RPC-Fehlersemantik

Operationale Sichtweise:

- Wie wird nach einem Timeout auf (vermeintlich?) nicht eintreffende Nachrichten, wiederholte Requests, gecrashte Prozesse reagiert?

---

1) Maybe-Semantik:

- Keine Wiederholung von Requests
- *Einfach und effizient*
- Keinerlei Erfolgsgarantien → nur ausnahmsweise anwendbar
  Mögliche Anwendungsklasse: Auskunftsdienste (Anwendung kann es evtl. später noch einmal probieren, wenn keine Antwort kommt)

2) At-least-once-Semantik:

1) und 2) werden etwas euphemistisch als "best effort" bezeichnet

- Hartnäckige automatische Wiederholung von Requests
- Keine Duplikatserkennung (*zustandsloses Protokoll* auf Serverseite)
- Akzeptabel bei idempotenten Operationen (z.B. Lesen einer Datei)

3) At-most-once-Semantik:

- Erkennen von Duplikaten (Sequenznummern, log-Datei etc.)
- Keine wiederholte Ausführung der Prozedur, sondern evtl. erneutes Senden des (gemerkten) Reply
- Geeignet auch für *nicht-idempotente* Operationen

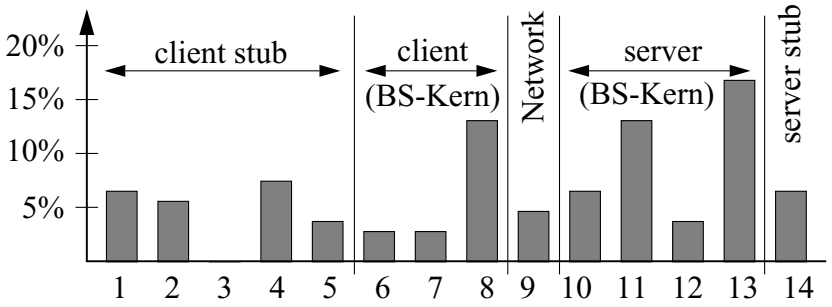
ist "exactly once" machbar?

- May-be → At-least-once → At-most-once → ...
  ist zunehmend aufwändiger zu realisieren
- man begnügt sich daher, falls es der Anwendungsfall gestattet, oft mit einer billigeren aber weniger perfekten Fehlersemantik
- Motto: so billig wie möglich, so „perfekt" wie nötig

RPC: Effizienz

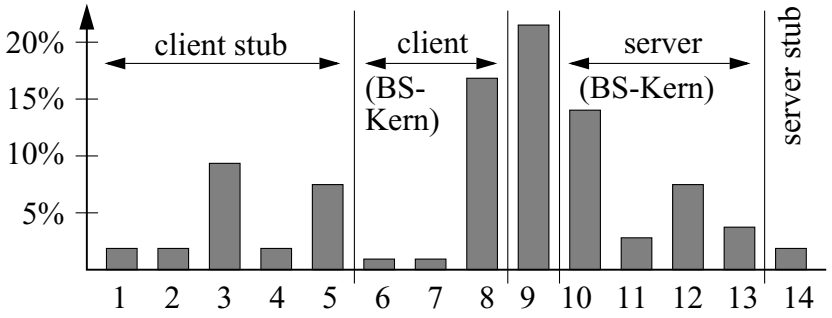
Analyse eines RPC-Protokolls (zitiert nach A. Tanenbaum)

a) Null-RPC (Nutznachricht der Länge 0, keine Auftragsbearbeitung):



1. Call stub
2. Get message buffer
3. Marshal parameters
4. Fill in headers
5. Compute UDP checksum
6. Trap to kernel
7. Queue packet for transmission
8. Move packet to controller over the bus
9. Network transmission time
10. Get packet from controller
11. Interrupt service routine
12. Compute UDP checksum
13. Context switch to user space
14. Server stub code

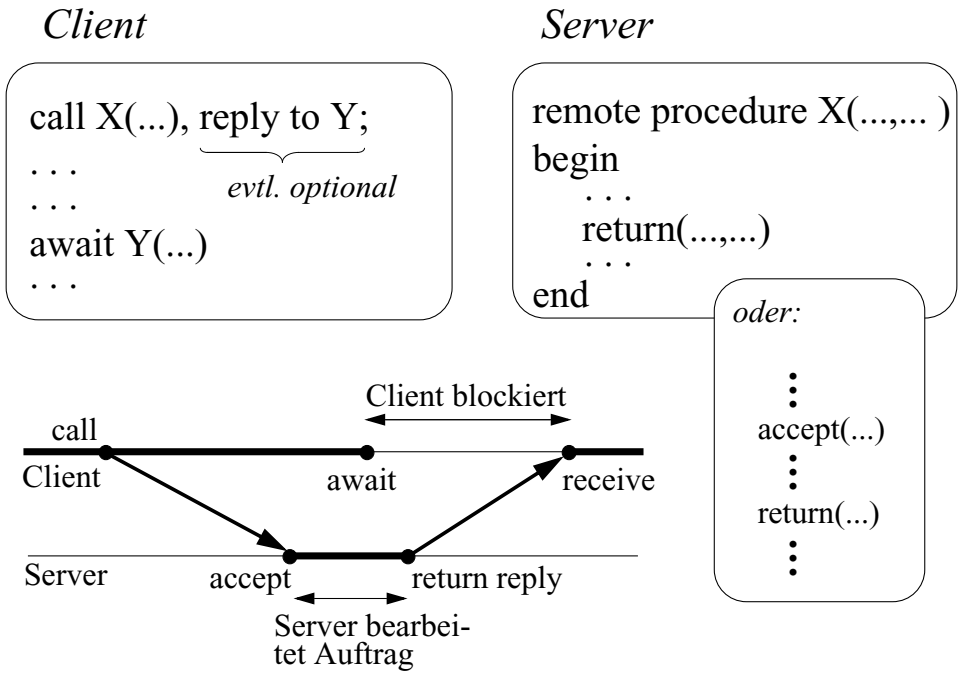
b) 1440 Byte Nutznachricht (ebenfalls keine Auftragsbearbeitung):



- Eigentliche Übertragung kostet relativ wenig
- Rechenoverhead (Prüfsummen, Header etc.) keineswegs vernachlässigbar
- Bei kurzen Nachrichten: Kontextwechsel zw. Anwendung und BS wichtig
- Mehrfaches Kopieren kostet viel

Asynchroner RPC

- Andere Bezeichnung: “Remote Service Invocation”
- Auftragsorientiert → Antwortverpflichtung



- Parallelverarbeitung von Client und Server möglich, solange Client noch nicht auf Resultat angewiesen

Future-Variablen

- Zuordnung Auftrag / Ergebnisempfang bei der asynchron-auftragsorientierten Kommunikation?
  - unterschiedliche Ausprägung auf Sprachebene möglich
  - “await” könnte z.B. einen bei “call” zurückgelieferten “handle” als Parameter erhalten, also z.B.: Y = call X(...); ... await (Y);
  - evtl. könnte die Antwort auch asynchron in einem eigens dafür vorgesehenen Anweisungsblock empfangen werden (vgl. Interrupt- oder Exception-Routine)

- Spracheinbettung evtl. auch durch “Future-Variablen”
  - Future-Variable = “handle”, der wie ein Funktionsergebnis in Ausdrücke eingesetzt werden kann
  - Auswertung der Future-Variable erst dann, wenn unbedingt nötig
  - Blockade nur dann, falls Wert bei Nutzung noch nicht feststeht
  - Beispiel:

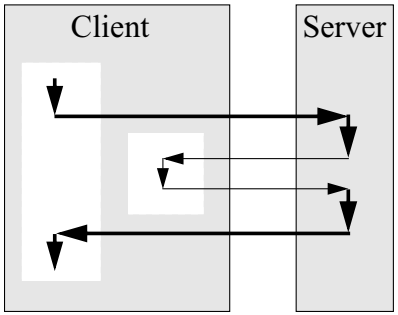
```
FUTURE f: integer;
…
f = call(…);
…
some_value = 4711;
print(some_value + f);
```

Beispiel: RPC bei DCE

- DCE (“Distributed Computing Environment”) ist eine Middleware, die in den 1990er-Jahren von einem herstellerübergreifenden Konsortium entwickelt wurde

- RPCs weisen dort einige interessante Besonderheiten auf:

- *Rückrufe* (“call back RPC”)
  - temporärer Rollentausch von Client und Server
  - um evtl. bei langen Aktionen Zwischenresultate zurückzumelden
  - um evtl. weitere Daten vom Client anzufordern
  - Client muss Rückrufadresse übergeben



- *Pipes* als spezielle Parametertypen
  - sind selbst keine Daten, sondern ermöglichen es, Daten stückweise zu empfangen (“pull”-Operation) oder zu senden (“push”)
  - evtl. sinnvoll bei der Übergabe grosser Datenmengen
  - evtl. sinnvoll, wenn Datenmenge erst dynamisch bekannt wird (“stream”)

- *Context-handles* zur aufrufglobalen Zustandsverwaltung
  - werden vom Server dynamisch erzeugt und an Client zurückgegeben
  - Client kann diese beim nächsten Aufruf unverändert wieder mitsenden
  - Kontextinformation zur Verwaltung von Zustandsinformation über mehrere Aufrufe hinweg z.B. bei Dateiserver (“read; read”) sinnvoll
  - vgl. “cookies”
  - Vorteil: Server arbeitet “zustandslos“

Beispiel: RPC bei DCE (2)

- Semantik für den *Fehlerfall* ist bei DCE-RPCs wählbar:

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

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

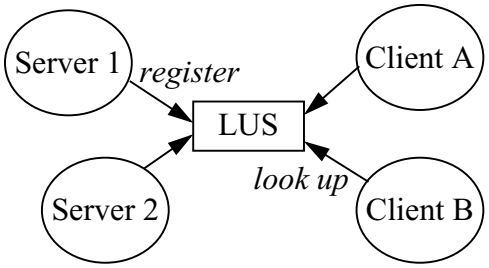
  (c) *maybe*
  - wie (b), aber ohne Rückmeldung über Erfolg oder Fehlschlag
  - noch effizienter, aber nur in speziellen Fällen anwendbar

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

- Wählbare Sicherheitsstufen bei der Kommunikation
  - Authentifizierung nur bei Aufbau der Verbindung (“binding”)
  - Authentifizierung pro RPC-Aufruf
  - Authentifizierung pro Nachrichtenpaket
  - Zusätzlich Verschlüsselung jedes Nachrichtenpaketes
  - Schutz gegen Verfälschung (verschlüsselte Prüfsumme)

Lookup-Service

- Problem: Wie finden sich Client und Server ?
  - haben i.Allg. verschiedene Lebenszyklen → kein gemeinsames Übersetzen / statisches Binden (fehlende gem. Umgebung)
  - Lookup Service (LUS) oder "registry"



- Server (-stub) gibt den Namen etc. seines Services (RPC-Routine) dem LUS bekannt
  - "*register*"; "exportieren" der RPC-Schnittstelle (Typen der Parameter,...)
  - evtl. auch wieder abmelden
- Client erfragt beim LUS die Adresse eines geeigneten Servers (aber wie?)
  - "*look up*"; "discovery"; "importieren" der RPC-Schnittstelle
- *Vorteile*: im Prinzip kann LUS
  - mehrere Server für den gleichen Service registrieren (→ Fehlertoleranz; Lastausgleich)
  - Autorisierung etc. überprüfen
  - durch Polling der Server die Existenz eines Services testen
  - verschiedene Versionen eines Dienstes verwalten
- *Probleme*:
  - lookup kostet Ausführungszeit (gegenüber statischem Binden)
  - zentraler LUS ist ein potentieller Engpass (LUS-Service geeignet replizieren / verteilen?)

Adressierung

- *Sender* muss in geeigneter Weise spezifizieren, wohin die Nachricht gesendet werden soll
  - evtl. mehrere Adressaten zur freien Auswahl (Lastverteilung, Fehlertoleranz)
  - evtl. mehrere Adressaten gleichzeitig (Broadcast, Multicast)
- *Empfänger* ist evtl. nicht bereit, jede beliebige Nachricht von jedem Sender zu akzeptieren
  - selektiver Empfang
  - Sicherheitsaspekte, Überlastabwehr
- Probleme
  - *Ortstransparenz*: Sender weiss *wer*, aber nicht *wo* (sollte er i.Allg. auch nicht!)
  - *Anonymität*: Sender und Empfänger kennen einander zunächst nicht (sollen sie oft auch nicht)

Kenntnis von Adressen?

- Adressen sind u.a. Geräteadressen (z.B. IP-Adresse oder Netzadresse in einem LAN), Portnamen, Socketnummern, Referenzen auf Mailboxes,...

- Woher kennt ein Sender die Adresse des Empfängers?

  1) Fest in den Programmcode integriert → unflexibel

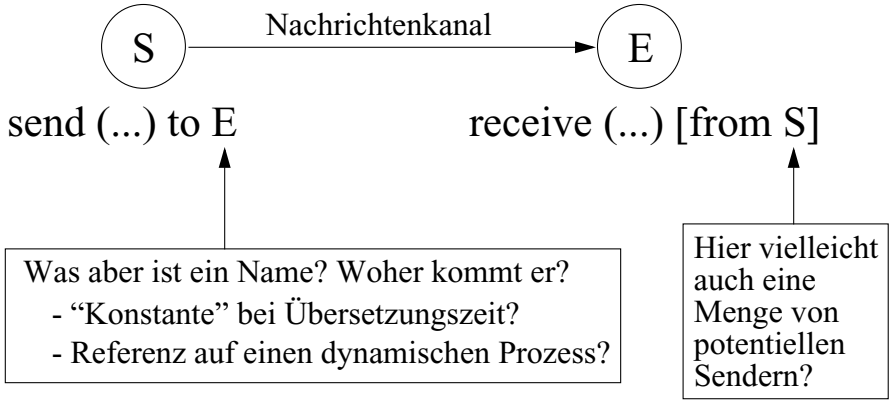
  2) Über Parameter erhalten oder von anderen Prozessen mitgeteilt

  3) Adressanfrage per Broadcast “in das Netz”
     - häufig bei LANs: Suche nach lokalem Nameserver, Router etc.

  4) Auskunft fragen (Namensdienst wie z.B. DNS; Lookup-Service)

Direkte Adressierung

- *Direct Naming* (1:1-Kommunikation):

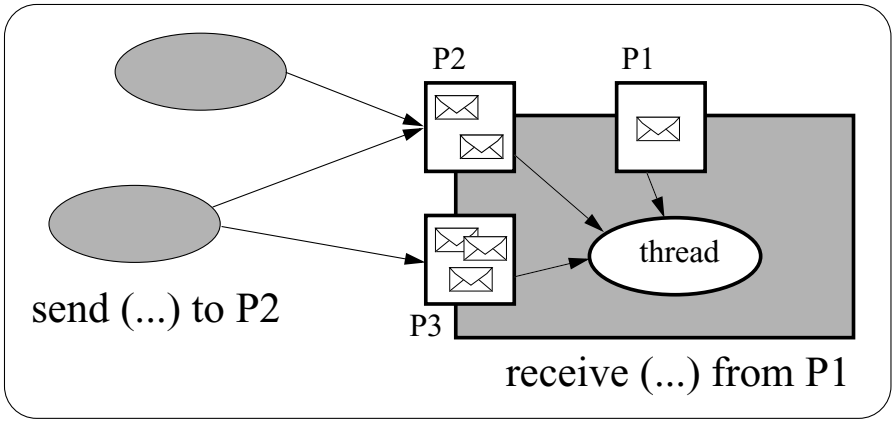


- Direct naming ist insgesamt relativ unflexibel

- Empfänger (= Server) sollten nicht gezwungen sein, potentielle Sender (= Client) explizit zu nennen
  - Symmetrie ist also i.Allg. gar nicht erwünscht

Indirekte Adressierung - Ports

- m:1-Kommunikation
- Ports "gehören" einem Empfänger
  - Kommunikationsendpunkt, der die interne Empfängerstruktur abkapselt
- Ein Objekt kann i.Allg. mehrere Ports besitzen

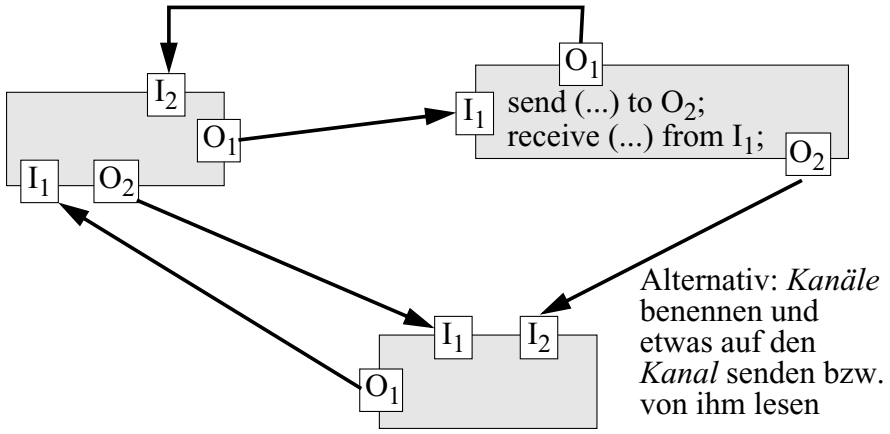


Pragmatische Aspekte (Sprachdesign etc.):

- Sind Ports statische oder dynamische Objekte?
- Wie erfährt ein Objekt den Portnamen eines anderen (dynamischen) Objektes?
  - können Namen von Ports verschickt werden?
- Sind Ports typisiert?
  - würde den selektiven Nachrichtenempfang unterstützen
- Können Ports geöffnet und geschlossen werden?
  - genaue Semantik?

Kanäle und Verbindungen

- Neben *Eingangsports* ("In-Port") lassen sich auch *Ausgangsports* ("Out-Port") betrachten



- Ports können als Ausgangspunkte für das Einrichten von *Verbindungen* ("Kanäle") gewählt werden
- Dazu werden je ein In- und Out-Port miteinander verbunden. Dies kann z.B. mit einer connect-Anweisung geschehen: **connect** p1 **to** p2
- Die Programmierung und Instanziierung eines Objektes findet so in einer anderen Phase statt als die Festlegung der Verbindungen

Konfigura-tionsphase

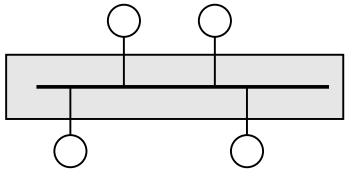
- Grössere Flexibilität durch die dynamische Änderung der Verbindungsstruktur
- Evtl. Broadcast-Kanäle: Nachricht geht an alle angeschlossenen Empfänger (z.B. "Event-Bus")

Ereigniskanäle für Software-Komponenten

- Stark entkoppelte Kommunikation
  - Software-Komponenten haben oft getrennte Lebenszyklen
  - Entkoppelung fördert bessere Wiederverwendbarkeit und Wartbarkeit
  - anonym: Sender / Empfänger erfahren nichts über die Identität des anderen
  - Auslösen von Ereignissen bei Sendern
  - Reagieren auf Ereignisse bei Empfängern
  - dazwischenliegende “third party objects” können Ereignisse speichern, filtern, umlenken,...

oder sogar die Existenz

- Ereigniskanal als “Softwarebus”
  - agiert als Zwischeninstanz und verknüpft die Komponenten
  - registriert Interessenten
  - Dispatching eingehender Ereignisse
  - evtl. Pufferung von Ereignisen

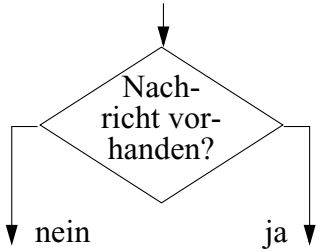


- Probleme
  - Ereignisse können “jederzeit” ausgelöst werden, von Empfängern aber i.Allg. nicht jederzeit entgegengenommen werden
  - falls Komponenten nicht lokal, sondern verteilt auf mehreren Rechnern liegen, die “üblichen” Probleme: verzögerte Meldung, u.U. verlorene Ereignisse, Multicastsemantik,...

- Beispiele
  - Microsoft-Komponentenarchitektur (DCOM / ActiveX / OLE / .NET / ...)
  - “Distributed Events” bei JavaBeans und Jini (event generator bzw. remote event listener)
  - event service von CORBA: sprach- und plattformunabhängig; typisierte und untypisierte Kanäle; Schnittstellen zur Administration von Kanälen; Semantik (z.B. Pufferung des Kanals) jedoch nicht genauer spezifiziert

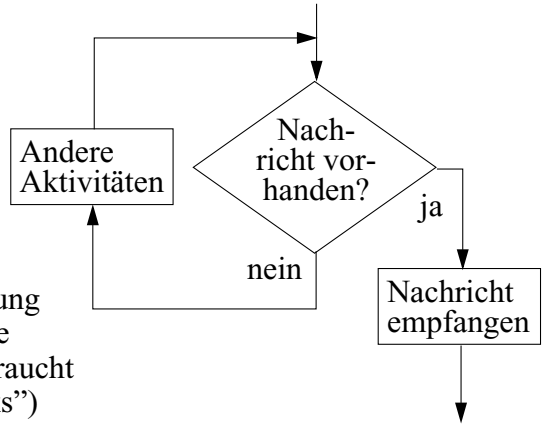
Nichtblockierendes Empfangen

- Typischerweise ist ein “receive” blockierend
- Aber auch *nichtblockierender* Empfang ist denkbar:



- “Non-blocking receive”
- Sprachliche Realisierung z.B. durch “Returncode” eines als Funktionsaufruf benutzten “receive” API

- Aktives Warten:
  (“busy waiting”)
  - Nachbildung des blockierenden Wartens wenn “andere Aktivitäten” leer
  - Nur für kurze Wartezeiten sinnvoll, da Monopolisierung der cpu, die ggf. für andere Prozesse oder threads gebraucht werden könnte (“spin locks”)



- Weitere Möglichkeit: unterbrechungsgesteuertes (“asynchrones”) Empfangen der Nachricht (→ nicht unproblematisch!)

Zeitüberwachte Kommunikation

- Empfangsanweisung soll maximal (?) eine gewisse Zeit lang blockieren ("timeout")
  - z.B. über return-Wert abfragen, ob Kommunikation geklappt hat

- Sinnvoll bei:
  - Echtzeitprogrammierung
  - Vermeidung von Blockaden im Fehlerfall (etwa: abgestürzter Kommunikationspartner)

    → dann sinnvolle Recovery-Massnahmen treffen ("exception")

    → timeout-Wert "sinnvoll" setzen!

    Quelle vielfältiger Probleme...

- Timeout-Wert = 0 kann u.U. genutzt werden, um zu testen, ob eine Nachricht "jetzt" da ist

---

- Analog evtl. auch für synchrones (!) *Senden* sinnvoll

  → Verkompliziert zugrundeliegendes Protokoll: Implizite Acknowledgements kommen nun "asynchron" an...

Zeitüberwachter Nachrichtenempfang

- Möglicher Realisierung:
  - Durch einen Timer einen *asynchronen Interrupt* aufsetzen und Sprungziel benennen
  - Sprungziel könnte z.B. eine Exception-Routine sein, die in einem eigenen Kontext ausgeführt wird, oder das Statement nach dem receive

  → "systemnahe", unstrukturierte, fehleranfällige Lösung; schlechter Programmierstil!

- Sprachliche Einbindung besser z.B. so:

```
receive ... delay t
    {...}
else
    {...}
end
```

Vorsicht!

Blockiert maximal t Zeiteinheiten

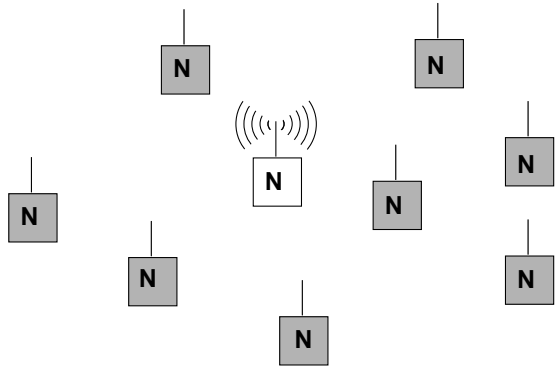
Wird nach *mind*. t Zeiteinheiten ausgeführt, wenn bis dahin noch keine Nachricht empfangen

- Genaue Semantik beachten: Es wird *mindestens* so lange auf Kommunikation gewartet. Danach kann (wie immer!) noch beliebig viel Zeit bis zur Fortsetzung des Programms verstreichen!

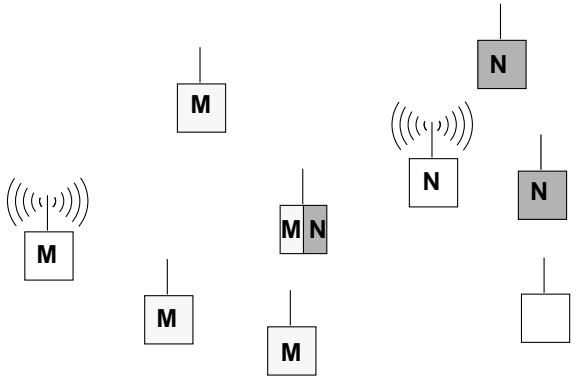
- Frage: Was sollte "delay 0" bedeuten?

Gruppen-kommunikation

Gruppenkommunikation



Broadcast: Senden an die *Gesamtheit* aller Teilnehmer

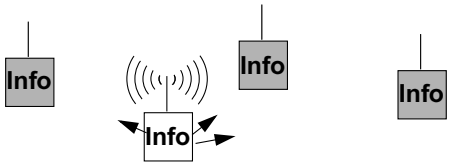


Multicast: Senden an eine *Untergruppe* aller Teilnehmer

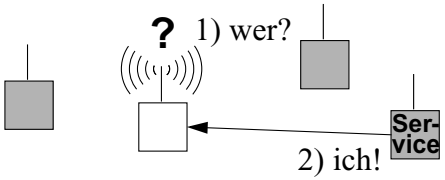
- entspricht Broadcast bezogen auf die Gruppe
- verschiedene Gruppen können sich ggf. überlappen
- jede Gruppe hat eine Multicastadresse

Anwendungen von Gruppenkommunikation

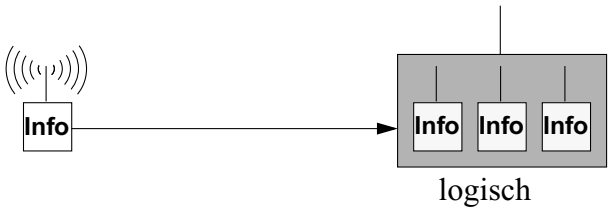
- *Informieren*
  - z.B. Newsdienste

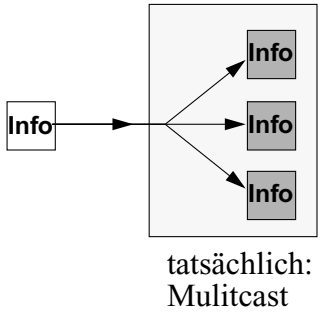


- *Suchen*
  - z.B. Finden von Objekten und Diensten



- *"Logischer Unicast"* an replizierte Komponenten

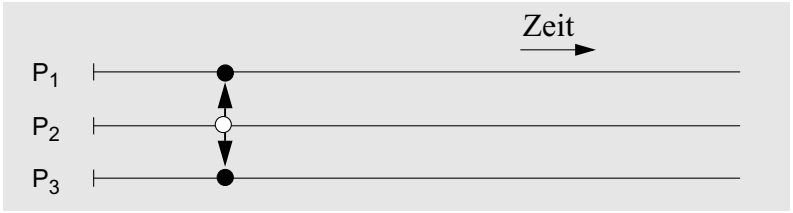




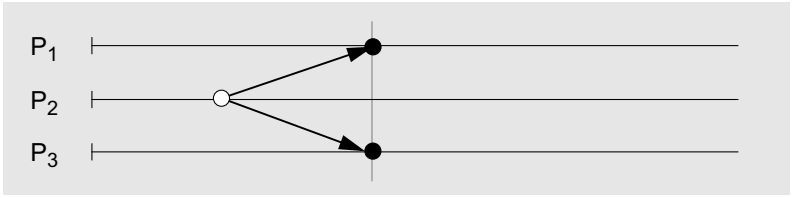
Typische Anwendungsklasse von Replikation: Fehlertoleranz

Gruppenkommunikation - idealisierte Semantik

- Modellhaftes Vorbild: Speicherbasierte Kommunikation in zentralistischen Systemen
  - augenblicklicher "Empfang"
  - vollständige Zuverlässigkeit (kein Nachrichtenverlust etc.)



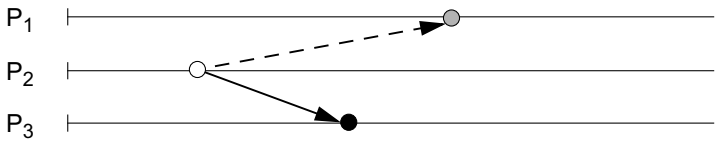
- Nachrichtenbasierte Kommunikation: Idealisierte Sicht
  - (verzögerter) *gleichzeitiger* Empfang
  - vollständige Zuverlässigkeit



Gruppenkommunikation - tatsächliche Situation

- Medium (Netz) ist oft nicht multicastfähig
  - LANs höchstens innerhalb von Teilstrukturen; WLAN als Funknetz a priori anfällig für Übertragungsstörungen
  - multicastfähige Netze sind typischerweise nicht verlässlich (keine Empfangsgarantie)
  - bei Punkt-zu-Punkt-Netzen: "Simulation" von Multicast durch ein Protokoll (z.B. Multicast-Server, der an alle einzeln weiterverteilt)

- Nachrichtenkommunikation ist nicht "ideal"
  - nicht-deterministische Zeitverzögerung → Empfang zu *unterschiedlichen* Zeiten
  - nur bedingte Zuverlässigkeit der Übermittlung



- Ziel von Broadcast / Multicast-Protokollen:
  - möglichst gute Approximation der Idealsituation
  - möglichst hohe Verlässlichkeit und Effizienz

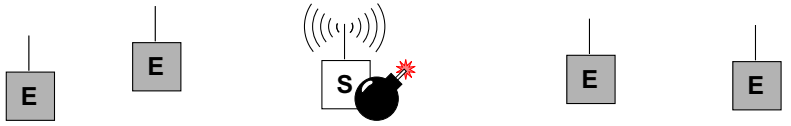
- Beachte: Verlust von Nachrichten und sonstige Fehler sind bei Broadcast ein ernsteres Problem als beim "Unicast"! (Wieso?)
- Hauptproblem bei der Realsierung von Broadcast: (1) Zuverlässigkeit und (2) garantierte Empfangsreihenfolge

Senderausfall beim Broadcast

Wenn Broadcast implementiert ist durch Senden vieler Einzelnachrichten (dann ist das Senden ein "nicht atomarer", länger andauernder Vorgang)

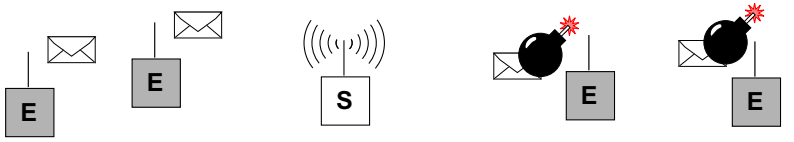
a) Sender fällt ganz aus: *kein* Empfänger erhält Nachricht

- „günstiger" Fall: *Einigkeit* unter den Überlebenden!



b) Sender fällt *während* des Sendevorgangs aus: nur *einige* Empfänger erhalten u.U. die Nachricht

- "ungünstiger" Fall: *Uneinigkeit*



- mögliche Abhilfe: Empfänger leiten zusätzlich die Nachricht auch untereinander weiter
- Uneinigkeit der Empfänger kann die Ursache für sehr ärgerliche Folgeprobleme sein! (Da wäre es manchmal besser, *gar kein* Prozess hätte die Nachricht empfangen!)

“Best effort”-Broadcasts

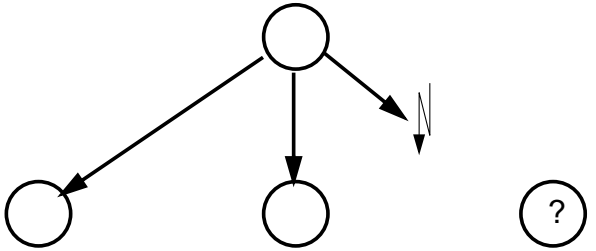
- Fehlermodell: Verlust von Nachrichten (und evtl. temporärer Crash von Prozessen)
  - Nachrichten können aus unterschiedlichen Gründen verloren gehen (z.B. Netzüberlastung, Empfänger hört gerade nicht zu,...)
- Euphemistische Bezeichnung, da keine extra Anstrengung
  - typischerweise einfache Realisierung ohne Acknowledgements etc.
- Keinerlei Garantien
  - unbestimmt, wieviele / welche Empfänger eine Broadcastnachricht tatsächlich empfangen
  - unbestimmte Empfangsreihenfolge

Kann z.B. beim Software-update über Satellit zu einem ziemlichen Chaos führen

- Allerdings effizient (im Erfolgsfall)
- Geeignet für die Verbreitung unkritischer Informationen
  - z.B. Informationen, die evtl. Einfluss auf die Effizienz haben, nicht aber die Korrektheit betreffen
- Evtl. als Grundlage zur Realisierung höherer Protokolle
  - diese basieren oft auf der A-priori-Broadcastfähigkeit von Netzen
  - günstig bei zuverlässigen physischen Kommunikationsmedien (wenn Fehlerfall sehr selten → aufwändiges Recovery auf höherer Ebene tolerierbar)

“Reliable Broadcast”

- Ziel: Unter gewissen (welchen?) Fehlermodellen einen “möglichst zuverlässigen” Broadcast-Dienst realisieren



- Quittung (“positives Acknowledgement”: ACK) für jede Einzelnachricht
  - im Verlustfall einzeln nachliefern oder (falls broadcastfähiges Medium vorhanden) einen zweiten Broadcast-Versuch? (→ Duplikaterkennung!)
  - viele ACKs → teuer / skaliert schlecht
- Skizze einer anderen Idee (“negatives Ack.”: NACK):
  - alle broadcasts werden vom Sender aufsteigend numeriert
  - Empfänger stellt beim *nächsten* Empfang evtl. eine Lücke fest
  - für fehlende Nachrichten wird ein “negatives ack” (NACK) gesendet
  - Sender muss daher Kopien von Nachrichten (wie lange?) aufbewahren und fehlende nachliefern
  - “Nullnachrichten” sind u.U. sinnvoll (→ schnelles Erkennen von Lücken)
  - Kombination von ACK / NACK mag sinnvoll sein
- Dies hilft aber nicht, wenn der Sender mittendrin crasht!

Reliable-Broadcast-Algorithmus

- Zweck: Jeder nicht gecrashte und zumindest indirekt erreichbare Prozess soll die Broadcast-Nachricht erhalten
  - Voraussetzung: zusammenhängendes "gut vermaschtes" Punkt-zu-Punkt-Netz
  - Fehlermodell: Prozesse und Verbindungen mit Fail-Stop-Charakteristik ("crash" ohne negative Seiteneffekte und ohne Neustart)

Sender *s*: Realisierung von **broadcast(*N*)**

- **send(*N*, *s*, *sequ_num*)** an alle Nachbarn (inklusive an *s* selber);
- sequ_num ++

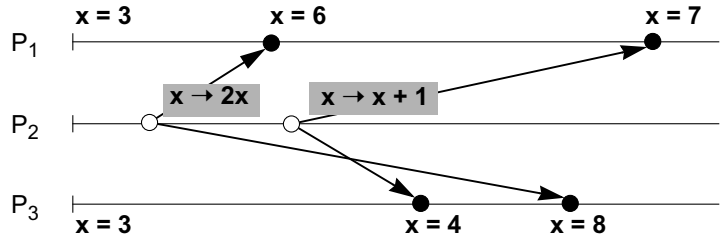
Empfänger *r*: Realisierung des Nachrichtenempfangs

- **receive(*N*, *s*, *sequ_num*);**
  wenn *r* noch kein **deliver(*N*)** für *sequ_num* ausgeführt hat, dann:
  wenn $r \neq s$ dann **send(*N*, *s*, *sequ_num*)** an alle Nachbarn von *r* ;
  Nachricht an die Anwendungsebene ausliefern ("deliver(N)") ;

- Prinzip: "Fluten" des Netzes
  - vgl. dazu "Echo-Algorithmus" und Vorlesung "Verteilte Algorithmen"
- Beachte: receive ≠ deliver
  - unterscheide Anwendungsebene und Transportebene
- Denkübungen:
  - müssen die Kommunikationskanäle bidirektional sein?
  - wie effizient ist das Verfahren (Anzahl der Einzelnachrichten)?
  - wie fehlertolerant? (wieviel darf kaputt sein / verloren gehen...?)
  - kann man das gleiche auch anders / besser erreichen?

Broadcast: Empfangsreihenfolge

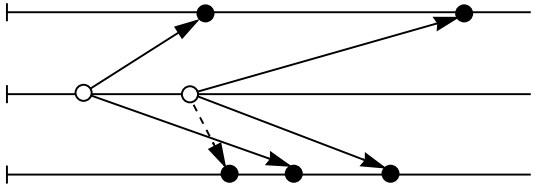
- Wie ist die Empfangsreihenfolge von Nachrichten?
  - problematisch wegen der i.Allg. ungleichen Übermittlungszeiten
  - Bsp.: Update einer replizierten Variablen mittels "function shipping":



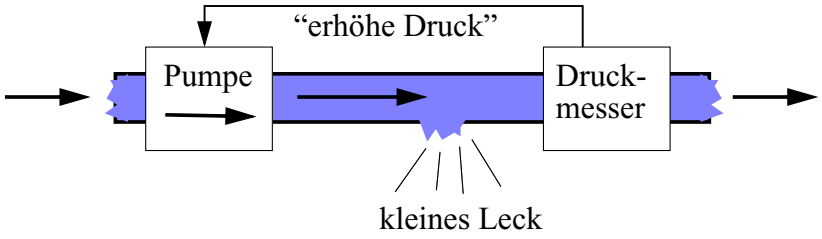
- Es sind verschiedene Grade des Ordnungserhalts denkbar
  - z.B. keine (ungeordnet), FIFO, kausal geordnet, total geordnet

- *FIFO-Ordnung:*

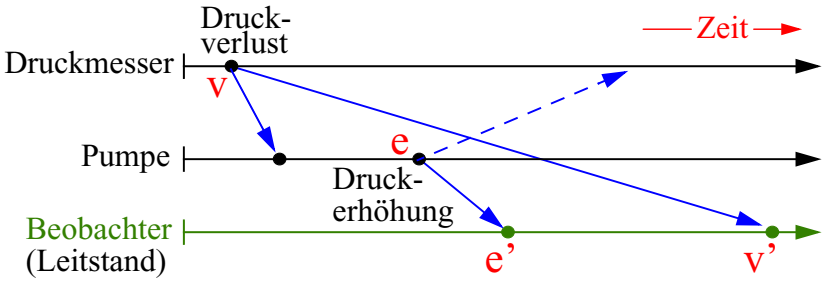
  Alle Multicast-Nachrichten eines (d.h.: *ein und des selben*) Senders an eine Gruppe kommen bei allen Mitgliedern der Gruppe in FIFO-Reihenfolge an
  - Denkübung: wie dies in einem Multicast-Protokoll garantieren?



Probleme mit FIFO-Broadcasts



- *Annahme*: Steuerelemente kommunizieren über FIFO-Broadcasts:



- "Irgendwie" kommt beim Beobachter die Reihenfolge durcheinander!

⇒ *Falsche Schlussfolgerung des Beobachters:*

"Aufgrund einer unbegreiflichen Pumpenaktivität wurde ein Leck erzeugt, wodurch schliesslich der Druck absank."

Man sieht also:

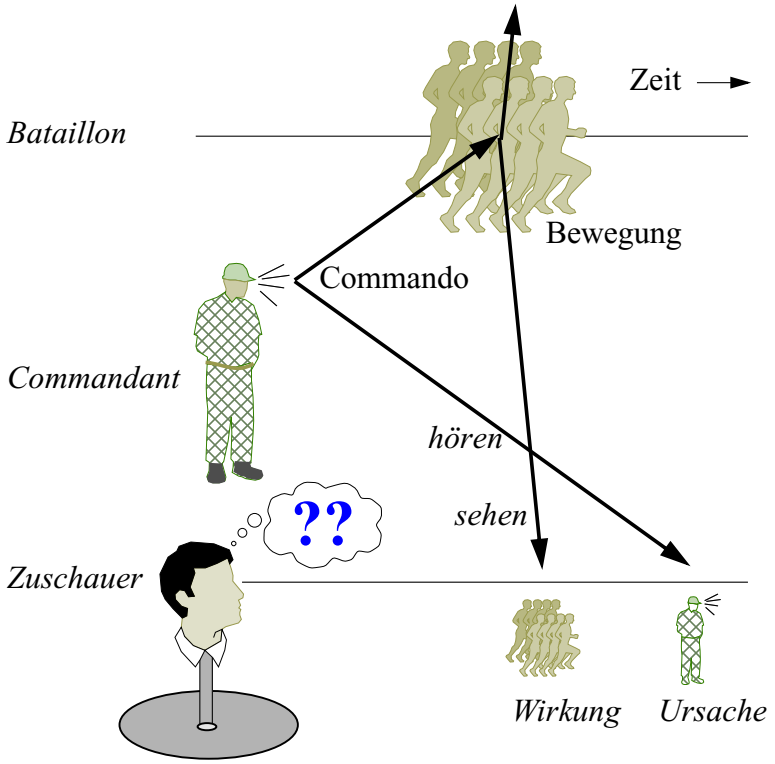
- FIFO-Reihenfolge reicht oft nicht aus, um Semantik zu wahren
- eine Nachricht *verursacht* oft das Senden einer anderen

Kausalität

Das "Broadcastproblem" ist nicht neu

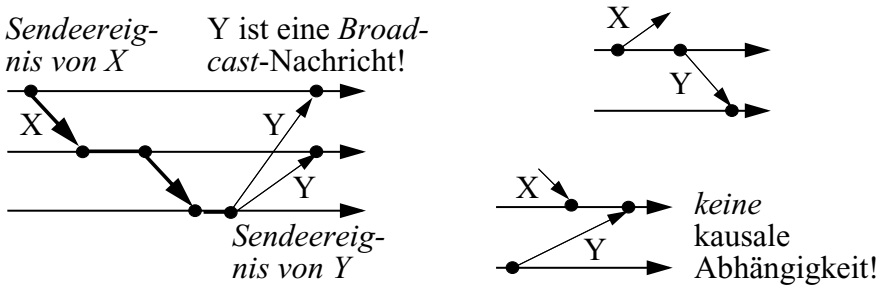
- Licht- und Schallwellen werden in natürlicher Weise per Broadcast verteilt
- Wann handelt es sich dabei um FIFO-Broadcasts?
- Wie ist es mit dem Kausalitätserhalt?

> Wenn ein Zuschauer von der Ferne das Exercieren eines Bataillons verfolgt, so sieht er übereinstimmende Bewegungen desselben plötzlich eintreten, *ehe* er die Commandostimme oder das Hornsignal hört; aber aus seiner Kenntnis der *Causalzusammenhänge* weiß er, daß die Bewegungen die *Wirkung* des gehörten Commandos sind, dieses also jenen *objectiv* vorangehen muß, und er wird sich sofort der Täuschung bewußt, die in der Umkehrung der Zeitfolge in seinen Perceptionen liegt.
>
> *Christoph von Sigwart* (1830-1904) *Logik* (1889)



Kausale Nachrichtenabhängigkeit

- Definition:
  Nachricht Y hängt kausal von Nachricht X ab,
  wenn es im Raum-Zeit-Diagramm einen von links nach rechts verlaufenden Pfad gibt, der vom Sendeereignis von X zum Sendeereignis von Y führt



Beachte:

- Dies lässt sich bei geeigneter Modellierung auch abstrakter fassen (→ logische Zeit später in der Vorlesung und auch Vorlesung "Verteilte Algorithmen")

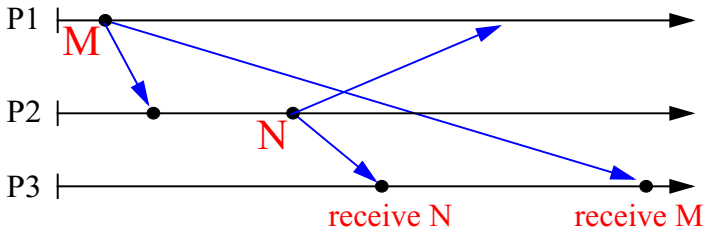
Kausaler Broadcast

Wahrung von Kausalität bei der Kommunikation:

- *Kausale Reihenfolge (Def.):* Wenn eine Nachricht N kausal von einer Nachricht M abhängt, und ein Prozess P die Nachrichten N und M empfängt, dann muss er M vor N empfangen haben

"causal order"

- "Kausale Reihenfolge" (und "kausale Abhängigkeit") lassen sich insbesondere auch auf *Broadcasts* anwenden:

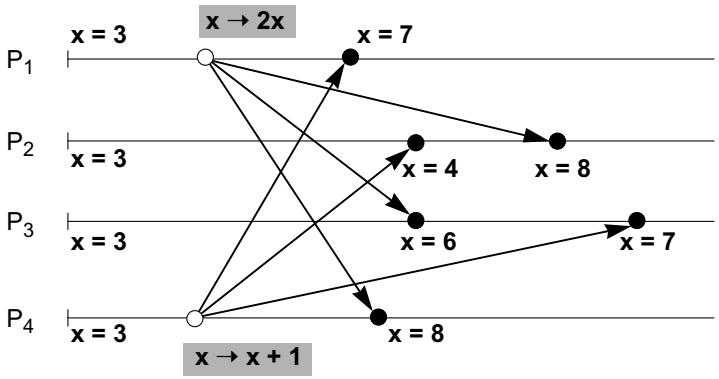


Gegenbeispiel: *Keine* kausalen Broadcasts

- Kausale Reihenfolge *impliziert FIFO-Reihenfolge:* kausale Reihenfolge ist eine Art "globales FIFO"
- Das *Erzwingen* der kausalen Reihenfolge ist mittels geeigneter Algorithmen möglich (→ Vorlesung "Verteilte Algorithmen", z.B. Verallgemeinerung der Sequenzzählermethode für FIFO)

Probleme mit kausalen Broadcasts ?

Beispiel: Aktualisierung einer replizierten Variablen x:



Problem: Ergebnis statt *überall entweder 7 oder 8* hier nun "beides"!

Konkrete Ursache des Problems:

- Broadcasts werden nicht überall "gleichzeitig" empfangen
- dies führt lokal zu verschiedenen Empfangsreihenfolgen

Abstrakte Ursache:

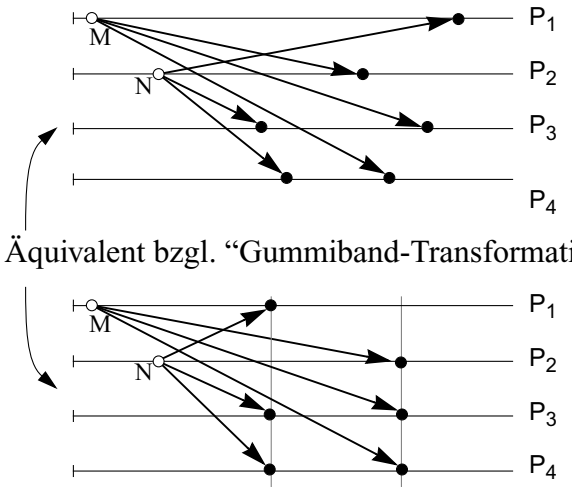
- die Nachrichtenübermittlung erfolgt (erkennbar!) *nicht atomar*

Also:

- auch kausale Broadcasts haben keine "perfekte" Semantik (d.h. Illusion einer speicherbasierten Kommunikation)

Atomarer bzw. "totaler" Broadcast

- *Totale Ordnung:* Wenn zwei Prozesse P_1 und P_2 beide die Nachrichten M und N empfangen, dann empfängt P_1 die Nachricht M vor N genau dann, wenn P_2 die Nachricht M vor N empfängt
  - das Senden wird dabei *nicht* als Empfang der Nachricht beim Sender selbst gewertet!
- Beachte: "Atomar" heisst hier *nicht* "alles oder nichts" (wie etwa beim Transaktionsbegriff von Datenbanken!)



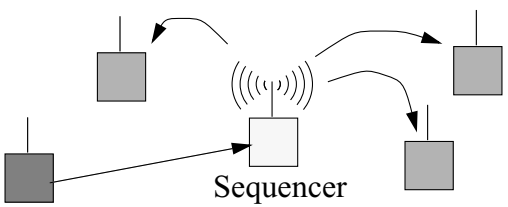
Anschaulich:

- Nachrichten eines Broadcasts werden "überall gleichzeitig" empfangen

Realisierung von atomarem Broadcast

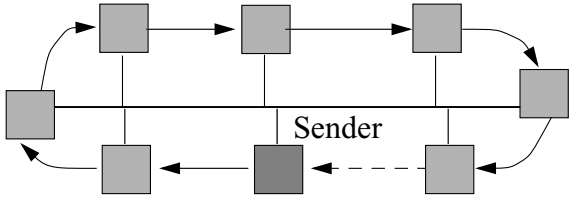
1) Zentraler „Sequencer“, der Reihenfolge festlegt

- ist allerdings ein potentieller Engpass!



- “Unicast” vom Sender zum Sequencer
- Multicast vom Sequencer an alle
- Sequencer *wartet* jew. auf alle Acknowledgements (oder genügt hierfür FIFO-Broadcast?)

2) Token, das auf einem (logischen) Ring kreist



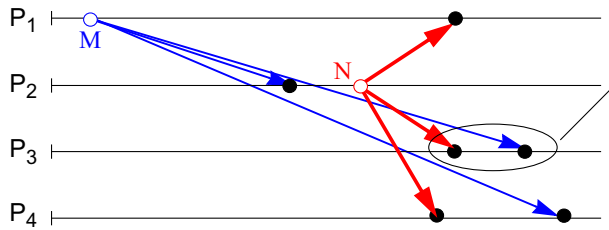
- Token = Senderecht (Token weitergeben!)
- Broadcast selbst könnte z.B. über ein zugrundeliegendes broadcastfähiges Medium erfolgen

- Token führt eine Sequenznummer (inkrementiert beim Senden), dadurch sind alle Broadcasts *global numeriert*
- Empfänger wissen, dass Nachrichten entsprechend der (in den Nachrichten mitgeführten Nummer) ausgeliefert werden müssen
- bei Lücken in den Nummern: dem Token einen Wiederholungswunsch mitgeben (Sender erhält damit implizit ein Acknowledgement)
- Tokenverlust (z.B. durch Prozessor-Crash) durch Timeouts feststellen (Vorsicht: Token dabei nicht versehentlich verdoppeln!)
- einen gecrashten Prozessor (der z.B. das Token nicht entgegennimmt) aus dem logischen Ring entfernen
- Variante (z.B. bei vielen Teilnehmern): Token auf Anforderung direkt zusenden (broadcast: “Token bitte zu mir”), dabei aber Fairness beachten

Denkübung: Geht es auch ohne zentrale Elemente (Sequencer, Token)?

Wie “gut” ist atomarer Broadcast?

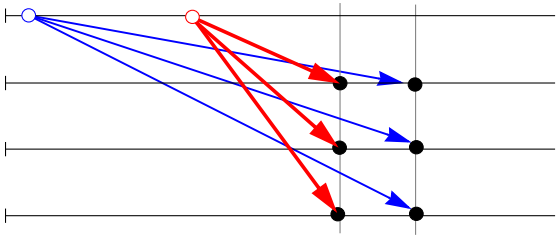
1) Ist atomar auch kausal?



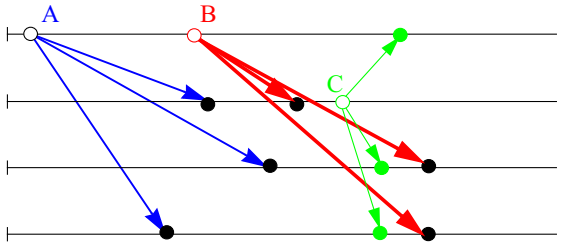
Nicht kausal!

Atomar: P3 und P4 empfangen beide M, N und zwar in gleicher Reihenfolge

2) Ist atomar wenigstens FIFO?



3) Ist atomar + FIFO vielleicht kausal?



Bem.: 1) ist ebenfalls ein Gegenbeispiel, da M, N FIFO-Broadcast ist!

Kausaler atomarer Broadcast

- Fazit:
  - atomare Übermittlung $\not\Rightarrow$ kausale Reihenfolge
  - atomare Übermittlung $\not\Rightarrow$ FIFO-Reihenfolge
  - atomare Übermittlung + FIFO $\not\Rightarrow$ kausale Reihenfolge

- Vergleich mit speicherbasierter Kommunikation:
  - Kommunikation über gemeinsamen Speicher ist *atomar* (alle „sehen“ das Geschriebene gleichzeitig - wenn sie hinschauen)
  - Kommunikation über gemeinsamen Speicher *wahrt Kausalität* (die Wirkung tritt unmittelbar mit der Ursache, dem Schreibereignis, ein)

- Vergleichbares Kommunikationsmodell per Nachrichten: *Kausaler atomarer Broadcast*
  - kausaler Broadcast + totale Ordnung
  - man nennt daher kausale, atomare Übermittlung auch *virtuell synchrone Kommunikation*
  - Denkübung: realisieren die beiden Implementierungen “zentraler Sequencer” bzw. “Token auf Ring” die virtuell synchrone Kommunikation?

Stichwort: Virtuelle Synchronität

- Idee: Ereignisse finden zu verschiedenen Realzeitpunkten statt, aber zur *gleichen logischen Zeit*
  - *logische Zeit* berücksichtigt nur die Kausalstruktur der Nachrichten und Ereignisse; die exakte Lage der Ereignisse auf dem “Zeitstrahl” ist verschiebbar (Dehnen / Stauchen wie auf einem Gummiband)

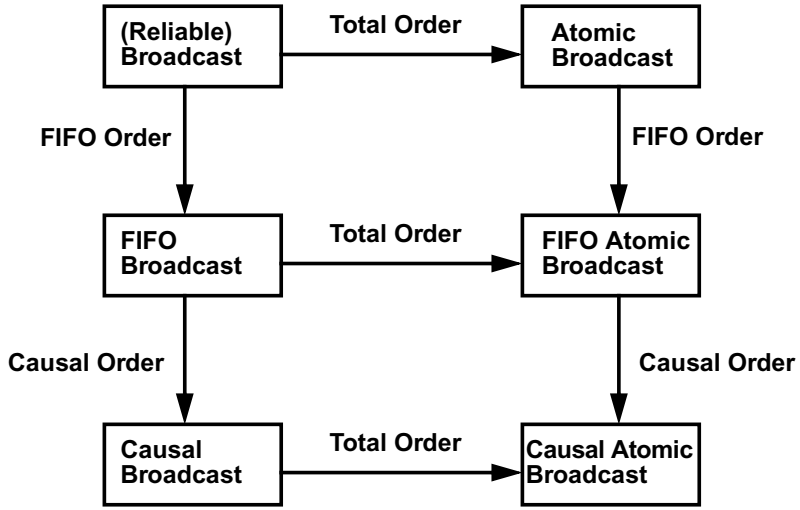
- *Innerhalb* des Systems ist synchron (im Sinne von “gleichzeitig”) und virtuell synchron *nicht unterscheidbar*
  - identische totale Ordnung aller Ereignisse
  - identische Kausalbeziehungen

- Folge: Nur mit Hilfe Realzeit / echter Uhr könnte ein externer Beobachter den Unterschied feststellen

---

Den Begriff “logische Zeit” werden wir später noch genauer fassen (mehr dazu dann wieder in der Vorlesung “Verteilte Algorithmen”)

Broadcast - schematische Übersicht



- Warum nicht ein einziger Broadcast, der alles kann? "Stärkere Semantik" hat auch Nachteile:
  - Performance-Einbussen
  - Verringerung der potentiellen Parallelität
  - aufwändiger zu implementieren

- Bekannte "Strategie":
  - man begnügt sich daher, falls es der Anwendungsfall gestattet, oft mit einer billigeren aber weniger perfekten Lösung
  - Motto: so billig wie möglich, so „perfekt" wie nötig
  - man sollte aber die Schwächen einer Billiglösung kennen!

$\Rightarrow$ grössere Vielfalt $\Rightarrow$ komplexer bzgl. Verständnis und Anwendung

Multicast

- Definition von Multicast (informell): "*Multicast ist ein Broadcast an eine Teilmenge von Prozessen*"
  - diese Teilmenge wird "*Multicast-Gruppe*" genannt

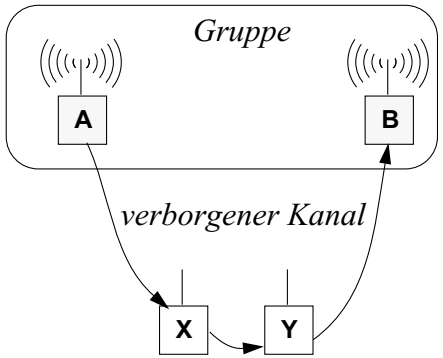
- Daher: Alles, was bisher über Broadcast gesagt wurde, gilt (innerhalb der Teilmenge) auch weiterhin:
  - zuverlässiger Multicast
  - FIFO-Multicast
  - kausaler Multicast
  - atomarer Multicast
  - kausaler atomarer Multicast

Unterschied: Wo bisher "alle Prozesse" gesagt wurde, gilt nun "alle Prozesse innerhalb der Teilmenge"

- Zweck Multicast-Gruppe
  - "Selektiver Broadcast"
  - Vereinfachung der Adressierung (z.B. statt Liste von Einzeladressen)
  - Verbergen der Gruppenzusammensetzung (vgl. Port-Konzept)
  - "Logischer Unicast": Gruppen ersetzen Individuen (z.B. für transparente Replikation)

Problem der “Hidden Channels”

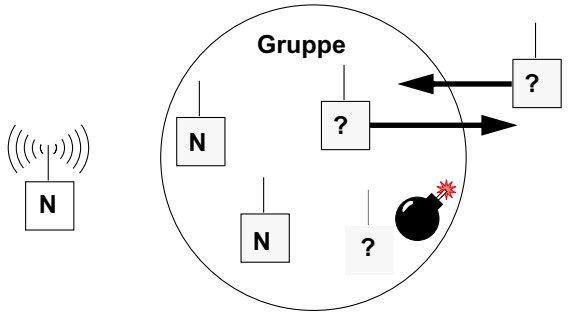
- Kausalitätsbezüge verlassen (z.B. durch Gruppenüberlappung) die Multicast-Gruppe und kehren später wieder



- Soll nun das Senden von B als kausal abhängig vom Senden von A gelten?

- *Global* gesehen ist das der Fall, *innerhalb* der Gruppe ist eine solche Abhängigkeit jedoch nicht erkennbar

Gruppen-Management / -Membership



- Dynamische Gruppe: wie sieht die Gruppe “momentan” aus?

- Haben alle Mitglieder (gleichzeitig?) die gleiche Sicht?

- Was bedeutet “alle Gruppenmitglieder”?

  - *Beitritt* (“join”) zu einer Gruppe
  - *Austritt* (“leave”) aus einer Gruppe
  - *Crash: “korrekt” → “fehlerhaft”*

  } *während* eines Multicasts?

- Beachte:

  - “Zufälligkeiten” (z.B. Beitrittszeitpunkt kurz vor / nach dem Empfang einer Einzelnachricht) sollten (soweit möglich) vermieden werden (→ Nichtdeterminismus; Nicht-Reproduzierbarkeit)

- Folge:

  - Zu jedem Zeitpunkt soll *Übereinstimmung* über *Gruppenzusammensetzung* und *Fehlerzustand* (“korrekt”, “ausgefallen” etc.) aller Mitglieder erzielt werden

- Problem: Wie erzielt man diese Übereinkunft?

Wechsel der Gruppenmitgliedschaft

"...während..." gibt es nicht (→ "virtuell synchron")

- Forderungen:
  - Eintritt und Austritt sollen *atomar* erfolgen:
    - Die Gruppe muss bei allen (potentiellen) Sendern an die Gruppe hinsichtlich der Ein- und Austrittszeitpunkte jedes Gruppenmitglieds übereinstimmen
  - *Kausalität* soll gewahrt bleiben

- Realisierungsmöglichkeit:
  - konzeptuell führt jeder Prozess eine Liste mit den Namen aller Gruppenmitglieder
    - Realisierung als zentrale Liste (Fehlertoleranz und Performance?)
    - oder Realisierung als verteilte, replizierte Liste
  - massgeblich ist die zum Sendezeitpunkt gültige Mitgliederliste
  - Listenänderungen werden (virtuell) synchron durchgeführt:
    - bei einer zentralen Liste kein Problem
    - bei replizierten Listen: verwende *kausalen atomaren Multicast*

      Schwierigkeit: *Bootstrapping-Problem* (mögliche Lösung: Service-Multicast zur dezentralen Mitgliedslistenverwaltung löst dies für sich selbst über einen zentralen Server)

Behandlung von Prozessausfällen

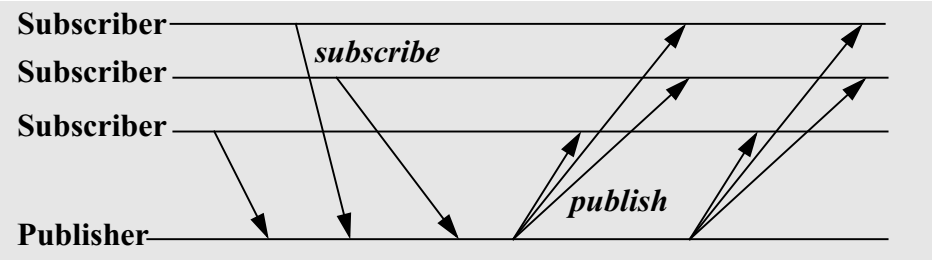
- Forderungen:
  - *Ausfall* eines Prozesses soll *global atomar* erfolgen:
    - Übereinstimmung über Ausfallzeitpunkt jedes Gruppenmitglieds
  - *Reintegration* nach einem vorübergehenden Ausfall soll *atomar* erfolgen:
    - Übereinstimmung über Reintegrationszeitpunkt

- Realisierungsmöglichkeit:
  - Ausfallzeitpunkt:
    - Prozesse dürfen nur Fail-Stop-Verhalten zeigen: *"Einmal tot, immer tot"*
    - Gruppenmitglieder erklären Opfer per kausalem, atomarem Multicast übereinstimmend für tot: *"Ich sage tot – alle sagen tot!"*
    - Beachte: *"Lebendiges Begraben"* ist nicht ausschliessbar! (Irrtum eines "failure suspectors" aufgrund zu langsamer Nachrichten)
    - Fälschlich für tot erklärte Prozesse sollten unverzüglich Selbstmord begehen
  - Reintegration:
    - Jeder tote (bzw. für tot erklärte) Prozess kann der Gruppe nur nach dem offiziellen Verfahren ("Neuaufnahme") wieder beitreten

- Damit erfolgen Wechsel der Gruppenmitgliedschaft und Crashes in "geordneter Weise" für *alle* Teilnehmer

Push bzw. Publish & Subscribe

- Im Unterschied zum klassischen “Request / Reply-” bzw. “Pull-Paradigma”
  - wo Clients die gewünschte Information aktiv anfordern müssen
  - ein Client aber nicht weiss, ob bzw. wann sich eine Information geändert hat
  - dadurch periodische Nachfrage beim Server notwendig sind (“polling”)
- Subscriber (= Client) meldet sich für den Empfang der gewünschten Information an
  - z.B. “Abonnement” eines Informationskanals (“channel”)
  - u.U. auch dynamische, virtuelle Kanäle (→ “subject-based addressing”)



- Subscriber erhält automatisch (aktualisierte) Information, sobald diese zur Verfügung steht
  - “callback” des Subscribers (= Client) durch den Publisher (= Server)
  - push: “event driven” ↔ pull: “demand driven”
- “Publish” entspricht (logischem) Multicast
  - “subscribe” entspricht dann einem “join” einer Multicast-Gruppe
  - Zeitaktualität, Stärke der Multicast-Semantik und Grad an Fehlertoleranz wird oft unscharf als “Quality of Service” bezeichnet

Tupelräume

- Gemeinsam genutzter (“virtuell globaler”) Speicher
- Blackboard- oder Marktplatz-Modell
  - Daten können von beliebigen Teilnehmern eingefügt, gelesen und entfernt werden
  - relativ starke Entkoppelung der Teilnehmer
- Tupel = geordnete Menge typisierter Datenwerte
- Entworfen 1985 von D. Gelernter (für die Sprache Linda)
- Operationen:
  - out (t): Einfügen eines Tupels t in den Tupelraum
  - in (t): Lesen und Löschen von t aus dem Tupelraum
  - read (t): Lesen von t im Tupelraum
- Inhaltsadressiert (“Assoziativspeicher”)
  - Vorgabe eines Zugriffsmusters (bzw. “Suchmaske”) beim Lesen, damit Ermittlung der restlichen Datenwerte eines Tupels (“wild cards”)
  - Beispiel: int i,j; in(“Buchung”, ?i, ?j) liefert *ein* “passendes” Tupel
  - analog zu einigen relationalen Datenbankabfragesprachen
- Synchrone und asynchrone Leseoperationen
  - ‘in’ und ‘read’ blockieren, bis ein passendes Tupel vorhanden ist
  - ‘inp’ und ‘readp’ blockieren nicht, sondern liefern als Prädikat (Daten vorhanden?) ‘wahr’ oder ‘falsch’ zurück

Tupelräume (2)

- Mit Tupelräumen sind natürlich die üblichen Kommunikationsmuster realisierbar, z.B. Client-Server:

```
/* Client */
...
out("Anfrage" client_Id, Parameterliste);
in("Antwort", client_Id, ?Ergebnisliste);
...
/* Server*/
...
while (true)
{  in("Anfrage", ?client_Id, ?Parameterliste);
   ...
   out("Antwort", client_Id, Ergebnisliste);
}
```

Beachte: Zuordnung des "richtigen" Clients über die client_Id

- Kanonische Erweiterungen des Modells
  - *Persistenz* (Tupel bleiben nach Programmende erhalten, z.B. in DB)
  - *Transaktionseigenschaft* (wichtig, wenn mehrere Prozesse parallel auf den Tupelraum bzw. gleiche Tupel zugreifen)
- Problem: effiziente, skalierbare Implementierung?
  - *zentrale Lösung*: Engpass
  - *replizierter Tupelraum* (jeder Rechner hat eine vollständige Kopie des Tupelraums; schnelle Zugriffe, jedoch hoher Synchronisationsaufwand)
  - *aufgeteilter Tupelraum* (jeder Rechner hat einen Teil des Tupelraums; 'out'-Operationen können z.B. lokal ausgeführt werden, 'in' evtl. mit Broadcast)
- Kritik: globaler Speicher ist der strukturierten Programmierung und der Verifikation abträglich
  - unüberschaubare potentielle Seiteneffekte

JavaSpaces

- "Tupelraum" für Java
  - gespeichert werden Objekte → neben Daten auch "Verhalten"
  - Tupel entspricht Gruppen von Objekten
- Teil der Jini-Infrastruktur für verteilte Java-Anwendungen
  - Kommunikation zwischen entfernten Objekten
  - Transport von Programmcode vom Sender zum Empfänger
  - gemeinsame Nutzung von Objekten
- Operationen
  - *write*: mehrfache Anwendung erzeugt verschiedene Kopien
  - *read*
  - *readifexists*: blockiert (im Gegensatz zu read) nicht; liefert u.U. 'null'
  - *takeifexists*
  - *notify*: Benachrichtigung (mittels eines Ereignisses), wenn ein passendes Objekt in den JavaSpace geschrieben wird
- Nutzen (neben Kommunikation)
  - atomarer Zugriff auf Objektgruppen
  - zuverlässiger verteilter Speicher
  - persistente Datenhaltung für Objekte

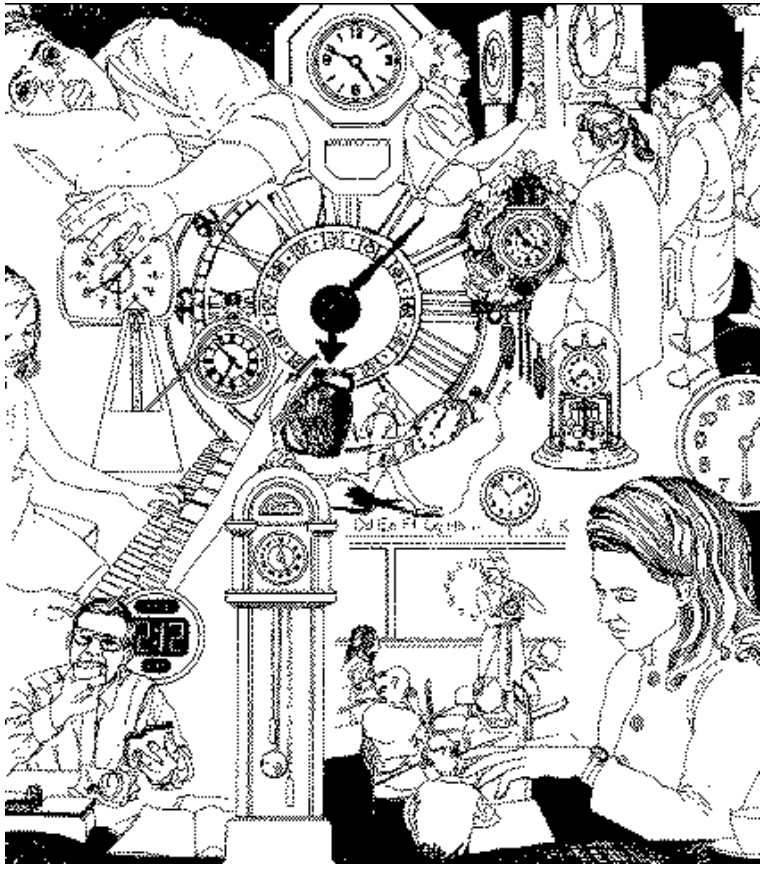
  } → aber keine Festlegung, ob eine Implementierung fehlertolerant ist und einen Crash überlebt
- *Implementierung* könnte z.B. auf einer relationalen oder objektorientierten Datenbank beruhen
- *Semantik*: Reihenfolge der Wirkung von Operationen verschiedener Threads ist nicht festgelegt
  - selbst wenn ein *write* vor einem *read* beendet wird, muss *read* nicht notwendigerweise das lesen, was *write* geschrieben hat

Logische Zeit und wechselseitiger Ausschluss

Zeit?

Ich halte ja eine Uhr für überflüssig.
Sehen Sie, ich wohne ja ganz nah beim Rathaus. Und jeden Morgen, wenn ich ins Geschäft gehe, da schau ich auf die Rathausuhr hinauf, wieviel Uhr es ist, und da merke ich's mir gleich für den ganzen Tag und nütze meine Uhr nicht so ab.

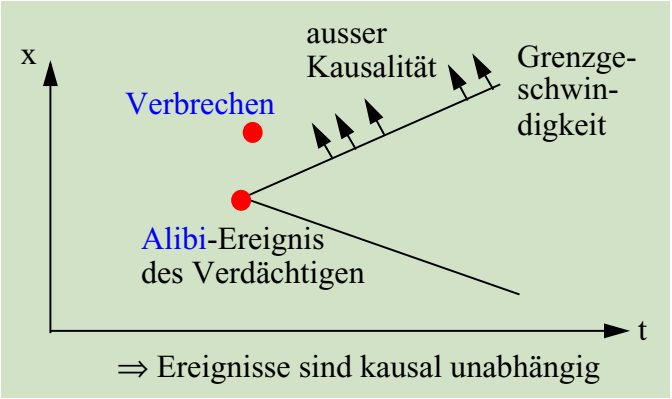
Karl Valentin

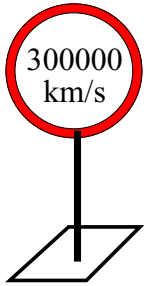


Kommt Zeit, kommt Rat

1. Volkszählung: Stichzeitpunkt in der Zukunft
   - liefert eine gleichzeitige, daher kausaltreue "Beobachtung"

2. Kausalitätsbeziehung zwischen Ereignissen ("Alibi-Prinzip")
   - wurde Y später als X geboren, dann kann Y unmöglich Vater von X sein
     → Testen verteilter Systeme: Fehlersuche / -ursache





"speed limit
of causality"
(P. Langevin)

3. Fairer wechselseitiger Ausschluss
   - bedient wird, wer am längsten wartet

4. Viele weitere nützliche Anwendungen von "Zeit" in unserer "verteilten realen Welt"
   - z.B. kausaltreue Beobachtung durch "Zeitstempel" der Ereignisse

Eigenschaften der "Realzeit"

Formale Struktur eines Zeitpunktmodells:

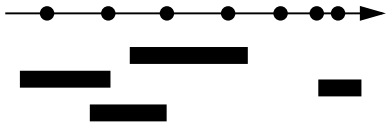
- transitiv
- irreflexiv
- linear

} → lineare Ordnung ("später")

- unbeschränkt ("Zeit ist ewig": Kein Anfang oder Ende)
- dicht (es gibt immer einen Zeitpunkt dazwischen)
- kontinuierlich
- metrisch
- vergeht "von selbst" → jeder Zeitpunkt wird schliesslich erreicht

---

Ist das "Zeitpunktmodell"
adäquat? Zeitintervalle?



- Wann tritt das Ereignis (?) "Sonne wird rot" am Abend ein?

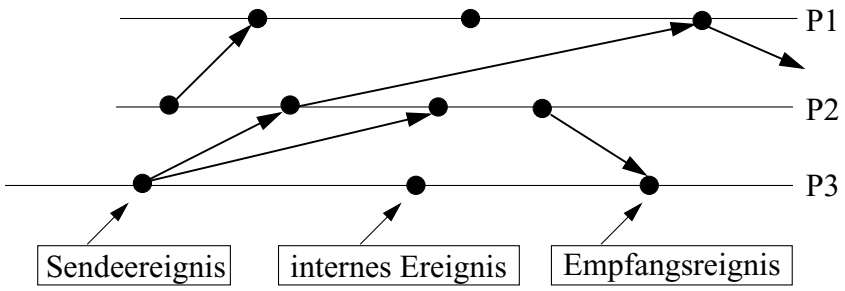
Welche Eigenschaften benötigen wir wirklich?

- dazu vorher klären: was wollen wir mit "Zeit" anfangen?

→ "billigeren" Ersatz für fehlende globale Realzeit!
(sind die rellen / rationalen / ganzen Zahlen gute Modelle?)

- wann genügt "logische" (statt "echter") Zeit? (Und was ist das genau??)

Raum-Zeitdiagramme und die Kausalrelation



- interessant: von links nach rechts verlaufende “Kausalitätspfade”

- Definiere eine *Kausalrelation* ‘<‘ auf der Menge E aller Ereignisse :

Es sei **x** < **y** genau dann, wenn:

1) **x** und **y** auf dem gleichen Prozess stattfinden und **x** vor **y** kommt, *oder*

2) **x** ist ein Sendeereignis und **y** ist das korrespondierende Empfangsereignis, *oder*

3) $\exists$ **z** mit **x** < **z** $\wedge$ **z** < **y**

- Relation wird oft als “*happened before*” bezeichnet
  - eingeführt von L. Lamport (1978)
  - aber Vorsicht: damit ist nicht direkt eine "zeitliche" Aussage getroffen!

Logische Zeitstempel von Ereignissen

- Zweck: Ereignissen eine Zeit geben (“dazwischen” egal)
- Gesucht: Abbildung C: E $\rightarrow$ **N**

Clock — natürliche Zahlen

- Für e $\in$ E heisst C(e) *Zeitstempel* von e
- C(e) bzw. e *früher* als C(e’) bzw. e’, wenn C(e) < C(e’)

- Sinnvolle Forderung:

Kausalrelation

Uhrenbedingung: e < e’ $\Rightarrow$ C(e) < C(e’)

Ordnungshomomorphismus — Zeitrelation “früher”

Interpretation (“Zeit ist kausaltreu”):

Wenn ein Ereignis e ein anderes Ereignis e’ beeinflussen kann, dann muss e einen kleineren Zeitstempel als e’ haben

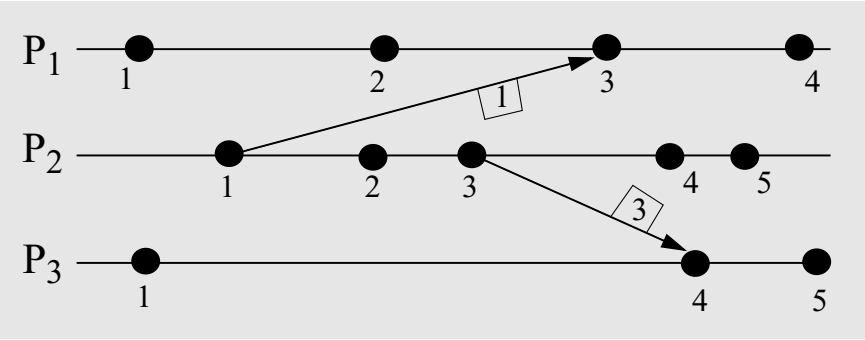
Logische Uhren von Lamport

Communications ACM 1978:
Time, Clocks, and the Ordering of Events in a Distributed System

C: (E,<) → (**N**,<) Zuordnung von Zeitstempeln

Kausal-relation

e < e' ⟹ C(e) < C(e') Uhrenbedingung



Protokoll zur Implementierung der Uhrenbedingung:

- Lokale Uhr (= "Zähler") *tickt* "bei" *jedem* Ereignis
- Sendeereignis: Uhrwert mitsenden (*Zeitstempel*)
- Empfangsereignis: *max*(lokale Uhr, Zeitstempel)

zuerst, erst danach "ticken"!

Behauptung:

Protokoll respektiert Uhrenbedingung

Beweis: Kausalitätspfade sind monoton...

Lamport-Zeit: Nicht-Injektivität

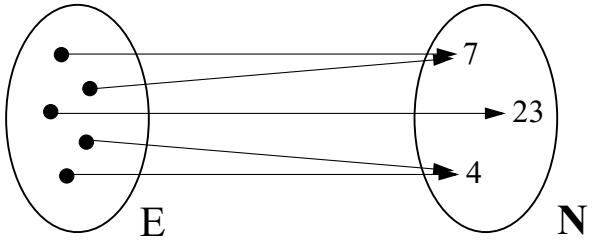


Abbildung ist nicht injektiv

- Wichtig z.B. für: "Wer die kleinste Zeit hat, gewinnt"

- Lösung:

  Lexikographische Ordnung (C(e),i), wobei i die Prozessnummer bezeichnet, auf dem e stattfindet

  Ist injektiv, da alle lokalen Ereignisse verschiedene Zeitstempel C(e) haben ("tie breaker")

  - *lin*. Ordnung (a,b) < (a',b') ⇔ a<a' ∨ a=a' ∧ b<b'

  → alle Ereignisse haben *verschiedene* Zeitstempel

  → Kausalitätserhaltende Abb. (E,<) → (**N**×**N**, <)

Jede (nicht-leere) Menge von Ereignissen hat so ein eindeutig "frühestes"!

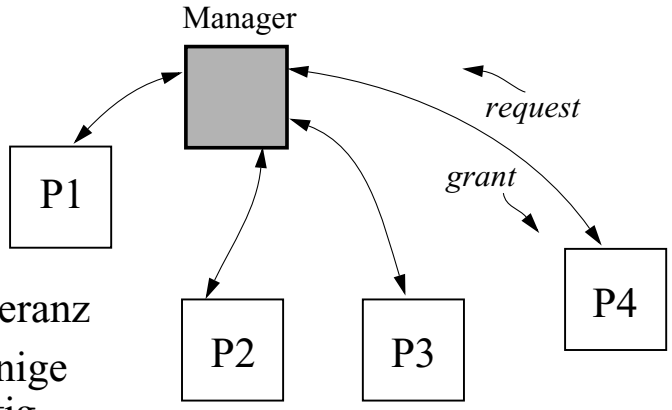
Wechselseitiger Ausschluss

- "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:

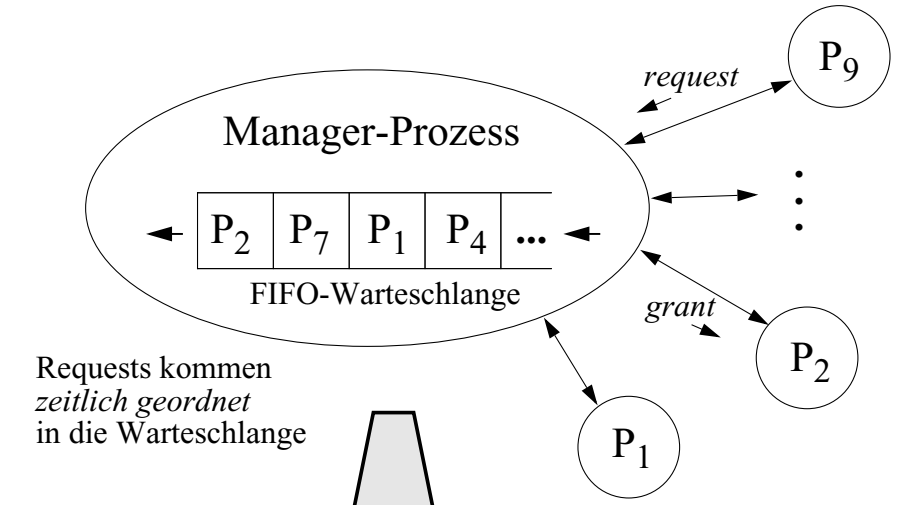


→ Engpass

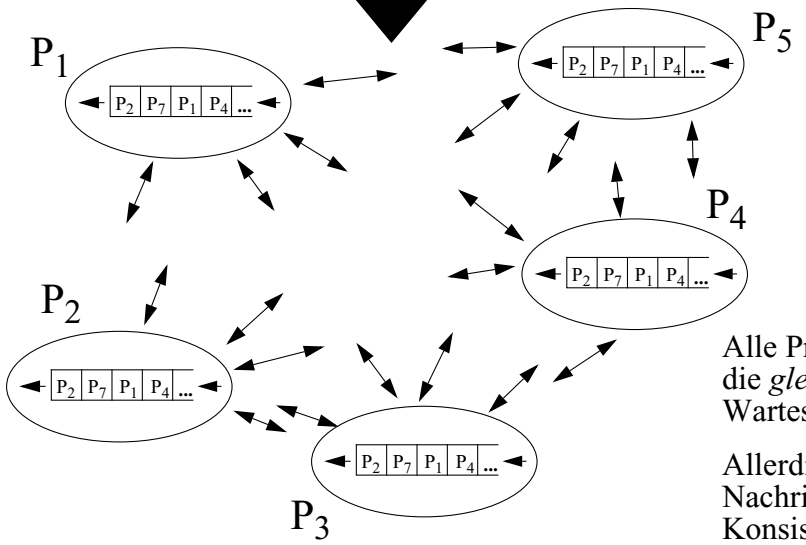
→ keine Fehlertoleranz

→ 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

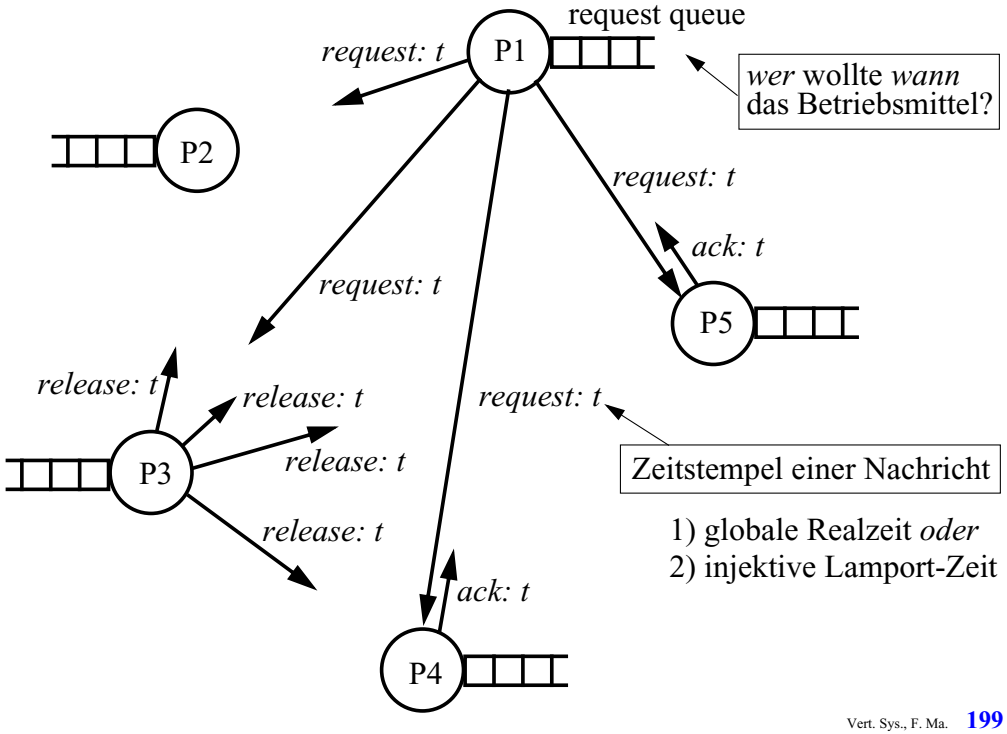
- Hier: Feste Anzahl von Prozessen; ein einziges exklusives Betriebsmittel
- Synchronisierung mit request- / release-Nachrichten

"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.

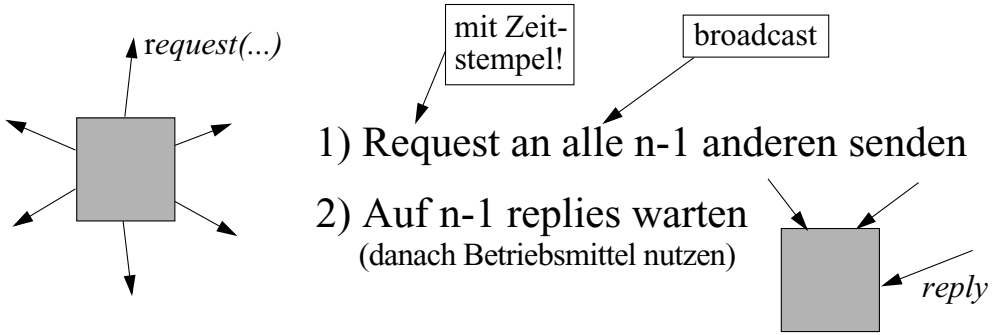
- Frühester request ist global eindeutig.
  ⇒ bei 5): sicher, dass kein früherer request mehr kommt (wieso?)
- 3(n-1) Nachrichten pro "request" (n = Zahl der Prozesse)

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)

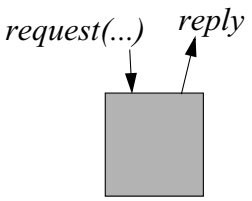
Ein anderer verteilter Algorithmus für den wechselseitigen Ausschluss

(Ricart / Agrawala, 1981)

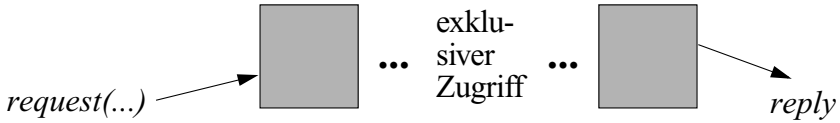
- 2(n-1) Nachrichten statt 3(n-1) wie bei obigem Verfahren
  (*reply-Nachricht* übernimmt Rolle von release und ack)



- 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 Adressen

- Namen dienen der (eindeutigen) *Bezeichnung* und der *Identifikation* von Objekten

- 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
  - höchstens evtl. bei Heirat, Annahme eines Alias etc.

- 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 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!

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)

Exkurs: 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.

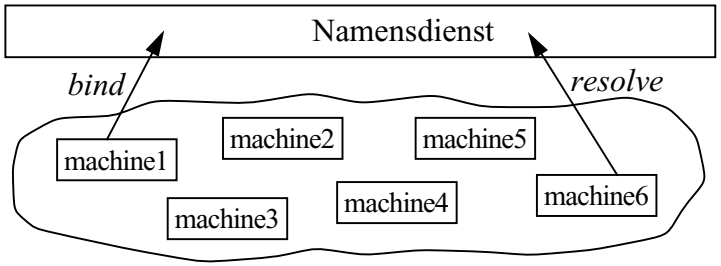
This *inherent randomness* in decay time has profound implications, which we will now exploit to generate random numbers. For if there's no way to know when a given Krypton-85 nucleus will decay then, given an collection of them, there's no way to know when the next one of them will shoot its electron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

Exkurs-Ende (Zufall)

Verteilte Namensverwaltung



logisch ein einziger (zentraler) Dienst; tatsächlich verteilt realisiert

- Jeder Kontext (Teilnamensraum, -bereich) 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

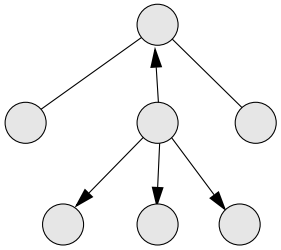
bzw. "user agent"

- 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

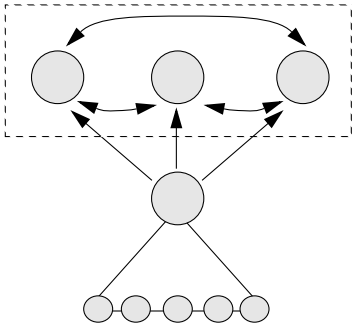
---

Broadcast

- falls zuständiger Nameserver unbekannt
  ("wer ist für XYZ zuständig?" oder: "wer ist hier der Nameserver?")
- ist aufwändig, 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

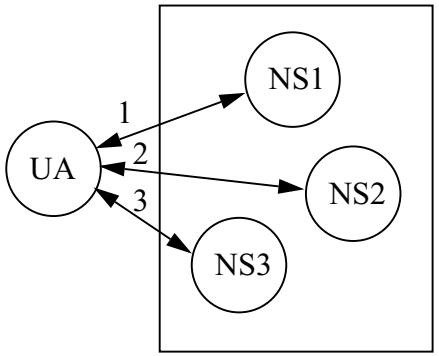


- Nameserver kennt mehrere übergeordnete Nameserver
- Broadcast an ganze Servergruppe, oder Einzelnachricht an "naheliegenden" Server; anderen Server erst nach Ablauf eines Timeouts befragen

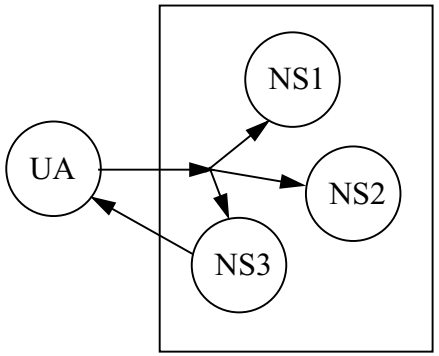
- Replizierte Server konsistent halten
  - evtl. nur von Zeit zu Zeit gegenseitig aktualisieren (falls veraltete Information tolerierbar)
  - Update auch dann sicherstellen, wenn einige Server zeitweise nicht erreichbar sind (periodisches Wiederholen von update-Nachrichten)
  - Einträge mit Zeitstempel 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

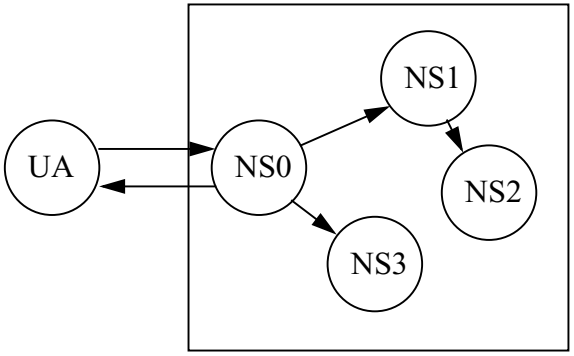
- User Agent (UA) bzw. “Name Agent” auf Client-Seite
  - hinzugebundene Schnittstelle aus Bibliothek, oder
  - eigener lokaler Service-Prozess



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

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.

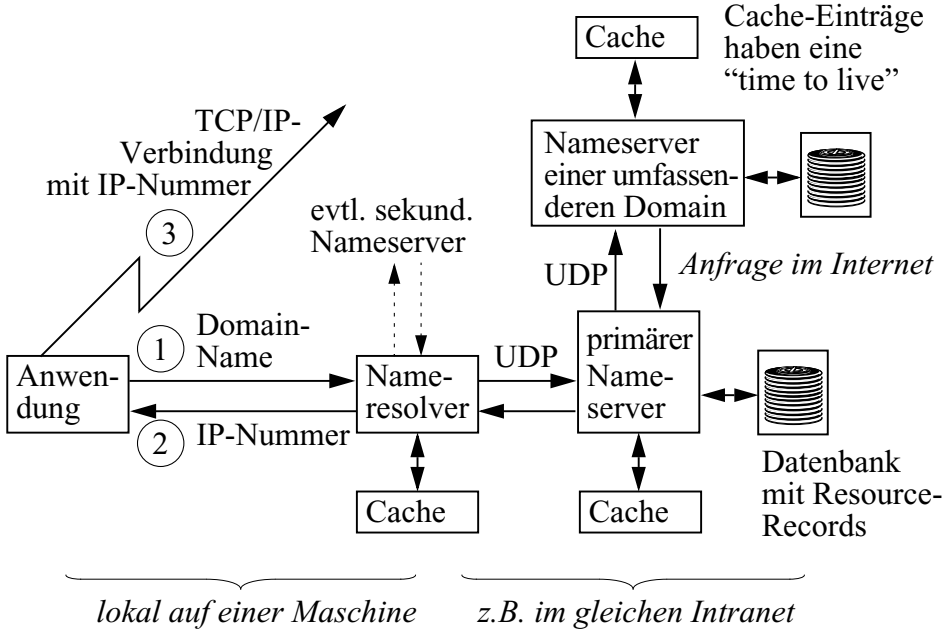
eu
uni-cooltown.eu
informatik.uni-cooltown.eu
nano.informatik.uni-cooltown.eu
pc6.nano.informatik.uni-cooltown.eu

- 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)

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...