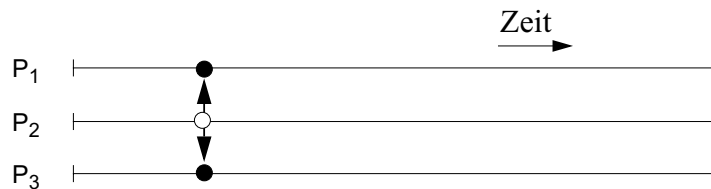


# Gruppenkommunikation - idealisierte Semantik

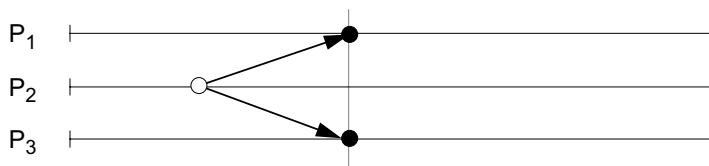
## - Modellhaftes Vorbild: Speicherbasierte Kommunikation in zentralistischen Systemen

- augenblicklicher Empfang
- vollständige Zuverlässigkeit (kein Nachrichtenverlust etc.)



## - Nachrichtenbasierte Kommunikation: Idealisierte Sicht

- (verzögerter) gleichzeitiger Empfang
- vollständige Zuverlässigkeit



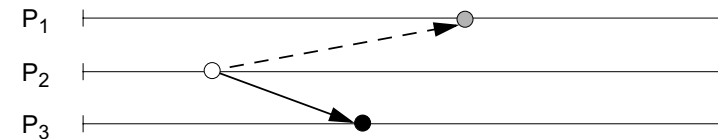
# Gruppenkommunikation - tatsächliche Situation

## - Medium (Netz) ist oft nicht multicastfähig

- LANs teilweise (z.B. klassisches Ethernet), jedoch i.a. nur innerhalb von Teilstrukturen; WLAN als Funknetz
- 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 ernsteres Problem als beim "Unicast"! (Wieso?)

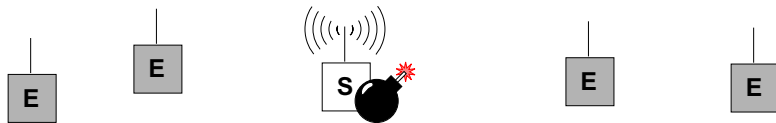
- Hauptproblem bei der Realisierung von Broadcast:  
(1) Zuverlässigkeit und (2) garantierte Empfangsreihenfolge

# Senderausfall beim Broadcast

Wenn Broadcast implementiert ist durch Senden vieler Einzelnachrichten

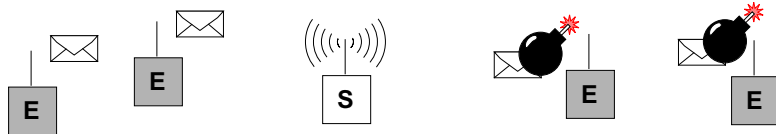
a) Sender fällt ganz aus: *kein* Empfänger erhält Nachricht

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



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

- “ungünstiger” Fall: *Uneinigkeit*



- mögliche Abhilfe: Empfänger leiten die Nachricht 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!)

# Zuverlässigkeitsgrad “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. Informationen, die ggf. Einfluss auf die Effizienz haben, nicht aber die Korrektheit betreffen

- Ggf. als Grundlage zur Realisierung höherer Protokolle

- oft basierend auf der A-priori-Broadcastfähigkeit von Netzen

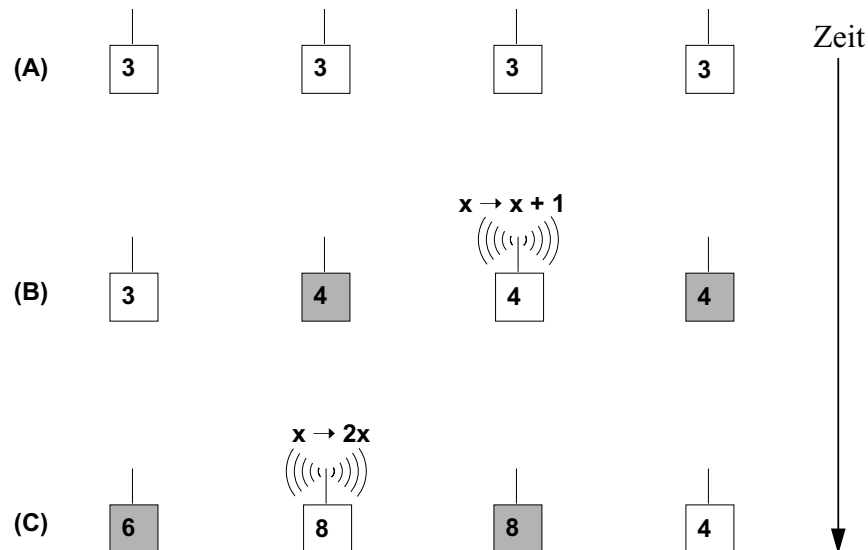
- günstig bei zuverlässigen physische Kommunikationsmedien (wenn Fehlerfall sehr selten → aufwendiges Recovery auf höherer Ebene tolerierbar)

# 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:

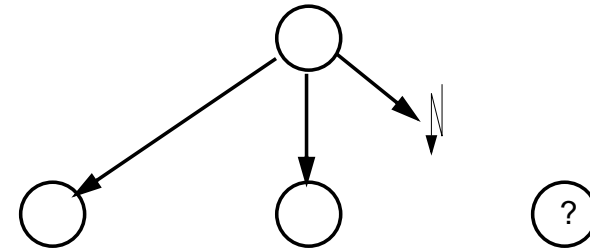
- der Zustand (repräsentiert durch eine Variable x) sei repliziert
- Zustandssynchronisation werde durch "function shipping" mittels 2-zuverlässigem Broadcast realisiert



- 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 nummeriert
  - 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 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: Knoten und Verbindungen mit Fail-Stop-Charakteristik

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

- **send( $N, s, sequ\_num$ )** an alle Nachbarn (inklusive an  $s$  selber);
- $sequ\_num++$

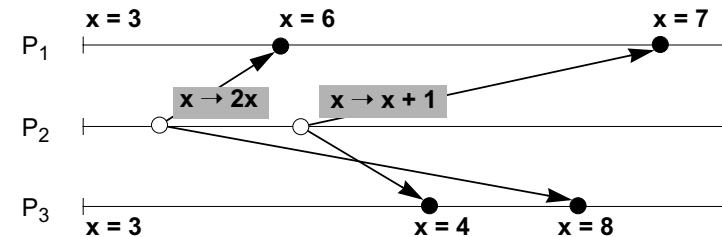
**Empfänger  $r$ :** Realisierung des Nachrichtenempfangs

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

# Broadcast: Empfangsreihenfolge

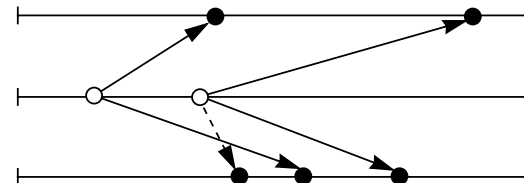
- Wie ist die Empfangsreihenfolge von Nachrichten?
  - problematisch wegen der i.a. ungleichen Übermittlungszeiten
  - Bsp.: Update einer replizierten Variablen mittels "function shipping":



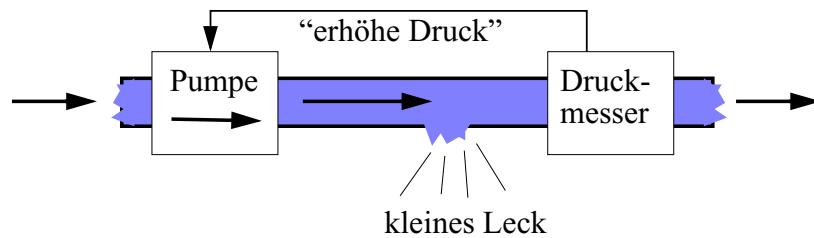
- Es sind verschiedene "Ordnungsgrade" denkbar
  - z.B. ungeordnet, FIFO, kausal geordnet, total geordnet
- **FIFO-Ordnung:**

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

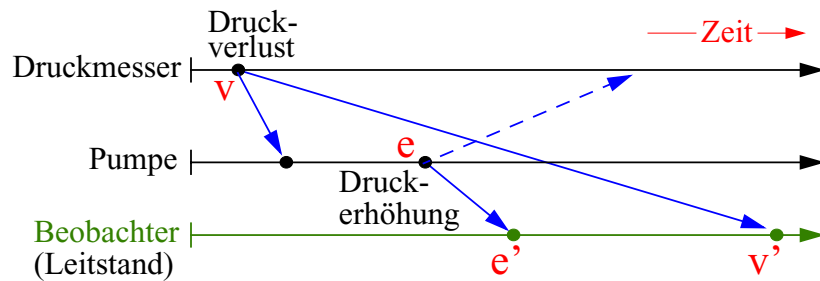
- Denkübung: wie dies in einem Multicast-Protokoll garantieren?



# Probleme mit FIFO-Broadcasts



- Annahme: Steuerelemente kommunizieren über FIFO-Broadcasts:



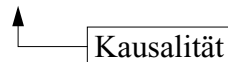
- "Irgendwie" kommt beim Beobachter die Reihenfolge durcheinander!

==> *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

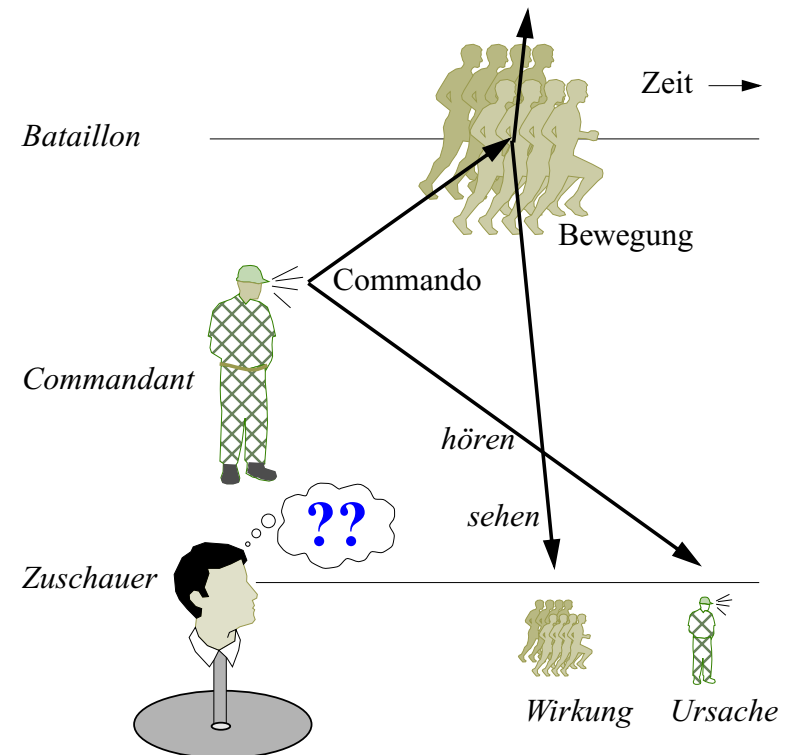


# Das "Broadcastproblem" ist nicht neu

- 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, **eh**e er die Commandostimme oder das Hornsignal **hört**; aber aus seiner Kenntnis der **Causalzusammenhänge** weiß er, daß die Bewegungen die **Wirkung** des gehörten Commands 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.

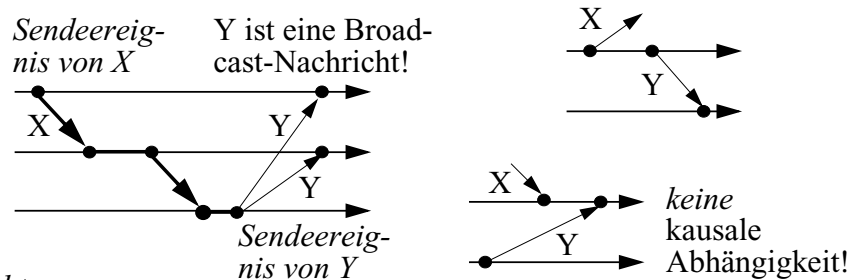
Christoph von Sigwart (1830-1904) *Logik* (1889)



# 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 Sendereignis von X zum Sendereignis von Y führt



Beachte:

- Dies lässt sich bei geeigneter Modellierung auch abstrakter fassen (→ vgl. logische Zeit später in der Vorlesung 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

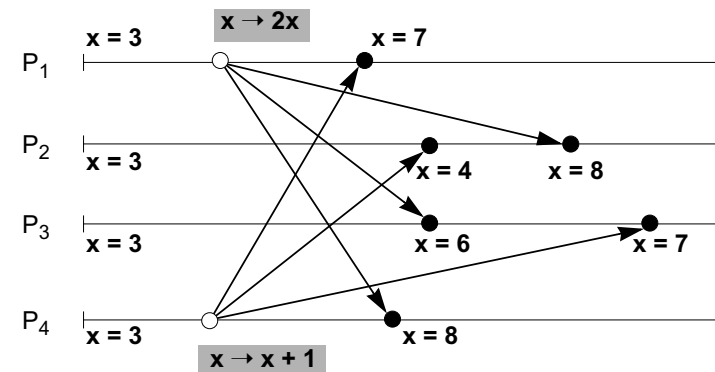
Beachte:

“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”
- 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:

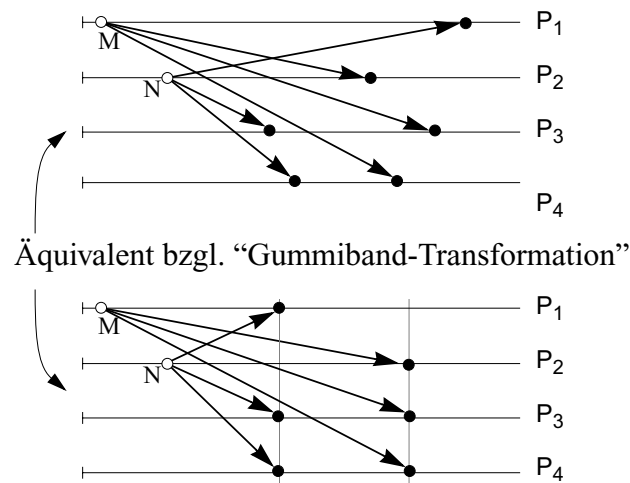
- die Nachrichtenübermittlung erfolgt (erkennbar!) *nicht atomar*

Also:

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

# Atomarer bzw. "totaler" Broadcast

- *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!)



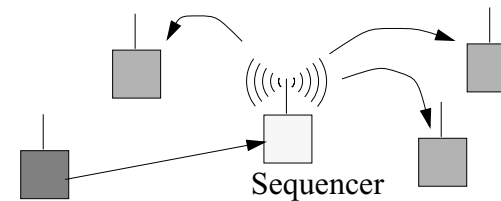
Anschaulich:

- Nachrichten eines Broadcasts werden "überall gleichzeitig" empfangen

# Realisierung von atomarem Broadcast

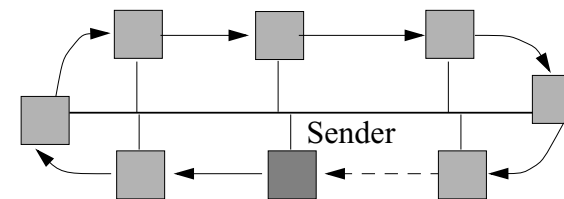
## 1) Zentraler „Sequencer“, der Reihenfolge festlegt

- ist allerdings ein potentieller Engpass!



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

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



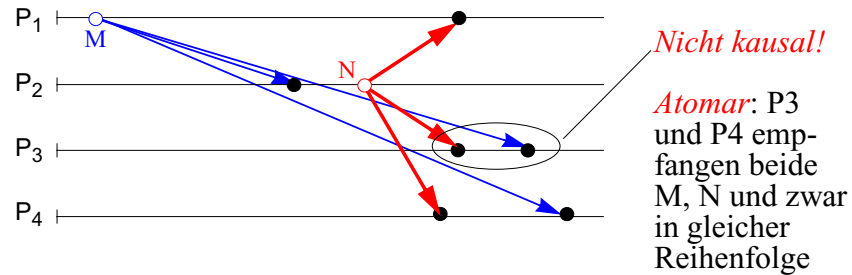
- Token = Senderecht (Token weitergeben!)
- Broadcast selbst könnte z.B. über ein zugrundeliegendes broadcast-fähiges Medium erfolgen

- 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")

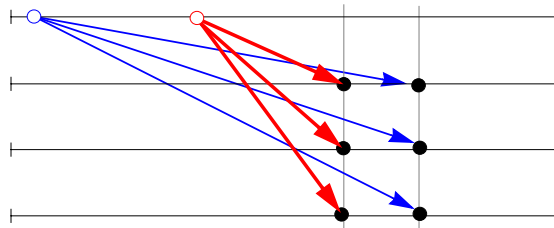
- Geht es auch ohne zentrale Elemente (Sequencer, Token)?

# Wie “gut” ist atomarer Broadcast?

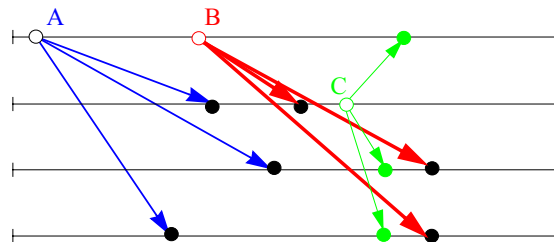
1) Ist **atomar** auch **kausal**?



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  $\neq$  kausale Reihenfolge
- atomare Übermittlung  $\neq$  FIFO-Reihenfolge
- atomare Übermittlung + FIFO  $\neq$  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 Realzeitpunkten 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 (im Sinne von „gleichzeitig“) und virtuell synchron *nicht unterscheidbar*

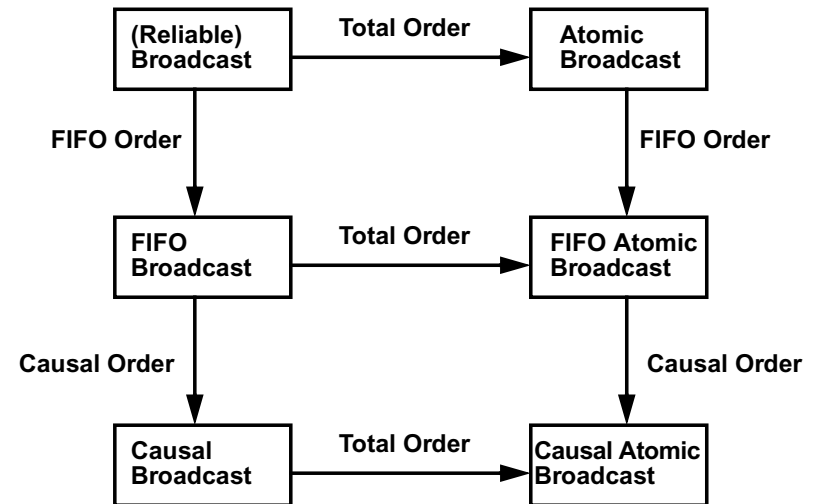
- identische totale Ordnung aller Ereignisse
- identische Kausalbeziehungen

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

---

Den Begriff „logische Zeit“ 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
- 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!

==> grössere Vielfalt ==> komplexer bzgl. Verständnis und Anwendung

# Multicast

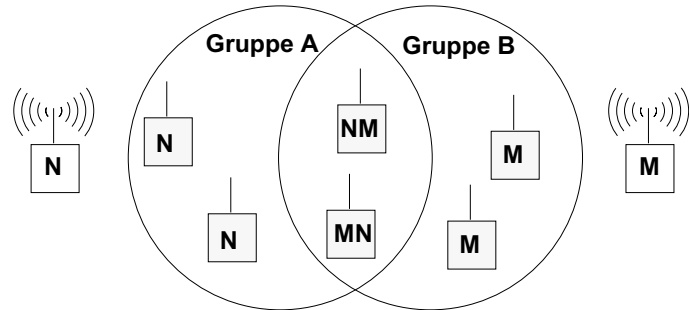
- Definition von Multicast (informell): “*Multicast ist ein Broadcast an eine Teilmenge von Prozessen*”
- Daher: Alles, was bisher über Broadcast gesagt wurde, gilt (innerhalb der Teilmenge) auch weiterhin:
  - zuverlässiger Multicast
  - FIFO-Multicast
  - kausaler Multicast
  - atomarer Multicast
  - kausaler atomarer Multicast
- Unterschied: Wo bisher “alle Prozesse” gesagt wurde, gilt nun “alle Prozesse innerhalb der Teilmenge”
- 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*: evtl. 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

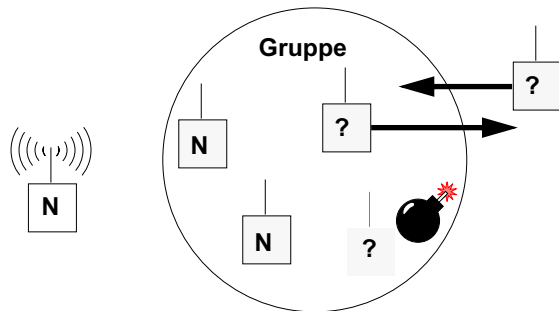
# Grundprobleme bei Gruppen

- Gruppenüberlappung



- was genau geschieht im Überlappungsbereich?

- Gruppen-Management und Membership-Problem



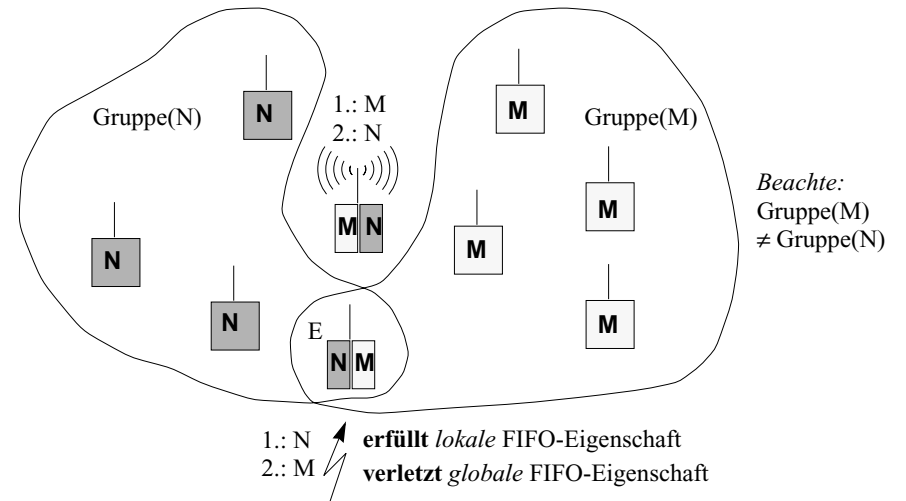
- dynamische Gruppe: wie sieht die Gruppe "momentan" aus?
- haben alle Mitglieder (gleichzeitig?) die gleiche Sicht?
- was tun bei Crashes?

# 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

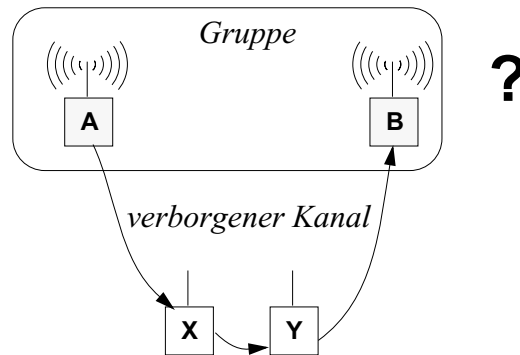
- *Globale FIFO-Reihenfolge*: Wenn ein Prozess erst M und dann N sendet und ein Empfänger in  $Gruppe(M)$  die Nachricht N empfängt, dann muss er zuvor auch M empfangen haben
- *Lokale FIFO-Reihenfolge*: Wenn ein Prozess erst M und dann N sendet mit  $Gruppe(N) = Gruppe(M)$  und ein Empfänger die Nachricht 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  $\text{Gruppe}(M) = \text{Gruppe}(N)$ , 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.

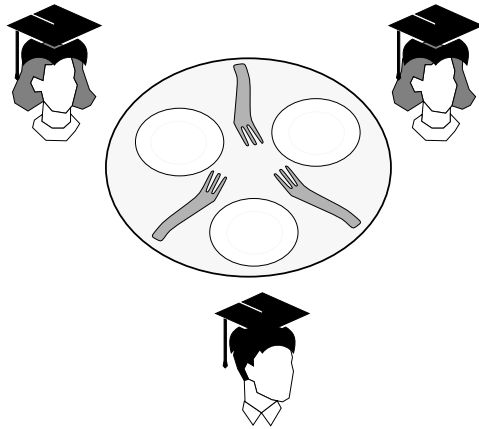
## Fragen

- 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**

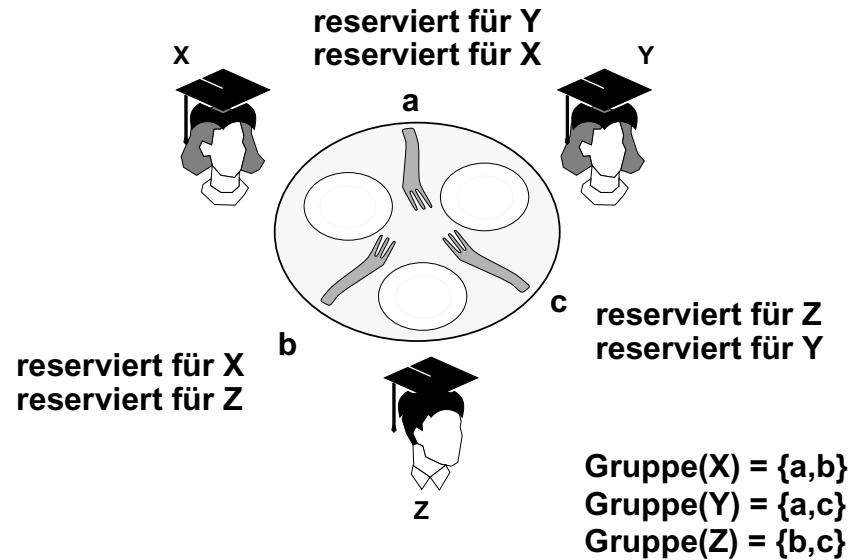
- Zum Essen sendet Philosoph X an seine benachbart liegenden Gabeln einen atomaren Multicast: *„reserviert für Philosoph X“*
- Gabel reserviert sich in der Reihenfolge der Anfragen.

→ **Durch paarweise totale Ordnung:**

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

# Übungsbeispiel zu Multicast (4)

Typisches Szenario:



- **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  $\text{Gruppe}(M) = \text{Gruppe}(N)$ , 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 Gruppenmitglieder”?

- *Beitritt* (“join”) zu einer Gruppe während einer Übermittlung?
- *Austritt* (“leave”) aus einer Gruppe während einer Übermittlung?
- *Wechsel* “korrekt” nach “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* (“korrekt”, “ausgefallen” etc.) aller Mitglieder erzielt werden

## - Frage:

- Wie erzielt man diese Übereinkunft?

# Wechsel der Gruppenmitgliedschaft

## - Forderungen:

“...während...” gibt es nicht  
(→ “virtuell synchron”)

- Eintritt und Austritt sollen *global atomar* erfolgen:
  - Die Gruppe muss bei allen (potentiellen) Sendern an die Gruppe hinsichtlich der Ein- und Austrittszeitpunkte 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 suspects”* 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:

- z.B. lokale oder globale Gültigkeit von Ordnungsbeziehungen?

## 3. Änderung der Gruppenmitgliedschaft ist kritisch:

- Lösen des Membership-Problems
- Lösen des Bootstrap-Problems

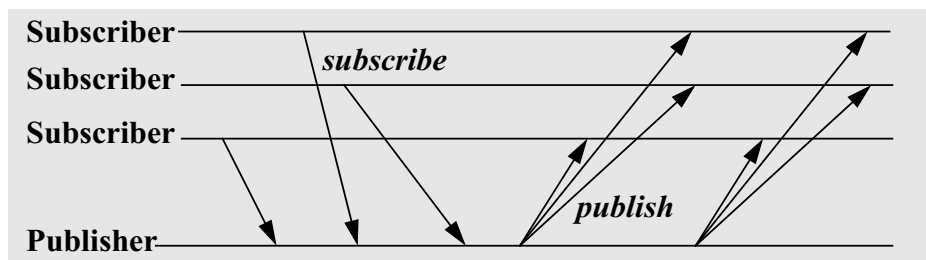
## 4. Das Tolerieren von Prozessausfällen ist schwierig:

- erfordert geeignetes Fehlermodell (z.B. Fail-Stop)
  - fälschliches Toterklären nicht immer vermeidbar
-



# Push bzw. Publish & Subscribe

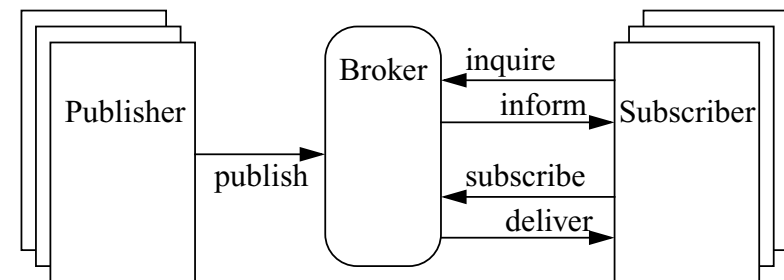
- Im Unterschied zum klassischen “Request / Reply-” bzw. “Pull-Paradigma”
  - wo Clients die gewünschte Information aktiv anfordern müssen
  - ein Client aber nicht weiss, ob bzw. wann sich eine Information geändert hat
  - dadurch periodische Nachfrage beim Server notwendig (“polling”)
- Subscriber (= 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 dann 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

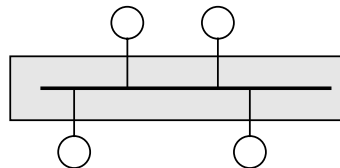
## - Stark entkoppelte Kommunikation

- 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
- ggf. Pufferung von Ereignissen



## - 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 und 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)

# 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 bereits 1985 von David Gelernter für die Sprache Linda

## - 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:  $\text{in } i, j$ ;  $\text{in}(\text{“Buchung”}, ?i, ?j)$  liefert *ein* “passendes” Tupel
- analog zu einigen Datenbankabfragesprachen (z.B. QbE)

## - Synchroner und asynchroner 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 sind 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. 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 enthält 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)
- *fehlertolerante Lösung?* (z.B. Replikation mit kausal atomarem Broadcast)

### - Kritik: globaler Speicher ist der strukturierten Programmierung und der Verifikation abträglich

- unüberschaubare potentielle Seiteneffekte
- evtl. 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
- 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

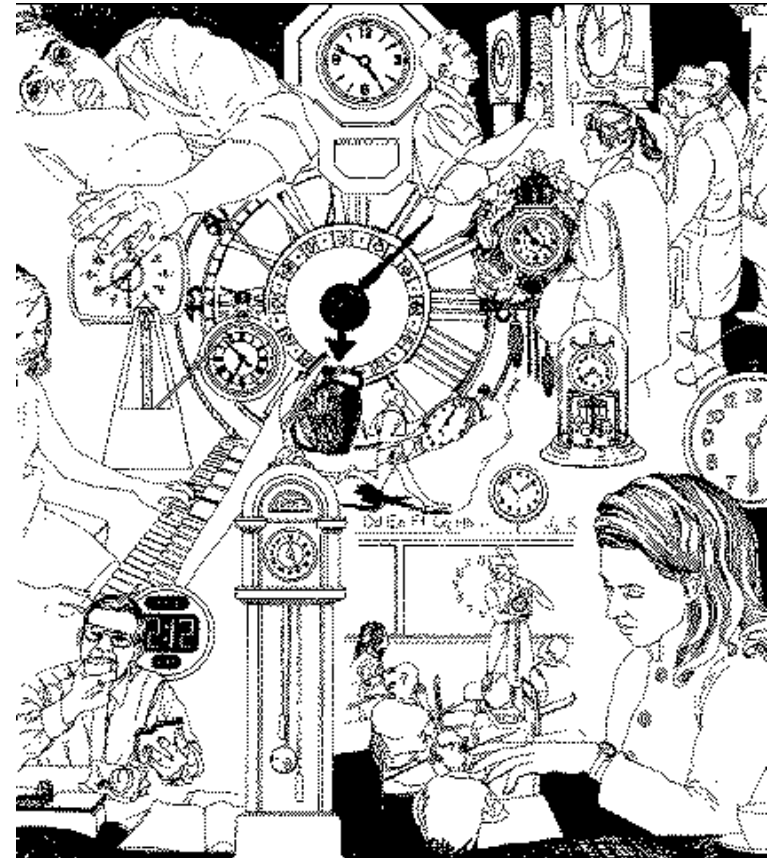
- selbst wenn ein *write* vor einem *read* beendet wird, muss *read* nicht notwendigerweise das lesen, was *write* geschrieben hat

# Logische Zeit und wechselseitiger Ausschluss

## Zeit?

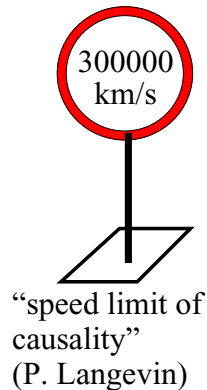
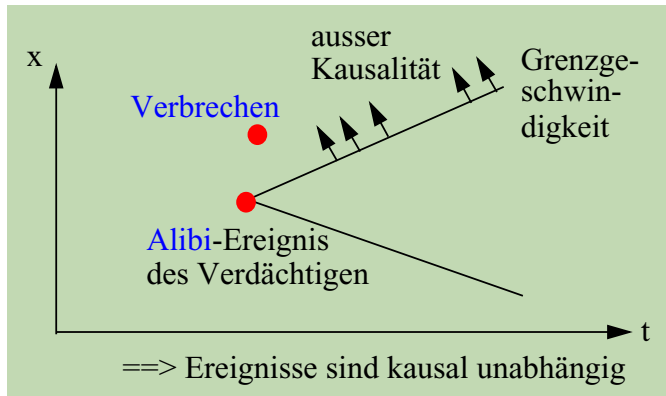
*Ich halte ja eine Uhr für überflüssig.  
Sehen Sie, ich wohne ja ganz nah beim Rathaus. Und  
jeden Morgen, wenn ich ins Geschäft gehe, da schau  
ich auf die Rathausuhr hinauf, wieviel Uhr es ist, und  
da merke ich's mir gleich für den ganzen Tag und  
nütze meine Uhr nicht so ab.*

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



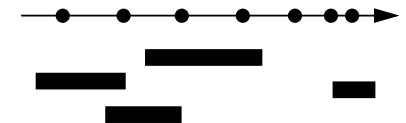
3. **Fairer wechselseitiger Ausschluss**
  - bedient wird, wer am längsten wartet
4. Viele weitere nützliche **Anwendungen** in unserer "verteilten realen Welt"
  - z.B. **kausaltreue Beobachtung** durch "Zeitstempel" der Ereignisse

# 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
  - homogen
  - vergeht "von selbst" → jeder Zeitpunkt wird schliesslich erreicht

Ist das "Zeitpunktmodell" adäquat? Zeitintervalle?

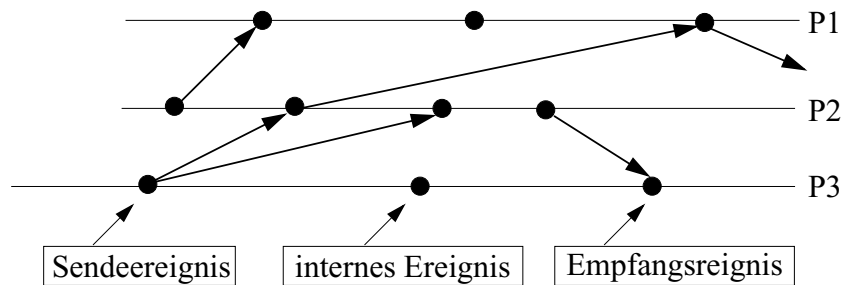


- Vgl. passé simple / imparfait im Französischen
- Wann tritt das Ereignis "Sonne wird rot" am Abend ein?

Welche Eigenschaften benötigen wir wirklich?

- Was wollen wir mit "Zeit" anfangen?
- "billigeren" Ersatz für fehlende globale Realzeit!  
(sind die reellen / rationalen / ganzen Zahlen gute Modelle?)
- Wann genügt "logische" Zeit? (Und was ist das genau??)

# Raum-Zeitdiagramme



- interessant: von links nach rechts verlaufende "Kausalitätspfade"

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

“Kleinste” Relation auf  $E$ , so dass  $x < y$  wenn:

- 1)  $x$  und  $y$  auf dem gleichen Prozess stattfinden und  $x$  vor  $y$  kommt, *oder*
- 2)  $x$  ist ein Sendereignis und  $y$  ist das korrespondierende Empfangsereignis, *oder*
- 3)  $\exists z$  mit  $x < z \wedge z < y$

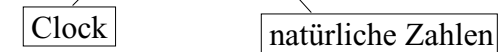
- Relation wird oft als “*happened before*” bezeichnet

- eingeführt von Lamport (1978)
- aber Vorsicht: damit ist nicht direkt eine "zeitliche" Aussage getroffen!

# Logische Zeitstempel von Ereignissen

- Zweck: Ereignissen eine Zeit geben ("dazwischen" egal)

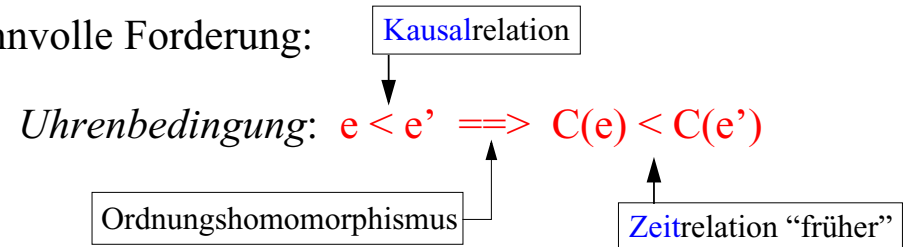
- Gesucht: Abbildung  $C: E \rightarrow \mathbb{N}$



- 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:



Interpretation ("Zeit ist kausaltreu"):

**Wenn ein Ereignis  $e$  ein anderes Ereignis  $e'$  beeinflussen kann, dann muss  $e$  einen kleineren Zeitstempel als  $e'$  haben**

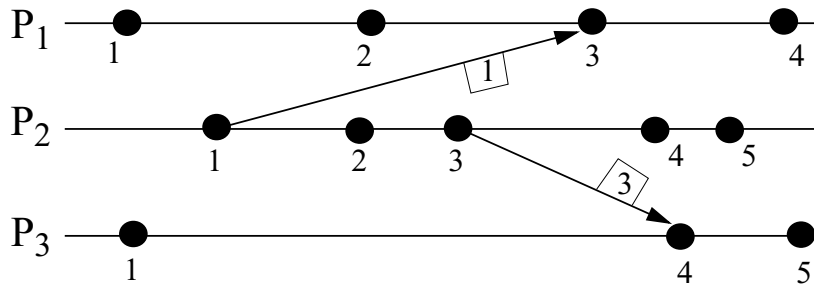
# Logische Uhren von Lamport

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

$C: (E, <) \rightarrow (N, <)$  Zuordnung von Zeitstempeln

Kausal-  
relation

$e < e' \implies C(e) < C(e')$  Uhrenbedingung



Protokoll zur Implementierung der Uhrenbedingung:

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

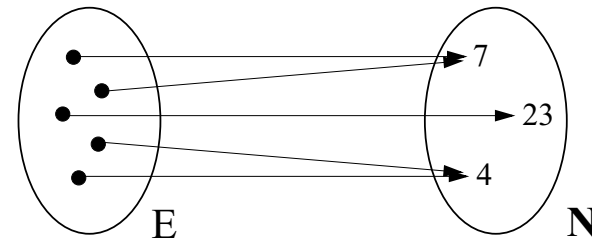
zuerst! danach "ticken"

*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  $(a, b) < (a', b') \iff a < a' \vee a = a' \wedge b < b'$

→ alle Ereignisse haben *verschiedene* Zeitstempel

→ Kausalitätserhaltende Abb.  $(E, <) \rightarrow (N \times N, <)$

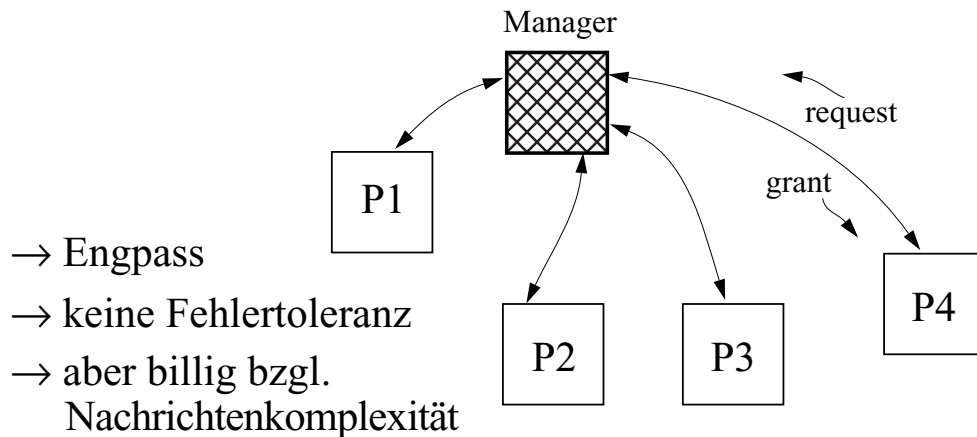
Jeder (nicht-leere) Menge von Ereignissen hat ein eindeutig "frühestes"!



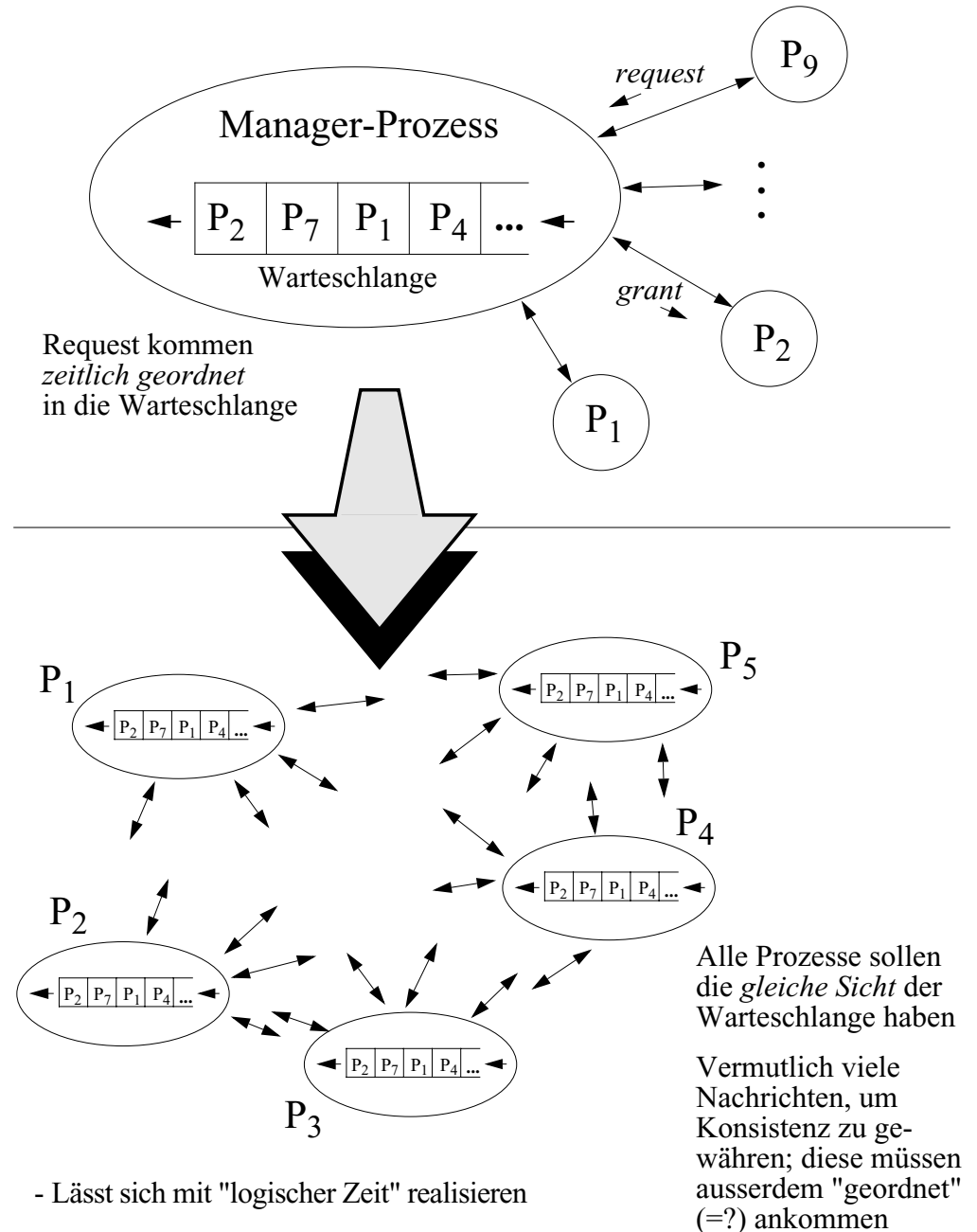
# Wechselseitiger Ausschluss

- Typischerweise exklusive Betriebsmittel
  - z.B. konkrete Betriebsmittel wie gemeinsamer Datenbus
  - oder abstrakte Betriebsmittel wie z.B. "Termin" in einem (verteilten) Terminkalendersystem
  - "kritischer Abschnitt" in einem (nebenläufigen) Programm
- Bei Einprozessormaschinen, shared memory etc.
  - Semaphore oder ähnliche Mechanismen
    - ==> Betriebssystem- bzw. Concurrency-Theorie
    - ==> interessiert uns hier nicht!
  - grundsätzlich interessant allerdings: was sind die elementaren Basis-mechanismen, um wechselseitigen Ausschluss realisieren zu können?

- Nachrichtenbasierte Lösung, die uns nicht interessiert, da asymmetrisch ("zentralisiert"):



# Replizierte Warteschlange?





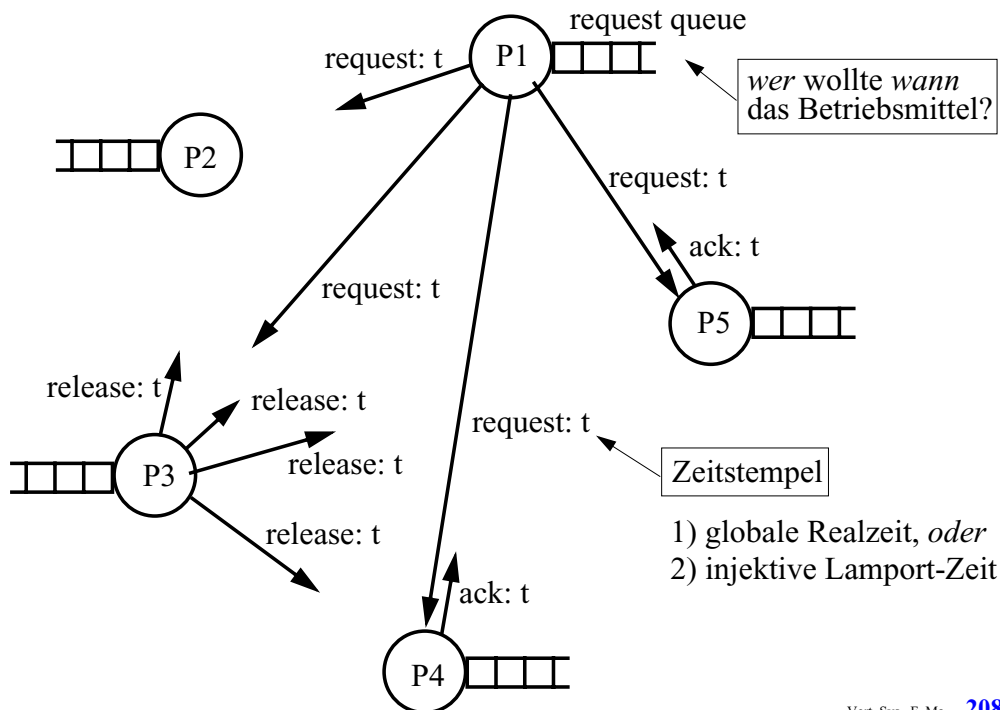
# Anwendung logischer Zeit für den wechselseitigen Ausschluss

- Feste Anzahl von Prozessen
- Ein exklusives Betriebsmittel
- Synchronisierung mit request / release-Nachrichten
- Fairness: Jeder request wird schliesslich erfüllt

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

Sehr schwache Fairnessanforderung

Idee: Replikation einer globalen request queue



## Der Algorithmus (Lamport '78):

- Voraussetzung: FIFO-Kommunikationskanäle
- *Alle* Nachrichten tragen (eindeutige!) Zeitstempel
- 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.
  - weiterer Nachr.typ
  - 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 (irgendeine) spätere Nachricht von allen anderen Prozessen bekommen.

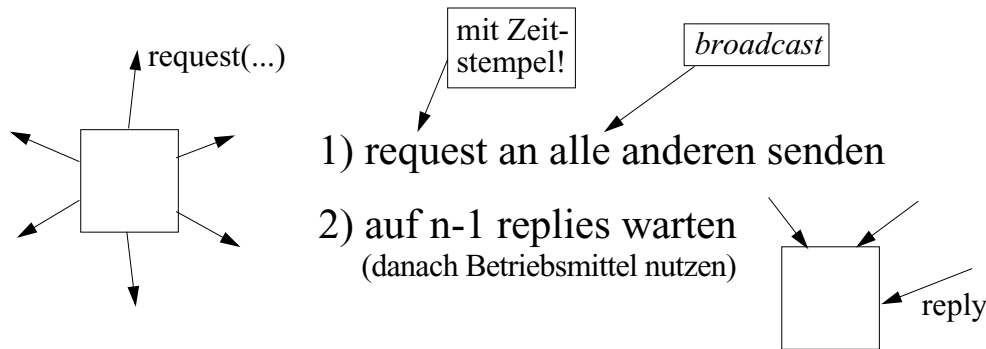
- Frühester request ist global eindeutig.
  - ==> bei 5): sicher, dass kein früherer request mehr kommt (wieso?)
- 3 (n-1) Nachrichten pro "request"

Denkübungen:

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

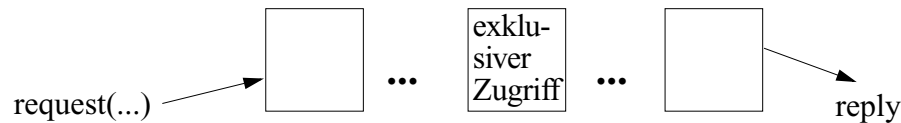
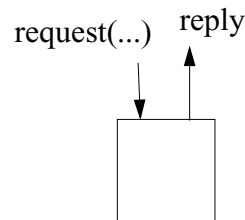
# Ricart / Agrawala 1981: "An Optimal Algorithm for..."

- $2(n-1)$  Nachrichten statt  $3(n-1)$  beim Lamport-Verfahren:  
(*reply-Nachricht* übernimmt Rolle von *release* und *ack*)



- Bei Eintreffen einer request-Nachricht:

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



- Ältester Bewerber setzt sich durch!

Denkübungen:

- Argumente für die Korrektheit? (Exklusivität, Deadlockfreiheit)
- Wie oft muss ein Prozess maximal nachgeben? ( $\rightarrow$  Fairness)
- Sind FIFO-Kanäle notwendig?
- Geht es wirklich nicht mit weniger Nachrichten? ("Optimal"?)