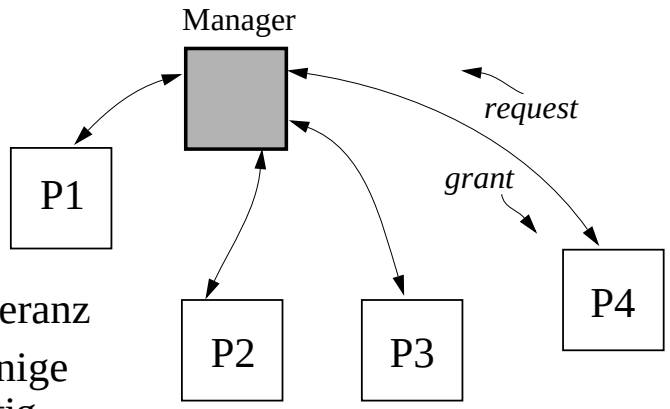
Wechselseitiger Ausschluss

- “Streit” um exklusive Betriebsmittel
  - z.B. konkrete Ressourcen wie gemeinsamer Datenbus
  - oder abstrakte Ressourcen wie z.B. “Termin” in einem (verteilten) Terminkalendersystem
  - “kritischer Abschnitt” in einem (nebenläufigen) Programm

- Lösungen für Einprozessormaschinen, shared memory etc. nutzen typw. Semaphore oder ähnliche Mechanismen
  - ⇒ Betriebssystem- bzw. Concurrency-Theorie
  - ⇒ interessiert uns hier (bei verteilten Systemen) aber nicht

---

- Nachrichtenbasierte Lösung, die auch uninteressant ist, da stark asymmetrisch (“zentralisiert”): Manager-Prozess, der die Ressource (in fairer Weise) zuordnet:

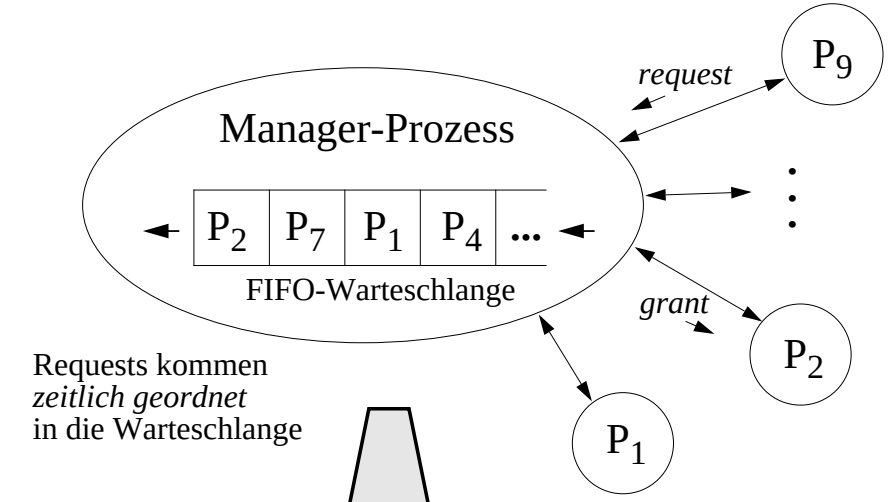


→ Engpass

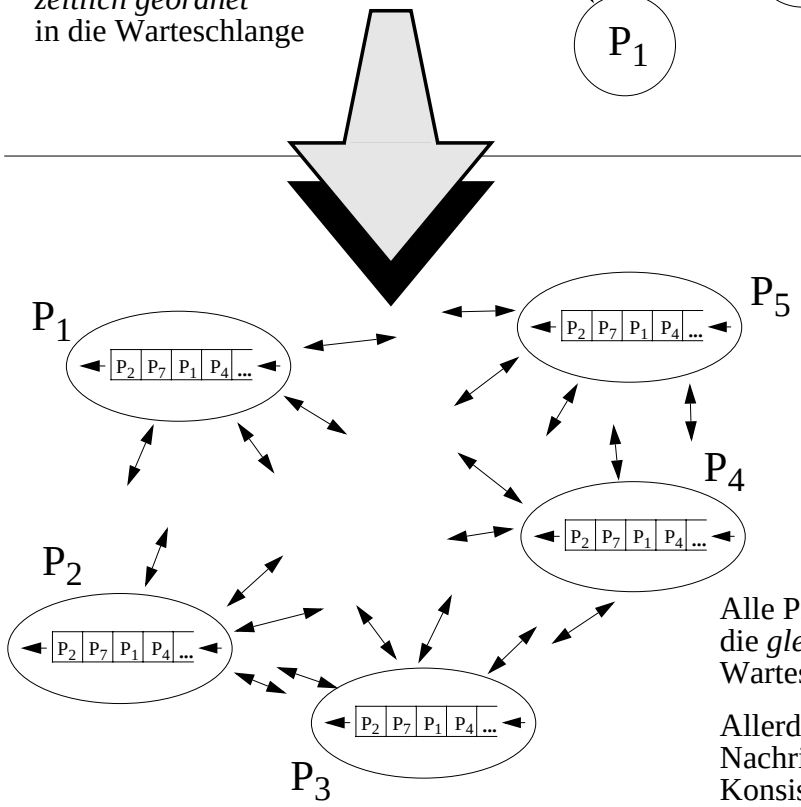
→ keine Fehlertoleranz

→ aber relativ wenige Nachrichten nötig

Replizierte Warteschlange?



Requests kommen *zeitlich geordnet* in die Warteschlange



- Lässt sich mit “logischer Zeit” realisieren, siehe nächste Seite!

Alle Prozesse sollen die *gleiche Sicht* der Warteschlange haben

Allerdings: viele Nachrichten, um Konsistenz zu gewähren; diese müssen ausserdem “geordnet” (=?) ankommen

Anwendung logischer Zeit für den wechselseitigen Ausschluss

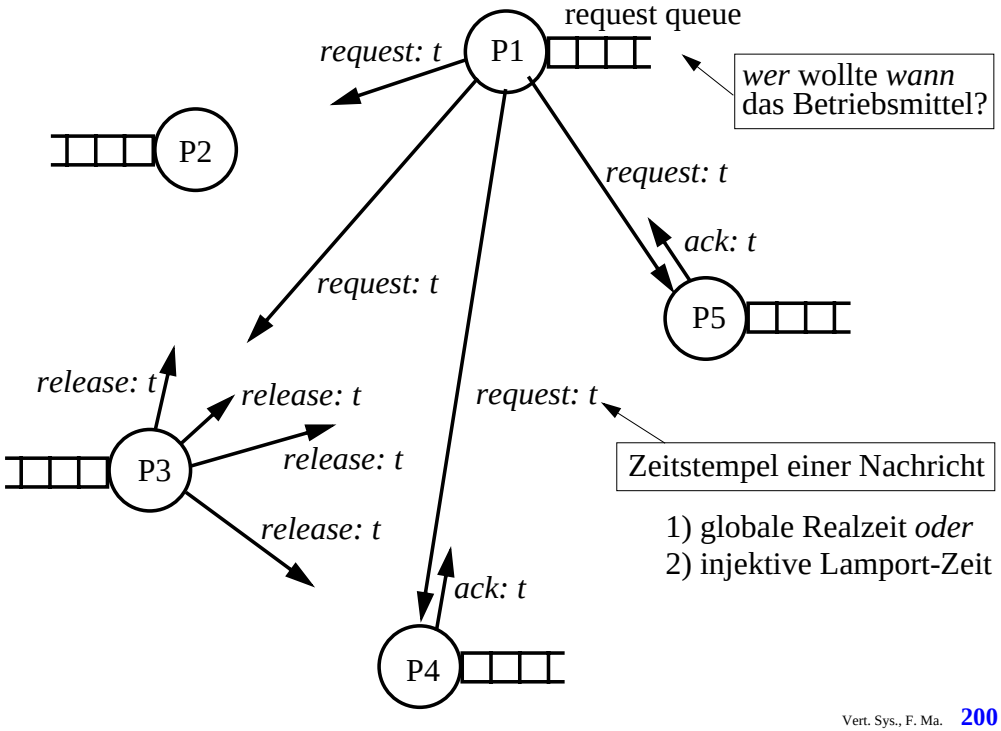
- Hier: Feste Anzahl von Prozessen; ein einziges exklusives Betriebsmittel
- Synchronisierung mit request- / release-Nachrichten

"request" / "release": → vor Betreten / bei Verlassen des *kritischen Abschnittes*

- Fairnessforderung: Jeder request wird "schliesslich" erfüllt

---

Idee: Replikation einer "virtuell globalen" request queue:



1) globale Realzeit *oder*
2) injektive Lamport-Zeit

Der Algorithmus (Lamport 1978):

- Voraussetzung: FIFO-Kommunikationskanäle
- *Alle* Nachrichten tragen (eindeutige!) Zeitstempel

z.B. logische Lamport-Zeit

- Request- und release-Nachrichten an *alle* senden

broadcast

1) Bei "request" des Betriebsmittels: Mit Zeitstempel request in die eigene queue und an alle versenden.

2) Bei Empfang einer request-Nachricht: Request in eigene queue einfügen, ack versenden.

wieso notwendig?

3) Bei "release" des Betriebsmittels: Aus eigener queue entfernen, release-Nachricht an alle versenden.

4) Bei Empfang einer release-Nachricht: Request aus eigener queue entfernen.

5) Ein Prozess darf das *Betriebsmittel benutzen, wenn:*
   - eigener request ist frühester in seiner queue und
   - hat bereits von jedem anderen Prozess (irgendeine) spätere Nachricht bekommen.

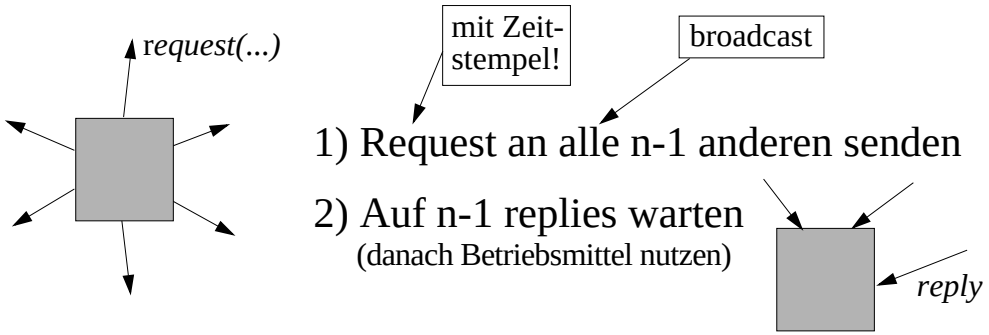
- Frühester request ist global eindeutig.
  ⇒ bei 5): sicher, dass kein früherer request mehr kommt (wieso?)
- 3(n-1) Nachrichten pro "request" (n = Zahl der Prozesse)

Denkübungen:
- wo geht Uhrenbedingung / Kausaltreue der Lamport-Zeit ein?
- sind FIFO-Kanäle wirklich notwendig? (Szenario hierfür?)
- bei Broadcast: welche Semantik? (FIFO, kausal,...?)
- was könnte man bei Nachrichtenverlust tun? (→ Fehlertoleranz)

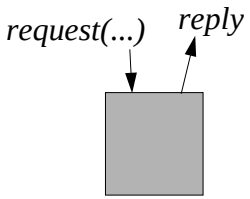
Ein anderer verteilter Algorithmus für den wechselseitigen Ausschluss

(Ricart / Agrawala, 1981)

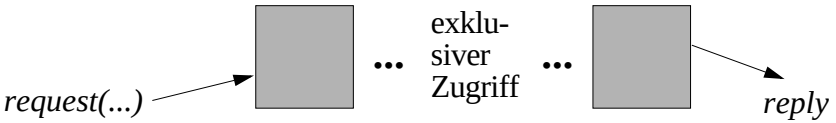
- 2(n-1) Nachrichten statt 3(n-1) wie bei obigem Verfahren
  (*reply-Nachricht* übernimmt Rolle von release und ack)



- Bei Eintreffen einer request-Nachricht:



  - reply sofort schicken, wenn nicht selbst beworben oder der Sender “ältere Rechte” (logische Zeit!) hat:
  - ansonsten reply erst später schicken, nach Erfüllen des eigenen requests (“verzögern”):



- Älteste Bewerbung setzt sich durch (injektive Lamport-Zeit!)

Denkübungen:

- Argumente für die Korrektheit? (Exklusivität, Deadlockfreiheit)
- wie oft muss ein Prozess maximal “nachgeben”? (→ Fairness)
- sind FIFO-Kanäle notwendig?
- geht wechsels. Ausschluss vielleicht mit noch weniger Nachrichten?

Namens-verwaltung

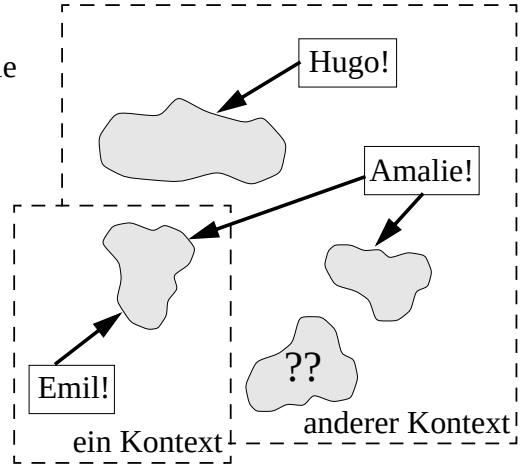
Namen und Namensverwaltung

Namen sind Schall und Rauch

Nomen est omen

- *Namen* sind Symbole, die typischerweise durch Zeichenketten repräsentiert werden

- Dienen der (eindeutigen) *Bezeichnung von Objekten* (inkl. Diensten etc.)
  - daher oft auch “Bezeichner”
  - es gibt auch *anonyme* Objekte (z.B. dynamische Instanzen, 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 *umbenennen* 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, Devices, Benutzer, Dateien, Betriebsmittel...

Zweck von Namen

Typ, Gestalt, Zweck...

- Können Aufschluss über die *Art* eines Objektes geben
  - falls Name (für Benutzer) sinnvoll gewählt
  - z.B. Konventionen xyz.c, xyz.pdf, www.ethz.ch 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
  - z.B. zwecks Manipulation der Objekte
  - über den Namen besteht i. Allg. eine Zugriffsmöglichkeit auf das Objekt
  - 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 für Objekte ermöglichen die *direkte Lokalisierung* und damit den direkten Zugriff

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

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

- *Dagegen*: Name eines Objektes ändert sich i.Allg. 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
  - Evtl. Indirektion durch das Laufzeitsystem
    - z.B. bei Polymorphie objektorientierter Systeme

- *Binden von Dienstaufrufen bei einzelnen 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 evtl. verlagert
    - haben evtl. unterschiedliche Lebenszyklen und -dauer
  - Binden muss daher ebenfalls *dynamisch* (“zur Laufzeit” bzw. beim Objektzugriff) erfolgen!

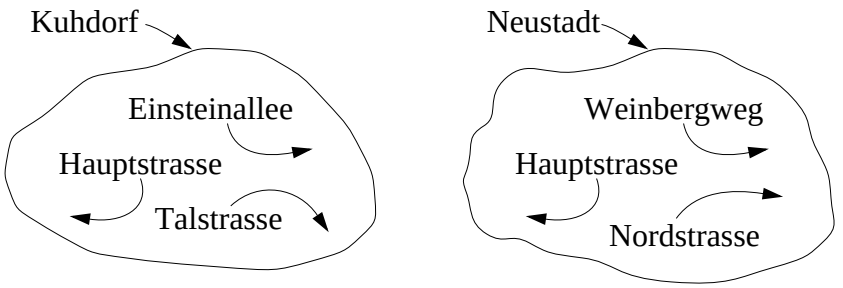
Namenskontext

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

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

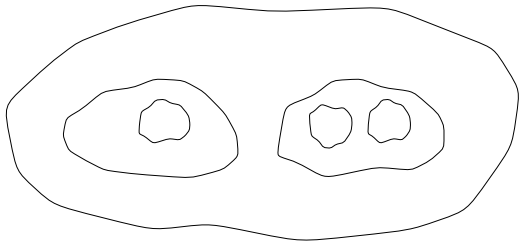
- Falls nur ein einziger Kontext existiert: *flacher Namensraum* (aus “absoluten Namen”)
  - Partition des Namensraum wird oft als “Domäne” bezeichnet

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



Hierarchische Namensräume

- Baumförmige Struktur von Namenskontexten



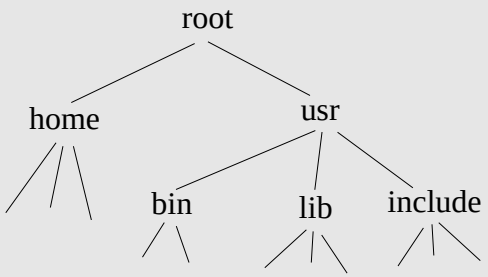
- Beispiel: Adressen im Briefverkehr
  - “Hans Meier, Deutschland” genügt nicht...

Sind das nicht eher Adress-räume als Namensräume?

- Beispiel: Telefonsystem
  - Landeskennung
  - Ortsnetzkennung
  - Teilnehmerkennung

32168 ist ein relativer Name, der z.B. im Kontext 04144 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
  - 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: +44 41 6432168 (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* sind 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 u.U. 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 Duplex-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 eindeutiger Gerätenummer; evtl. auch lange Zufallsbitfolge)

Vgl. "klassische" Dienste beim Telefonsystem:

- Telefonbuch
- Auskunft

} Abbildung Name → Telefonnummer

- evtl. mehrstufig / dezentral: Auslandsauskunft wendet sich an lokale Auskunft im Ausland... (→ hierarchische Namenskontexte notwendig!)
- lokale Telefonbücher 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.

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.

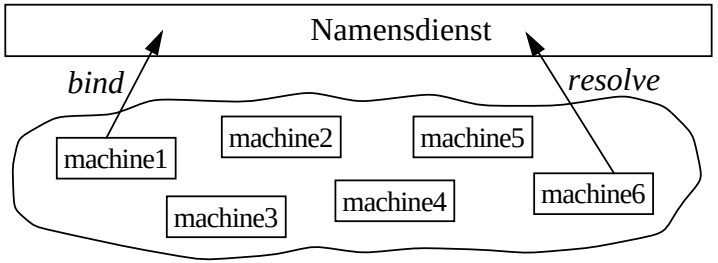
So, given a Krypton-85 nucleus, there is *no way whatsoever to predict when it will decay*. If we have a large number of them, we can be confident half will decay in 10.73 years; but if we have a single atom, pinned in a laser ion trap, all we can say is that is there's even odds it will decay sometime 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 (zentraler) Dienst; tatsächlich verteilt realisiert

- Jeder Kontext wird (logisch) von einem dedizierten *Nameserver* verwaltet
  - evtl. ist ein Nameserver aber für mehrere Kontexte zuständig
  - evtl. Aufteilung des Namensraums / Replikation des Nameservers → höhere Effizienz, Ausfallsicherheit
- Typisch: *kooperierende* einzelne Nameserver, die den gesamten Verwaltungsdienst realisieren
  - hierzu geeignete Architektur der Server vorsehen
  - Protokoll zwischen den Nameservern (für Fehlertoleranz, update der Replikate etc.)
  - Dienstschnittstelle wird typw. durch lokale Nameserver realisiert

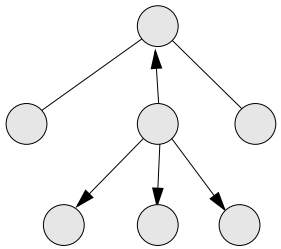
bzw. "user agent"

- Typischerweise *hierarchische Namensräume*
  - entsprechend strukturierte Namen und kanonische Aufteilung der Verwaltungsaufgaben
  - Zusammenfassung Namen gleichen Präfixes vereinfacht Verwaltung
- *Annahmen*, die Realisierungen i.Allg. 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.Allg. 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

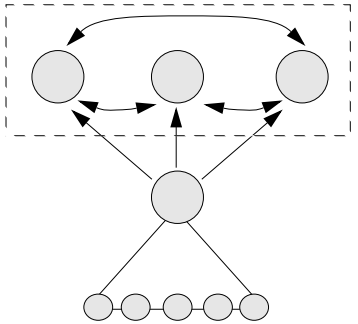
---

Broadcast

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

Repliklation von Nameservern

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

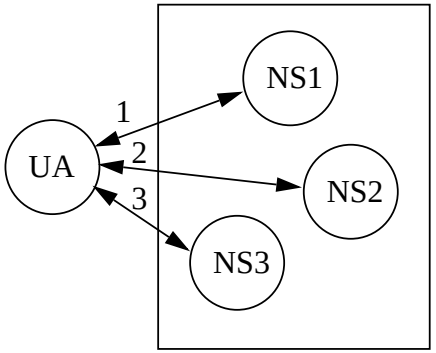


- Nameserver kennt mehrere übergeordnete Nameserver
- Broadcast an ganze Servergruppe, oder Einzelnachricht an "naheliegenden" Server; anderen Server erst nach Ablauf eines Timeouts befragen

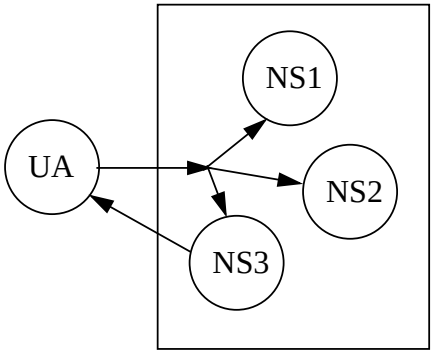
- Replizierte Server konsistent halten
  - evtl. nur von Zeit zu Zeit gegenseitig aktualisieren (falls veraltete Information tolerierbar)
  - Update auch dann sicherstellen, wenn einige Server zeitweise nicht erreichbar sind (periodisches Wiederholen von update-Nachrichten)
  - Einträge mit Zeitstempel versehen → jeweils neuester Eintrag dominiert (global synchronisierte Zeitbasis notwendig!)
- Symmetrische Server / Primärserver-Konzept:
  - *symmetrische Server:* jeder Server einer Gruppe 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

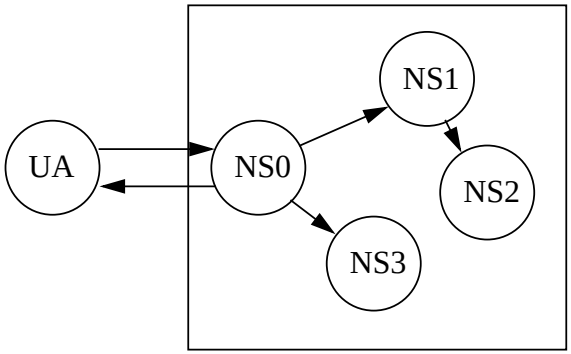
- User Agent (UA) bzw. “Name Agent” auf Client-Seite
  - hinzugebundene Schnittstelle aus Bibliothek, oder
  - eigener lokaler Service-Prozess



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 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* oder:

(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
- Cache-Eintrag u.U. allerdings veraltet (evtl. Lebensdauer beschränken)
- Platz der Tabelle ist beschränkt → unwichtige / alte Einträge verdrängen
- Neue Information wird als Seiteneffekt einer Anfrage im Cache 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

(im allgemeineren Sinne auch als “Verzeichnisdienste” bezeichnet)

- Domain Name System (DNS) im Internet
  - in der UNIX-Welt oft eingesetzte Implementierung: BIND (“Berkeley Internet Domain Name”)

- Portmapper für TCP- oder UDP-basierte Dienste
  - eher rudimentär; nicht verteilt

- Network Information Service (NIS)
  - entwickelt von Sun Microsystems
  - zunächst hauptsächlich zur Verwaltung von Dateizugriffsrechten in lokal vernetzten UNIX-Systemen
  - später erweitert: Verwaltung von Nutzeraccounts, Passwörtern, Diensten
  - basiert auf RPC
  - nutzt Primär- / Sekundärserverprinzip (“Master” / “Replica Server”)

- LDAP (Lightweight Directory Access Protocol)
  - TCP-IP-basierter hierarchischer Verzeichnisdienst für objektbezogene Daten (Beispiel: Adressbuch bei OpenOffice)
  - moderner als NIS (und ersetzt NIS zunehmend)
  - hervorgegangen aus dem OSI-X500-Protokoll (für E-Mail-Anwend.)

- Lookup-Service (“LUS”) bei Jini und ähnlichen Systemen

- UDDI
  -Web-Service-Standard; XML-basiert