Beispiele für Namensdienste

- Domain Name System (DNS) im Internet
  - in der UNIX-Welt oft eingesetzte Implementierung: BIND ("Berkeley Internet Domain Name")
- LDAP (Lightweight Directory Access Protocol)
- Network Information Service (NIS)
  - entwickelt von Sun Microsystems
  - hauptsächlich zur Verwaltung von Dateizugriffsrechten in lokal vernetzten Systemen
  - später erweitert zur Verwaltung von Benutzern, Passwörtern, Diensten...
  - basiert auf RPC
  - nutzt Primär- / Sekundärserverprinzip ("Master" / "Replica Server")
- Portmapper für TCP- oder UDP-basierte Dienste
  - eher rudimentär; nicht verteilt
- Lookup-Service ("LUS") bei Jini und ähnlichen Systemen

Internet Domain Name System (DNS)

- Jeder Rechner im Internet hat eine IP-Adresse
  - 32 Bit lang (bei IPv4), typischerweise als 4 Dezimalzahlen geschrieben
  - Bsp.: 192.130.10.121 (= 11000000.10000010.00001010.01111001)
- Symbolische Namen sind für Menschen geeigneter
  - z.B. Domain-Namen wie www.nanocomp.uni-cooltown.eu
  - gut zu merken; relativ unabhängig von spezifischer Maschine
  - muss *vor* Verwendung bei Internet-Diensten (WWW, E-Mail, ssh, ftp,...) in eine IP-Adresse umgesetzt werden
  - Umsetzung in IP-Adresse geschieht im Internet mit DNS
- Domains
  - hierarchischer Namensraum der symbolischen Namen im Internet
  - "Toplevel domains" com, de, fr, ch, edu,...
  - Domains (ggf. rekursiv) gegliedert in Subdomains, z.B.

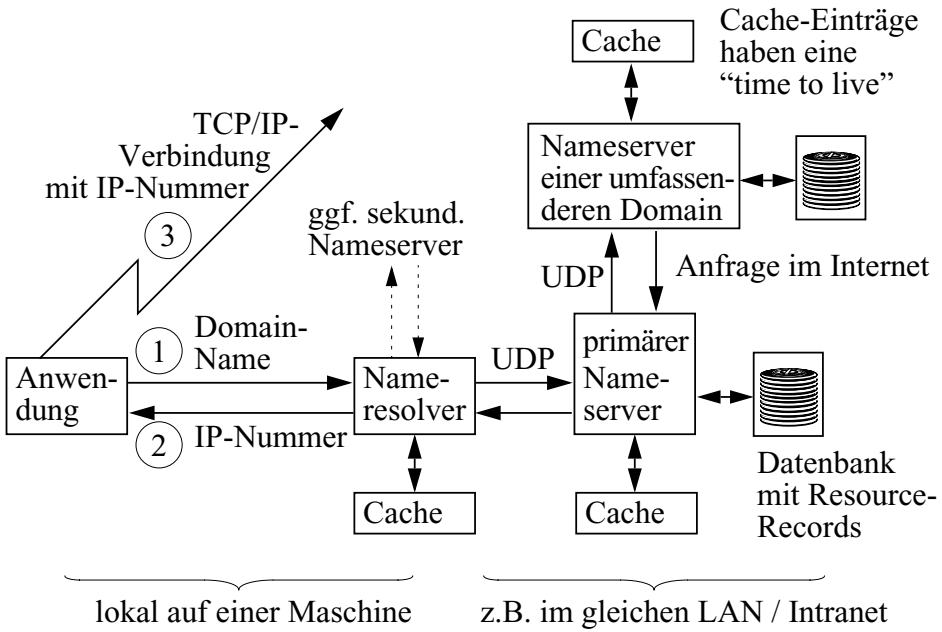
eu
uni-cooltown.eu
informatik.uni-cooltown.eu
nano.informatik.uni-cooltown.eu
pc6.nano.informatik.uni-cooltown.eu

- Für einzelne (Sub)domains bzw. einer Zusammenfassung einiger (Sub)domains (sogenannte "Zonen") ist jeweils ein Domain-Nameserver zuständig
  - primärer Nameserver (www.switch.ch für die Domains .ch und .li)
  - optional zusätzlich einige weitere sekundäre Nameserver
  - oft sind Primärserver verschiedener Zonen gleichzeitig wechselseitig Sekundärserver für die anderen
  - Nameserver haben also nur eine Teilsicht!

Namensauflösung im Internet

- Historisch: Jeder Rechner hatte eine Datei hosts.txt, die jede Nacht von zentraler Stelle aus verteilt wurde

- Später: lokaler Nameresolver mit einer Zuordnungsdatei /etc/hosts für die wichtigsten Rechner, der sich ansonsten an einen seiner nächsten Nameserver wendet
  - IP-Nummern der "nächsten" Nameserver stehen in lokalen Systemdateien



- Sicherheit und Verfügbarkeit sind wichtige Aspekte
  - Verlust von Nachrichten, Ausfall von Komponenten etc. tolerieren
  - absichtliche Verfälschung, denial of service etc. verhindern

Resource Records

- Datenbank eines DNS-Nameservers besteht aus einer Menge von Resource-Records, z.B.:

```
fb22.tu-da.de          IN  SOA ...
sys1.fb22.tu-da.de     IN  A 130.83.200.63
sys1.fb22.tu-da.de     IN  A 130.83.253.12
www.fb22.tu-da.de      IN  CNAME  robin.fb22.tu-da.de
ftp.fb22.tu-da.de      IN  CNAME  robin.fb22.tu-da.de

fb23.tu-da.de          IN  NS 130.83.193.77

boss                   IN  A 130.83.200.17
helga                  IN  A 130.83.200.39
laser-printer          IN  A 130.83.201.75
```

symbolischer Name — IP-Adresse (bei A-records)

- Verschiedene Record-Formate, z.B.:
  - A für "Address"
  - SOA ("Start of Authority"): Parameter zur Zone (z.B. für Caching etc.)
  - CNAME ("Canonical Name"): für Spezifikation eines Alias
  - NS: Nameserver für eine Subdomain

- Einige weitere Angaben stehen in anderen Dateien, z.B.:
  - IP-Adresse der übergeordneten Nameserver
  - ob Primär- oder Sekundärserver etc.

nslookup

nslookup - query name servers interactively

nslookup is an interactive program to query Internet domain name servers. The user can contact servers to request information about a specific host, or print a list of hosts in the domain.

```
> pc20
Name:    pc20.nanocomp.inf.ethz.ch
Address:  129.132.33.79
Aliases:  ftp.nanocomp.inf.ethz.ch

> google.com
Name:    google.com
Addresses:  216.239.57.104,
216.239.59.104, 216.239.39.104

> google.com
Name:    google.com
Addresses: 216.239.59.104,
216.239.39.104, 216.239.57.104

> cs.uni-sb.de
Name:    cs.uni-sb.de
Addresses:  134.96.254.254, 134.96.252.31
```

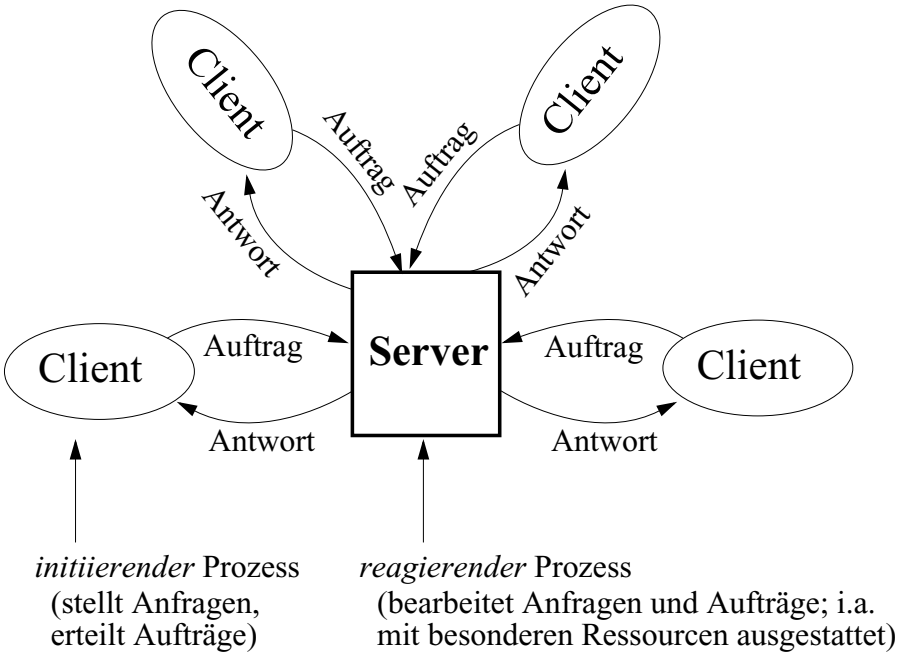
Dies deutet auf einen "round robin"-Eintrag hin: Der Nameserver von google.com ändert alle paar Minuten die Reihenfolge der Einträge, die bei anderen Nameservern auch nur einige Minuten lang gespeichert bleiben dürfen. Da Anwendungen i.a. den ersten Eintrag nehmen, wird so eine Lastverteilung auf mehrere google-Server vorgenommen.

Router an zwei Netzen

Note: nslookup is deprecated and may be removed from future releases. Consider using the 'dig' or 'host' programs instead.

Client/Server-Modell

Das Client/Server-Modell



- Aufgabenteilung und asymmetrische Struktur
  - *Clients*: typischerweise Anwendungsprogramme und graphische Benutzungsschnittstelle (“front end”) für einen Nutzer
  - *Server*: zuständig für Dienstleistungen für viele Clients
- Typisches Kommunikationsparadigma: RPC

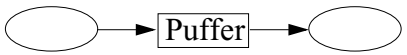
Eignung des Client/Server-Paradigmas

- Populär wegen des eingängigen Modells
  - entspricht Geschäftsvorgängen in unserer Dienstleistungsgesellschaft
  - gewohntes Muster → intuitive Struktur, gute Überschaubarkeit
- Effizienz durch spezialisierte „Dienstleister“
  - grosszügige Ausstattung (CPU-Leistung, Speicherkapazität usw.)
  - bestückt mit spezieller Software (Datenbank etc.)
- Kosteneffektivität durch bessere Auslastung wertvoller Ressourcen (z.B. bei “Compute Server”)
  - Clients brauchen oft kurzfristig Spitzenleistung
  - einzelner Client kann Ressourcen aber nicht dauerhaft auslasten
- Passend für viele Kooperationsbeziehungen, z.B.
  - Client erbittet Auskunft von einem spezialisierten Service
  - gefährdete Clients geben wertvolle Daten in Obhut des (gegen Missbrauch, Verlust, 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:

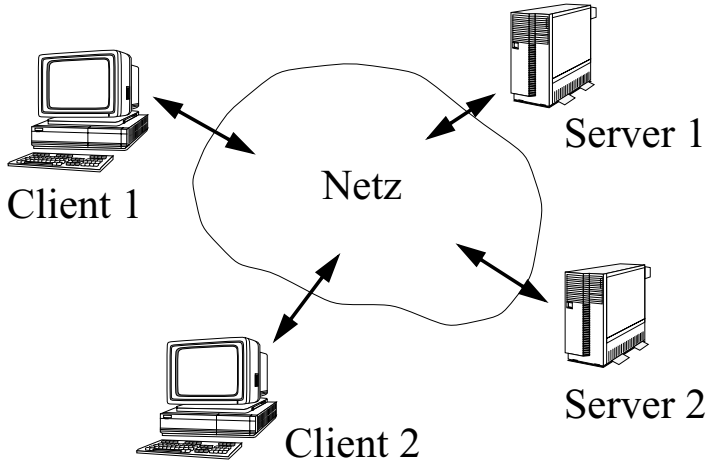


Produzent Konsument

- 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

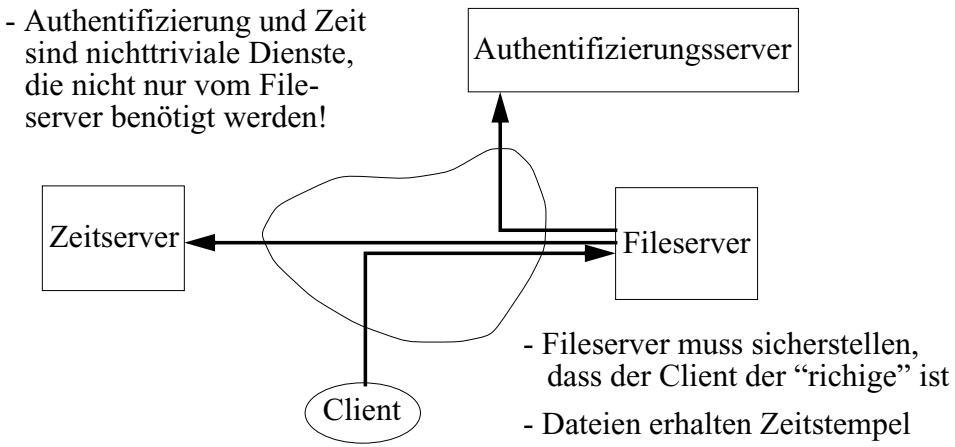
- Übertragung des Client/Server-Modells auf *Rechner*



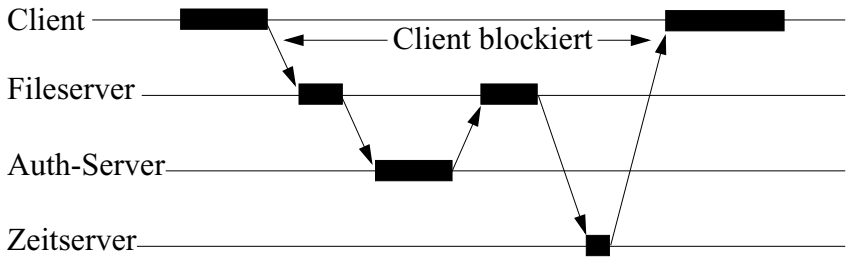
- Typischerweise PCs als Clients
  - u.a. mit graphischem Benutzungsinterface
- Andere, leistungsfähigere Rechner als Server
  - “zentrale” Dienste (z.B. Speicherserver)
  - gemeinsam benutzte Betriebsmittel
- Im allgemeinen müssen sich aber Server- und Client-*Prozesse* nicht auf dedizierten Rechnern befinden!

Client/Server-Rollen

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



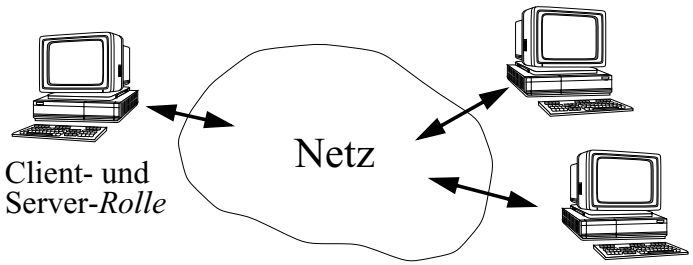
- Fileserver hat prinzipiell die *Rolle* eines Servers, zwischenzeitlich jedoch die *Rolle* eines Clients



Peer-to-Peer-Strukturen

"Gleichrangiger"

- Im Gegensatz zum asymmetrischen 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
- In der Reinform keine zentralisierten Elemente
  - dies wird gelegentlich in "politischer" Weise artikuliert (vgl. Tauschbörsen)

- *Nachteile*:
  - "Anarchischer" als *maschinenbezogene* Client/Server-Architektur
  - Rechner müssen leistungsfähig genug sein (cpu-Leistung, Speicher-ausbau), 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...

Zu vielen Aspekten von Peer-to-Peer-Systemen: R. Steinmetz, K. Wehrle (Eds): *Peer-to-Peer Systems and Applications*, Springer-Verlag, 2005

Zustandsändernde /-invariante Dienste

- Verändern Aufträge den Zustand des Servers wesentlich?
- Typische *zustandsinvariante* Dienste:
  - Auskunftsdienste (z.B. Name-Service)
  - 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)
  - vgl. auch frühere Diskussion bzgl. RPC-Fehlersemantik!

Zustandslose / -behaftete Server

stateless — statefull

“session”

- Hält der Server Zustandsinformation über Aufträge hinweg?
  - 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, z.B. über die Position des Dateizeigers geöffneter Dateien

  - 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!

- File-Server wurden sowohl zustandslos (NFS) als auch zustandsbehaftet (RFS) realisiert

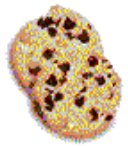
Sind Webserver zustandslos?

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

Lösungsmöglichkeiten (Korrelation von Web-Seiten zur Realisierung von “Warenkörben” im WWW):

- IP-Adresse des Kunden an Auftrag anheften?
  - Problem: Proxy-Server → viele Kunden haben gleiche IP-Adresse
  - Problem: dynamische IP-Adressen → keine Langzeitwiedererkennung
- “URL rewriting” und dynamische Web-Seiten
  - Einstiegsseite eine eindeutige Nummer anheften, wenn der Kunde diese erstmalig aufruft
  - diese Nummer jedem link der Seite anheften und mit zurückübertragen
- Cookies
  - kleine Textdatei, die ein Server einem Browser (= Client) schickt und die im Browser gespeichert wird
  - nur der Sender des Cookies kann 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 its expiration date arrives if the number of cookies exceeds its internal limits.

Cookies (2)

- Anwendung von cookies ist teilweise umstritten (Ausspionieren des Verhaltens); dazu kam zeitweise 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.

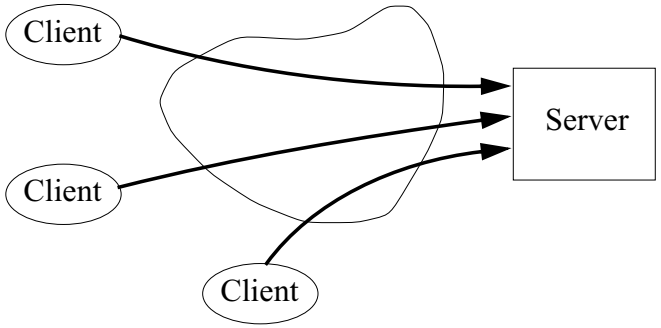
- Ganz problemlos ist das allerding nicht:

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

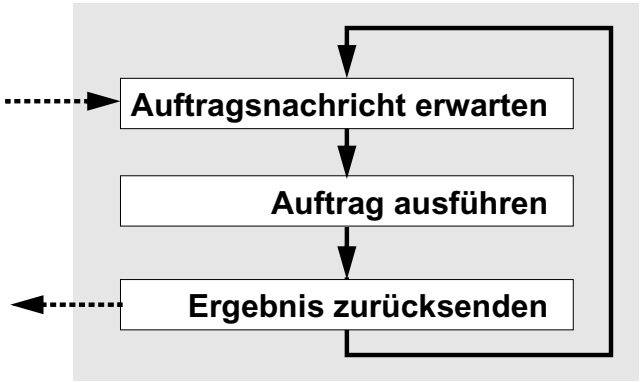
  Unfortunately, the original intent of the cookie has been subverted by some unscrupulous entities who have found a way to use this process to actually track your movements across the Web. They do this by surreptitiously planting their cookies and then retrieving them in such a way that allows them to 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.

Gleichzeitige Server-Aufträge?

- Problem: Oft viele “gleichzeitige” Aufträge



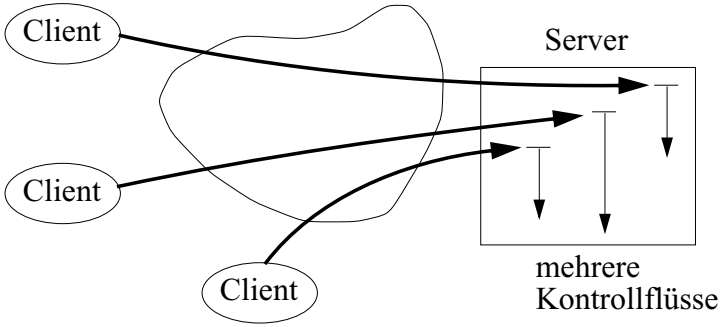
- *Iterative Server* bearbeiten nur einen Auftrag pro Zeit



- “single threaded” (nur ein einziger Kontrollfluss)
- eintreffende Anfragen während Auftragsbearbeitung: abweisen, puffern oder einfach ignorieren
- einfach zu realisieren
- bei “trivialen” Diensten sinnvoll (mit kurzer Bearbeitungszeit)

Konkurrente (“nebenläufige”) Server

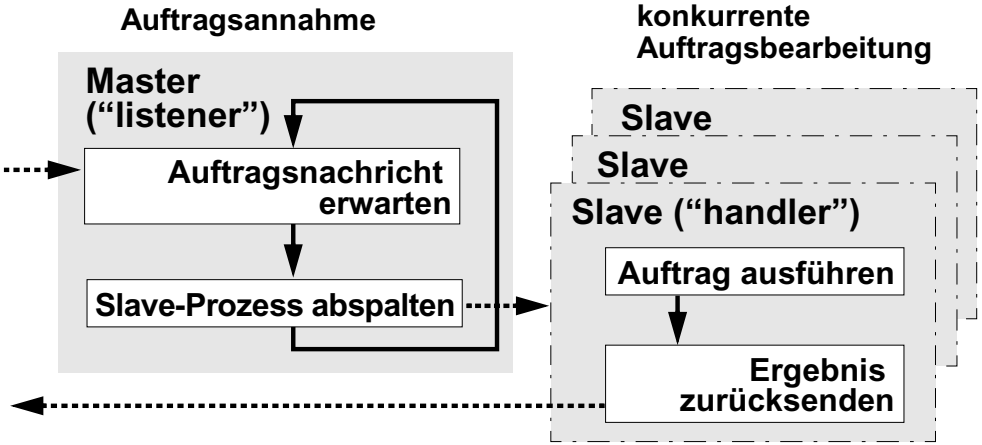
- Gleichzeitige Bearbeitung mehrerer Aufträge
  - sinnvoll (d.h. effizienter für Clients) bei langen Aufträgen (z.B. in Verbindung mit E/A)
  - Beispiel: Web-Server oder Suchmaschinen



- Ideal bei physischer Parallelität (z.B. multi-core)
  - 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 bzw. Multicore-Prozessoren
  - Verbund verschiedener Server-Maschinen (Server-Farm, -Cluster)
  - dynamische Prozesse (bei Monoprozessor-Systemen)
  - dynamische threads
  - feste Anzahl vorgegründeter Prozesse

Konkurrente Server mit dynamischen Handler-Prozessen

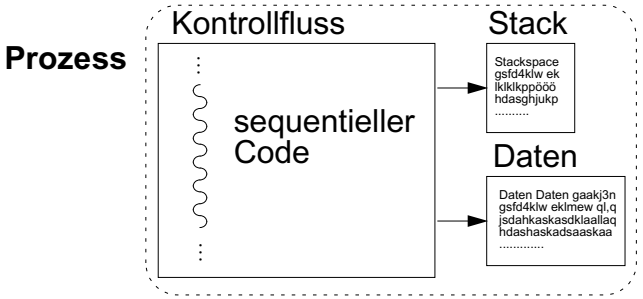
- Für jeden Auftrag gründet der *Master* einen neuen *Slave*-Prozess und wartet dann auf einen neuen Auftrag
  - neu gegründeter Slave ("handler") übernimmt den Auftrag
  - Client kommuniziert dann direkt mit dem Slave (z.B. über dynamisch eingerichteten Kanal bzw. Port)
  - Slaves sind ggf. Leichtgewichtsprozesse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

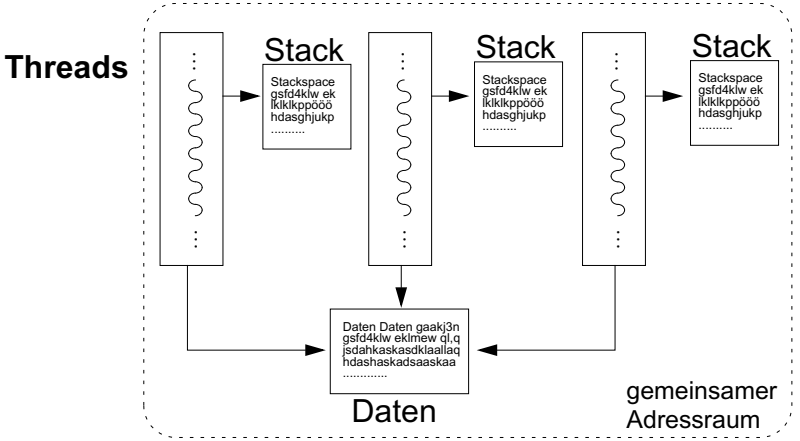


- Alternative: "Process preallocation": Feste Anzahl statischer Slave-Prozesse
  - ggf. effizienter (u.a. Wegfall der Erzeugungskosten)
- Übungsaufgaben:
  - herausfinden, wie es bei Web-Servern konkret gemacht wird
  - wie sollte man bei Internet-Suchmaschinen vorgehen?

Threads

- Threads = leichtgewichtige Prozesse mit gemeinsamem Adressraum

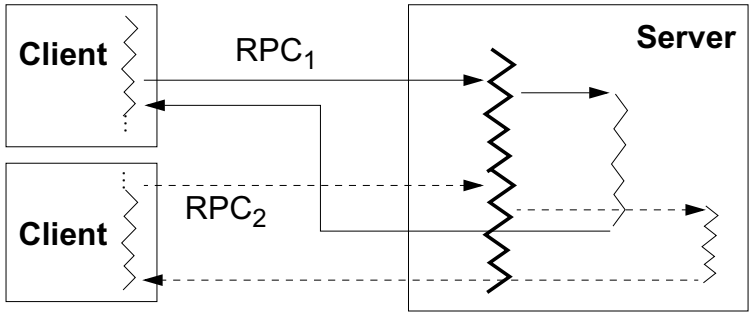




- Einfache Kommunikation zwischen Kontrollflüssen
  - aber: kein gegenseitiger Schutz; ggf. Synchronisation bzgl. Speicher
- Thread hat weniger Zustandsinformation als ein Prozess
- Kontextwechsel daher i.a. wesentlich schneller
  - kein Umschalten des Adressraumkontexts
  - Cache und Translation Look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

Wozu Multithreading bei Client-Server-Middleware?

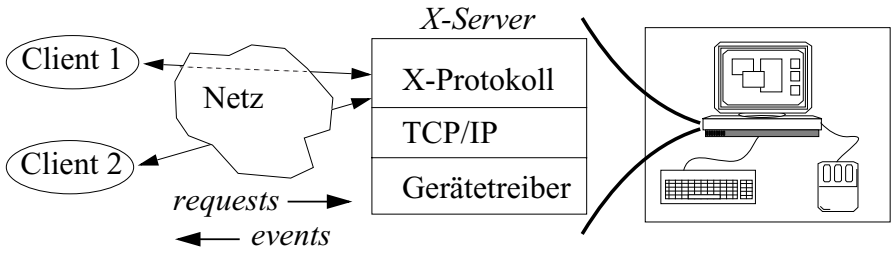
- *Server*: quasiparallele Bearbeitung von Aufträgen
  - Server bleibt ständig empfangsbereit



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

“X-Window” als Client/Server-Modell

- Erstes internetbasiertes Graphik- und Fenstersystem für seinerzeit neue pixelorienterte Displays
  - entwickelt Mitte der 1980er Jahre am MIT (zusammen mit DEC und IBM)



- i.a. bedient ein Server mehrere Client-Prozesse (“Applikationen”), die ihre Ausgabe auf dem gleichen Display erzeugen
- *Window-Manager*: Spezieller Client, der Grösse und Lage der Fenster und Icons steuert (fungiert als “Bedienoberfläche”)

→ X windows system protocol (über TCP)

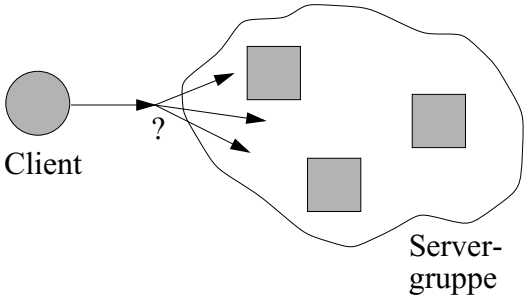
- *Requests*: Service-Anforderung an den X-Server (z.B. Linie in einer bestimmten Farbe zwischen zwei Koordinatenpunkten zeichnen); zugehörige Routinen stehen in einer Bibliothek (*Xlib*)
- *X-Library* (Xlib) ist die Programmierschnittstelle zum X-Protokoll; damit manipuliert ein Client vom Server verwaltete 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
- *X-Terminal* hatte Server-Software im ROM bzw. beim Booten geladen (heute gegenüber PC preislich kaum ein Vorteil, vgl. auch “Web-Terminal”)
- 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*

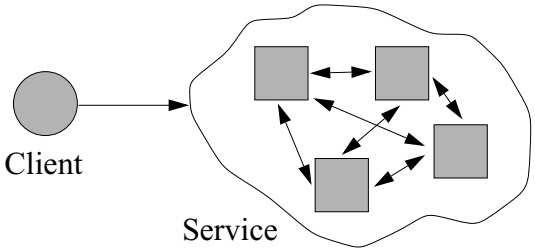
- Jeder einzelne Server kann den Dienst erbringen
- Zweck:
  - *Leistungssteigerung* (Verteilung der Arbeitslast auf mehrere Server) ← "Lastverbund"
  - *Fehlertoleranz* durch Replikation (Verfügbarkeit auch bei vereinzelten Server-Crashs) ← "Überlebensverbund"



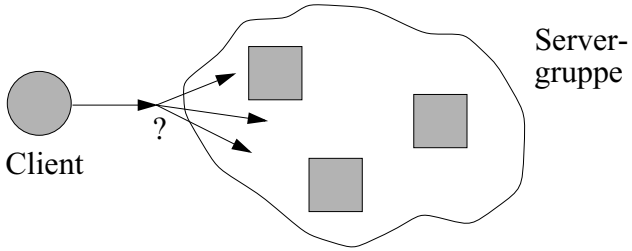
- z.B. Multicast an eine Server*gruppe*
- oder Auswahl eines *bestimmten* Servers (z.B. nahegelegenen oder wenig belasteten)

b) *Kooperative Server* ← "Know-how-Verbund"

- ein Server allein kann den Dienst nicht erbringen



Serverwahl bei einem Lastverbund

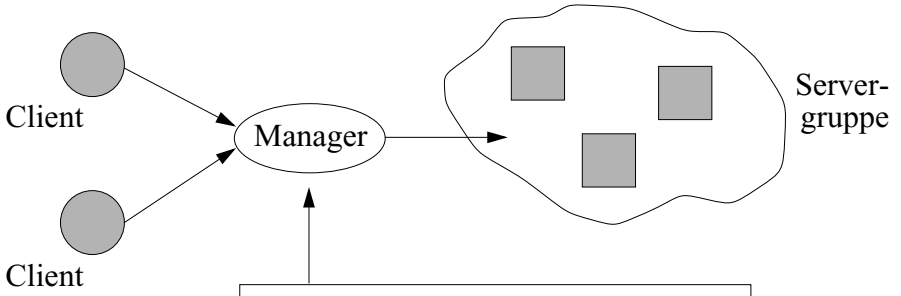


1) *Zufallsauswahl*

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

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

2) *Zentraler Service-Manager*

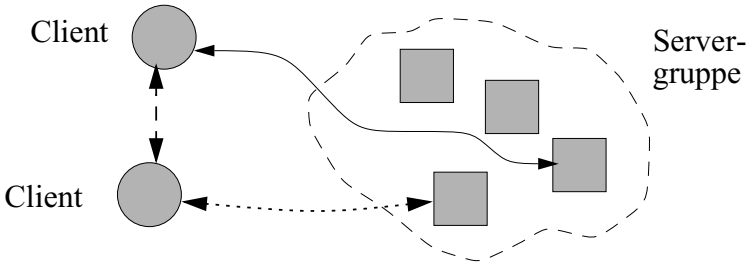


- sorgt für sinnvolle Verteilung (wie?)
- behält ggf. Überblick über Aufträge
- informiert sich ggf. von Zeit zu Zeit über die Server-Lastsituation

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

Serverwahl bei Lastverbund (2)

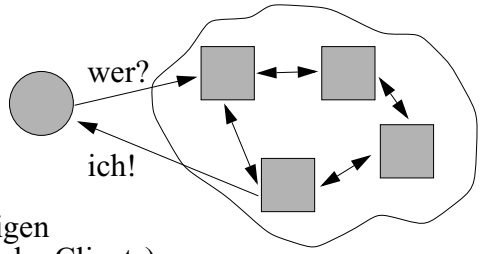
3) *Clients einigen sich untereinander*



- 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

- Abstimmung (aber fehlertolerant wegen möglichen Server-Ausfällen)



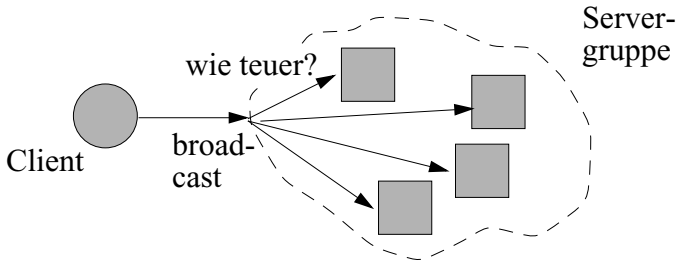
- i.a. nur sinnvoll bei wenigen Servern (relativ zur Zahl der Clients)
- Server führen Abstimmung diszipliniert durch (verlässlicher als Clients)

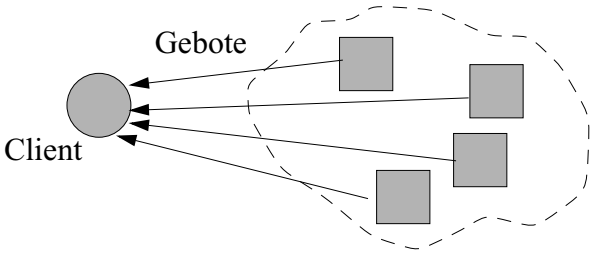
5) *"Anycast"*: der nächstliegende (=?) Server wird von der Netzinfrastruktur (Router etc.) bestimmt

Serverwahl bei Lastverbund (3)

6) *Bidding-Protokoll*

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





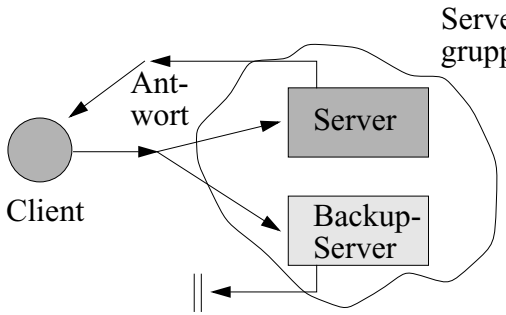
- Variante: nur *Stichprobe* befragen (Multicast statt Broadcast; sehr kleine Teilmenge von vielen Servern genügt i.a.!)

- 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:

- evtl. Subaufträge werden mehrfach erteilt (vom Server und Backup-Server) → Probleme mit zustandsändernden bzw. gegenseitig ausgeschlossenen Subdiensten

- Reihenfolge der Aufträge muss bei allen Servern identisch sein (→ Semantik von multicast insbesondere bei mehreren Clients)

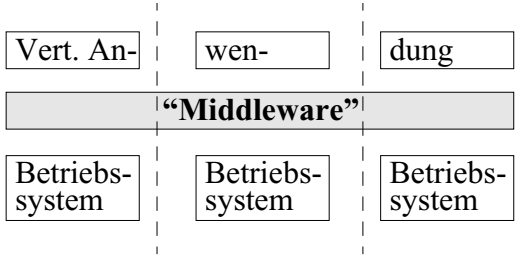
- Resynchronisation nach einem Crash: Nach Neustart muss ein (passiver) Server mit dem aktuellen Zustand des aktiven Servers initialisiert werden (Zustand kopieren bzw. replay)

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:
  - Verteiltheit (für die Anwendung) möglichst transparent machen (z.B. globaler Namensraum, globale Zugreifbarkeit, Ortstransparenz)
  - zumindest aber die Verteiltheit 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 und -dienste für verteiltes Programmieren anbieten, z.B.
  - automatische Schnittstellenanpassung (Schnittstellenbeschreibungs-sprache, Stub-Compiler...)
  - Verzeichnis- und Suchdienste (name service, lookup service,...)

Übersicht: “Historische” Entwicklung

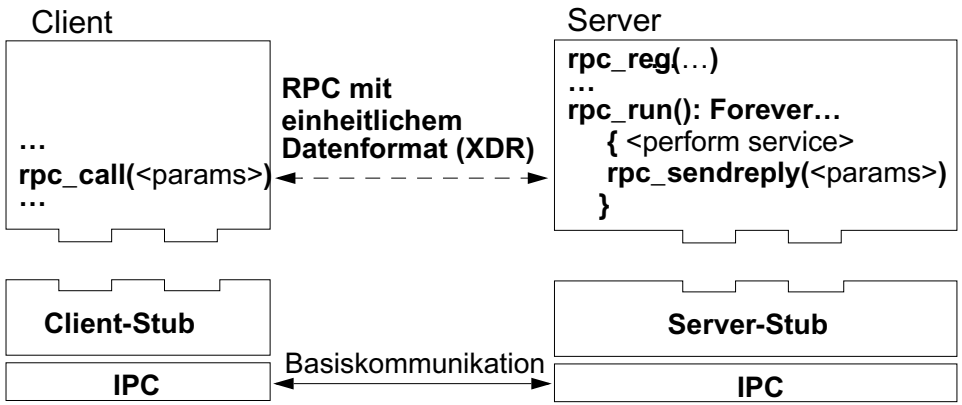
1. RPC-Bibliotheken: z.B. Sun-RPC
   - Client-Server-Paradigma, RPC-Kommunikation
   - Schnittstellen-Beschreibungssprache, Datenformatkonversion, Stubgeneratoren
   - Sicherheitskonzepte (Authentifizierung, Autorisierung, Verschlüsselung)
2. Client-Server-Verteilungsplattformen: z.B. DCE
   - Zeitdienst, Verzeichnisdienst
   - 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 im Laufe der Zeit zunehmend verwässert worden

- oft nicht nur gebraucht im technischen Sinne als Verteilungsplattform und Kommunikations- und Diensteinfrastruktur
- sondern auch für fast alles, was nicht direkt Anwendung oder Betriebssystem ist, also z.B. auch Datenbanken etc.

Sun-RPC

- RPC-”Package” der Firma Sun, welches unabhängig von der Rechnerarchitektur vielfältig einsetzbar ist
  - hier nur Überblick, Einzelheiten siehe Online-Dokumentation
  - im Laufe der Zeit sind leicht unterschiedliche Varianten enstanden
- 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

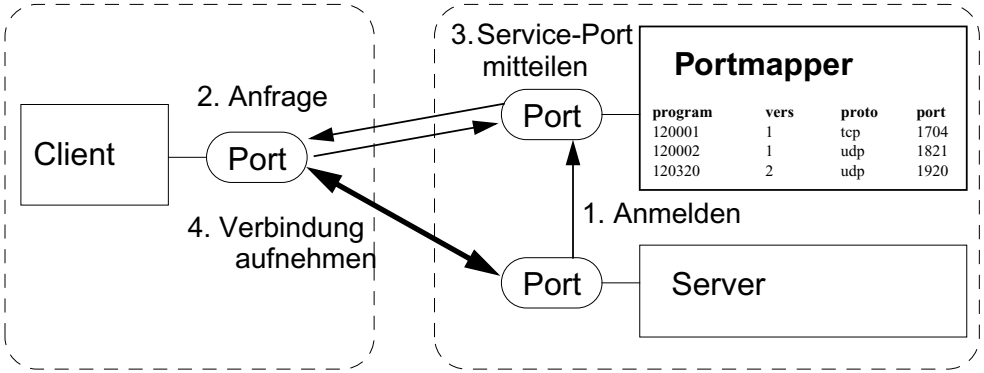
- RPC-Library: Vielzahl aufrufbarer Funktionen (“API”)
  - z.B. rpc_reg, rpc_run, rpc_call (Bezeichnung variiert je nach Variante)
  - daneben auch Funktionen einer Low-Level-Schnittstelle: z.B. Spezifikation von Timeout-Werten oder eines Authentifizierungsprotokolls
- rpcgen: Stub-Generator
- Portmapper: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

---

- Sicherheitskonzepte
  - z.B. diverse Authentifizierungsvarianten unterschiedlicher “Stärke”
- Semantik: “at least once”
  - im Detail 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 selbst hat immer den “well-known port” 111

Portmapper (2)

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

```
   program vers proto   port  service
    100000    2   tcp    111  portmapper
    100001    2   udp  32830  rstatd
    100004    1   udp    743  ypserv
    100004    1   tcp    744  ypserv
    100011    1   udp  32776  rquotad
    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)

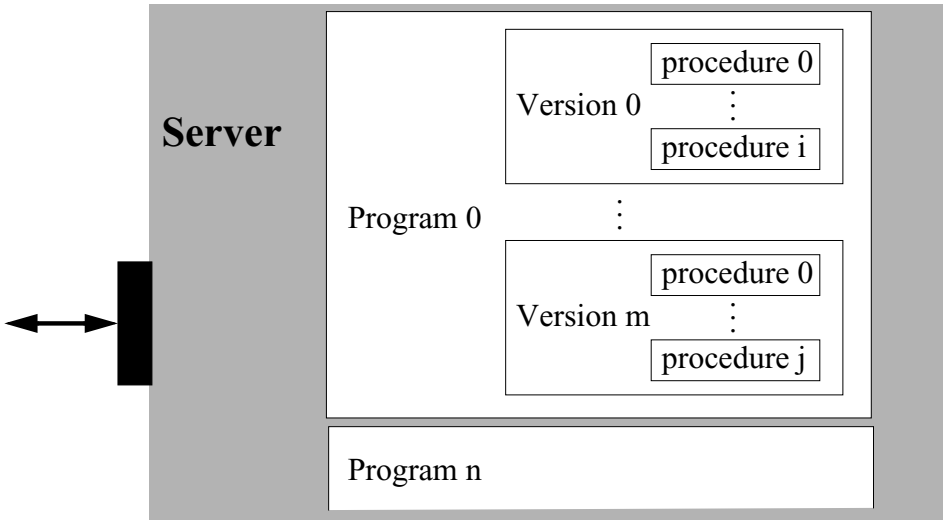
---

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 parameters; procname must be a procedure that returns a pointer to its static result; *inproc* is used to XDR-decode the parameters while *outproc* is used to XDR-encode the results.

- Welche Programmnummer bekommt ein Service?

  → Einige Programmnummern für *Standarddienste* sind vom System bereits fest 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.

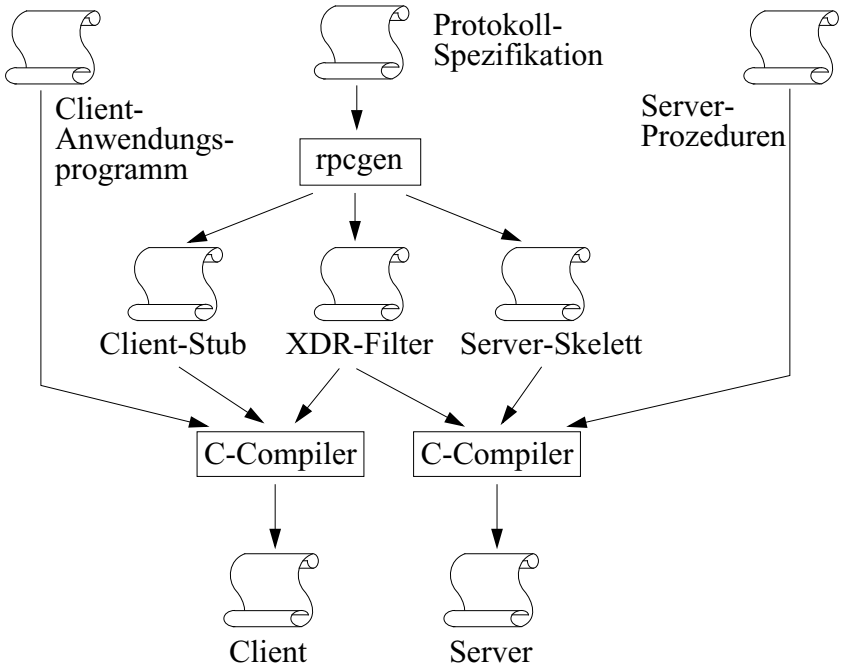
- Es gibt auch eine Broadcast-Variante:

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_broadcas*t() 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*:

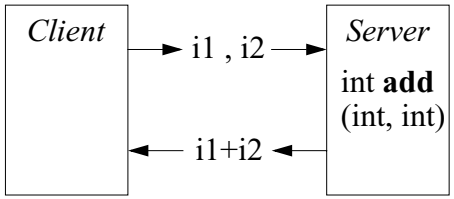
```
struct i_result
{ int x; };

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-Sicherheitsdienst muss dann natürlich installiert sein

  - *GSS:* Generic Security Service: zusätzlich auch Verschlüsselung

UNIX/SYS-Flavor bei Sun-RPC

- Im Sinne der UNIX-Sicherheitsphilosophie wird der Zugang zu gewissen Diensten auf bestimmte Benutzer / Benutzergruppen beschränkt

- 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

  - 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 eine Zeitfenstergrösse 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

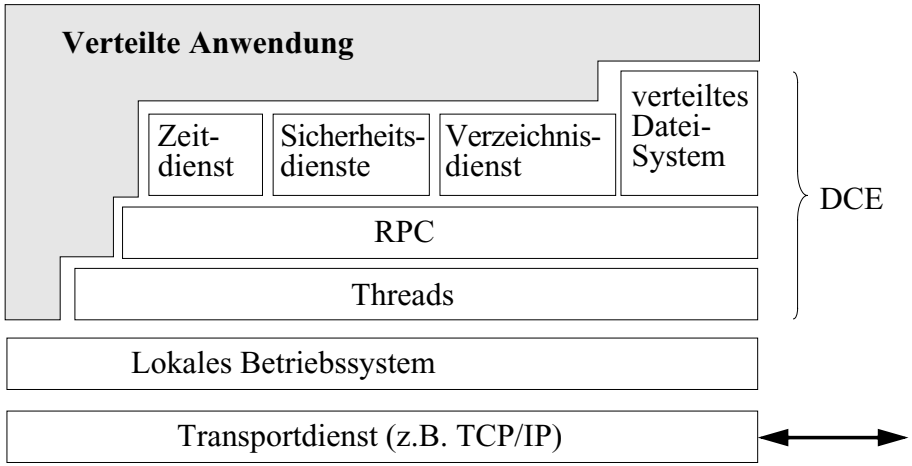
- Entwickelt von einem herstellerübergreifenden Konsortium (“OSF” - Open Software Foundation)
  - Anfang der 1990er Jahre, u.a. DEC, IBM, Siemens, HP...
  - trotz CORBA, Web Services etc. noch lange eingesetzt in der Praxis
- Ziel: Schaffung eines “offenen” Industriestandards für verteilte Verarbeitung auf diversen Plattformen
- Vorgehensweise pragmatisch: Soweit möglich, Nutzung geeigneter existierender Technologiekomponenten

---

Kritik an DCE

- Funktionsfülle (> 200 Funktionen)
  - wann benutzt man was?
  - Problem der wechselseitigen Beeinflussung („feature interaction“)
- Grösse
- Mangelnde Effizienz

Hauptkomponenten des DCE



Neu gegenüber Sun-RPC:

- *Threads* als Basisfunktionalität, die von Systemdiensten (aber auch von Anwendungen) benutzt werden können

- Höhere *Dienste*: u.a. vert. Dateisystem, Zeitdienst, Verzeichnis- und Namensdienst, Sicherheitsdienst

- *Tools* zum Systemmanagement

DCE: Komplexes Threadkonzept

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

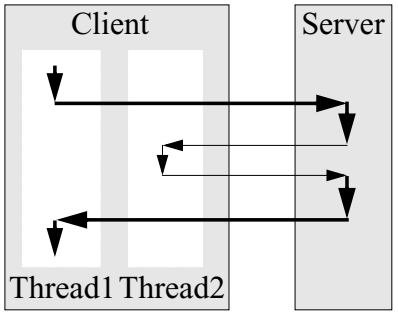
  - z.B. synchrone E/A, die *alle* Threads des Prozesses blockieren würde, statt nur den aufrufenden Thread
  - "Lösung": Verwendung asynchroner Operationen zum Test auf mögliche Blockaden

---

- 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: Besonderheiten

- Nahezu *beliebige Parametertypen*
  - Zeiger werden automatisch dereferenziert und als Wert übergeben
- *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: 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

  (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 an mehrere lokale Server geschickt
  - RPC ist beendet mit der ersten empfangenen Antwort

- 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)