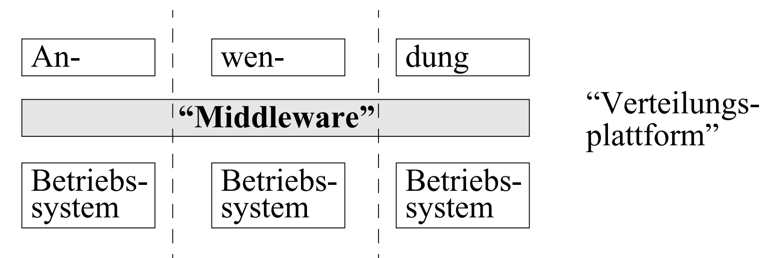
Middleware

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: “Middleware”



- Aufgabe:
  - Verteilung (für die Anwendung) möglichst transparent machen (z.B. globaler 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 Betriebsystemen (z.B. durch einheitliche Datenformate)
  - einheitliche „Umgangsformen“: Schnittstellen, Protokolle

- Sollte gewisse Basismechanismen für verteiltes Programmieren anbieten, z.B.
  - Verzeichnis- und Suchdienste (Nameservice, lookup service,...)
  - automatische Schnittstellenanpassung (Schnittstellenbeschreibungs-sprache, Stub-Compiler...)

Der Weg zum „Netzwerkrechner“

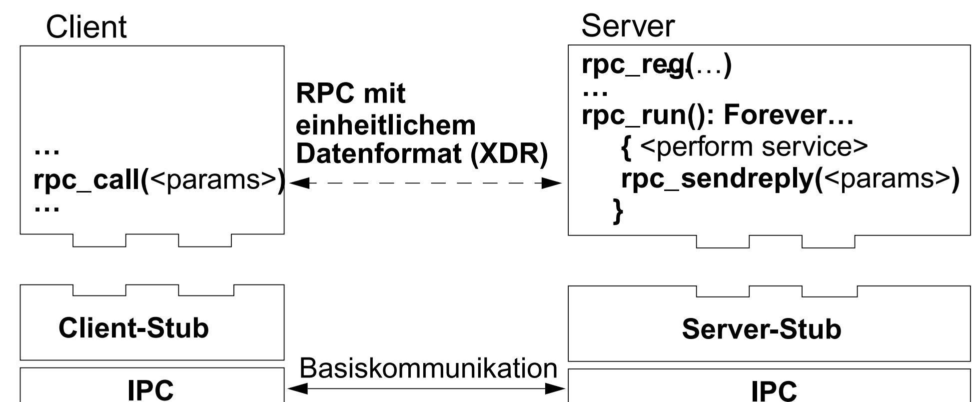
1. RPC-Pakete: z.B. Sun-RPC
   - Client-Server-Paradigma, RPC-Kommunikation
   - Schnittstellen-Beschreibungssprache, Datenformatkonversion, Stubgeneratoren
   - Sicherheitskonzepte (Authentifizierung, Autorisierung, Verschlüsselung)
2. Client-Server-Verteilungsplattformen: z.B. DCE
   - Zeitdienst, Verzeichnis- und Suchdienst
   - globaler Namensraum, globales Dateisystem
   - Programmierhilfen: Synchronisation, Multithreading ...
3. Objektbasierte Verteilungsplattformen: z.B. CORBA
   - Kooperation zwischen verteilten Objekten
   - objektorientierte Schnittstellenbeschreibungssprache, Vererbung
   - Objekt Request Broker
4. Web-Services
   - Dienstorientierung aufbauend auf dem WWW als Plattform
5. 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 Diensteinfrastruktur
- 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 Gerüstes



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

Sun-RPC: Komponenten

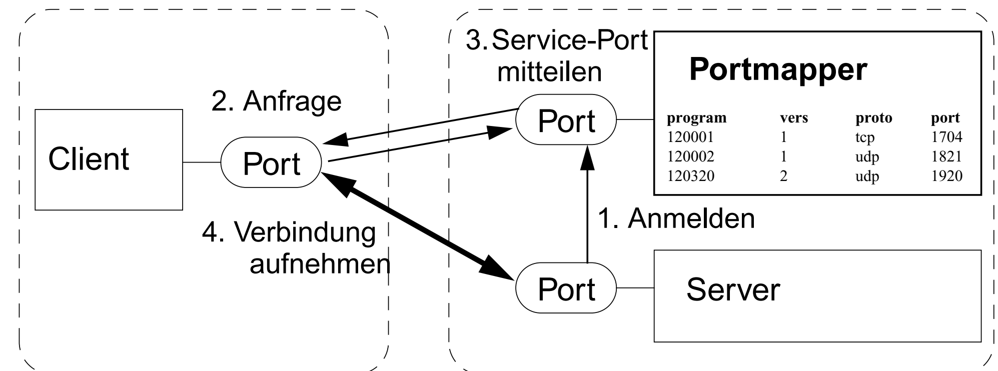
- RPC-Library: Vielzahl aufrufbarer Funktionen ("API")
  - z.B. rpc_reg, rpc_run, rpc_call
  - daneben auch Funktionen einer Low-Level-Schnittstelle: z.B. Spezifikation von Timeout-Werten oder eines Authentifizierungsprotokolls
- rpcgen: Stub-Generator
- Portmapper: Zuordnung Dienstnummer ↔ Portadresse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

---

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

Der Portmapper

- Bei Kommunikation über TCP oder UDP muss stets eine Portnummer angegeben werden
  - Portnummer ist zusammen mit der IP-Adresse Teil jedes UNIX-Sockets
- Jeder Dienst meldet sich beim lokalen Portmapper mit Programm-, Versions- und Portnummer an
  - Programmnummer ist primäre Kennzeichnung des Dienstes
  - ein Dienst kann in mehreren verschiedenen Versionen ("Releases") gleichzeitig vorliegen (Koexistenz von Versionen in der Praxis wichtig)



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

Portmapper (2)

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

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

Dynamisch generierte Port- und Programmnummern

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

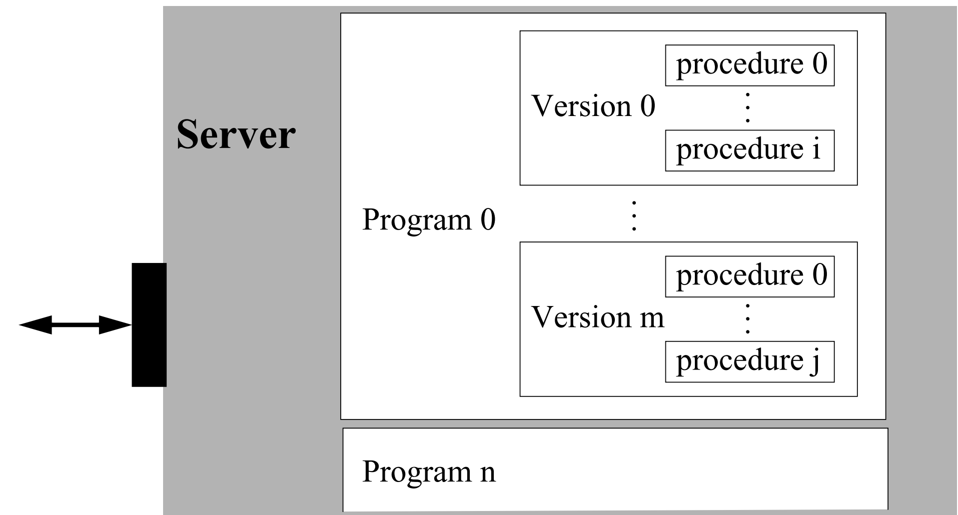
---

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

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

Service-Identifikation

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



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

- Prozedur Nummer 0 ist vereinbarungsgemäss für die “Nullprozedur” reserviert
  - keine Argumente, kein Resultat, sofortiger Rückkehr (“ping-Test”)

- Mit der Nullprozedur kann ein Client feststellen, ob ein Dienst in einer bestimmten Version existiert:
  - falls Aufruf von Version 4 des Dienstes XYZ nicht klappt, dann versuche, Version 3 aufzurufen...

Service-Registrierung

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

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

- Welche Programmnummer bekommt ein Service?

→ Einige Programmnummern für *Standarddienste* sind vom System bereits konfiguriert:

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

Linke Spalte: Servicename

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

Rechte Spalte: Kommentar

→ Ansonsten freie Nummer wählen:

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.

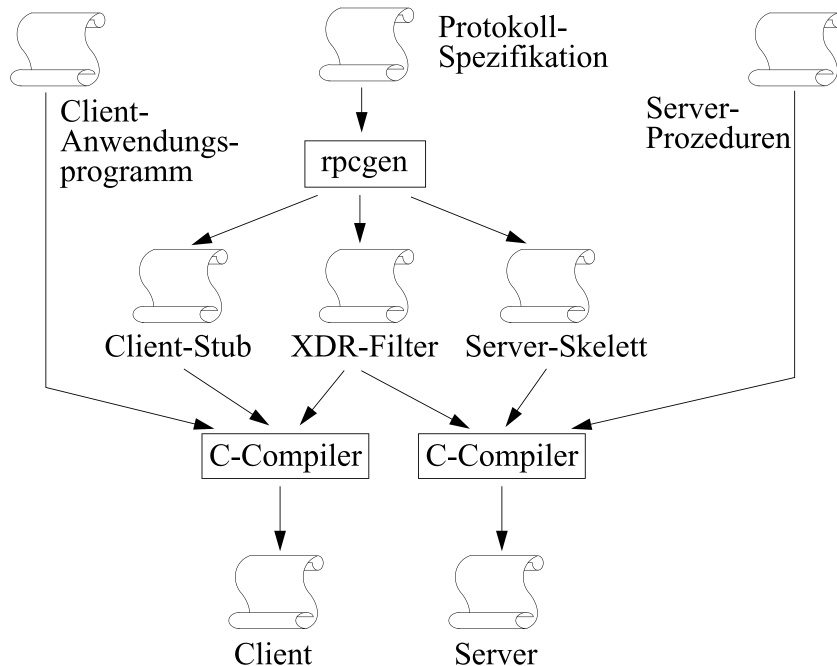
- Es gibt auch eine entsprechende Broadcast-Variante:

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.

Stub- und Filtergenerierung

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



Beispiel zu rpcgen

Die Ausgangsdatei add.x mit der *Protokollspezifikation*:

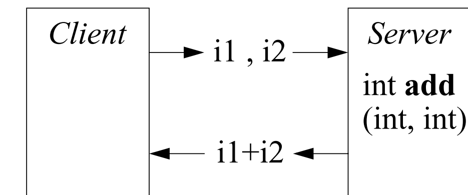
```
struct i_result
{ int x; };

struct i_param
{ int i1;
  int i2; };

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

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

Beispiel: ein “Additionsserver”:



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

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

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

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

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

Generierter Client-Code (Auszug)

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

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

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

     clnt = clnt_create(host, ADD_PROG, ADD_VERS, "netpath");

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

im handle "clnt" stecken die weiteren Angaben

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

hier Server ("host") lokalisieren!

hier Parameter setzen!

eigentlicher Prozeduraufruf

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

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

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

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

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

Generierter Server-Code (Auszug)

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

svc_run();
```

svc_reg funktioniert analog zu rpc_reg

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

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

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

Bem.: Server-Code ist über 200 Zeilen lang

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

---

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

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

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:* sichere Authentifizierung (“Secure RPC”)

  - *KERB:* sichere Authentifizierung mit Kerberos
    - Kerberos-Dienst 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 (...);
};
```


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

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

- Probleme...
  - gleiche Benutzer müssen auf verschiedenen Systemen die gleiche (numerische) uid-Kennung haben
  - ungesichert gegenüber Manipulationen
  - Homogenität: 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 *(“shared secret”)*

- 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 (ebenfalls verschlüsselt) W-1
  - “zufälliges” Generieren einer ersten Nachricht ist nahezu unmöglich
  - replay (bei kleinem W) ist ebenfalls erfolglos
  - W ist verschlüsselt, um Angreifern keine Informationüber die Fenstergrösse und auch kein Klartext-Schlüsseltext-Paar zu geben

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

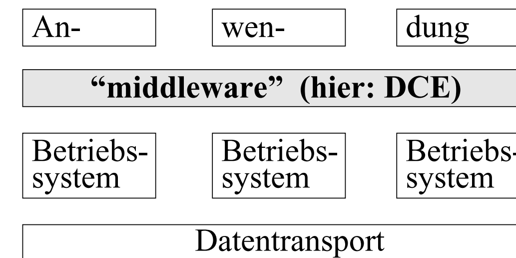
- Die Antwort des Servers enthält (verschlüsselt) den letzten erhaltenen Zeitstempel-1 (→ Authentifizierung!)

- 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 1990er Jahre, u.a. DEC, IBM, Siemens, HP...
  - trotz CORBA, Web Services etc. noch lange eingesetzt in der Praxis

- System aus zusammenwirkenden Softwarekomponenten (Werkzeuge, Dienste, Laufzeitmechanismen) zur Realisierung verteilter Anwendungen in offenen heterogenen Umgebungen

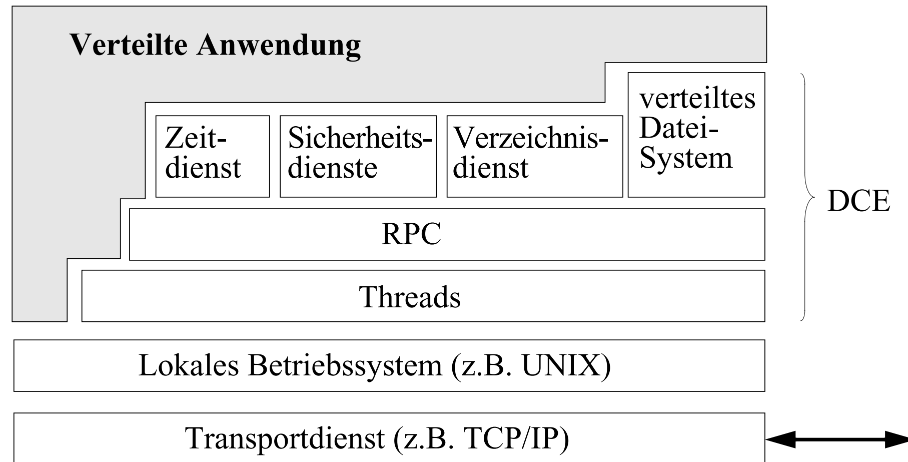


- Ziel: Schaffung eines “offenen” Industriestandards für verteilte Verarbeitung

- Vorgehensweise pragmatisch: Soweit möglich, Nutzung geeigneter existierender Technologiekomponenten

- Realisierung auf verschiedenen Plattformen
  - Hardware: IBM, Sun, ...
  - Software: UNIX-basiert, z.B. HP-UX, Solaris; aber auch Windows, OS/390, Macintosh,...

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

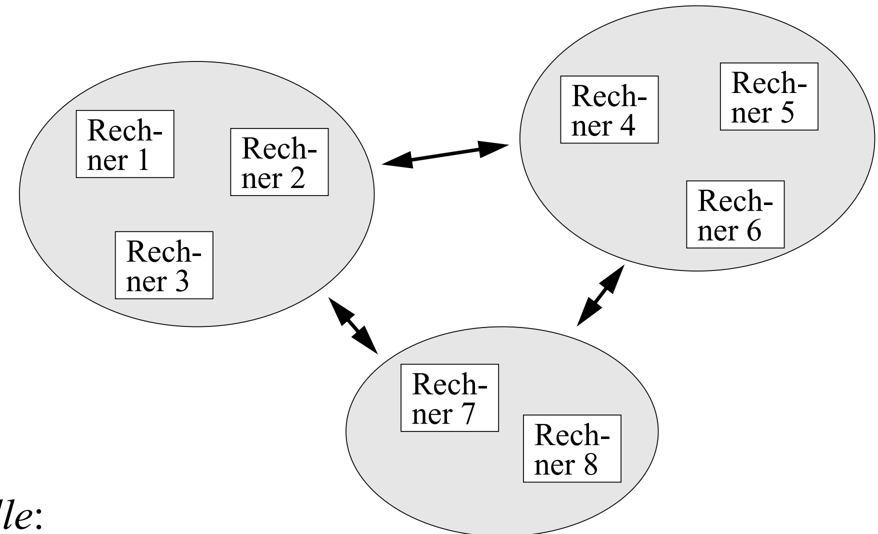
- Höhere ("verteilte") *Dienste*: u.a. Dateisystem, Verzeichnis- und Namensdienst, Sicherheitsdienst

- 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



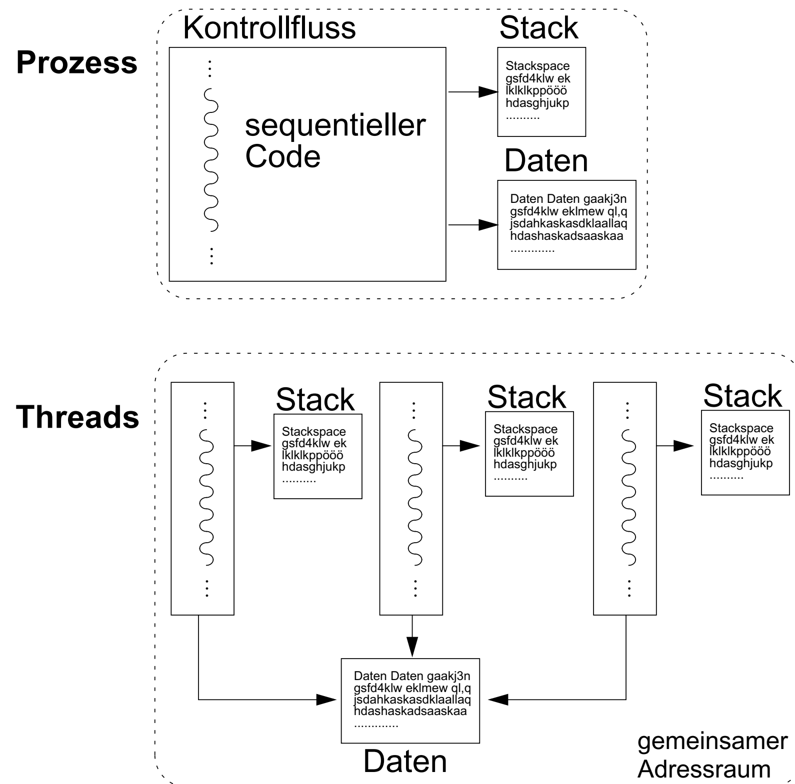
- *Zelle*:
  - Ist eine abgeschlossene organisatorische Einheit aus Rechnern, Ressourcen, Benutzern
    - z.B. Abteilung einer Firma
    - i.a. jeweils verantwortlicher Systemverwalter notwendig
    - bildet jeweils eine eigene Schutzzone bzgl. Sicherheitsaspekte
  - Hat *Cell Directory Service* (CDS), *Sicherheitsdienst* und *Zeitdienst* eingerichtet
    - realisiert durch dauerhafte Prozesse ("Dämonen")
    - ggf. weitere Dienste, z.B. Distributed File System (DFS)

- Prozesse können per RPC auch zellübergreifend kommunizieren (bei Kenntnis entfernter Adressen)

- *Zellübergreifende Services* (z.B. Zeitservice, Namensverwaltung...) mittels dedizierter Protokolle

DCE: Threads

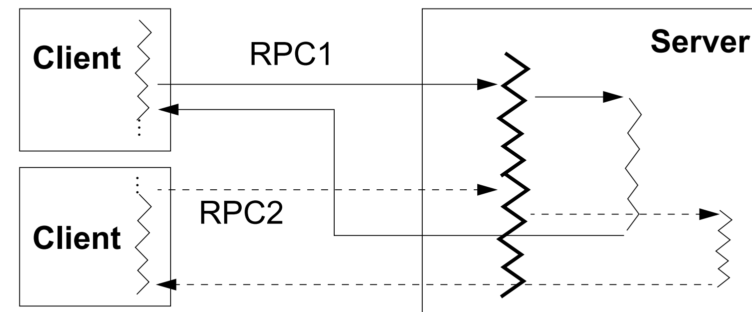
- Threads = leichtgewichtige Prozesse mit gemeinsamem Adressraum



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

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

- 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 Auschluss mehrerer Threads (“mutex”)
  - nebenläufige Signalverarbeitung und Ausnahmebehandlung

- Pro Adressraum existiert ein eigener *Thread-Scheduler*
  - verschiedene Schedulingstrategien wählbar (z.B. FIFO, Round Robin)
  - wahlweise Verwendung von Zeitscheiben (“präemptiv“)
  - wahlweise Berücksichtigung von Prioritätsstufen

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 asynchroner Operationen zum Test auf mögliche Blockaden

---

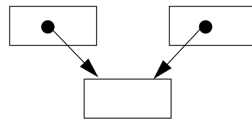
- Prinzipielle Probleme der Thread-Verwendung:
  - fehlender gegenseitiger Adressraumschutz → schwierige Fehler
  - Stackgrösse muss bei Gründung i.a. statisch festgelegt werden → unkalkulierbares Verhalten bei Überschreitung
  - von asynchronen Meldungen (“Signale”, “Interrupts”) an den Prozess soll 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

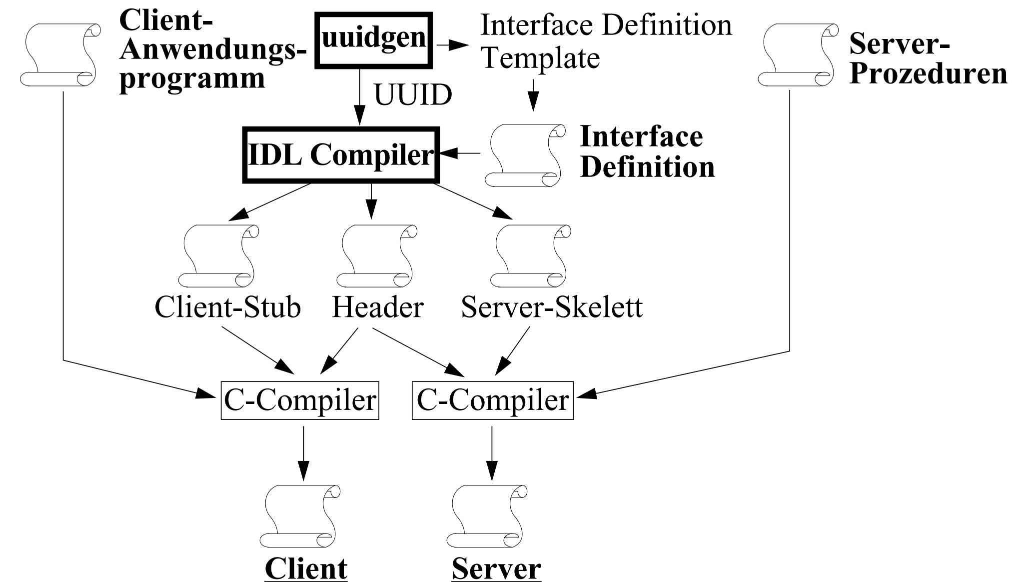
- Nahezu beliebige Parametertypen
  - alle C-Datentypen ausser Prozeduradressen ← müsste “remote” interpretiert werden
  - auch verzeigerte Strukturen, dynamische arrays
  - Zeiger werden automatisch dereferenziert und als Wert übergeben; jedoch Vorsicht bei Aliaszeigern!



- 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

DCE: Erzeugen von Client- und Server-Programmen

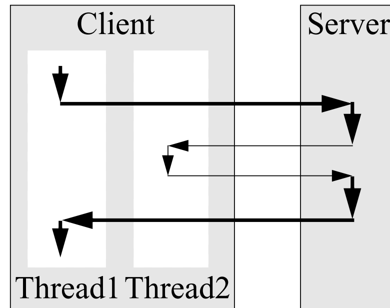


- UUID (Universal Unique Identifier) ist eine aus Uhrzeit und Rechnerkennung generierte systemweit eindeutige Kennung der Schnittstelle

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 marshall and unmarshall 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 marshall 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, 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

DCE: Sicherheit

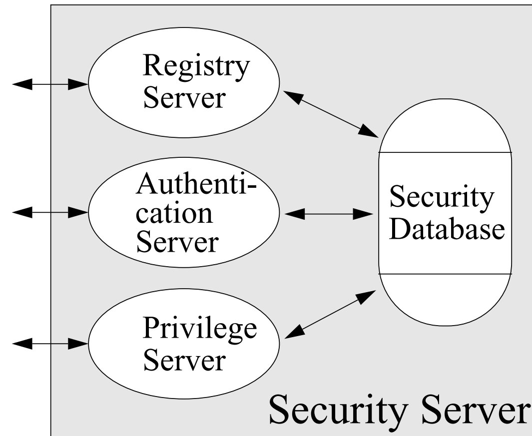
- Verwendung des Kerberos-Protokolls
  - Vertraulichkeit durch Sitzungsschlüssel (→ DES)
  - gegenseitige Authentifizierung
  - selektive Autorisierung von Clients für bestimmte Dienste
  - Schlüsselverwaltung
  - zusätzlich auch asymmetrische Verfahren ("public key")

- Wählbare Sicherheitsstufen bei der Kommunikation
  - Authentifizierung nur bei Aufbau der Verbindung ("binding")
  - Authentifizierung pro RPC-Aufruf
  - Authentifizierung pro Nachrichtenpaket
  - Zusätzlich Verschlüsselung jedes Nachrichtenpaketes
  - Schutz gegen Verfälschung (verschlüsselte Prüfsumme)

- Autorisierung ist mittels Zugriffkontroll-Listen realisiert
  - es gibt zahlreiche verschiedene Typen von Rechten
  - Gruppenbildung von Benutzern / Clients möglich
  - ACL-Manager bei den Servern verwaltet lokale Kontroll-Listen
  - Clients schicken eine verschlüsselte, authentische und gegen Replays gesicherte Repräsentation ihrer Rechte mit jedem Aufruf mit (PAC = Privilege Attribute Certificate); wird vom ACL-Manager überprüft

- Werkzeuge zur Systemadministration
  - Eintragen / Ändern von Rechten etc.
  - Installation zellübergreifender Sicherheitsdienste
  - hierzu spezieller "Registry-Server"

DCE-Sicherheitsdienste



- *Registry Server:* Verwaltung von Benutzerrechten; Dienste für Systemverwaltung
- *Datenbasis* enthält private Schlüssel (u.a. Passwörter in verschlüsselter Form...)
- *Privilege-Server* überprüft Zugangsberechtigung; u.a. bei login

- Sicherheitsdienst kann *repliziert* werden, um hohe Verfügbarkeit zu erreichen
  - nur Primärkopie kann Daten aktualisieren, Replikate sind "read only"
  - Primärkopie aktualisiert gezielt die Replikate

- *Zellenübergreifende Sicherheitsdienste:*

  - ein Security Server A nimmt gegenüber einem Security Server B eine Clientrolle ein ("vertritt" die Clients seiner Zelle)
  - ein Security Server besitzt im Gegensatz zu anderen Clients nicht einen einzigen geheimen Schlüssel, sondern es werden paarweise spezifische Schlüssel vereinbart

Weitere DCE-Komponenten

- Cell Directory Service (CDS)
  - realisiert Zuordnung von Namen und Adressen
  - verwaltet Namen (mit Attributen) einer Zelle
  - Beispiel für Attribute: *Name*, *Standort, Typ* eines Druckers
  - Replikation (zwecks Fehlertoleranz) möglich (dabei "Konvergenzlevel" einstellbar)

- Global Directory Service (GDS)
  - Bindeglied zwischen verschiedenen CDS
  - hierarchischer Namensraum

(CDS und GDS: Namensverwaltung)

- Distributed File System (DFS)
  - ortstransparenter Dateizugriff
  - Caching beim Client steigert Effizienz ("Session-Semantik")
  - mehrere Read-only-Replikate möglich
  - Unterstützung von Recovery, Migration und Backup
  - Synchronisation gleichzeitiger Zugriffsversuche
  - Gruppierung durch "File Sets" (Gruppen von Dateien, die zusammen gelagert werden sollten)
  - nutzt DCE-RPC

- Distributed Time Service (DTS)
  - Synchronisationsprotokoll zwischen mehreren lokalen Zeitservern
  - Einbeziehung externer Zeitgeber (z.B. Funk- und Atomuhren)
  - Kopplung mit NTP-Protokoll möglich

DCE: Pragmatisches

Es gibt verschiedene Administrationstools

- Anzeigen und verändern von Information
- command line interface oder graphische Benutzungsoberfläche

---

Kritik an DCE: Komplexität

- Funktionsfülle (> 200 Funktionen)
  - wann benutzt man was?
  - Problem der wechselseitigen Beeinflussung („feature interaction“)
  - Semantik bei Kombination verschiedener Mechanismen u.U. unklar
- Grösse
- mangelnde Effizienz