Internet Domain Name System (DNS)

- Jeder Rechner im Internet hat eine IP-Adresse
  - bei IPv4: 32 Bit lang, typw. als 4 Dezimalzahlen geschrieben
  - Bsp.: 192.130.10.121 (= 11000000.10000010.00001010.01111001)

- Symbolische Namen sind für Menschen eher geeignet
  - 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 (meist rekursiv) gegliedert in Subdomains, z.B.

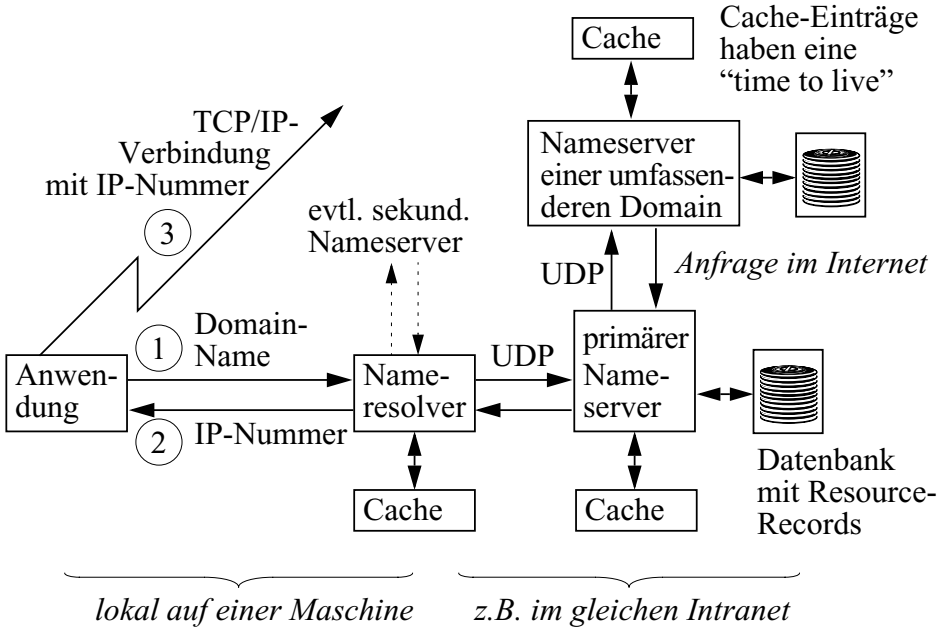
eu
uni-cooltown.eu
informatik.uni-cooltown.eu
nano.informatik.uni-cooltown.eu
pc6.nano.informatik.uni-cooltown.eu

- 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 (→ DNSSEC)

DNS-Resource-Records

- Datenbank eines DNS-Nameservers besteht aus einer Menge von Resource-Records
  - aus historischen Gründen einfache Struktur und einfaches Format

```
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 an anderer Stelle, z.B.:
  - IP-Adresse der übergeordneten Nameserver
  - ob Primär- oder Sekundärserver etc.

Interaktive DNS-Anfrage

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:  74.125.57.104,
74.125.59.104, 74.125.39.104

> google.com
Name:    google.com
Addresses:  74.125.59.104,
74.125.39.104, 74.125.57.104

> 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.Allg. den ersten Eintrag nehmen, wird so eine Lastverteilung auf mehrere google-Server vorgenommen; stellt gleichzeitige eine rudimentäre Fehlertoleranz bereit.

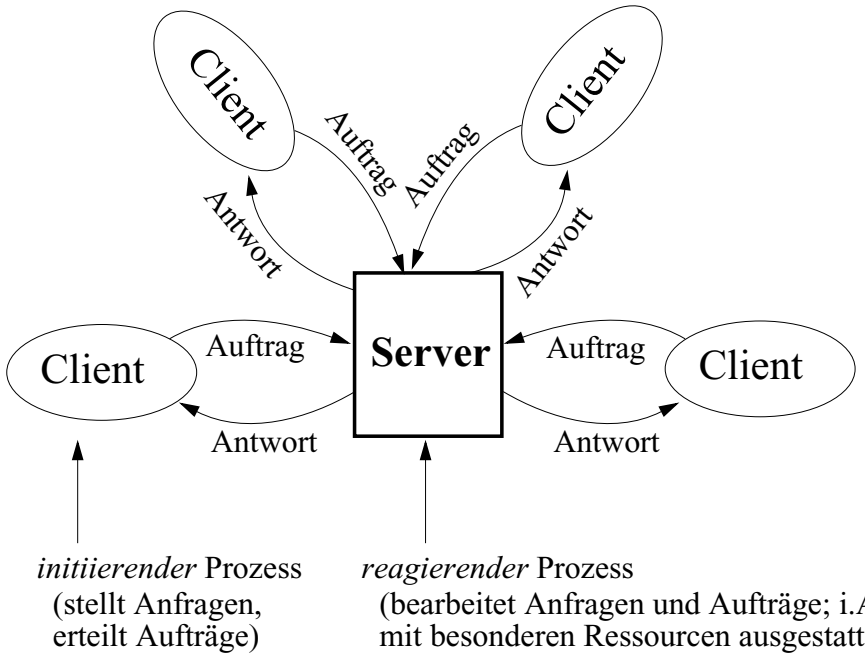
Router an zwei Netzen

---

dig - DNS lookup utility

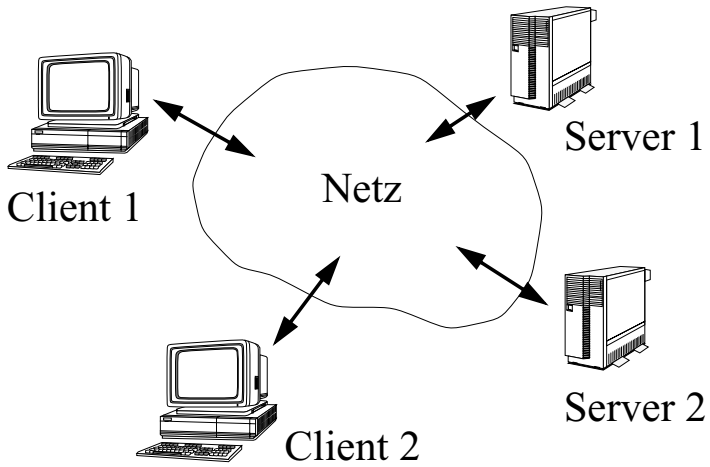
dig (domain information groper) is a flexible tool for interrogating DNS name servers. It performs DNS lookups and displays the answers that are returned from the name server(s) that were queried. Most DNS administrators use dig to troubleshoot DNS problems because of its flexibility, ease of use...

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

Client- und Server-Maschinen



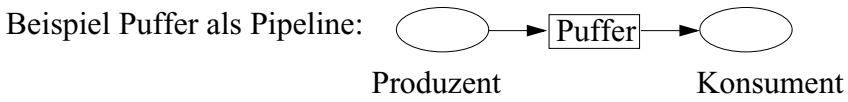
- Typischerweise PCs, Smartphones etc. als Clients und andere, leistungsfähigere Rechner als Server
  - "zentrale" Dienste
  - gemeinsam benutzte Betriebsmittel
- Evtl. ganze *Server-Farmen*
  - Virtualisierung von Server-Computern
  - grid computing, cloud computing

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 nur 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

---

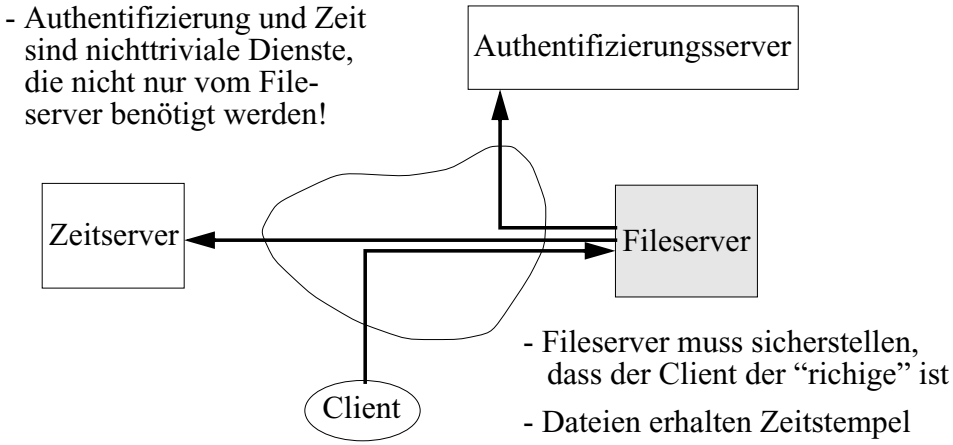
- Modell ist für viele Zwecke geeignet, jedoch nicht für alle (z.B. Pipelines, “peer-to-peer”, asyn. Mitteilung)!



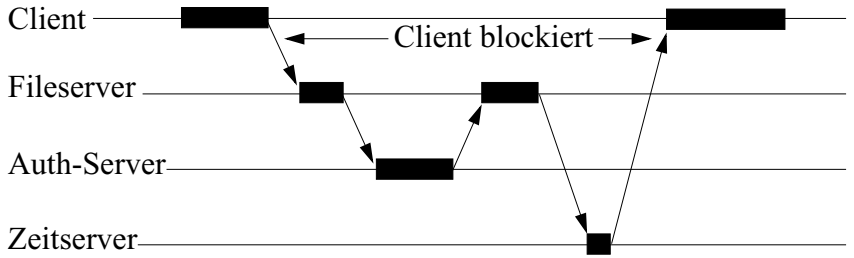
- Puffer ist weder Client noch Server, sondern hat beide Rollen! (passiv gegenüber Produzent; aktiv gegenüber Konsument)

Client/Server-Rollen

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



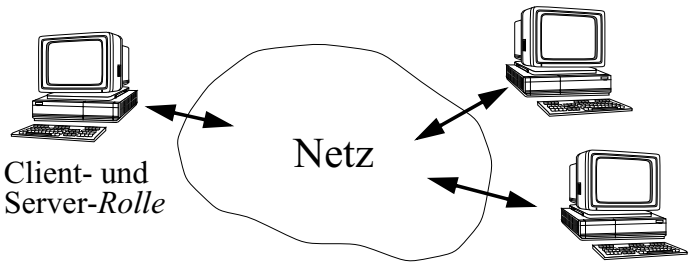
- Fileserver hat prinzipiell die *Rolle* eines Servers, zwischenzeitlich jedoch die *Rolle* eines Clients



Peer-to-Peer-Strukturen

"Gleichrangiger"

- Im Gegensatz zum asymmetrischen Client/Server-Modell



- Ein Client fungiert zugleich als Server für seine Partner
  → keine (teuren) dedizierten Server notwendig

- In der Idealform keine zentralisierten Elemente
  - dies wird gelegentlich in "politischer" Weise artikuliert (vgl. Tauschbörsen)

- *Nachteile*:
  - "Anarchischer" als *maschinenbezogene* Client/Server-Architektur
  - Computer müssen leistungsfähig genug sein (cpu-Leistung, Speicherausbau), um für den "Besitzer" leistungstransparent zu sein
  - geringere Stabilität (Besitzer kann seine Maschine ausschalten...)
  - Datensicherung i.Allg. problematischer als bei zentralen Servern
  - 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.)
- zustandsbehaftete Server daher i.Allg. effizienter

notw. Zustandsinformation ist beim Client

- 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!

- Datei-Server wurden sowohl schon zustandslos (z.B. NFS) als auch zustandsbehaftet (z.B. RFS) realisiert

Sind Webserver zustandslos?

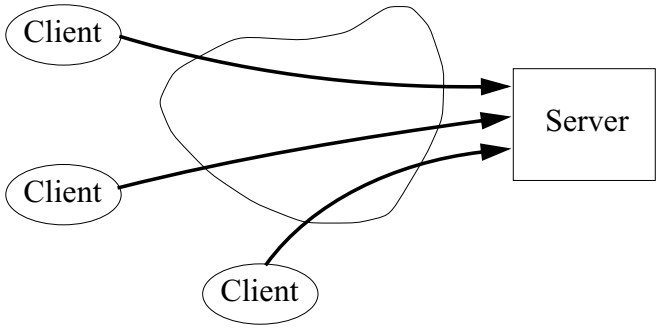
- Beim HTTP-Zugriffsprotokoll wird über den Auftrag hinweg keine Zustandsinformation gehalten
  - jeder link, den man anklickt, löst eine neue "Transaktion" aus
- Stellt ein Problem beim E-Commerce dar
  - gewünscht sind Transaktionen über mehrere Klicks hinweg und
  - Wiedererkennen von Kunden (beim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

Lösungsmöglichkeiten (z.B. zur Realisierung von "Warenkörben" im WWW):

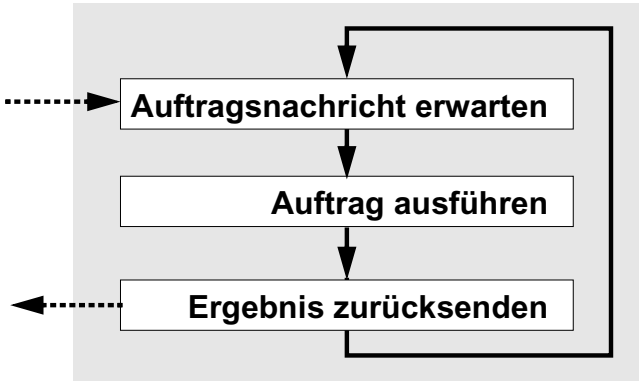
- IP-Adresse des Kunden an Auftrag anheften?
  - Problem: Proxy-Server → mehrere Kunden haben gleiche IP-Adresse
  - Problem: dynamische IP-Adressen → keine Langzeitwiedererkennung
- "URL rewriting" und dynamische Web-Seiten
  - Einstiegsseite eine eindeutige Nummer anheften, wenn der Kunde diese erstmalig 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
  - der Sender des Cookies kann dieses später wieder lesen und damit den Kunden wiedererkennen

Gleichzeitige Server-Aufträge?

- Problem: Oft viele “gleichzeitige” Aufträge



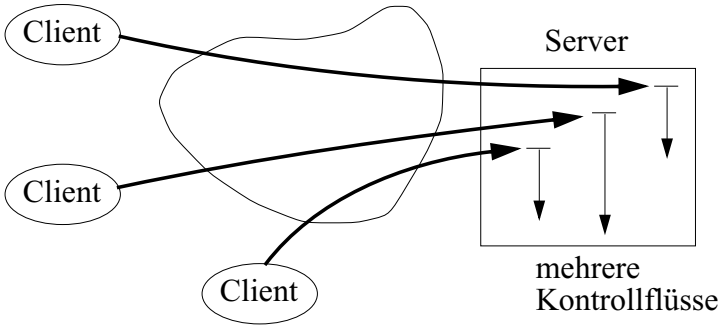
- *Iterative Server* bearbeiten nur einen Auftrag pro Zeit



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

Konkurrente (“nebenläufige”) Server

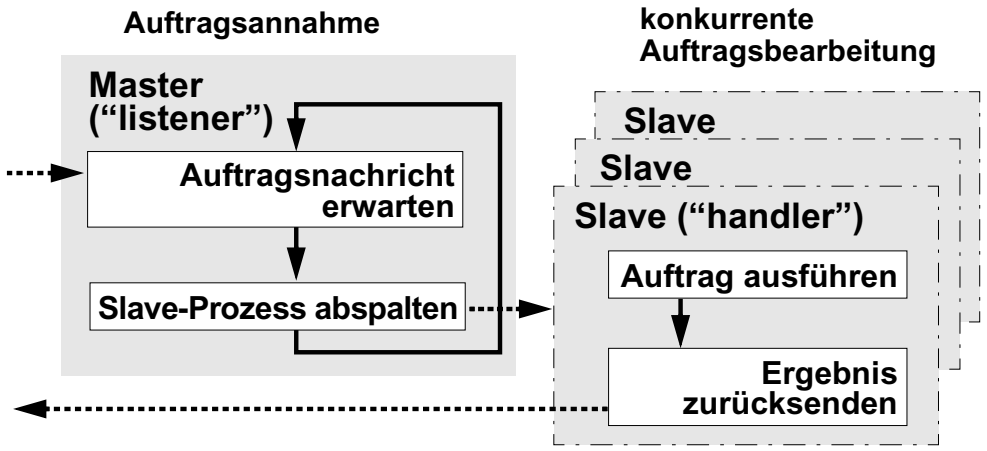
- Gleichzeitige Bearbeitung mehrerer Aufträge
  - sinnvoll (d.h. effizienter für Clients) bei längeren Aufträgen



- Ideal bei physischer Parallelität (z.B. multicore)
  - 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 evtl. Lastbalancierung beachten
- Verschiedene denkbare Realisierungen, z.B.
  - mehrere Prozessoren bzw. Multicore-Prozessoren
  - Verbund verschiedener Server-Maschinen (Server-Farm, -Cluster)
  - feste Anzahl vorgegründeter Prozesse oder dynamische 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 oft Leichtgewichtsprozesse (“threads”)
  - Slaves terminieren i.Allg. nach Beendigung des Auftrags
  - die Anzahl gleichzeitiger Slaves sollte begrenzt werden



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

Master/Slave

Subject: Identification of equipment sold to LA County
Date: Tue, 18 Nov 2003 14:21:16 -0800
From: "Los Angeles County"

The County of Los Angeles actively promotes and is committed to ensure a work environment that is free from any discriminatory influence be it actual or perceived. As such, it is the County’s expectation that our manufacturers, suppliers and contractors make a concentrated effort to ensure that any equipment, supplies or services that are provided to County departments do not possess or portray an image that may be construed as offensive or defamatory in nature.

One such recent example included the manufacturer’s labeling of equipment where the words "Master/Slave" appeared to identify the primary and secondary sources. Based on the cultural diversity and sensitivity of Los Angeles County, this is not an acceptable identification label.

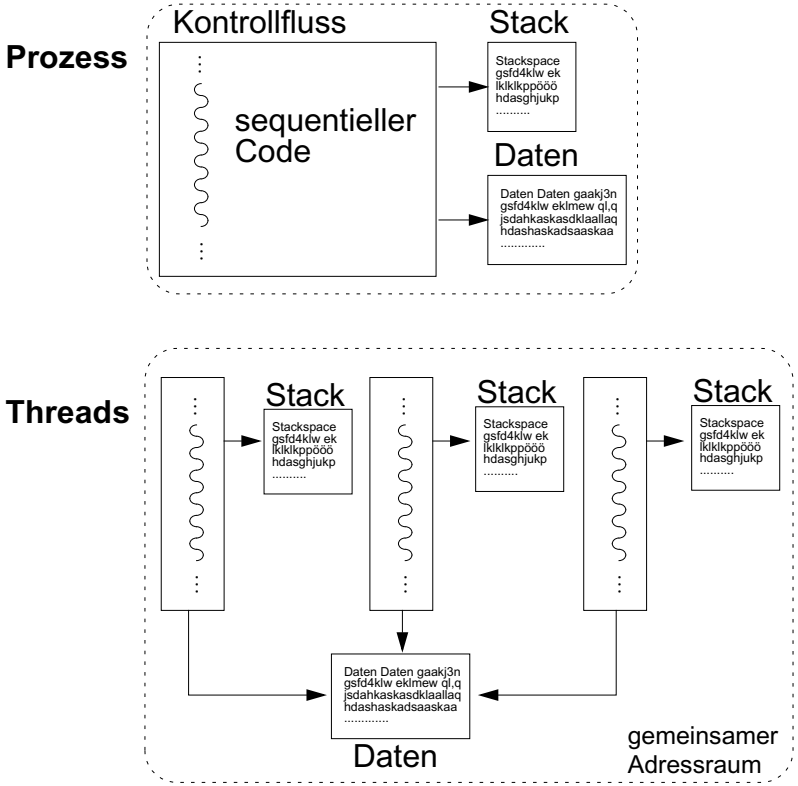
We would request that each manufacturer, supplier and contractor review, identify and remove/change any identification or labeling of equipment or components thereof that could be interpreted as discriminatory or offensive in nature before such equipment is sold or otherwise provided to any County department.

Thank you in advance for your cooperation and assistance.

Joe Sandoval, Division Manager
Purchasing and Contract Services
Internal Services Department
County of Los Angeles

Threads

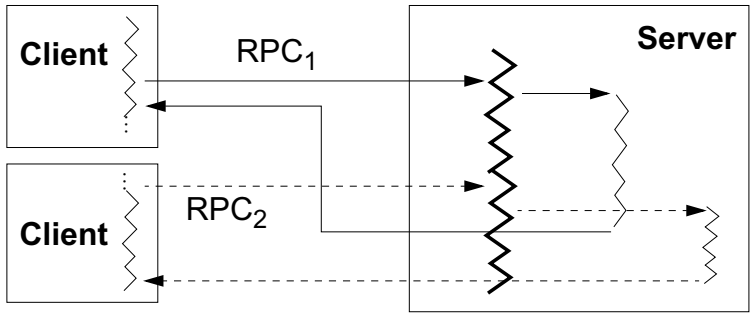
- Threads = leichtgewichtige Prozesse mit gemeinsamem Adressraum



- Einfache Kommunikation zwischen Kontrollflüssen
  - aber: kein gegenseitiger Schutz; Synchronisation bzgl. gem. Speicher
- Thread hat weniger Zustandsinformation als ein Prozess
- Kontextwechsel daher i.Allg. wesentlich schneller
  - kein Umschalten des Adressraumkontexts
  - Cache und Translation Look Aside Buffer (TLB) bleiben erhalten
  - kein aufwändiger Wechsel in / über 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

Problematik von Threads

- Fehlender gegenseitiger Adressraumschutz
  → schwierig zu findende Fehler
- Stackgrösse muss bei Gründung i.Allg. statisch festgelegt werden
  → unkalkulierbares Verhalten bei Überschreitung
- Von asynchronen Meldungen ("Signale", "Interrupts") an den Prozess soll i.Allg. nur ein einziger (der "richtige") Thread betroffen werden
- Schwierige Synchronisation → Deadlockgefahr

Aufrufe des Betriebssystem-Kerns können problematisch sein, wenn diese nicht dafür geeignet ("thread safe") sind:

a) *nicht ablaufinvariante* ("non-reentrant") Systemroutinen

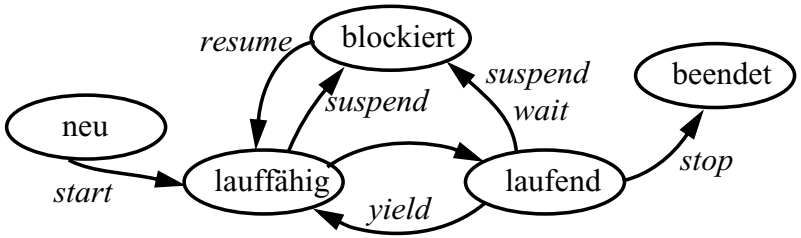
- interne Statusinformation, die ausserhalb des Stacks der Routine gehalten wird, kann bei paralleler Verwendung überschrieben werden
- z.B. *printf*: ruft intern Speichergenerierungsroutine auf; diese benutzt prozesslokale Freispeicherliste, deren "gleichzeitige" nicht-atomare Manipulation zu Fehlverhalten führt
- "Lösung": Verwendung von "Wrapper-Routinen", die gefährdete Routinen kapseln und Aufrufe wechselseitig ausschliessen

b) *blockierende* ("synchrone") Systemroutinen

- z.B. synchrone E/A, die *alle* Threads eines Prozesses blockieren würde (statt nur den einen aufrufenden Thread)

Exkurs: Threads in Java

- Hier nur ein *Überblick*; zu weiteren Aspekten vgl. die Dokumentation (online bzw. in Büchern)
- *Konstruktor*:
  - **public** Thread()
- *Methoden*:
  - **void** start()
  - **void** suspend()
  - **void** stop()
  - **void** resume()
  - **void** wait()
  - **static void** yield()



- **static void** sleep(**long** millis) // blockiert für eine gewisse Zeit
- **void** join() // Synchronisation zweier Threads
- **void** setPriority(**int** prio)
- **int** getPriority()
- **void** setDaemon (**boolean** on)

"Hintergrundprozess": terminiert nicht mit dem Erzeuger

- Jeder Thread (genauer: jede von Thread abgeleitete Klasse) muss eine void-Methode *run()* enthalten
  - diese macht die eigentlichen Anweisungen des Threads aus!
  - "run" ist in Thread nur abstrakt definiert

Erzeugen von Threads

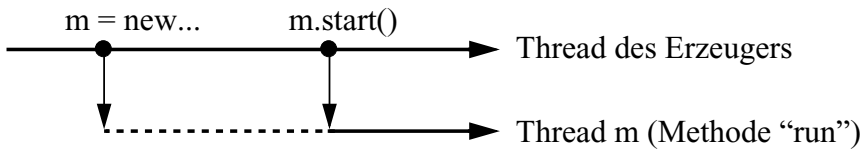
- Typisches Gerüst für einen Thread:

```
class Mythread extends Thread
{ int mynumber;
  public Mythread(int number)
  { mynumber = number; }

  public void run()
  { // hier die Anweisungen des Threads
    ...
  }

  // hier andere Methoden
}
```

Konstruktor



- Erzeugen aus einem anderen Thread heraus:

```
Mythread m = new Mythread(5);
m.start();
```

mit dieser Nummer identifizieren wir einen Thread individuell

damit kann man den Thread kontrollieren (z.B. m.suspend();)

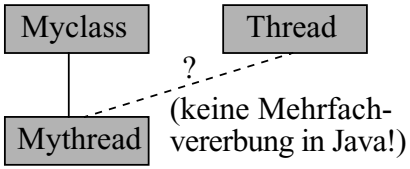
- Alternative: “Anonyme” Erzeugung

```
new Mythread(5).start();
```

Mythread “kann” start, da dies von Thread ererbt ist

- Zusammenfassen der beiden Anweisungen
- dann aber keine Kontrolle möglich, da kein Zugriff auf den Thread

Abgeleitete Klassen als Threads



- Myclass sei eine Klasse, die nicht von Thread abgeleitet ist
- Mythread soll Unterklasse von Myclass sein; gleichzeitig aber auch einen Thread darstellen

- Lösung über die Runnable-Schnittstelle:

```
class Mythread extends Myclass implements Runnable
{ ...
  public void run()
  { ... }
  ...
}
```

Definition (“Implementierung”) von run muss hier erfolgen

- Erzeugung aus einem anderen Thread heraus:

```
Mythread m = new Mythread(...);

Thread t = new Thread (m);
t.start();
```

m “kann” kein start, da dies nicht in runnable enthalten; erst t als Thread kann start

zweite Form des Konstruktors!

- Es geht auch so:

```
Mythread m = new Mythread(...);
```

In der Klasse Mythread dann an geeigneter Stelle:

```
t = new Thread(this);
t.start();
```

Assoziation des Threads zur Methode run herstellen

Ein Thread-Beispiel

```
class T extends Thread
{ String wer; int delay = 0;

  T (String s, int zeit)
  { wer = s; delay = zeit;
    System.out.println(wer + " : " + delay);
  }

  public void run()
  { try
    { while (true)
      { System.out.println(wer);
        sleep(delay);
      }
    }
    catch (InterruptedException e) {return;}
  }

  public static void main(String args[])
         throws InterruptedException
  { int i = Integer.parseInt(args[0]);
    int j = Integer.parseInt(args[1]);
    new T("She loves me", i).start();
    sleep(i/2);
    new T("  not", j).start();
    System.out.println("Jetzt sind beide
                    Threads gestartet");
  }
}
```

Konstruktor

Exception, falls während des sleep ein Interrupt ausgelöst wird

Methode run und damit den Thread beenden

statt try / catch

Gründen zweier Threads als Instanzen der eigenen Klasse

Die Threads in Aktion

> java T 500 500

She loves me: 500
She loves me
 not : 500
Jetzt sind beide Threads gestartet
 not
She loves me
 not
She loves me
 not
She loves me
 not
She loves me
 not
She loves me
 not
She loves me
....

Vielleicht hätte man für ein Happy-End doch die Parameter anders wählen sollen?

> java T 1000 300

She loves me: 1000
She loves me
 not : 300
Jetzt sind beide Threads gestartet
 not
 not
She loves me
 not
 not
 not
She loves me
 not
 not
 not
 not
She loves me
 not
 not
 not
She loves me
 not
 not
 not
She loves me
....

Ein weiteres Thread-Beispiel

- Ein Thread “Hin” schreibt Sterne; ein Thread “Her” löscht Sterne - beide arbeiten (quasi)parallel
  - nach einer Idee von Ralf Kühnel (“Die Java-Fibel”, Addison-Wesley)

```
public class HinHer
{ public static void main (String args[])
  { System.out.print("**********");
    System.out.flush();
    new Hin().start();
    new Her().start();
  }
}

class Hin extends Thread
{ public void run()
  { try
    { while(true)
      { sleep((int)(Math.random() * 1000));
        repaint();
        System.out.flush();
      }
    }
    catch (InterruptedException e) {return;}
  }

  public void repaint()  // Sterne hinzufuegen
  { System.out.print("*****"); }
}

class Her extends Hin
{ public void repaint()  // Sterne loeschen
  { System.out.print("\b\b\b\b\b    \b\b\b\b\b"); }
}
```

Übung: Ausprobieren, wer gewinnt!

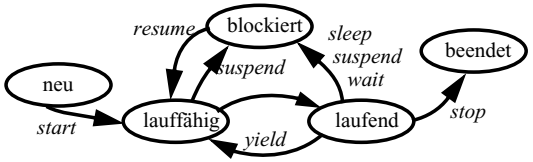
Thread-Kontrolle

- Ein Thread läuft so lange, bis
  - seine run-Methode zuende ist
  - er mit stop() abgebrochen wird

laufend oder lauffähig

von aussen oder durch sich selbst

- Ein Thread kann *sich selbst*
  - die cpu entziehen: *yield()*
    (Übergang in Zustand “lauffähig”; wird automatisch wieder “laufend”, wenn keine wichtigeren Threads mehr laufen möchten)
  - schlafen legen: *sleep()*
    (automatisches resume nach gegebener Zeit)
  - anhalten: *suspend()* bzw. *wait()*
  - beenden: *stop()*
  - in der Priorität verändern: *setPriority()*



Prioritäten: normal 5, minimal 1, maximal 10 (anfangs: Priorität des erzeugenden Prozesses)

- Ein Thread kann einen *anderen* Thread t
  - *starten*: t.start()
  - *anhalten*: t.suspend()
  - *fortsetztbar machen*: t.resume();
  - *beenden*: t.stop()
  - in der *Priorität verändern*: t.setPriority()

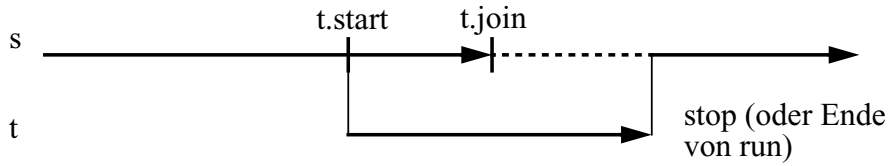
hierzu ist eine Referenz auf den Thread notwendig

Beachte: *stop*, *suspend* und *resume* führen, unbedacht angewendet, zu unsicheren Programmen und sollte daher eigentlich nicht mehr verwendet werden

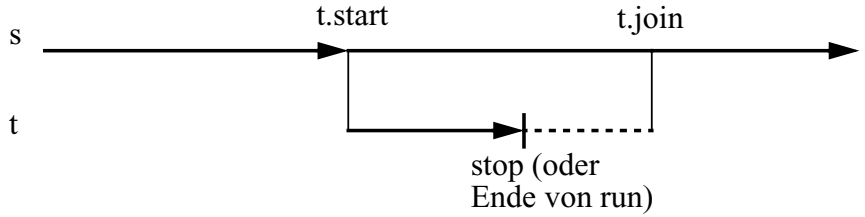
Warten auf Threads

- Methode *join* verwenden, wenn auf die Beendigung eines anderen Threads gewartet werden soll
  - z.B. weil auf die von ihm berechneten Daten zugegriffen werden soll
- Das Objekt eines beendeten Threads existiert weiter
  - auf dessen Zustand kann also noch zugegriffen werden
- Auf einen beendeten Thread kann *start* aufgerufen werden
  - run-Methode wird dann erneut ausgeführt

- Beispiel: Thread s wartet tatsächlich auf t:



- Thread t ist eher fertig:



- Nach t.join ist jedenfalls garantiert, dass t beendet ist

Thread-Scheduling

- Scheduling: Planvolle Zuordnung der cpu an die einzelnen Threads (jeweils für eine gewisse Zeit)
- Genaue Scheduling-Strategie ist *nicht* Bestandteil des Java-Sprachstandards
  - kann jede Implementierung für sich entscheiden (und damit Eigenheiten des zugrundeliegenden Betriebssystems effizient nutzen)
  - man darf sich daher nicht auf "Erfahrungen" verlassen
  - genauer: nicht auf die Wirkung von Zeitscheiben oder Prioritäten etc.

  sonst nicht deterministisch und nicht portabel!

- Konsequenzen:
  - Test und Debugging ist sehr schwierig
  - alle denkbaren verzahnten Abläufe ("interleavings") berücksichtigen
  - Menge der verzahnten Abläufe durch geeignete *Synchronisation* einschränken (nur "korrekte" Abläufe zulassen)
  - ggf. mit "yield" Scheduling teilweise selbst realisieren (z.B. um andere Prozesse am Verhungern zu hindern)
- Vorgabe: Ein Thread-Scheduler *soll* Threads mit höherer Priorität bevorzugen
  - Priorität eines Threads entspricht zunächst der des Erzeugers
  - Priorität kann verändert werden (*setPriority*)
  - Thread mit der höchsten Priorität *sollte* immer laufen (ohne Garantie!)
  - wenn ein Thread mit höherer Priorität als der gegenwärtig ausgeführte lauffähig wird, wird der gegenwärtige i.a. unterbrochen

  auch bei einem Rechner mit zwei cpus?

Thread-Scheduling (2)

im Sinne von “laufend”

(sofortiger) Threadwechsel ist aber nicht garantiert!

- Wie lange läuft ein Thread?
  - bis ein Thread mit höherer Priorität lauffähig wird
  - bis er sich beendet (mit “stop” oder dem Ende von “run”)
  - bis er in den “blockiert”-Zustand übergeht (explizit mit “suspend”, “wait”, “sleep” etc. ; implizit durch E/A etc. - es ist aber umgekehrt nicht garantiert, dass ein auf E/A-wartender Thread die cpu freigibt!)
  - bis er mit “yield” die Kontrolle dem Scheduler übergibt

- Scheduling mit Zeitscheiben *kann* vom System realisiert sein, *muss* aber *nicht*
  - Thread läuft längstens bis zum Ablauf der Zeitscheibe (dann i.a. Round-Robin-Scheduling unter Threads gleicher maximaler Priorität)

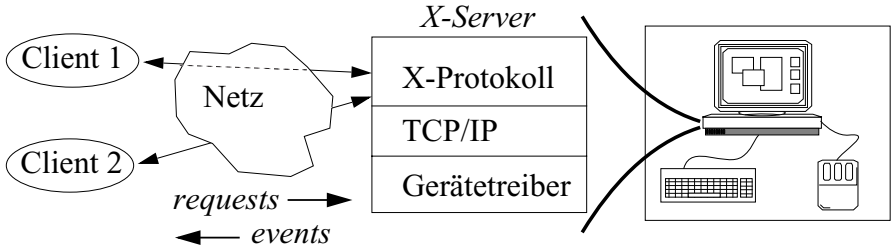
- Konsequenzen:
  - Thread mit Endlosschleife kann gegebenenfalls das ganze System blockieren (andere Threads “verhungern”)
  - Threads gleicher Priorität verhalten sich besonders willkürlich
  - “yield” ist insbesondere bei Systemen ohne Zeitscheiben wichtig
  - Prioritäten sollten besser nicht als Synchronisationsmittel (zum Erzwingen einer bestimmten Reihenfolge etc.) eingesetzt werden
  - Portabilität ist bei diletantischer Nutzung von Threads eingeschränkt

---

Exkurs-Ende (Threads in Java)

Historische Notiz: “X-Window” als Client/Server-Modell

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



- i.Allg. 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)

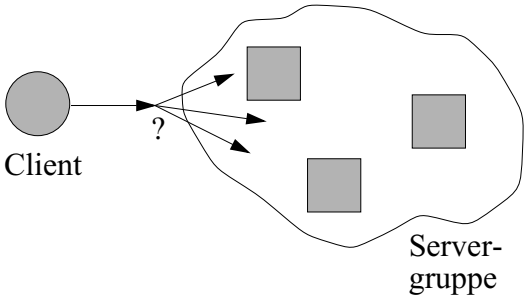
- *Requests*: Service-Anforderung an den X-Server (z.B. Linie in einer bestimmten Farbe zwischen zwei Koordinatenpunkten zeichnen); zugehörige Routinen stehen in einer Bibliothek (*Xlib*)
- *X-Library* (Xlib) ist die Programmierschnittstelle zum X-Protokoll; damit manipuliert ein Client vom Server verwaltete 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*

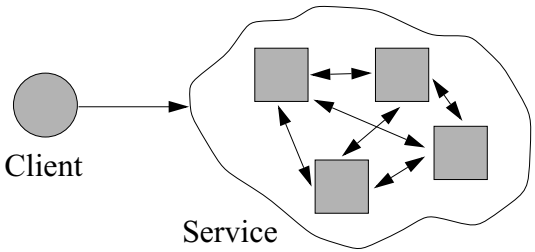
- Jeder einzelne Server kann den Dienst erbringen
- Zweck:
  - *Leistungssteigerung* (Verteilung der Arbeitslast auf mehrere Server) ← "Lastverbund"
  - *Fehlertoleranz* durch Replikation (Verfügbarkeit auch bei vereinzelten Server-Crashs) ← "Überlebensverbund"



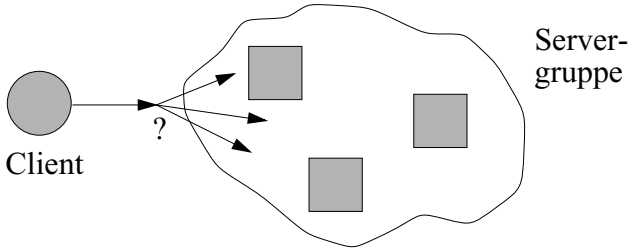
- z.B. Multicast an eine Server*gruppe*
- oder Auswahl eines *bestimmten* Servers (z.B. nahegelegenen oder wenig belasteten)

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

- ein Server allein kann den Dienst nicht erbringen



Serverwahl bei einem Lastverbund

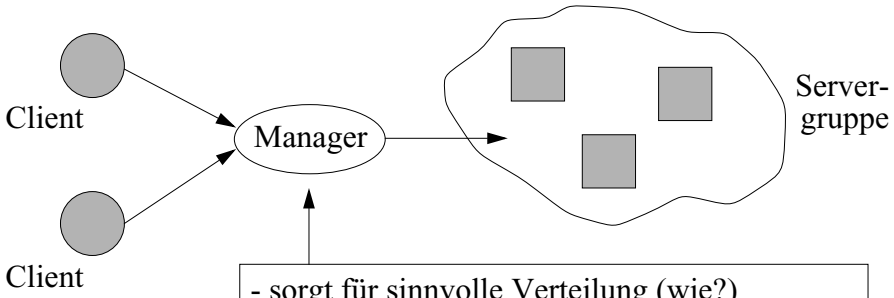


1) *Zufallsauswahl*

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

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

2) *Zentraler Service-Manager*



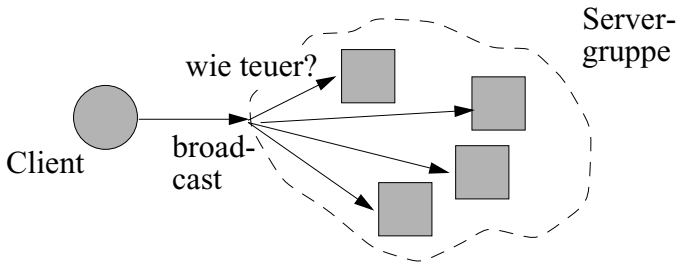
- sorgt für sinnvolle Verteilung (wie?)
- behält Überblick über Aufträge
- informiert sich von Zeit zu Zeit über die Server-Lastsituation
- fährt bei hoher Last evtl. weitere Server hoch

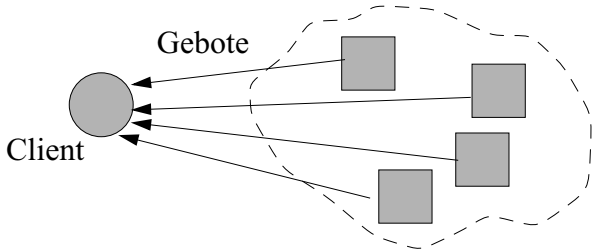
- Nachteile:
  - Overhead bei trivialen Diensten
  - ggf. Überlastung des Managers
  - Dienstblockade bei Ausfall des Managers (Redundanz?)

Serverwahl bei Lastverbund (2)

3) *Bidding-Protokoll*

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





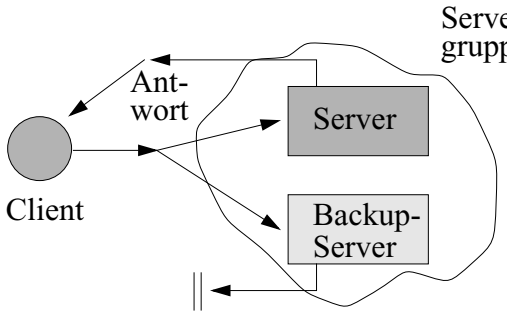
- Variante: nur *Stichprobe* befragen (Multicast statt Broadcast; sehr kleine Teilmenge von vielen Servern genügt i.Allg.!)

4) *“Anycast”*: der nächstliegende (=?) Server wird von der Netzinfrastruktur (Router etc.) bestimmt

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 (evtl. mehr als einer)

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

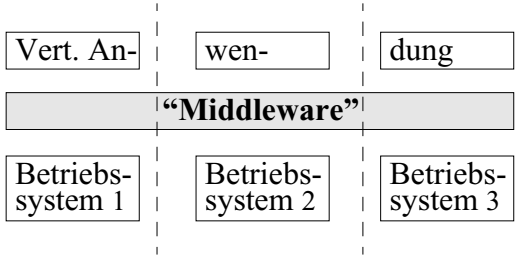
Probleme:

- 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

- Zweck: Durch eine geeignete Softwareinfrastruktur die Realisierung verteilter Anwendungen vereinfachen
  - kann man für viele Anwendungen gemeinsame Aspekte herausfaktorisieren?



- Aufgabe von Middleware:
  - 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
- Bietet gewisse Basismechanismen und -dienste für verteiltes Programmieren an, z.B.
  - Verzeichnis- und Suchdienste (name service, lookup service,...)
  - Weiterleiten von events

Übersicht: “Historische” Entwicklung

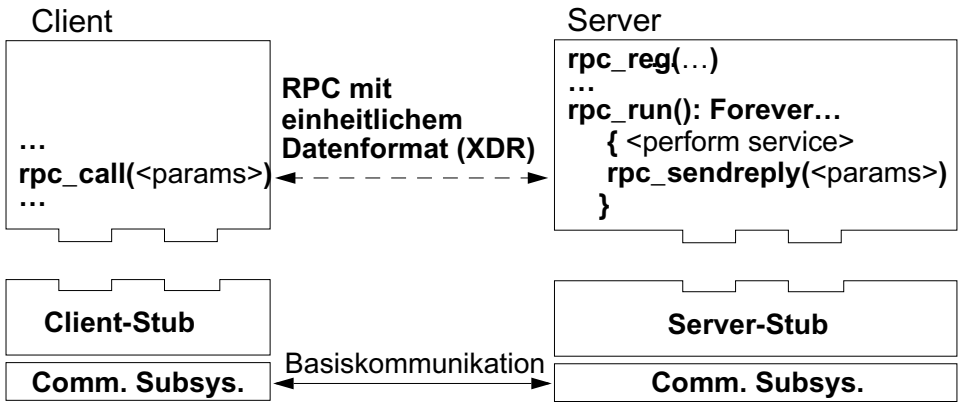
1. RPC-Bibliotheken: z.B. Sun-RPC
   - Unterstützung von Client-Server-Paradigma und RPC-Kommunikation
   - Schnittstellen-Beschreibungssprache, Datenformatkonversion, Stubgeneratoren
   - Sicherheitskonzepte (Authentifizierung, Autorisierung, Verschlüsselung)
2. Client-Server-Verteilungsplattformen: z.B. DCE
   - Zeitdienst, Verzeichnisdienst
   - globaler Namensraum, globales Dateisystem
   - Programmierhilfen: Synchronisation, Multithreading,…
3. Objektbasierte Verteilungsplattformen: z.B. CORBA
   - Kooperation zwischen verteilten Objekten
   - objektorientierte Schnittstellenbeschreibungssprache
   - “Object Request Broker”
4. Web-Services
   - Dienstorientierung aufbauend auf WWW als Plattform (SOAP, XML)
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 konkreter Systemarchitektur vielfältig einsetzbar ist
  - im Laufe der Zeit sind leicht unterschiedliche Varianten enstanden
  - auch bezeichnet als ONC-RPC (“Open Network Computing”)
- Beobachtung beim RPC: Grundgerüst ist immer gleich
  - → Grossteil des Aufrufrahmens vorkonfektionierbar
  - → automatische Generierung von Client-Stubs und Server-Gerüst



- Der Server meldet sich mit je einem *rpc_reg* für jeden Service beim Portverwalter an
- 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

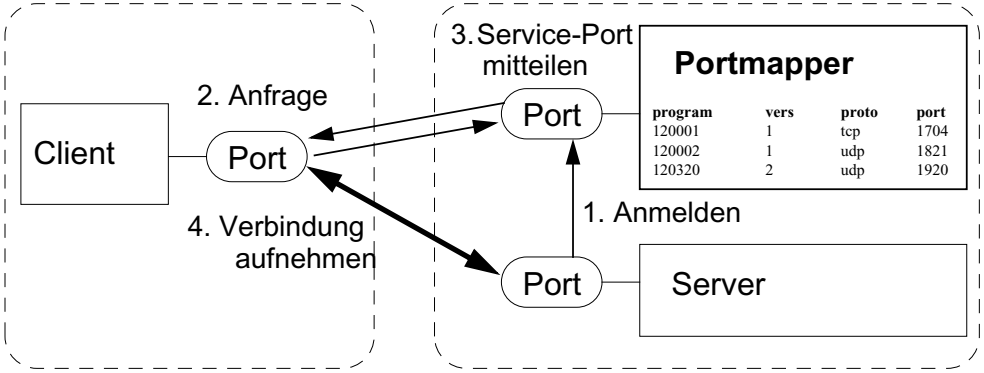
- RPC-Library: Vielzahl aufrufbarer Funktionen (“API”)
  - z.B. rpc_reg, rpc_run, rpc_call (Bezeichnung variiert je nach Variante)
  - daneben auch Funktionen einer Low-Level-Schnittstelle: z.B. Spezifikation von Timeout-Werten oder eines Authentifizierungsprotokolls
- rpcgen: Stub-Generator
- XDR-Library: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

---

- Sicherheitskonzepte
  - 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
  - IP-Adressen sind global eindeutig, Portnummern haben aber i.Allg. nur eine lokale Bedeutung
- 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)
  - 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)

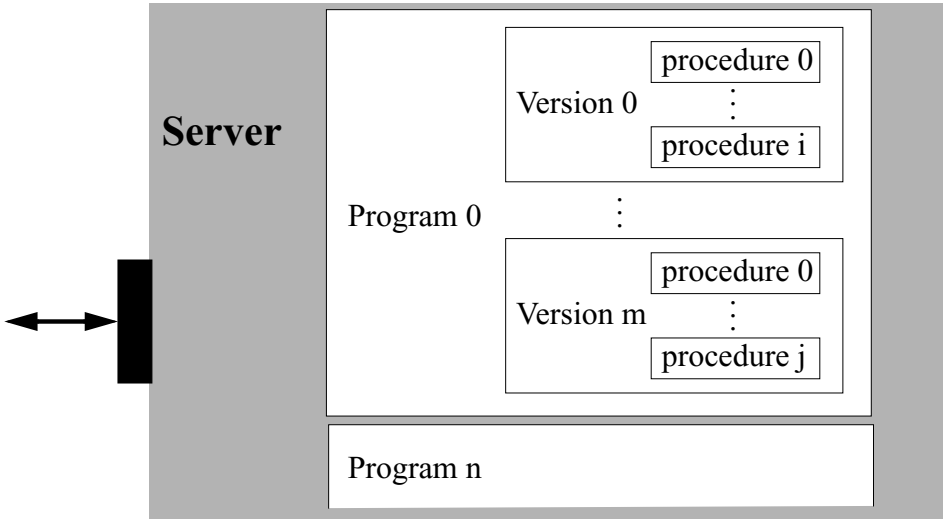
---

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.

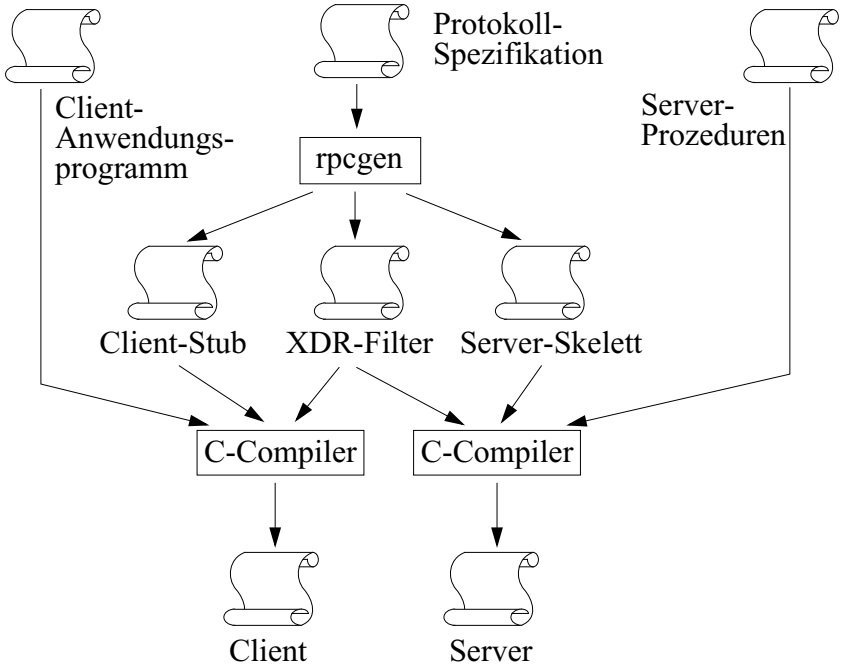
- Es gibt auch eine Broadcast-Variante:

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_broadcast*() waits for more replies.

Stub- und Filtergenerierung

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



- XDR: Standardrepräsentation von Daten bei der Kommunikation (d.h. Kodierungskonvention)
  - “Filter” Kodieren und Decodieren die Daten

Exkurs: Beispiel zu rpcgen

Die Ausgangsdatei add.x mit der *Protokollspezifikation*:

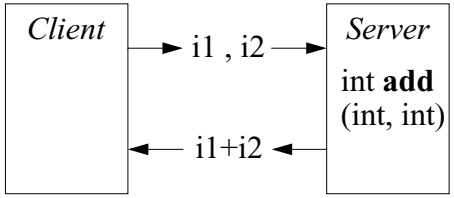
```
struct i_result
{ int x; };

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.

Exkurs-Ende (rpcgen)

Sicherheitskonzept des Sun-RPC

- Nur Unterstützung zur Authentifizierung
  - Autorisierung, d.h. Zugriffskontrolle, muss der Server selbst realisieren

- Feld im Header einer RPC-Nachricht spezifiziert eine der möglichen Authentifizierungsarten ("flavors"):
  - *NONE:* keine Authentifizierung
    - Client kann oder will sich nicht identifizieren
    - Server interessiert sich nicht für die Client-Identität
  - *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

- Mit dem RPC-Request wird folgende Struktur versandt:

```
{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 evtl. 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 dieses (historisch frühen) Ansatzes:
  - 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

- Im Unterschied zum UNIX-Flavor: Weltweit eindeutige Benutzernamen (“netname”) als String
  - in UNIX z.B. mittels user2netname() generiert aus 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”)*
  - wird zum Verschlüsseln der Nachrichten genutzt
- Mit jeder Request-Nachricht wird ein mit K kodierter Zeitstempel als Verifier mitgesandt
- 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)

Weiter auf Seite 441 (CORBA)

CORBA

- **C**ommon **O**bject **R**equest **B**roker **A**rchitecture
  - erste brauchbare (d.h. interoprable) Version: 1997 (CORBA 2.0)

eine *Architektur*, kein Produkt!

- Propagiert durch die OMG (Object Management Group)
  - herstellerübergreifendes Konsortium
  - Ziel: Bereitstellung von Konzepten für die Entwicklung verteilter Anwendungen
  - genauer: Definition und Entwicklung einer Architektur für kooperierende objektorientierte Softwarebausteine und Services in verteilten heterogenen Systemen (→ *"Middleware"*)

Beachte: *Objektorientierung* selbst ist eigentlich ein "altes" Konzept

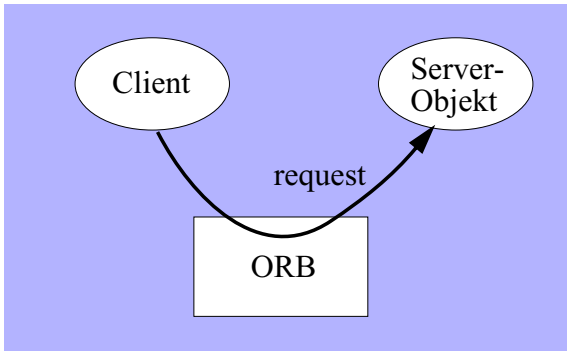
- Mitte der 1960er-Jahre (Programmiersprache "Simula")
- damals bereits fast alle Aspekte der Objektorientierung (Klassenhierarchien, virtuelle Klassen, Polymorphismus,...)

Man lese zu CORBA auch: Michi Henning: *The rise and fall of CORBA.* Commun. ACM, Vol. 51, No. 8 (August 2008), pp. 52-57

Mehr zur CORBA allgemein: Oliver Haase: *Kommunikation in verteilten Anwendungen (2. Auflage).* R. Oldenbourg Verlag, 2008, Kapitel 7

CORBA - Übersicht

- *Objektmodell*
- *IDL* (**I**nterface **D**escription **L**anguage) mit entsprechenden Generatoren und Compilern
- *ORB* (**O**bject **R**equest **B**roker) als Vermittlungsinfrastruktur

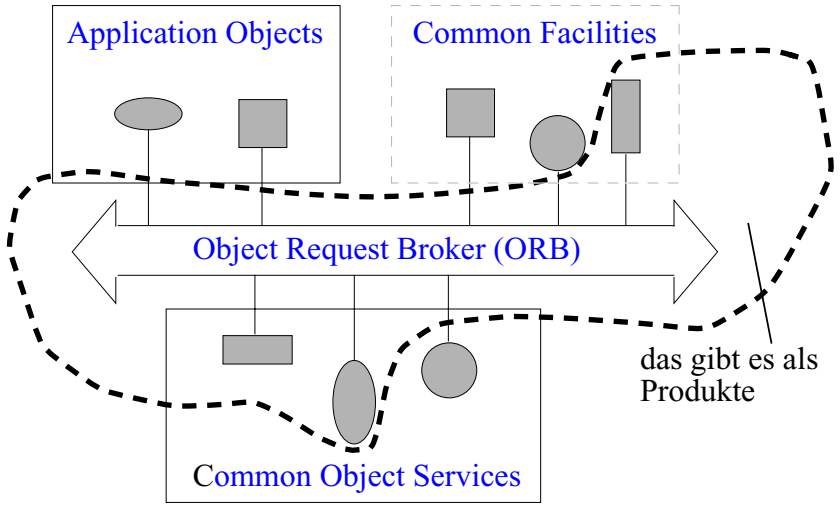


- *Systemfunktionen* in Form von Object Services
- Unterstützung von Anwendungen durch *Common Facilities*
- *Konventionen* bezüglich Schnittstellen, Protokollen etc.

- CORBA-Implementierungen sind also *Infrastrukturen* und unterstützen die *Ausführung* vert. objektorientierter Anwendungen in heterogenen Systemen
- *Entwurfs- und Spezifikationsaspekte* solcher Systeme werden dagegen mit anderen Konzepten unterstützt, z.B. *UML* ("Unified Modeling Language"), mit der u.a. Diagrammnotationen standardisiert werden

Object Management Architecture

- "OMA" ist eine *Referenzarchitektur*, welche die wesentlichen Bestandteile einer Plattform für verteilte objektorientierte Applikationen definiert



- *Application Objects*: Objekte der eigentlichen Anwendung
  - gehören damit nicht zur CORBA-Infrastruktur
- *ORB*: Vermittlung zwischen verschiedenen Objekten; Weiterleitung von Methodenaufrufen etc.
  → Ortstransparenz, Kommunikation,...
- *Object Services*: Schnittstelle zu standardisierten wichtigen Diensten
- *Common Facilities*: allgemein nützliche Funktionalität
  - nicht Teil aller CORBA-Implementierungen, oft sind nur wenige Funktionen davon realisiert

Object Services (1)

- COSS (Common Object Services Specification) als Basisdienste für eine systemweite Infrastruktur
  - mit objektorientierter Schnittstelle
  - nicht alle Dienste wurden aber vollständig spezifiziert (oder gar realisiert)

---

1) Ereignismeldung
   - Weiterleitung asynchroner Ereignisse an Ereigniskonsumenten
   - Einrichten von "event channels" mit Operationen wie push, pull,...
2) Persistenz
   - Dauerhaftes Speichern von Objekten auf externen Medien
3) Naming
   - Erzeugung von Namensräumen
   - Abbildung von Namen auf Objektreferenzen
   - Lokalisierung von Objekten
4) Trading
   - Matching von Services zu einer Service-Beschreibung eines Clients
5) Time
   - Uhrensynchronisation etc.
6) Security

Object Services (2)

7) Concurrency control

- Semaphore, Locks,...

8) Transaktionen

- 2-Phasen-Commit etc.

9) Replikation

- Sicherstellung der Konsistenz replizierter Objekte in einer verteilten Umgebung

10) Externalization

- Export von Objekten in sequentielle Dateien

- und noch einige weitere (hier nicht relevante) Services

Common Facilities

- Basisfunktionalität, die für ein breites Spektrum von Anwendungsbereichen nützlich ist, z.B.:
  - internationalization ("...will enable developers to use an application in their own language using their own cultural conventions...will allow the developer to use a culture's numeric and currency conventions...")
  - user interface
  - information management (z.B. Speicherung komplexer Objektstrukturen, Formatkonvertierungen,...)
  - systems management (z.B. Installation und Konfiguration von Objekten)
  - task management (Workflow, lange Transaktionen,...)

- Von den Common Facilities ist jedoch in Implementierungen und Produkten nur ein kleiner Teil tatsächlich realisiert worden

OMA Component Definitions

Object Request Broker - commercially known as CORBA, the ORB is the communications heart of the standard. It provides an infrastructure allowing objects to converse, independent of the specific platforms and techniques used to implement the objects. Compliance with the Object Request Broker standard guarantees portability and interoperability of objects over a network of heterogeneous systems.

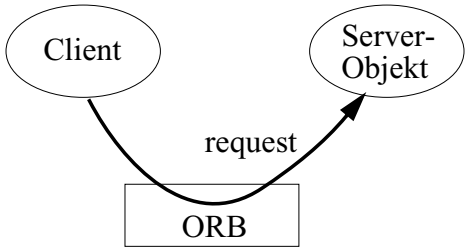
Object Services - these components standardize the life-cycle management of objects. Interfaces are provided to create objects, to control access to objects, to keep track of relocated objects, and to control the relationship between styles of objects (class management). Also provided are the generic environments in which single objects can perform their tasks. Object Services provide for application consistency and help to increase programmer productivity.

Common Facilities - Common Facilities provide a set of generic application functions that can be configured to the specific requirements of a particular configuration. These are facilities that sit closer to the user, such as printing, document management, database, and electronic mail facilities. Standardization leads to uniformity in generic operations and to better options for end users for configuring their working environments.

Quelle: www.omg.org/omaov.htm

Kommunikation zwischen Objekten

- Menge interagierender Objekte in zwei typischen Rollen
  - Client-Objekt (Aufrufer)
  - Server-Objekt



- Methodenaufruf durch requests unterschiedl. Semantik
  - *synchron* (mit Rückgabewerten; analog zu RPC)
  - *verzögert synchron* (Aufrufer wartet nicht auf das Ergebnis, sondern holt es sich später ab)
  - *one way* (asynchron: Aufrufer wartet nicht; keine Ergebnisrückgabe)

- Für den transparenten Transport eines Methodenaufrufs vom Client zum Server ist der ORB zuständig

Interface Description Language (IDL)

- Sprache zur Definition von Schnittstellen (Parameter, Attribute, Superklasse bzgl. Vererbung, Exceptions,...)
- Sprachneutral, aber lexikalisch an C++ angelehnt
- Bsp: **`oneway void move (in long x, in long y)`**

Eingabe-, Rückgabe- oder Ein- und Rückgabeparameter

- Grundsätzlicher Aufbau:

[oneway] <op_type_spec> <identifier> ($param_1$,....,$param_L$)
[raises($except_1$,...,$except_N$)] [context($name_1$,...,$name_M$)]

erlaubt Angabe möglicher Exceptions, die bei Durchführung der Methode auftreten können

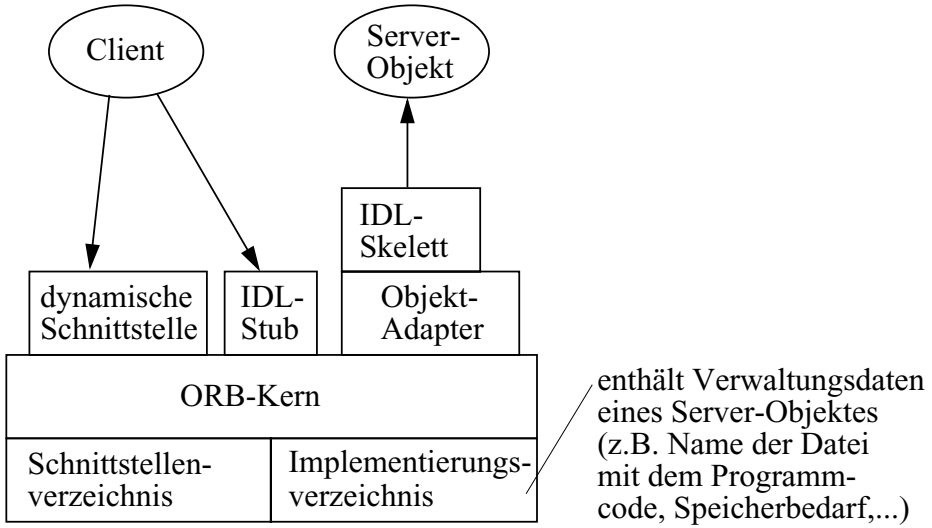
methodenspezifische Informationen, die dem Methodenaufruf mitgegeben werden können

- Aus einer Schnittstellenspezifikation in IDL erzeugt ein Compiler einen IDL-Stub für Clients und ein IDL-Skelett für Server

---

- Mehr zur CORBA-IDL siehe z.B.: Oliver Haase: *Kommunikation in verteilten Anwendungen (2. Auflage)*. R. Oldenbourg Verlag, 2008, Kapitel 7.2

ORB



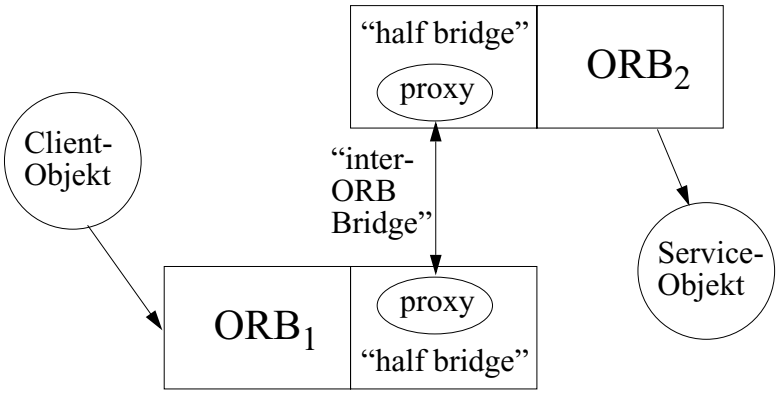
- ORB bietet Clients zwei Arten von Schnittstellen für den Methodenaufruf an
  - *statische Schnittstelle* (Erzeugung von Stubs aus der IDL-Beschreibung mit Compiler analog zu RPCs)
  - *dynamische Schnittstelle* (Client kann zur Laufzeit das Schnittstellenverzeichnis abfragen und einen geeigneten Methodenaufruf zusammenstellen)
- Objektadapter: Steuert anwendungsunabhängige Funktionen des Server-Objekts
  - u.A. Aktivierung des Server-Objektes bei Eintreffen eines requests, Authentifizierung von requests, Zuordung von Objektreferenzen zu Objektinstanzen etc.
  - zuständig ausserdem für Registrierung von Services
  - es gibt einen standardisierten Basic Object Adapter (BOA), der für viele Anwendungen ausreichende Grundfunktionalität bereitstellt

Server-Objekte

- Bereitsstellung von Services für Clients
  - analog zu Prozeduren, die im Rahmen von RPCs verwendet werden
- Objekte können ein aus der IDL-Spezifikation generiertes Objekt-Skelett nutzen
- Objekt muss sich beim lokalen Objekt-Adapter anmelden und dabei eine "server policy" angeben:
  - *Shared Server:* kann zusammen mit mehreren anderen aktiven Server-Objekten von einem einzigen Prozess verwaltet werden
  - *Unshared Server*
  - *Server per Method:* Start eines eigenen Prozesses bei Methodenaufruf
  - *Persistent Server:* ein Shared Server, der von CORBA initial bereits gestartet wurde
- Objekt muss sich ferner beim Implementierungs-verzeichnis anmelden
  - damit es bei einem Methodenaufruf durch Clients gefunden wird

ORB Bridges

- Interoperabilität von ORBs verschiedener Herstellerimplementierungen



- ORB Bridge: Formatkonvertierung und Weiterleitung eines requests etc. an einen anderen ORB
  - Schnittstellen und Konventionen für solche Bridges sind im CORBA-Standard festgelegt
  - Bridge besteht aus zwei Teilen mit einer CORBA-Schnittstelle, welche bei Bedarf Proxy-Objekte für die Aufrufkonvertierung erzeugen
- Inter-ORB-Kommunikation mittels GIOP (General Inter-ORB Protocol; z.B. TCP/IP-basiert)

CORBA - weitere Entwicklungen

Ab ca. 2000 entstand der Wunsch nach einer wesentlichen Erweiterung der CORBA-Funktionalität. Gründe:

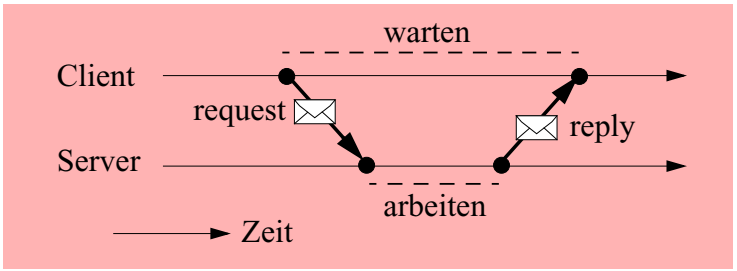
- Anforderungen durch E-Commerce-Anwendungen
- Ausbreitung des WWW (und später: Web-Services, SOAP,...)
- Aufkommen von Java (und später: EJB, Jini,...)
- Aufkommen mobiler Geräte

Dem sollte durch Weiterentwicklung ("CORBA 3.0") der Spezifikation Rechnung getragen werden:

- Messaging Service und asynchroner Methodenaufruf
- Objects by Value
- Persistente Objekte
  - "Abspeichern" von Objekten
- Komponenten-Modell
- Java-Unterstützung
  - Generieren von IDL aus Java bzw. Java-RMI ("reverse mapping")
- Firewall-Unterstützung
  - klassische Firewalltechnik (z.B. Services identifiziert mit Portnummern) versagt teilweise; Callbacks erscheinen als Aufruf von aussen...
- Minimum CORBA
  - Unterstützung von embedded systems (i.w. Weglassen von Dynamik)
- Realzeit-Unterstützung
- Fault Tolerant CORBA
  - durch redundante Einheiten

Messaging Service

- Asynchrones Kommunikationsparadigma
- Motivation:
  - mobile Geräte (PDA, Laptop,...) sind oft nicht online
  - bei sehr grossen verteilten Systemen sind fast immer einige Geräte bzw. Services nicht erreichbar (Netzprobleme etc.)
- CORBA basierte bisher auf einer engen ("synchronen") Kopplung von Client und Server



- Asynchron:
  - Entkopplung von Sender / Empfänger
  - Sender blockiert nicht solange, bis Nachricht angekommen ist
  - Nachricht kann von Hilfsinstanzen zwischengespeichert werden, bis Kommunikationspartner erreichbar ist ("store & forward")
  - Antwort kann evtl. von anderem Client als ursprünglichem Sender entgegengenommen werde

Asynchronous Method Invocation

- Bisherige Möglichkeiten eines Methodenaufrufs in CORBA:
  - *synchron* (insbes. bei Rückgabewerten; analog zu RPC)
  - *verzögert synchron* (Aufrufer wartet nicht auf das Ergebnis, sondern holt es sich später ab)
  - *one way* (Aufrufer wartet nicht) mit "best effort"-Semantik ("fire and forget")

gedacht war an UDP-Implementierung; Semantik (z.B. Fehlermeldung bei Misslingen?) implementierungsabhängig

---

- Neu: Asynchronous Method Invocation (AMI)
  - bisher eher umständlich und fehleranfällig durchThreads zu simulieren
- Zwei Aufruftechniken bei AMI:
  - (1) Callback
    - Client gibt dem Aufruf eine Objektreferenz für die Antwort mit
    - Callback-Objekt kann sich im Prinzip irgendwo befinden
    - Kommunikations-Exceptions werden im Callback-Objekt ausgelöst
  - (2) Polling
    - Client erhält sofort ein Objekt zurück, das er für Polling oder zum Warten auf Antwort nutzen kann

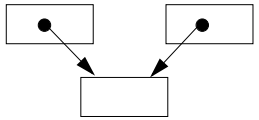
Time-independent Invocation

- Aufruf von Objekten, die nicht aktiv sind oder zeitweise nicht erreichbar sind
- Aufruf-Nachrichten werden von zwischengeschalteten "Router Agents" verwaltet
  - Store and Forward-Prinzip
  - Router Agent beim Client ermöglicht disconnected operations
  - Router Agent beim Server kann dessen Eingangsqueue verwalten
- "Interoperable Routing Protocol" sorgt dafür, dass Router Agents verschiedener Hersteller interagieren
- Sogen. Quality of Service steuerbar (als "Policy")
  - z.B. max. Round Trip-Zeit: dadurch brauchen Router Agents die Nachrichten nicht beliebig lange aufbewahren
  - oder z.B. Aufrufreihenfolge: Soll Router seine gespeicherten Aufträge zeitlich geordnet oder nach Prioritäten oder... ausliefern?

Objects by Value

- Bisher war in CORBA nur *Referenzübergabe* möglich
  - um es Objekten zu gestatten, Methoden anderer Objekte aufzurufen, konnten bisher Objektreferenzen als Parameter übergeben werden
  - Objekt selbst bleibt aber "am Platz", Aufruf wird also immer als Fernaufruf über das Netz geschickt
  - ferner besteht bei Referenzübergabe die Gefahr unerwünschter Alias-Effekte (unbedachte Rückwirkungen auf das "Originalobjekt")

- Bei *Wertübergabe* wird das Objekt serialisiert und im Adressraum des Empfängers eine Kopie angelegt
  - Marshalling des Objektzustandes (d.h. der *Daten*)
  - auf Empfängerseite existiert eine sogen. Factory, die das Objekt als Kopie (mit eigener Identität) erzeugt
  - was geschieht bei Alias-Zeigern bzw. Zyklen bei der Serialisierung komplexer Strukturen?



- Bei heterogenen Umgebungen: Wie transportiert man das *Verhalten* des Objektes zum Empfänger?
  - es handelt sich um ausführbaren Code (für welche Plattform?)
  - kann von verschiedenen Sprachen (C, Java,...) erzeugt worden sein
  - Factory beim Empfänger muss sich den passenden Code besorgen (aus lokaler Bibliothek, übers Netz,...)

- Leider können jedoch keine normalen CORBA-Objekte per Value übergeben werden, nur sogen. "valuetypes"
  - neues Konstrukt der IDL (→ für Anwender dadurch kompliziert)
  - Wertübergabe beliebiger Objekte wäre zu aufwändig

Real-Time CORBA

- Ziel: Vorhersagbares Ende-zu-Ende-Verhalten (insbesondere beim entfernten Methodenaufruf)
  - sowohl für "Hard Real-Time"
  - als auch für "Soft Real-Time" (mit nur statistischen Aussagen)

- Setzt zugrundeliegendes Realzeit-Betriebssystem voraus

- Einige Mechanismen:
  - Prioritäten
  - Timeouts für Aufrufe
  - Multithreading (mit geeignetem Scheduling), Threadpools
  - private (statt gemultiplexte) Verbindungen
  - austauschbare Kommunikationsprotokolle

- Anwendungsbereiche z.B.:
  - Prozessautomatisierung
  - eingebettete Systeme
  - ...

CORBA-Probleme

- Die Weiterentwicklung von CORBA geriet ins Stocken
  - zu weitreichende Anforderungen → komplex / ineffizient
  - kommerzielle Implementierungen zögerlich
  - fehlende Unterstützung durch Microsoft (eigene Architektur)
  - OMG versuchte, es jedem Recht zu machen (widersprüchliche Interessen, barocke Konstrukte durch Kompromisse,...)
  - aufkommende Konkurrenzsysteme, die z.T. besser an die neuen Anforderungen angepasst sind

---

Konkurrenzsysteme bzw. alternative Ansätze sind z.B.:

- Microsoft: DCOM und .Net
- Java-Technologie
  - z.B. Enterprise Java Beans (EJB), RMI
- Web-Services (und verwandte Systeme)
  - XML, WSDL, SOAP,...
  - Integrationsplattformen (z.B. WebSphere von IBM)