Multicast

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

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

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

Multicast-Gruppen

- Zweck 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

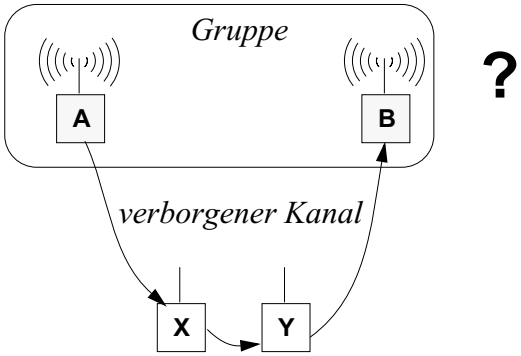
- *Offen*: Nicht-Gruppenmitglieder dürfen Multicast-Nachrichten an die Gruppe senden (im Gegensatz zu *geschlossenen* Gruppen)

- Statische / dynamische Gruppen

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

Problem der “Hidden Channels”

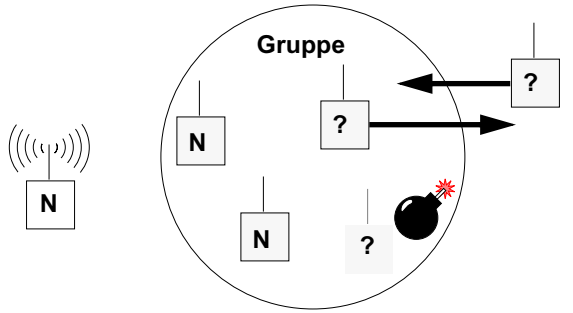
- Kausalitätsbezüge verlassen (z.B. durch Gruppenüberlappung) die Multicast-Gruppe und kehren später wieder



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

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

Gruppen-Management / -Membership



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

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

- Was bedeutet “alle Gruppenmitglieder”?

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

  } *während* eines Multicasts?

- Beachte:

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

- Folge:

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

- Problem: Wie erzielt man diese Übereinkunft?

Wechsel der Gruppenmitgliedschaft

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

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

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

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

Behandlung von Prozessausfällen

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

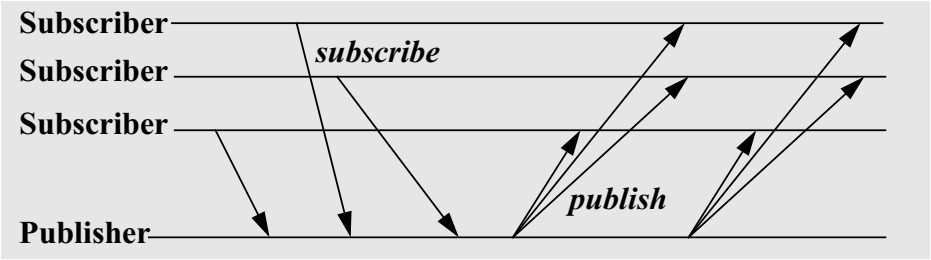
- Realisierungsmöglichkeit:
  - Ausfallzeitpunkt:
    - Prozesse dürfen nur Fail-Stop-Verhalten zeigen: *"Einmal tot, immer tot"*
    - Gruppenmitglieder erklären Opfer per kausalem, atomarem Multicast übereinstimmend für tot: *"Ich sage tot – alle sagen tot!"*
    - Beachte: *"Lebendiges Begraben"*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

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

Push bzw. Publish & Subscribe

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

- Subscriber (= Client) meldet sich für den Empfang der gewünschten Information an
  - z.B. “Abonnement” eines Informationskanals (“channel”)
  - u.U. auch dynamische, virtuelle Kanäle (→ “subject-based addressing”)



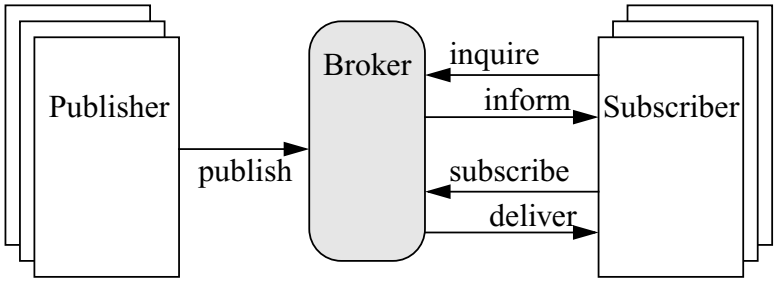
- Subscriber erhält automatisch (aktualisierte) Information, sobald diese zur Verfügung steht
  - “callback” des Subscribers (= Client) durch den Publisher (= Server)
  - push: “event driven” ↔ pull: “demand driven”

- “Publish” entspricht (logischem) Multicast
  - “subscribe” entspricht dann einem “join” einer Multicast-Gruppe
  - Zeitaktualität, Stärke der Multicast-Semantik und Grad an Fehlertoleranz wird oft unscharf als “Quality of Service” bezeichnet

Broker als Informationsvermittler

- Publisher und Subscriber müssen nicht direkt miteinander in Kontakt stehen

- Dazwischengeschalteter Broker verwaltet und vermittelt die Informationskanäle
  - führt zu noch stärkerer Entkoppelung von Sender und Empfänger
  - i.Allg. mehrerer verschiedener Publisher



- Subscriber erfahren vom Broker, welche Kanäle abonniert werden können
  - Broker kann auch noch andere nützliche Dienste realisieren

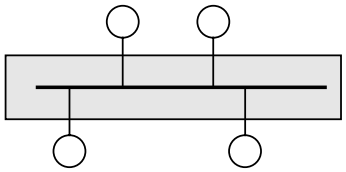
- Typische Rollenverteilung:
  - Publisher als *Produzent* von Information
  - Subscriber als *Konsument*

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
  - evtl. Pufferung von Ereignisen



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

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

Tupelräume

- Gemeinsam genutzter (“virtuell globaler”) Speicher
- Blackboard- oder Marktplatz-Modell
  - Daten können von beliebigen Teilnehmern eingefügt, gelesen und entfernt werden
  - relativ starke Entkoppelung der Teilnehmer
- Tupel = geordnete Menge typisierter Datenwerte
- Entworfen 1985 von D. Gelernter für die Sprache Linda
- Operationen:
  - out (t): Einfügen eines Tupels t in den Tupelraum
  - in (t): Lesen und Löschen von t aus dem Tupelraum
  - read (t): Lesen von t im Tupelraum
- Inhaltsadressiert (“Assoziativspeicher”)
  - Vorgabe eines Zugriffsmusters (bzw. “Suchmaske”) beim Lesen, damit Ermittlung der restlichen Datenwerte eines Tupels (“wild cards”)
  - Beispiel: int i,j; in(“Buchung”, ?i, ?j) liefert *ein* “passendes” Tupel
  - analog zu einigen relationalen Datenbankabfragesprachen (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)

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

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

Beachte: Zuordnung des "richtigen" Clients über die client_Id

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

JavaSpaces

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

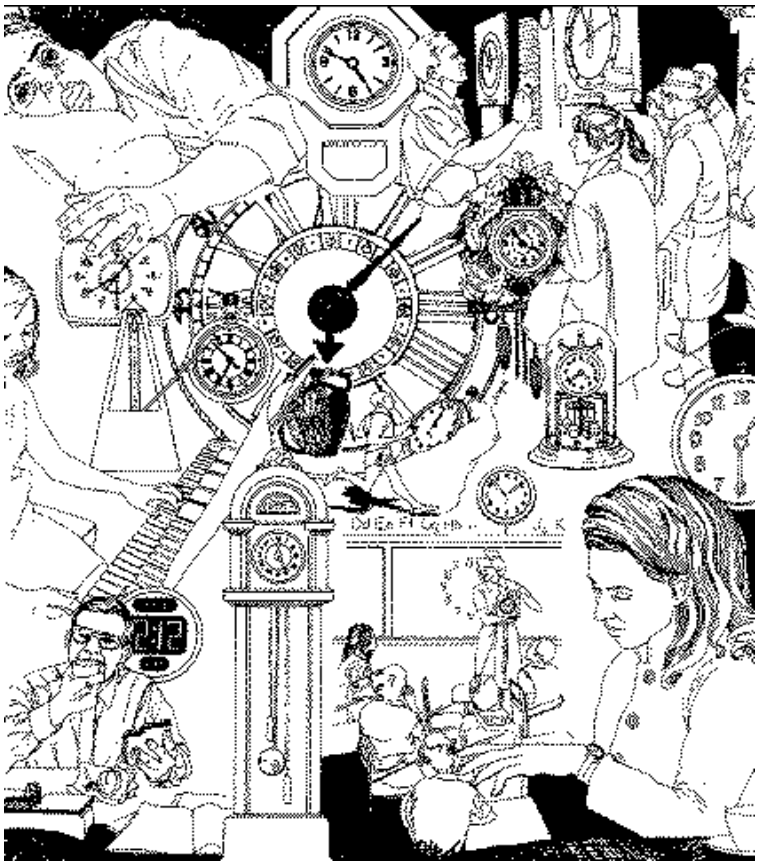
  } → aber keine Festlegung, ob eine Implementierung fehlertolerant ist und einen Crash überlebt
- *Implementierung* könnte z.B. auf einer relationalen oder objektorientierten Datenbank beruhen
- *Semantik*: Reihenfolge der Wirkung von Operationen verschiedener Threads ist nicht festgelegt
  - selbst wenn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

Logische Zeit und wechselseitiger Ausschluss

Zeit?

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

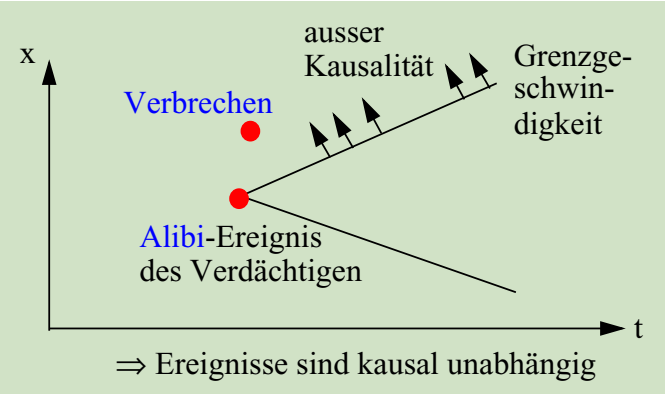
Karl Valentin

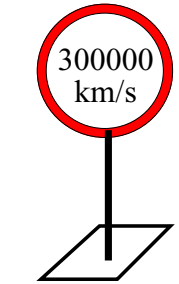


Kommt Zeit, kommt Rat

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

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





“speed limit
of causality”
(P. Langevin)

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

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

Eigenschaften der “Realzeit”

Formale Struktur eines Zeitpunktmodells:

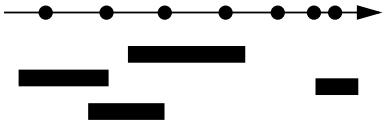
- transitiv
- irreflexiv
- linear

} → lineare Ordnung (“später”)

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

---

Ist das "Zeitpunktmodell"
adäquat? Zeitintervalle?



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

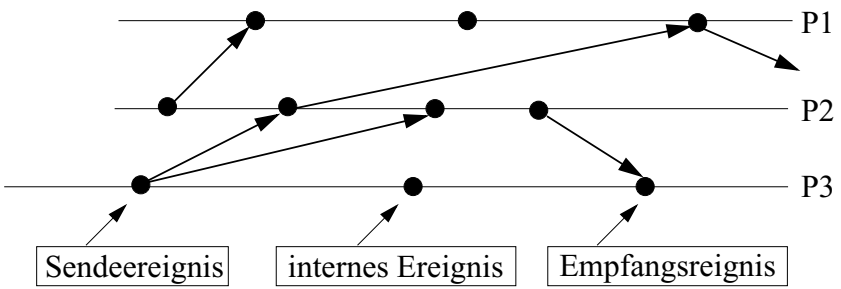
Welche Eigenschaften benötigen wir wirklich?

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

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

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

Raum-Zeitdiagramme



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

Logische Zeitstempel von Ereignissen

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

- e < e’: Kausalrelation
- $\Rightarrow$: 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

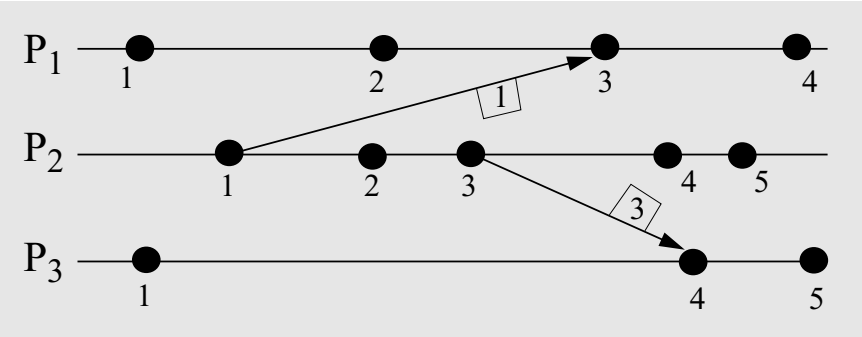
Logische Uhren von Lamport

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, erst danach "ticken"!

Behauptung:

Protokoll respektiert Uhrenbedingung

Beweis: Kausalitätspfade sind monoton...

Lamport-Zeit: Nicht-Injektivität

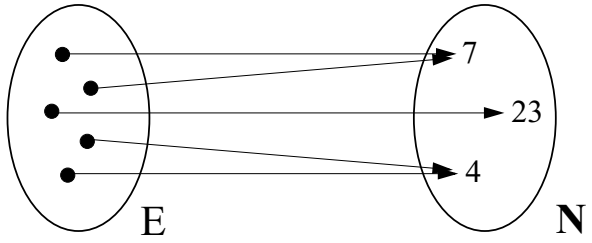


Abbildung ist nicht injektiv

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

- Lösung:

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

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

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

  → alle Ereignisse haben *verschiedene* Zeitstempel

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

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