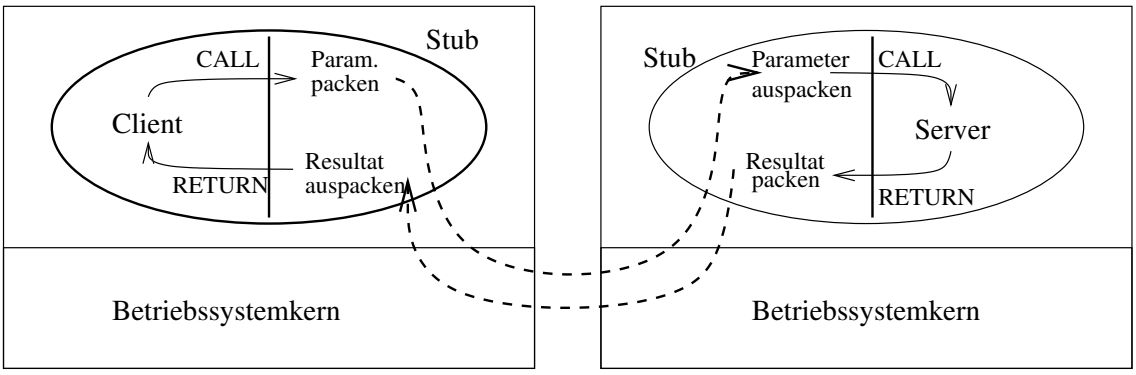
1.2.1 RPC-Aufrufmechanismus

RPC verwendet auf Client- und Serverseite sogenannte „Prozedurstümpfe“ („Stubs“), die für die Nachrichtenübertragung zuständig sind. Die Client-Stub-Funktion ist für die Versendung der Argumente zum Server und für die Entgegennahme der Resultate zuständig.

Auf der Serverseite sind zwei Aufgaben zu erledigen:

- Die Argumente müssen ausgepackt und ggf. konvertiert werden.
- Die Resultate müssen berechnet werden.
- Die Resultate müssen in Nachrichten verpackt und zurückgesendet werden. (vor dem Verpacken ggf. konvertiert)

Das Auspacken der Argumente, Konvertieren und Verpacken der Resultate wird dem „Server-Stub“ überlassen.



1.2.2 Probleme bei RPC

- Parameter- und Resultatsübergabe bei unterschiedlichen Adreßräumen (Referenzparameter, Zeiger)
- Netzwerk-Kommunikation
- Darstellungsunterschiede in inhomogenen Netzen

Darstellungsdifferenzen in heterogenen Netzen

- „little endian“ / „big endian“
- ASCII / EBCDIC
- Diverse float-Formate
- Zeichenketten
- usw.

Philosophien zur Lösung von Darstellungsunterschieden:

- Empfänger transformiert in seine Darstellung
- kanonische Darstellung

RPC-Prozedur-Stümpfe („stubs“)

Aufgabe:

- Parameter verpacken und versenden
- Resultate empfangen und auspacken

Automatische Generierung möglich (z.B. RPCgen)

Referenzparameter und Pointer

- verbieten
- Call-by-Reference durch Copy-Restore ersetzen
- Benutzerdefinierte Konversionsfunktionen verwenden
- Optimiertes Copy-Restore bei Einweg-Transport
- Datenanforderung bei Pointer-Dereferenzierung

1.2.1 RPC-Semantik bei Fehlern

- Client findet Server nicht
- Anforderung geht verloren
- Antwort geht verloren
  (Client weiß nicht warum, nicht alle Anfragen idempotent)
  Beispiel: Banküberweisung
- Server-Absturz
  Client weiß nicht, ob Server Anforderung bearbeitet hat

  - „mindestens einmal“-Semantik
  - „höchstens einmal“-Semantik
  - undefinierte Semantik
  - „genau einmal“-Semantik
- Client-Absturz
  - verwaiste Berechnungen
  - Exterminierung bei Reboot mittels Log-Eintrag
  - Reinkarnation (Definition neuer Epochen)
  - Sanfte Reinkarnation
  - Timeout

Protokolle für RPC

- verbindungslos / verbindungsorientiert
- Standard (z.B. TCP, UDP, IP) oder RPC spezifisch
- Flußkontrolle

1.2.2 Verfügbare RPC-Systeme

- viele inkompatible RPC-Mechanismen
- zwei System-übergreifend verbreitete RPC-Mechanismen: Sun-ONC und Apollo-NCA

Bestandteile eines RPC-Systems

- API für RPC-Aufrufe
- Bibliotheken
- Stub-Generator / RPC-Compiler
- Sprachdefinition für Datenbeschreibung
- Registraturdienst

Die ONC- und DCE-RPC-Systeme werden weiter unten anhand konkreter Beispiele vorgestellt.

1.3 Transaktionskonzept

Zweck: Atomarität komplexer Operationen als high-level Synchronisationskonzept
Beispiel:

- Computer-Banking im Netz, Atomarität einer Überweisung

- Flugreservierung bei mehreren Teilstrecken

Implementierung im verteilten System:

Private Arbeitskopien

- BEGIN-OF-TRANSACTION

  ein Prozeß benötigt eine private Kopie aller beteiligten Daten zur Modifikation (erst bei END-OF-TRANSACTION werden die modifizierten Daten für die *Außenwelt* sichtbar)

  Optimierungen notwendig, z.B. copy-on-write, shadow-blocks

- ABORT-TRANSACTION

  Löschen der privaten Transaktionsdaten

- END-OF-TRANSACTION

  Zurückschreiben der privaten Transaktionsdaten in das Dateisystem

Write-Ahead-Logging

- Vor jeder Modifikation wird ein Log-Record erstellt, der den alten und neuen Zustand eines Blocks und die anderen Transaktionsdaten enthält.
- Nach erfolgreichen Schreiben des Log-Record wird die Datei direkt modifiziert
- Eine COMMIT-Operation schreibt einen COMMIT-Record in die Log-Datei
- Eine ROLLBACK-Operation wird anhand der Log-Records in der umgekehrten Reihenfolge jede Operation der Transaktion rückgängig gemacht.
- Bei Systemabsturz kann die Log-Datei zum Wiederherstellen eines konsistenten Zustands verwendet werden

2-Phasen Commit-Protokoll

In verteilten Systemen sind die Daten einer Transaktion ggf. verteilt, so daß mehrere Prozesse an einer Transaktion beteiligt sind.

Ein Prozeß fungiert als Koordinator, die anderen sind untergeordnet.

Beispiel: Datenbank-Applikation als Koordinator, mehrere Server-Prozesse im verteilten DBMS untergeordnet.

Ablauf:

PHASE 1

1. Koordinator schreibt PREPARE-Log-Record
2. Koordinator schickt PREPARE-Nachricht an alle an der Transaktion beteiligten untergeordneten Prozesse

3. untergeordneter Prozeß schreibt READY-Log-Record
4. untergeordneter Prozeß schickt READY-Nachricht an Koordinator
5. Koordinator wartet auf alle READY-Nachrichten

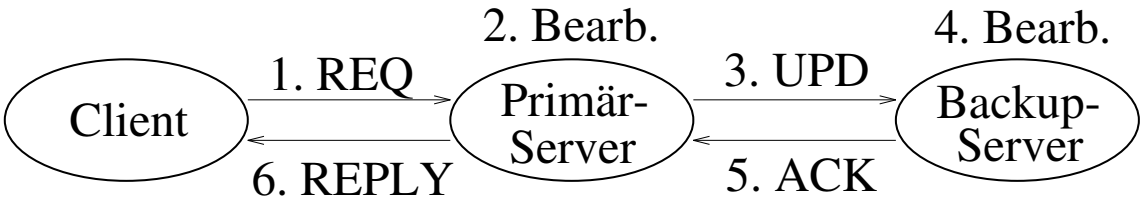
PHASE 1

1. Koordinator schreibt COMMIT-Log-Record
2. Koordinator schickt COMMIT-Nachricht an alle an der Transaktion beteiligten untergeordneten Prozesse
3. untergeordneter Prozeß schreibt COMMIT-Log-Record
4. untergeordneter Prozeß führt COMMIT-Operation aus
5. untergeordneter Prozeß schickt FINISHED-Nachricht an Koordinator

1.4 Primärserver-Backupserver

Prinzip: Ein Server ist hauptverantwortlich für alle Anfragen, Backup-Server übernehmen die Bearbeitung nur bei Fehlfunktion des Primärservers.

Beispiel:



2 Verteilte Dateisysteme

2.1 Architektur verteilter Dateisysteme

- Dateidienste
- Katalogdienste

2.1.1 Dateidienste

- „Upload/Download“-Modell (Dateiverarbeitung auf Client-Rechner)
  Schnittstellen-Funktionen:
  - Transfer einer Datei zum Client
  - Transfer einer Datei zum Server
- „Remote Access“-Modell (Dateiverarbeitung auf Server-Rechner)
  typische Schnittstellen-Funktionen:
  - Datei erzeugen, löschen
  - Lesen, Schreiben von Datensätzen
  - Attribute lesen u. modifizieren

2.1.2 Katalogdienste

Hierarchische Struktur durch Verzeichnisse

- Verzeichnisstruktur: Baum oder Graph ?
- gleiche Sicht der Hierarchie für alle Prozesse / Rechner ?
- globales Root-Verzeichnis ?

Transparenz

Namenskonventionen

- Server+Pfad
- Einbetten in lokale Hierarchie durch Montage
- globaler Namensraum, für alle Maschinen identisch

Dateiidentifikationsformen

- Maschineninterne Identifikation

- Netzweite interne Identifikation
- Symbolischer Name
- Symbolische Verweise
- Capability
- Listen von Identifikationen / Capabilities bei Replikation

Transparenz-Kriterien

- Ortstransparenz
- Ortsunabhängigkeit

2.1.3 Semantik gemeinsamer Dateibenutzung

- UNIX-Semantik
  - jede Modifikation ist sofort für alle sichtbar
  - absolute Zeit-Ordnung für alle Dateioperationen
  - einfach verstehbar
  - Probleme durch Client-Caching aus Effizienzgründen
- Sitzungs-Semantik
  - Sichtbarkeit von Änderungen durch Client erst bei Schließen der Datei
  - Bei konkurrierenden Modifikationen:
    - letzte Modifikation ist wirksam
    - irgendeine Modifikation ist wirksam
- nur unveränderbare Dateien
  - Dateien können nicht modifiziert werden
  - Ersatz durch neue Version möglich
  - Konkurrenz beim Ersetzen:
    - Strategien wie bei konkurrierenden Modifikationen
  - Ersetzen während ein anderer Prozeß die Datei liest
    - Strategie gemäß UNIX
    - Fehler bei nachfolgenden Lese-Operationen
- Transaktionskonzept

2.2 Implementierung verteilter Dateisysteme

2.2.1 Dateinutzungs-Charakteristika

Die Beurteilung eines verteilten Dateisystems hängt stark von typischen Nutzungsmustern ab.
Beispiel: Nutzungs-Eigenschaften in einem universitären UNIX-Sytem

- fast alle Dateien sind klein ($<$10KB)
- Lesen ist viel häufiger als Schreiben
- wahlfreier Zugriff ist selten
- Lebensdauer der Dateien ist kurz
- gemeinsame Nutzung ist selten
- Prozesse nutzen wenige Dateien
- verschiedene Dateiklassen mit unterschiedluchen Eigenschaften existieren

Beispiele für naheliegende Konsequenzen bei der Implementierung:

- Dateien komplett übertragen
- Sitzungssemantik
- Optimierungen für temporäre Dateien
- Unterschiedliche Replikationsstrategien für die Dateiklassen

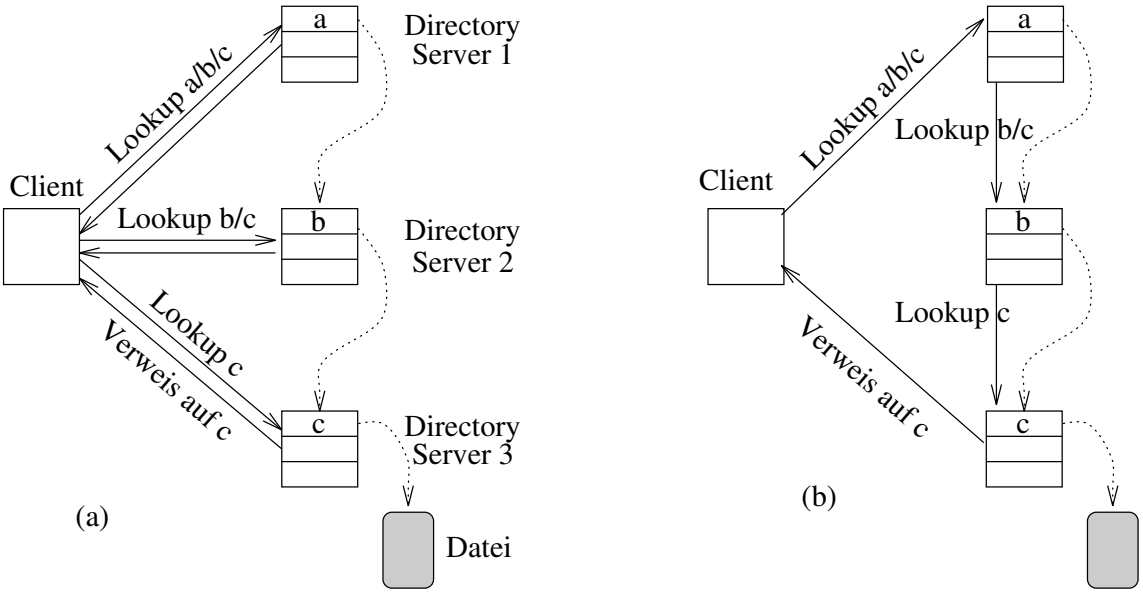
Aber: Ob und wieweit solche Aussagen auf andere Umgebungen und Systeme übertragen werden können, ist schwer zu sagen

2.2.2 Systemstruktur

Design-Alternativen

- Unterscheidung zwischen Client und Server-Software ?
- strikte Trennung von Dateidiensten und Katalogdiensten ?
- zustandslose Server ?

(a) iterative Pfadverfolgung für `a/b/c`
(b) automatische Pfadverfolgung für `a/b/c`



Zustandslose Server - Vorteile:

- Fehlertoleranz
- keine OPEN/CLOSE-Aufrufe
- keine Tabellen beim Server (Speicherersersparnis)
- beliebig viele offene Dateien bei Clients
- keine Probleme mit CLient-Abstürzen

Zustandsvolle Server - Vorteile:

- kürzere Anforderungen
- Effizienz besser
- Readahead möglich
- Idempotenz einfacher
- Datei Locking möglich

2.2.3 Caching

- Caching im Server-Hauptspeicher $\Longrightarrow$ Plattenzugriffe verringern

- Caching beim Client $\Longrightarrow$ Netzverkehr verringern
  - Prozeß-Adreßraum
    nur bei DBMS sinnvoll ?
  - Kernel-Adreßraum
    Systemcall-Overhead
  - separater Cache-Manager-Prozeß
    zusätzliche (lokale) RPCs
    flexibel
    ggf. Speicherverwaltungsprobleme

Zeitgewinn abhängig von CPU- und Netzwerkgeschwindigkeit

Konsistenzprobleme bei Client-Caching

Probleme:

- Wie behandelt man gleichzeitige Modifikationen einer Datei durch mehrere Clients bei Client-Cache-Verwendung ?
- Welchen Zustand einer gerade modifizierten Datei sieht ein zur selben Zeit lesender Prozeß ?

Lösungsansätze

- „write through“ zum Server mit zusätzlicher Prüfung der Cache-Konsistenz beim Lesen
- Sitzungssemantik
- verzögertes Zurückschreiben
- zentrale Kontrolle
- nicht änderbare Dateien

2.2.4 Replikation

Gründe für die Replikation von Dateien auf verschiedenen Servern

- Schutz vor Datenverlust
- ständige Verfügbarkeit der Daten
- Lastverteilung
  - Server-Belastung
  - Netzlast

Design-Alternativen

- Transparenz der Replikation ?
- Replikationstechnik:
  - explizit durch Client
  - verzögerte automatische Replikation mit Master-Server
  - simultane Gruppen-Replikation
- Update-Protokolle für existierende Repliken
  - Primärserver-Update
  - Server-Abstimmung bei Lesen und Schreiben
  - Server-Abstimmung mit Geister-Servern

3 ONC - Open Network Computing

- von Sun entwickelt
- besteht aus
  - ONC-RPC
  - NFS (Network File System) – verteiltes Dateisystem
  - NIS (Network Information System) – verteilter Katalogdienst
- inzwischen weiterentwickelt zu ONC+

3.1 ONC-RPC

3.1.1 Gegenüberstellung der RPC-Systeme ONC – NCA

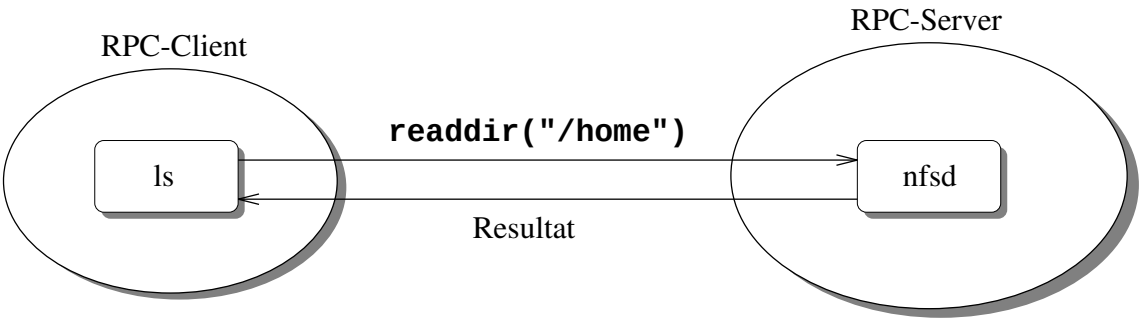
ONC-RPC (Open Network Computing) von Sun Microsystems

- Grundlage der ONC-Dienste: NFS, NIS/NIS+
- Datenbeschreibungssprache: XDR (External Data Representation)
- RPC-Compiler: rpcgen
- für alle UNIX-Systeme, Windows NT u.a. Systeme verfügbar
  (auch Novell *Netwise* basiert auf ONC-RPC)

NCA-RPC (Network Computing Architecture) von Apollo Computers

- Übernommen in das OSF DCE (Distributed Computing Environment) (Microsoft RPC entspricht DCE-RPC)
- Datenbeschreibungssprache: IDL (Interface Definition Language)

Beispiel: RPC-Einsatz bei NFS



3.1.2 ONC-RPC-Merkmale

- Identifikation eines RPC-Dienstes:
  - Programmnummer
  - Versionsnummer
  - Prozedurnummer
  - Nachrichtenübertragungsprotokoll
- RPC-Datenbasis
  - erlaubt die Zuordnung von Programmnamen
  - `/etc/rpc` oder *rpc*-NIS-Datenbasis
  - Auszug (Nummern hexadezimal):

```
nfs     100003  nfsprog
ypserv  100004  ypprog
mountd  100005  mount showmount
ypbind  100007
```

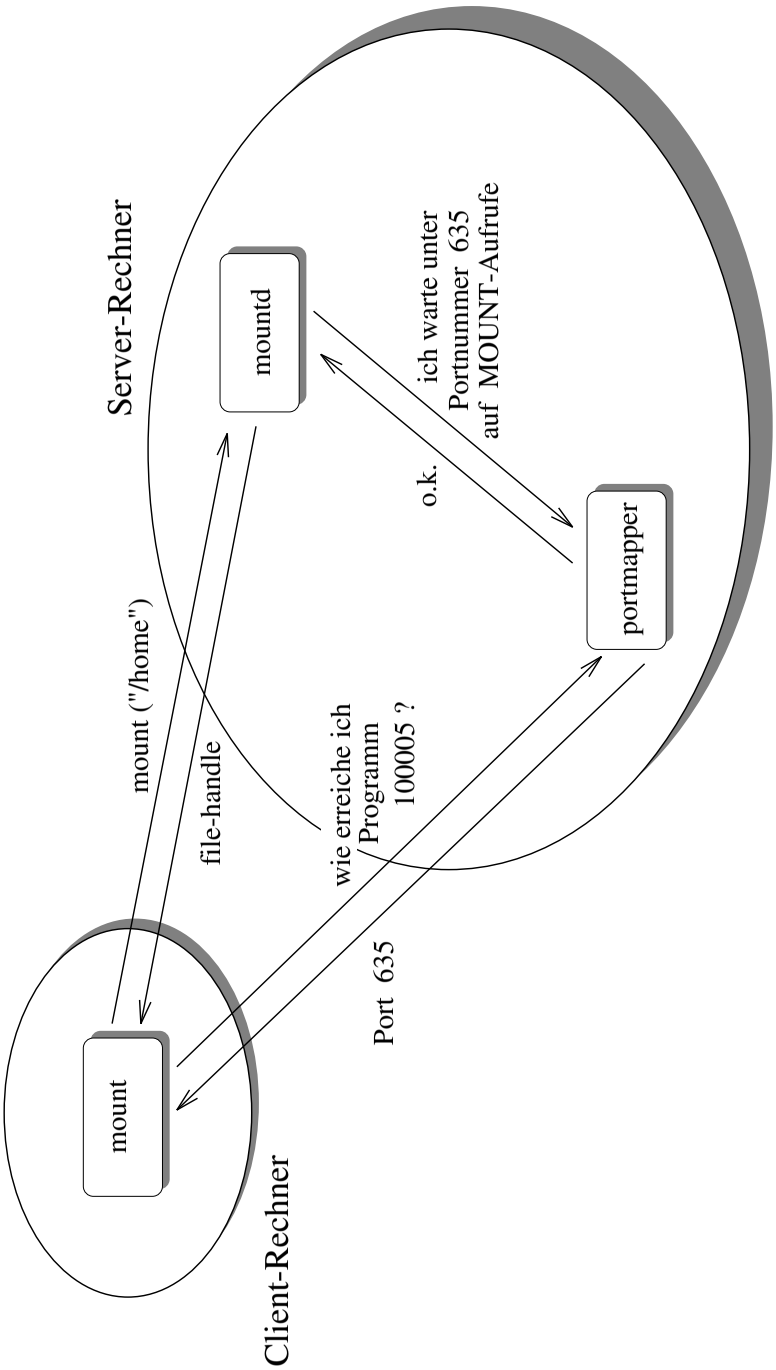
- *well-known services*: 0x000000 - 0x199999
- Wo stehen die Prozedurnummern ?

3.1.3 Der Portmapper

- Dämon leistet Registratur-Service für RPC-Dienste
- Kommando: *portmapper*, *rpc.portmap*, *rpc.bind* o.ä.
- RPC-Server:

  - initialisiert seine Kommunikationsports
  - registriert seine Prozeduren mit den zugehörigen Portnummern beim Portmapper
- RPC-Client:
  - ermittelt IP-Adresse des Server-Rechners (Host-Datenbasis)
  - ermittelt die Programmnummer des benötigten RPC-Programms (RPC-Datenbasis)
  - kontaktiert Portmapper des Server-Rechners (IP-Port 111)
  - fragt nach IP-Port für die Programmnummer
  - erhält IP-Port-Auskunft vom Portmapper
  - kontaktiert Server-Prozeß über diesen IP-Port

Beispiel:



3.1.4 XDR – External Data Representation

(vgl. RFC 1832)

- kanonische Datenrepräsentation
- implizite Typisierung (i.G. zu ASN 1)
- *rpcgen* als Stub-Compiler

Basistypen:

- `void`
- `bool`
- `[unsigned] [hyper] int`
- `float, double, quadruple`
- `opaque[], opaque<>`
- `string[], string<>`

Konstruktoren:

- Arrays: `[]` und <>
- Structures: struct
- Unions:

```
union switch (discriminant-declaration) {
case discriminant-value-A:

   arm-declaration-A;
case discriminant-value-B:
   arm-declaration-B;
...
default: default-declaration;
} identifier;
```

- Optionale Daten: `type-name *identifier;`

Beispiel aus RFC 1832:

```
const MAXUSERNAME = 32;     /* max length of a user name */
const MAXFILELEN = 65535;   /* max length of a file      */
const MAXNAMELEN = 255;     /* max length of a file name */

/*
 * Types of files:
 */
enum filekind {
   TEXT = 0,       /* ascii data */
   DATA = 1,       /* raw data   */
   EXEC = 2        /* executable */
};

/*
 * File information, per kind of file:
 */
union filetype switch (filekind kind) {
```

```
case TEXT:
   void;                              /* no extra information */
case DATA:
   string creator<MAXNAMELEN>;        /* data creator         */
case EXEC:
   string interpretor<MAXNAMELEN>; /* program interpretor  */
};

/*
 * A complete file:
 */
struct file {
   string filename<MAXNAMELEN>; /* name of file    */
   filetype type;               /* info about file */
   string owner<MAXUSERNAME>;   /* owner of file   */
   opaque data<MAXFILELEN>;     /* file data       */
};
```

Aus RFC 1832:

```
Suppose now that there is a user named "john" who wants to store his
lisp program "sillyprog" that contains just the data "(quit)".  His
file would be encoded as follows:

    OFFSET  HEX BYTES       ASCII    COMMENTS
    ------  ---------       -----    --------
     0      00 00 00 09     ....     -- length of filename = 9
     4      73 69 6c 6c     sill     -- filename characters
     8      79 70 72 6f     ypro     -- ... and more characters ...
    12      67 00 00 00     g...     -- ... and 3 zero-bytes of fill
    16      00 00 00 02     ....     -- filekind is EXEC = 2
    20      00 00 00 04     ....     -- length of interpretor = 4
    24      6c 69 73 70     lisp     -- interpretor characters
    28      00 00 00 04     ....     -- length of owner = 4
    32      6a 6f 68 6e     john     -- owner characters
    36      00 00 00 06     ....     -- length of file data = 6
    40      28 71 75 69     (qui     -- file data bytes ...
    44      74 29 00 00     t)..     -- ... and 2 zero-bytes of fill
```

3.1.5 ONC-RPC-Beispiel

Nachfolgend wird ein einfaches Beispiel für die Verwendung von ONC-RPC mit dem RPC-Compiler `rpcgen` vorgestellt.

Folgende Dateien gehören zum Beispiel:

`test5.x`	Schnittstellenspezifikation in „RPC-Language“ (XDR)
`test5_client.c`	generiert, Beispiel-Client zum Vervollständigen
`test5_clnt.c`	generiert, Client-Stub
`test5_server.c`	generiert, Beispiel-Server zum Vervollständigen
`test5_svc.c`	generiert, Server-Stub
`test5_xdr.c`	generiert, C-Datentypen und Konversionsroutinen

Die generierten Client- und Server-Rahmenprogramme sind vervollständigt:

- auf Client-Seite `test5_client.c` ist eine Beispiel-Liste konstruiert, die entfernte Längenfunktion aufgerufen und das Resultat ausgegeben. Der manuell eingefügte Quelltext ist gekennzeichnet.
- auf Server-Seite `test5_server.c` ist die Längenfunktion implementiert. Der manuell eingefügte Quelltext ist gekennzeichnet.

Man beachte, dass insgesamt nur wenige Zeile von Hand erstellt werden müssen.

`test5.x`

```
struct liste {
  int head;
  struct liste * tail;
};

program pointerprog {
  version v1 {
    int len ( liste l ) = 1;
    } = 1;
} = 20000001 ;
```

`test5_client.c`

```
/*
 * This is sample code generated by rpcgen.
 * These are only templates and you can use them
 * as a guideline for developing your own functions.
 */

#include "test5.h"


void
pointerprog_1( char* host )
{
        CLIENT *clnt;
        int  *result_1;
        liste  len_1_arg;
        liste l2;
```

```
        clnt = clnt_create(host, pointerprog, v1, "udp");
        if (clnt == NULL) {
                clnt_pcreateerror(host);
                exit(1);
        }


        /* hier beginnt der manuell erstellte Code */

        /* Beispiel-Liste erzeugen: */

        len_1_arg.head = 1;
        len_1_arg.tail = &l2;
        l2.head=2;
        l2.tail=(liste*) NULL;

        /* Stub-Aufruf: */

        result_1 = len_1(&len_1_arg, clnt);
        if (result_1 == NULL) {
                clnt_perror(clnt, "call failed:");
        }
        else
          printf("Client: Resultat=%d\n", *result_1);

        /* hier endet der manuell erstellte Code */


        clnt_destroy( clnt );
}


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

        if(argc < 2) {
                printf("usage: %s server_host\n", argv[0]);
                exit(1);
        }
        host = argv[1];
        pointerprog_1( host );
}
```

test5_clnt.c

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

#include <memory.h> /* for memset */
#include "test5.h"
```

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

int *
len_1(liste *argp, CLIENT *clnt)
{
        static int clnt_res;

        memset((char *)&clnt_res, 0, sizeof(clnt_res));
        if (clnt_call(clnt, len, xdr_liste, argp,
                      xdr_int, &clnt_res, TIMEOUT) != RPC_SUCCESS) {
                return (NULL);
        }
        return (&clnt_res);
}
```

test5_server.c

```
/*
 * This is sample code generated by rpcgen.
 * These are only templates and you can use them
 * as a guideline for developing your own functions.
 */

#include "test5.h"

int *
len_1_svc(liste *argp, struct svc_req *rqstp)
{

        static int  result=1;

        /*
         * insert server code here
         */

        /* hier beginnt der manuell erstellte Code */

        while(argp -> tail != (liste*) NULL) {
            argp = argp -> tail;
            result++;
        }

        /* hier endet der manuell erstellte Code */


        return(&result);
}
```

test5_svc.c

```
/*
```

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

#include "test5.h"
#include <stdio.h>
#include <stdlib.h>/* getenv, exit */
#include <rpc/pmap_clnt.h> /* for pmap_unset */
#include <string.h> /* strcmp */
#include <memory.h>
#include <sys/socket.h>
#include <netinet/in.h>

#ifdef __STDC__
#define SIG_PF void(*)(int)
#endif

static void
pointerprog_1(struct svc_req *rqstp, register SVCXPRT *transp)
{
        union {
                liste len_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 len:
                xdr_argument = (xdrproc_t) xdr_liste;
                xdr_result = (xdrproc_t) xdr_int;
                local = (char *(*)(char *, struct svc_req *)) len_1_svc;
                break;

        default:
                svcerr_noproc(transp);
                return;
        }
        (void) memset((char *)&argument, 0, sizeof (argument));
        if (!svc_getargs(transp, xdr_argument, (caddr_t) &argument)) {
                svcerr_decode(transp);
                return;
        }
        result = (*local)((char *)&argument, rqstp);
        if (result != NULL && !svc_sendreply(transp, xdr_result, result)) {
                svcerr_systemerr(transp);
        }
        if (!svc_freeargs(transp, xdr_argument, (caddr_t) &argument)) {
```

```
                fprintf(stderr, "unable to free arguments");
                exit(1);
        }
        return;
}

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

        (void) pmap_unset(pointerprog, v1);

        transp = svcudp_create(RPC_ANYSOCK);
        if (transp == NULL) {
                fprintf(stderr, "cannot create udp service.");
                exit(1);
        }
        if (!svc_register(transp, pointerprog, v1, pointerprog_1, IPPROTO_UDP)) {
                fprintf(stderr, "unable to register (pointerprog, v1, udp).");
                exit(1);
        }

        transp = svctcp_create(RPC_ANYSOCK, 0, 0);
        if (transp == NULL) {
                fprintf(stderr, "cannot create tcp service.");
                exit(1);
        }
        if (!svc_register(transp, pointerprog, v1, pointerprog_1, IPPROTO_TCP)) {
                fprintf(stderr, "unable to register (pointerprog, v1, tcp).");
                exit(1);
        }

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

test5_xdr.c

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

#include <rpc/types.h>
#include <rpc/xdr.h>

#include "test5.h"

bool_t
xdr_liste(XDR *xdrs, liste *objp)
{
```

```
          register long *buf;

          if (!xdr_int(xdrs, &objp->head)) {
                  return (FALSE);
          }
          if (!xdr_pointer(xdrs,
                           (char **)&objp->tail,
                           sizeof(liste),
                           (xdrproc_t)xdr_liste)) {
                  return (FALSE);
          }
         return (TRUE);
}
```

3.1.6 ONC-RPC Nachrichtenformat

Das Nachrichtenformat ist in RPC-Language definiert.

```
THE RPC MESSAGE PROTOCOL
------------------------

      enum msg_type {
         CALL  = 0,
         REPLY = 1
      };

      enum reply_stat {
         MSG_ACCEPTED = 0,
         MSG_DENIED   = 1
      };

      enum accept_stat {
         SUCCESS       = 0, /* RPC executed successfully             */
         PROG_UNAVAIL  = 1, /* remote hasn't exported program        */
         PROG_MISMATCH = 2, /* remote can't support version #        */
         PROC_UNAVAIL  = 3, /* program can't support procedure       */
         GARBAGE_ARGS  = 4, /* procedure can't decode params         */
         SYSTEM_ERR    = 5  /* errors like memory allocation failure */
      };

      enum reject_stat {
         RPC_MISMATCH = 0, /* RPC version number != 2          */
         AUTH_ERROR = 1    /* remote can't authenticate caller */
      };

      enum auth_stat {
         AUTH_OK           = 0,  /* success                          */

         /*
          * failed at remote end
          */
```

```
   AUTH_BADCRED      = 1,  /* bad credential (seal broken)     */
   AUTH_REJECTEDCRED = 2,  /* client must begin new session    */
   AUTH_BADVERF      = 3,  /* bad verifier (seal broken)       */
   AUTH_REJECTEDVERF = 4,  /* verifier expired or replayed     */
   AUTH_TOOWEAK      = 5,  /* rejected for security reasons    */
   /*
    * failed locally
    */
   AUTH_INVALIDRESP  = 6,  /* bogus response verifier          */
   AUTH_FAILED       = 7   /* reason unknown                   */
};

struct rpc_msg {
   unsigned int xid;
   union switch (msg_type mtype) {
   case CALL:
      call_body cbody;
   case REPLY:
      reply_body rbody;
   } body;
};

struct call_body {
   unsigned int rpcvers;       /* must be equal to two (2) */
   unsigned int prog;
   unsigned int vers;
   unsigned int proc;
   opaque_auth  cred;
   opaque_auth  verf;
   /* procedure specific parameters start here */
};

union reply_body switch (reply_stat stat) {
case MSG_ACCEPTED:
   accepted_reply areply;
case MSG_DENIED:
   rejected_reply rreply;
} reply;

struct accepted_reply {
   opaque_auth verf;
   union switch (accept_stat stat) {
   case SUCCESS:
      opaque results[0];
      /*
       * procedure-specific results start here
       */
    case PROG_MISMATCH:
       struct {
          unsigned int low;
          unsigned int high;
       } mismatch_info;
    default:
       /*
```

```
         * Void.  Cases include PROG_UNAVAIL, PROC_UNAVAIL,
         * GARBAGE_ARGS, and SYSTEM_ERR.
         */
        void;
    } reply_data;
};

union rejected_reply switch (reject_stat stat) {
case RPC_MISMATCH:
   struct {
      unsigned int low;
      unsigned int high;
   } mismatch_info;
case AUTH_ERROR:
   auth_stat stat;
};
```

3.2 NIS - Network Information System

- Bestandteil von Sun ONC
- Zweck: zentrale Pflege von Informationen, die im Netz benötigt werden
- früher: „Yellow Pages“, umbenannt in NIS
- Bestandteile:
  - C/C++-API
  - Kommandozeilen-Schnittstelle
  - modifizierte C-Bibliohek (z.B. *getpwnam*, *gethostbyaddr*)
  - Dämonen
  - Server-Client-Protokoll
  - Master-Server – Slave-Server-Protokoll
- Gültigkeitsbereich für NIS-Informationen: (NIS-) Domain
- Funktionsweise:
  - NIS-Server hält Informationen bereit
  - NIS-Client ruft Informationen bei Bedarf ab
- mehrere Server (Master-Slave-Prinzip)
  - Verfügbarkeit
  - Lastverteilung
- Inzwischen NIS+ verfügbar mit erheblich erweiterter Funktionalität (NIS für LANs, NIS+ für unternehmensweite Netzwerke)

3.2.1 NIS-Maps

- bestehen aus Schlüssel und zugeordnetem Wert
  (dbm-Datenbasen)
- werden mittels dbm aus ASCII-Quelltexten erzeugt (Makefile)
- werden vom Master-Server an die Slave-Server kopiert
- ersetzen oder ergänzen die Standard-Datenbasen (host, group, passwd, ...)
- Benutzerdefinierte Maps (z.B. Telefonlisten) sind (relativ einfach) integrierbar

einige Standard-Maps

hosts	Rechnernamen + IP-Nummern
passwd	Benutzerdatenbasis
group	UNIX-Benutzergruppen
rpc	RPC-Dienste
cred	Authentifizierungsschlüssel für Secure-RPC (nur NIS+)
netgroup	Gruppen von Rechnern oder Benutzern
auto_home	NFS-Automounter
auto_master	NFS-Automounter

3.2.2 NIS-Dämon-Prozesse

Programm	**Rechner**	**Aufgabe**
ypbind	NIS-Client	Serververmittlung
ypserv	NIS-Server	Informationsserver
yppasswdd	Master-Server	Update für Passwd-Map
ypupdated	Master-Server	Update für Passwd-Map

3.2.3 Einige NIS-Kommandos

ypbind	Client-Dämon, „An-/Ausschalter“ für NIS
ypserv	Server-Dämon
ypcat	NIS-Datenbasis anzeigen
ypwhich	Server-Status abfragen
yppush	Slave-Server aktualisieren

3.2.4 NIS+

Warum NIS+ :

- NIS ist auf lokale Netzwerke zugeschnitten
- Unternehmensweite Netzwerke sind LAN-übergreifend („Enterprise Networking“)
- zusätzliche Anforderungen
  - Dezentrale Verwaltung der Datenbasen (mehrere Administratoren an unterschiedlichen Orten mit unterschiedlichen Rechten)
  - hierarchische Domain-Struktur (spiegelt hierarchische Unternehmensstruktur wieder)
  - effiziente Map-Verteilung (LAN-übergreifend: teilweise Verbindungen mit geringer Bandbreite)
  - Zugriffskontrolle/Authentifizierung (LAN-übergreifend: teilweise Verbindungen mit geringer Sicherheit)

Vergleich NIS und NIS+

Funktionsmerkmal	**NIS**	**NIS+**
Namensraum	flach	hierarchisch
Datenbasis	ein Verzeichnis	hierarchisch durch Verzeichnisse strukturiert
logische Map-Struktur	mehrere 2-spaltige Relationen	eine mehrspaltige Relation, jede Spalte kann als Schlüssel verwendet werden
Map-Repräsentation	dbm-Map	komplexe Datenstruktur mit diversen Attributen
Replikation	ein Server pro Subnetz nötig	Server unterstützt mehrere Subnetze
Verwaltung	Superuser	jeder autorisierte Benutzer
Slave-Updates	ganze Maps	inkrementell
Zugriff	uneingeschränkt	feine Zugriffskontrolle
Zugriff auf andere Domains	nicht unterstützt	unterstützt
Pflege der Daten	Quelltextdatei	direkt in der Map

3.2.5 Der hierarchische Namensraum

Domain-Hierarchie

- Domain-Konzept – abstraktes Konzept, durch Verzeichnisse implementiert
- Domains können beliebig hierarchisch strukturiert werden
- Eine Domain definiert
  - Tabellen
  - NIS-Gruppen (für die Tabellen-Zugriffsrechte)

  - Unterdomains
- Domainnamen spiegeln die Domain-Hierarchie wieder:
  `systemtechnik.mni.giessen.fhgi.`

NIS+-Verzeichnisse

- Verzeichnisse strukturieren den Namensraum
- Ein Verzeichnis enthält Einträge für
  - Unterverzeichnisse
  - Tabellen
  - NIS-Gruppen
- Alle Namen sind hierarchisch gegliedert
- Eine Domain ist ein Verzeichnis mit folgenden Eigenschaften:
  - NIS+-Gruppen stehen im Unterverzeichnis namens `group_dir`
  - NIS+-Tabellen stehen im Unterverzeichnis namens `org_dir`
- Verzeichnis-Administration in Anlehnung an Dateiverzeichnisse
  (*nisls*, *nischmod*, *nismkdir*, ...)

NIS+-Namen

Definition (NIS+ *principal*)

Ein NIS+-Mandant (Principal) ist

- ein in den NIS+-Maps verzeichneter Benutzer oder
- der Superuser eines Rechners

- ein normaler Benutzer wird unter seinem Login-Namen geführt
  `jaeger.mni.fh-giessen.de`
- ein Superuser wird unter dem Rechnernamen verzeichnet:
  `sun30.mni.fh-giessen.de`

Problem:

Einem NIS+-Namen sieht man nicht an, welches Objekt (oder Subjekt) er repräsentiert:

`mni.giessen.fhgi.`	Domain (als Domain interpretiertes Verzeichnis)
`org_dir.mni.giessen.fhgi.`	Verzeichnis (keine Domain)
`jaeger.mni.fhgi.`	Benutzer – Mandant (principal)
`sun30.mni.fhgi.`	Rechner – Mandant
`admin.group_dir.giessen.fhgi`	NIS+-Gruppe

Für die verkürzte Namensangabe kann die Stammdomain weggelassen werden. Zusätzlich kann ein Suchpfad als Environment-Variable NIS_PATH definiert werden.

Beispiel: `hosts.org_dir`, `sun30`

3.2.6 NIS+-Maps

NIS+-Maps sind wesentlich komplexere Datenstrukturen als NIS-Maps

Beispiel:

```
$ niscat -o hosts.org_dir
Object Name       : hosts
Owner             : sun30.systemtechnik.mni.giessen.fhgi
Group             :
Domain            : org_dir.systemtechnik.mni.giessen.fhgi
Access Rights     : ----rmcdrmcdr---
Time to Live      : 12.0.0
Objekt Type       : Table
Number of Columns : 4
Character Separator :
Search Path       :
Columns           :
  [0]    Name           : cname
         Attributes     : (SEARCHABLE, TEXTUAL DATA, CASE INSENSITIVE)
         Access Rights  :
  [1]    Name           : name
         Attributes     : (SEARCHABLE, TEXTUAL DATA, CASE INSENSITIVE)
         Access Rights  :
```

```
[2]     Name            : addr
        Attributes      : (SEARCHABLE, TEXTUAL DATA, CASE INSENSITIVE)
        Access Rights   :
[3]     Name            : comment
        Attributes      : (TEXTUAL DATA)
        Access Rights   :
```

Zugriffsrechte

4 Benutzerklassen:

- *nobody* – unautorisierter Benutzer
- *owner* – autorisierter Eigentümer
- *group* – autorisiertes Mitglied der Gruppe
- *world* – autorisierter sonstiger Benutzer

Zugriffsrechte einer Klasse

- read
- modify
- create
- delete

Die Zugriffsrechte können für Spalten und Einträge einer Tabelle einzeln gesetzt werden.

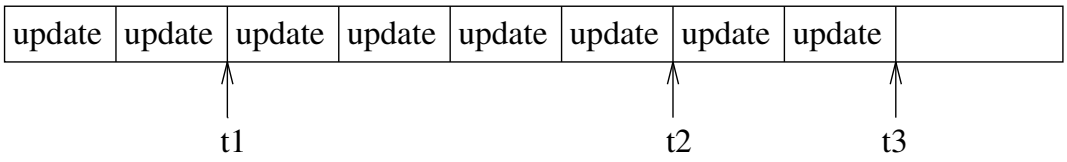
Beispiel:

```
$ nisls -l org_dir
T ----rmcdrmcdr--- sun30.mni. Mon Mar 12 13.44.23 1994 auto_master
T ----rmcdrmcdr--- sun30.mni. Mon Mar 12 13.46.12 1994 auto_home
T ----rmcdrmcdr--- sun30.mni. Mon Mar 12 13.56.53 1994 bootparams
T r---rmcdrmcdr--- sun30.mni. Mon Mar 12 15.25.12 1994 cred
...
```

3.2.7 Update-Protokoll für Slave-Server

- Master hält Originale aller Objekte (Verzeichnisse, Maps u.a.)
- Master legt modifizierte Objekte (komplette Verzeichnisse, Map-Einträge) verzögert (10 s) ab:

  - auf Platte und
  - in Transaktionsliste
- Transaktionsliste enthält Zeitstempel für jeden Update



- Master teilt Slave mit, daß Updates vorliegen
- Slave sendet Master den Zeitstempel seines letzten Update
- Master sendet Slave die seit dieser Zeit modifizierten Objekte
- Master löscht Transaktionsliste

Sonderfall:

Slave braucht (alte) Updates, die in der Transaktionsliste nicht mehr vorhanden sind.

→ Resynchronisation (alle Objekte werden vom Master zum Slave gesendet)

3.2.8 Server-Suche: Verzeichnis-Cache und Kaltstartdatei

- Client kann Information aus beliebigen Domains (Verzeichnissen) zugreifen
- jedes Verzeichnis wird durch (mindestens) einen Server bedient
- in jedem Verzeichnis sind die zuständigen Server vermerkt:
  - Netz-Namen
  - IP-Adressen und
  - Public Keys
- Verzeichnis-Information wird im Verzeichnis-Cache des Client gesammelt (`NIS_SHARED_DIRCACHE`, *nisshowcache*)
- Der Cache wird mit der *Kaltstartdatei* des Client (`NIS_COLDSTART`) initialisiert
- `NIS_COLDSTART` enthält i.d.R. Kopie des Client-Heimatverzeichnisses
- Steht ein Verzeichnis nicht im Cache, fragt Client seinen Stamm-Server
- Stamm-Server
  - schickt Client die Information aus eigenem Cache oder
  - sendet Client Kopie seines eigenen Heimatverzeichnisses
- ein Cache-Eintrag *verfällt* nach einem definierbaren (*nischttl*) Zeitpunkt

3.2.9 Beispiele für die NIS+-Nutzung

Listenverwaltung:
Liste erzeugen

```
nistbladm -c telefonliste Name=S Nummer=S friedberg.fhgi.
```

Eintrag anlegen

```
nistbladm -a Name=jaeger Nummer=520 telefonliste
```

oder

```
nistbladm -a [Name=jaeger,Nummer=520]telefonliste
```

Listeninhalt anzeigen

```
$  niscat -h telefonliste
#Name:Nummer
jaeger:520
letschert:523
...
```

Listenattribute anzeigen

```
$  niscat -o telefonliste
```

Info in Liste suchen

```
nismatch Nummer=520 telefonliste
nisgrep Name=[Mm][ae][iy]er telefonliste
```

3.2.10 Befehlsübersicht

- Map-Update
  - nisping
  - nislog
- Client-Verzeichnis-Cache
  - nischttl
  - nis_cachemgr
  - nis_showcache
- Listenverwaltung
  - nistbladm
  - nisaddent
    NIS+ - Maps aus ASCII-/NIS-Datenbasen erstellen und umgekehrt

    (replace, append, merge)
  - niscat

- Initialisierungsskripten
  - nispopulate
  - nisserver
  - nisclient
  - nissetup
- Verzeichnisverwaltung
  - nismkdir (auch Slave-Server-Anbindung)
  - nisln
  - nis

  ...
- Schlüssel: nisaddcred, nisupdkeys
- Gruppen: nisgrpadm

3.2.11 nisswitch – Konfiguration der Dienste

- Konfigurationsdatei: `/etc/nisswitch.conf`
- Informationsquelle pro Datenbasis einstellbar:
  - lokale Dateien (*files*)
  - NIS-Maps (*nis*)
  - NIS+-Maps (*nisplus*)
  - DNS (nur bei *hosts*) (*dns*)
  - Kompatibilitätsmodus (Dateien ergänzt durch NIS-Maps) (*compat*)
- 3 Default-Varianten `nisplus`, `nis`, `files`
- Format: *Datenbasis Quellen*

Status nach Abbarbeitung einer Datenbasis:

- SUCCESS
- UNAVAIL
- NOTFOUND
- TRYAGAIN

Abhängig vom Status kann die Suche abgebrochen oder fortgesetzt werden

Beispiel

```
hosts:   nisplus dns [NOTFOUND=return] files
passwd:  files nisplus
```

4 DCE - Distributed Computing Environment

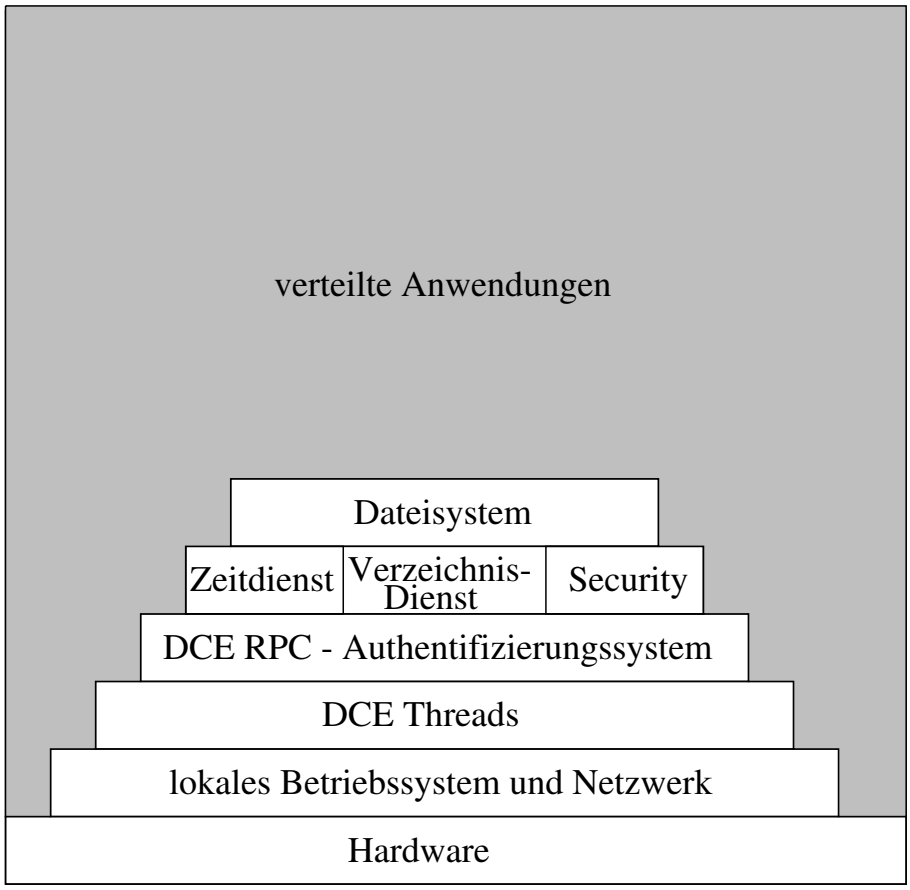
- Dienstleistungsschicht und Tools für verteilte Applikationen
- Client-Server-Modell (nicht objektorientiert)
- direkte Konkurrenz zu Suns ONC-Umgebung (ONC-RPC, NIS+, NFS usw.)
- vereinheitlicht und erweitert Dienste vorhandener Plattformen:
  - Betriebssysteme: UNIX, VMS, Windows NT, OS/2 u.a.
    teilweise nur Client-Funktionalität (z.B. Windows 3.1, OS/400, MVS)
  - Netzwerke: TCP/IP, OSI
  - Namensdienste: DNS, X.500
- Entwicklung: OSF (Open Software Foundation = HP, DEC, IBM u.a.)
- Stand: diverse Portierungen fertig, diverse fast fertig, aber erst seit kurzer Zeit ernsthafte Nutzung
- Source für („vanilla“) DCE frei verfügbar
- Info:
  - Web: `http://www.osf.org/dce/index.html`
  - News: `comp.soft-sys.dce`, `comp.client-server`
    (dce-faq auch monatlich in `comp.answers`)
  - Literatur: siehe dce-faq (deutsches Einführungsbuch inzwischen erhältlich)

4.1 DCE-Architektur

DCE besteht aus Programmierschnittstellen (Threads, RPC) und Tools. Die wichtigsten Komponenten:

- Thread-Bibliothek
- Remote Procedure Call
- Zugriffsschutzmechanismen (Security Service) – Kerberos-basiert
- Distributes File Service (DFS)
- Distributed Time Service (DTS)
- Directory Service

Schichtensicht von DCE



(vereinfacht - Schichten sind teilweise rekursiv voneinander abhängig)

4.2 Zellen

- Zelle = Gruppe von Benutzern, Rechnern und anderen Ressourcen
- Zelle = Administrationseinheit im Hinblick auf Namensräume und Zugriffsrechte
- effiziente Zell-lokale Kommunikationsprotokolle
- unterschiedlichen Kriterien zur Zell-Bildung, z.B.
  - gekoppelt an administrative Einheiten (z.B. Firma, Niederlassung, Abteilung)
  - funktionsbezogen (z.B. Verkauf, Produktion, Marketing)
  - produktbezogen
- Jede Zelle benötigt
  - time server
  - security server
  - directory server

(Eine DCE-Zelle entspricht in etwa einer NIS+-Domain)

4.3 DCE-Threads

Beispiel: UNIX-DCE-Implementierung

- POSIX-Threads als Basis
- User-Level-Implementierung oder Abbildung auf ggf. vorhandene Kern-Konzepte
- Zustände:
  - running
  - ready
  - waiting
  - terminated
- spezielle Signalbehandlung notwendig (synchrone / asynchrone Signale)
- spezielle Behandlung von Bibliotheksaufrufen, falls nicht „reentrant“

4.3.1 Thread-Scheduling

- Prioritätenbasiert
- 3 Varianten verfügbar:
  - FIFO für Threads der höchsten Priorität (ohne Verdrängung)
  - Round robin für Threads der höchsten Priorität mit festem Quantum
  - Round robin für alle Threads, Quantum abhängig von Priorität

4.3.2 Synchronisation von Threads

- **fast mutex** – entspricht *Lock* mit „test and set“, Versuch, Sperre zu setzen, führt zu Deadlock, falls schon gesetzt
- **recursive mutex** – entspricht Binär-Semaphor (mit Warteschlangenverwaltung)
- **nonrecursive mutex** – ohne Warteschlange, Versuch, Sperre zu setzen, führt zu Fehler, falls schon gesetzt
- **condition variable**

4.3.3 Thread-Funktionen

Beim Erzeugen von Objekten (Threads, Mutexe usw.) werden deren Eigenschaften durch Template-Strukturen spezifiziert, die als Parameter an die Erzeugungsfunktionen übergeben werden.

Übersicht über die wichtigsten Funktionen:

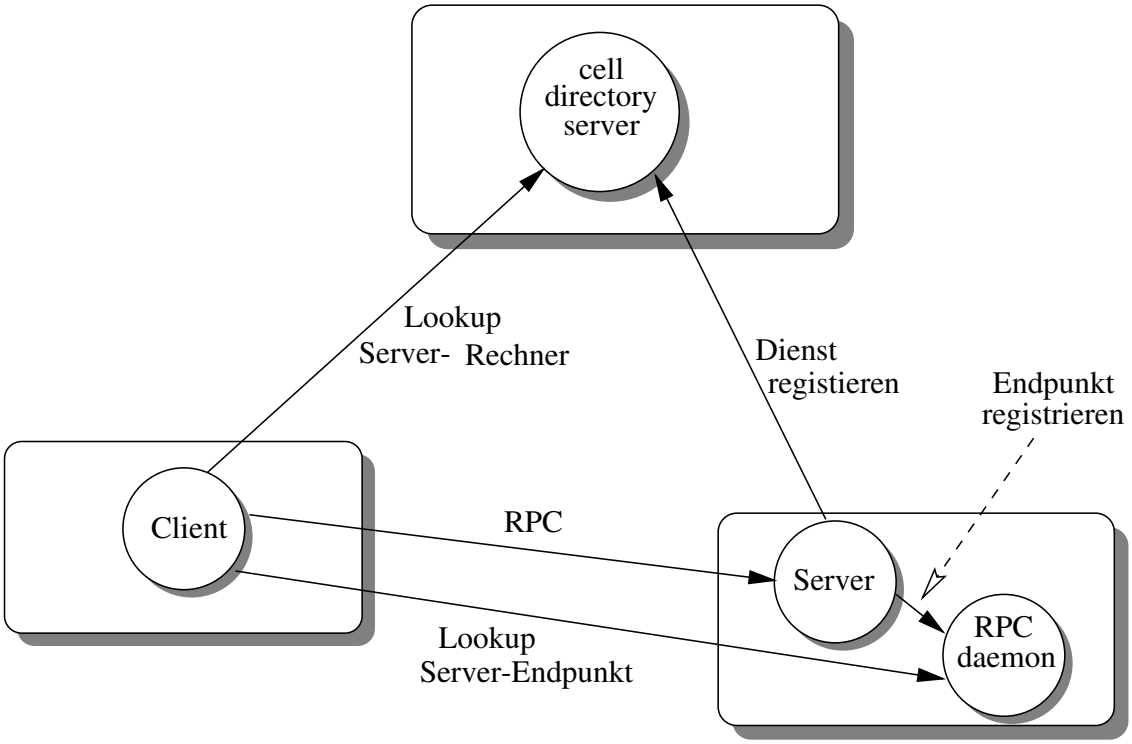
Thread-Verwaltung	
pthread_create	Subthread erzeugen
pthread_exit	beenden
pthread_join	auf Subthread warten
pthread_detach	Zombie-Thread vermeiden
Template-Verwaltung	
attr_create	Thread-Template erzeugen

attr_delete	Thread-Template löschen
attr_setprio	Priorität setzen
attr_getprio	Priorität lesen
attr_setstacksize	Stackgröße setzen
attr_getstacksize	Stackgröße lesen
attr_mutexattr_create	Mutex-Template erzeugen
attr_mutexattr_delete	Mutex-Template löschen
attr_mutexattr_setkind_np	Mutex-Typ setzen
attr_mutexattr_getkind_np	Mutex-Typ lesen
attr_condattr_create	Bedingungsvariablen-Template erzeugen
attr_condattr_delete	Bedingungsvariablen-Template löschen
Mutex-Verwaltung	
mutex_init	erzeugen
mutex_destroy	löschen
mutex_lock	sperren / warten
mutex_trylock	testen / sperren
mutex_unlock	freigeben
Verwaltung von Bedingungsvariablen	
cond_init	erzeugen
cond_destroy	löschen
cond_wait	warten
cond_signal	höchstens einen wartenden Thread wecken
cond_broadcast	alle wartenden Threads wecken
Verwaltung von globalen Thread-Variablen	
keycreate	erzeugen
setspecific	Wertzuweisung
getspecific	Wert lesen
Kill-Operationen	
cancel	Versuch, anderen Thread zu beenden
setcancel	Cancel-Aufrufe durch andere Thread (nicht) ignorieren
Scheduling-Operationen	
setscheduler	Algorithmus wählen
getscheduler	Algorithmus lesen
setprio	Priorität setzen
getprio	Priorität lesen

4.4 DCE - Remote Procedure Call

- Zielsetzung ungefähr wie bei ONC-RPC
- Schnittstellenbeschreibungssprache: IDL (nicht mit CORBA-IDL verwechseln)
- IDL-Compiler generiert aus IDL-Source Client- u. Server-Stubs
- Jede Schnittstelle wird durch einen eindeutigen Bezeichner identifiziert
- *uuidgen* generiert Bezeichner (Zeit und Ort wird hineinkodiert)
- Die Schnittstelle zum RPC-Mechanismus ist komplex (mehr als 100 Funktionen, mehr als 400 Seiten Manual)
- DCE-RPC wird in leicht modifizierter und nicht ganz vollständiger Form auch für Windows NT genutzt.

4.4.1 Client-Server-Bindung



DCE-Server sind „zustandsvolle“ Server

4.4.2 DCE-RPC-Beispiel

Zu jeder RPC-Applikation gehören

- eine IDL-Schnittstellenspezifikation (Dateiendung `.idl`)
- eine Konfigurationsdatei (Application configuration file – Dateiendung `.acf`)

Der IDL-Compiler *midl* gerneriert, ähnlich wie ONC *rpcgen*, die Stubs. Der Anwender muss die Server-Schnittstelle noch implementieren.

Nachfolgend sind die manuell erstellten Dateien zu einem einfachen Beispiel angegeben.

DISNEY.ACF

```
[implicit_handle(handle_t DisneyTour_IfHandle)]
interface DisneyTour
{

}
```

DISNEY.IDL

```
[ uuid (906B0CE0-C70B-1067-B317-00DD010662DA)]
interface DisneyTour
```

```
{
    void MagicMountainTour  ();
    void MickeyMouseSighting();
    void CloseDisneyTour([in, string] char * pszClientName);
}
```

SERVER.C

```
//////////////////////////////////////////////////////////////////
//                        Disney Tour Example
//
//    FILE:       server.c
//
//    PURPOSE:    DisneyTour Server
//
//    FUNCTIONS:  main() - Registers the Server and waits for client to call it
//
//    COMMENTS:
//
//////////////////////////////////////////////////////////////////

#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>
#include "disney.h"    // header file generated by MIDL compiler


void _CRTAPI1 main(int argc, char * argv[])
{
    RPC_STATUS status;
    unsigned char * pszProtocolSequence = "ncacn_np";
    unsigned char * pszSecurity         = NULL;
    unsigned char * pszEndpoint         = "\\pipe\\hello";
    unsigned int    cMinCalls           = 1;
    unsigned int    cMaxCalls           = 20;

    // Inform RPC runtime the protocol supported and the corresponding endpoint
    status = RpcServerUseProtseqEp(pszProtocolSequence,
                                   cMaxCalls,
                                   pszEndpoint,
                                   pszSecurity);  // Security descriptor
    printf("RpcServerUseProtseqEp returned 0x%x\n", status);
    if (status) {
        exit(status);
    }

    // Register the interface
    status = RpcServerRegisterIf(DisneyTour_ServerIfHandle,  // interface to register
                                 NULL,   // MgrTypeUuid
                                 NULL);  // MgrEpv; null means use default
    printf("RpcServerRegisterIf returned 0x%x\n", status);
    if (status) {
        exit(status);
    }
```

```
    //Beging servicing the Clients
    printf("Calling RpcServerListen\n");
    status = RpcServerListen(cMinCalls,
                             cMaxCalls,
                             FALSE);
    printf("RpcServerListen returned: 0x%x\n", status);
    if (status) {
        exit(status);
    }

}  // end main()

void MagicMountainTour()
{
    printf("Welcome to Magic Mountain.\n");
}

void MickeyMouseSighting()
{
    printf("Here is Mickey!\n");
}

void CloseDisneyTour(unsigned char * pszClientName)
{
    RPC_STATUS status;

    if (stricmp(pszClientName, "Walt Disney")==0)
    {
        printf("Calling RpcMgmtStopServerListening\n");
        status = RpcMgmtStopServerListening(NULL);
        printf("RpcMgmtStopServerListening returned: 0x%x\n", status);
        if (status) {
            exit(status);
        }

        printf("Calling RpcServerUnregisterIf\n");
        status = RpcServerUnregisterIf(NULL, NULL, FALSE);
        printf("RpcServerUnregisterIf returned 0x%x\n", status);
        if (status) {
            exit(status);
        }
    }
    else
        printf("User [%s] not authorized to close DisneyTour\n", pszClientName);
}
```

CLIENT.C

```
//////////////////////////////////////////////////////////////////
//                        Disney Tour Example
//
//    FILE:       client.c
//
```

```
//    PURPOSE:    DisneyTour Client
//
//    FUNCTIONS:  main() - binds to server and calls remote procedure
//
//    COMMENTS:   Calls functions in DisneyTour interface
//
////////////////////////////////////////////////////////////////////

#include <stdlib.h>
#include <stdio.h>
#include <ctype.h>
#include "disney.h"    // header file generated by MIDL compiler

void _CRTAPI1 main(int argc, char **argv)
{
    RPC_STATUS status;
    unsigned char * pszUuid             = NULL;
    unsigned char * pszProtocolSequence = "ncacn_np";
    unsigned char * pszNetworkAddress   = NULL;
    unsigned char * pszEndpoint         = "\\pipe\\hello";
    unsigned char * pszOptions          = NULL;
    unsigned char * pszStringBinding    = NULL;
    unsigned char * pszUserName         = NULL;
    unsigned long ulCode;
    int i;

    // See if the user wants to send a shutdown message
    for (i = 1; i < argc; i++)
    {
        if ((*argv[i] == '-') || (*argv[i] == '/'))
        {
            switch (tolower(*(argv[i]+1)))
            {
            case 's':  //
                pszUserName = argv[++i];
                break;
            }
        }
    }

    // Create binding string
    status = RpcStringBindingCompose(pszUuid,
                                     pszProtocolSequence,
                                     pszNetworkAddress,
                                     pszEndpoint,
                                     pszOptions,
                                     &pszStringBinding);
    printf("RpcStringBindingCompose returned 0x%x\n", status);
    printf("pszStringBinding = %s\n", pszStringBinding);
    if (status) {
        exit(status);
    }

    // Bind to the Server
```

```
    status = RpcBindingFromStringBinding(pszStringBinding,
                                         &DisneyTour_IfHandle);
    printf("RpcBindingFromStringBinding returned 0x%x\n", status);
    if (status) {
        exit(status);
    }

    printf("Calling the remote procedure 'MagicMountainTour'\n");

    RpcTryExcept
    {
        MagicMountainTour ( );
        if (pszUserName != NULL)
        {
            printf("Calling the remote procedure 'Shutdown'\n");
            CloseDisneyTour(pszUserName);  // shut down the server side
        }
    }
    RpcExcept(1)
    {
        ulCode = RpcExceptionCode();
        printf("Runtime reported exception 0x%lx = %ld\n", ulCode, ulCode);
    }
    RpcEndExcept

    // Disconnect from Server
    status = RpcStringFree(&pszStringBinding);  // remote calls done; unbind
    printf("RpcStringFree returned 0x%x\n", status);
    if (status) {
        exit(status);
    }

    // Release the binding string
    status = RpcBindingFree(&DisneyTour_IfHandle);  // remote calls done; unbind
    printf("RpcBindingFree returned 0x%x\n", status);
    if (status) {
        exit(status);
    }

    exit(0);

}  // end main()
```

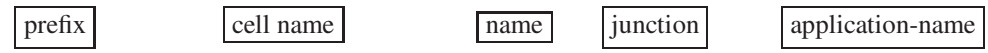
4.5 Zeit-Service

- DCE-Zeitverwaltung liefert die aktuelle Uhrzeit als Intervall
- Zeitzonen werden berücksichtigt
- 33 Bibliotheksfunktionen als Schnittstelle
- „time server"-Dämonen liefert aktuelles Intervall

- „time clerk“-Dämon aud Client-Rechnern korrigiert lokale Uhr nach Bedarf (Mittelwert der „glaubwürdigen“ Time-Server-Auskünfte)
- „time server“-Dämonenen gleichen ihre Uhrzeiten gegeneinander ab

4.5.1 Directory Service

- **Zweck:** zentrale Verwaltung von Ressourcen (Rechner, Benutzer, Zugriffsrechte, RPC-Schnittstellen usw.)
- zweistufig: Zell-intern / global
- Zell-intern speichert der **Cell Directory Service** (CDS) die Ressourcen der Zelle
- Als globale Namensdienste konkurrieren
  - der **Global Directory Service (GDS)** – (X.500)
  - der **Domain Name Service (DNS)**
- globale Anfragen (anders als bei ONC) werden von einem Dämonprozeß (GDA – Global Directory Agent) ausgeführt.
- DCE-Namen bestehen aus 5 Teilen:



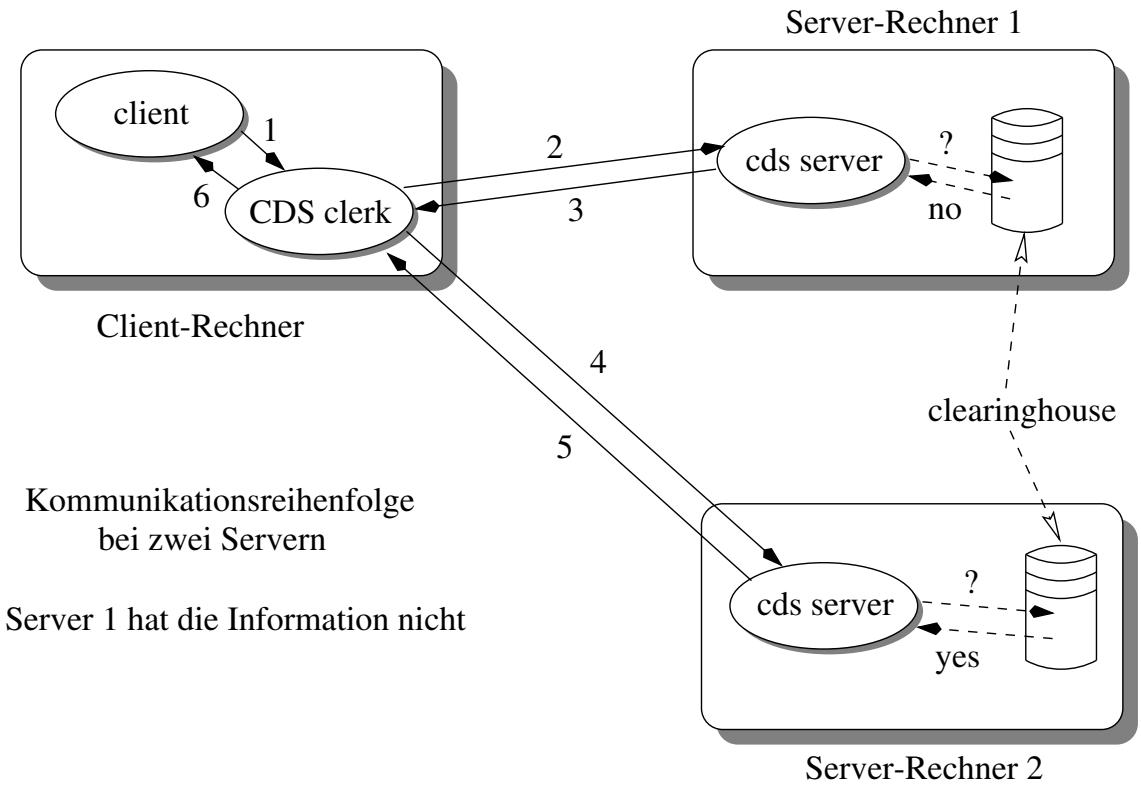
Beispiele:

```
/...        /C=US/O=IBM                  /sec    /principals/jaeger
/.:     /mni.fh-giessen.de  /subsys                    /file
```

  - globale Namen (`/...`) müssen eine Zelladresse beinhalten
  - lokale Namen (`/.:`) müssen keine Zelladresse beinhalten
- Zellen können in X.500- oder DNS-Notation adressiert werden

4.5.2 CDS – Cell Directory Service

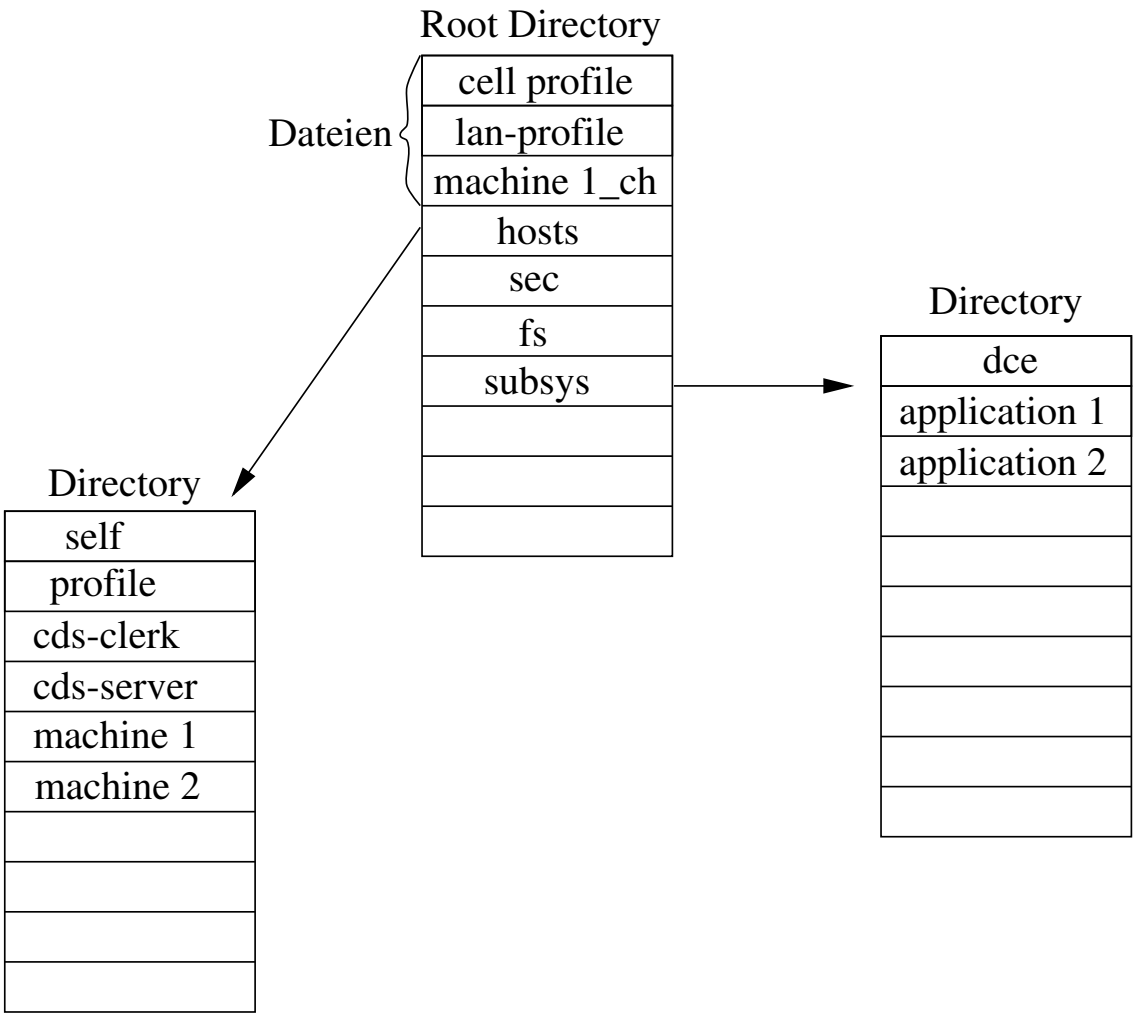
- Hierarchische Organisation der Namensräume mittels Directories (auch Soft-Links)
- Directories können repliziert werden (Master/Slave-Replikation)
- Directory steht innerhalb einer phys. Datenbasis („clearinghouse“)
- Information ist i.a. auf mehrere Server verteilt
- CDS-Server-Dämon nimmt Anfragen entgegen, schickt Antwort aus lokalem Clearinghouse zurück oder liefert Identifikation des zuständigen Servers
- CDS Server geben ihre Existenz durch Broadcast bekannt
- CDS-Clerk-Dämon läuft auf jedem Client-Rechner, übernimmt Caching der Daten
- Anfragen einer Applikation laufen immer über CDS-Clerk-Dämon



Kommunikationsreihenfolge
bei zwei Servern

Server 1 hat die Information nicht

4.5.3 Zell-Beispiel



4.5.4 X.500

- X.500 spezifiziert Datenbanken für Ressourcen gemäß ISO 9594
- Ressourcen-Beispiele: Staaten, Firmen, Personen, Rechner
- Bei der objektorientierten Definition einer Ressourcenklasse werden die Attribute für die Objekte dieser Klasse festgelegt
- Die Attribute können optional sein
- Die X.500-Notation listet Paare der Form
  Attributname=Wert auf, z.B.
  `/C=US/O=YALE/OU=CS/TITLE=PROF/OFFICE=210/SURNAME=LIN`
  Es müssen genügend Attribute zur eindeutigen Identifikation angegeben werden
- X.500 Namensräume sind hierarchisch organisiert
- zusätzlich gibt es Verweise (alias) auf andere Objekte

- Ein Baum wird durch ein aus drei Tabellen bestehendes Schema definiert:
  - Strukturregeltabelle - definiert die Hierarchie des Namensraums (Vorgänger-Nachfolger-Beziehung zwischen Objekten)
  - Klassentabelle - definiert die Vererbungsbeziehungen zwischen Klassen
  - Attributtabelle - definiert Namen und Typen der Attribute (in ASN.1)
  - Die Sichtbarkeit kann definiert werden als: PUBLIC, STANDARD, SENSITIVE
  - zusätzlich kann der Zugriff durch Objekt-spezifische ACLs (für jede Sichtbarkeitsklasse) kontrolliert werden
- In der Objekt-Hierarchie wird ein absoluter Pfad als DN (Distinguished name) und ein relativer als RDN (relative DN) bezeichnet

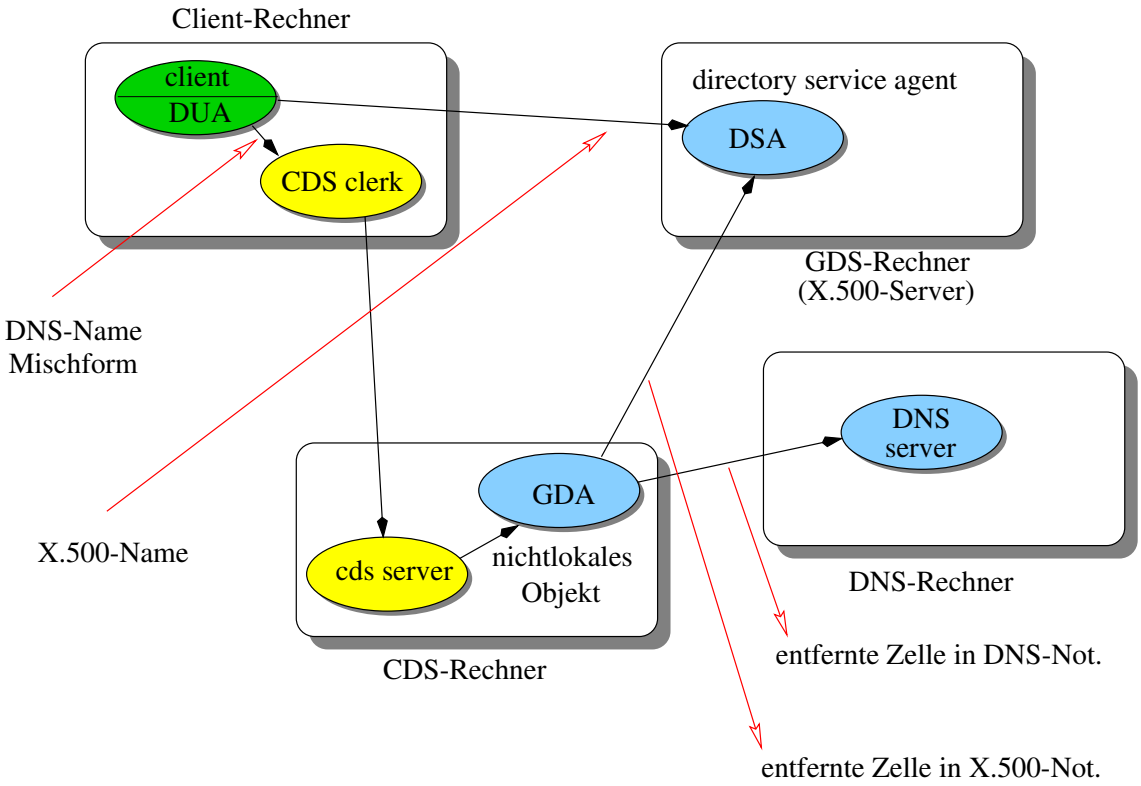
4.5.5 Programmierschnittstelle zu X.500

- Der Zugriff auf X.500-Datenbanken erfolgt vom DCE aus über die 13 Aufrufe der XDS-Bibliothek (XDS = X/Open Directory Service)

Wichtige XDS-Funktionen	
add_entry	Objekt oder Alias erzeugen
modify_entry	Objekt oder Alias modifizieren
remove_entry	Objekt oder Alias löschen
list	liefert Nachfolger in Objekthierarchie
read	liefert Attribute eines Objekts
compare	vergleicht einen Attributwert
modify_rdn	Umbenennen eines Objekts
search	Objekt-Hierarchie durchsuchen

- Daneben gibt es eine zweite API namens XOM (X/Open Objekt Management)

4.5.6 Auflösung nichtlokaler Namen



4.6 Security Service

- Ein *Mandant* (principal) ist ein Benutzer oder ein Rechner mit einer eindeutigen Identifikation (UUID – unique user identifier)
- *Authentifizierung* ist die Prüfung, ob ein Mandant sich korrekt identifiziert und nicht im Namen eines anderen auf eine Ressource zugreift
- Der Authentifizierungsmechanismus benutzt Kerberos vom MIT
- Kerberos basiert auf traditionellen kryptographischen Methoden (im Gegensatz zu ONC, wo „public key“-Verfahren verwendet werden)
- *Authorisierung* ist die Berechtigung eines Mandanten, auf eine Ressource zuzugreifen
- Der Authorisierungsmechanismus benutzt ACLs

4.6.1 Kerberos

- Verteiltes Verschlüsselungs- und Authentifizierungssytem (MIT-Projekt „Athena“)
- Betriebssystem-unabhängig
- basiert auf DES (Data Encryption Standard)
- Kerberos ist ein Server, der Kryptographiedienste für unterschiedliche Verwendungsszwecke anbietet
- aktuelle Version 5 ist in RFC 1510 spezifiziert

4.6.2 Rollenspiele und Begriffe

Kerberos sichert die Kommunikation zwischen zwei beliebigen Rechnern, die wir „Client“ (C) und „Server“ (S) nennen.

- Das KDC („key destribution center“) fungiert als vertrauenswürdige 3. Partei, die für Client und Server geheime Schlüssel verwaltet
- Das KDC besteht aus:
  - Authentifizierungsserver (AS)
  - „Ticket Granting Server“ (TGS)
  - Kerberos-Administrationsserver (KADM)

  (unterschiedliche Kommunikationsprotokolle)
- AS liefert dem Client (beim Login) ein Ticket zur Kommunikation mit dem TGS. Das Ticket kann nur mit dem Client-Passwort entschlüsselt werden.
- TGS liefert C auf Anfrage ein Ticket für S und einen Kommunikationsschlüssel (Sitzungsschlüssel).
- C sendet S das Ticket. S extrahiert aus dem Ticket den Kommunikationsschlüssel. C und S kommunizieren.

4.6.3 Tickets

Kommunikationstickets enthalten

- Servername
- Clientname
- Client IP-Nummmer
- Aktuelle Zeit
- Lebensdauer des „Ticket“
- kurzlebigen Kommunikationsschlüssel (Sitzungsschlüssel)

Die Kommunikationstickets werden mit dem Server-Schlüssel verschlüsselt (Client kann Ticket nicht manipulieren).

Client-Authentifizierung: Client-Identität und IP-Adresse sind Bestandteil des Ticket. Der Client kann das Ticket nicht modifizieren.

Server-Authentifizierung: Der Sitzungsschlüssel ist Bestandteil des Ticket. Da das Ticket mit dem Server-Schlüssel verschlüsselt ist, kann nur der Server den Sitzungsschlüssel extrahieren. Wenn der Client eine mit dem Sitzungsschlüssel verschlüsselte Nachricht erhält muss diese also vom Server stammen.

Nachrichten können verschlüsselt oder als Klartext mit **Authentizitätsnachweis** verschickt werden. Der Authentizitätsnachweis enthält verschlüsselt: Absender, MD5-Prüfsumme der Nachricht und Zeitstempel.

4.6.4 Weitere Merkmale

- hierarchiche Kerberos Domänen („Realms“)
- erneuerbare Tickets
- vordatierte Tickets
- proxy tickets
- forwardable tickets

4.7 DFS – Distributed File Service

- DCE-Rechner verwalten ihre Dateien lokal mit Episode (UNIX-Reimplementierung mit Verbesserungen) oder nativem Dateisystem
- Zugriff auf entfernte Dateien erfolgt auf Basis der anderen DCE-Dienste über die DFS-Dienste
- Identifikation von Dateien durch weltweit eindeutige Pfade, z.B.
  `/.../mni.fh-giessen.de/home/hg52` (Internet-Format)
  `/.../C=DE/O=FH-GIESSEN/OU=MNI/fs/home/hg52` (X.500-Format)
- File-Server werden über den Directory Service verwaltet
- Zugriffsrechte werden über ACLs kontrolliert (Nutzung des Security Service)
- Dateizugriff erfolgt über DCE-RPC ggf. mit erhöhter Sicherheit

4.7.1 Directories, Filesets und Aggregate

- Dateien können in herkömmlicher Weise mittels **Directories** hierarchisch organisiert werden
- Ein **Aggregat** entspricht einem „logischen Gerät“ in UNIX, d.h. i.d.R. einem Dateisystem auf einer Plattenpartition
- Ein **Fileset** ist eine Subhierarchie eines Aggregats
- Filesets lassen sich „clonen“ und replizieren (Master/Slave)
- Filesets sind die Einheiten der Datensicherung
- Filesets werden dynamisch an einen Server gebunden, Verlagerungen bleiben den Nutzern gegenüber transparent

4.7.2 Episode

- unterstützt effizientes „Cloning“ von Filesets mit copy-on-write für die Daten und kopieren der Verwaltungsstruktur
- benutzt a-node-Abstraktion zur Verwaltung von Dateien, verzeichnissen, Filesets, ACLs usw.
- nutzt Log-Files für Absturz-Recovery
- liefert exakte UNIX-Semantik bei gleichzeitigem Dateizugriff über Token-Mechanismus

4.7.3 Client-Caching

- Client benötigt zum Öffnen/Schreiben/Lesen/Sperren usw. jeweils spezifische Tokens
- Token werden über den Tokenmanager (Client-Rechner) vom Server bei Bedarf angefordert und beim Client aufbewahrt
- File-Server führt Buch über Tokens
- Bei konkurrierendem Zugriff fordert File-Server vom Tokeninhaber Token zurück und liefert es an anderen Client aus
- Read/Write-Zugriff erfolgt in 64KB-Einheiten (auch für plattenlose Rechner geeignet)
- Token verfallen nach 2 Minuten (wenig Aufwand bei Client-Absturz)

4.8 DFS-Server

- Fileset Server – Fileset-Operationen
- Fileset Location Server – verwaltet Fileset-Server Bindungen
- Replication Server – sorgt für Konsistenz der Repliken
- Basic Overseer Server – kontrolliert Funktionsfähigkeit der anderen Server

5 SQL

SQL = „structured query language“
Einsatzmöglichkeiten:

- interaktive Abfragesprache
- Programmiersprache
  - embedded
  - X/open API
  - spezielle SQL-Sprachen
- Datendefinition und Administration
- Netzwerkschnittstelle auch zu nicht-rel. DBMS

5.1 ISO-Spezifikationen für SQL

5.1.1 SQL-89

(herkömmliches SQL)

5.1.2 SQL-92

Merkmale:

- Client-Server
  - Agenten
  - CONNECT / SET CONNECTION
- feinere Transaktionskontrolle
- Embedded SQL für C, Ada, MUMPS
- dynamisches SQL
- BLOBs u.a. neue Datentypen (TIME, TIMESTAMP, VARCHAR)
- Constraints (Acceptable values)
- versch. Sorten von Joins

Statisches / Dynamisches SQL: Geschwindigkeit gegen Flexibilität

5.1.3 SQL 3

Aktueller Standard, Spezifikation > 1000 Seiten !

Merkmale

- Triggers, Rules
- Roles (Gruppenkonzept f. Zugriffsrechtsverwaltung)
- Benutzerdefinierte ADTs mit Vererbung
- Rekursive Queries
- CLI Call Level Interface
- SQL PSM (Persistent Storage Module)
  „stored procedures“ + prozedurale Erweiterungen
- SSQL/Transactions- Teilnahme an globalen Transaktionen
- SSQL/Temporal- Zeitserien
  („Wie waren die Umsätze 1990 ?“)
- SQL/MM - Multimedia - ADTs

5.2 SQL-Server

Funktion:

- Kontrolle und Ausführung von SQL-Kommandos
- macht SIchten auf den Datenbestand verfügbar
- erstellt optimierten Zugriffsplan
- hat meist auch Administrations-Schnittstelle
- verwaltet dynamische Verzeichnisse, die die DB beschreiben

Standard-Konformität:

- meist SQL-92 verfügbar + einige Konzepte aus SQL 3,
- teilweise Hersteller-spezifische Varianten

Ziele der Hersteller:

- reine DBMS-Hersteller wollen ihre Server zur „eierlegenden Wollmilchsau“ erweitern (Sybase, Oracle, Informix, Ingres)
- Hersteller mit diversifizierter Angebotspalette halten sich eher an die Standards

Server-Architekturen:

- Process-per-Client (sicher, starker Ressourcenverbrauch) (DB2/2, Informix, Oracle 6)
- Multithreaded (schnell, portabel, schlechte Scheduler) (Sybase, SQL-Server)
- Hybrid (Oracle7)

vgl. Abbildungen 10-2, 10-3, 10-4

5.3 Stored Procedures

vgl Abb.: 10-5, S. 163

Erstmalig: Sybase 1986

RPC-artig aufrufbare, in der DB gespeicherte Prozeduren in erweitertem SQL programmiert. Die klassische Trennung von Daten und Applikationslogik wird aufgehoben!

- gegenüber Client-seitiger Ausführung der Anweisungen effizienter
- ADT-Vorteile
- Verteilbarkeit für Applikationslogik steuerbar, Applikationslogik kann vom Client zum Server verschoben werden

Nachteile gespeicherter Prozeduren:

- Verknüpfung mit Transaktionskonzept unflexibel: jede SP ist eine Transaktion
- Hauptnachteil: Meist Non-standard, vgl. mit RPC, vor allem Parameterübergabe
  Bsp.:
  - teilw. keine Cursor-Unterstützung
  - teilw. nur eine Zeile
  - teilw. proprietäre SQL-ERweiterungen, teilw. klassische Programmiersprachen
  - teilw. Speichern in der DB, teilw. ausserhalb
  - teilw. nicht nebenläufig aktivierbar

vgl. Tabelle 10-2, [ECSSG, S. 165]

5.3.1 Triggers, Rules

vgl. Bild 10-6, [ECSSG, S. 168]

Hersteller-Unterschiede:

- Anzahl Trigger pro SQL-Operation/Table unterschiedlich beschränkt
- before/after-statement-Trigger
- Trigger per Row/Trigger per Command
- Zeit-gesteuerte Triggers
- Reihenfolge bei mehreren Trigers

5.4 SQL-Middleware

Problem: Kommunikation in heterogenen Systemen

hier: heterogene DB-Landschaften (Client- + Server-Seite)

Was bekommt man von einem einzigen DBMS-Hersteller?

- API für mehrere Betriebssysteme, SQL89-Standard mit herstellerspezifischen Erweiterungen
- ESQL und/oder CLI
- Hersteller-spezifische FAP (SQL-Treiber)

- FAP-Unterstützung für mehrere Transportstacks
- Gateways für DB2 (eingeschränkt nutzbar fü dyn. SQL-Anfragen ohne verteilte Transaktionen)
- Remote Administrations-Tools
- Entwicklungstools und GUI-Query-Tools

vgl. Bild 11-1 [ECSSG S. 173]
Kommunikation zwischen DBMSen verschiedener Hersteller: N Hersteller $\rightarrow N^2$-Problem:

- untersch. APIs
- untersch. FAPs
- unterschiedl. Transport-Unterstützung
- unterschiedl. Admin. Tools

vgl. Abb. 11-3, [ECSSG S. 177]
1. Verbesserung: gemeinsame API
vgl. Abb. 11-4, [ECSSG S. 178]
Ansätze dazu: Embedded SQL, X/Open SAG CLI, ODBC EDA/SQL API
2. Verbesserung

- Ziel: gemeinsames SQL-Gateway = API + FAP + Stacks
- Aber: DBMS-Admin immer noch spezifisch
- Ansätze dazu: ISO/SAG RDA, IBM DRDA, EDA/SQL

3. Verbesserung: Idealzustand

- einheitliche DBMS-Server-Schnittstelle
  (Wollen nur die Anwender !)
- Ansätze für Vereinheitlichung der Administration:
  MIB-Standard für SNMP-basierte Verwaltung, IBM-Data-Hub, ...

5.4.1 X/Open SAG CLI

- 1988 von 44 DB-Herstellern gegründet: SQL Access Group
- Ziel: offener STandar für FAP + CLI
- 1994 CLI fertig, an X/Open übergeben
- dynamisches SQL, in etwa SQL89-Umfang $\rightarrow$ langsam

5.4.2 MS ODBC

- von SAG CLI abgeleitet, erweitert,
- derzeit de facto Standard für gemeinsame API
- API wird von anderen Herstellern als Zweit-API unterstützt
- Hersteller erweitern die API oft, MS z.B. für Applikationsunterstützung

- ungewisse Zukunft, ändert sich ständig, nicht gut dokumentiert
- komplizierte Treiber
- langsam, ca. 10 % eines nativen Treibers
- Treiber sind gebündelt mit DBMS oder Applikationen, die es benutzen
- unterstützt auch nur dynamisches SQL

5.4.3 IBI-EDA SQL

- IBI Information Builders Inc.
- EDA Enterprise Data Acces
- Bestandteile:
  - CLI mit asynchronen Aufrufen
  - RPC für TP-Monitor-Kontrolle oder sonstiges
- Clients für DOS, Windows, NT, SOlaris, HPUX, OS/2, OS400, OS/390, MACOS, ...
- EDA-Anbindung ausgewählter Applikationen (z.B. Lotus 1-2-3)
- Anbindung an 12 Protocoll Stacks
- Authentifizoerungsmechnismen
- Server (kein DBMS-Server !) transformiert auf der DBMS-Serverseite Client-Requests in DBMS-Server-Requests
- für jedes bekannte DBMS (> 50)

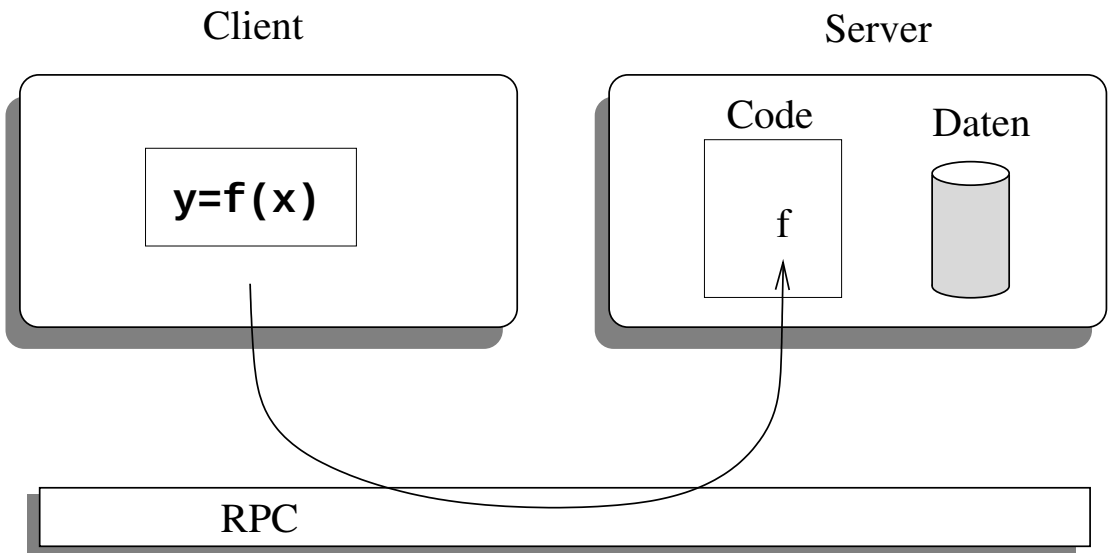
5.4.4 ISO/SAG RDA

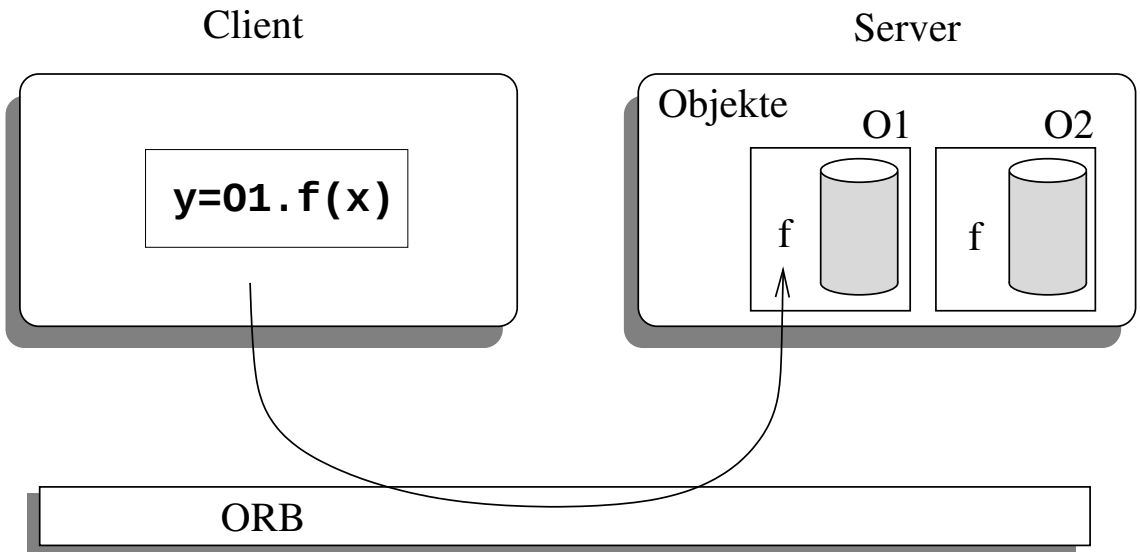
- OSI-Stack basierter Datenzugriffsstandard
- dyn. SQL
- async. Reuqests
- Nachrichtenformat mit expliziter Syntax pro Feld
- nicht akzeptiert

5.4.5 IBM DRDA

- Distributed Relational Database Architecture
- Clie/Server-Protokoll
- SERVER/SERVER-Protokoll
- Distributed Units of Work (verteilter Zugriff, aber jede Operation betrifft einen Server)
- Distributed ACCESS (inklusive Multi-Site Join u. ä.)
- Statisches + Dynamisches SQL unterstützt

6 CORBA


Client
Server
y=f(x)
Code
Daten
f
RPC


Client
Server
y=O1.f(x)
Objekte
O1
O2
f
f
ORB