Namens-verwaltung

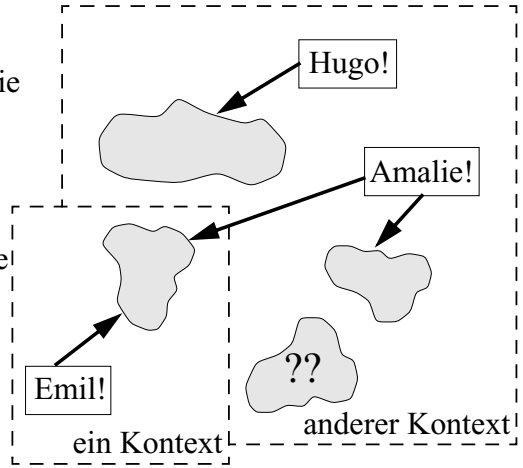
Namen und Namensverwaltung

Namen sind Schall und Rauch

Nomen est omen

- *Namen* sind Symbole, die typischerweise durch Zeichenketten repräsentiert werden
  - benutzerorientierte Namen haben im Unterschied zu Adressen (oder maschinenorientierten Namen) i.a. keine feste Länge

- Dienen der (eindeutigen) *Bezeichnung von Objekten*
  - daher oft auch “Bezeichner”
  - es gibt auch *anonyme* Objekte (z.B. dynamische Variablen, die mit “new” erzeugt werden)
  - ein Objekt kann u.U. mehrere Namen haben (“*alias*”)
  - innerhalb eines *Kontextes* sollte ein Name *eindeutig* sein
  - Benutzer soll ein Objekt einfach *umbennen* können
  - gleicher Name kann zu *verschiedenen Zeiten* unterschiedliche Objekte bezeichnen



- Beispiele für bezeichnete Objekte
  - in Programmiersprachen:

    Variablen, Prozeduren, Datentypen, Konstanten...
  - in verteilten Systemen:

    Dienste, Server, Maschinen, Benutzer, Dateien, Betriebsmittel...

Zweck von Namen

Typ, Gestalt, Zweck...

- Geben Aufschluss über die Art eines Objektes
  - falls Name (für Benutzer) sinnvoll gewählt
  - z.B. Konventionen xyz.c, xyz.o, xyz.ps oder “printer”

- Dienen der *Identifizierung* von Objekten
  - daher oft auch “Identifikator” für “Name”
  - Sprechweise oft: “Objekt A” statt “das mit ‘A’ bezeichnete Objekt”

- Ermöglichen die *Lokalisierung* von Objekten
  - zwecks Manipulation der Objekte
  - über den Namen besteht eine Zugriffsmöglichkeit auf das Objekt selbst
  - Namen selbst sind aber oft unabhängig von der Objektlokation
  - besondere Herausforderung: Lokalisieren von *mobilen* Objekten

- Sind URLs Namen?
  - oder eher Adressen?
  - www.fuzzycomp.eu/Studium/bewerbung.html
  - 121.73.129.200/Studium/bewerbung.html

Namen und Adressen

- Jedes Objekt hat eine Adresse
  - Speicherplatzadressen
  - Internetadressen (IP-Nummern)
  - Netzadressen
  - Port-Nummer bei TCP
  - ...

- Adressen sind “physische” Namen

Namen der untersten Stufe

- Adressen ermöglichen die *direkte Lokalisierung* eines Objektes

- Adressen sind ebenfalls innerhalb eines Kontextes (“Adressraum”) eindeutig

- Adresse eines Objektes ist u.U. *zeitabhängig*
  - mobile Objekte
  - “relocatable”

- *Dagegen*: Name eines Objektes ändert sich i.a. nicht
  - vielleicht aber bei Heirat, Zuweisung eines Alias...!

- Entkoppelung von Namen und Adressen unterstützt die *Ortstransparenz*

- Zuordnung Name → Adresse nötig
  - vgl. persönliches Adressbuch
  - “Binden” eines Namens an eine Adresse

Binden

- Binden = Zuordnung Name → Adresse
  - konzeptuell daher auch: Name → Objekt
  - Namen, die bereits Ortsinformationen enthalten: “impure names”

- *Binden bei Programmiersprachen:*
  - Beim Übersetzen / Assemblieren
    → “relative” Adresse
  - Durch Binder (“linker”) oder Lader
    → “absolute” Adresse
  - Ggf. Indirektion durch das Laufzeitsystem
    - z.B. bei Polymorphie objektorientierter Systeme

- *Binden von Dienstaufrufen bei klassischen Systemen*
  - Dienstaufruf durch Trap / Supervisor-Call (“SVC”)
    - Name = SVC-Nummer (oder “symbolische” Bezeichnung)
  - Bei *Systemstart* wird eine Verweistabelle angelegt
    - “SVC table”, “switch vector”
  - Dienstadresse ändert sich bis zum reboot nicht

- *Binden in verteilten / offenen Systemen*
  - Dienste entstehen dynamisch, werden ggf. verlagert
    - haben ggf. unterschiedliche Lebenszyklen und -dauer
  - Binden muss daher ebenfalls *dynamisch* (“zur Laufzeit” bzw. beim Objektzugriff) erfolgen!

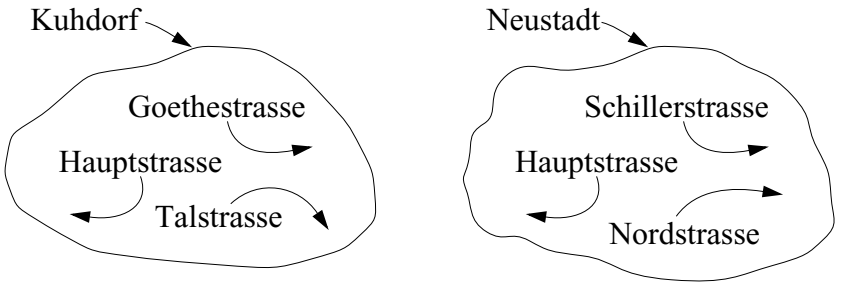
Namenskontext

- Namen werden relativ zu einem *Kontext* interpretiert (Namensraum)
  - “relative Namen” (gleiche Namen in verschiedenen Kontexten möglich)
  - *Interpretation* = Abbildung auf die gebundene Adresse oder einen Namen niedrigerer Stufe
  - Interpretation erfolgt oft mehrstufig, z.B.: Dateiname → Adresse des Kontrollblocks → Spur / Sektor auf einer Platte

- Namen sollen innerhalb eines Kontextes eindeutig sein
  - bzw. durch zusätzliche Attribute eindeutig identifizierbar sein

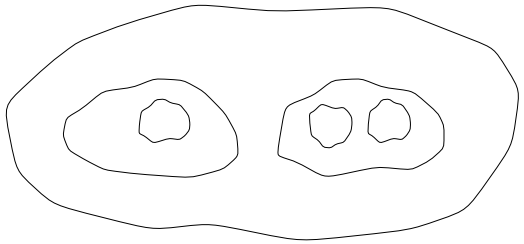
- Falls nur ein einziger Kontext existiert: *flacher Namensraum* (aus “absoluten Namen”)
  - Partition des Namensraum wird als “Domäne” bezeichnet

- Namenskontexte sind (i.a. abstrakte) Objekte, die selbst wieder einen Namen haben können
  - z.B. benannte Dateiverzeichnisse (“directory”)
  - übergeordneter Kontext → *Hierarchie*



Hierarchische Namensräume

- Baumförmige Struktur von Namenskontexten



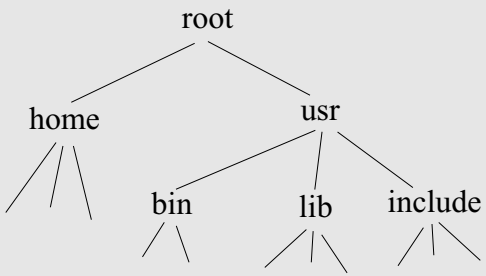
- Beispiel: Adressen im Briefverkehr
  - "Hans Meier, Deutschland" genügt nicht...

Sind das nicht eher Adress-räume als Namensräume?

- Beispiel: Telefonsystem
  - Landeskennung
  - Ortsnetzkennung
  - Teilnehmerkennung

32168 ist ein relativer Name, der z.B. im Kontext 08977 interpretiert werden muss

- Beispiel: UNIX-Dateisystem



Hierarchische Namensräume (2)

- Eignen sich gut für verteilte Systeme
  - besser als flache Namensräume
  - leichter skalierbar (z.B. zur Gewährleistung der Eindeutigkeit)
  - dezentrale Verwaltung der Kontexte durch eigenständige Autoritäten, die wieder anderen Autoritäten untergeordnet sind
  - Namensinterpretation stufenweise durch verteilte Instanzen
  - erleichtert Systemrekonfiguration
  - eindeutige absolute Namen durch Angabe des ganzen Pfades

- Strukturierte Namen
  - bestehen aus mehreren Komponenten
  - Komponenten bezeichnen typischerweise Kontexte
  - Bsp: root/usr/bin
  - Bsp: Meier.Talweg 2.Kuhdorf.Oberpfalz.Deutschland
  - Bsp: +49 08977 32168 (präfixfreier Code!)
  - oft geographisch oder thematisch gegliedert

- *Synonyme Namen* bezeichnen das gleiche Objekt
  - Bsp: der relative Name 'c' im Kontext 'a' bezeichnet das gleiche Objekt wie der absolute Name 'a.c'

- *Alias-Namen*: Synonyme im gleichen Kontext

Namensverwaltung ("name service")

- Verwaltung der Zuordnung Name → Adresse
  - Eintragen: "bind (Name, Adresse)" sowie Ändern, Löschen etc.
  - Eindeutigkeit von Namen garantieren
  - Zusätzlich ggf. Verwaltung von Attributen der bezeichneten Objekte

- Auskünfte ("Finden" von Ressourcen, "lookup")
  - z.B. Adresse zu einem Namen ("resolve": Namensauflösung)
  - z.B. alle Dienste mit gewissen Attributen (etwa: alle Postscript-Drucker) "yellow pages" ↔"white pages"

- Evtl. Schutz- und Sicherheitsaspekte
  - Capability-Listen, Schutzbits, Autorisierungen...
  - Dienst selbst soll hochverfügbar und sicher (z.B. bzgl. Authentizität) sein

- Evtl. Generierung eindeutiger Namen
  - UUID (Universal Unique Identifier)
  - innerhalb eines Kontextes (z.B. mit Zeitstempel oder lfd. Nummer)
  - bzw. global eindeutig (z.B. eindeutigen Kontextnamen als Präfix vor knotenlokale laufende Nummer; evtl. auch lange Zufallsbitfolge)

Vgl. "klassische" Dienste beim Telefonsystem:

- Telefonbuch
- Auskunft

} Abbildung Name → Telefonnummer

- ggf. mehrstufig / dezentral: Auslandsauskunft wendet sich an Ortsauskunft im Ausland... (→ hierarchische Namenskontexte notwendig!)

- lokale Telefonbücher sowie Ortsvorwahlverzeichnis sind repliziert
  - sonst Überlastung des zentralen Dienstes
  - Problem der verzögerten Aktualisierung (veraltete Information)

- "gelbe Seiten": Suche nach Dienst über Attribute

Zufällige UUIDs? Echter Zufall?

http://webnz.com/robert/true_rng.html

The usual method is to amplify *noise* generated by a *resistor* (Johnson noise) or a semi-conductor *diode* and feed this to a comparator or Schmitt trigger. If you sample the output (not too quickly) you (hope to) get a series of bits which are statistically independent.

www.random.org

Random.org offers *true random numbers* to anyone on the *Internet*.

Computer engineers chose to introduce randomness into computers in the form of *pseudo-random number generators*. As the name suggests, pseudo-random numbers are not truly random. Rather, they are computed from a mathematical formula or simply taken from a precalculated list.

For *scientific experiments*, it is convenient that a series of random numbers can be replayed for use in several experiments, and pseudo-random numbers are well suited for this purpose. For *cryptographic use*, however, it is important that the numbers used to generate keys are not just seemingly random; they must be truly *unpredictable*.

The way the random.org random number generator works is quite simple. A radio is tuned into a frequency where nobody is broadcasting. The *atmospheric noise* picked up by the receiver is fed into a Sun SPARC workstation through the microphone port where it is sampled by a program as an eight bit mono signal at a frequency of 8KHz. The upper seven bits of each sample are discarded immediately and the remaining bits are gathered and turned into a stream of bits with a high content of entropy. *Skew correction* is performed on the bit stream, in order to insure that there is an approximately even distribution of 0s and 1s.

The skew correction algorithm used is based on transition mapping. Bits are read two at a time, and if there is a *transition between values* (the bits are 01 or 10) one of them - say the first - is passed on as random. If there is no transition (the bits are 00 or 11), the bits are discarded and the next two are read. This simple algorithm was originally due to *Von Neumann* and completely eliminates any bias towards 0 or 1 in the data. [***Why??***]

A Kr85-based Random Generator

http://www.fourmilab.ch/hotbits/ ...by John Walker

The *Krypton-85* nucleus (the 85 means there are a total of 85 protons and neutrons in the atom) spontaneously turns into a nucleus of the element *Rubidium* which still has a sum of 85 protons and neutrons, and a *beta particle (electron) flies out*, resulting in no net difference in charge. What's interesting, and ultimately useful in our quest for random numbers, is that even though we're absolutely certain that if we start out with, say, 100 million atoms of Krypton-85, 10.73 years later we'll have about 50 million, 10.73 years after that 25 million, and so on, there is *no way even in principle* to predict when a given atom of Krypton-85 will decay into Rubidium.

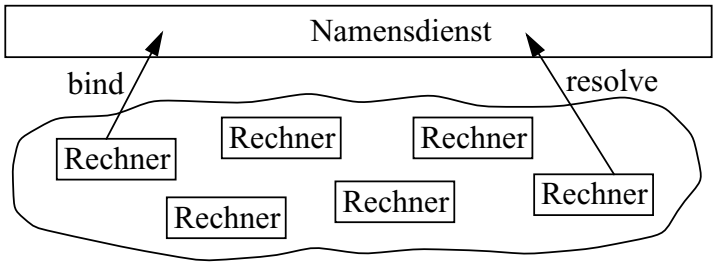
So, given a Krypton-85 nucleus, there is *no way whatsoever to predict when it will decay*. If we have a large number of them, we can be confident half will decay in 10.73 years; but if we have a single atom, pinned in a laser ion trap, all we can say is that is there's even odds it will decay sometime in the next 10.73 years, but as to precisely when we're fundamentally quantum clueless. The only way to know when a given Krypton-85 nucleus decays is after the fact--by detecting the ejecta.

This *inherent randomness* in decay time has profound implications, which we will now exploit to generate random numbers. For if there's no way to know when a given Krypton-85 nucleus will decay then, given an collection of them, there's no way to know when the next one of them will shoot its electron bolt.

Since the time of any given decay is random, then *the interval between two consecutive decays is also random*. What we do, then, is measure a pair of these intervals, and *emit a zero or one bit based on the relative length of the two intervals.* If we measure the same interval for the two decays, we discard the measurement and try again, to avoid the risk of inducing bias due to the resolution of our clock.

To create each random bit, we wait until the first count occurs, then measure the time, T1, until the next. We then wait for a third pulse and measure T2, yielding a pair of durations... if *T1 is less than T2* we emit a *zero* bit; if T1 is *greater* than T2, a *one* bit. In practice, to avoid any residual bias resulting from non-random systematic errors in the apparatus or measuring process consistently favouring one state, the sense of the comparison between T1 and T2 is reversed for consecutive bits.

Verteilte Namensverwaltung



logisch ein einziger Dienst; tatsächlich verteilt realisiert

- Jeder Kontext wird (logisch) von einem autonomen *Nameserver* verwaltet
  - evtl. ist ein Nameserver für mehrere Kontexte zuständig
  - evtl. Aufteilung / Replikation des Nameservers
    → höhere Effizienz, Ausfallsicherheit
- Typischerweise *hierarchische Namensräume*
  - entsprechend strukturierte Namen
  - entsprechend kanonische Aufteilung der Verwaltungsaufgaben
  - Zusammenfassung Namen gleichen Präfixes vereinfacht Verwaltung
- Typisch: *kooperierende* Nameserver, die den gesamten Verwaltungsdienst realisieren
  - hierzu geeignete Architektur der Server vorsehen
  - Protokoll zwischen den Nameservern (für Fehlertoleranz, update der Replikate etc.)
  - Dienstschnittstelle wird i.a. durch lokale Nameserver realisiert

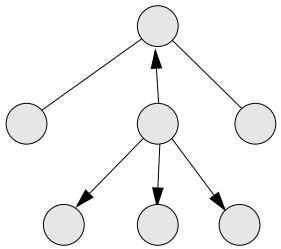
bzw. "user agent"

- *Annahmen*, die Realisierungen i.a. zugrundeliegen:
  - *lesende* Anfragen viel häufiger als schreibende ("Änderungen")
  - *lokale* Anfragen (bzgl. eigenem Kontext) dominieren
  - seltene, temporäre *Inkonsistenzen* können toleriert werden

ermöglicht effizientere Realisierungen (z.B. Caching, einfache Protokolle...)

Namensinterpretation in verteilten Systemen

- Ein Nameserver kennt den Nameserver der nächst höheren Stufe



- Ein Nameserver kennt alle Nameserver der untergeordneten Kontexte (sowie deren Namensbereiche)

- Hierarchiestufen sind i.a. klein (typw. 3 oder 4)

- Blätter verwalten die eigentlichen Objektadressen und bilden die Schnittstelle für die Clients

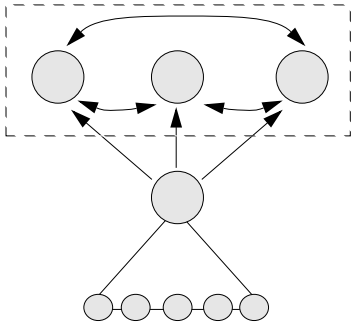
- Nicht interpretierbare Namen werden an die nächst höhere Stufe weitergeleitet (bei strukturierten Namen!)

Broadcast

- falls zuständiger Nameserver unbekannt
  ("wer ist für XYZ zuständig?" oder: "wer ist hier der Nameserver?")
- ist aufwendig, falls nicht durch Hardware etc. unterstützt (wie z.B. bei LAN oder Funknetzen)
- nur in begrenzten Kontexten anwendbar

Repliklation von Nameservern

- Zweck: Erhöhung von Effizienz und Fehlertoleranz
- Vor allem auf höherer Ebene relevant
  - dort viele Anfragen
  - Ausfall würde grösseren Teilbereich betreffen

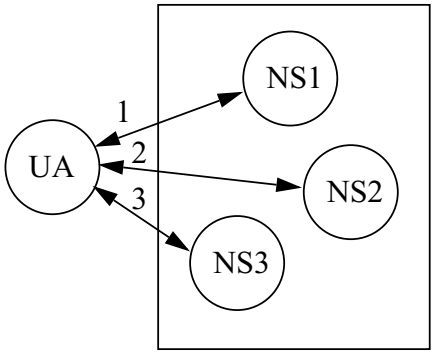


- Server kennt alle übergeordneten Server
- Broadcast an ganze Servergruppe, oder Einzelnachricht an "nächsten" Server; anderen Server erst nach Ablauf eines Timeouts befragen

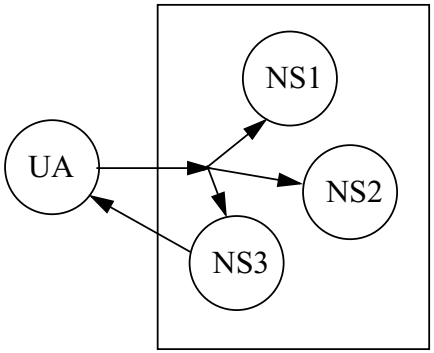
- Replizierte Server konsistent halten
  - ggf. nur von Zeit zu Zeit gegenseitig aktualisieren (falls veraltete Information tolerierbar)
  - Update auch dann sicherstellen, wenn einige Server zeitweise nicht erreichbar sind (periodisches Wiederholen von update-Nachrichten)
  - Einträge mit Zeitstempel versehen → jeweils neuester Eintrag dominiert (global synchronisierte Zeitbasis notwendig!)
- Symmetrische Server / Primärserver-Konzept:
  - *symmetrische Server*: jeder Server kann updates initiieren
  - *Primärserver*: nur dieser nimmt updates entgegen
    - Primärserver aktualisiert gelegentlich "read only" Sekundärserver
    - Rolle des Primärservers muss im Fehlerfall von einem anderen Server der Gruppe übernommen werden

Strukturen zur Namensauflösung

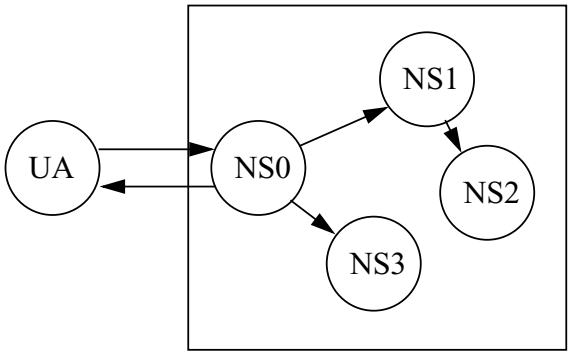
- User Agent (UA) bzw. “Name Agent” auf Client-Seite
  - hinzugebundene Schnittstelle aus Bibliothek, oder
  - eigener Service-Prozess



Iterative Navigation: NS1 liefert Adresse eines anderen Nameservers zurück bzw. UA probiert einige (vermutlich) zuständige Nameserver nacheinander aus



Multicast-Navigation: Es antwortet derjenige, der den Namen auflösen kann (u.U. auch mehrere)



“Rekursive” Namensauflösung, wenn ein Nameserver ggf. den Dienst einer anderen Ebene in Anspruch nimmt

Serverkontrollierte Navigation: Der Namensdienst selbst in Form des Serververbundes kümmert sich um die Suche nach Zuständigkeit

Caching von Bindungsinformation

- Zweck: Leistungsverbesserung, insbesondere bei häufigen nichtlokalen Anfragen

(a) Abbildung Name → Adresse des *Objektes*

(b) Abbildung Name → Adresse des *Nameservers* der tiefsten Hierarchiestufe, der für das Objekt zuständig ist

- Zuordnungstabelle (Cache) wird lokal gehalten
- vor Aufruf eines Nameservers überprüfen, ob Information im Cache
- Information könnte allerdings veraltet sein!
- Platz der Tabelle ist beschränkt → unwichtige / alte Einträge verdrängen
- Neue Information wird als Seiteneffekt einer Anfrage eingetragen

---

- Vorteil von (b): Inkonsistenz aufgrund veralteter Information kann vom Nameservice entdeckt werden
  - veralteter Cache-Eintrag kann transparent für den Client durch eine automatisch abgesetzte volle Anfrage ersetzt werden
- Bei (a) muss der *Client* selbst falsche Adressen *beim Zugriff* auf das Objekt erkennen und behandeln

---

- Caching kann bei den Clients stattfinden (z.B. im Web-Browser) und / oder bei den Nameservern

Beispiele für Namensdienste

- Domain Name System (DNS) im Internet
  - in der UNIX-Welt oft eingesetzte Implementierung: BIND ("Berkeley Internet Domain Name")
- Neben DNS als Quasistandard existiert u.a. die internationale X500-Norm ("CCITT / ISO-OSI directory service") als globaler Namensdienst
  - Normung der Struktur der Einträge und der Protokolle
  - OSF-DCE nutzt X500 als zellübergreifenden Namensservice
  - neue Attribute definierbar (Name plus Syntaxdefinition in ANS.1)
- LDAP (Lightweight Directory Access Protocol)
- Network Information Service (NIS)
  - entwickelt von Sun Microsystem (ursprünglich: "Yellow Pages")
  - 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 besser
  - z.B. Domain-Namen wie www.nanocomp.uni-cooltown.eu
  - gut zu merken; relativ unabhängig von spezifischer Maschine
  - muss *vor* Verwendung bei Internet-Diensten (ssh, ftp, E-Mail, WWW...) 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.

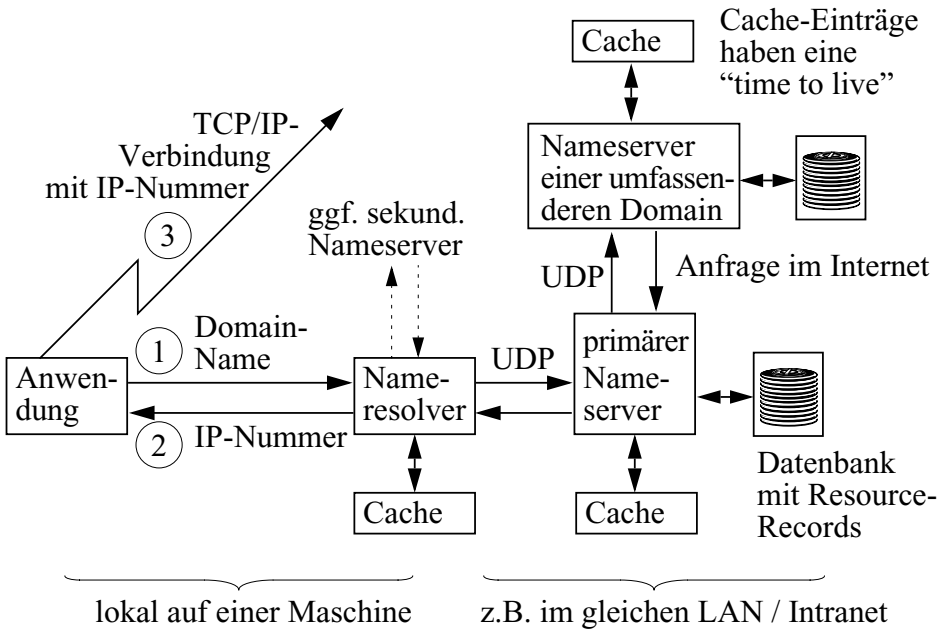
eu
uni-cooltown.eu
informatik.uni-cooltown.eu
nano.informatik.uni-cooltown.eu
pc6.nano.informatik.uni-cooltown.eu

- 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 "dependability" 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
fb22.tu-da.de          IN  MX  mailgate.fb22.tu-da.de
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
laserjet               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.)
  - MX: forwarding von E-mail ("Mail eXchanger")
  - 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.

```
> sun20
Name:    sun20.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
```

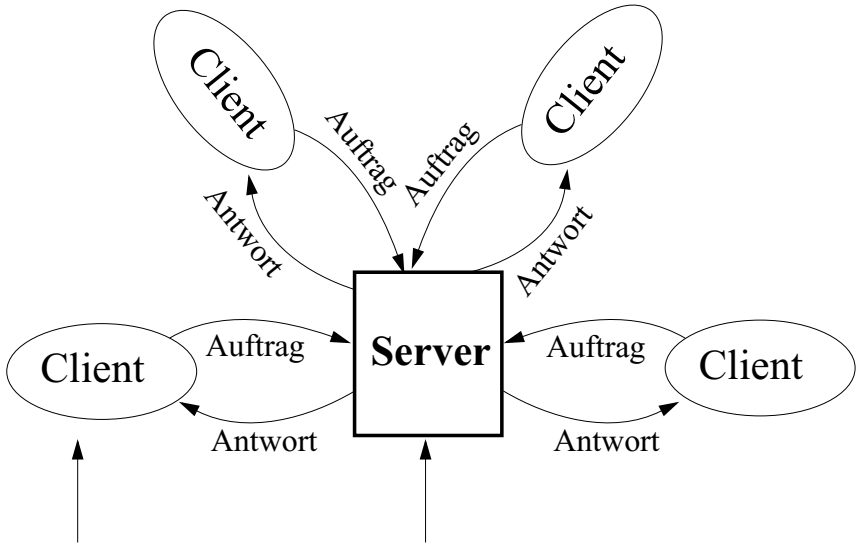
Dies deutet auf einen "round robin"-Eintrag hin: Der Nameserver von google.com ändert alle paar Minuten die Reihenfolge der Einträge, die bei anderen Nameservern auch nur einige Minuten lang gespeichert bleiben dürfen. Da Anwendungen i.a. den ersten Eintrag nehmen, wird so eine Lastverteilung auf mehrere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

---

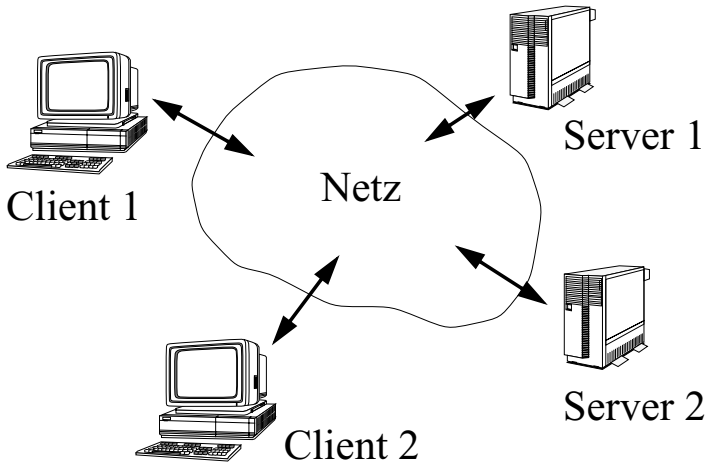
- Modell ist für viele Zwecke geeignet, jedoch nicht für alle (z.B. Pipelines, "peer-to-peer", asyn. Mitteilung)!

Beispiel Puffer als Pipeline:

- 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

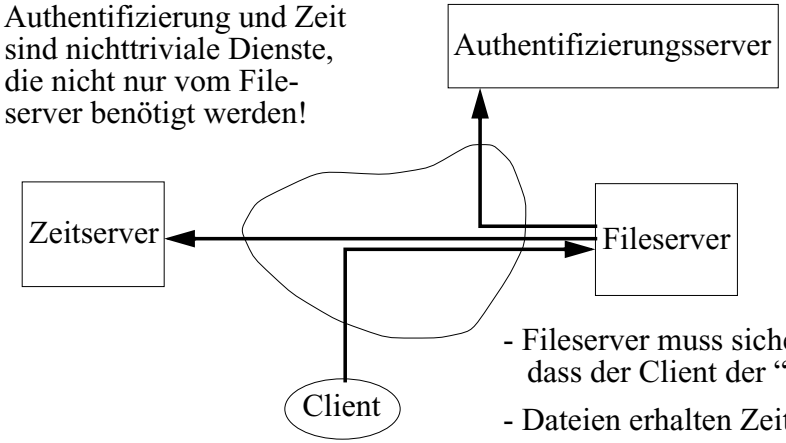
- Andere, leistungsfähigere Rechner als Server
  - "zentrale" Dienste (z.B. Speicherserver)
  - gemeinsam benutzte Betriebsmittel

- Im allgemeinen müssen sich aber Server- und Client-*Prozesse* nicht auf dedizierten Rechnern befinden!

Client/Server-Rollen

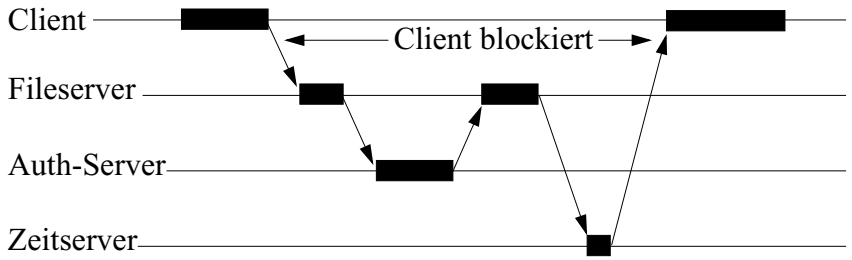
- Server müssen ggf. zur Durchführung eines Dienstes die Dienstleistungen anderer Server in Anspruch nehmen

- Authentifizierung und Zeit sind nichttriviale Dienste, die nicht nur vom File-server benötigt werden!



- Fileserver muss sicherstellen, dass der Client der "richige" ist
- Dateien erhalten Zeitstempel

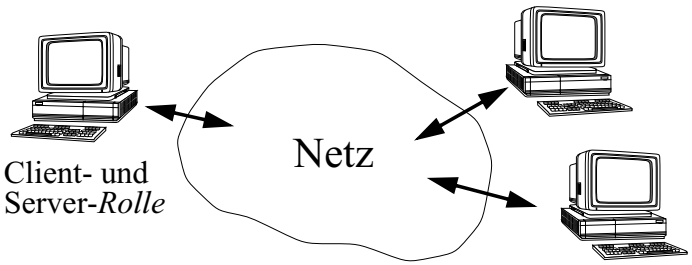
- Fileserver hat prinzipiell die *Rolle* eines Servers, zwischenzeitlich jedoch die *Rolle* eines Clients



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, Speicherausbau), um für den "Besitzer" leistungstransparent zu sein
  - geringere Stabilität (Besitzer kann seine Maschine ausschalten...)
  - Datensicherung muss ggf. dezentral durchgeführt werden
  - Sicherheit und Schutz kritisch: Lizenzen, Viren, Integrität...

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.a. 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!

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

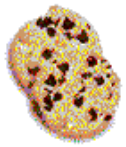
Sind Webserver zustandslos?

- 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 darf dieses später wieder lesen

Cookies



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

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

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

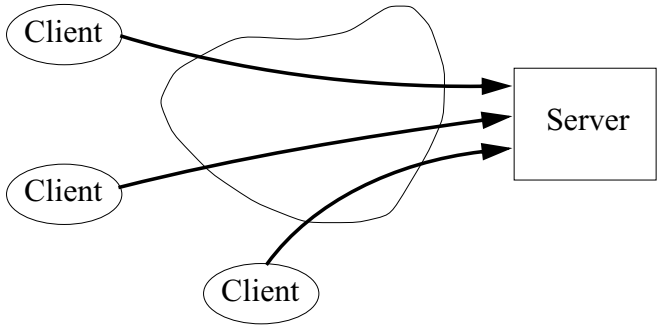
This simple mechanism provides a powerful new tool which enables a host of new types of applications to be written for web-based environments. Shopping applications can now store information about the currently selected items, for fee services can send back registration information and free the client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

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

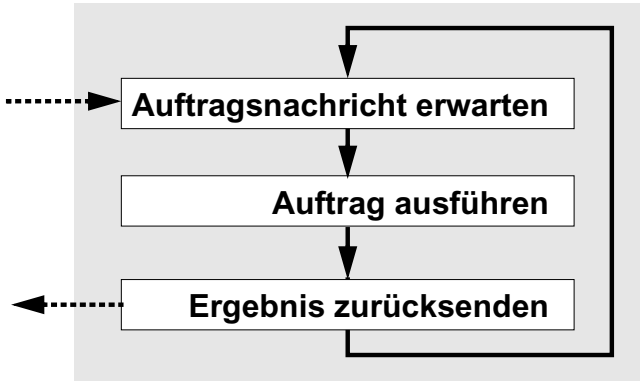
- Denkübung: Müssen Proxy-Server geeignete Massnahmen vorsehen?
- Übung: Man finde heraus, was doubleclick.net macht (und wie)

Iterative Server

- Problem: Viele “gleichzeitige” Aufträge



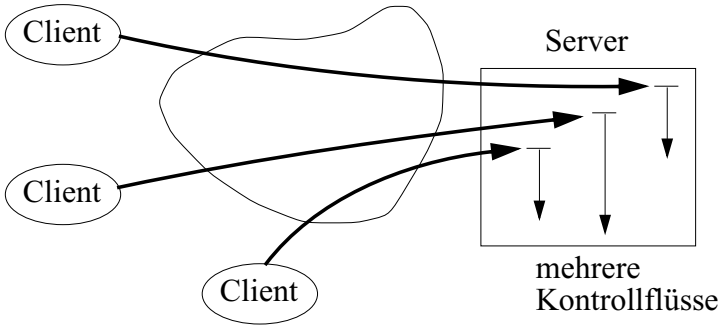
- *Iterative Server* bearbeiten nur einen Auftrag pro Zeit



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

Konkurrente ("nebenläufige") Server

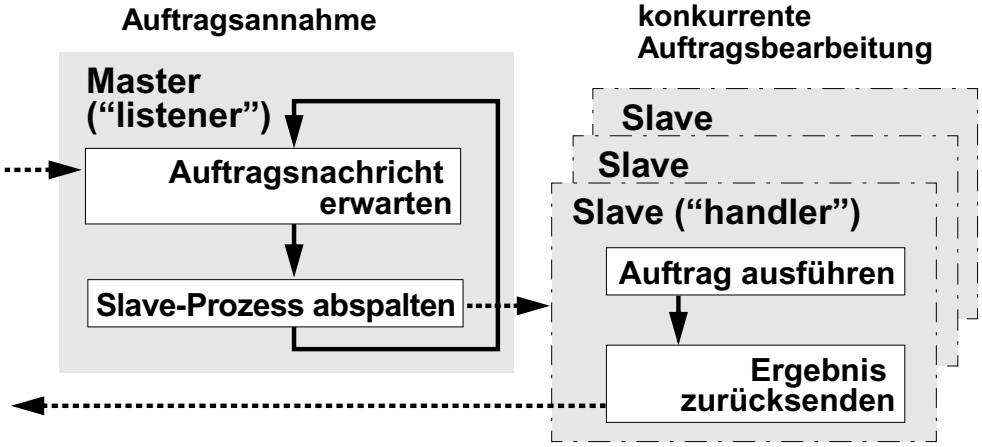
- Gleichzeitige Bearbeitung mehrerer Aufträge
  - sinnvoll (d.h. effizienter für Clients) bei langen Aufträgen (z.B. in Verbindung mit E/A)
  - Beispiel: Web-Server oder Suchmaschinen



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

Konkurrente Server mit dynamischen Handler-Prozessen

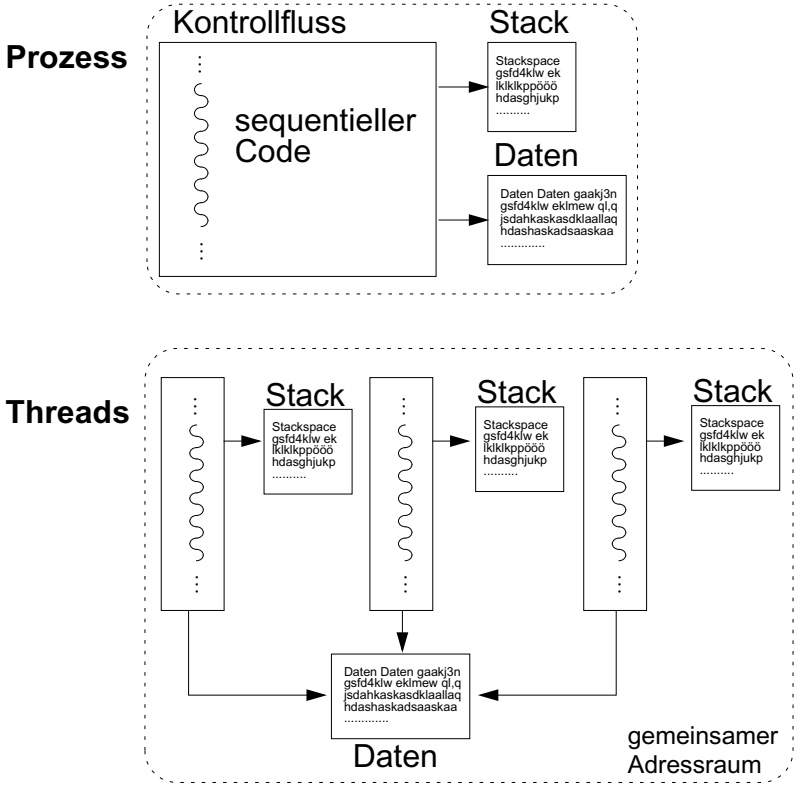
- Für jeden Auftrag gründet der *Master* einen neuen *Slave*-Prozess und wartet dann auf einen neuen Auftrag
  - neu gegründeter Slave ("handler") übernimmt den Auftrag
  - Client kommuniziert dann ggf. direkt mit dem Slave (z.B. über dynamisch eingerichteten Kanal bzw. Port)
  - Slaves sind ggf. Leichtgewichtsprozesse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



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

Threads

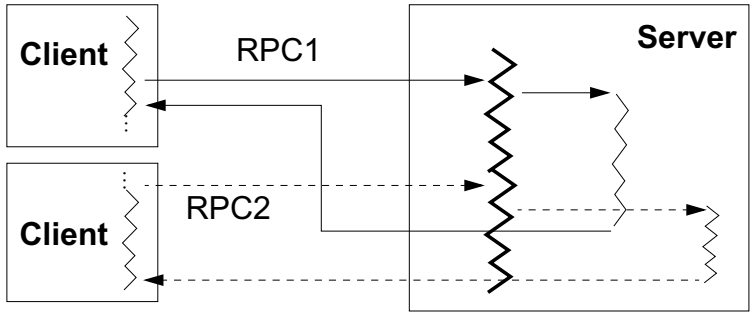
- Threads = leichtgewichtige Prozesse mit gemeinsamem Adressraum



- 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?

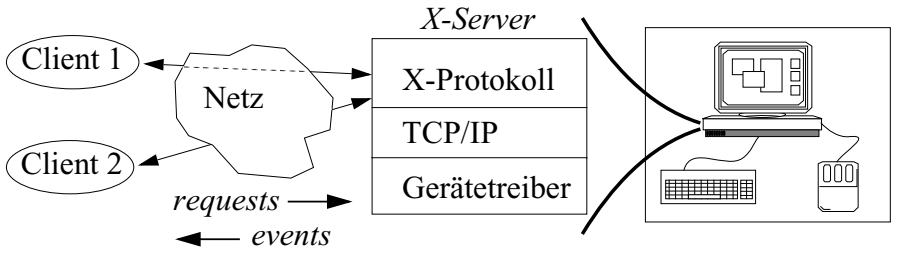
- *Server*: quasiparallele Bearbeitung von RPC-Aufträgen
  - Server bleibt ständig empfangsbereit



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

“X-Window” als Client/Server-Modell

- Erstes netzwerkunabhängiges Graphik- und Fenstersystem für seinerzeit neue pixelorienterte Displays
  - entwickelt Mitte der 80er Jahre am MIT, zusammen mit der Firma DEC



- 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

→ X windows system protocol (über TCP)

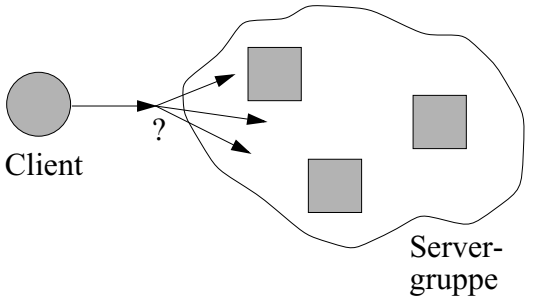
- *Requests*: Service-Anforderung an den X-Server (z.B. Linie in einer bestimmten Farbe zwischen zwei Koordinatenpunkten zeichnen); zugehörige Routinen stehen in einer Bibliothek (*Xlib*)
- *X-Library* (Xlib) ist die Programmierschnittstelle zum X-Protokoll; damit manipuliert ein Client vom Server verwaltete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

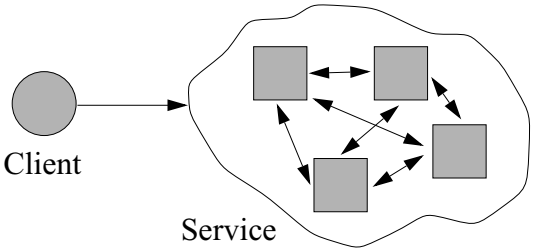


- z.B. Multicast an eine Server*gruppe*
- oder Auswahl eines *bestimmten* Servers (z.B. nahegelegenen oder wenig belasteten)

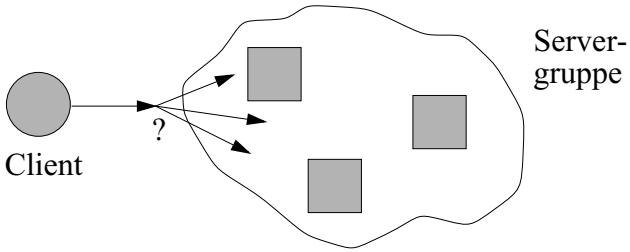
b) *Kooperative Server*

← “Know-how-Verbund”

- ein Server allein kann den Dienst nicht erbringen



Serverwahl bei einem Lastverbund

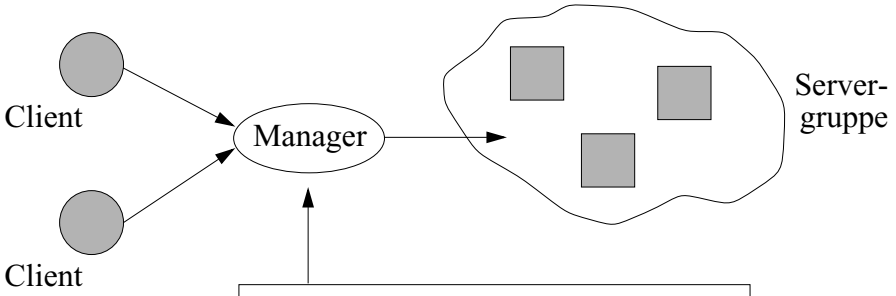


1) *Zufallsauswahl*

- 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*

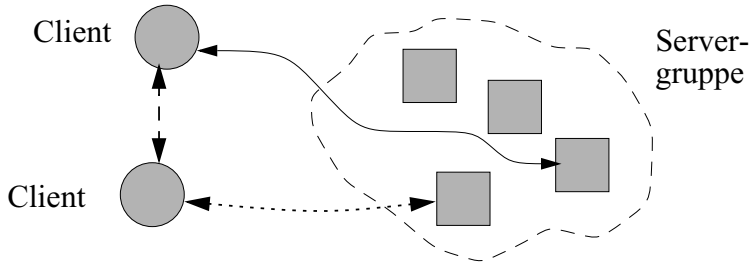


- sorgt für sinnvolle Verteilung (wie?)
- behält ggf. Überblick über Aufträge
- informiert sich ggf. von Zeit zu Zeit über die Server-Lastsituation

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

Serverwahl bei Lastverbund (2)

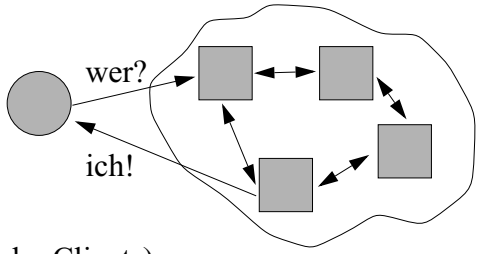
3) *Clients einigen sich untereinander*



- 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 bei wenigen Servern (relativ zur Zahl der Clients)



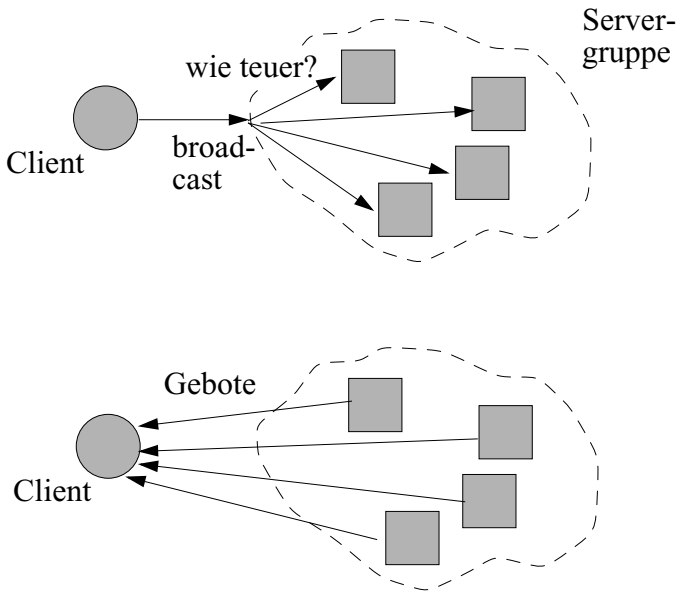
- Server führen Abstimmung diszipliniert durch (verlässlicher als Clients)

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



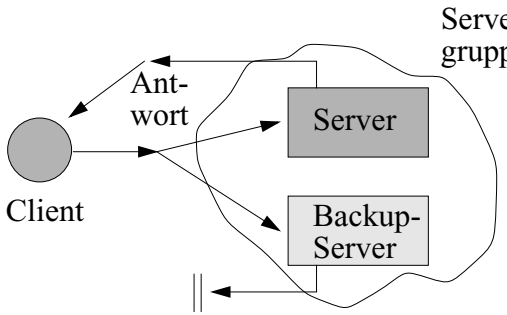
- Variante: nur *Stichprobe* befragen (multicast statt broadcast; sehr kleine Teilmenge von vielen Servern genügt i.a.!)

- 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:

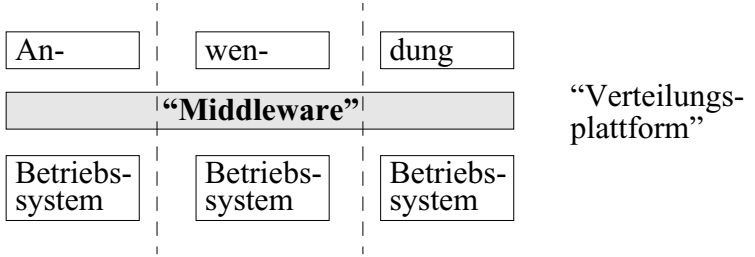
- evtl. Subaufträge werden mehrfach erteilt (vom Server und Backup-Server) → Probleme mit zustandsändernden bzw. gegenseitig ausgeschlossenen Subdiensten
- Reihenfolge der Aufträge muss bei allen Servern identisch sein (→ Semantik von multicast insbesondere bei mehreren Clients)
- Resynchronisation nach 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 für verteiltes Programmieren anbieten, z.B.
  - Verzeichnis- und Suchdienste (Nameservice, lookup service,...)
  - automatische Schnittstellenanpassung (Schnittstellenbeschreibungs-sprache, Stub-Compiler...)

Übersicht: “Historische” Entwicklung

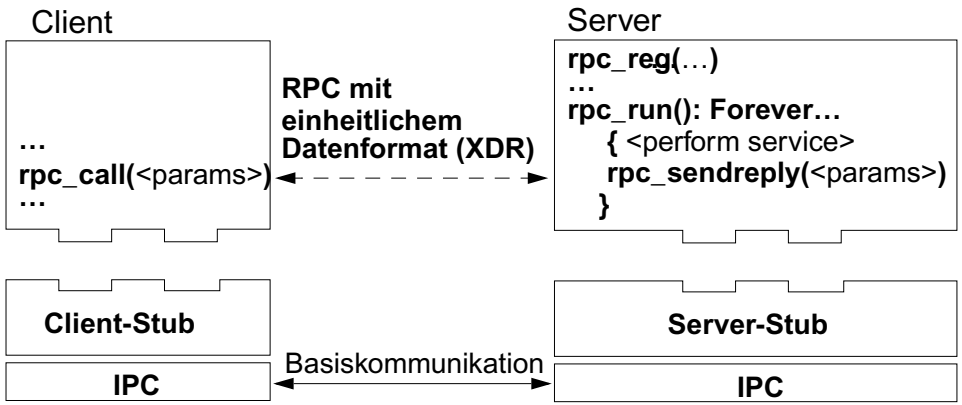
1. RPC-Bibliotheken: z.B. Sun-RPC
   - Client-Server-Paradigma, RPC-Kommunikation
   - Schnittstellen-Beschreibungssprache, Datenformatkonversion, Stubgeneratoren
   - Sicherheitskonzepte (Authentifizierung, Autorisierung, Verschlüsselung)
2. Client-Server-Verteilungsplattformen: z.B. DCE
   - Zeitdienst, 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 weniger gebraucht im technischen Sinne als Verteilungsplattform und Kommunikations- und Diensteinfrastruktur
- sondern 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

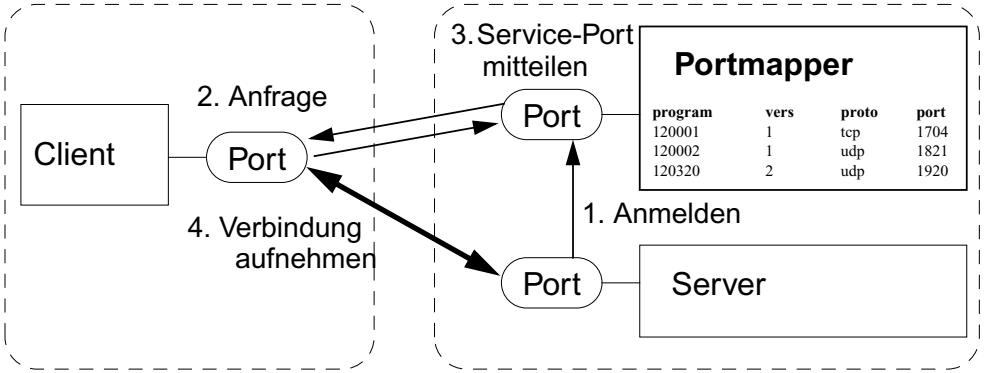
- RPC-Library: Vielzahl aufrufbarer Funktionen (“API”)
  - z.B. rpc_reg, rpc_run, rpc_call (Bezeichnung variiert je nach Variante)
  - daneben auch Funktionen einer Low-Level-Schnittstelle: z.B. Spezifikation von Timeout-Werten oder eines Authentifizierungsprotokolls
- rpcgen: Stub-Generator
- Portmapper: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

---

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

Der Portmapper

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



- Portmapper ist ein Service, der die Zuordnung zwischen Programmnummern und Portnummern verwaltet
- Client kontaktiert vor einem RPC zunächst den Portmapper der Servermaschine, um den Port herauszufinden, wohin die Nachricht gesendet werden soll
  - Portmapper 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
    100004    2   udp    743  ypserv
    100004    1   udp    743  ypserv
    100004    1   tcp    744  ypserv
    100001    2   udp  32830  rstatd
    100029    1   udp    657  keyserv
    100003    2   udp   2049  nfs
      ...
536870928    1   tcp   4441
536870912    1   udp   2140
536870912    1   tcp   4611
      ...
```

Dynamisch generierte Port- und Programmnummern

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

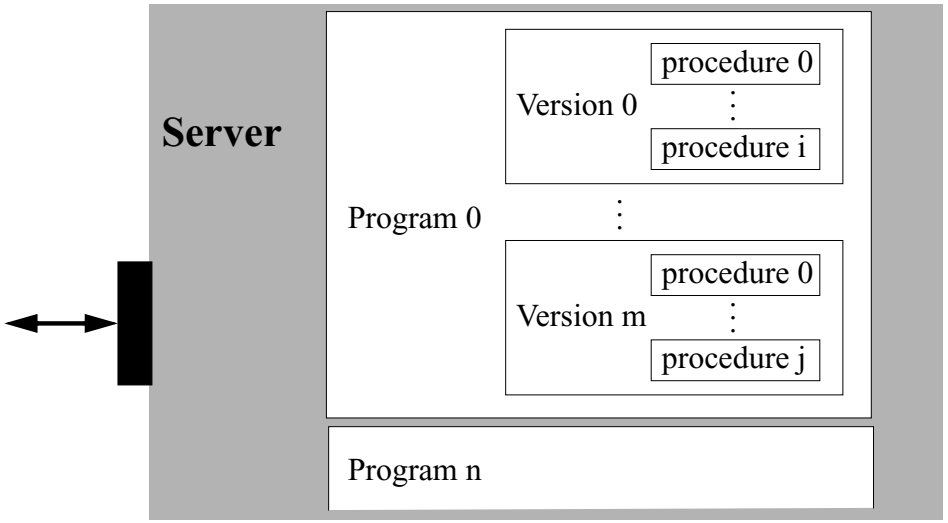
---

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

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

Service-Identifikation

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



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

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

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

Service-Registrierung

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

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

- Welche Programmnummer bekommt ein Service?

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

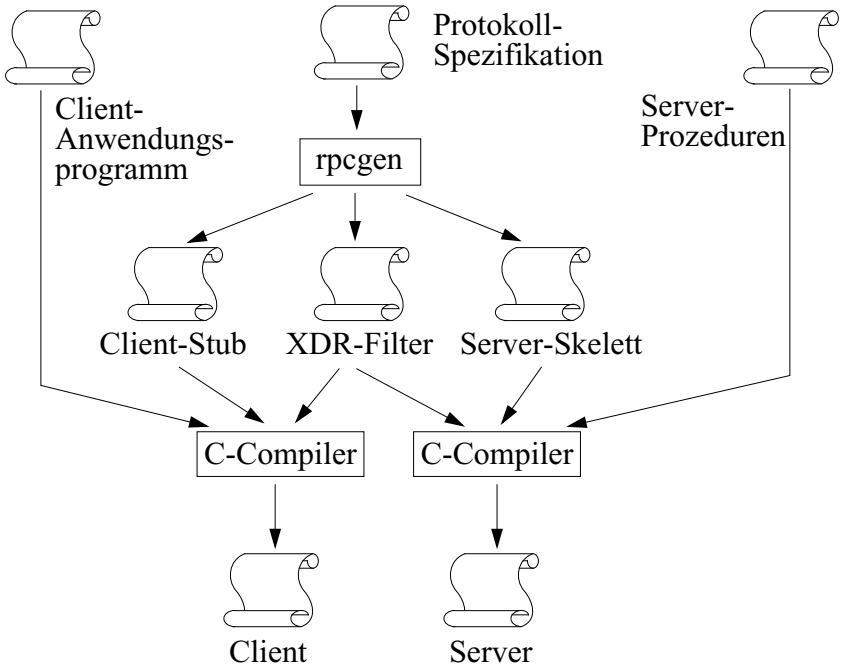
- Es gibt auch eine Broadcast-Variante:

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



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
    - alternativ: Andere auf Public-Key-Systemen beruhende Protokolle

SYS/UNIX-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 die Window-Grösse W als zeitliches Toleranzintervall sowie (ebenfalls verschlüsselt) W-1
  - "zufälliges" Generieren einer ersten Nachricht ist nahezu unmöglich
  - replay (bei kleinem W) ist ebenfalls erfolglos
  - W ist verschlüsselt, um Angreifern keine Informationüber die Fenstergrösse und auch kein Klartext-Schlüsseltext-Paar zu geben

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

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

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