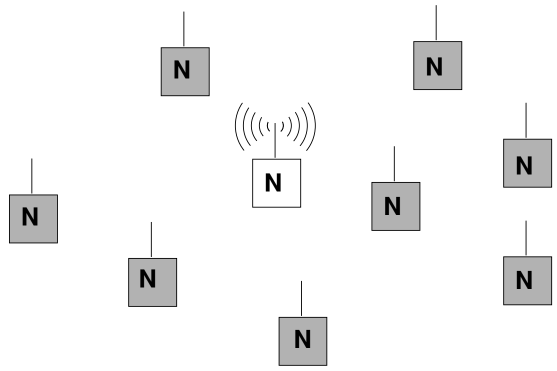
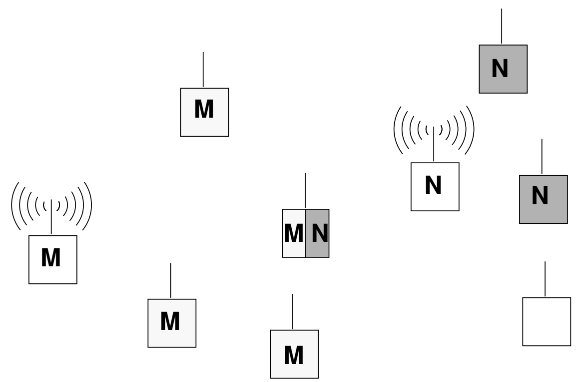
Gruppen-kommunikation

Gruppenkommunikation



Broadcast: Senden an die *Gesamtheit* aller Teilnehmer

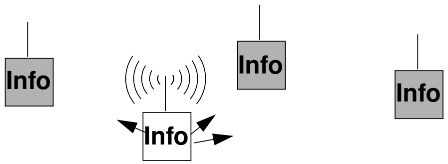


Multicast: Senden an eine *Untergruppe* aller Teilnehmer

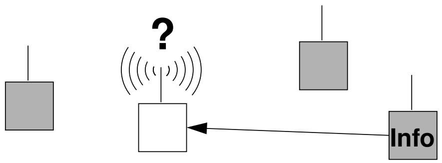
- entspricht Broadcast bezogen auf die Gruppe
- verschiedene Gruppen können sich ggf. überlappen
- jede Gruppen hat eine Multicastadresse

Anwendungen von Gruppenkommunikation

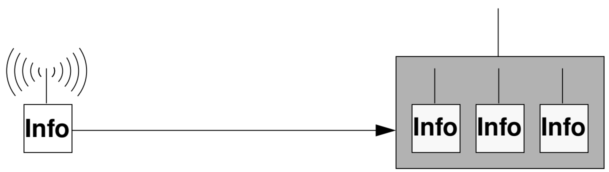
- *Informieren*
  - z.B. Newsdienste, Konferenzsysteme etc.



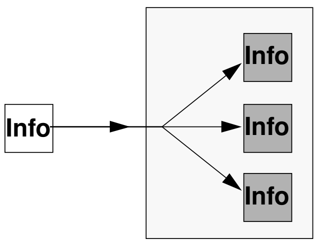
- *Suchen*
  - z.B. Lokation von Objekten und Diensten



- *“Logischer Unicast”* an replizierte Komponenten



logisch

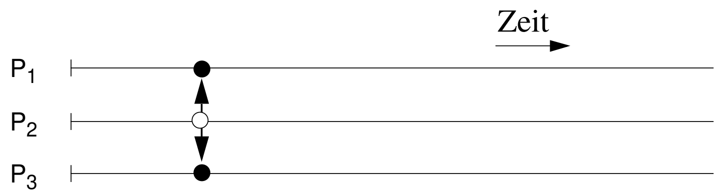


tatsächlich

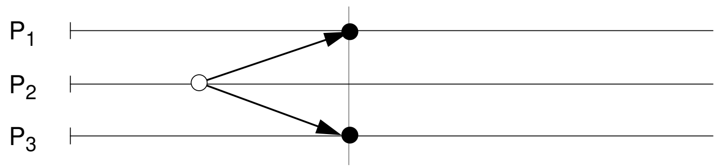
Typische Anwendungs-klasse von Replikation: Fehlertoleranz

Gruppenkommunikation - idealisierte Semantik

- Modellhaftes Vorbild: Speicherbasierte Kommunikation in zentralistischen Systemen
  - augenblicklicher Empfang
  - vollständige Zuverlässigkeit



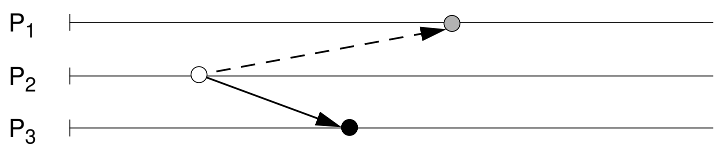
- Nachrichtenbasierte Kommunikation: Idealisierte Sicht
  - (verzögerter) gleichzeitiger Empfang
  - vollständige Zuverlässigkeit



Gruppenkommunikation - tatsächliche Situation

- Medium (Netz) ist oft nicht multicastfähig
  - LANs teilweise (z.B. klassisches Ethernet), jedoch i.a. nur innerhalb eines einzigen Segments (vgl. aber VLANs)
  - multicastfähiges Netz ist effizient (Hardwareunterstützung), typischerweise jedoch 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"
  - indeterministische Zeitverzögerung --> Empfang zu unterschiedlichen Zeiten
  - nur bedingte Zuverlässigkeit der Übermittlung



- Ziel von Broadcast / Multicast-Protokollen:
  - möglichst gute Approximation einer speicherbasierten Kommunikation
  - möglichst hohe Verlässlichkeit und Effizienz

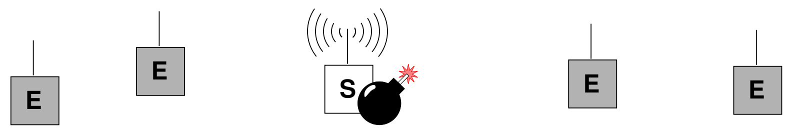
- Beachte: Verlust von Nachrichten und sonstige Fehler sind bei Broadcast ein viel wichtigeres Problem als beim "Unicast"! (Wieso?)

- Hauptproblem bei der Realsierung von Broadcast: Zuverlässigkeit und garantierte Empfangsreihenfolge

Senderausfall beim Broadcast

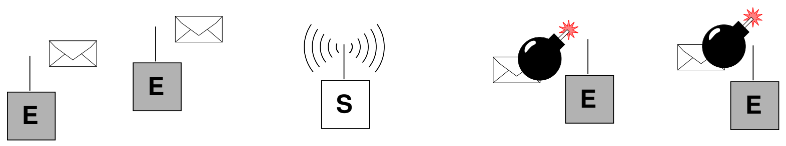
a) Sender fällt aus: *kein* Empfänger erhält Nachricht

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



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

- wenn Broadcast durch Senden vieler Einzelnachrichten realisiert ist
- "ungünstiger" Fall: *Uneinigkeit*



- mögliche Abhilfe: Empfänger leiten die Nachricht untereinander weiter (Denkübung: Müssen die Empfänger dazu wissen, ob ein Sender abgestürzt ist? Wenn ja: wie? Wenn nein: schadet eine falsche Verdächtigung nicht?)

- Uneinigkeit der Empfänger kann die Ursache für sehr ärgerliche Folgeprobleme sein! (Da wäre es manchmal besser, *kein* Prozess hätte die Nachricht empfangen!)

Zuverlässigkeitsstrategie “Best effort” bei Broadcasts

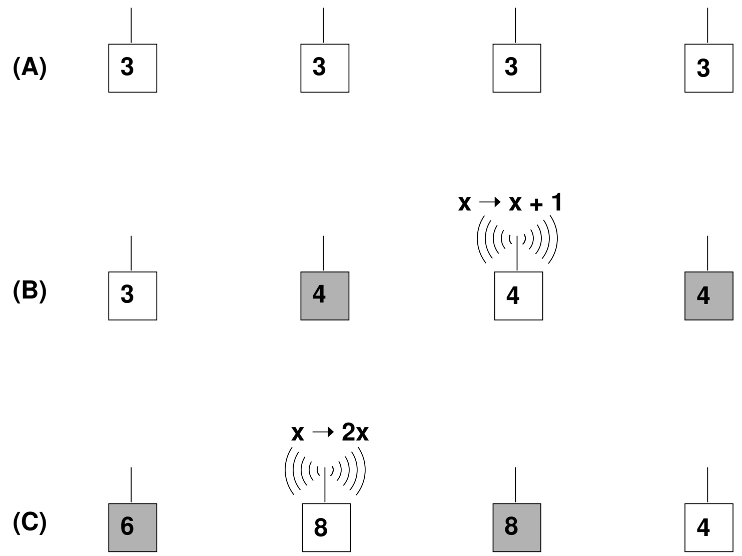
- Fehlermodell: Verlust von Nachrichten (und ggf. temporärer Crash von Prozessen)
  - Nachrichten können aus unterschiedlichen Gründen verloren gehen (z.B. Netzüberlastung, Empfänger hört gerade nicht zu...)
- Euphemistische Bezeichnung, da keine extra Anstrengung
  - typischerweise einfache Realisierung ohne Acknowledgements etc.
- Keinerlei Garantien
  - unbestimmt, wieviele / welche Empfänger eine Broadcastnachricht im Fehlerfall 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. Lastdaten oder unverbindliche “Tips“ und “Empfehlungen“
  - Information, die ggf. Einfluss auf die Effizienz haben, nicht aber die Korrektheit betreffen
- Ggf. als Grundlage zur Realisierung höherer Protokolle
  - oft basierend auf multicastfähigen Netzen
  - günstig bei zuverlässigen physische Kommunikationsmedien (wenn Fehlerfall sehr selten --> aufwendiges Recovery auf höherer Ebene tolerierbar)

k-Zuverlässigkeit

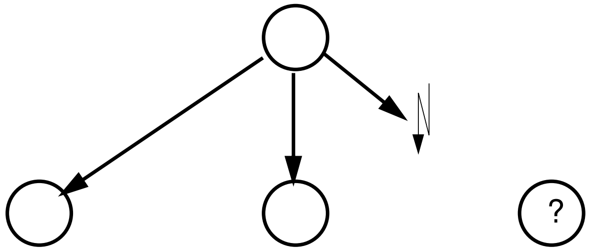
- Mindestens k Teilnehmer haben die Nachricht empfangen
  - grössere Werte von k sind “teurer”
  - Denkübungen: Wie realisiert man einen k-zuverlässigen Multicast? Ist ein “100%ig zuverlässiger” Broadcast überhaupt möglich? Wo lässt sich dies (für welches k?) sinnvoll verwenden?
- Problem der Fehlerakkumulation:
  - der Zustand (repräsentiert durch eine Variable x) sei repliziert
  - Zustandssynchronisation werde durch “function shipping” mittels 2-zuverlässigem Multicast realisiert



- Ergebnis nach einiger Zeit: Alle Replikate sind verschieden!
- in einem solchen Fall hilft also k-Zuverlässigkeit nicht viel

“Reliable Broadcast”

- Ziel: Unter gewissen Fehlermodellen einen “möglichst zuverlässigen” Broadcast-Dienst realisieren



- Quittung (“positives Acknowledgement”: ACK) für jede Einzelnachricht ist teuer
  - im Verlustfall einzeln nachliefern oder (falls broadcastfähiges Medium vorhanden) einen zweiten Broadcast-Versuch? (--> Duplikaterkennung!)

- Skizze einer anderen Idee (“negatives Ack.”: NACK):
  - alle broadcasts werden vom Sender aufsteigend numeriert
  - Empfänger stellt beim *nächsten* Empfang u.U. eine Lücke fest
  - für fehlende Nachrichten wird ein “negatives ack” (NACK) gesendet
  - Sender muss daher Kopien von Nachrichten (wie lange?) aufbewahren
  - “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: Knoten und Verbindungen mit Fail-Stop-Charakteristik

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

– **send(*N*, *s*, *sequ_num*)** an alle Nachbarn (inclusive 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)”) ;

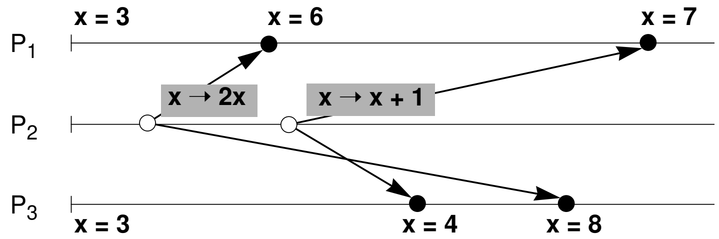
- Prinzip: “Fluten” des Netzes
  - vgl. dazu Vorlesung “Verteilte Algorithmen”

- Beachte: receive ≠ deliver
  - unterscheide Anwendungsebene und Transportebene

- Fragen:
  - müssen die Kommunikationskanäle bidirektional sein?
  - wie effizient ist das Verfahren (Anzahl der Einzelnachrichten)?
  - Optimierungen? Varianten?
  - wie fehlertolerant? (wieviel darf kaputt sein / verloren gehen...?)
  - kann man das gleiche auch ganz anders erreichen?

Empfangsreihenfolge

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

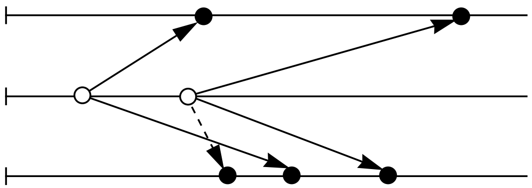


- Es sind verschiedene “Ordnungsgrade” denkbar
  - z.B. ungeordnet, FIFO, kausal geordnet, total geordnet

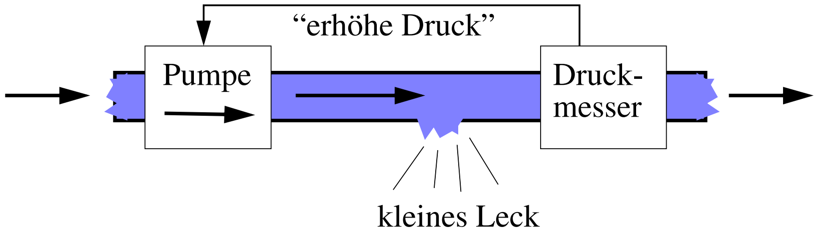
- *FIFO-Ordnung:*

  Alle Multicast-Nachrichten *eines* Senders an eine Gruppe kommen bei allen Mitgliedern der Gruppe in FIFO-Reihenfolge an

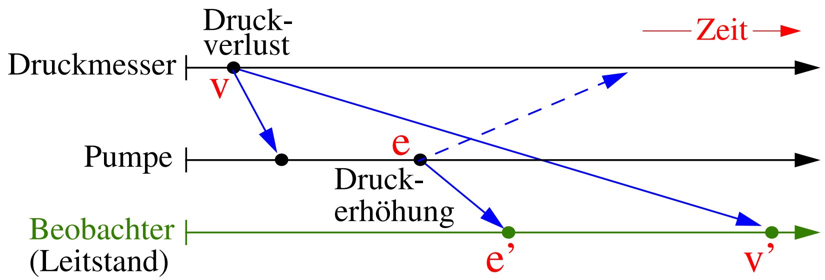
  - Denkübung: wie dies in einem Multicast-Protokoll garantieren?



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

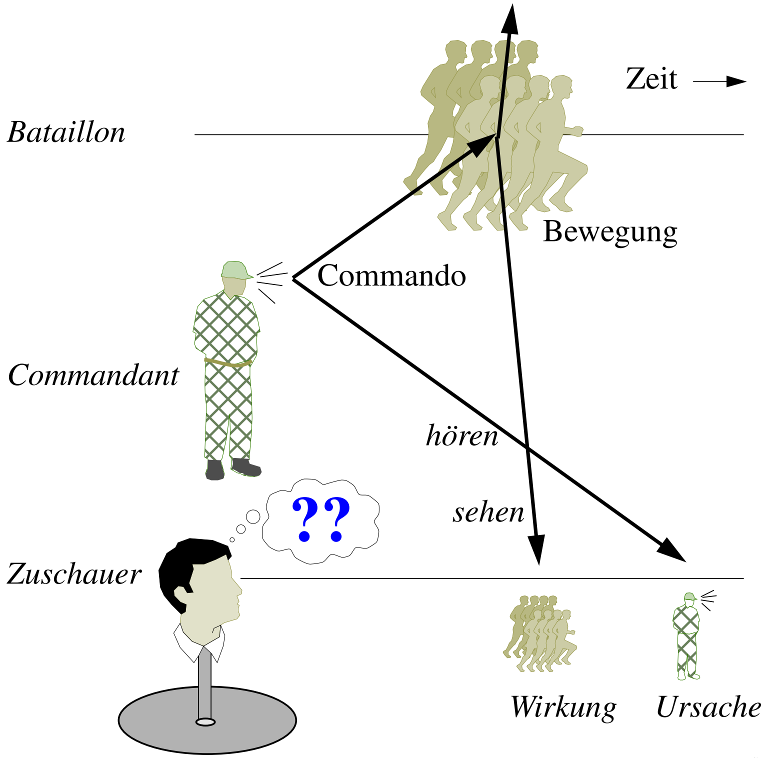
Kausalität

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.

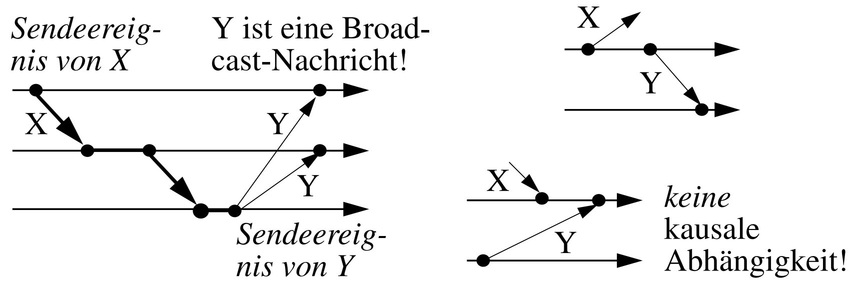
Christoph von Sigwart (1830-1904) *Logik* (1889)



Kausale Broadcasts

Kausale Abhängigkeit zwischen Nachrichten (Def.):

- *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 (--> vgl. L. Lamport und auch Vorlesung "Verteilte Algorithmen")

Wahrung von Kausalität bei der Kommunikation:

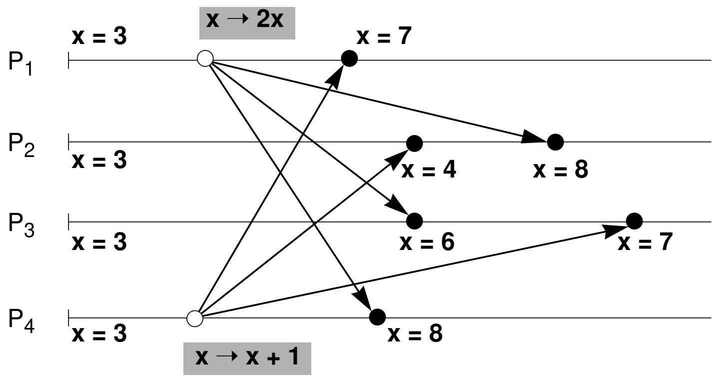
- *Kausale Reihenfolge (Def.):* Wenn eine Nachricht N kausal von einer Nachricht M abhängt, und ein Prozess P die Nachrichten N und M empfängt, dann muss er M vor N empfangen haben

Beachte: "causal order"

- "Kausale Reihenfolge" und "kausale Abhängigkeit" lassen sich insbesondere auch auf *Broadcasts* anwenden
- Kausale Reihenfolge *impliziert FIFO-Reihenfolge:* kausale Reihenfolge ist eine Art "globales FIFO" *("Dreiecksungleichung")*
- 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: Statt *überall* 7 *oder* 8 als Ergebnis: Hier “beides”!

Konkrete Ursache des Problems:

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

Abstrakte Ursache:

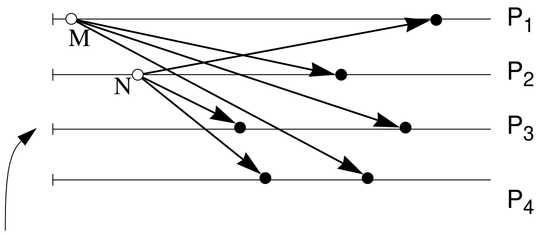
- die Nachrichtenübermittlung erfolgt (erkennbar!) *nicht atomar*

Also:

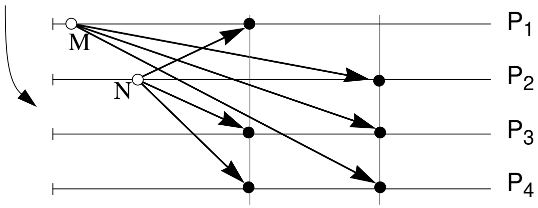
- auch kausale Broadcasts haben keine “perfekte” Semantik (d.h. Illusion einer speicherbasierten Kommunikation)

Atomarer bzw. “totaler” Broadcast

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



Äquivalent bzgl. “Gummiband-Transformation”

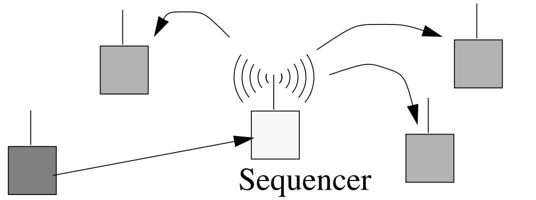


Anschaulich:

- Nachrichten eines Broadcasts werden “überall gleichzeitig” empfangen

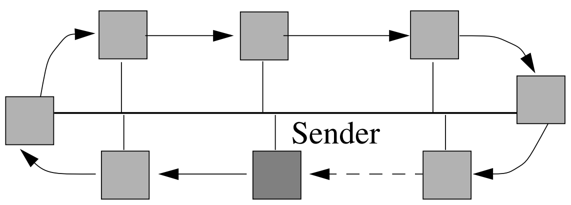
Realisierung von atomarem Broadcast

- 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 FIFO-Broadcast?)

- Token, das auf einem (logischen) Ring kreist



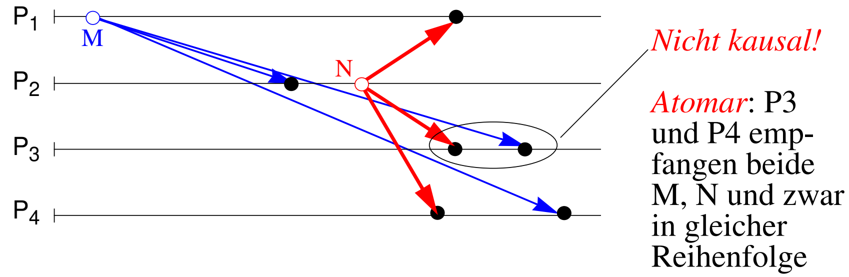
- Token = Senderecht (Token weitergeben!)
- Broadcast selbst könnte z.B. über ein zugrundeliegendes broadcastfähiges Medium erfolgen

  - 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 zu vielen Teilnehmern): Token auf Anforderung direkt zusenden (broadcast: “Token bitte zu mir”), dabei aber Fairness beachten (vgl. analoge Prinzipien bei Algorithmen für den wechselseitigen Ausschluss in Netzen --> Vorlesung “Verteilte Algorithmen”)

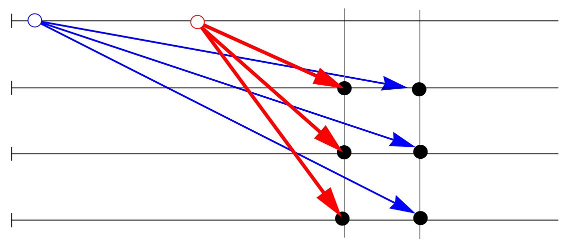
- Geht es auch ohne zentrale Elemente (Sequencer, Token)?

Wie “gut” ist atomarer Broadcast?

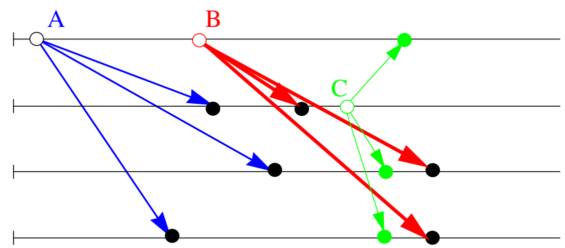
1) Ist atomar auch kausal?



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 =/=> kausale Reihenfolge
  - atomare Übermittlung =/=> FIFO-Reihenfolge
  - atomare Übermittlung + FIFO =/=> kausale Reihenfolge

- Vergleich mit speicherbasierter Kommunikation:
  - Kommunikation über gemeinsamen Speicher ist *atomar* (alle „sehen" das Geschriebene gleichzeitig)
  - 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 Real-Zeitpunkten statt, aber zur *gleichen logischen Zeit*
  - in Bezug auf die bis dahin empfangenen Nachrichten
  - vorläufig: „logische Zeit" = Menge aller vergangenen Ereignisse

aber in welchem Sinne?

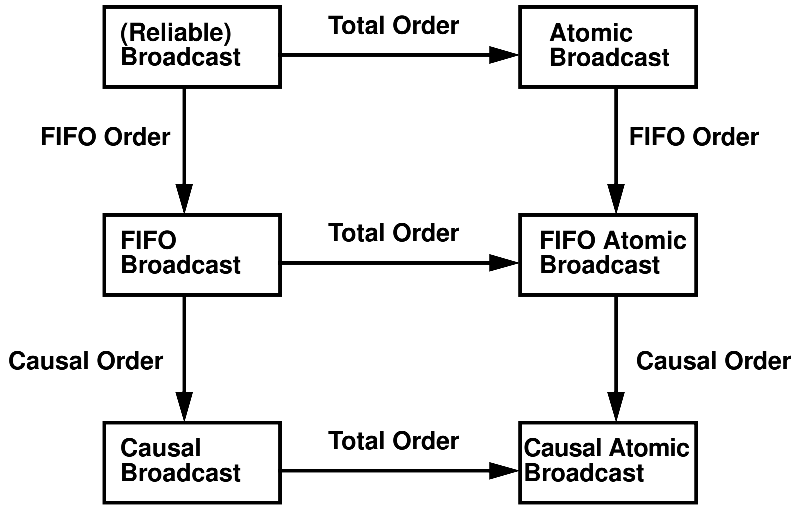
- *Innerhalb* des Systems ist synchron und virtuell synchron *nicht unterscheidbar*
  - identische totale Ordnung aller Ereignisse
  - identische Kausalbeziehungen

- Folge: Nur mit Hilfe einer globalen Uhr könnte ein externer Beobachter den Unterschied feststellen

---

Den Begriff "logische Zeit" müssen wir aber irgendwann 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
  - aufwendiger 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!

Multicast

- Definition von Multicast (grob): "*Multicast ist ein Broadcast an eine Teilmenge von Prozessen*"
- Somit: 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"
- Wesentlich: Man kann *verschiedene* Teilmengen bilden
- Neu: Begriff der *Multicast-Gruppe* = Teilmenge von Prozessen

Multicast-Gruppen

- Funktion einer Multicast-Gruppe

• “Selektiver Broadcast”

• Vereinfachung der Adressierung (z.B. statt Liste von Einzeladressen)

• Verbergen der Gruppenzusammensetzung (vgl. Mailbox/Port-Konzept)

• “Logischer Unicast”: Gruppen ersetzen Individuen (z.B. für transparente Replikation)

- Gruppenadressierung

• *Explizite Benennung:* Sender nennt den Namen der Gruppe *(“...grüsse den Kuckuckszuchtverein in Gimbelhausen”)*

• *Aufzählung der Mitglieder*: u.U. Multicast über Broadcast-Medium; gestattet dynamische Gruppen *(“...grüsse Susi, Hugo & Erni”)*

• *Prädikatadressierung*: Ein potentieller Empfänger akzeptiert die Nachricht nur, wenn ein mitgesendetes Prädikat im lokalen Zustand des Empfängers ‘wahr’ ergibt *(“...grüsse alle, die mich lieben”)*

- Offene / geschlossene Gruppen

• *Offen*: Nicht-Gruppenmitglieder dürfen Multicast-Nachrichten an die Gruppe senden

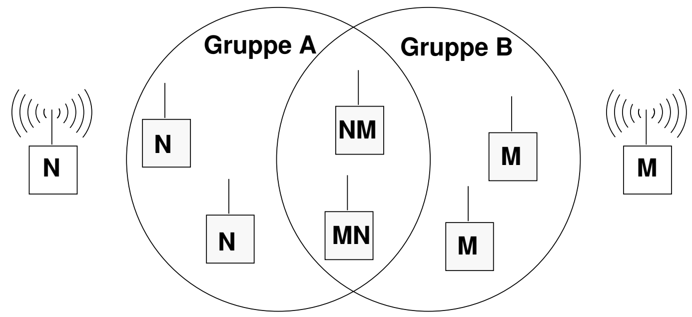
• *Geschlossen*: Nur Gruppenmitglieder dürfen...

- Statische / dynamische Gruppen

• *Dynamisch*: Gruppenzusammensetzung ändert sich ggf. im Laufe der Zeit: Gruppeneintritt, Gruppenaustritt, Ausfall eines Gruppenmitglieds; sind schwieriger zu verwalten als statische Gruppen

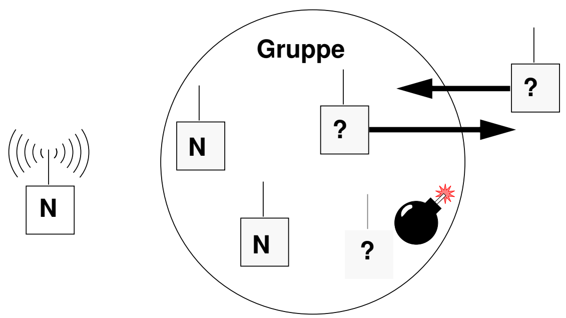
Grundprobleme bei Gruppen

• Gruppenüberlappung



- was genau geschieht im Überlappungsbereich?

• Gruppen-Management und Membership-Problem



- dynamische Gruppe: wie sieht die Gruppe “momentan” aus?
- haben alle Mitglieder (gleichzeitig?) die gleiche Sicht?
- was tun mit Crashes?

Probleme der Gruppenüberlappung am Beispiel von FIFO-Multicast

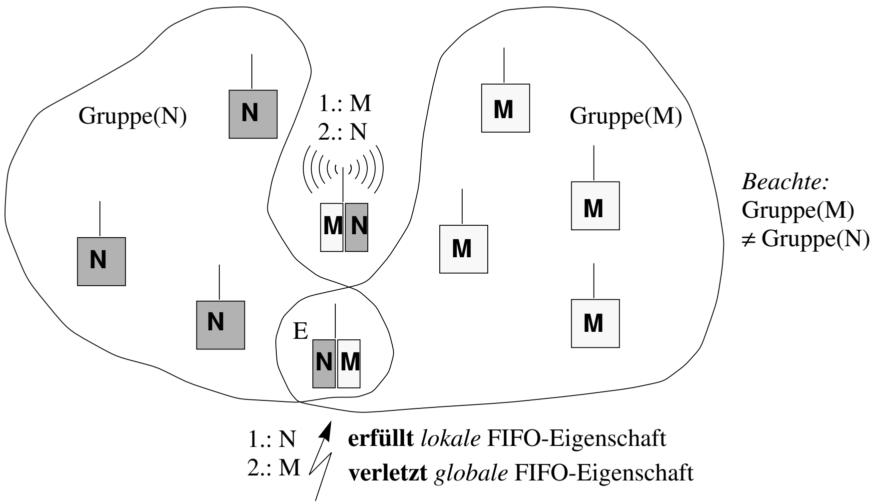
was ist *sinnvoll?*

- Auf was genau soll sich die FIFO-Eigenschaft beziehen?

- Bezeichne *Gruppe(X)* die Multicast-Gruppe, an die die Nachricht X gesendet wird

  • *Globale FIFO-Reihenfolge:* Wenn ein Prozess erst M und dann N sendet und ein Empfänger in Gruppe(M) die Nachricht N empfängt, dann muss er zuvor auch M empfangen haben

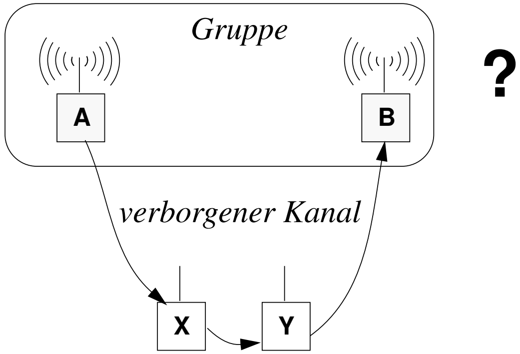
  • *Lokale FIFO-Reihenfolge:* Wenn ein Prozess erst M und dann N sendet mit Gruppe(N) = Gruppe(M) und ein Empfänger die Nachricht N empfängt, dann muss er zuvor auch M empfangen haben



- Analoge Unterscheidungen bzgl. lokaler / globaler Gültigkeit auch bei *kausalen* und *atomaren* Multicasts

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

• Wie lautet die *sinnvolle* Definition von kausalem Multicast?

Übungsbeispiel zu Multicast (1)

(Nach einer Idee von Reinhard Schwarz)

Sinnvolle Definition von atomarem Multicast?

- **Lokale Totale Ordnung:** Wenn zwei Prozesse P_1 und P_2 beide die Nachrichten M und N empfangen mit Gruppe(M) = Gruppe(N), dann empfängt P_1 M vor N genau dann, wenn P_2 die Nachricht M vor N empfängt.
- **Paarweise Totale Ordnung:** Wenn zwei Prozesse P_1 und P_2 beide die Nachrichten M und N empfangen, dann empfängt P_1 M vor N genau dann, wenn P_2 die Nachricht M vor N empfängt.

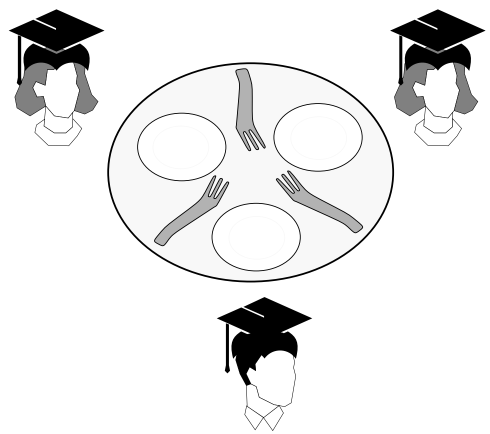
Fragen

- Wo braucht man solche Eigenschaften?
- Wann reichen die geforderten Eigenschaften, wann nicht?

Übungsbeispiel zu Multicast (2)

Beispiel: Problem der „speisenden Philosophen“

- Ein Philosoph **denkt** oder **speist**
- Zum Speisen benötigt ein Philosoph **rechte und linke Gabel.**
- Beim Denken gibt ein Philosoph beide Gabeln frei.



Wie stellt man sicher, dass die Philosophen nicht verhungern?

Übungsbeispiel zu Multicast (3)

Lösungsansatz:

Koordination der Gabelbenutzung per **paarweise atomarem Multicast**

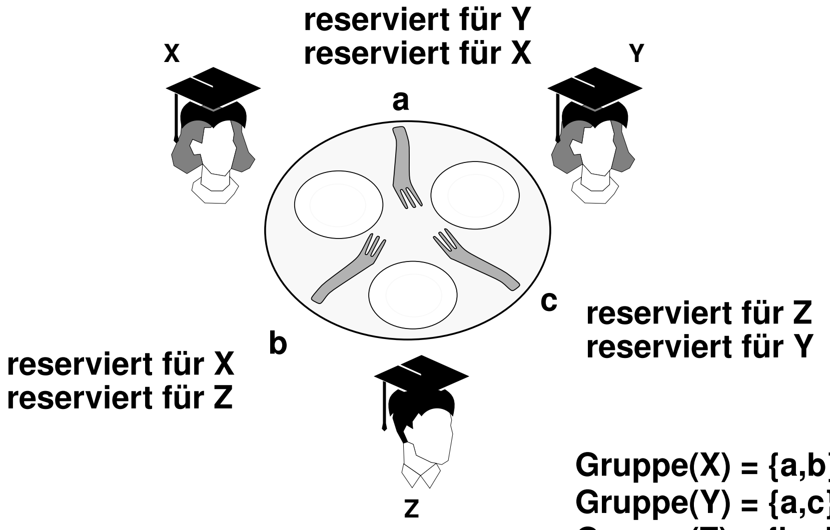
- Zum Essen sendet Philosoph X an seine benachbart liegenden Gabeln einen atomaren Multicast: *“reserviert für Philosoph X”*
- Gabel reserviert sich in der Reihenfolge der Anfragen.

→ **Durch paarweise totale Ordnung:**

- Reservierungen werden verklemmungsfrei vorgenommen, **oder?…**

Übungsbeispiel zu Multicast (4)

Typisches Szenario:



Gruppe(X) = {a,b}
Gruppe(Y) = {a,c}
Gruppe(Z) = {b,c}

→ ***Reservierungen paarweise atomar:***
Kein Paar von Nachrichten von mehr als einer Gabel empfangen

→ ***Dennoch:*** Z wartet auf X, Z wartet auf Y, X wartet auf Z …
Deadlock!

Übungsbeispiel zu Multicast (5)

Atomarer Multicast – zweiter Versuch

wie bisher:

- Lokale Totale Ordnung: Wenn zwei Prozesse P_1 und P_2 beide die Nachrichten M und N empfangen mit Gruppe(M) = Gruppe(N), dann empfängt P_1 M vor N genau dann, wenn P_2 die Nachricht M vor N empfängt.
- Paarweise Totale Ordnung: Wenn zwei Prozesse P_1 und P_2 beide die Nachrichten M und N empfangen, dann empfängt P_1 M vor N genau dann, wenn P_2 die Nachricht M vor N empfängt.

neu:

- **Globale Totale Ordnung:** Paarweise Totale Ordnung +
  Wenn eine Nachricht *M* von einem Prozess P_1 vor Nachricht *N* ausgeliefert wurde und *N* von einem Prozess P_2 vor Nachricht *O*, so liefert kein Prozess die Nachricht *O* vor *M* aus (Transitivität der Ordnungsrelation).

Membership

- Was bedeutet “alle (korrekten) Gruppenmitglieder”?

- *Beitritt* (“join”) zu einer Gruppe während einer Übermittlung
- *Austritt* (“leave”) aus einer Gruppe während einer Übermittlung
- *Wechsel zwischen “korrekt” und “fehlerhaft”* während einer Übermittlung

- Beachte:

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

- Folge:

- Zu jedem Zeitpunkt muss *Übereinstimmung* über *Gruppenzusammensetzung* und *Zustand* aller Mitglieder erzielt werden

- Frage:

- Wie erzielt man diese Übereinkunft?

Wechsel der Gruppenmitgliedschaft

"...während..." gibt es nicht
(--> "virtuell synchron")

- Forderungen:

  • Eintritt und Austritt sollen *global atomar* erfolgen:

    — Die Gruppe muss mit allen (potentiellen) Sendern an die Gruppe über den Ein- und Austrittszeitpunkt jedes Gruppenmitglieds übereinstimmen

  • *globale 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 zentraler Liste kein Problem

    — bei replizierten Listen:
    verwende *global kausalen, global 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 *global atomar* erfolgen:

    — Übereinstimmung über Reintegrationszeitpunkt

  • *Globale Kausalität* soll gewahrt bleiben

- 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 langsamer Nachrichten)

    — Fälschlich für tot erklärte Prozesse müssen 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

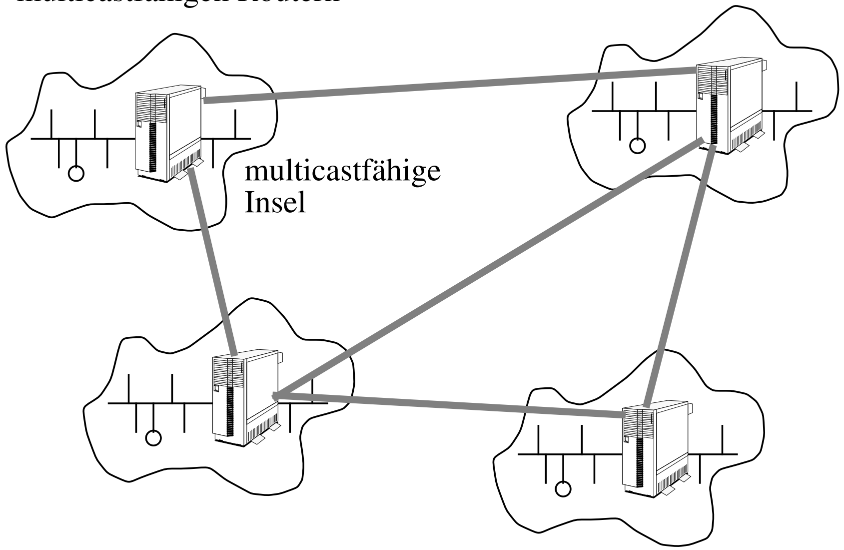
Multicast: Fazit

1. Wesentliche Neuerung gegenüber Broadcast:
   — Multicast-Gruppe
2. Gruppenüberlappungen schaffen Probleme:
   — lokale, paarweise oder globale Gültigkeit von Ordnungsbeziehungen?
3. Änderung der Gruppenmitgliedschaft ist kritisch:
   — Lösen des Membership-Problems
   — Lösen des Bootstrap-Problems
4. Das Tolerieren von Prozessausfällen ist schwierig:
   — erfordert geeignetes Fehlermodell (z.B. Fail-Stop)
   — fälschliches Toterklären nicht immer vermeidbar

- Anwendungen in der Praxis (Multi- und Broadcast):
  - gegenwärtig noch etwas zögerlich / rudimentär
  - Anwendungsgebiete: ausfallsichere Systeme, Telekom-Anwendungen, kooperatives Arbeiten...
  - es gibt Systeme, die Multicast-Kommunikationsprimitive enthalten
  - Sinnhaftigkeit und Notwendigkeit von Multicasts für grössere verteilte Anwendungen wird zunehmend erkannt
  - unsichere Multicasts im Internet zur (i.a. unidirektionalen) Verbreitung von Multimediadaten üblich (MBone)

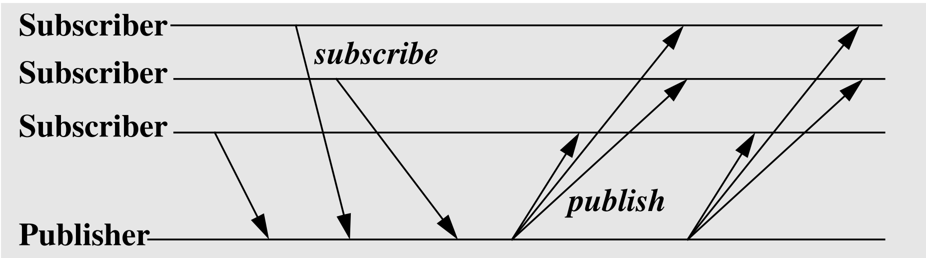
MBone (Multicast Backbone)

- Multicaststruktur im Internet zur (i.a. unidirektionalen) Verbreitung von Audio- und Videodaten
  - IP-basiert; Class-D-Multicastadressen (224.0.0.0 bis 239.255.255.255)
  - unsicher (möglicher IP-Paketverlust)
- Übertragung entlang eines Spannbaums
  - Nachrichten werden nicht mehrfach über die gleiche Strecke übertragen (nur ein Mal für alle Knoten im Unterbaum)
  - Kanten, die nicht mehr benötigt werden (z.B. bei Austritt aus der Gruppe) werden vom Spannbaum entfernt
- Router sollten multicastfähig sein
  - multicastfähige Inseln des Internet werden durch "Tunnel" miteinander verbunden
  - hierzu existiert ein MBone-Overlaynetz zwischen den multicastfähigen Routern



Push bzw. Publish & Subscribe

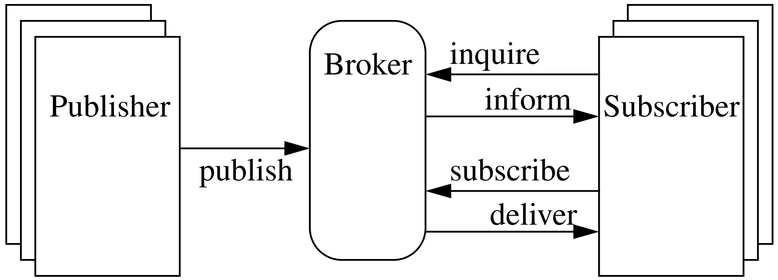
- Im Unterschied zum klassischen “Request / Reply-” bzw. “Pull-Paradigma”
  - wo Clients die gewünschte Information aktiv anfordern müssen
  - ein Client aber nicht weiss, ob bzw. wann sich eine Information geändert hat
  - dadurch periodische Nachfrage beim Server notwendig (“polling”)
- Subscriber (= Client) meldet sich für den Empfang der gewünschten Information an
  - z.B. “Abonnement” eines Informationskanals (“channel”)
  - i.a. existieren mehrere verschiedene Kanäle
  - 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”
  - Beispiel für push-Paradigma: Publikation aktueller Börsenkurse
- Für “publish” typischerweise Multicast einsetzen
  - “subscribe” entspricht offenbar einem “join” einer Multicast-Gruppe
  - Zeitverzögerung, Stärke der Multicast-Semantik und Grad an Fehlertoleranz wird oft als “Quality of Service” bezeichnet

Broker als Informationsvermittler

- Publisher und Subscriber müssen nicht direkt miteinander in Kontakt stehen
- Dazwischengeschalteter Broker verwaltet und vermittelt ggf. die Informationskanäle
  - i.a. mehrerer verschiedener Publisher
  - noch stärkere Entkopplung von Sender und Empfänger



- Subscriber erfahren vom Broker, welche Kanäle abonniert werden können
  - Broker kann auch noch andere Dienste realisieren
- Subscriber melden sich beim Broker statt beim Publisher an
- Typische Rollenverteilung:
  - Publisher als *Produzent* von Information
  - Subscriber als *Konsument*

Ereigniskanäle für Software-Komponenten

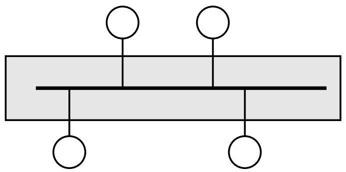
- Stark entkoppelte Kommunikation bei komponenten-basierter Software-Architektur
  - Komponenten haben getrennte Lebenszyklen

“hot pluggable” --> “spontaneous networking”

  - 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
  - ggf. Pufferung von Ereignisen



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

- Beispiele
  - Microsoft-Komponentenarchitektur DCOM / ActiveX / OLE
  - “Distributed Events” bei JavaBeans im Rahmen von Jini (event generator bzw. remote event listener; beliebige Objekte als “Parameter” möglich)
  - event service von CORBA (sprach- und plattformunabhängig.; typisierte und untypisierte Kanäle; Schnittstellen zur Administration von Kanälen; Semantik nicht genauer spezifiziert [z.B. Pufferung des Kanals])

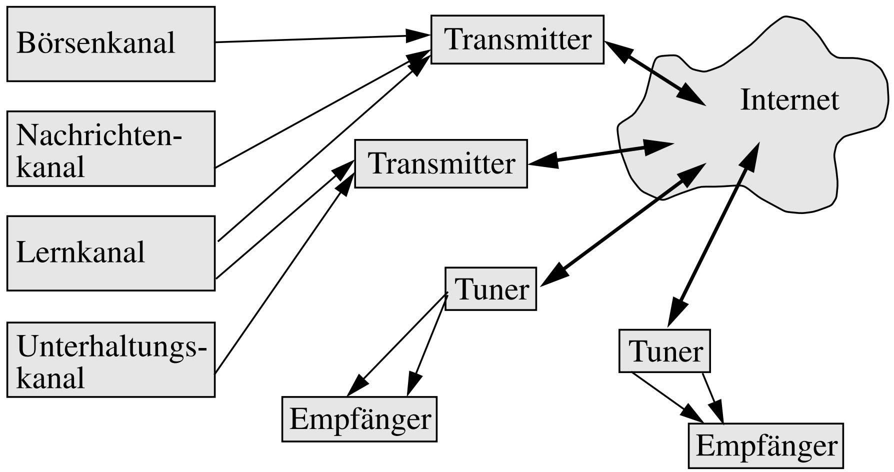
Push-Prinzip: Beispielprodukte

- “Information Bus” von TIBCO
  - publish / subscribe im Rahmen von Middleware
  - Motivation: Propagieren von “business events” (z.B. Finanzsektor)
  - Einsatzgebiete z.B. data monitoring oder real time decision support
  - “call back” einer Funktion beim Eintreffen einer Nachricht

- Pointcast (www.pointcast.com)
  - “broadcasting personalized news and information directly to your computer”

- CDF (Channel Definition Format) von Microsoft

- Castanet von Marimba (www.marimba.com)
  - ursprünglicher Zweck: “automatically distribute and maintain software applications and content within a company or across the Internet”
  - Gruppen von Kanälen; sichere (signierte, zertifizierte) Kanäle



- Identifizierung der Transmitter über eine URL
- Tuner erhält WWW-Seite vom angesprochenen Transmitter
- WWW-Seite enthält alle Kanäle eines Transmitters in Form von Links
- push wird simuliert durch periodisches polling (Subscriber kann Aktualisierungsrate festlegen)

Tupelräume

- Gemeinsam genutzter ("virtuell globaler") Speicher
- Blackboard- oder Marktplatz-Modell
  - Daten können von beliebigen Teilnehmern eingefügt, gelesen und entfernt werden
  - relativ starke Entkopplung der Teilnehmer

-Tupel = geordnete Menge typisierter Datenwerte

- Entworfen von D. Gelernter (Yale University) für die Sprache Linda (ca. 1985)
- Operationen:
  - out (t): Einfügen eines Tupels t in den Tupelraum
  - in (t): Lesen und Löschen von t im 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 Datenbankabfragesprachen (z.B. QbE)
- 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)

- Damit natürlich auch übliche 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

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

JavaSpaces

- Für Java, orientiert sich am Linda-Vorbild
  - gespeichert werden Objekte --> “Verhalten” neben den Daten
  - Tupel entspricht Gruppen von Objekten

- Teil der Jini-Infrastruktur für verteilte Java-Anwendungen
  - Kommunikation zwischen entfernten Objekten
  - Transport von Programmcode (“Verhalten”) vom Sender zum Empfänger
  - gemeinsame Nutzung von Objekten

- Operationen
  - *write*: mehrfache Anwendung erzeugt verschiedene Kopien
  - *read*
  - *readifexists*: blockiert (im Gegensatz zu read) nicht; liefert ggf. ‘null’
  - *takeifexists*
  - *notify*: Benachrichtigung (mittels eines Ereignisses), wenn ein passendes Objekt in den JavaSpace geschrieben wird

- Nutzen (neben Kommunikation)
  - atomarer Zugriff auf Objektgruppen
  - sicherer (“reliable”) verteilter Speicher
  - persistente Datenhaltung für Objekte

aber keine Festlegung, ob eine Implementierung fehlertolerant ist und einen Crash überlebt

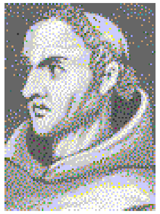
- *Implementierung*: könnte z.B. auf einer relationalen oder objektorientierten Datenbank beruhen

- *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

Occam

Programmiersprache Occam

- William of Occam
  [Wilhelm von Ockham]
  - Franziskaner; ca. 1290-1350
  - Oxford --> Paris --> München



- Prinzip der Denkökonomie ("Occam's Razor"):
  - *Entia non sunt multiplicanda praeter necessitatem*
  - Minimalprinzip: Keine unnötigen Existenzannahmen

Sprache Occam:

Communicating Sequential Processes

- Sehr spartanisch
- Aufbauend auf CSP (Hoare, 1978)
- Entwickelt ab 1982 (David May, INMOS)
- Occam --> Occam 2 --> Occam 3

(Gleitpunktzahlen, Funktionen, Datentypen,...)

Idee: - Programmieren durch kommunizierende Prozesse
- Kommunikation, Prozessverwaltung, Synchronisat. billig wie jede andere "primitive" Operation
- Programmiermodell realisiert durch spezielle Prozessoren (Transputer bzw. Netz von Transputern)

- *The Occam archive:* http://www.comlab.ox.ac.uk/archive/occam.html
- *Occam resources*: http://www.hensa.ac.uk/parallel/occam/documentation/
- *Occam reference manual*:
  http://www.hensa.ac.uk/parallel/occam/documentation/oc21refman.ps.gz

Occam: Zuweisungen

x := y

x,y,z := a,b,c   alle Ausdrücke der rechten Seite werden vor der Zuweisung ausgewertet -->

Äquivalent zu:

Schlüsselworte gross

```
SEQ
  PAR
    t1 := a    -- Das ist ein Kommentar
    t2 := b
    t3 := c
  x := t1      -- Diese 3 Zuweisungen
  y := t2      -- werden sequentiell
  z := t3      -- ausgeführt
```

Jeweils 2 Zeichen eingerückt

Jeweils auf eigene Zeile

So geht es auch:

```
SEQ
  PAR
    ... -- Zuweisung an temporäre Variablen
  PAR
    ... -- Zuw. der temp. Var. an die linke Seite
```

x, y := y, x -- vertauscht den Inhalt der Variablen!

Occam: Parallele Prozesse

```
PAR
    x := y+1
    z := y+2
```

Gesamtprozess beendet, wenn beide Einzelprozesse beendet

Die beiden *Prozesse* werden parallel ausgeführt (oder in beliebiger Reihenfolge)

Achtung: Kein paralleler Prozess darf eine Variable ändern, die in einem der anderen auftritt!

Bemerkung: Lesen gemeinsamer / globaler Variablen ist prinzipiell möglich. Aber Vorsicht, wenn die Prozesse auf verschiedenen Prozessoren ausgeführt werden (--> Kommunikation)!

In Occam gibt es also:

- anonyme dynamische Prozesse
- geschachtelte Prozesse (die allerdings nicht nebenläufig arbeiten)
- globale (und lokale) Variablen
- keine weiteren Synchronisationsmittel ausser *synchroner* Kommunikation über *Kanäle*

Occam: Kommunikation

- Kommunikation geschieht *synchron* über *Kanäle*!

```
PAR
    c ! x+y
    c ? z
```

hat die gleiche Wirkung wie z:=x+y

Kanal

$\boxed{x+y} \xrightarrow{c} \boxed{z}$

Kanäle nur zwischen je *zwei* Prozessen, wobei einer schreibt, der andere liest.

```
P1              P2

SEQ             SEQ
    ch1 ! a         ch1 ? x
    ch2 ? b         ch2 ! y
```

funktioniert

```
P1              P2

SEQ             SEQ
    ch1 ! a         ch2 ! y
    ch2 ? b         ch1 ? x
```

Deadlock!

WHILE-Schleifen in Occam

```
INT a:
SEQ
    in ? a
    WHILE  a <= 32767
        SEQ
            a := a*a
            out ! a
            in ? a
```

Deklaration

Nur ein Prozess ("SEQ") nach While

Doppelpuffer in Occam

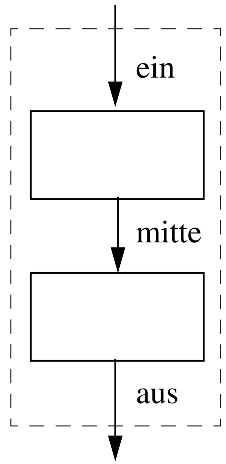
mit Kapazität 2

Pipeline-Lösung (“fall through buffer”):

Deklaration typisierter Kanäle

zwei endlos existierende parallele Prozesse

```
CHAN OF INT: ein, mitte, aus:
INT x, y:
PAR
    WHILE TRUE
        SEQ
            ein ? x
            mitte ! x
    WHILE TRUE
        SEQ
            mitte ? y
            aus ! y
```



Alternierende Puffer:

```
CHAN OF INT ein, aus:
INT x,y:
SEQ
    ein ? x
    WHILE TRUE
        SEQ
            PAR
                ein ? y
                aus ! x
            PAR
                ein ? x
                aus ! y
```

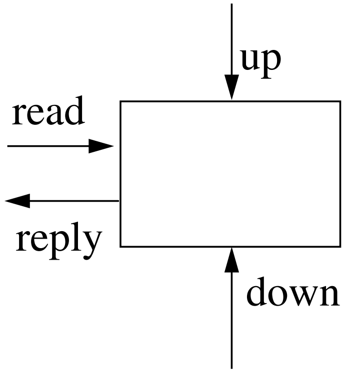
- Haben beide Lösungen die gleiche Semantik? (FIFO, keine unnötige Blockierung von Konsument oder Produzent...) Beweis?
- Vor-/ Nachteile der Implementierungen?
- Welche Realisierung ist besser?
- Lösungen skalierbar für mehr als zwei Pufferplätze?

Occam: ALT-Konstrukt

- Gleichzeitiges Warten auf Empfang bzgl. mehrerer Kanäle
- Höchstens eine der Alternativen wird gewählt

Beispiel:

```
ALT
  up ? increment
    x := x + increment
  down ? decrement
    x := x - decrement
  read ? request
    reply ! x
```

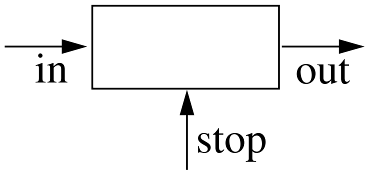


Beispiel zeigt, wie der wechselseitig ausgeschlossene Zugriff auf eine Variablen geregelt werden kann (*Monitor-Konzept*)

Oft sinnvoll: Umfassende WHILE-Schleife

```
WHILE going
  ALT
    in ? z
      out ! z
    stop ? z
      going := FALSE
```

"Ventil":



Occam: Guards

Beispiel "Stack":

Boole'scher Ausdruck

```
ALT
  (level <= max) & push ? Wert
    SEQ
      .....
  (level > 0) & pop ? req
    ....
```

- Spricht etwas dagegen, in einem ALT-Konstrukt auch ein '!' ("output guard") anstelle eines '?' ("input guard") zuzulassen?

(alternatives Empfangen / Senden, je nachdem, was gerade - abhängig von den Kommunikationspartnern - geht)

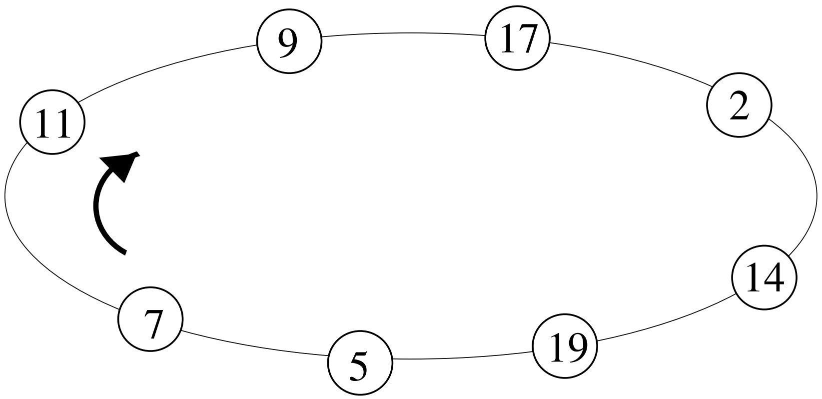
Election-Algorithmus in Occam

Dezentrale “Wahl” eines eindeutigen “Masters”

- Jeder Prozess mit Identität *p* hat lokale Variable *m*

- *m* ist initial *p*; am Ende enthält *m* die Identität des Masters

- Voraussetzung hier: Ring, wobei alle Identitäten der beteiligten Prozesse verschieden sind



- Lösung: Message-extinction-Prinzip:
  - Der Prozess mit der *kleinsten* Identität p soll gewinnen
  - Jeder Prozess gibt neue “Approximation” des globalen Minimums an Nachbarn weiter
  - Schlechtere empfangene “Approximationen” werden verschluckt

IF-Anweisung

```
IF
  a >= b
     max := a
  a <= b
     max := b
```

- betont willkürliche Auswahl im Falle a=b

- Beliebige Anzahl von Zweigen

- Erster ’true-Zweig’ wird genommen

- STOP falls kein ’true-Zweig’ gefunden

"elementarer" Prozess, der nie beendet wird

```
IF
  x < 0
     x := -x
  TRUE
     SKIP
```

’else-Fall’

Prozess, der nichts tut

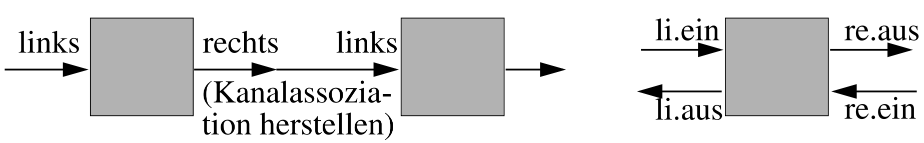
- Frage: Was ist die Semantik von:

```
WHILE TRUE
   SKIP
```

- Ist dies äquivalent zu STOP ?

Lösungsansatz Election-Algorithmus?

Unidirektionaler bzw. bidirektionaler Ring




```
...
m := p          <-- p hat in jedem Prozess einen anderen Wert
WHILE  aktiv
    SEQ
        links ? x       <-- ALT
                              li.ein ? x
                                  SKIP
                              re.ein ? x
                                  SKIP
        IF
          (x < m)
            SEQ
              m := x
              rechts ! m    <-- PAR
                                  li.aus ! m
                                  re.aus ! m
        ...
```

So geht's auch:
```
PAR
    m := x
    rechts! x
```

Führt das nicht zu einem Deadlock, wenn zwei benachbarte Prozesse sich gleichzeitig etwas gegenseitig zusenden wollen?

==> So geht es nicht, es kommt nichts in Gang! (Alle warten)

- Andere Idee: Zuerst senden!

```
m := ...
rechts ! m
WHILE  aktiv
...
```

So geht es auch nicht: Deadlock!

Frage: Ginge es bei asynchroner Kommunikation?

Das Output-Guard-Problem

zur Lösung des unidirektionalen Election-Problems

- Sollte man vielleicht "gleichzeitig" oder "alternativ" senden *und* empfangen? Wie ist es mit folgenden Ansätzen?

```
PAR
    links ? x
    rechts ! m
```

Das gefällt uns nicht:
Man kann doch nicht immer zusätzlich senden, wenn man empfangen will (bzw. umgekehrt)!
Aber: funktioniert es prinzipiell?

```
ALT
    links ? x
      ...
    rechts ! m
      ...
```

Alternatives senden und "output guards" sind in Occam verboten! Wieso?

- Problem der "gemischten Kommunikationswächter"
  - Symmetrie muss "irgendwie" automatisch gebrochen werden, und zwar in dezentraler Weise!

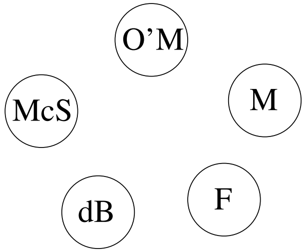
  ==> "output-guard-Problem"

- Wie wäre folgender Ansatz?

```
m := ...
PAR
    rechts ! m        <-- anfangs gleichzeitig senden und
    WHILE  aktiv      <-- den eigentlichen Prozess ausführen
        SEQ
          links ? x
          ...
```

- würden hier nicht zwei parallele Prozesse auf dem gleichen Kanal senden?
- anders aber ähnlich?

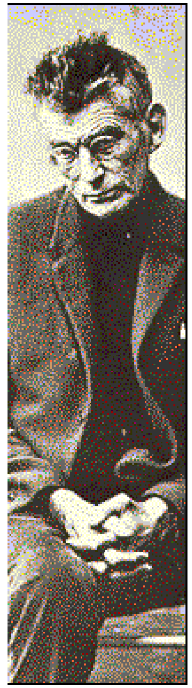
Samuel Beckett hat bereits1944 das Koordinationsproblem beim “alternativen” Senden / Empfangen beschrieben!



Samuel Beckett (1906-1989)**: Watt**

...Dann begannen sie, einander anzublicken, und es verging eine ganze Weile, ehe es ihnen gelang. Nicht, dass sie einander lange angeblickt hätten, nein, so dumm waren sie nicht. Aber wenn, falls fünf Personen einander anblicken, theoretisch dazu nur zwanzig Blicke erforderlich sind, da jeder viermal blickt, so reicht diese Zahl praktisch selten aus, wegen der vielen Blicke, die sich verirren.

Zum Beispiel, Mr. Fitzwein blickt Mr. Magershon zu seinen Rechten an. Aber Mr. Magershon blickte gerade nicht Mr. Fitzwein zu seiner Linken an, sondern Mr. O’Meldon zu seiner Rechten. Aber Mr. O’Meldon blickte nicht Mr. Magershon zu seiner Linken an, sondern, seinen Kopf nach vorn streckend, Mr. MacStern, vier Sitze links von ihm, am anderen Ende des Tisches. Aber Mr. MacStern blickte nicht, seinen Kopf nach vorn streckend, Mr. O’Meldon, vier Sitze rechts von ihm, am anderen Ende des Tisches an, sondern sitzt kerzengerade und blickt Mr. de Baker zu seiner Rechten an. Aber Mr. de Baker blickt nicht gerade Mr. MacStern zu seiner Linken an, sondern Mr. Fitzwein zu seiner Rechten...



Samuel Beckett

...Und das ist noch nicht alles. Denn viele, viele Blicke können noch geworfen werden, und viel, viel Zeit kann noch verloren gehen, ehe jedes Auge das Auge sieht, das es sucht, und in jeden Geist die Kraft, der Trost und die Zuversicht fliessen, die nötig sind, um zur Tagesordnung zurückzukehren. Und all dies aus Mangel an Methode...

...Und eine der besten Methoden... ist vielleicht die, dass Nummern an die Mitglieder des Komitees gegeben werden, eins, zwei, drei, vier, fünf, sechs, sieben, und so weiter, ebensoviele Nummern, wie es Mitglieder des Komitees gibt, so dass jedes Mitglied des Komitees seine Nummer hat und kein Mitglied des Komitees nummernlos ist...

Wenn dann der Augenblick kommt... sollen alle Mitglieder ausser Nummer eins gemeinsam Nummer eins anblicken, und Nummer eins soll sie alle der Reihe nach anblicken... Dann sollen...

.
.
.

Alternatives Senden, Output-Guards

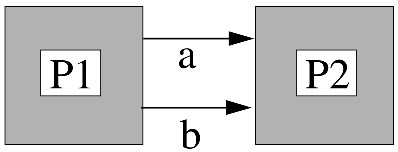
... ist in Occam nicht möglich!

Wieso? - Semantik zunächst nicht klar (Fairness, livelocks, Determinismus...)
- Implementierung schwierig / ineffizient

Es war lange unklar, ob bzw. wie man solche Programme in solche ohne diese Konstrukte transformieren kann!

- Was könnte man mit '!' in ALT noch tun?

Was bewirken folgende zwei Prozesse?



P1:

```
ALT
    a ! x
        Wert := 0
    b ! x
        Wert := 1
```

P2: (Sei W $\in$ {0,1})

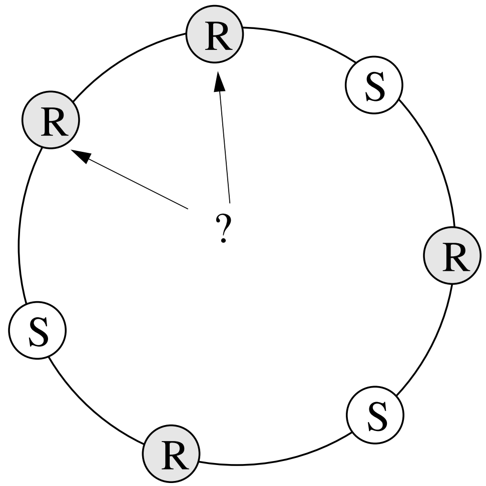
```
ALT
    W=0  &  a ? y
        SKIP
    W=1  &  b ? y
        SKIP
```

- Lässt sich iterieren! (Bitweise Übertragung)
- Geht nicht bei asynchroner Kommunikation!
- Anwendung bei Telefon mit defektem Mikrofon (P2)...
- Vorsicht mit Beweisen, dass etwas *nicht* geht!

Election - Lösungsversuch mit Occam

Idee: Zwei Typen von Prozessen
- Typ S: Versendet zunächst
- Typ R: Empfängt zunächst

Lösung ist allerdings syntaktisch nicht mehr ganz symmetrisch!



Idee: S-, R-Prozesse abwechselnd plazieren ("Schachbrettmuster")

```
m := p
links ? x
IF
(x < m)
    m := x
TRUE
    SKIP
rechts ! m
WHILE  aktiv ...
```

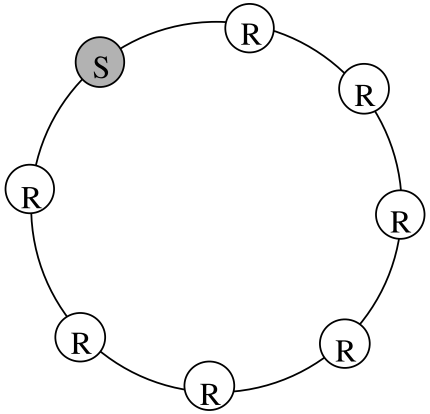
Nur bei R-Prozessen

Garantiert, dass jeder Prozess mindestens ein Mal sendet!

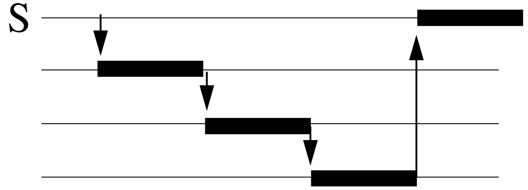
Frage: Welches zahlenmässige Verhältnis von S-/R-Prozessen ist günstig?

Variation der Anzahl der S/R-Prozesse

- Annahme: Senden am *Ende* einer Aktivitätsphase

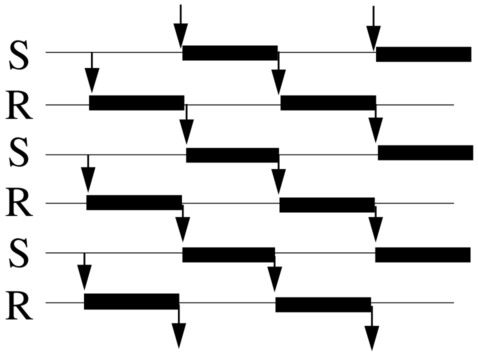


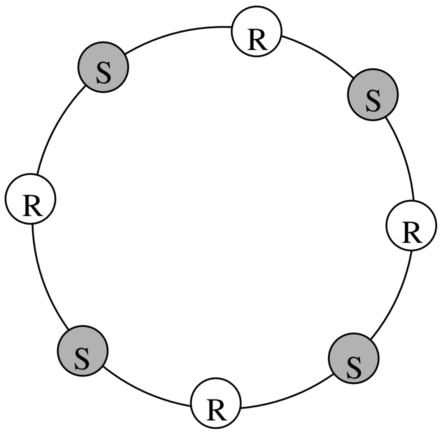
1) *Nur ein einziger Sende-Prozess*



Beobachtung: keine Parallelität (höchstens ein Prozess arbeitet)

2) *Abwechselnd S/R-Prozesse ("Schachbrettmuster")*

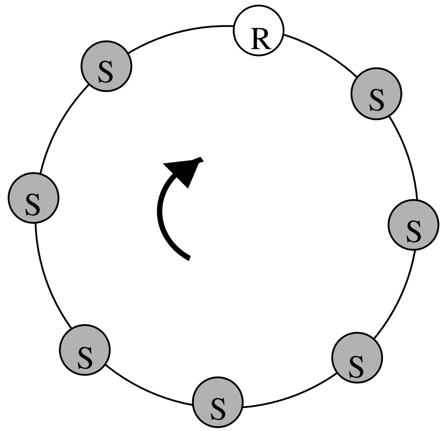




Beobachtung: etwa die Hälfte aller Prozesse ist gleichzeitig aktiv

Geht es nicht besser?

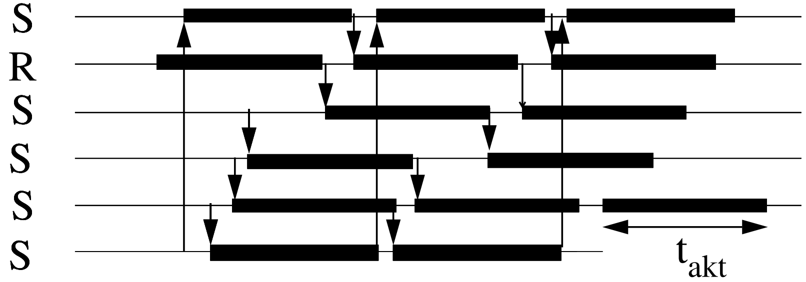
3) *Nur ein einziger R-Prozess*



Beobachtung:

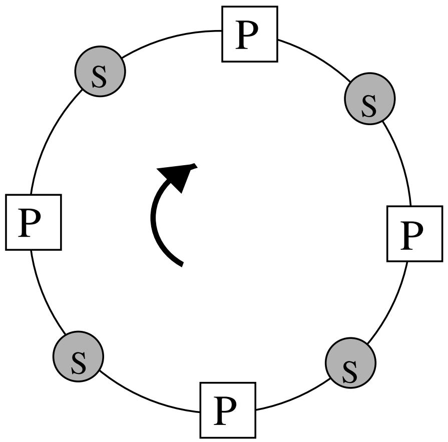
Empfangsbereitschaft läuft schnell (t_{trans}) in umgekehrter Richtung!

Bei langen Aktivitätsphasen ($t_{akt} > n\ t_{trans}$): fast immer n-1 Prozesse aktiv!



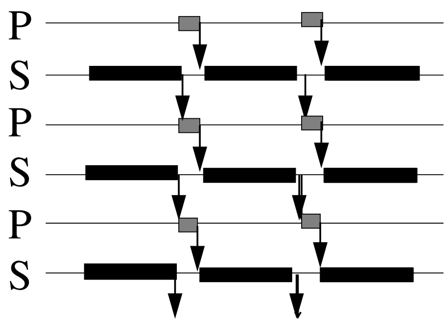
- Geht es noch besser?
  - sogar syntaktisch symmetrisch?

Election mit Puffern



- Wieviele Puffer?
  - zwischen je 2 Prozesse?
  - genügt ein einziger?
- Kapazität der Puffer?
- Overhead, da Puffer als eigene Prozesse realisiert werden müssen?

- Effekt: Simulation von *asynchroner* Kommunikation



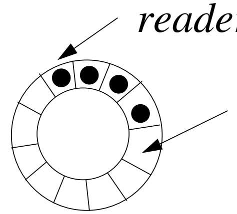
Abzüglich der Puffer-Verzögerungszeit arbeiten alle Arbeitsprozesse gleichzeitig!

- Welche Realisierung von Puffern ist hier günstig?
  (Man überdenke eine Lösung mit einem Pipeline-Puffer der Kapazität 1!)

```
WHILE TRUE
   SEQ
      input ? x
      output ! x
```

Beachte: Wir müssten bei der Lösung des Election-Problems auch noch die Terminierungserkennung einbauen! (Gewinner erhält seine eigene Zahl zurück.)

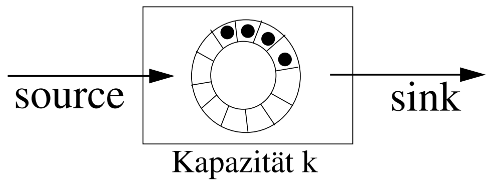
Ein Beispiel: Beschränkter zyklischer FIFO-Puffer in Occam



reader (vorderstes Element)

writer (nächster freier Platz)

count (Anzahl der freien Plätze)



Folgendes geht *nicht*:

array

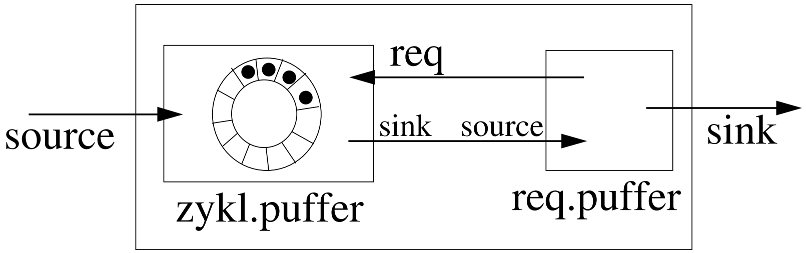
Rest bei der Division durch k

```
ALT
   count > 0  &  source ? store[writer]
      count, writer := count-1, (writer+1) \ k
   count < k  &  sink ! store[reader]
      count, reader := count+1, (reader+1) \ k
```

- Es gibt keine “output-guards” in Occam!
  - vielleicht automatisch in eines ohne output-guards transformieren?
- Lösung wie gehabt: Inversion der Kommunikation

Die Puffer-Architektur

- Idee: Inversion der Kommunikationsbeziehung für die Benutzer des Puffers transparent machen!



fifo.puffer

Prozedur

als Parameter

```
PROC  zykl.puffer (CHAN  OF  INT  source, sink, req)
   INT  reader, writer, count, x:
   [k] INT store:
   SEQ
      count, reader, writer := k, 0, 0
      WHILE  TRUE
      ALT
         count > 0  &  source ? store[writer]
            count, writer := count-1, (writer+1) \ k
         count < k  &  req ? x
            SEQ
               sink ! store[reader]
               count, reader := count+1, (reader+1) \ k
:
```

Deklaration des arrays

Variable x wird nicht verwendet

Passive Rolle ("Server")

Puffer-Architektur (2)

Aktive Rolle ("Client"):

```
PROC  req.puffer (CHAN  OF  INT  source, sink, req)
   WHILE  TRUE
      INT  x:
      SEQ
         req ! 0
         source ? x
         sink ! x
:
```

Wieso sollte man besser *nicht parallel hierzu* schon den nächsten request an den zykl. Puffer senden?

```
PROC  fifo.puffer (CHAN  OF  INT  source, sink)
   CHAN  OF  INT req, intern:
   PAR
      zykl.puffer(source, intern, req)
      req.puffer(intern, sink, req)
:
```

- Vergleich von Pipeline-Puffer und dieser Lösung?
  - Aufwand
  - Verzögerungszeit (Abh. von der Kapazität k?)
  - Durchsatz
  - Anwendbarkeit (wann blockiert der Puffer seine Anwender?)

Naive Primzahlberechnung als Beispiel verteilter Berechnungen mit Occam

Sack S:

2, 3, 4, 5, 6, ...

,..., 9999, 10000

while $\exists$ x,y $\in$ S, $\exists$ n $\in$ **N**:
n > 1 $\wedge$ y =nx
do y aus S entfernen

Nichtdeterministisch, Kontrollabstraktion

- *Implementierung*:

geschachtelte Schleife $\left\{ \begin{array}{l} \forall x: \\ \quad \forall y>x: \\ \qquad y/x \in \mathbf{N} \text{ --> y streichen} \end{array} \right.$

- *Realisierung* z.B. mittels Bit-array ("Indikatorleiste"):

2 3 ~~4~~ 5 ~~6~~ 7 ~~8~~ ~~9~~ ~~10~~ 11 ~~12~~ 13 ~~14~~ ~~15~~ ~~16~~ 17 ~~18~~ 19 ~~20~~

1 1 0 1 0 1 0 0 0 1 0 1 0 0 0 1 0 1 0

--> *Sieb des Eratosthenes*

- Aber wie dies parallel bzw. verteilt lösen?

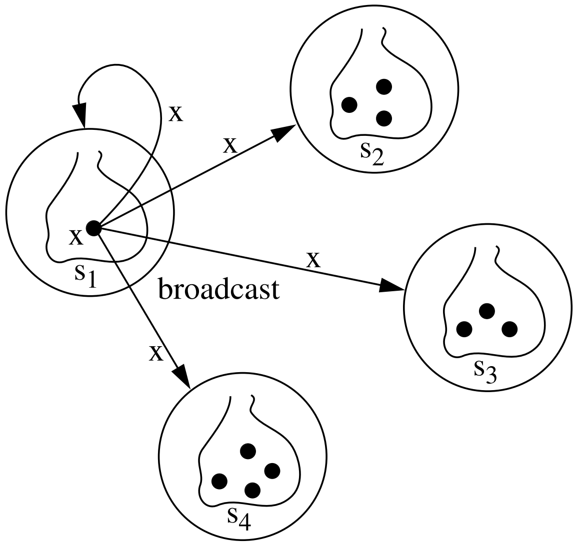
Sieb des Eratosthenes

Aus: "Nebenläufige Programme" von R. G. Herrtwich und G. Hommel (Springer-Verlag)

Primzahlberechnung verteilt?

Idee: Sack S aufteilen in kleinere Säcke s_i: $S = \bigcup s_i$

Frage: Partition? ($i \neq j \Rightarrow s_i \cap s_j = \emptyset$)



- Nach und nach: Sende alle Werte aus s_i an alle anderen
- Bei Ankunft von x: Entferne alle Vielfachen von x aus dem lokalen Sack
- Nach Ende der Berechnung ggf. alle Säcke vereinigen
- Strategie (?): Vor dem Senden von x warten, ob nicht x noch (wg. eintreffender Nachrichten) gestrichen wird
- Zunächst vielleicht so viel wie möglich lokal streichen
- Wie S aufteilen? (Extremfall: einelementige Säcke)
- Speedup möglich?
- Wie das Ende der verteilten Berechnung feststellen?

Die Primzahl-Pipeline

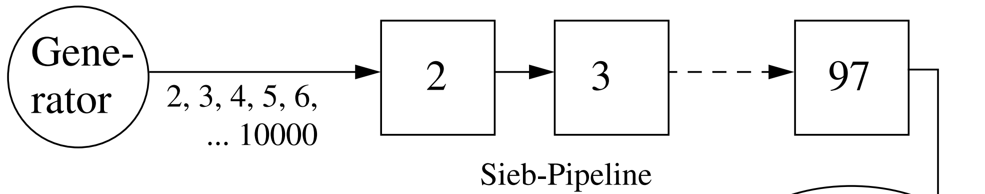
Aufgabe: Berechne alle Primzahlen bis 10000

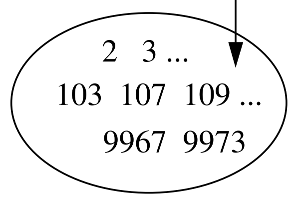
Bsp.: Ist 127 Primzahl?

1) durch 2, 3, 4, 5, 6, 7, ..., 126 teilbar ? Nein --> prim

2) durch 2, 3, 4, 5, 6,. 7, ..., 13 teilbar? ($13^2 > 127$)

3) durch 2, 3, 5, 7, 11, 13 teilbar?
(--> Primzahlen $\leq \sqrt{127}$ genügen als “Testfaktoren”!)





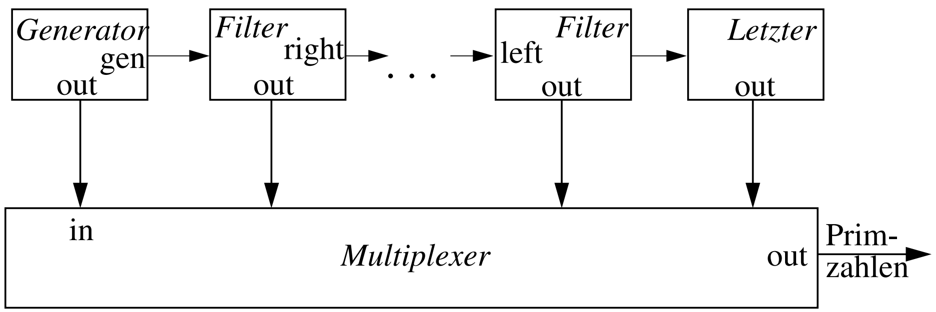
- Jedes Sieb "p" prüft, ob empfangene Zahl x durch p (ohne Rest) teilbar ist:

  Ja --> Zahl "verschlucken"

  Nein --> Zahl weiterreichen

- Pipeline ggf. dynamisch aufbauen: Hinten kommen nur Primzahlen heraus, die zur Verlängerung der Pipeline benutzt werden können

Primzahl-Pipeline




```
PROC  Generator (CHAN OF INT  gen, out)
    INT i:
    SEQ
        out ! 2
        i := 3
        WHILE  i <= 10000
            SEQ
                gen ! i
                i := i+2
        gen ! 0                      -- Schluss-Signal
:
```


Der Filter-Prozess

```
PROC  Filter (CHAN  OF  INT  left, right, out)
    INT  p, mp, x:
    SEQ
        left ? p        -- sei p <> 0
        out ! p
        mp := p*p    -- immer ungerade
        x := p
        WHILE  x<> 0
            SEQ
                left ? x
                WHILE  x > mp
                    mp := (p+p)+mp
                IF                      -- jetzt x ≤ mp
                    x < mp
                        right ! x
                    x = mp
                        SKIP            -- verschlucken
:
```

Beachte: Es finden keine (zeitaufwendigen) Probedivisionen oder Multiplikationen statt!

Wird das Schlusssignal ‘0’ durch alle Siebe propagiert?

Mit Puffer:

Ohne Puffer könnten Prozesse vorne in der Pipeline durch weiter hinten liegende gebremst werden!

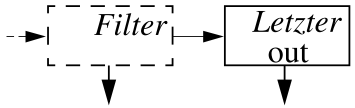
```
PAR
    IF
        ... ! ...
    left ? y
x := y
```

gleichzeitig schreiben und lesen

... sowie einige weitere offensichtliche Anpassungen

Die beiden restlichen Prozesse

```
PROC Letzter (CHAN OF INT left, out)
  INT x:
  SEQ
    x := 1
    WHILE x <> 0
      SEQ
        left ? x
        out ! x
```




```
PROC Multiplexer ([]CHAN OF INT in,
                  CHAN OF INT out)
INT p, i:
SEQ
  i := 0
  WHILE i <= Anzahl.Filter + 1
    SEQ
      in[i] ? p
      out ! p
      i := i+1
  WHILE p <> 0
    SEQ
      in[Anzahl.Filter + 1] ? p
      out ! p
```

Es müssen nun die Prozesse noch gegründet und konfiguriert werden (d.h. über die richtigen Kanäle miteinander verbunden werden und auf die vorhandenen Prozessoren geeignet verteilt werden; ggf. unter Beachtung von Lastausgleich).

Denkübung: Wie sollten 100 Filter auf 7 Prozessoren verteilt werden?

Übung (lehrreich!!): Dies z.B. in Java oder C realisieren, verteilt auf einigen Rechnern implementieren und den Speedup maximieren!

Namens-verwaltung

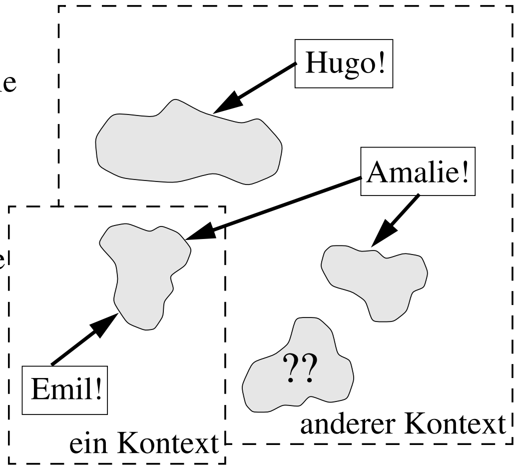
Namen und Namensverwaltung

Namen sind Schall und Rauch

Nomen est omen

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

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



- Beispiele für bezeichnete Objekte
  - in Programmiersprachen:

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

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

Zweck von Namen

Typ, Gestalt, Zweck...

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

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

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

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

Namen und Adressen

- Jedes Objekt hat eine Adresse
  - Speicherplatzadressen
  - Internetadressen
  - Ethernetadressen
  - Port-Nummer bei TCP
  - ...
- Adressen sind “physische” Namen ← Namen der untersten Stufe
- Adressen ermöglichen die *direkte Lokalisierung* eines Objektes
- Adressen sind ebenfalls innerhalb eines Kontextes (“Adressraum”) eindeutig

- Adresse eines Objektes ist u.U. *zeitabhängig*
  - mobile Objekte
  - “relocatable”
- *Dagegen*: Name eines Objektes ändert sich i.a. nicht
  - vgl. aber: Namensänderung bei Heirat, Zuweisung eines Alias...!

- Entkoppelung von Namen und Adressen unterstützt die *Ortstransparenz*
- Zuordnung Name --> Adresse nötig
  - vgl. persönliches Adressbuch
  - “Binden” eines Namens an eine Adresse

Binden

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

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

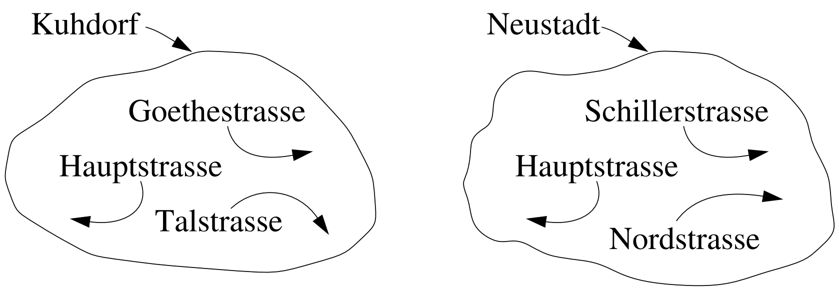
- *Binden von Dienstaufrufen bei klassischen Systemen*
  - Dienstaufruf durch Trap / Supervisor-Call (“SVC”)
    - Name = SVC-Nummer (oder “symbolische” Bezeichnung)
  - Bei *Systemstart* wird eine Verweistabelle angelegt
    - “SVC table”, “switch vector”
  - Dienstadresse ändert sich bis zum reboot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

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

Namenskontext

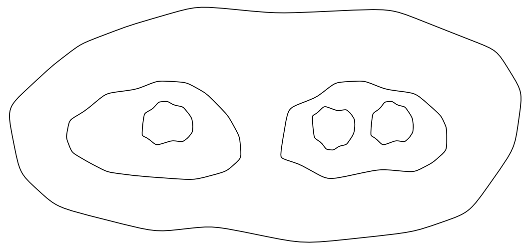
Namensraum

- Namen werden relativ zu einem *Kontext* interpretiert
  - "relative Namen" (gleiche Namen in verschiedenen Kontexten möglich)
  - *Interpretation* = Abbildung auf die gebundene Adresse oder einen Namen niedrigerer Stufe
  - Interpretation erfolgt oft mehrstufig, z.B.: Dateiname --> Adresse des Kontrollblocks --> Spur / Sektor auf einer Platte
- Namen sollen innerhalb eines Kontextes eindeutig sein
  - bzw. durch zusätzliche Attribute eindeutig identifizierbar sein
- Falls nur ein einziger Kontext existiert: *flacher Namensraum* (aus "absoluten Namen")
  - Partition des Namensraum wird als "Domäne" bezeichnet
- Namenskontexte sind (i.a. abstrakte) Objekte, die selbst wieder einen Namen haben können
  - z.B. benannte Dateiverzeichnisse ("directory")
  - übergeordneter Kontext --> *Hierarchie*



Hierarchische Namensräume

- Baumförmige Struktur von Namenskontexten

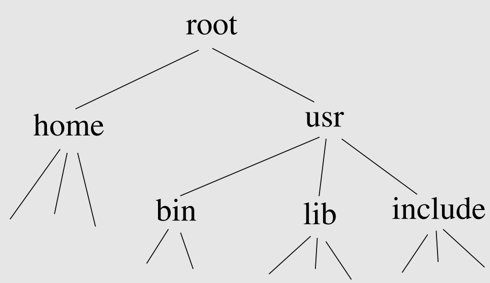


- Beispiel: Adressen im Briefverkehr
  - "Hans Meier, Deutschland" genügt nicht...

- Beispiel: Telefonsystem
  - Landeskennung
  - Ortsnetzkennung
  - Teilnehmerkennung

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

- Beispiel: UNIX-Dateisystem



Hierarchische Namensräume (2)

- Eignen sich gut für verteilte Systeme
  - besser als flache Namensräume
  - leichter skalierbar (z.B. zur Gewährleistung der Eindeutigkeit)
  - dezentrale Verwaltung der Kontexte durch eigenständige Autoritäten, die wieder anderen Autoritäten untergeordnet sind
  - Namensinterpretation stufenweise durch verteilte Instanzen
  - erleichtert Systemrekonfiguration
  - eindeutige absolute Namen durch Angabe des ganzen Pfades
- Strukturierte Namen
  - bestehen aus mehreren Komponenten
  - Komponenten bezeichnen typischerweise Kontexte
  - Bsp: root/usr/bin
  - Bsp: Meier.Talweg 2.Kuhdorf.Oberpfalz.Deutschland
  - Bsp: +49 08977 32168 (präfixfreier Code!)
  - oft geographisch oder thematisch gegliedert

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

Namensverwaltung (“name service”)

- Verwaltung der Zuordnung Name --> Adresse
  - Eintragen: “bind (Name, Adresse)” sowie Ändern, Löschen etc.
  - Eindeutigkeit von Namen garantieren
  - Zusätzlich ggf. Verwaltung von Attributen der bezeichneten Objekte
- Auskünfte (“Finden” von Ressourcen)
  - z.B. Adresse zu einem Namen (“resolve”: Namensauflösung)
  - z.B. alle Dienste mit gewissen Attributen (etwa: alle Postscript-Drucker) “yellow pages” <--> “white pages”
- Ggf. Schutz- und Sicherheitsaspekte
  - Capability-Listen, Schutzbits, Autorisierungen...
  - Dienst selbst soll hochverfügbar und sicher (z.B. bzgl. Authentizität) sein
- Ggf. Generierung eindeutiger Namen
  - innerhalb eines Kontextes (z.B. mit Zeitstempel oder lfd. Nummer)
  - bzw. global eindeutig (z.B. eindeutigen Kontextnamen als Präfix vor knotenlokale laufende Nummer; ggf. auch lange Zufallsbitfolge)

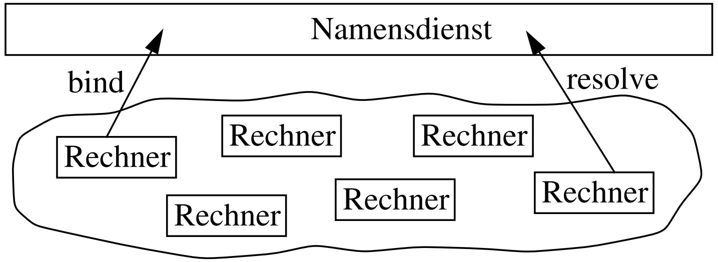
Vgl. “klassische” Dienste beim Telefonsystem:

- Telefonbuch
- Auskunft

} Abbildung Name --> Telefonnummer

- ggf. mehrstufig / dezentral: Auslandsauskunft wendet sich an Ortsauskunft im Ausland... (--> hierarchische Namenskontexte notwendig!)
- lokale Telefonbücher sowie Ortsvorwahlverzeichnis sind repliziert
  - sonst Überlastung des zentralen Dienstes
  - Problem der verzögerten Aktualisierung (veraltete Information)
- “gelbe Seiten”: Suche nach Dienst über Attribute

Verteilte Namensverwaltung



logisch ein einziger Dienst; tatsächlich verteilt realisiert

- Jeder Kontext wird (logisch) von einem autonomen *Nameserver* verwaltet
    - ggf. ist ein Nameserver für mehrere Kontexte zuständig
    - ggf. Aufteilung / Replikation des Nameservers --> höhere Effizienz, Ausfallsicherheit

- Typischerweise *hierarchische Namensräume*
    - entsprechend strukturierte Namen
    - entsprechend kanonische Aufteilung der Verwaltungsaufgaben
    - Zusammenfassung Namen gleichen Präfixes vereinfacht Verwaltung

- Typisch: *kooperierende* Nameserver, die den gesamten Verwaltungsdienst realisieren
    - hierzu geeignete Architektur der Server vorsehen
    - Protokoll zwischen den Nameservern (für Fehlertoleranz, update der Replikate etc.)
    - Dienstschnittstelle wird i.a. durch lokale Nameserver realisiert

bzw. “user agent”

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

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