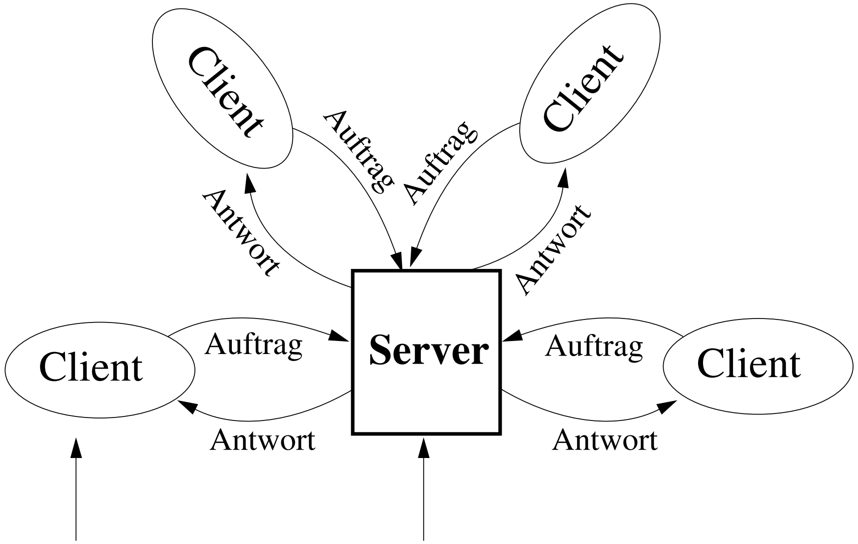
Das Client/Server-Modell



initiierender Prozess
(stellt Anfragen, erteilt Aufträge)

reagierender Prozess
(bearbeitet Anfragen und Aufträge; i.a. mit besonderen Ressourcen ausgestattet)

- Aufgabenteilung und asymmetrische Struktur
    - *Clients*: typischerweise Anwendungsprogramme und graphische Benutzungsschnittstelle ("front end")
    - *Server*: zuständig für Dienstleistungen
- Typisches Kommunikationsparadigma: RPC

Client-Kellner: Eine solche Architektur beschreibt wie zwei Wesenheiten (der Kunde und der Kellner) können in einem Netz kommunizieren. Der Kunde sendet eine Bitte(Antrag) dem Kellner, der ihn ausführt und ihm das Ergebnis zurücksendet.

Client-serveur: Une telle architecture décrit comment deux entités (le client et le serveur) peuvent communiquer sur un réseau. Le client envoie une requête au serveur, qui l’exécute et lui renvoie le

http://perso.wanadoo.fr/lexiqueinformatique/

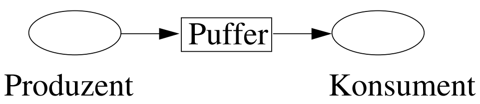
Eignung des Client/Server-Paradigmas

- Populär wegen des eingängigen Modells
  - entspricht Geschäftsvorgängen in unserer Dienstleistungsgesellschaft
  - gewohntes Muster --> intuitive Struktur, gute Überschaubarkeit
- Effizienz durch spezialisierte „Dienstleister“
  - verfügen ggf. über Spezialhardware
  - grosszügige Ausstattung (CPU-Leistung, Speicherkapazität usw.)
  - bestückt mit Spezialsoftware (Datenbank etc.)
- Kosteneffektivität durch bessere Auslastung wertvoller Ressourcen
  - Clients brauchen oft kurzfristig Spitzenleistung
  - einzelner Client kann Ressourcen aber nicht dauerhaft auslasten
- Passend für viele Kooperationsbeziehungen, z.B.
  - schwacher Client erbittet Hilfe vom „grossen Bruder“
  - gefährdete Clients geben wertvolle Daten in Obhut des (gegen Missbrauch, Brand, Diebstahl usw.) hoch gesicherten Servers

---

- Modell ist für viele Zwecke geeignet, jedoch nicht für alle (z.B. Pipelines, “peer-to-peer”, asyn. Mitteilung)!

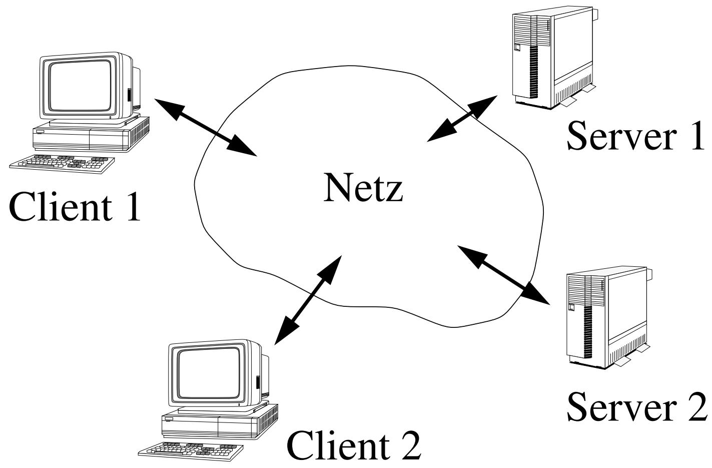
Beispiel Puffer als Pipeline:



- Puffer ist weder Client noch Server, sondern hat beide Rollen! (passiv gegenüber Produzent; aktiv gegenüber Konsument)
- Inversion der Kommunikationsbeziehung bei einem Puffer-Server!

Client- und Server-Maschinen

- Übertragung des Client/Server-Modells auf *Rechner*



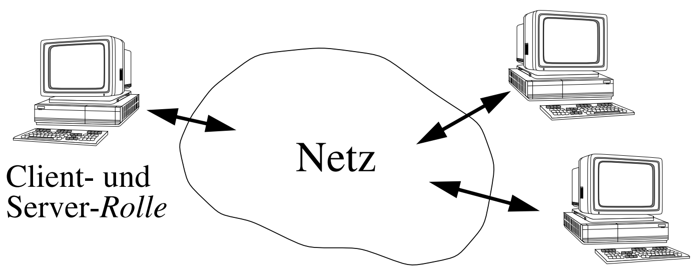
- Typischerweise PCs oder Workstations als Clients
  - u.a. mit graphischem Benutzungsinterface
- Andere Rechner als Server
  - “zentrale” Dienste (z.B. Plattenserver)
  - gemeinsam benutzte Betriebsmittel

- Im allgemeinen müssen sich aber Server- und Client-*Prozesse* nicht auf dedizierten Rechnern befinden!
- “Client/Server-Computing” wird heute in der Praxis oft missbräuchlich als synonym zu “Distributed Computing” gebraucht
  - beachte: Kern der Rechenleistung wird *zentral* auf dem Server erbracht!

Peer-to-peer-Strukturen

"Gleichrangiger"

- Im "Gegensatz" zum Client/Server-Modell



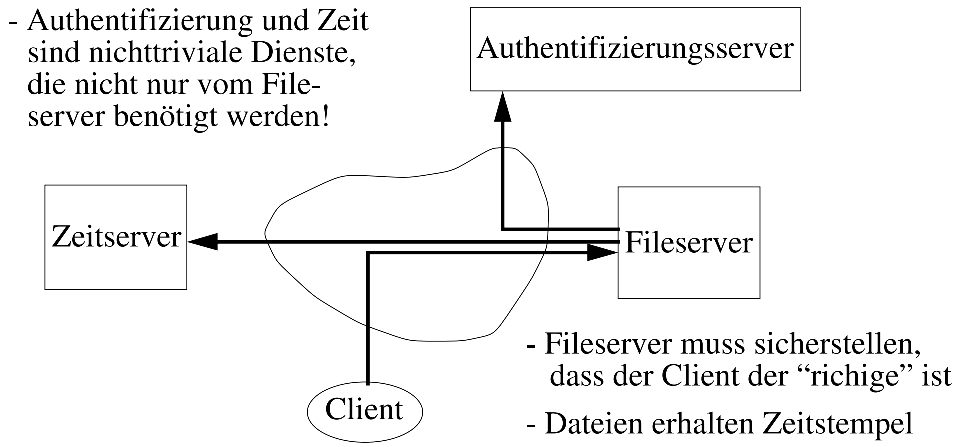
- Jeder Client fungiert zugleich als Server für seine Partner
  - --> keine (teuren) dedizierten Server notwendig
  - --> oft als Billiglösung von "echtem" Client/Server-Computing angesehen

- Ggf. können zusätzliche dedizierte Server existieren

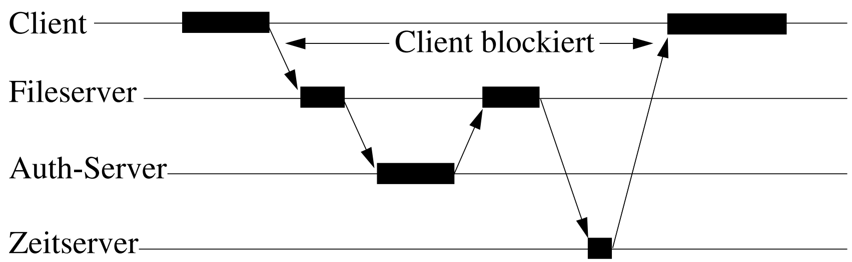
- *Nachteile*:
  - "Anarchischer" als *maschinenbezogene* Client/Server-Architektur
  - Rechner müssen leistungsfähig genug sein (cpu-Leistung, Speicherausbau), um für den "Besitzer" leistungstransparent zu sein
  - geringere Stabilität (Besitzer kann seine Maschine ausschalten...)
  - Datensicherung muss ggf. dezentral durchgeführt werden
  - Sicherheit und Schutz kritisch: Lizenzen, Viren, Integrität...

Client/Server-Rollen

- Server müssen ggf. zur Durchführung eines Dienstes die Dienstleistungen anderer Server in Anspruch nehmen



- Fileserver hat prinzipiell die Rolle eines Servers, zwischenzeitlich jedoch die Rolle eines Clients



Zustandsändernde /-invariante Dienste

(Vgl. frühere Diskussion bzgl. RPC-Fehlersemantik!)

- Verändern Aufträge den Zustand des Servers?
  - nur zwischenzeitlich (--> Atomizität) oder generell?
- Typische *zustandsinvariante* Dienste:
  - Auskunftsdienste (Name-Service; aktuelle Lastsituation;...)
  - Zeitservice
- Typische *zustandsändernde* Dienste:
  - Datei-Server

Idempotente Dienste / Aufträge

- Wiederholung eines Auftrags liefert gleiches Ergebnis

nicht zustandsinvariant!

  - Beispiel: “Schreibe in Position 317 von Datei XYZ den Wert W”
  - Gegenbeispiel: “Schreibe ans Ende der Datei XYZ den Wert W”
  - Gegenbeispiel: “Wie spät ist es?” aber zustandsinvariant!

Wiederholbarkeit von Aufträgen

- Bei Idempotenz oder Zustandsinvarianz kann bei Verlust des Auftrags (timeout beim Client) dieser erneut abgesetzt werden (--> einfache Fehlertoleranz)

Zustandslose / -behaftete Server

stateless statefull

“session”

- Hält der Server Zustandsinformation über Aufträge?
  - z.B. (Protokoll)zustand des Clients
  - z.B. Information über frühere damit zusammenhängende (Teil)aufträge
- Aufträge an zustandslose Server müssen autonom sein

- *Beispiel*: Datei-Server

```
open(“XYZ”);
read;
read;
close;
```

In klassischen Systemen hält sich das Betriebssystem Zustandsinformation über die Position des Dateizeigers geöffneter Dateien (in UNIX ausserdem i-node-Nummer etc.)

  - bei zustandslosen Servern entfällt open/close; jeder Auftrag muss vollständig beschrieben sein (Position des Dateizeigers etc.)

notw. Zustandsinformation ist beim Client

  - zustandsbehaftete Server daher i.a. effizienter
  - Dateisperren sind bei echten zustandslosen Servern nicht (einfach) möglich
  - zustandsbehaftete Server können wiederholte Aufträge erkennen (z.B. durch Speichern von Sequenznummern) --> Idempotenz

- *Crash* eines Servers: Weniger Probleme im zustandslosen Fall (--> Fehlertoleranz)!

entscheidender Vorteil!

- NFS (Network File System von Sun) ist zustandslos
- RFS (Remote File System von UNIX System V) ist zustandsbehaftet

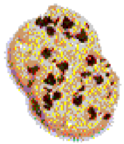
Sind WWW-Server zustandslos?

- Beim HTTP-Zugriffsprotokoll wird über den Auftrag hinweg keine Zustandsinformation gehalten
    - jeder Hyperlink, den man anklickt, löst eine neue “Transaktion” aus
- Stellt ein Problem beim Online-Commerce dar
    - gewünscht sind Transaktionen über mehrere clicks hinweg und
    - Wiedererkennen von Kunden (beim nächsten Klick oder Tage später)
    - erforderlich z.B. für Realisierung von “Einkaufskörben” von Kunden
    - gewünscht vom Marketing (Verhaltensanalyse von Kunden)

Lösungsmöglichkeiten (für Einkaufskörbe im WWW)

- IP-Adresse des Kunden an Auftrag anheften
    - Problem: Proxy-Server --> viele Kunden haben gleiche IP-Adresse
    - Problem: dynamische IP-Adressen --> keine Langzeitwiedererkennung
- “tag propagation”
    - Einstiegsseite eine eindeutige Nummer anheften, wenn der Kunde diese erstmalig aufruft
    - diese Nummer jedem link anheften und mit zurückübertragen
- Cookies
    - erstmalig im Netscape Navigator 1.0.1 verwendet
    - Textdatei < 4k, die ein Server einem Browser (= Client) schickt und die im Browser gespeichert wird
    - nur der Sender des Cookies darf dieses später wieder lesen

Cookies



Auszug aus
http://home.netscape.com/newsref/std/cookie_spec.html:

Cookies are a general mechanism which server side connections (such as CGI scripts) can use to both store and retrieve information on the client side of the connection. The addition of a simple, persistent, client-side state significantly extends the capabilities of Web-based client/server applications.

A server, when returning an HTTP object to a client, may also send a piece of state information which the client will store... Any future HTTP requests made by the client... will include a transmittal of the current value of the state object from the client back to the server. The state object is called a cookie, for no compelling reason.

This simple mechanism provides a powerful new tool which enables a host of new types of applications to be written for web-based environments. Shopping applications can now store information about the currently selected items, for fee services can send back registration information and free the client from retyping a user-id on next connection, sites can store per-user preferences on the client, and have the client supply those preferences every time that site is connected to.

A cookie is introduced to the client by including a Set-Cookie header as part of an HTTP response... The expires attribute specifies a date string that defines the valid life time of that cookie. Once the expiration date has been reached, the cookie will no longer be stored or given out... A client may also delete a cookie before it’s expiration date arrives if the number of cookies exceeds its internal limits.

- Denkübung: Müssen Proxy-Server geeignete Massnahmen vorsehen?
- Übung: Man finde heraus, was doubleclick.net macht (und wie)

Cookies (2)

- Anwendung von cookies war und ist umstritten (Ausspionieren des Verhaltens); dazu kam eine gewisse Paranoia:

http://www.cookiecentral.com/creport.htm

http://www.ciac.org/ciac/bulletins/i-034.shtml

The Energy Department's Computer Incident Advisory Capability (CIAC) recently issued a report on cookie technology and its use on the web....

The report stressed that there's a sense of paranoia involved with cookies, cookies cannot harm your computer or pass on private information such as an email address without the user's intervention in the first place. Paranoia has recently been sparked by one rumour involving AOL's new software, it claimed that AOL were planning to use cookies to obtain private information from users hard drives.

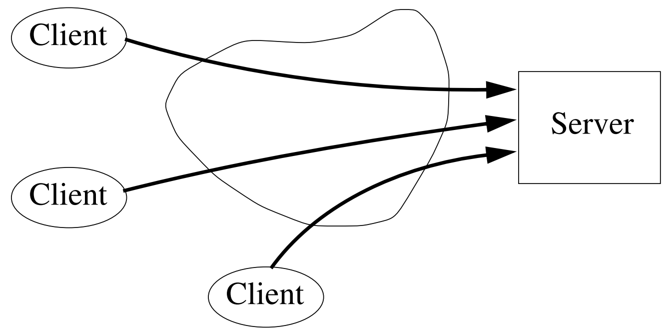
- Problemlos ist das allerding nicht:

http://www.cookiecentral.com/cookie5.htm

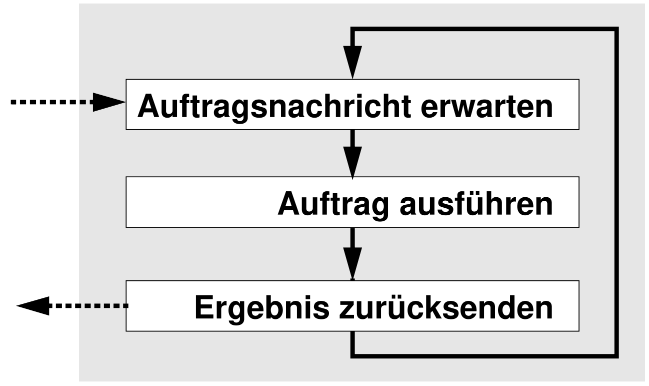
Unfortunately, the original intent of the cookie has been subverted by some unscrupulous entities who have found a way to use this process to actually track your movements across the Web. They do this by surreptitiously planting their cookies and then retrieving them in such a way that allows them to build detailed profiles of your interests, spending habits, and lifestyle... it is rather scary to contemplate how such an intimate knowledge of our personal preferences and private activities might eventually be used to brand each of us as members of a particular group.

Iterative Server

- Problem: Viele "gleichzeitige" Aufträge



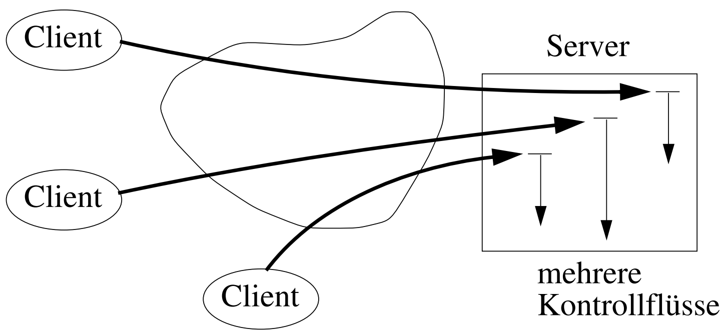
- *Iterative Server* bearbeiten nur einen Auftrag pro Zeit



- häufige Bezeichnung: "single threaded"
- eintreffende Anfragen während Auftragsbearbeitung: abweisen, puffern oder einfach ignorieren
- einfach zu realisieren
- bei "trivialen" Diensten sinnvoll (mit kurzer Bearbeitungszeit)

Konkurrente (“nebenläufige”) Server

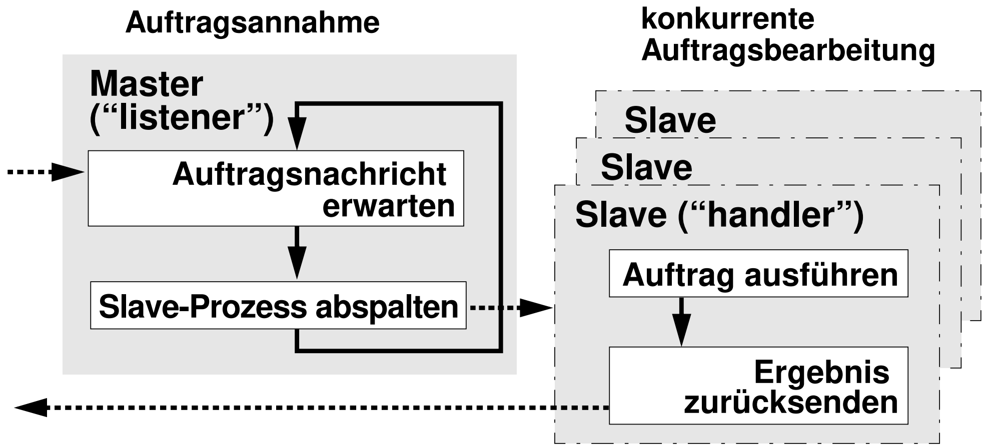
- Gleichzeitige Bearbeitung mehrerer Aufträge
  - sinnvoll (d.h. effizienter für Clients) bei langen Aufträgen (z.B. in Verbindung mit E/A)
  - Beispiel: Remote-login-Server; WWW-Suchmaschinen



- Ideal bei Mehrprozessormaschinen (physische Parallelität)
  - aber auch bei Monoprozessor-Systemen (vgl. Argumente bei Timesharing-Systemen): Nutzung erzwungener Wartezeiten eines Auftrags für andere Jobs; kürzere mittlere Antwortzeiten bei Jobmix aus langen und kurzen Aufträgen
- Interne Synchronisation bei konkurrenten Aktivitäten sowie ggf. Lastbalancierung beachten
- Verschiedene denkbare Realisierungen, z.B.
  - mehrere Prozessoren
  - Verbund verschiedener Server-Maschinen (z.B. LAN-Cluster)
  - dynamische Prozesse (bei Monoprozessor-Systemen)
  - dynamische threads
  - feste Anzahl vorgegründeter Prozesse
  - internes Scheduling und Multiprogramming

Konkurrente Server mit dynamischen Handler-Prozessen

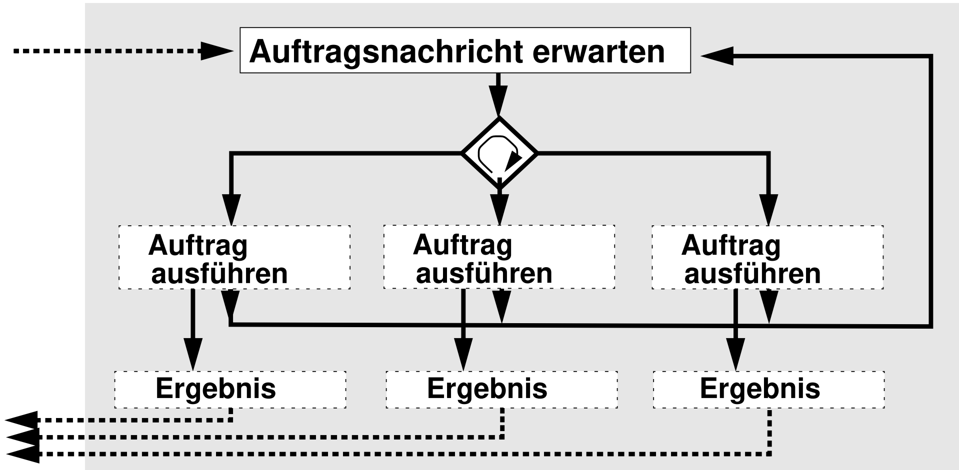
- Für jeden Auftrag gründet der *Master* einen neuen *Slave*-Prozess und wartet dann auf einen neuen Auftrag
  - neu gegründeter Slave (“handler”) übernimmt den Auftrag
  - Client kommuniziert dann ggf. direkt mit dem Slave (z.B. über dynamisch eingerichteten Kanal bzw. Port)
  - Slaves sind ggf. Leichtgewichtsprozesse (“thread”)
  - Slaves terminieren i.a. nach Beendigung des Auftrags
  - die Anzahl gleichzeitiger Slaves sollte begrenzt werden



- Alternative: “Process preallocation”: Feste Anzahl statischer Slave-Prozesse
  - ggf. effizienter (u.a. Wegfall der Erzeugungskosten)
- Übungsaufgaben:
  - herausfinden, wie es bei WWW-Servern gemacht wird (z.B. Apache)
  - wie sollte man bei grossen WWW-Suchmaschinen vorgehen?

Quasi-konkurrente Server

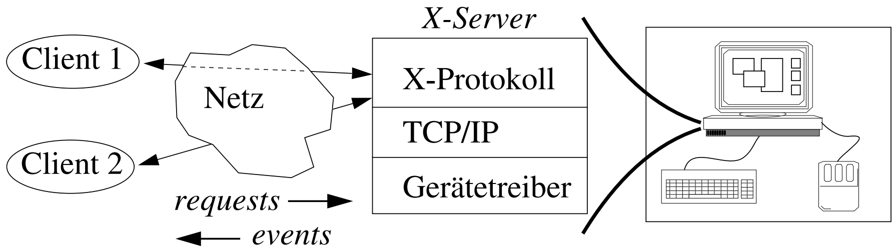
- Server besteht aus einem *einzigen Prozess*, der im Multiplexmodus mehrere Aufträge verschränkt abarbeitet
  - ggf. sinnvoll, wenn z.B. Clients grosse Datenmengen “stückweise” senden und die Wartezeiten dazwischen für die Bearbeitung der anderen Aufträge verwendet werden kann



- Keine Neugründung von Slave-Prozessen
- Keine Adressraumgrenzen zwischen Auftragsdaten
  - keine kostspieliger Kontextwechsel
  - auftragsübergreifende gemeinsame Datenhaltung effizienter (vgl. X-Server: Alle Clients (z.B. xclock) schreiben Display-Daten in einen gemeinsamen “Display-Puffer”)
- Potentielle Nachteile: kein Adressraumschutz zwischen verschiedenen Aufträgen; ggf. unnötige Wartezeiten z.B. bei blockierenden Betriebssystemaufrufen

“X-Window” als Client/Server-Modell

- Betriebssystem- und netzwerkunabhängiges Graphik- und Fenstersystem für Bildschirme
- Entwickelt Mitte der 80er Jahre am MIT, zusammen mit DEC



- i.a. bedient ein Server mehrere Client-Prozesse (“Applikationen”), die ihre Ausgabe auf dem gleichen Bildschirm erzeugen
- *Window-Manager*: Spezieller Client, der Grösse und Lage der Fenster und Icons steuert (Beispiele: twm, mwm, fvwm)

→ X windows system protocol (über TCP)

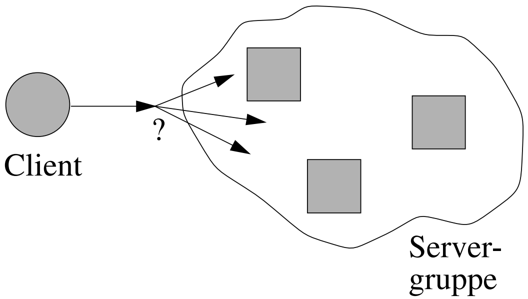
- *Requests*: Service-Anforderung an den X-Server (z.B. Linie in einer bestimmten Farbe zwischen zwei Koordinatenpunkten zeichnen); zugehörige Routinen stehen in einer Bibliothek (*Xlib*)
- *X-Library* (Xlib) ist die Programmierschnittstelle zum X-Protokoll; damit manipuliert ein Client vom Server verwaltete Ressourcen (Window, font...); höhere Funktionen (z.B. Dialogboxen) in einem (von mehreren) X-Toolkit
- *Events*: Tastatur- und Mauseingaben (bzw. -bewegungen) werden vom X-Server asynchron an den Client des “aktiven Fensters” gesendet (keine klassische Server-Rolle --> schwierig mit RPCs zu realisieren!)
- X ist ein *verteiltes System*: Client-Prozesse können sich auf verschiedenen Rechnern befinden (“Fenster verschiedener Rechner”)
- *X-Terminal* hat Server-Software im ROM (bzw. lädt sie beim Booten)
- es gibt vielfältige Standard-*Utilities* und *Tools* (xterm, xclock, xload...)

Servergruppen und verteilte Server

- Idee: Ein Dienst wird nicht von einem einzigen Server, sondern von einer Gruppe von Servern erbracht

a) *Multiple Server*

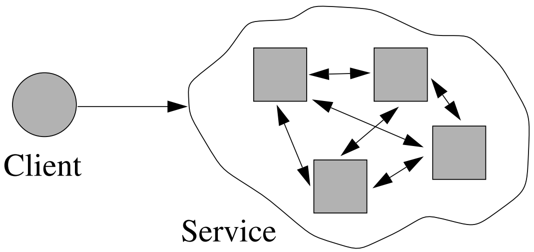
- Jeder einzelne Server kann den Dienst erbringen
- Zweck:
  - *Leistungssteigerung* (Verteilung der Arbeitslast auf mehrere Server) ← “Lastverbund”
  - *Fehlertoleranz* durch Replikation (Verfügbarkeit auch bei vereinzelten Server-Crashs) ← “Überlebensverbund”



- z.B. Multicast an eine Server*gruppe*
- oder Auswahl eines *bestimmten* Servers (z.B. nahegelegenen oder wenig belasteten)

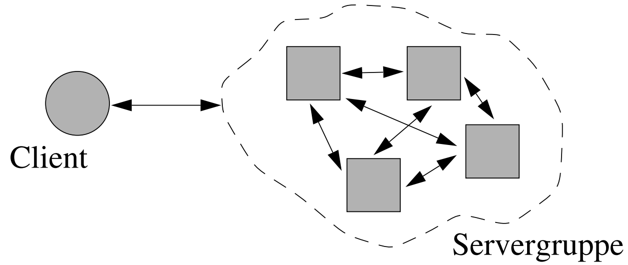
b) *Kooperative Server* ← “Know-how-Verbund”

- ein Server allein kann den Dienst nicht erbringen
- z.B. rwho; Gesamtauslastung...

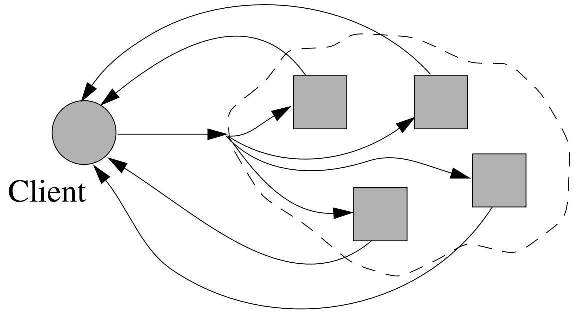


Strukturen kooperativer Server

1) Echte Kooperation: Server liefern gemeinschaftlich ein Gesamtergebnis

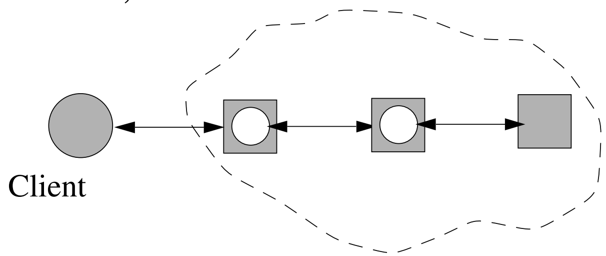


2) Paarweise Kooperation mit dem Client: Client akkumuliert Teilergebnisse



Z.B.: Broadcast stösst einzelne Rückmeldungen an und Client “akkumuliert” die Information

3) Kaskadierung: Dienst als Menge von Teildiensten realisiert, z.B.:



- Server während der Auftragsbearbeitung als Client bzgl. Teilaufträgen

Beispiel: ruptime

- “remote uptime”
- “r-Kommandos”: Verteilte Dienste, die bekannte UNIX-Kommandos auf die Verwendung im Netzbetrieb erweitern.
- Weitere r-Kommandos, z.B: rlogin, rsh, rexec, rcp, rwho, rusers.

```
NAME
     ruptime - show host status of local machines

     ruptime gives a status line like uptime for each machine  on
     the  local  network; these are formed from packets broadcast
     by each host on the network once a minute.

     Machines for which no status report has been received for  5
     minutes are shown as being down.

BUGS
     Broadcasting does not work through gateways.
```

Router etc.

```
----------------------------------------------------------------------

sol[52] [~] ruptime
cadsun        up 34+12:39,     0 users,  load 1.28, 1.28, 1.06
hssun2      down     1:21
martine       up  5+10:55,     0 users,  load 0.10, 0.05, 0.04
nuriel        up  5+11:04,     0 users,  load 0.11, 0.11, 0.11
octopus       up  5+10:43,     0 users,  load 0.02, 0.04, 0.03
paloma        up  5+07:10,     0 users,  load 0.00, 0.08, 0.06
quantas       up  5+10:52,     0 users,  load 0.00, 0.02, 0.02
sbecserver    up 39+13:18,     4 users,  load 2.05, 1.21, 0.52
sbsvax.cs.un  up  2+05:18,     0 users,  load 0.00, 0.06, 0.06
sol           up  1+06:27,    11 users,  load 5.12, 5.12, 5.12
```

verschiedene Maschinen

Der rwhod-Dämon

- “Dämon”: Server, der auf das Auftreten von Ereignissen wartet, und dann darauf reagiert; wird i.a. bei Systemstart gegründet.

```
NAME
     rwhod - system status server

DESCRIPTION
     rwhod is the server which maintains the database used by the
     rwho(1C) and ruptime(1C) programs.

     rwhod operates as both a producer  and  consumer  of  status
     information.   As  a producer of information it periodically
     queries the state of the system and constructs  status  mes-
     sages  which  are  broadcast on a network.  As a consumer of
     information, it listens for other rwhod servers’ status mes-
     sages,...

     Status messages are generated approximately  once  every  60
     seconds.

BUGS
     This service takes up progressively more  network  bandwidth
     as  the  number  of  hosts  on the local net increases.  For
     large networks, the  cost  becomes  prohibitive.

-------------------------------------------------------------------
```

- kein echtes Client/Server-Modell!
- aus Performance-Gründen oft deaktiviert
- Neuimplementierung (‘rup’ statt ‘ruptime’): kein default-Broadcast, sondern nur Broadcast bei Aufruf des Kommandos; rstatd-Dämonen anderer Rechner antworten dann

kernel statistics server

Das rup-Kommando

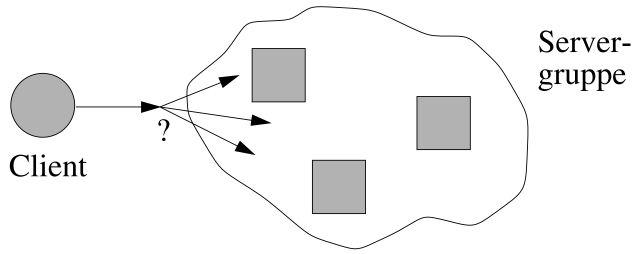
```
DESCRIPTION
     rup gives a status similar to uptime  for  remote  machines.
     It  broadcasts  on  the  local  network,  and  displays  the
     responses it receives.

     Normally, the listing is in the  order  that  responses  are
     received.

BUGS  Broadcasting does not work through gateways.
```

```
 -------------------------------------------------------------
sun10    up           11:56,    load average: 2.01, 2.01, 1.96
sun33    up 10 days,   2:51,    load average: 0.98, 1.00, 1.01
sun72    up            9:26,    load average: 0.21, 0.25, 0.30
sun13    up  1 day,   10:29,    load average: 0.02, 0.04, 0.04
sun14    up           15:24,    load average: 0.10, 0.05, 0.04
sun45    up  1 day,   11:07,    load average: 0.00, 0.02, 0.04
sun16    up 22 days,   9:36,    load average: 0.07, 0.02, 0.03
sun17    up           15:29,    load average: 0.02, 0.05, 0.05
sun18    up  2 days, 15:15,     load average: 0.01, 0.01, 0.01
sun19    up  2 days, 15:31,     load average: 0.84, 0.37, 0.21
sun20    up 10 days, 15:17,     load average: 0.00, 0.02, 0.05
sun27    up  9 days, 15:21,     load average: 1.00, 1.05, 1.07
sun18    up 14 days, 13:37,     load average: 0.09, 0.08, 0.07
sun31    up 65 days, 12:42,     load average: 0.04, 0.03, 0.05
sun34    up 23 days,  3:15,     load average: 0.02, 0.02, 0.02
sun56    up  2 days, 15:06,     load average: 0.00, 0.02, 0.04
sun57    up 22 days,  9:03,     load average: 0.02, 0.04, 0.04
sun58    up  3 days,  8:34,     load average: 0.00, 0.01, 0.03
sun59    up  3 days, 15:22,     load average: 0.05, 0.05, 0.04
sun60    up           15:23,    load average: 0.00, 0.02, 0.03
sun61    up 31 days,  7:10,     load average: 0.01, 0.03, 0.04
```


Serverwahl bei einem Lastverbund

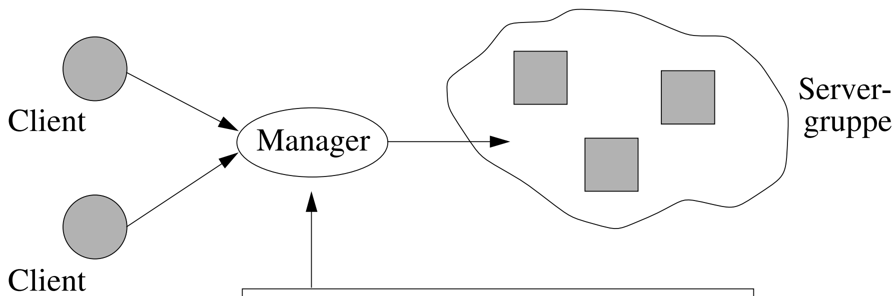


1) *Zufallsauswahl*

- Einfaches, effizientes Protokoll
- Nachteile:
  - Client muss alle Server kennen
  - ggf. ungleichmässige Auslastung

Stellen Verfahren mit “round robin”-Einträgen im DNS-System eine solche Zufallsauswahl dar?

2) *Zentraler Service-Manager*

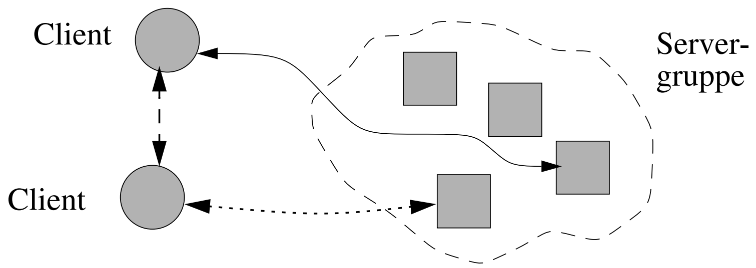


- sorgt für sinnvolle Verteilung (wie?)
- behält ggf. Überblick über Aufträge
- informiert sich ggf. von Zeit zu Zeit über die Server-Lastsituation

- Nachteile:
  - Overhead bei trivialen Diensten
  - ggf. Überlastung des Managers
  - Dienstblockade bei Ausfall des Managers

Serverwahl bei Lastverbund (2)

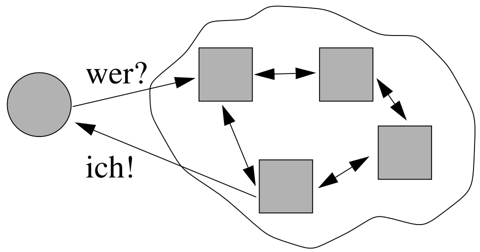
3) *Clients einigen sich untereinander*



- u.U. grosser Kommunikationsaufwand zwischen vielen Clients
- Clients kennen sich i.a. nicht (z.B. bei dynamisch gegründeten)

4) *Server einigen sich untereinander*, wer den Auftrag ausführt

- Election-Protokoll (aber fehlertolerant wegen möglichen Server-Ausfällen)
- ggf. Kooperations-topologie festlegen

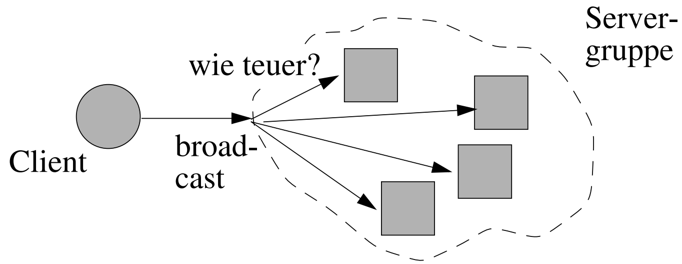


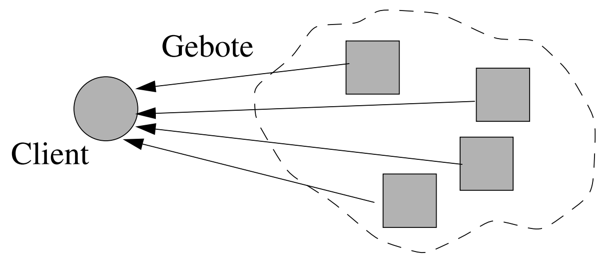
- i.a. nur wenige Server (relativ zur Zahl der Clients)
- Server führen Abstimmung diszipliniert durch (verlässlicher als Clients)

Serverwahl bei Lastverbund (3)

5) *Bidding-Protokoll*

- Client fragt per Broadcast nach Geboten
- Server mit "billigstem" Angebot wird ausgewählt





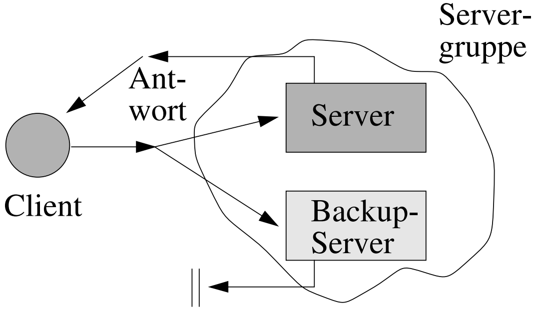
- Variante: nur *Stichprobe* befragen (multicast statt broadcast; sehr kleine Teilmenge von vielen Servern genügt i.a.!)

- Generelles Problem: Lastsituation kann veraltet sein!

Serverreplikation in Überlebensverbunden

1) *Zustandsinvariante Dienste*: im Prinzip einfach - nach Crash anderen Server nehmen...

2) *Zustandsändernde Dienste* (hier “hot stand by”):



aktiver Server

passiver Server im “hot stand-by”-Modus (ggf. mehr als einer)

- im Fehlerfall kann *unmittelbar* auf den Backup-Server umgeschaltet werden
- beachte: Replikation sollte transparent für die Clients sein!
- Auftrag wird per Multicast an alle Server verteilt
- nur die Antwort des aktiven Servers wird zurückgeliefert

Probleme:

(vom Server und Backup-Server)

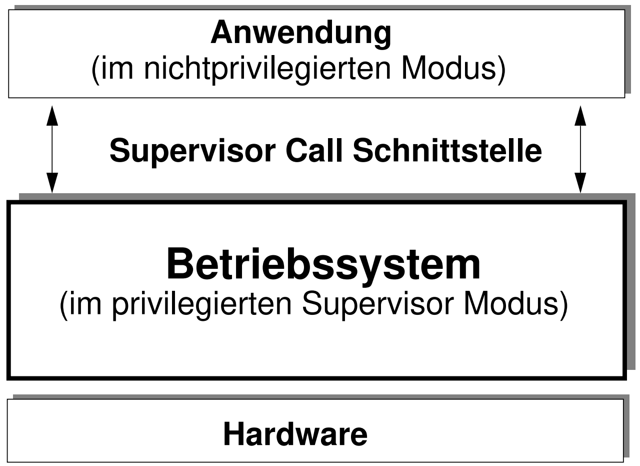
- evtl. Subaufträge werden mehrfach erteilt --> Probleme mit zustandsändernden bzw. gegenseitig ausgeschlossenen Subdiensten
- Reihenfolge der Aufträge muss bei allen Servern identisch sein (--> Semantik von multicast insbesondere bei mehreren Clients)
- Resynchronisation nach einem Crash: Nach Neustart muss ein (passiver) Server mit dem aktuellen Zustand des aktiven Servers initialisiert werden (Zustand kopieren bzw. replay)

Replikation

- Daten mehrfach halten; möglicher Zweck:
  - Effizienzsteigerung: Daten schneller verfügbar machen (z.B. Caches)
  - erhöhte Verfügbarkeit (auch bei Ausfall einzelner Server)
  - Fehlertoleranz durch Majoritätsvotum
- Forderungen: Transparenz und Erfüllung gewisser Konsistenzeigenschaften (“Kohärenz” der Replikate)
- Replikationsmanagement
  - *asynchron*: nur periodische Aktualisierung zwischen den Replikaten (inkohärente Replikate nach Änderung bis zur nächsten Synchronisation)
  - *synchron*: immer kohärente Replikate; logische Sicht eines einzelnen Servers (z.B. hot stand by)
  - Aufwand wächst, je näher man sich dem synchronen Modell annähert
- *Nicht-transparente* Replikation: Client führt Änderungen explizit auf allen Replikaten durch
- *Transparente* Replikation; unterschiedlich realisiert:
  - per Gruppenkommunikation (Semantik und Zuverlässigkeitsaspekte des Kommunikationssystems entscheidend)
  - Hauptserver (“primary”), der Sekundärserver aktualisiert
    - Schreibzugriffe nur beim Primärserver; Lesezugriffe beliebig
    - Hauptserver kann “sofort” oder schubweise (“gossip-Nachricht”) die Sekundärserver aktualisieren (Konsistenzproblematik beachten!)
  - symmetrische Server, die sich jeweils untereinander abgleichen
- Voting-Verfahren: zum Schreiben und Lesen auf jeweils mehr als 1+N/2 Server zugreifen
  - Abgleich (voting) bzgl. neuester Versionsnummer

Dienste in Betriebssystemen (1)

- Struktur eines *klassischen Betriebssystems*
    - monolithischer Aufbau

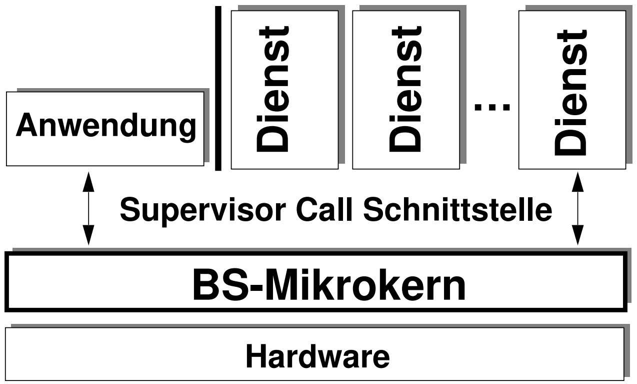


Zusätzlich:
- Dienstprogramme
- Libraries

- *Aufgaben*:
    - Verwaltung der Betriebsmittel
        - Prozessor, Speicher, Uhr, E/A-Geräte, Dateien...
    - Virtuelle Maschine für Anwendungen
        - Virtualiserung der Hardware
    - Abschottung "paralleler" Anwendungen
- Einzelne BS-Funktionen nicht isolierbar / austauschbar
- Ergänzung neuer BS-Dienste sehr aufwendig
- BS-Schicht sehr „dick“

Dienste in Betriebssystemen (2)

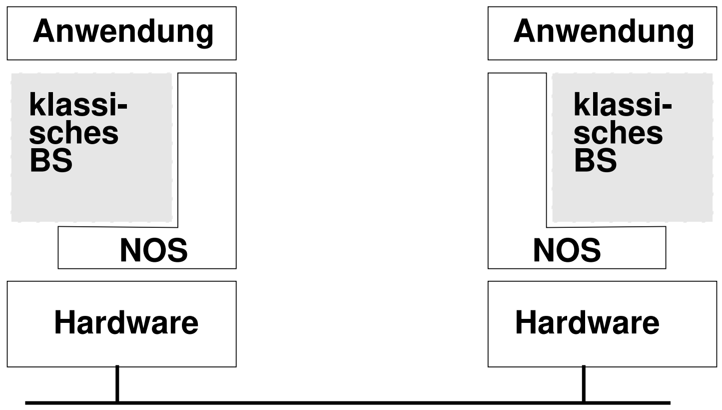
- Client/Server-orientierte, mikrokernbasierte Systeme:



- Funktionale Kapselung in "autonome" Dienstmoduln
    - z.B. Dateiverwaltung, Prozessmanagement, Kommunikation, Speichermanagement (paging), Schutz / Sicherheit (Authentisierung)...
    - klare, nachrichtenorientierte Schnittstelle zwischen den Diensten
- Mikrokern: nur noch minimale Basisfunktionalität
    - low-level I/O, basic memory management, basic interrupt handling,...
- Vorteile gegenüber monolithischer Struktur
    - bessere Wartbarkeit,Weiterentwickelbarkeit, Anpassbarkeit
    - potentiell bessere Fehlertoleranz (Ausfall eines einzelnen Dienstes gefährdet nicht unbedingt andere Dienste; redundante Dienste...)
    - im Prinzip sehr einfach verteilbar (kein wesentlicher Unterschied zwischen zentralistischer und verteilter Architektur)
- Potentielle Nachteile
    - Effizienzminderung (insbes. bei Verteilung)

Netzwerk-Betriebssysteme

- Historisch: Schritt zum verteilten Betriebssystem mittels Network Operating System (NOS)
    - “Kapselung” klassischer, nicht netzwerkfähiger Betriebssysteme



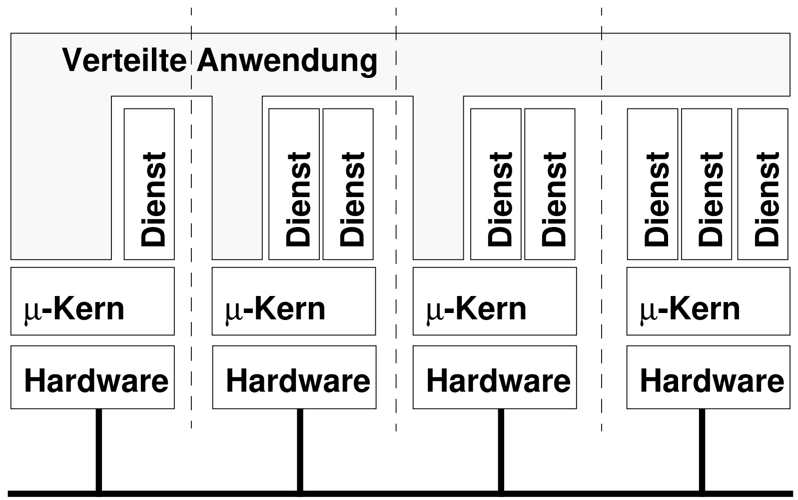
- NOS steuert Netzwerk und Netzwerkzugang, benutzt dazu u.a. Dienste des lokalen Betriebssystems
    - jeder Rechner besitzt weiterhin Kontrolle über seine Betriebsmittel
    - keine unmittelbare, gemeinsame Nutzung von Betriebsmitteln

- NOS stellt Anwendungen weitere Dienste bereit
    - file sharing, file transfer
    - Zugriffsschutz (Benutzergruppen...) ← War z.B. in UNIX als klassisches Mehrbenutzersystem bereits vorhanden!
    - Namensverwaltung
    - Netzverwaltung
    - remote login
    - E-mail

Verteilte Betriebssysteme

- Wesentliches Charakteristikum: Ortstransparenz
    - einheitliche Systemsicht für Benutzer (Anwendung; Clients)

- Menge der Dienstleistungserbringer ist räumlich verteilt
    - minimale Funktionalität pro Rechner: Mikrokern, Kommunikationsdienst



- Dienste in ihrer Gesamtheit erbringen Leistung eines einziges Betriebssystems (--> “virtueller Monorechner”)
    - Prozesse auf einem Rechner können (“ortstransparent”) einen entfernten Service nutzen, der lokal nicht angeboten wird

- Ggf. spezielle Client-Rechner: keine oder wenig globale Dienste; ggf. eingeschränkter Betriebssystemkern

- Realisierung verteilter Betriebssysteme nicht trivial
    - z.B. fehlende unmittelbare globale Sicht

Aufgaben eines verteilten Betriebssystems

- Klassische BS-Aufgaben
- Herstellung von weitgehender Ortstransparenz
- Lokalisierung von Objekten; Namensdienst
- Anbieten von gegen Fehler gesicherter Kommunikation
- Zugriffsschutz

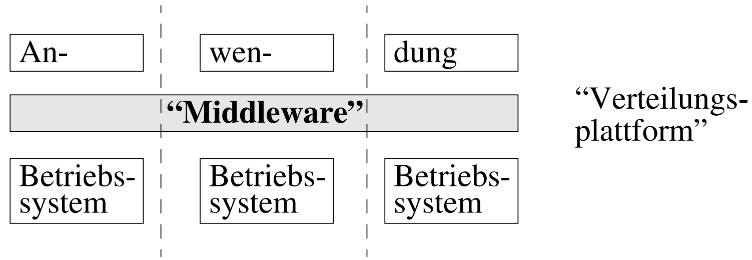
Und ggf. auch noch auf dem Wunschzettel:

- Uhrensynchronisation
- Lastverteilung
- Prozessmigration
- Konsistenzsicherung replizierter Daten
- Implementierung von “virtual shared memory”
- ...

Middleware

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

- Lösung: Zauberwort “Middleware”



- Aufgabe:
  - Verteilung (für die Anwendung) möglichst transparent machen (z.B. umspannender Namensraum, globale Zugreifbarkeit, Ortstransparenz)
  - zumindest aber die Verteilung einfach handhabbar machen

- Soll insbesondere Kommunikation und Kooperation zwischen Anwendungsprogrammen unterstützen
  - Verbergen von Heterogenität von Rechnern und Betriebsystemen (z.B. durch einheitliche Datenformate)
  - einheitliche „Umgangsformen“: Schnittstellen, Protokolle

- Sollte gewisse Basismechanismen für verteiltes Programmieren anbieten, z.B.
  - Verzeichnis- und Suchdienste (Nameservice, Tradingservices...)
  - automatische Schnittstellenanpassung (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 gleichberechtigten („peer-to-peer“-) Objekten
   - objektorientierte Schnittstellenbeschreibungssprache, Vererbung
   - Objekt Request Broker

4. Infrastruktur für spontane Kooperation (z.B. Jini)
   - unterstützt Dienstorientierung, Mobilität, Dynamik

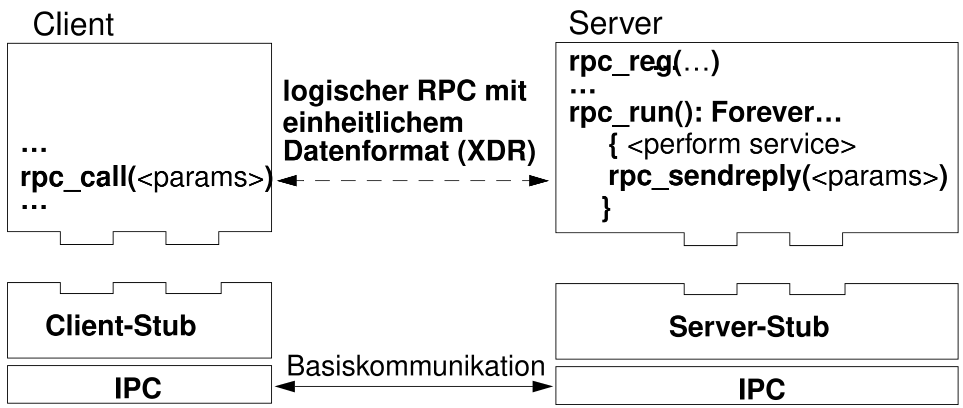
Beachte: Der Begriff “Middleware” ist leider im Laufe der Zeit zunehmend verwässert worden

- oft weniger gebraucht im technischen Sinne als Verteilungsplattform und Kommunikations- und 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 5 Mal alle 5 Sekunden wiederholt

Sun-RPC: Komponenten

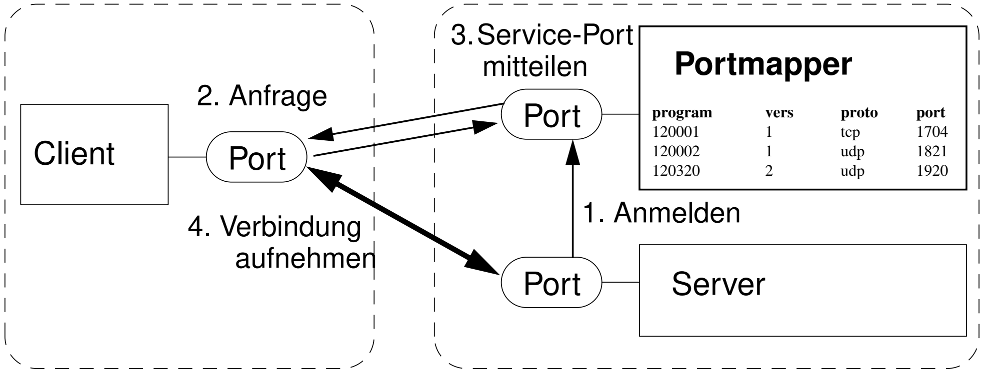
- RPC-Library: Vielzahl aufrufbarer Funktionen ("API")
  - z.B. rpc_reg, rpc_run, rpc_call
  - daneben auch Funktionen einer Low-level-Schnittstelle: z.B. Spezifikation von Timeout-Werten oder eines Authentifizierungsprotokolls
- rpcgen: Stub-Generator
- Portmapper: Zuordnung Dienstnummer <--> Portadresse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

---

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

Der Portmapper

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



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

Portmapper (2)

- Interaktive Anfrage beim Portmapper (UNIX Sun-OS)
  - shell > rpcinfo -p

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

Dynamisch generierte Port- und Programmnummern

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

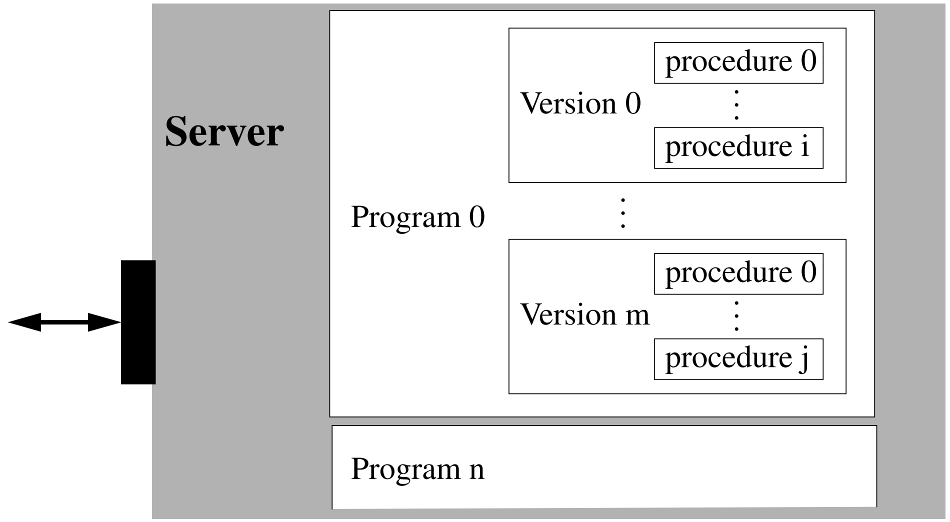
---

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

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

Service-Identifikation

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



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

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

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

Service-Registrierung

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

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

- Welche Programmnummer bekommt ein Service?

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

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

Linke Spalte: Servicename

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

Rechte Spalte: Kommentar

--> Ansonsten freie Nummer wählen:

neu und “enhanced”: “rpcb_set”

TCP oder UDP

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

Service-Aufruf

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

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

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

- Es gibt auch eine entsprechende Broadcast-Variante:

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

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

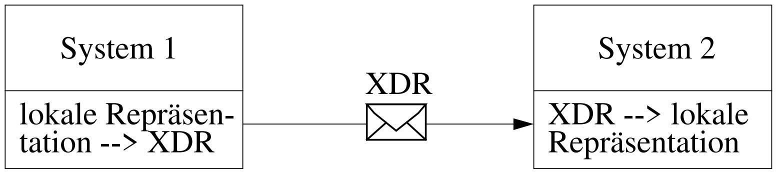
XDR (eXternal Data Representation)

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

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

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

- Prinzip der XDR-Datenkonversion:

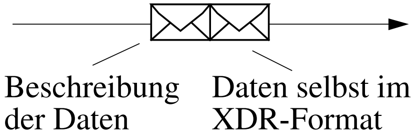


  - Beachte: Jeweils zwei Konvertierungen erforderlich; für jeden Systemtyp jeweils Kodierungs- und Dekodierungsroutinen vorsehen

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

XDR (2)

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



  - *Vorteil*: Empfänger kann Format auf Richtigkeit prüfen

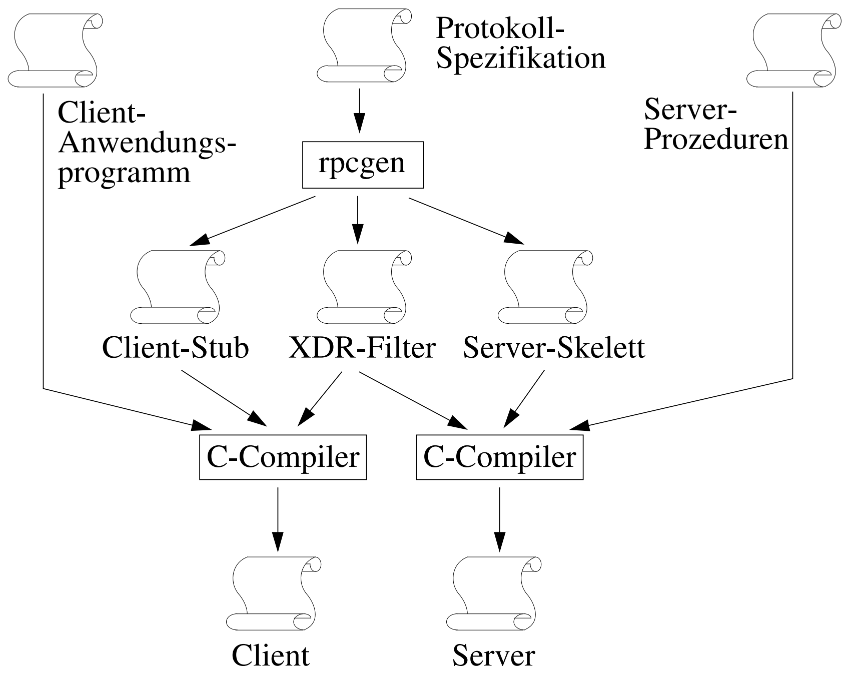
  - *Nachteil*: Grösserer Aufwand (Empfänger sollte eigentlich wissen, was für Daten er erwartet)

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

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

Stub- und Filtergenerierung

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



Beispiel zu rpcgen

Die Ausgangsdatei add.x mit der *Protokollspezifikation*:

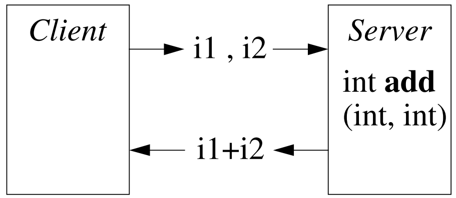
```
struct i_result
{ int x; };

struct i_param
{ int i1;
  int i2; };

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

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

Beispiel: ein “Additionsserver”:



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

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

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

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

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

Generierter Client-Code (Auszug)

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

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

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

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

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

im handle "clnt" stecken die weiteren Angaben

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

hier Server ("host") lokalisieren!

hier Parameter setzen!

eigentlicher Prozeduraufruf

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

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

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

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

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

Generierter Server-Code (Auszug)

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

svc_run();
```

svc_reg funktioniert analog zu rpc_reg

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

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

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

Bem.: Server-Code ist über 200 Zeilen lang

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

---

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

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

Generierte XDR-Konversionsroutinen

```
...
bool_t xdr_i_result(xdrs, objp)
 XDR *xdrs;
 i_result *objp;
{
 if (!xdr_int(xdrs, &objp->x)) return (FALSE);
 return (TRUE);
}


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


Sicherheitskonzept des Sun-RPC

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

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

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

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

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

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

  - *SYS:* Authentifizierung im UNIX-Stil

  - *DES:* echte Authentifizierung (“Secure RPC”)

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

SYS-Flavor bei Sun-RPC

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

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

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

Effektive user-id des Client

Effektive Gruppen-id

Weitere Gruppen, in denen der Client Mitglied ist

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

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

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

Secure RPC mit DES

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

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

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

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

- Server überprüft jeweils, ob:

    (a) Zeitstempel grösser als letzter Zeitstempel

    (b) Zeitstempel innerhalb des Zeitfensters

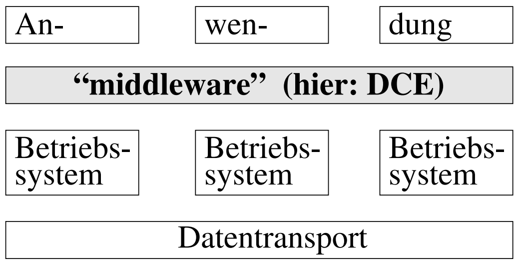
- Die Antwort des Servers enthält (verschlüsselt) den letzten erhaltenen Zeitstempel-1 (--> Authentifizierung!)

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

DCE - Distributed Computing Environment

- Entwickelt von einem herstellerübergreifendes Konsortium (“OSF” - Open Software Foundation)
  - Anfang der 90er Jahre, u.a. DEC, IBM, Siemens, HP...
  - trotz CORBA noch vielfältig eingesetzt in grossen Organisationen

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



- Ziel: Schaffung eines Industriestandards für verteilte Verarbeitung

- Vorgehensweise pragmatisch: Soweit möglich, Nutzung geeigneter existierender Technologiekomponenten

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

Offene Systeme

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

---

Vorteile für den Nutzer:

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

Standardisierung daher notwendig; z.B. Schnittstellen

- zur Systemumgebung
- zwischen (unabhängigen) Systemkomponenten
- zum Benutzer

--> Konsortien: Empfehlungen, Richtlinien...

--> internationale Standardisierungsorganisationen (z.B. ISO; ITU): Normen, Standards

Konsortien (für offene Systeme)

- Non-profit-Vereinigungen von Herstellern
    - ggf. auch unter Einbeziehung von Nutzern

- Sicherstellung von Plattformunabhängigkeit, Portabilität und Interoperabilität von Systemen
    - Normen, Testszenarien, Zertifizierungsgremien...
    - Arbeitsgruppen zu technischen Fragen und Standards
    - Richtlinien für unabhängige Softwareentwickler
    - Verpflichtung einzelner Mitglieder, gemeinsame Entscheidungen zu unterstützen

- Auswahl, Entwicklung, Anpassung, Zusammenstellung von “Technologiekomponenten”
    - z.B. Betriebssystem, Protokolle, Benutzeroberflächen, Programmbibliotheken, Management-Tools, Entwicklungssysteme...

- Etablierung oft aus marktstrategischen Gründen
    - Investitionsschutz (Entwicklungskosten, Lizenzen...)
    - Vermeidung von Mehrfachentwicklung
    - strategisch / politische Erwägungen (z.B. EU-Richtlinien)
    - Durchsetzen von Normen
    - beschleunigte Produktentwicklung durch strategische Allianzen
    - Gegenallianzen zu Monopolisten

Konsortien...

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

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

Common Object Request Broker Architecture

- Mehr als 500 Mitglieder (HP, ATT, Sun...)

- Ziel: Bereitstellung von Konzepten für die Entwicklung verteilter Anwendungen mit objektorientierten Modellen

- OMA (Object Management Architecture) und OSA (Object Services Architecture): Dienste zur Verwaltung von Objekten in verteilten heterogenen Systemen (z.B. Trading, verteilte Transaktionen, Replikation, Speicherung, Namensverwaltung, Persistenz, Migration, Security...)

DCE - Der Hoffnungsträger (1993)

DCE
mit InterFace

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

Die InterFace Computer GmbH hat sich schon während der Entwicklungsphase intensiv mit der DCE-Technologie beschäftigt. Früher als andere sind wir deshalb in der Lage, den neuen Standard in praktische Lösungen umzusetzen – im Hinblick auf Systemberatung, Integration bestehender oder Entwicklung neuer Systeme ebenso wie durch eigens entwickelte Schulungsangebote. So stehen Ihnen von Anfang an alle durch DCE ermöglichten Vorteile zur Verfügung: Höhere Effizienz durch verteilte Programme; die automatische Zuteilung freier Rechnerkapazitäten; das globale Zusammenspiel all Ihrer Hard- und Softwarekomponenten in einem offenen System. Die Komplexität des Netzes bleibt dabei für Anwender unsichtbar; der integrierte Security Service gewährleistet Datenschutz und Datensicherheit auf allen Ebenen und in allen Betriebszuständen.

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

iF

InterFace Computer GmbH
[illegible] Straße 4
D-8000 München 2
Tel.: 089/51030-0
Fax: 089/51030-23

All together **now.**

Werbesprüche...

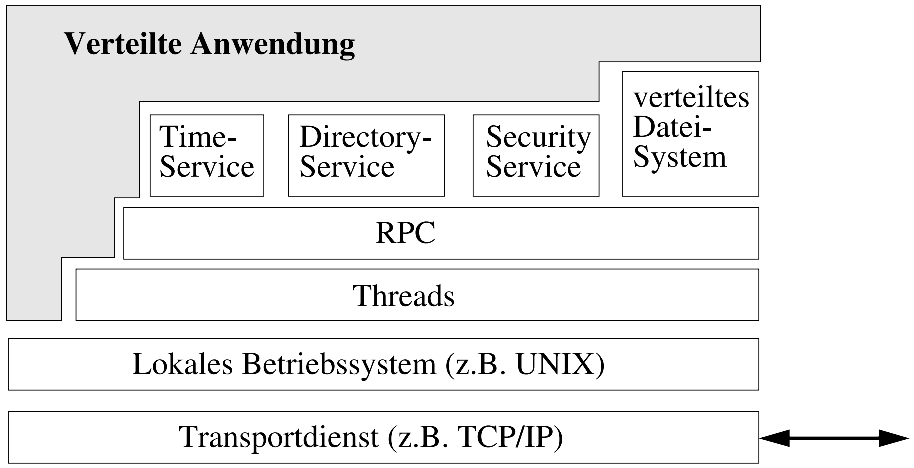
Mit DCE (Distributed Computing Environment) existiert endlich ein internationaler, von der OSF entwickelter Standard für die intelligente Integration unterschiedlicher Computerplattformen – vom PC über UNIX-Netze bis hin zu Mainframes oder externen Rechenzentren. An die Stelle eines Nebeneinanders tritt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

Die InterFace Computer GmbH hat sich schon während der Entwicklungsphase intensiv mit der DCE-Technologie beschäftigt. Früher als andere sind wir deshalb in der Lage, den neuen Standard in praktische Lösungen umzusetzen – im Hinblick auf Systemberatung, Integration bestehender oder Entwicklung neuer Systeme ebenso wie durch eigens entwickelte Schulungsangebote. So stehen Ihnen von Anfang an alle durch DCE ermöglichten Vorteile zur Verfügung: Höhere Effizienz durch verteilte Programme; die automatische Zuteilung freier Rechnerkapazitäten; das globale Zusammenspiel all Ihrer Hard- und Softwarekomponenten in einem offenen System. Die Komplexität des Netzes bleibt dabei für Anwender unsichtbar; der integrierte Security Service gewährleistet Datenschutz und Datensicherheit auf allen Ebenen und in allen Betriebszuständen.

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

now.

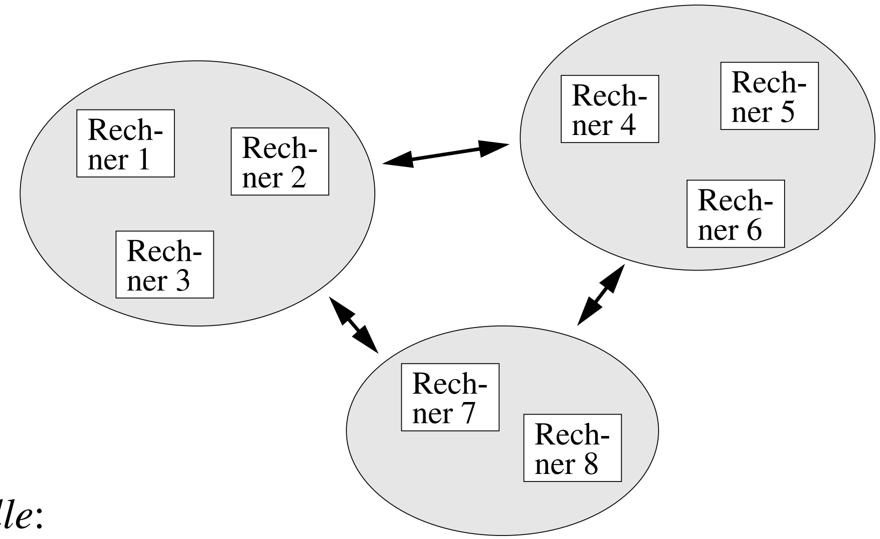
Hauptkomponenten des DCE



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

Globale DCE-Architektur: Zellen

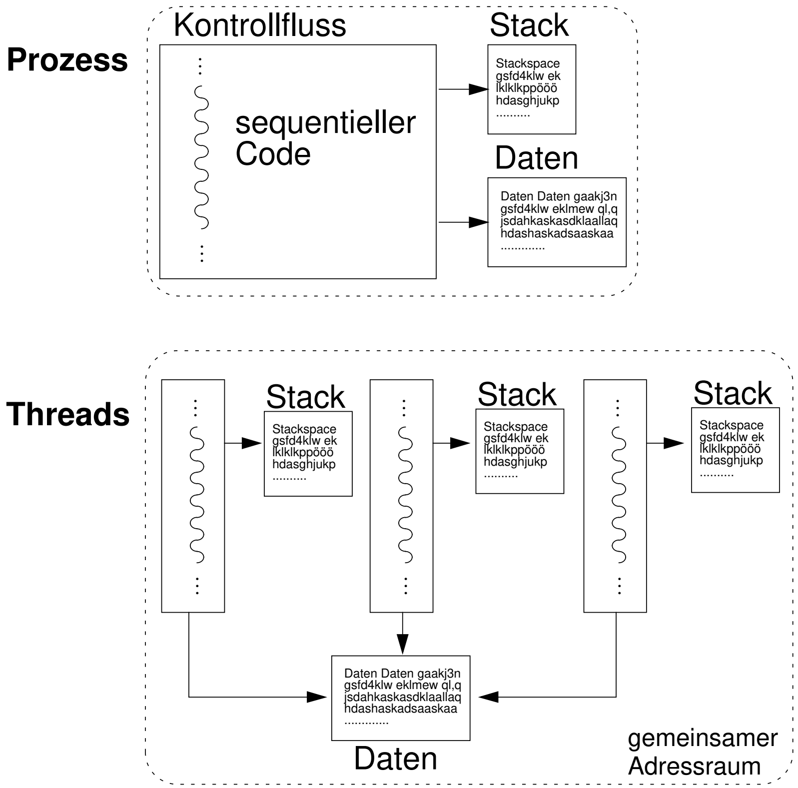
- Partitionierung der Rechner in sogen. *Zellen*
  - Subsysteme machen grosse Systeme handhabbarer



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

DCE: Threads

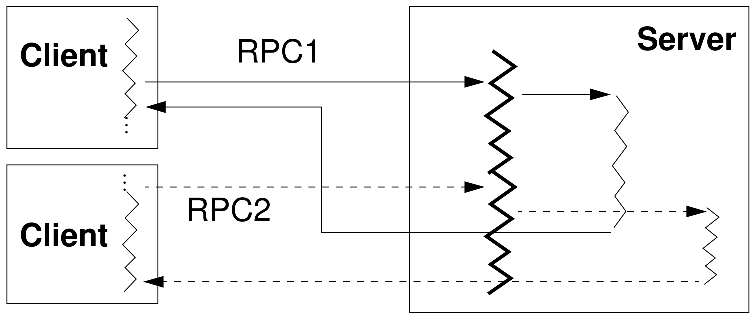
- Threads = leichtgewichtige Prozesse mit gemeinsamem Adressraum



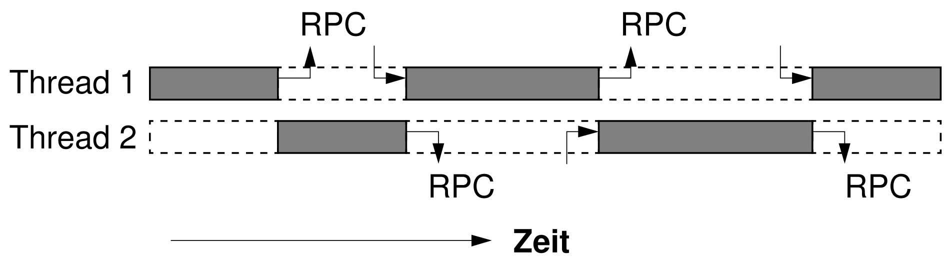
- Einfache Kommunikation zwischen Kontrollflüssen
  - aber: kein gegenseitiger Schutz; ggf. Synchronisation bzgl. Speicher
- Thread hat weniger Zustandsinformation als ein Prozess
- Kontextwechsel i.a. wesentlich schneller
  - kein Umschalten des Adressraumkontexts
  - Cache und Translation Look Aside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

Wozu Multithreading bei Client-Server-Middleware?

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



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



DCE-Threadkonzept

- Thread-Konzept basiert auf *POSIX-Standard* 1003.4a

- Grössere Zahl von *C-Bibliotheksfunktionen*
  - Erzeugen, Löschen von Threads
  - Synchronisation durch globale Sperren, Semaphore, Bedingungsvariablen
  - warten eines Threads auf ein Ereignis eines anderen Threads
  - wechselseitiger Auschluss mehrerer Threads (“mutex”)
  - nebenläufige Signalverarbeitung und Ausnahmebehandlung

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

Problematik von DCE-Threads

- Aufrufe des Betriebssystem-Kerns sind i.a. problematisch

  a) *nicht ablaufinvariante* (“non-reentrant”) Systemroutinen
  - interne Statusinformation, die ausserhalb des Stacks der Routine gehalten wird, kann bei paralleler Verwendung überschrieben werden
  - z.B. printf: ruft intern Speichergenerierungsroutine auf; diese benutzt prozesslokale Freispeicherliste, deren “gleichzeitige” nicht-atomare Manipulation zu Fehlverhalten führt
  - “Lösung”: Verwendung von “Jacket-Routinen” (wrapper), die gefährdete Routinen kapseln und Aufrufe wechselseitig ausschliessen

  b) *blockierende* (“synchrone”) Systemroutinen
  - z.B. synchrone E/A, die alle Threads des Prozesses blockieren würde statt nur den aufrufenden Thread
  - “Lösung”: Verwendung asynchrone Operationen zum Test auf mögliche Blockaden

---

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

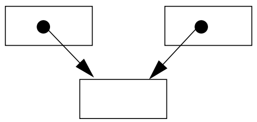
DCE-RPC

- Weder kompatibel zu Sun-RPC noch zur OSI-Norm

- Ein- und Ausgabeparameter: out, in, in/out

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

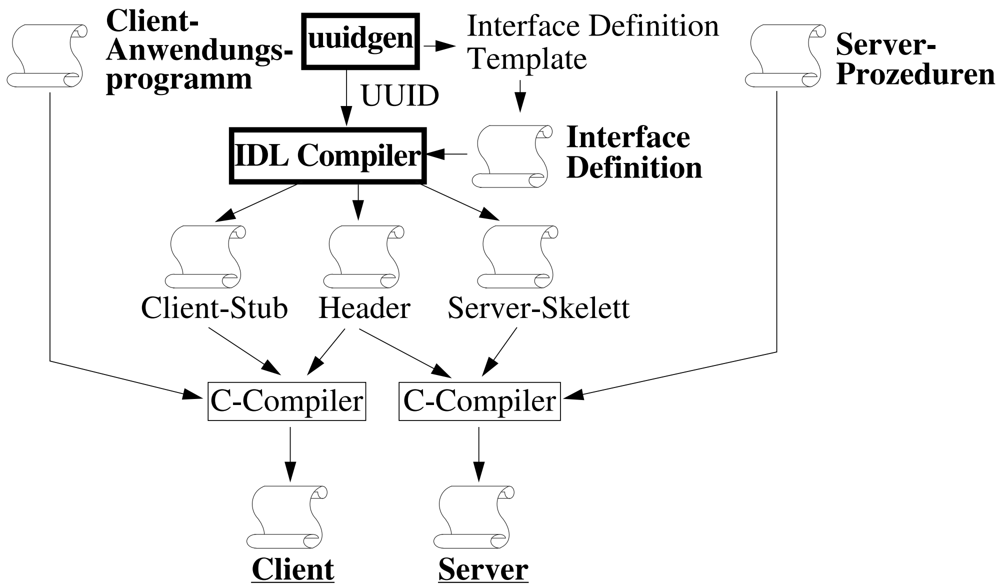
müsste “remote” interpretiert werden



- 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

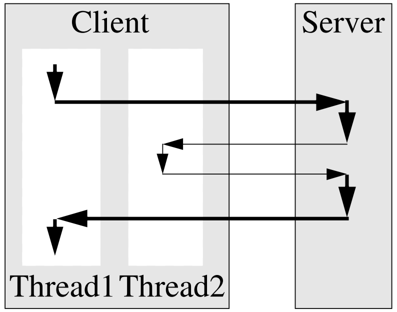


- UUID (Universal Unique Identifier) ist eine aus Uhrzeit und Rechnerkennung generierte systemweit eindeutige Kennung der Schnittstelle

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, welches Protokoll verwendet werden soll
  - Angabe, wie viele Aufrufe serverseitig gepuffert werden sollen

- Schliesslich teilt der Dienst dem RPC-Laufzeitsystem mit, dass er bereit ist, Aufrufe entgegenzunehmen (“listen”)
  - Angabe, wieviele Aufrufe maximal gleichzeitig bearbeitet werden können --> automatisches Erzeugen von Threads

Bindevorgang beim DCE-RPC

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

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

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

---

- Statt des o.g. *automatischen Bindens*, das für den Client transparent abläuft, ist auch *explizites Binden* möglich:
  - umständlicher, aber flexibler
  - z.B. programmierte Auswahl eines Backup-Servers, wenn Bindevorgang mit Primärserver unmöglich
  - z.B. explizite Auswahl eines Servers einer Gruppe (Lastausgleich etc.)

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

DCE-RPC: Semantik

- Semantik für den *Fehlerfall* ist wählbar:

  (a) *at most once*
  - bei temporär gestörter Kommunikation wird Aufruf automatisch wiederholt; eventuelle Aufrufduplikate werden gelöscht
  - Fehlermeldung an Client bei permanentem Fehler
  - ist default

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

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

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

DCE: Sicherheit

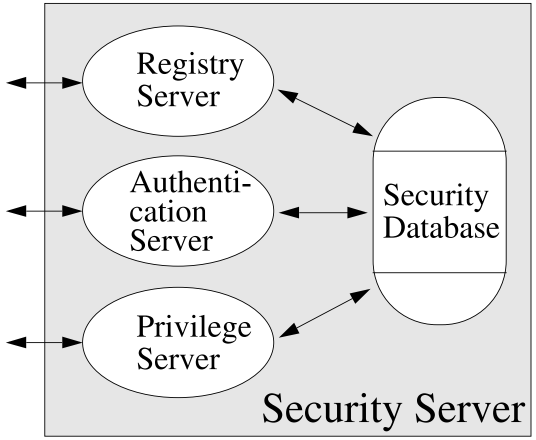
- Verwendung des Kerberos-Protokolls
  - Vertraulichkeit durch Sitzungsschlüssel (--> DES)
  - gegenseitige Authentifizierung
  - selektive Autorisierung von Clients für bestimmte Dienste
  - Schlüsselverwaltung
  - zusätzlich (ab Version 1.2.2) auch asymmetrische Verfahren

- 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



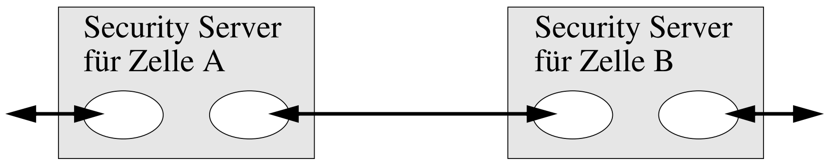
- *Registry Server:* Verwaltung von Benutzerrechten; Dienste für Systemverwaltung

- *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 ("Surrogate") vereinbart

Weitere DCE-Komponenten

- Cell Directory Service (CDS)
  - realisiert Zuordnung von Namen und Adressen
  - verwaltet Namen (mit Attributen) einer Zelle
  - Beispiel für Attribute: *Druckername*, *Standort*, *Art* für einen Drucker (mit spezifischen Werten z.B. *pr99*, *Raum7*, *color600dpi*)
  - Replikation (zwecks Fehlertoleranz) möglich (dabei "Konvergenzlevel" einstellbar)

Namensverwaltung

- Global Directory Service (GDS)
  - Bindeglied zwischen verschiedenen CDS
  - hierarchischer Namensraum
  - Namenformat basiert auf X.500 oder DNS

- 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