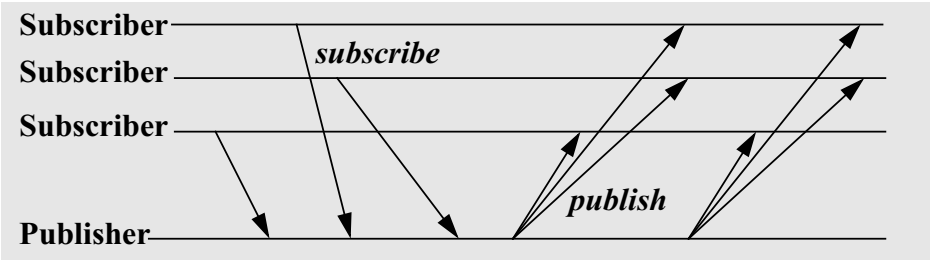
Push bzw. Publish & Subscribe

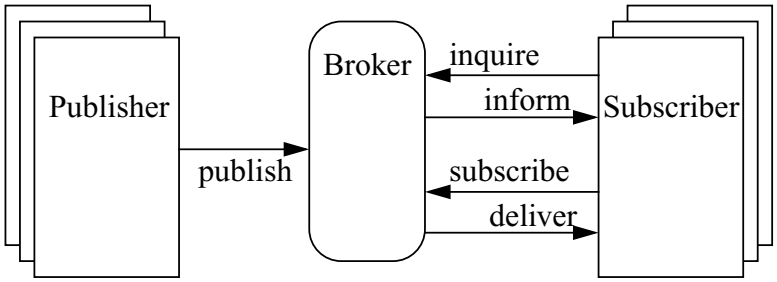
- Im Unterschied zum klassischen “Request / Reply-” bzw. “Pull-Paradigma”
  - wo Clients die gewünschte Information aktiv anfordern müssen
  - ein Client aber nicht weiss, ob bzw. wann sich eine Information geändert hat
  - dadurch periodische Nachfrage beim Server notwendig sind (“polling”)
- Subscriber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



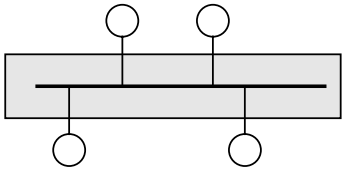
- Subscriber erfahren vom Broker, welche Kanäle abonniert werden können
  - Broker kann auch noch andere nützliche Dienste realisieren
- Subscriber melden sich beim Broker (statt beim Publisher) an
- Typische Rollenverteilung:
  - Publisher als *Produzent* von Information
  - Subscriber als *Konsument*

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 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 und Jini (event generator bzw. remote event listener)
  - event service von CORBA: sprach- und plattformunabhängig; typisierte und untypisierte Kanäle; Schnittstellen zur Administration von Kanälen; Semantik 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: int i,j; in(“Buchung”, ?i, ?j) liefert *ein* “passendes” Tupel
  - analog zu einigen relationalen 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 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

- "Tupelraum" für Java, orientiert sich am Linda-Vorbild
  - gespeichert werden Objekte → neben Daten auch "Verhalten"
  - Tupel entspricht Gruppen von Objekten
- Teil der Jini-Infrastruktur für verteilte Java-Anwendungen
  - Kommunikation zwischen entfernten Objekten
  - Transport von Programmcode ("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

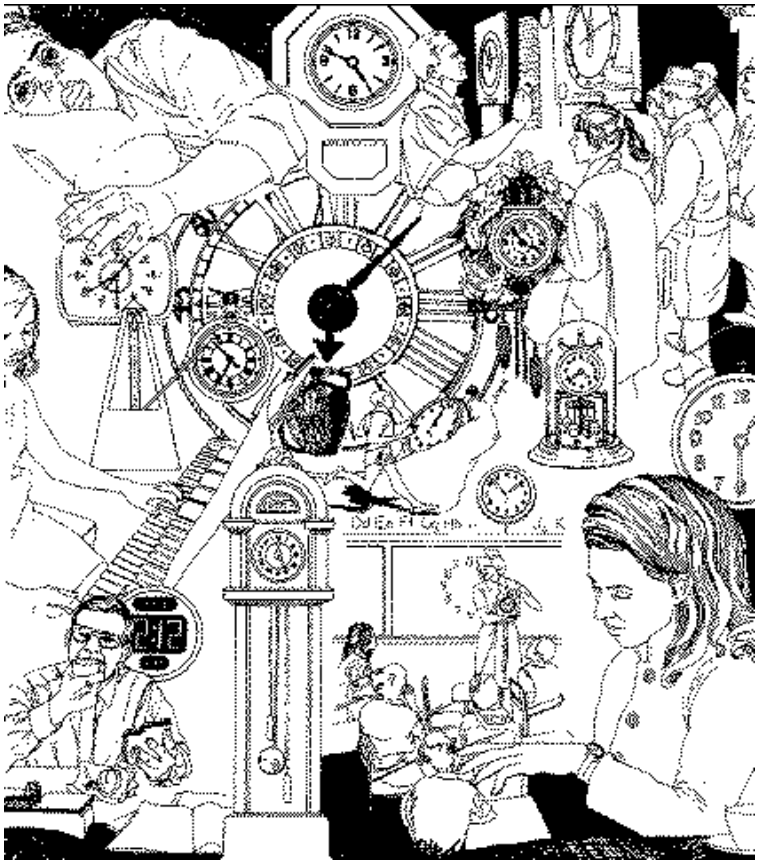
  } aber keine Festlegung, ob eine Implementierung fehlertolerant ist und einen Crash überlebt
- *Implementierung*: könnte z.B. auf einer relationalen oder objektorientierten Datenbank beruhen
- *Semantik*: Reihenfolge der Wirkung von Operationen verschiedener Threads ist nicht festgelegt
  - selbst wenn 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.

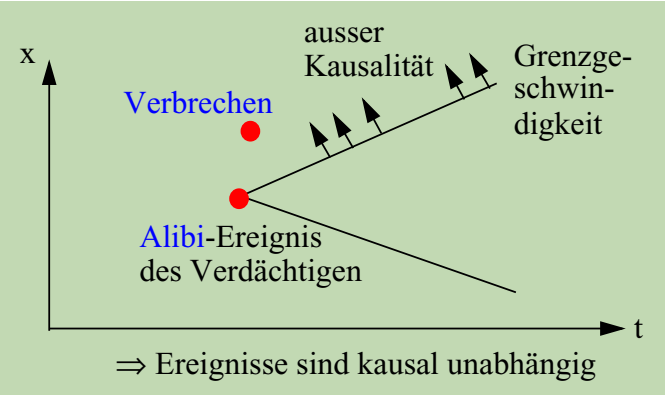
Karl Valentin

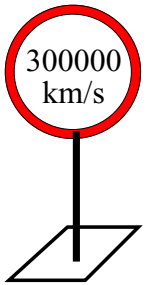


Kommt Zeit, kommt Rat

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





"speed limit
of causality"
(P. Langevin)

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:

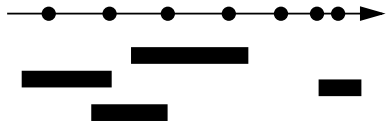
- transitiv
- irreflexiv
- linear

} → 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?



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

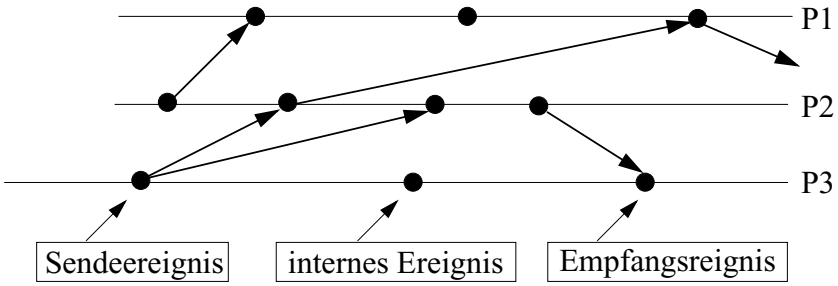
Welche Eigenschaften benötigen wir wirklich?

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

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

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

Raum-Zeitdiagramme



- 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 Sendeereignis und **y** ist das korrespondierende Empfangsereignis, *oder*

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

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

Logische Zeitstempel von Ereignissen

- Zweck: Ereignissen eine Zeit geben ("dazwischen" egal)
- Gesucht: Abbildung C: E $\rightarrow$ **N**
  - C: Clock
  - **N**: natürliche Zahlen
- Für e $\in$ E heisst C(e) *Zeitstempel* von e
- C(e) bzw. e *früher* als C(e') bzw. e', wenn C(e) < C(e')

- Sinnvolle Forderung:

Uhrenbedingung: e < e' ==> C(e) < C(e')

- e < e': Kausalrelation
- ==>: Ordnungshomomorphismus
- C(e) < C(e'): Zeitrelation "früher"

Interpretation ("Zeit ist kausaltreu"):

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

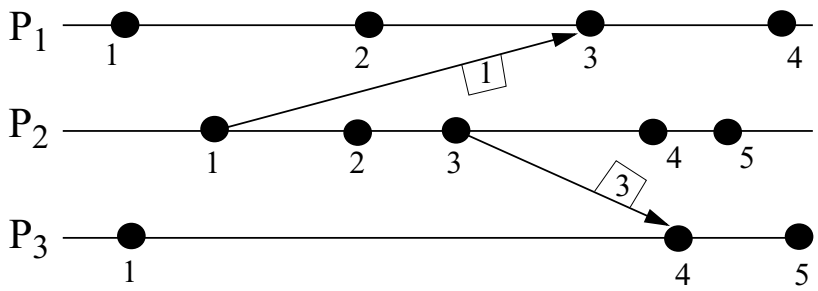
Logische Uhren von Lamport

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

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

Kausal-relation

e < e' ==> C(e) < C(e') Uhrenbedingung



Protokoll zur Implementierung der Uhrenbedingung:

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

zuerst! danach "ticken"

Behauptung:
Protokoll respektiert Uhrenbedingung

Beweis: Kausalitätspfade sind monoton...

Lamport-Zeit: Nicht-Injektivität

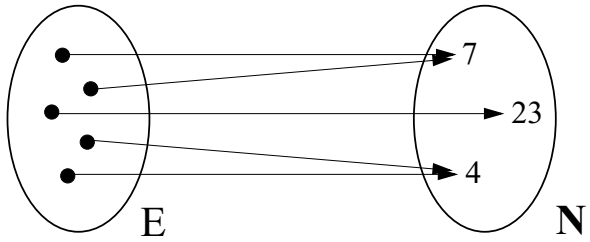


Abbildung ist nicht injektiv

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

- Lösung:

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

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

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

  → alle Ereignisse haben *verschiedene* Zeitstempel

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

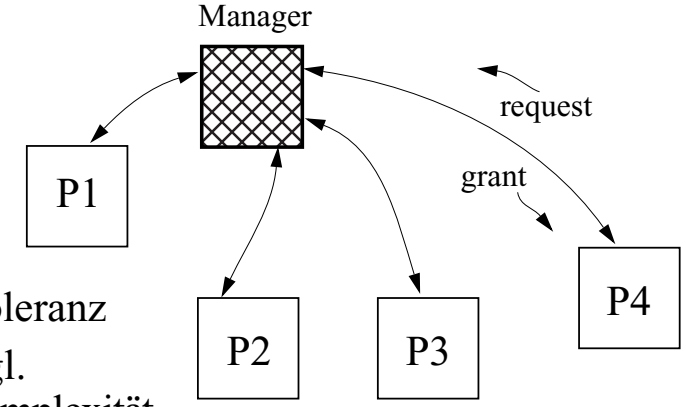
Jede (nicht-leere) Menge von Ereignissen hat so ein eindeutig "frühestes"!

Wechselseitiger Ausschluss

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

- Lösungen für Einprozessormaschinen, shared memory etc. nutzen typw. Semaphore oder ähnliche Mechanismen
  - ⇒ Betriebssystem- bzw. Concurrency-Theorie
  - ⇒ interessiert uns hier (bei verteilten Systemen) nicht!
  - grundsätzlich interessant allerding: was sind die elementaren Basis-mechanismen, um wechselseitigen Ausschluss realisieren zu können?

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

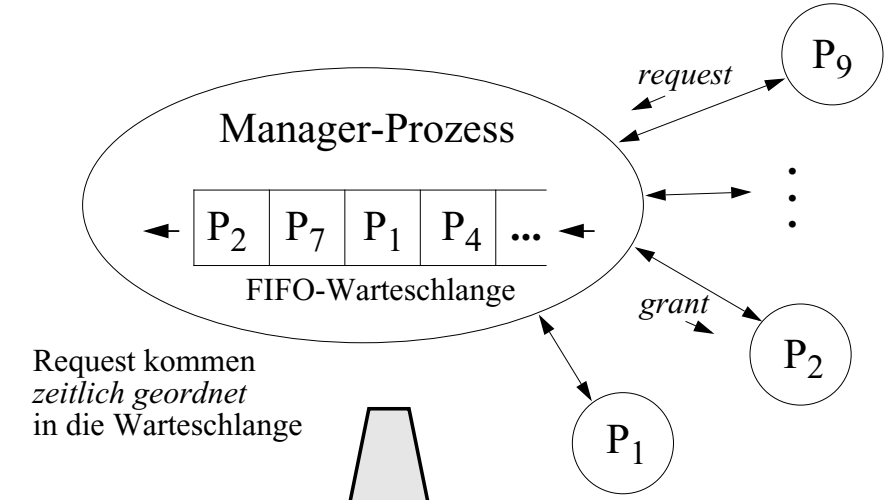


→ Engpass

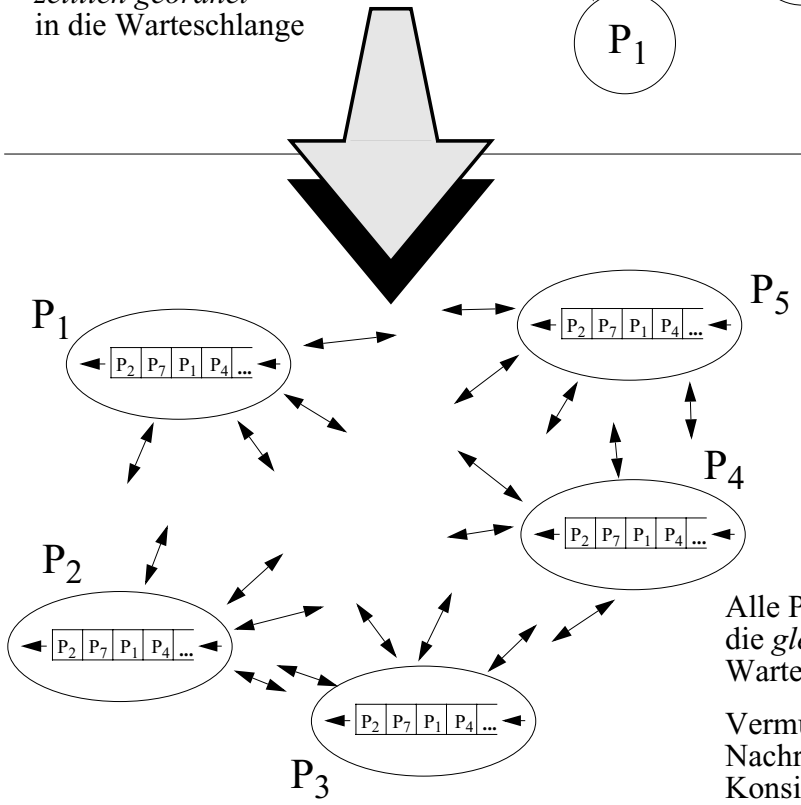
→ keine Fehlertoleranz

→ aber billig bzgl. Nachrichtenkomplexität

Replizierte Warteschlange?



Request kommen *zeitlich geordnet* in die Warteschlange



- Lässt sich mit "logischer Zeit" realisieren

Alle Prozesse sollen die *gleiche Sicht* der Warteschlange haben

Vermutlich viele Nachrichten, um Konsistenz zu gewähren; diese müssen ausserdem "geordnet" (=?) ankommen

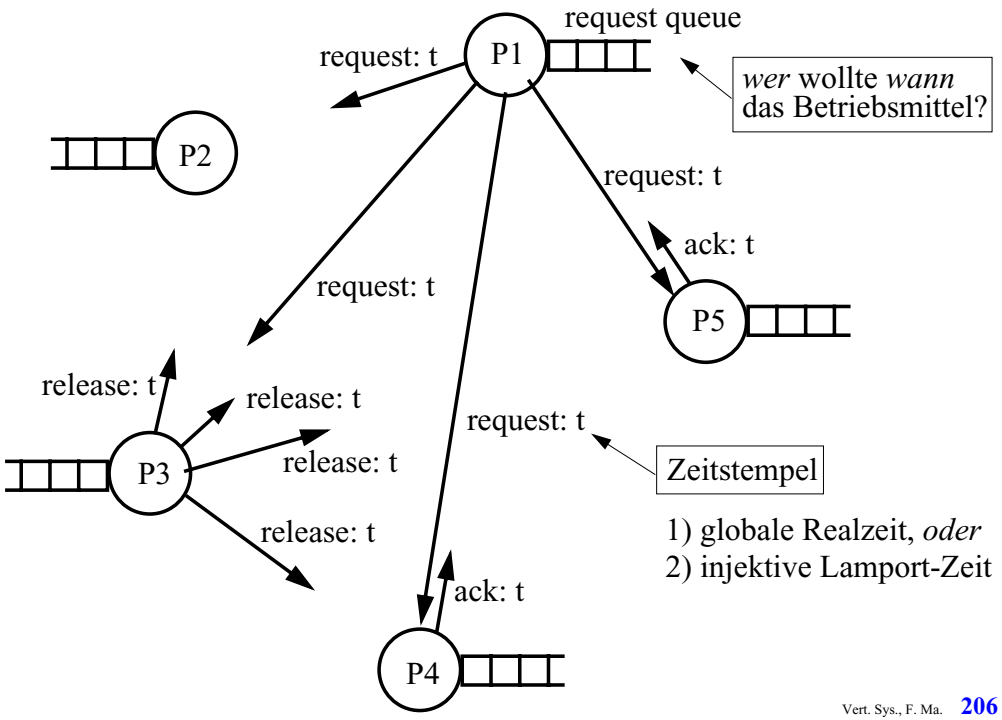
Anwendung logischer Zeit für den wechselseitigen Ausschluss

- 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



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

Der Algorithmus (Lamport 1978):

- 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: wieso notwendig?
Request in eigene queue einfügen, ack versenden.

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

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

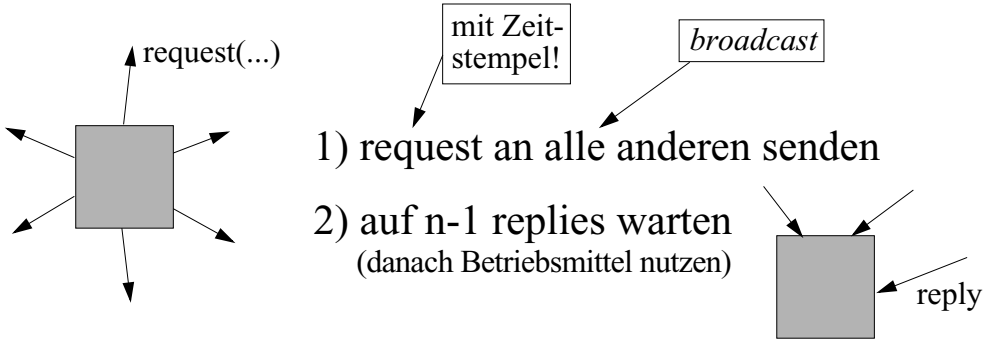
5) Ein Prozess darf das Betriebsmittel benutzen, wenn:
- eigener request ist frühester in seiner queue und
- hat bereits von jedem anderen Prozess (irgendeine) spätere Nachricht bekommen.

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

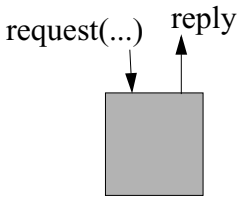
Denkübungen:
- wo geht Uhrenbedingung / Kausaltreue der Lamport-Zeit ein?
- sind FIFO-Kanäle wirklich notwendig? (Szenario hierfür?)
- bei Broadcast: welche Semantik? (FIFO, kausal,...?)
- was könnte man bei Nachrichtenverlust tun? (→ Fehlertoleranz)

"An Optimal Algorithm for..." (Ricart und Agrawala, 1981)

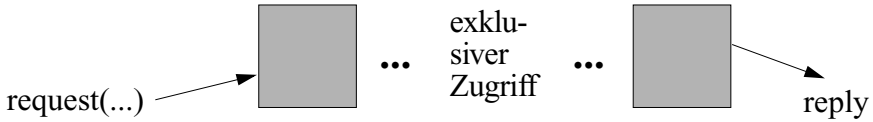
- 2(n-1) Nachrichten statt 3(n-1) beim Lamport-Verfahren:
  (*reply-Nachricht* übernimmt Rolle von release und ack)



- Bei Eintreffen einer request-Nachricht:



  - reply sofort schicken, wenn nicht selbst 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!
  -injektive Lamport-Zeit!

Denkübungen:
- Argumente für die Korrektheit? (Exklusivität, Deadlockfreiheit)
- wie oft muss ein Prozess maximal “nachgeben”? (→ Fairness)
- sind FIFO-Kanäle notwendig?
- geht es wirklich nicht mit weniger Nachrichten? ("Optimal"?)