Verteilte Systeme

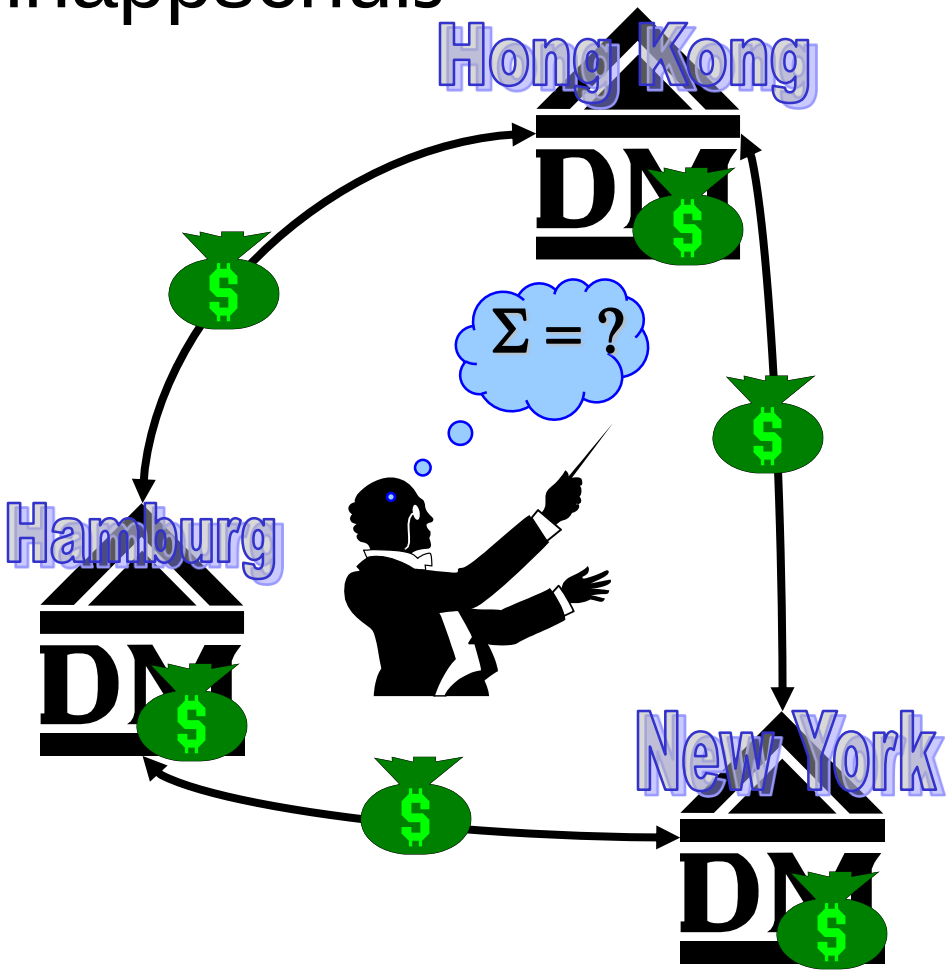
Zeit, Synchronisation und globale Zustände

Probleme

Problem: Schnappschuß

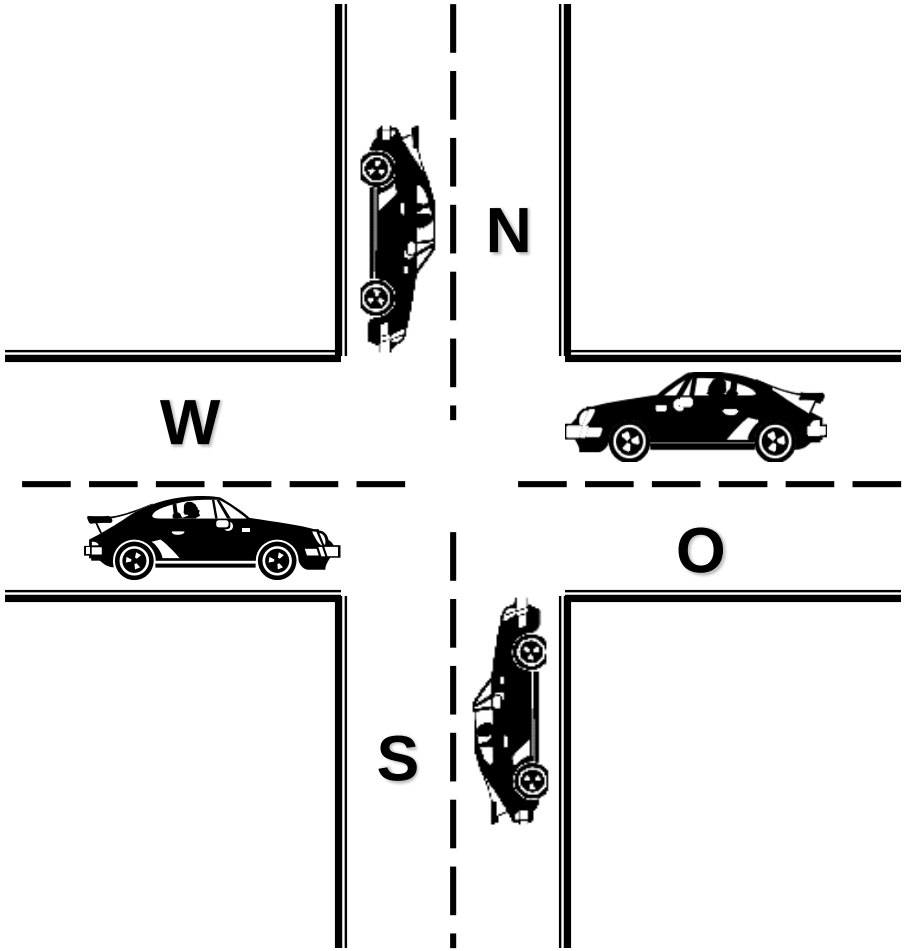
- **Globaler Zustand** eines verteilten Systems besteht aus den **lokalen Zuständen** aller Prozesse und aller unterwegs befindlichen **Nachrichten**

- **Erschwerte Bedingung**:
  - Niemand hat eine globale Sicht
  - Es gibt keine gemeinsame Zeit („Stichtag“)



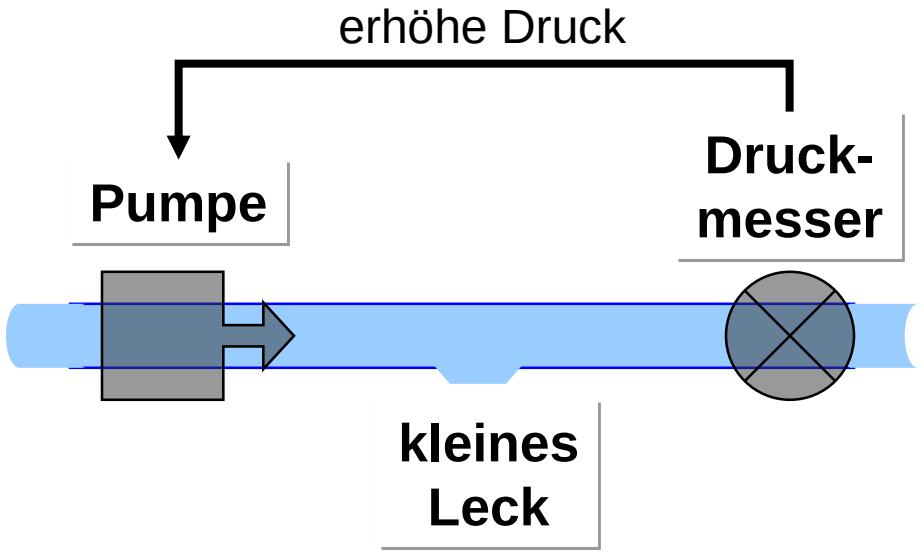
Problem: Phantom-Deadlock

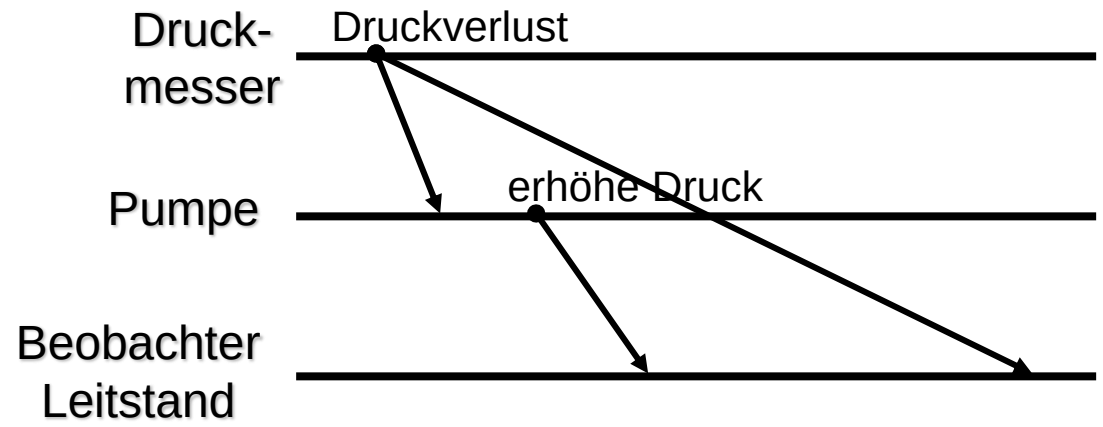
- Vier **Einzelbeobachtungen** der Autos:
  - N wartet auf W
  - S wartet auf O
  - O wartet auf N
  - W wartet auf S
- Einzelbeobachtungen **zu verschiedenen Zeitpunkten**
- Liefert **falschen Gesamteindruck**, als würden alle zu einem Zeitpunkt aufeinander warten



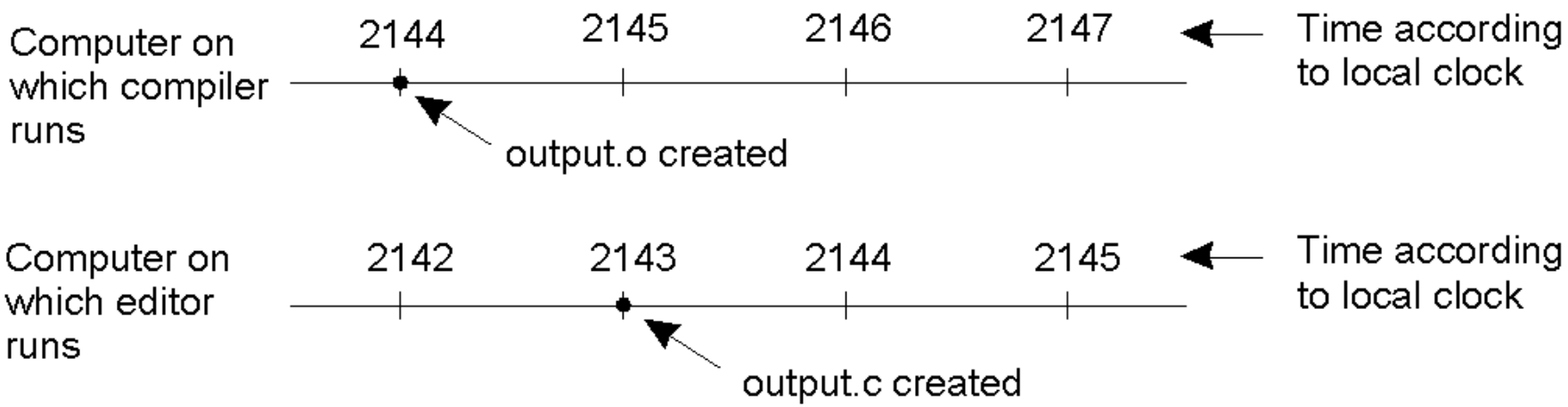
Problem: kausal inkonsistente Beobachtung

- Gewünscht: **Ursache** stets **vor** ihrer **Wirkung** beobachten
- Ohne Vorkehrung möglich: **Ursache** wird **nach Wirkung** beobachtet
- Kann man kausal konsistente Beobachtungen erzwingen ?





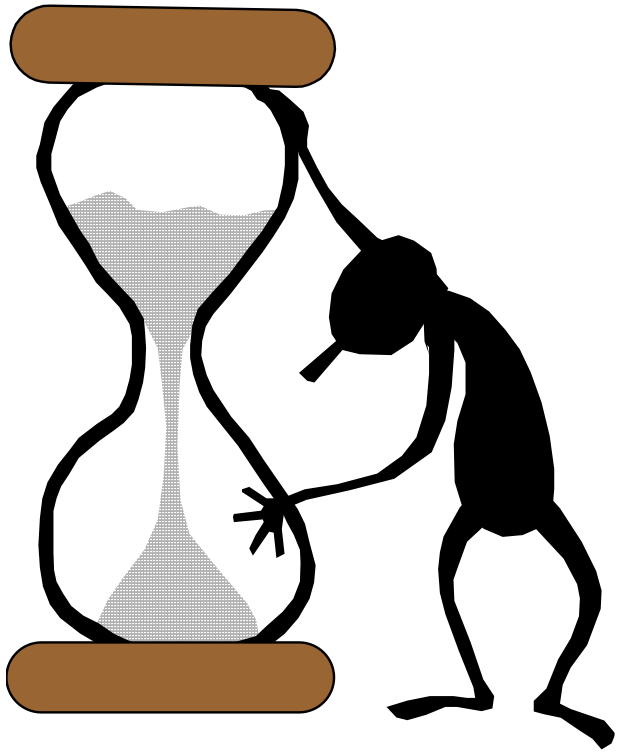
Beispiel: Verteilte Softwareentwicklung



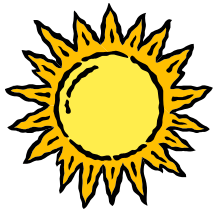
Die Quelldatei ist augenscheinlich **älter als** die Zieldatei.

Ergebnis: Sie wird bei einem erneuten `make` nicht neu übersetzt.

Physikalische Uhren



Physikalische Uhren: Sonne

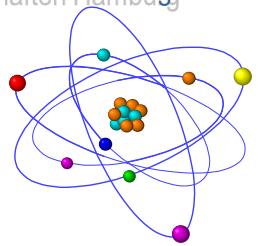


- **Sonnendurchlauf**: Zeitpunkt, zu dem die Sonne ihren höchsten Stand erreicht
- **Sonnentag**: Intervall zwischen zwei Sonnendurchläufen
- **Sonnensekunde**: Sonnentag / 86400

- **Problem**: Rotationsgeschwindigkeit der Erde nicht konstant
  - **längerfristig**: Verlangsamung durch Reibungsverluste
  - **kurzfristig**: Schwankungen durch Turbulenzen im flüssigen Erdkern

Daher **Mittel** über große Anzahl von Tagen

⇒ Mittlere Sonnensekunde

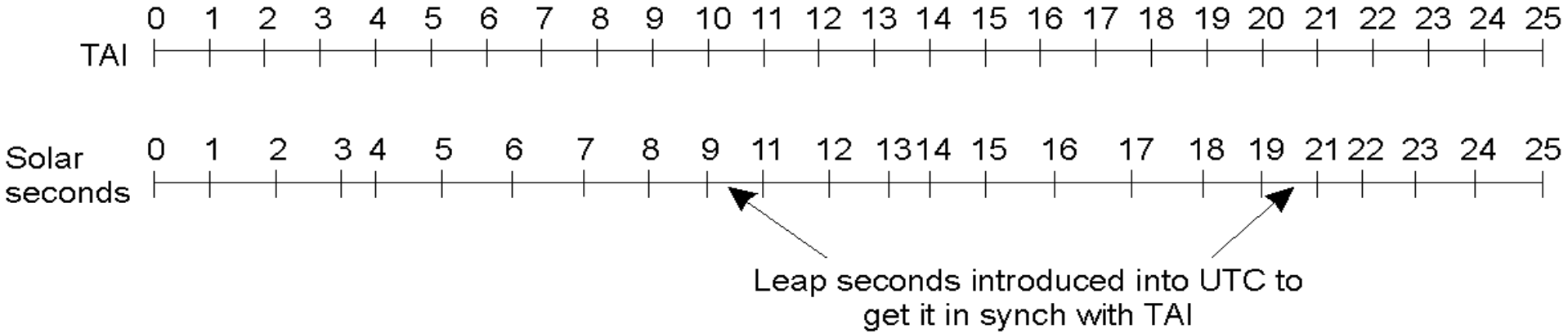


Physikalische Uhren: Atomzeit

- entsteht durch **Zählen von Zustandsübergängen** in Atomen
- erlaubt **präzisere Zeitmessung**
- definiert durch *Bureau International de l`Heure* (BIH) in Paris
- definiert **eine Sekunde** als die Zeit, die ein Cäsium-133-Atom für **9 192 631 770 Zustandsübergänge** benötigt
- Am **1.1.1958** entsprach diese **Atomsekunde** genau einer **Sonnensekunde** (wegen der Erdreibung wird die Sonnensekunde immer länger).
- Mittel über eine größere Anzahl von Atomuhren (betrieben von Forschungseinrichtungen) führt zur:

TAI (International Atomic Time)

TAI: Schaltsekunde



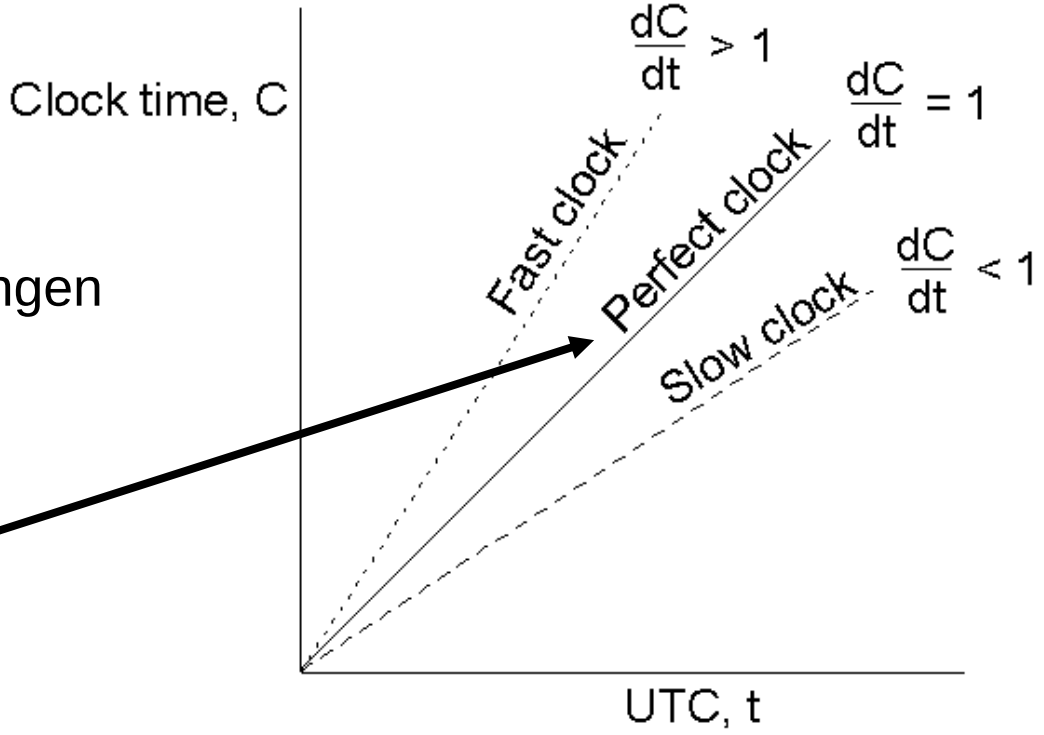
- **Problem**: Atomzeit und Sonnenzeit driften auseinander. Ein TAI-Tag ist zur Zeit ca. 3 msec kürzer als ein Sonnentag
- **Lösung**: BHI fügt **Schaltsekunden** (leap second) ein, wenn die Differenz zwischen Sonnenzeit und TAI auf 800 msec angewachsen ist.
- Dies führt zu

Universal Time Coordinated (UTC)

als internationale Basis für die Zeitmessung

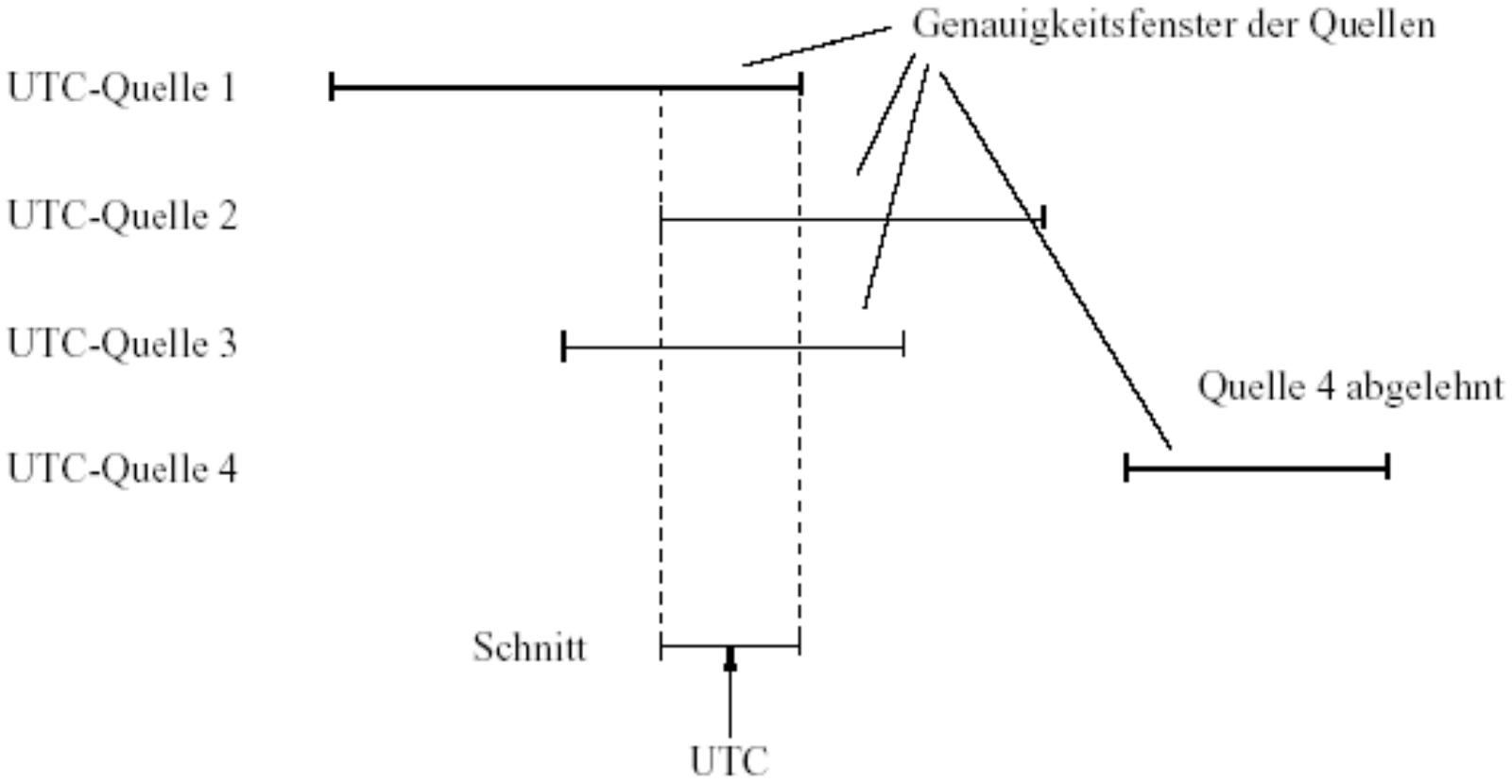
UTC

- **1967**: Koordinierte Universalzeit (UTC) berücksichtigt die Schaltsekunden.
- UTC-Signale werden von Satelliten (GEOS: Geostationary Environmental Operational Satellite, GPS: Global Positioning System) und Rundfunksendern ausgestrahlt.
- Genauigkeiten:
  - **Rundfunk**
    - ca. ± 1 msec
    - ca. ± 10 msec
      durch atmosphärische Störungen
  - **Satelliten**
    - GEOS: ca. ± 0,1 msec
    - GPS: ca. ± 1 msec
- **Problem**: Die perfekte Uhr



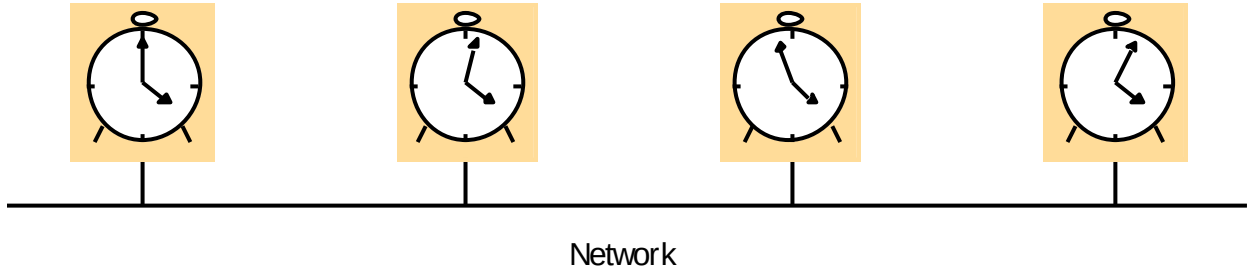
UTC: Verbesserung der Genauigkeit

- Verwendung **mehrerer UTC-Quellen** kann durch Bildung des Mittels eine Verbesserung der Genauigkeit bewirken.
- Prinzip zur Postionsbestimmung bei GPS



Synchronisation physikalischer Uhren

- **Bisher**: UTC-Empfänger nicht Standardausstattung eines Rechners.
- **Daher**: rechnerinterne physikalische Uhr wird von einem **Taktgeber** abgeleitet und zählt ein Uhrenregister hoch.
- Der Zählerinhalt oder ein durch Software daraus berechnetes **Zeit-Datum** kann als Zeitstempel für ein Ereignis verwendet werden.



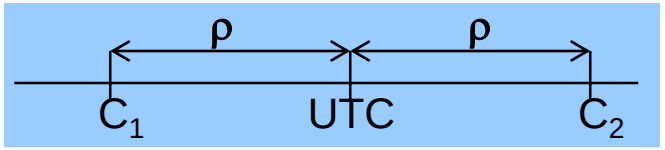
- **Problem**:
  - Uhren driften ca. 1 Sekunde in 10 Tagen, d.h. es besteht eine **Uhrendriftrate** von ca. 10^{-6}. Jede Uhr hat eigene Drift!
  - Die beteiligten Rechner **starten** nicht zum gleichen Zeitpunkt.

Synchronisation physikalischer Uhren

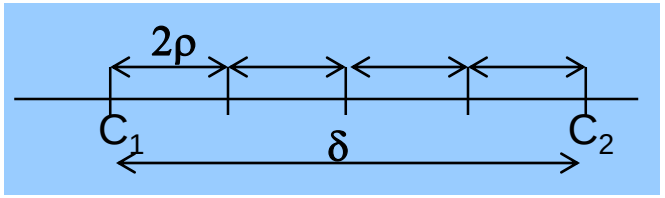
- Sei $C_i(t)$ der Wert, den die Uhr von Rechner i zum Zeitpunkt (UTC) t anzeigt (perfekt: immer $C_i(t) = t$)
- Die maximale Abweichung ρ der Uhr, vorgegeben durch den Hersteller, ist definiert durch

$$1 - \rho \leq dC_i/dt \leq 1 + \rho$$

  Ist die Abweichung größer, erfüllt sie nicht ihre Spezifikation
- Falls zwei Uhren von der UTC in entgegengesetzter Richtung abdriften, können sie nach Δt Zeiteinheiten nach ihrer Synchronisation um $2\rho\Delta t$ abweichen.



- Sollen zwei Uhren nicht mehr als δ Sekunden voneinander abweichen, so müssen sie alle $\Delta t = \delta/2\rho$ Sekunden synchronisiert werden



Absolute Uhrensynchronisation

Setze lokale Uhr auf $t := t_{UTC}$

Algorithmus von Cristian (1989)

Problem: Nachrichtenlaufzeiten T_{N1} & T_{N2} ?

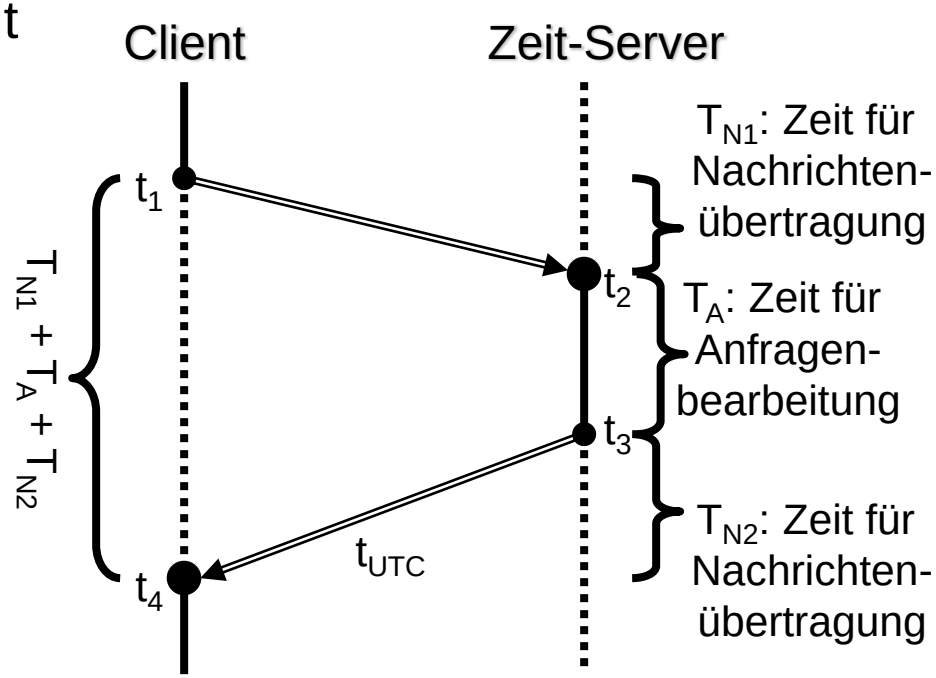
Verbesserung: Schätze Nachrichtenlaufzeit auf $(t_4-t_1)/2$, also $t := t_{UTC} + (t_4-t_1)/2$

Problem: Anfragebearbeitungszeit T_A ?

Verbesserung: Falls T_A (= t_3-t_2) bekannt: $t := t_{UTC} + ((t_4-t_1)-T_A)/2$

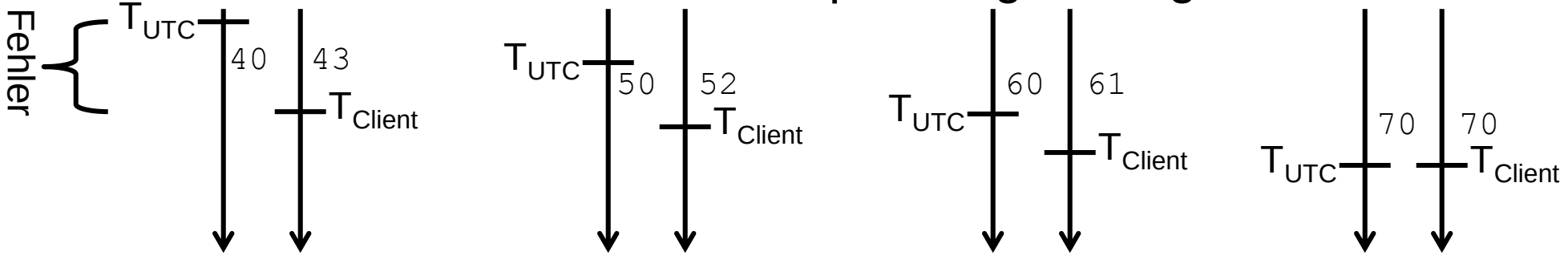
Problem: Einfluß der Netzlast ?

Verbesserung: Mehrere Messungen von (t_4-t_1) bzw. $((t_4-t_1)-T_A)$: Messungen über einem Schwellwert ablehnen, die andern zur Durchschnittsbildung verwenden



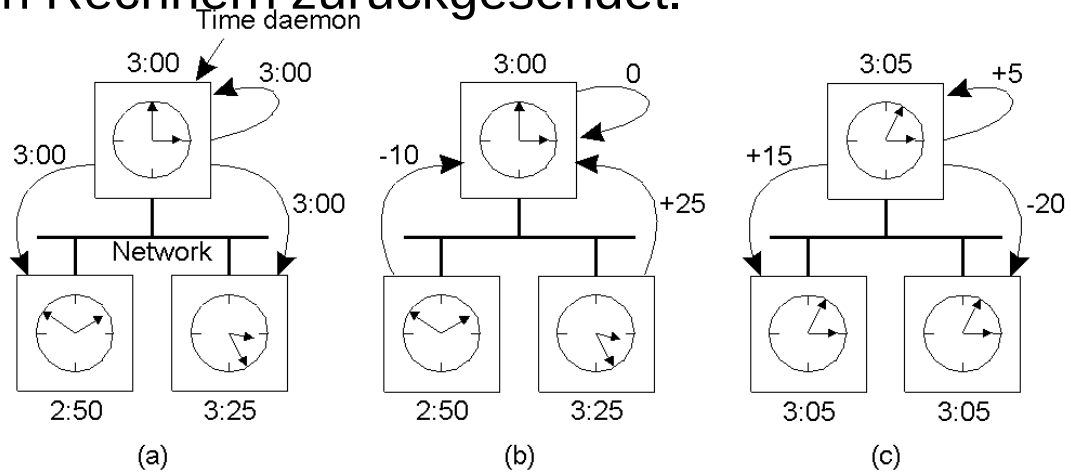
Absolute (externe) Uhrensynchronisation

- Falls Uhr **nachging**, so kann sie auf die UTC vorgestellt werden.
- Falls Uhr **vorging**, ist ein Zurückstellen problematisch, da die Zeit auch für den Rechner ein Kontinuum sein sollte: Eine **rückwärtslaufende Zeit** ist in aller Regel **nicht tolerabel** (Eindeutigkeit geht verloren).
- Daher Uhranpassung durch **Verzögerung** der Uhr. Beispiel:
  - Hardware-Timer erzeugt 100 Unterbrechungen / sec
  - Software-Timer erhöht die lokale Zeit nicht um 10 msec, sondern um 9 msec, bis die Anpassung vollzogen ist



Relative (interne) Uhrensynchronisation

- Falls eine einheitliche Zeit benötigt wird (**ohne UTC-Empfänger**)
- **Berkeley (UNIX) Algorithmus** (1989): Ein Rechner ist der *Koordinator*:
  - Zeit-Server (= Koordinator) fragt periodisch alle Rechner nach ihrer Uhrzeit
  - Aus den erhaltenden Antworten werden die lokalen Zeiten durch Schätzung der Nachrichtenlaufzeiten ermittelt. Antworten, die zu lange auf sich warten lassen, werden ignoriert.
  - Aus den geschätzten lokalen Zeiten wird das arithmetische Mittel gebildet.
  - Die jeweiligen Abweichungen vom Mittel werden als neuer aktueller Uhrenwert den Rechnern zurückgesendet.



Bewertung

Cristian Algorithmus

- Die interne Uhr muß um ein Vielfaches genauer sein, als die Laufzeit im Netz beträgt. (z.B. interne Uhr bei 10^{-6}, Laufzeit bei 1-10 msec)
- Die Laufzeit im Netz muß kurz gegenüber der geforderten Genauigkeit sein. (z.B. Laufzeit bei 1-10 msec, Genauigkeit von 1 sec)
- Hin- und Rücknachricht weden oft unterschiedlich lange brauchen.
- Fehleranfälligkeit durch die zentrale Architektur (z.B. Serverausfall).
- Keine Behandlung von fehlerhaften (oder betrügerischen) Zeit-Servern.

Berkeley (UNIX) Algorithmus

- Fehleranfälligkeit durch die zentrale Architektur. (Wahl eines neuen Zeit-Server bei Serverausfall notwendig)
- Im LAN lassen sich Genauigkeiten von ± 10 msec erreichen. Versuch der „Erfinder“: 15 Computer, synchronisiert (durch diesen Algorithmus) auf 20-25 msec. Maximale Laufzeit ca. 10 msec. Lokale Abweichgeschwindigkeit war weniger als $2 * 10^{-5}$ msec

Verteilter Zeitservice: Network Time Protocol

- **Absoluter (UTC-) Zeit-Service in großen Netzen (Internet)**:
  u.a. Statistische Techniken zur Überbrückung der großen und variablen Nachrichtenverzögerungen

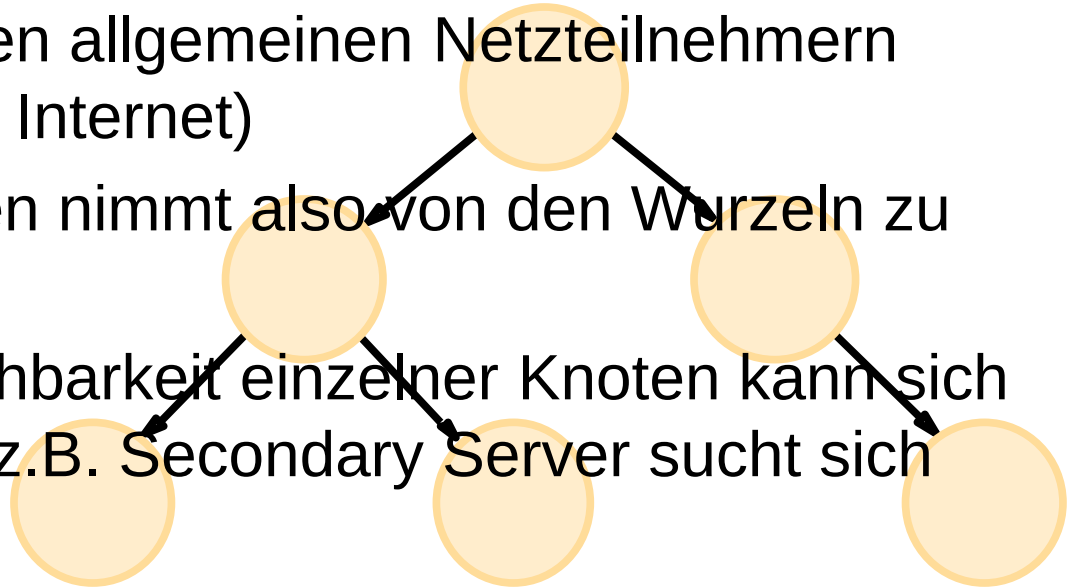
- **Hohe Zuverlässigkeit** (durch Fehlertoleranz):
  u.a. Redundante Server und Pfade zwischen Servern

- **Hohe Verfügbarkeit**:
  Server sind auf große Zugriffszahlen ausgelegt

- **Schutz gegen Verfälschungen**:
  u.a. Authentifizierungstechniken

Network Time Protocol Architektur

- Zeit-Server bilden ein hierarchisches **baumartiges Subnetz**
- **Primary Server** (Wurzelknoten, Schicht 1) besitzen UTC-Empfänger
- **Secondary Server** (Schicht 2) werden von Primary Server synchronisiert
- **Die Blätter** werden von den allgemeinen Netzteilnehmern gebildet (Workstations am Internet)
- Die **Genauigkeit** der Uhren nimmt also von den Wurzeln zu den Blättern ab
- **Bei Ausfall** oder Unerreichbarkeit einzelner Knoten kann sich das Netz rekonfigurieren, z.B. Secondary Server sucht sich neuen Primary Server



NTP Synchronisations Modi

- **Multicast**-mode
  - für schnelle lokale Netze; erzielt nur **geringe Genauigkeit**
  - Server schickt periodisch aktuelle Zeit per Multicast an anderer Rechner im LAN
- **Procedure-call**-mode
  - ähnlich Cristian's Algorithmus; falls Multicast nicht möglich oder zu ungenau (erzielt **mittlere Genauigkeit**)
  - Server beantwort Zeitanfragen mit dem aktuellen Zeitstempel
- **Symmetric**-mode
  - zur Synchronisation zwischen Zeit-Servern (erzielt **hohe Genauigkeit**); auf anderer Ebene, falls hohe Genauigkeit gefordert
  - Wechselseitiger Austausch von Zeitstempeln. Messen der Netzwerkverzögerung und Uhrengenauigkeit
- In allen Fällen wird das **UDP-Transportprotokoll** verwendet, d.h. mit Nachrichtenverlust muß gerechnet werden.

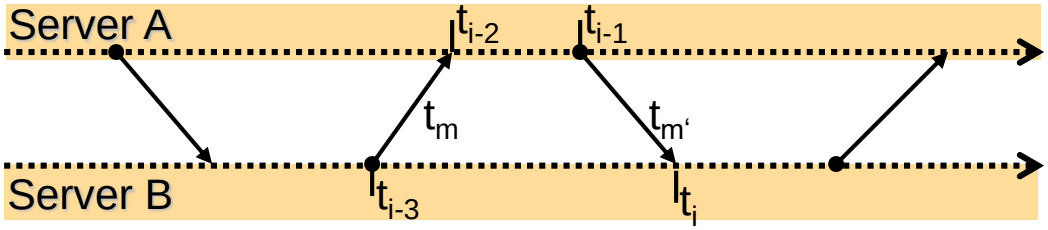
Da keine Information über die Laufzeit!

Da Laufzeiten als gleichlang abgeschätzt!

Da detaillierte Informationen über Laufzeiten bestehen!

NTP Details

- Vom Multicast abgesehen, werden Nachrichten **in Paaren** ausgetauscht
  Sende- und Empfangszeiten werden vermerkt und verschickt.

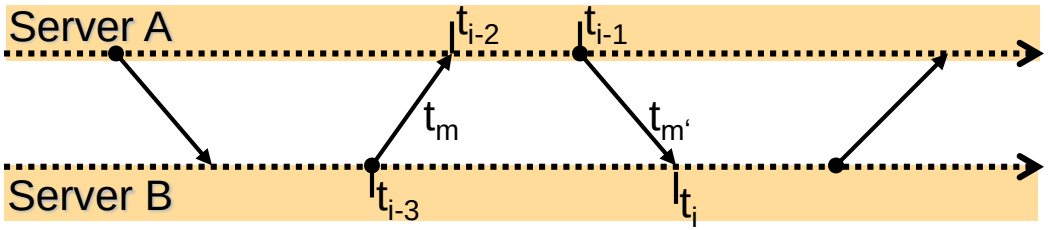


  - $o := t_A - t_B$ die wahre Zeitdifferenz von B relativ zu A (Offset)
  - o_i der Schätzwert für o
  - t_m und $t_{m'}$ die jeweiligen Nachrichtenlaufzeiten für m bzw. m'
  - $d_i = t_m + t_{m'}$ die gesamte Nachrichtenlaufzeit (Hin- und Rückweg)

- d_i kann gemessen werden: $\mathbf{d_i = (t_i - t_{i-3}) - (t_{i-1} - t_{i-2})} = (t_i - t_{i-1}) + (t_{i-2} - t_{i-3}) = t_m + t_{m'}$
- Sende- und Empfangszeiten werden mitverschickt!

NTP Details (2)

- $o := t_A - t_B$
- o_i Schätzwert f. o
- t_m und $t_{m'}$
- $d_i = t_m + t_{m'}$

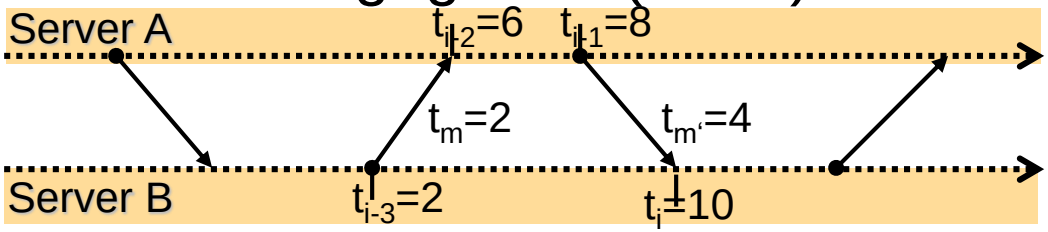


- Es gilt: **Empfangszeit – Sendezeit = Übertragungszeit + Offset**
- Also: $t_{i-2} - t_{i-3} = t_m + o$
- Und: $t_{i-1} - t_i = - t_{m'} + o$
- Annahme: $t_m = t_{m'}$ (Laufzeit ist symmetrisch)
- Dann gilt für den Schätzer: $\mathbf{o_i = ((t_{i-2} - t_{i-3}) - (t_i - t_{i-1}))/2} = (t_m - t_{m'})/2$

- wegen $|t_{m'} - d_i/2| \leq d_i/2$; $|t_m - d_i/2| \leq d_i/2$ gilt: $\mathbf{o_i - (d_i/2) \leq o \leq o_i + (d_i/2)}$
- $\mathbf{o_i}$ ist **Schätzung** für die Abweichung $\mathbf{d_i}$ ist die **Güte** dieser Schätzung

Beispiel: NTP Details

- Es sei folgende Situation gegeben (**o = 2**):

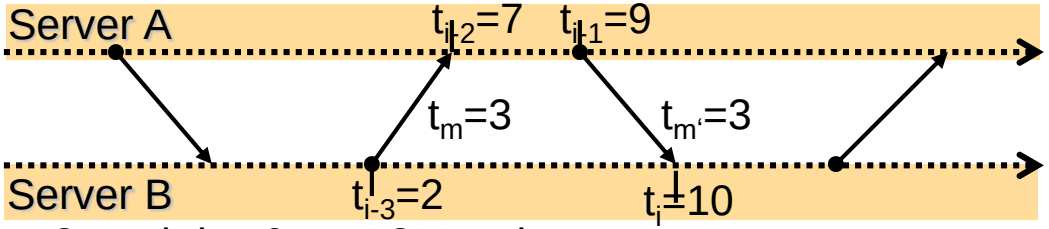


- d_i berechnet sich dann als $\mathbf{d_i}$ = (10 - 2) - (8 - 6)
  = (10 - 8) + (6 - 2) **= 6**
- o_i berechnet sich dann als $\mathbf{o_i}$ = ((6 - 2) - (10 - 8))/2
  = (4 - 2)/2 **= 1**
- o kann dann abgeschätzt werden durch

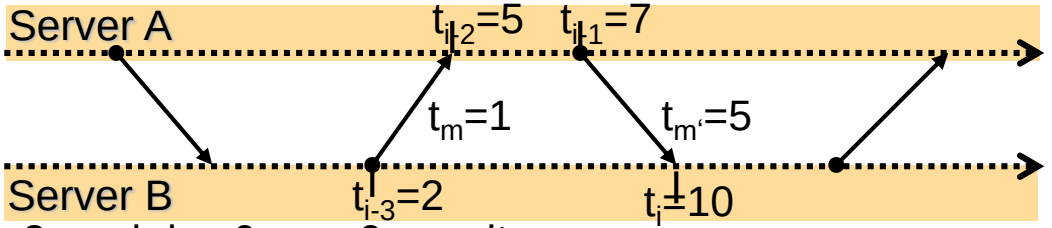
  1 - (6/2) = **-2** ≤ **o** ≤ **4** = 1 + (6/2)
- Die Uhr des Client würde nun auf 11 gestellt werden, womit die neue Abweichung **o = 1** wäre.

Beispiel: NTP Details

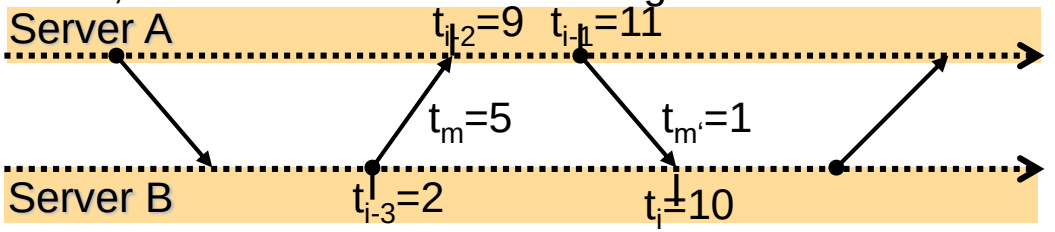
- Seien nun $\mathbf{t_m = t_{m'} = 3}$ (o = 2):



- Es ist dann t_i=10;t_{i-1}=9; t_{i-2}=7;t_{i-3}=2, und d_i **=** 6; o_i = 2, somit
  2 - 3 = **-1 ≤ o ≤ 5** = 2 + 3
- Die Uhr würde auf 12 gestellt werden , womit die neue Abweichung **o = 0** wäre.
- Seien nun $\mathbf{t_m = 1\ t_{m'} = 5}$ (o = 2):

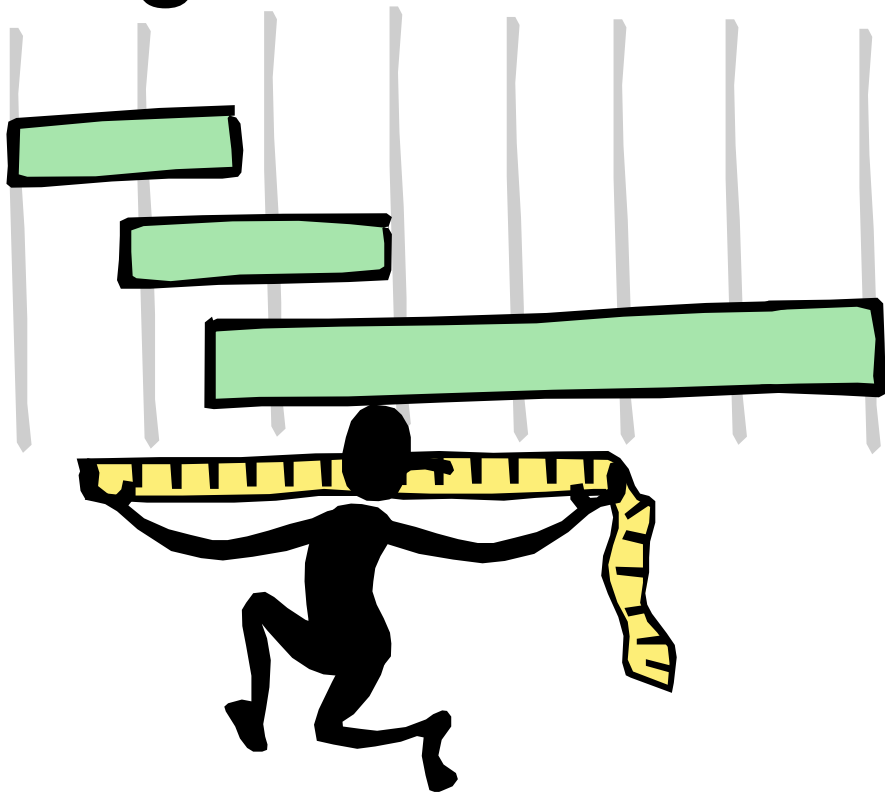


- Es ist dann t_i=10;t_{i-1}=7; t_{i-2}=5;t_{i-3}=2, und d_i **=** 6; o_i = 0, somit
  0 - 3 = **-3 ≤ o ≤ 3** = 0 + 3
- Die Uhr würde auf 10 gestellt werden , womit die neue Abweichung **o = 2** wäre.
- Seien nun $\mathbf{t_m = 5\ t_{m'} = 1}$ (o = 2):



- Es ist dann t_i=10;t_{i-1}=11; t_{i-2}=9;t_{i-3}=2, und d_i **=** 6; o_i = 4, somit
  4 - 3 = **1 ≤ o ≤ 7** = 4 + 3
- Die Uhr würde auf 14 gestellt werden , womit die neue Abweichung **o = -2** wäre.

Logische Uhren

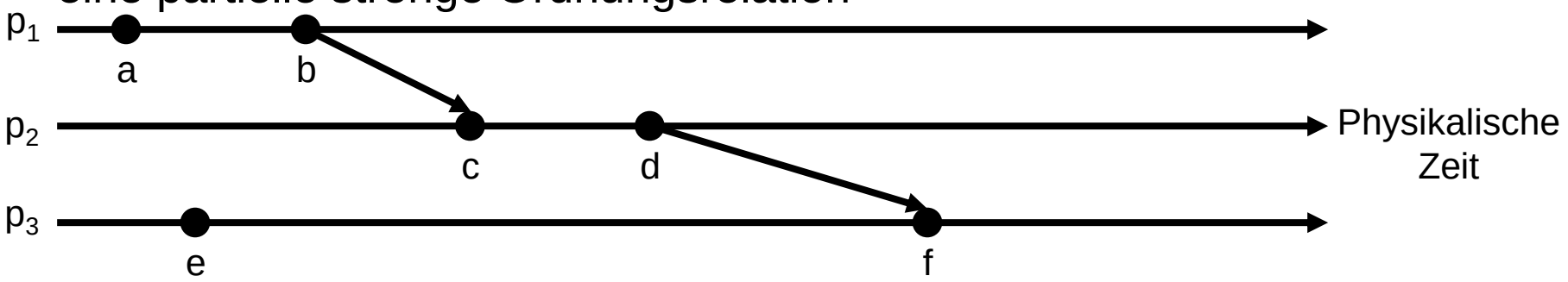


Logische Uhren

- **Konzept** einer logischen Uhr, bzw. einer logischen Zeit.
  - Ordnung der Ereignisse in relativer Lage (statt absoluter Lage)
  - bis auf Instruktionsgranularität möglich
  - Skalierung der Zeitachse ist beliebig
  - nur die Ereignisse, für die eine Zeitaussage notwendig ist
- **Eigenschaften** der logischen Zeit
  - **Kausale Abhängigkeiten** sollen **korrekt** berücksichtigt werden
  - Ereignisse aus unabhängigen Abläufen müssen nicht relativ zueinander geordnet werden.
- Die **Lösung** dafür (relative Lage):
  - Wenn zwei Ereignisse **im selben Prozess** stattfinden, dann fanden sie in der **Reihenfolge ihrer Beobachtung** statt.
  - Wenn eine Nachricht zwischen zwei Prozessen ausgetauscht wird, dann ist das **Sendeereignis** immer **vor** dem **Empfangsereignis**.

Lamport Zeit: Happened-Before Relation

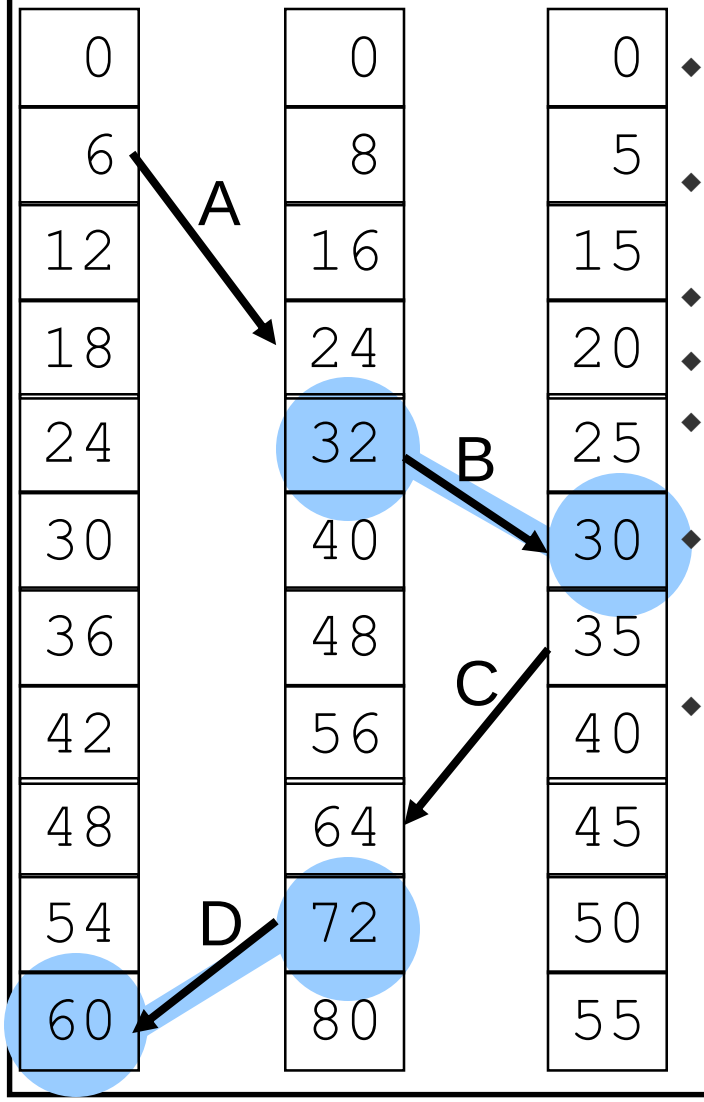
- Aus diesen Beobachtungen entwickelte Lamport 1978 die **Happened-Before Relation** „→“oder auch die „*relation of causal ordering*“:
  - Sind `a` und `b` Ereignisse im gleichen Prozeß p_i und **`a` ereignet sich vor `b`**, dann gilt: `a`→`b` (*`a` happened before `b`*) bzw. für das gesamte System **`a`→`b`**
  - Ist `send(m)` ein Sendeereignis für die Nachricht `m` in einem Prozeß p_i und `receive(m)` das zugehörige Empfangsereignis in einem anderen Prozeß p_j, dann gilt: **`send(m)`→`receive(m)`**.
  - → ist irreflexiv, asymmetrisch und transitiv und beschreibt damit eine **strenge Ordnungsrelation**
  - Zwei Ereignisse `a` und `b`, die nicht in der → Relation stehen, werden als **nebenläufig** (concurrent) bezeichnet und notiert mit `a||b`. → bildet damit eine partielle strenge Ordnungsrelation



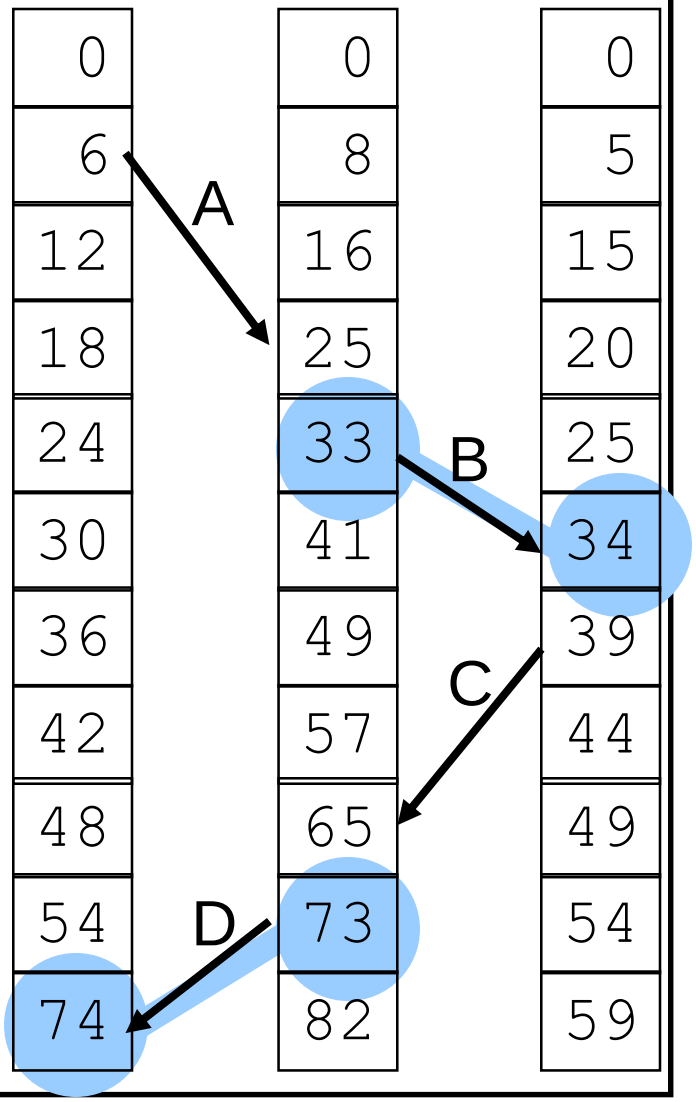
Lamport Zeit: Algorithmus

- Jeder Prozess p_i hat eine logische Uhr, die beim Auftreten eines Ereignisses `a` abgelesen wird und den **Zeitstempel** $C_i(a)$ liefert.
- Dieser Wert muss so angepasst werden, dass er als `C(a)` eindeutig im ganzen verteilten System ist.
- Ein Algorithmus, der die logischen Uhren entsprechend richtig stellt, muss folgendes umsetzen: Wenn $a \rightarrow b$ dann $C(a)<C(b)$
- **Lamport-Algorithmus** (hier beschrieben für den Prozeß p_i):
  - *Initalisierung:* $C_i := 1;$
  - *lokales Ereignis*: $C_i := C_i + 1;$
  - *Sendeereignis*: $C_i := C_i + 1;$ `send(message,`C_i`);`
  - *Empfangsereignis*: `receive(message,`C_k`);` $C_i := \max(C_i, C_k) + 1;$

Lamport Zeit: Beispiel



- Drei Prozesse mit unterschiedlichen Taktraten
- Lamport‘s Algorithmus löst das Problem
- **Links**: Ohne Lamport-Zeit
- **Rechts**: Mit Lamport-Zeit
- Wenn `a → b` dann `C(a)<C(b)`
- Keine Umkehrung: Aus `C(a)< C(b)` **folgt nicht** `a → b`
- Die Lamport-Zeit ist zwar konsistent mit der Kausalität, d.h. sie widerspricht ihr nicht, aber sie charakterisiert sie nicht.



Vektorzeit

- **Ziel**: Charakterisierung der Kausalität, d.h. eine Zeitrechnung, die äquivalent zur Kausalität ist: „$VT(a) < VT(b)$ ***genau*** *dann wenn* $a \rightarrow b$“
- **Konzept**, entwickelt von Mattern 1989:
  - Eine **Vektoruhr** VT für ein System ist ein **Array der Länge N**. Jeder Prozeß P_i führt seine eigene Vektoruhr VT_i.
  - VT(a) bezeichnet den Zeitstempel des Ereignisses a.
  - $VT_i[i]$ gibt die Anzahl der bisherigen Ereignisse in Prozeß P_i an.
  - $VT_i[k]$, $i \neq k$ gibt die Anzahl der Ereignisse in Prozeß P_k an, die den Prozeß P_i möglicherweise beeinflusst haben. Es gilt: $VT_i[k] \leq VT_k[k]$.
- **Definition**: Vektorzeit. a und b seien Ereignisse. Dann gilt
  - **a = b** genau dann, wenn für alle i gilt: VT(a)[i] = VT(b)[i].
  - **a ≤ b** genau dann, wenn für alle i gilt: VT(a)[i] ≤ VT(b)[i].
  - **a < b** (a vor b) genau dann, wenn für alle i gilt: VT(a)[i] < VT(b)[i].
  - **a||b** (a nebenläufig zu b) genau dann, wenn weder a ≤ b noch b ≤ a gilt.

Vektorzeit: Algorithmus

- **Initialisierung**:

  $\forall k: VT_i[k] := 0;$

- **lokales Ereignis**:

  $VT_i[i] := VT_i[i] + 1;$

- **Sendeereignis**:

  $VT_i[i] := VT_i[i] + 1;$

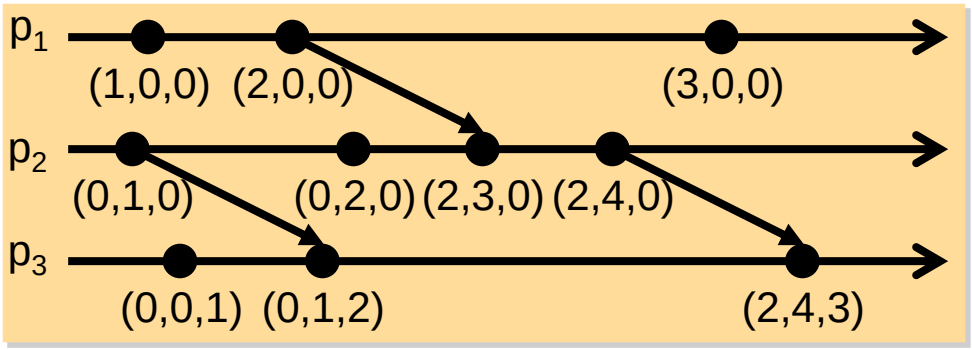
  $send(message, VT_i);$

- **Empfangsereignis**:

  $VT_i[i] := VT_i[i] + 1;$

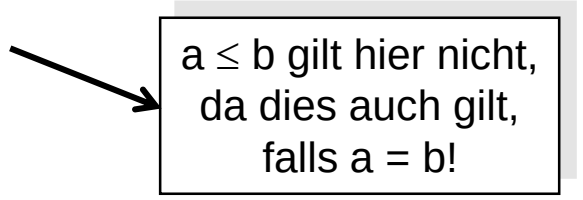
  $receive(message, VT_k);$

  $\forall j: VT_i[j] := max(VT_i[j], VT_k[j]);$



Vektorzeit: Eigenschaften

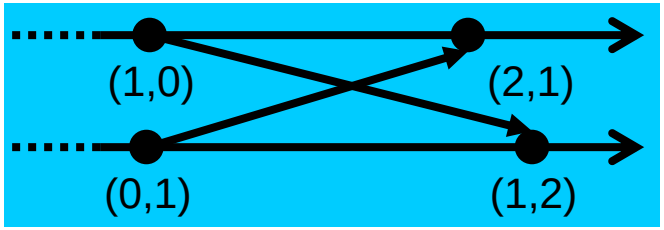
- $<$ ist irreflexiv, asymmetrisch und transitiv und beschreibt damit eine (partielle) **strenge Ordnungsrelation**;
- $\leq$, da reflexiv und antisymmetrisch, ist nur **Ordnungsrelation**.
- $||$ ist nur symmetrisch und symbolisiert die **partielle Ordnung**: a||b heißt, a und b stehen in keinem kausalen Zusammenhang.
- **Interpretation** der Relation:
  - a = b: a und b beschreiben das selbe Ereignis
  - $a \leq b$: a liegt vor b *oder* ist gleich b, beide haben ggf. bzgl. der Ereignisse eines anderen Prozesses den gleichen Kenntnisstand.
  - $a < b$: a liegt vor b, und b hat bei allen Prozessen den aktuelleren Kenntnisstand.
- **Zusammenhang** zwischen Lamport's $\rightarrow$ und **<** der Vektorzeit:
  - $a \rightarrow b \Leftrightarrow VT(a) < VT(b)$
  - $a \,||\, b \Leftrightarrow VT(a) \,||\, VT(b)$



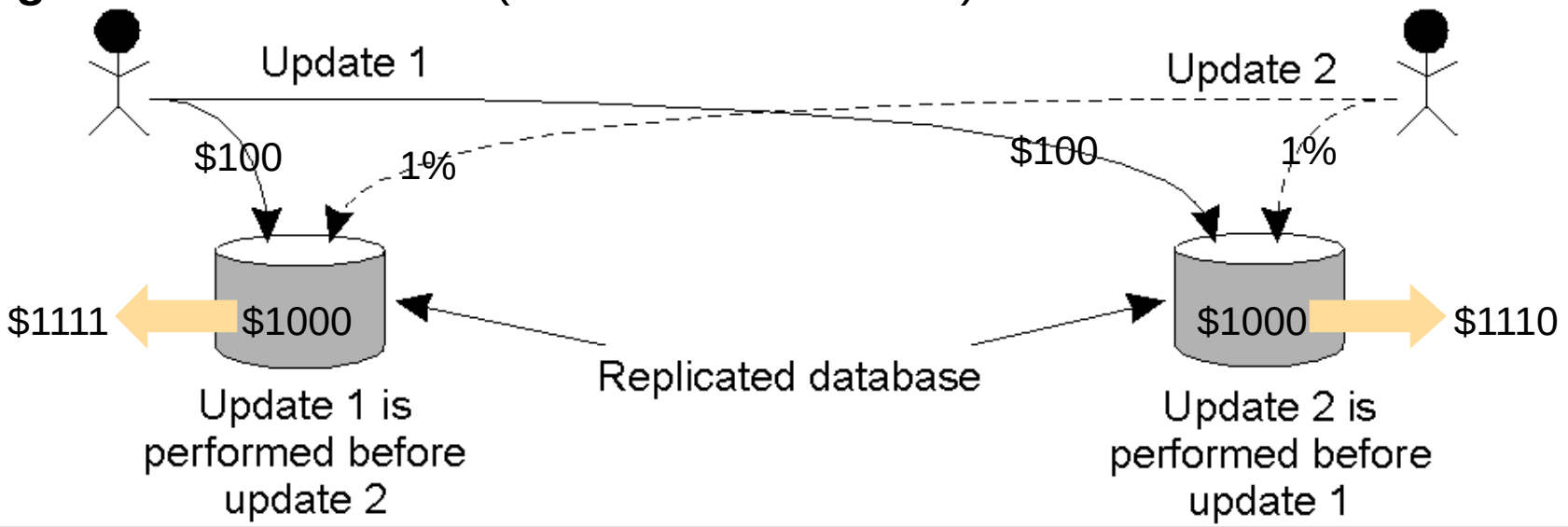
Vektorzeit: Eigenschaften

- Die Vektorzeit **charakterisiert die Kausalbeziehung** in dem Sinne, daß Ereignisse, die in einer Kausalrelation stehen, auch bezüglich der Vektorzeit geordnet sind, **und** daß bezüglich der Vektorzeit geordnete Ereignisse auch in einer Kausalrelation stehen.
- Die Vektorzeit bildet die aufgrund des lokalen Wissens „**beste Schätzung**“ der globalen Reihenfolgeverhältnisse.
- Die Vektorzeit liefert **nicht mehr Informationen** über zeitliche Zusammenhänge, als Lamport‘s Zeit!
- **||** sollte als „**steht in keinem kausalen Zusammenhang**“ gelesen werden, da „ist nebenläufig zu“ manchmal den Gedanken aufkommen lässt, die Ereignisse würden zur gleichen Zeit ablaufen.

Partielle kausale Ordnung: Problem



- Zwei Kopien der selben Datenbank
  - Kunde überweist $100 auf sein $1000-Konto (**update1**).
  - **Gleichzeitig** (nebenläufig) überweist ein Bankangestellter 1% Zinsen (**update2**).
- Die Operationen werden per Multicast an beide DBS geschickt und kommen dort in unterschiedlicher Reihenfolge an.
- **Ergebnis**: Inkonsistenz ($1111 versus $1110)

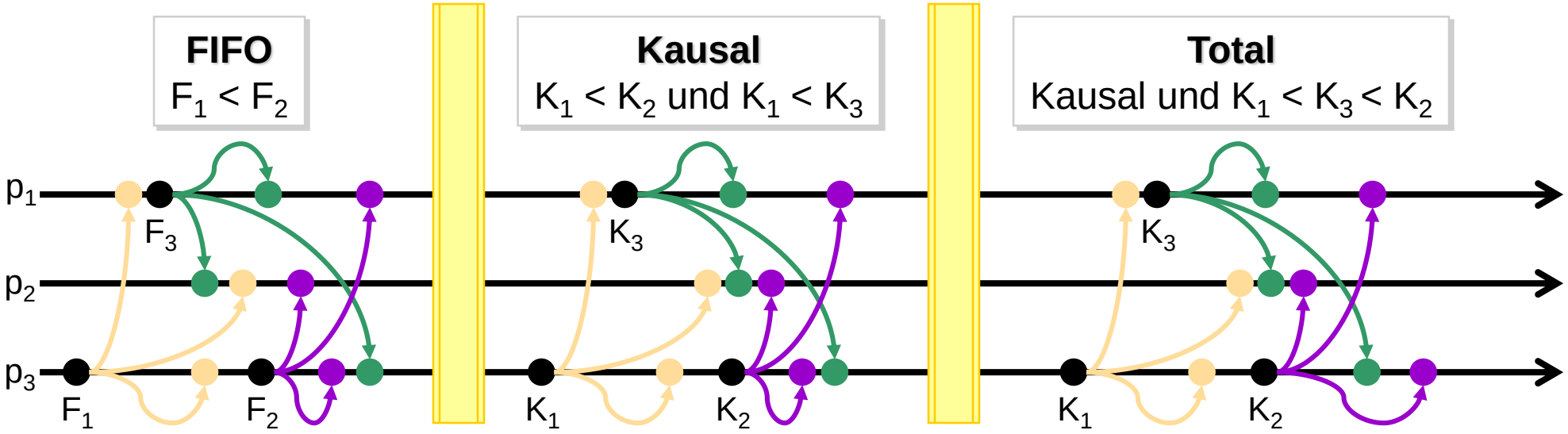


Beispiel: Multicast

- **Anforderung**: alle Replikate müssen immer im gleichen Zustand sein
- **Lösung**: total geordneter Multicast, d.h., alle Nachrichten werden bei allen Empfängern in derselben Reihenfolge ausgeliefert.
- **Annahmen**:
  - Reihenfolgenerhaltung
  - Nachrichten gehen nicht verloren
- **Implementierung** z.B.:
  - **FIFO Ordnung**, d.h. alle Nachrichten eines Senders kommen bei der Gruppe in der gesendeten Reihenfolge an: Jeder Sender verwendet (prozessspezifische) Folgenummern für seine Nachrichten.
  - **Kausale Ordnung**, d.h. FIFO Ordnung und kausal abhängige Nachrichten kommen in der richtigen Reihenfolge an**:** Verwendung kausalspezifischer Folgenummern z.B. durch Lamport-Zeit.
  - **Totale Ordnung**, d.h. kausale Ordnung und alle anderen Nachrichten werden von allen Mitgliedern der Gruppe in der gleichen Reihenfolge empfangen**:** Verwendung gruppenspezifischer Folgenummern.

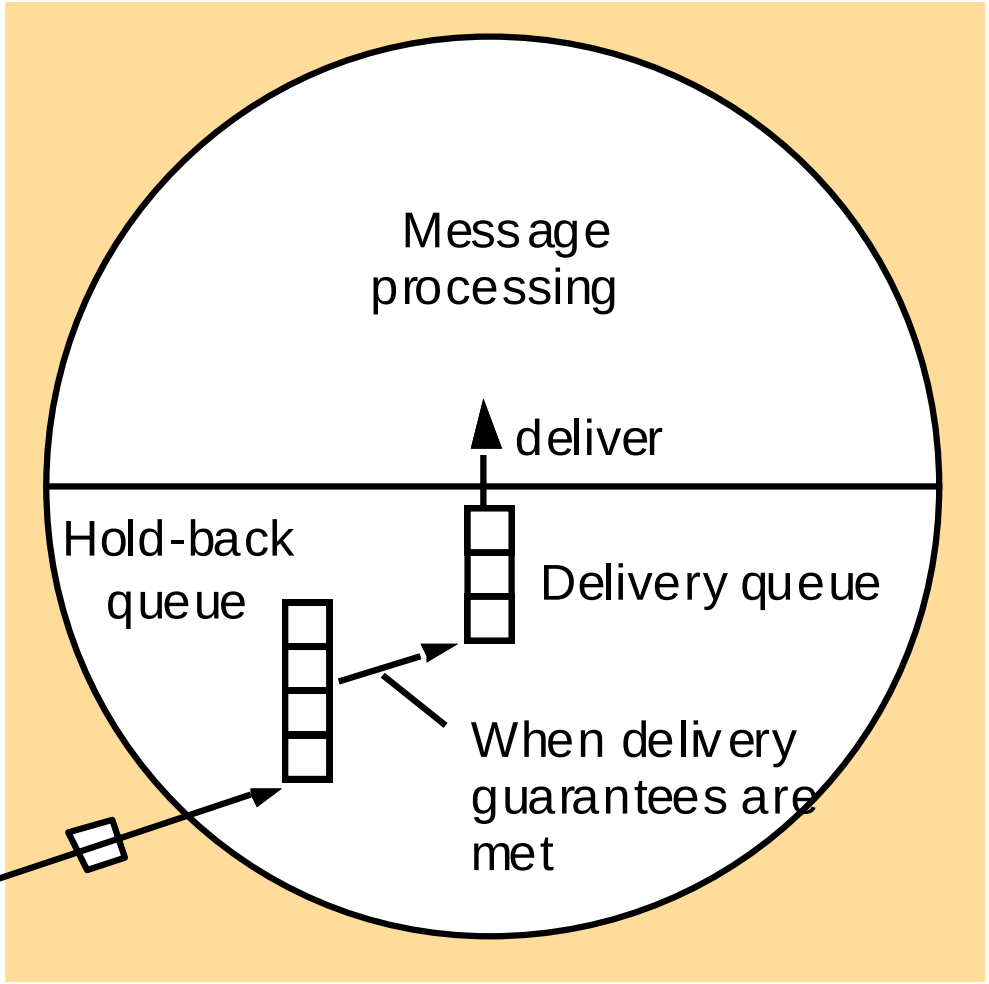
Beispiel: Multicast i.allg.

- **Gruppenkommunikation** (Broadcast: Nachricht an alle!)
- **Zuverlässigkeitsgrade**: Seien N-Mitglieder in der Gruppe registriert.
  - *keine Zuverlässigkeit*: 0 bis N haben die Nachricht erhalten
  - *k-zuverlässigkeit*: mindestens k haben die Nachricht erhalten
  - *atomar*: alle haben die Nachricht erhalten oder keiner
- **Ordnungsgrade**: neben diesen drei gibt es auch **keine Ordnung**



Beispiel: Implementierung

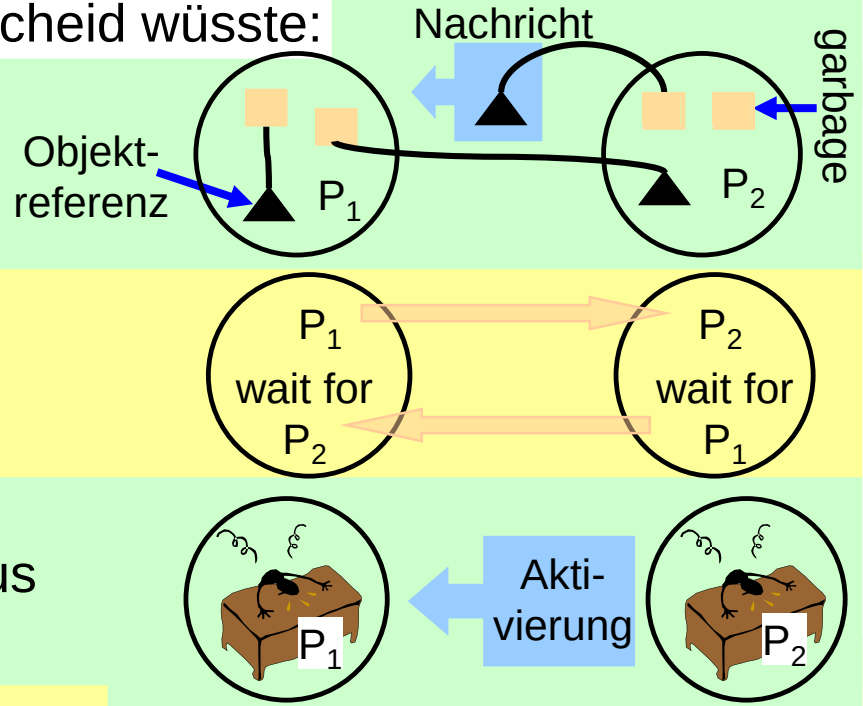
- Jeder Prozess hat eine lokale Warteschlange für eingehende Nachrichten, geordnet nach Nachrichtennummern, die **Hold-Back Queue**. Hierin enthaltene Elemente dürfen nicht ausgeliefert werden.
- Außerdem hat jeder Prozess eine **Delivery-Queue**. Hierin enthaltene Elemente werden entsprechend ihrer Reihefolge an die Anwendungen ausgeliefert.



Globale Systemzustände

Globale Systemzustände

- Es gibt eine Reihe von Gelegenheiten, bei denen man gern über den Gesamtzustand des verteilten Systems Bescheid wüsste:
  - *Verteiltes garbage collection*:
    Ein Objekt ist „Müll“, wenn es keinerlei Verweise mehr darauf gibt.
  - *Verteilte Deadlock-Erkennung*:
    Ein Zyklus im „Warte auf“-Graphen.
  - *Verteilte Terminierungs-Erkennung*:
    Terminierung eines verteilten Algorithmus erkennen.
  - *Verteiltes Debugging*:
    Zur gleichen Zeit bestehende Variablenbelegungen abfragen. (z.B. Uhren-synchronisation. Fehlerverdacht: Abweichung δ wird nicht eingehalten.)



- Der **Gesamtzustand** des Systems besteht aus
  - Den **lokalen Zuständen** der Einzelkomponenten (der Prozesse) **und**
  - **Allen Nachrichten**, die sich zur Zeit in der Übertragung befinden.

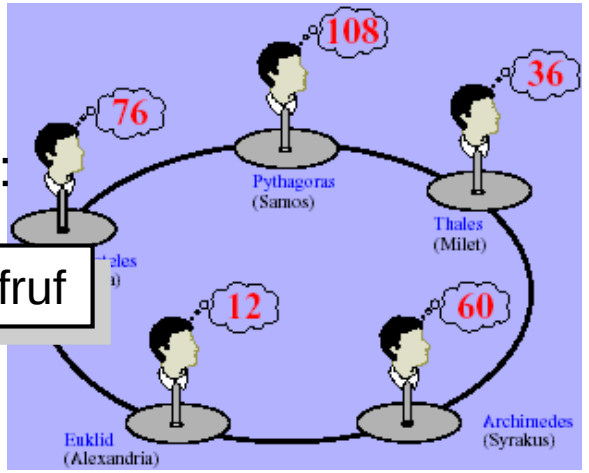
Beispiel: Verteilter Algorithmus

- **Satz von Euklid**: Der grösste gemeinsame Teiler (ggT) zweier positiver ganzer Zahlen x, y (mit x≥y>0) ist gleich dem ggT von y und dem Rest, der bei ganzzahliger Division von x durch y entsteht.
- **Eigenschaften**:
  - Offenbar ist **ggT(x,x) = x** für alle x
  - Man setzt nun noch **ggT(x,0) := x** für al
  - Rekursive Realisierung: **ggt(x,y) := ggt(y,mod(x,y))**
- **Erweiterung**: **mod*(x,y)** := **mod(x-1,y)+1**
- **Verteilter Algorithmus**:
  - Jeder Prozeß P_i hat seine eigene Variable M_i.
  - ggT aller am Anfang bestehender M_i wird berechnet:

```
{Eine Nachricht <y> ist eingetroffen}
if y < Mi
   then Mi := mod(Mi-1,y)+1;
        send <Mi> to all neighbours;
fi
```

Noch nicht geklärt:
- Wie wird der Algorithmus gestartet ?
- Wie erkennt man die Terminierung ?
- Wo steht das Ergebnis ?

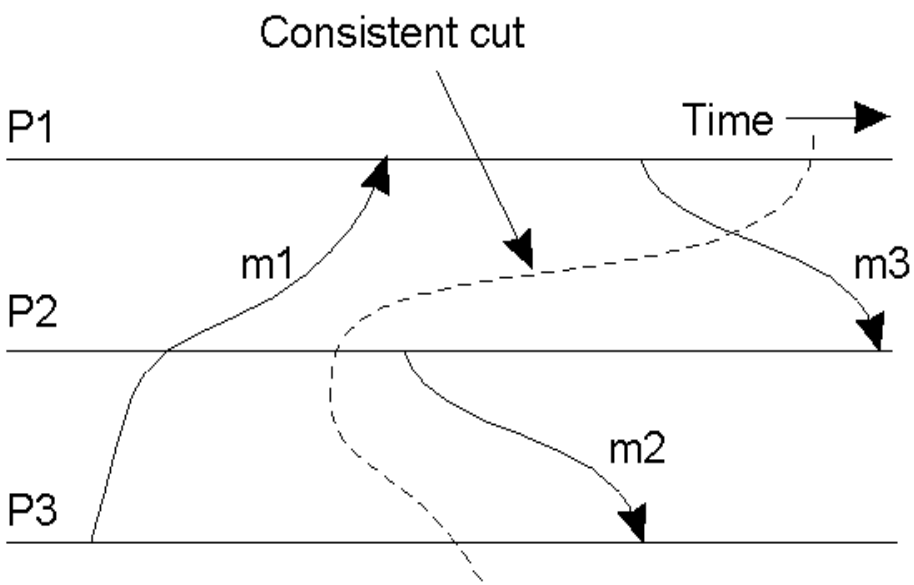
Entspricht rekursivem Aufruf



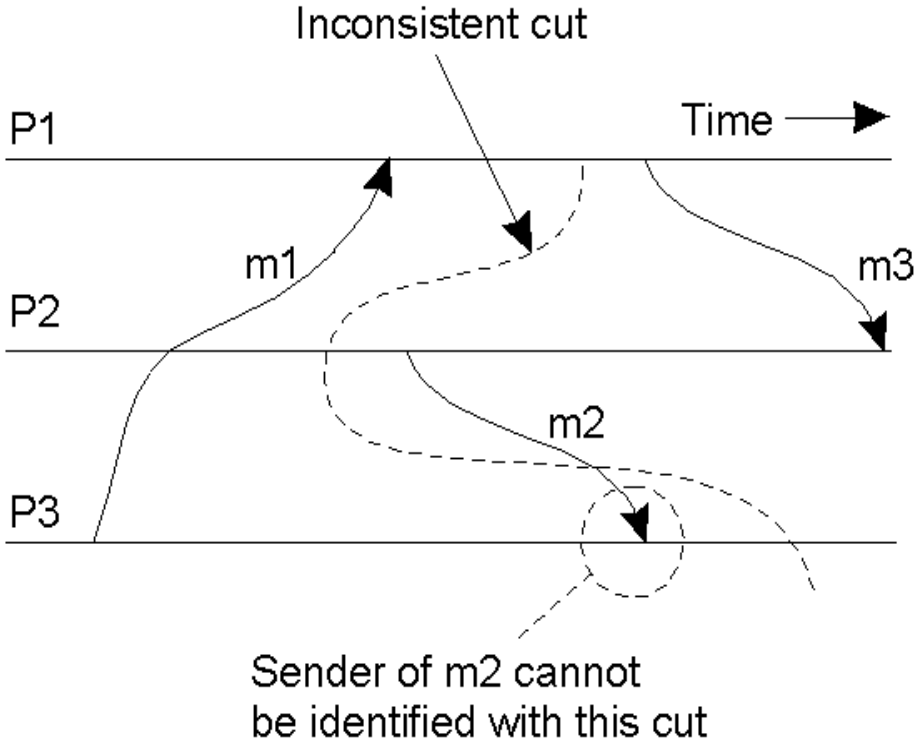
Verteilter Schnappschuß

- **Problem**: Den globalen Zustand exakt zur selben Zeit bestimmen zu können ist so unmöglich wie die perfekte Synchronisation von Uhren. Es läßt sich daher **kein globaler Zeitpunkt** festlegen, an dem alle Prozesse ihre Zustände festhalten sollen. Zudem besteht das Problem der „**verborgenen Nachrichten**“.
- **Lösung** von Chandy und Lamport (1985): Distributed Snapshot
  - ermittle einen globalen Zustand, der in Teilen zu **verschiedenen realen Zeiten** stattgefunden hat,
  - der aber auf jeden Fall **konsistent** ist
- **Konsistenz bedeutet** insbesondere: wenn festgehalten wurde, dass Prozess P eine Nachricht m von einem Prozess Q empfangen hat, dann muss auch festgehalten sein, dass Q diese Nachricht geschickt hat. Sonst kann das System nicht in diesem Zustand gewesen sein.
- **Definition** der Konsistenz **über** den sogenannten „**cut**“, der für jeden Prozess das letzte aufgezeichnete Ereignis angibt.

Der Cut: Beispiel



(a)



(b)

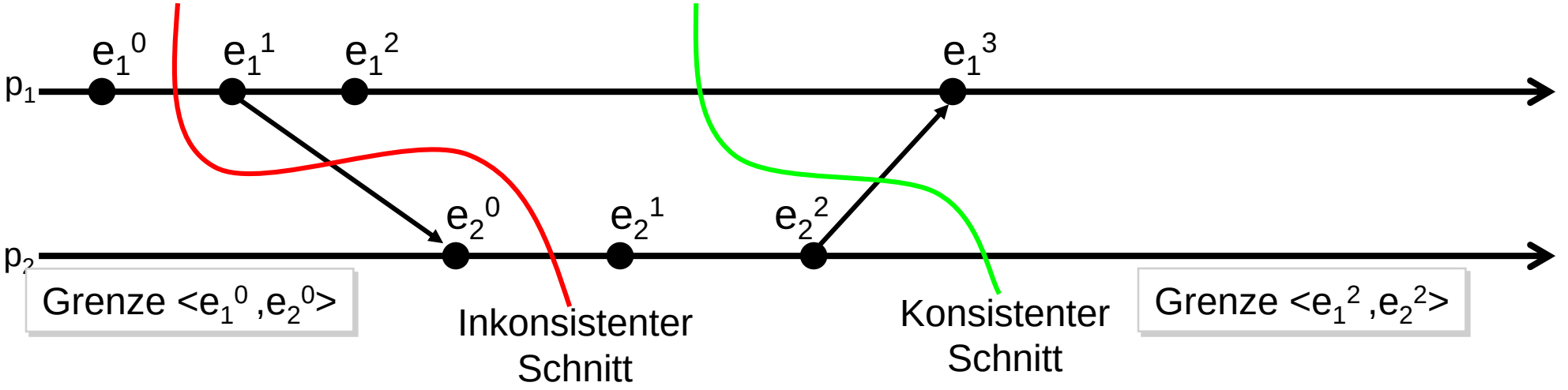
Der Cut: Definition

- **Gegeben** sei ein System VS von N Prozessen p_i (i=1,...,N).
- Der **globale Zustand** S = (s_1,...,s_N) des Systems, sei beschrieben durch Teilzustände s_i, die die lokalen Zustände der Prozesse p_i beschreiben.
- In jedem Prozess p_i findet eine **Serie von Ereignissen** e_i^z (z=1,2,...) statt, womit der Zustand eines jeden Prozesses mittels einer **History** charakterisiert werden kann:
  - history(p_i) = h_i= <e_i^0,e_i^1,e_i^2, ...>
  - h_i^k := <e_i^0,e_i^1,e_i^2, ..., e_i^k>, bezeichnet einen endlichen **Präfix** der History eines Prozesses
- Die **globale History** wird beschrieben durch:

  $H := h_1 \cup h_2 \cup ... \cup h_N$

Der Cut: Definition

- Ein **Cut** wird beschrieben durch: $C := h_1^{C_1} \cup h_2^{C_2} \cup ... \cup h_N^{C_N}$, wobei C_i jeweils der Index des zu C gehörenden Präfixes ist.
- Der Teilzustand s_i im globalen Zustand S, der dem Cut C entspricht, ist der von p_i, unmittelbar nach Ausführung von $e_i^{C_i}$ erreichte Zustand. Die Menge aller $e_i^{C_i}$. werden auch als **Grenze** von C beschrieben.



Der Cut: konsistenter Cut

- Ein **Cut** ist dann **konsistent**, wenn er für jedes Ereignis, das er enthält, auch alle Ereignisse enthält, die zu diesem Ereignis in der Happened-Before Relation stehen, formal:

$$\forall\, e_i^k \in C: e_g^h \rightarrow e_i^k \Rightarrow e_g^h \in C$$

- Ein **globaler Zustand** ist **konsistent**, wenn er mit einem konsistenten Cut korrespondiert.
- *Mit anderen Worten*: Bezeichnet man die Menge der Ereignisse in einem Cut C als **Vergangenheit** und alle anderen Ereignisse als **Zukunft**, so darf in einem konsistenten Zustand kein Empfangsereignis der Vergangenheit eine Nachricht von einem Sendeereignis aus der Zukunft erhalten!

Verteilter Schnappschuss: Wellenalgorithmus

Ein **Wellenalgorithmus** wird verwendet: Prozesse und Nachrichten sind **schwarz** oder **rot** gefärbt.

1. **Start**: Initiator färbt sich rot und speichert lokalen Zustand ab. Er versendet eine Kontrollnachricht über den Schnappschuss an alle anderen Prozesse.
2. **Ein Prozess erhält die Kontrollnachricht**: Er färbt sich rot und sendet seinen lokalen Zustand an den Initiator. Er sendet ab jetzt nur noch rote Nachrichten.
3. **Behandlung unterwegs befindlicher Nachrichten**:
   a) **Schwarze Nachrichten, die bei roten Prozessen ankommen** werden lokal verarbeitet und dem Initiator wird eine Kopie zugesendet (nicht der lokale Zustand, da der evtl. schon eine rote Nachricht „kennt“!)
   b) **Ein schwarzer Prozess erhält eine rote Nachricht**: Entweder gehe zu 2. oder blockiere solange die rote Nachricht, bis die Kontrollnachricht eingetroffen ist.

Verteilter Schnappschuss: Wellenalgorithmus

a) **Terminierungsproblem**: Wann ist letzte schwarze Nachricht beim Initiator eingetroffen ?

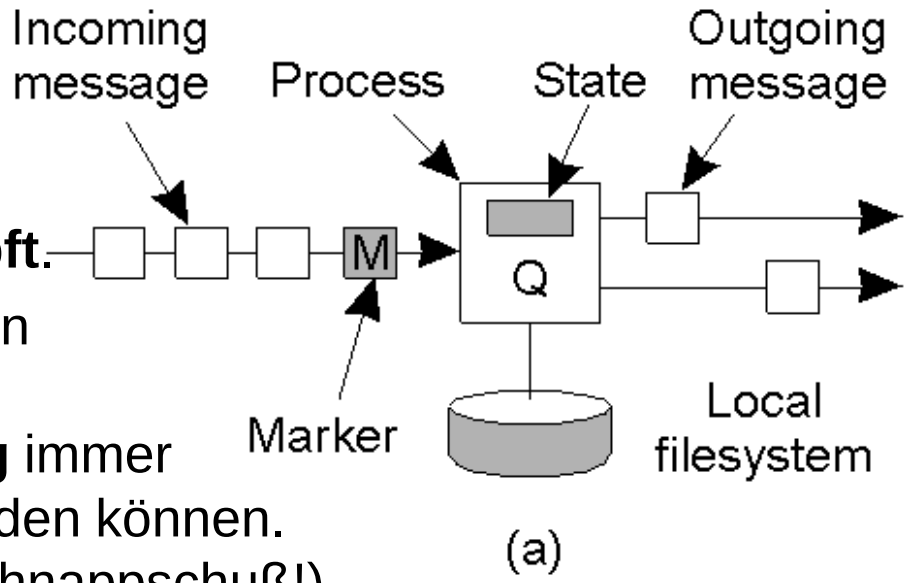
- Mögliche **Lösung**: „Defizitzähler" der Anzahl der (gesendeten – empfangenen) schwarzen Nachrichten werden als Teil des konsistenten Zustands von den einzelnen Prozessen mit gesendet.

b) **Mehrfachstart**: Starten mehrere Prozesse unabhängig voneinander den Algorithmus, entsteht **ein globaler Systemzustand**!

- Die Verwendung von mehr Farben hilft hier nicht, da bei dem Ablauf dann jeder Prozess diese Farben in der richtigen Reihenfolge durchlaufen muss. Das liegt vor allem daran, dass die Fehlersituation eines inkonsistenten Cut‘s zunächst zugelassen wird und dann korrigiert wird (siehe 3.b) des Algorithmus)
- Bei dem nachfolgenden Algorithmus können gleichzeitig unterschiedliche Schnappschüsse erstellt werden, da hier die Fehlersituation eines inkonsistenten Cut‘s durch die eigene Kommunikationsebene nicht zugelassen wird.

Schnappschuß: Chandy Lamport Algorithmus

- Hält **konsistente globale Zustände** fest
- **Konzept**:
  - Prozesse sind mittels **gerichteten Punkt-zu-Punkt Kanälen** verbunden.
  - Dieser Graph ist **vollständig verknüpft**.
  - Ein oder mehrere Prozesse starten den Algorithmus zur Feststellung eines Systemzustands, so dass **gleichzeitig** immer mehrere Schnappschüsse erstellt werden können. (Erfordert **eindeutigen Marker** pro Schnappschuß!)

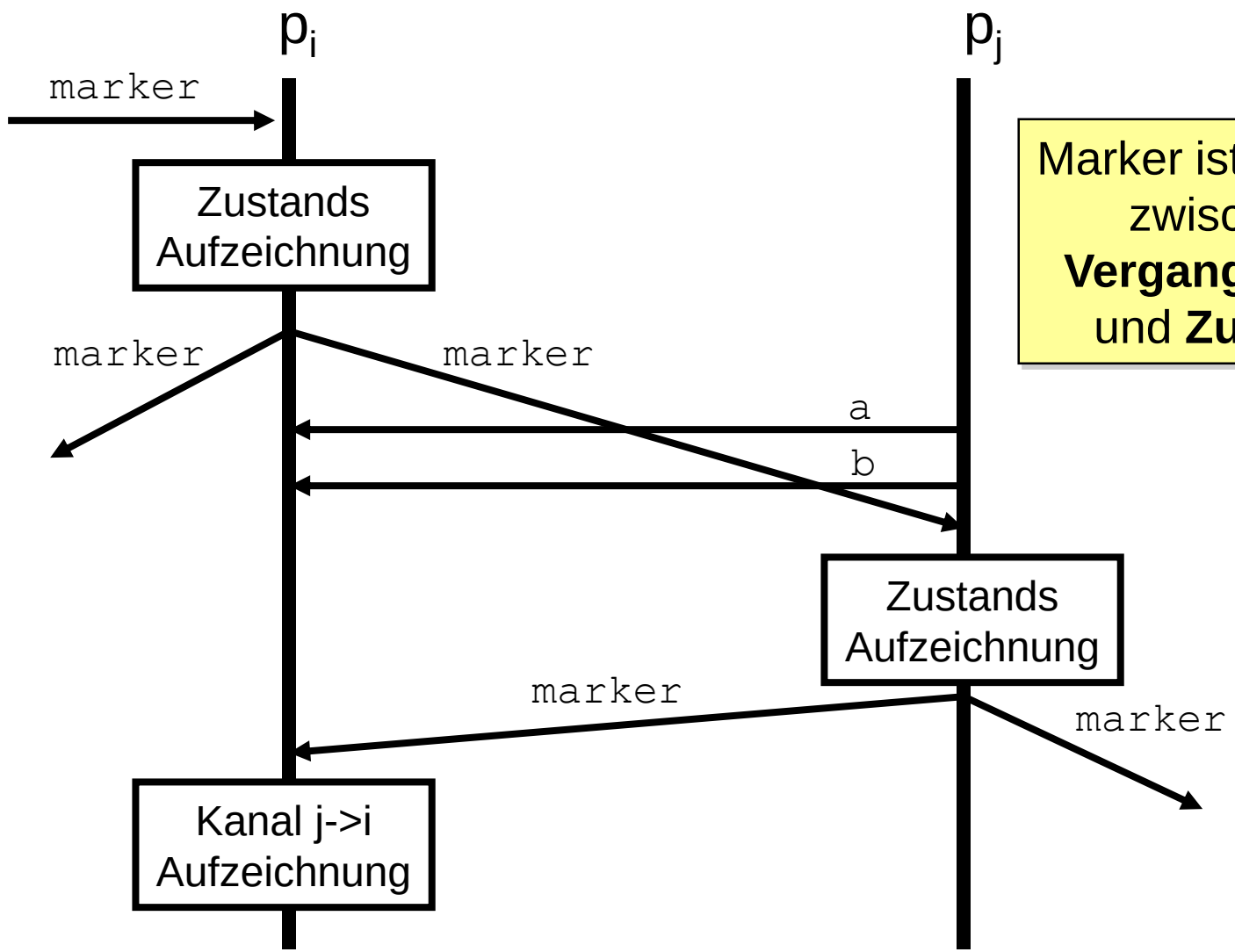


  - Das **System läuft** unterdessen **ungehindert weiter**.
  - Die Prozesse verständigen sich über **Markierungsnachrichten** über die Notwendigkeit der Speicherung eines Systemzustands.
  - Das System wird letztlich **„eingefroren“**, alle noch in den Kanälen befindlichen Nachrichten werden zugestellt.

Verteilter Schnappschuß: Algorithmus

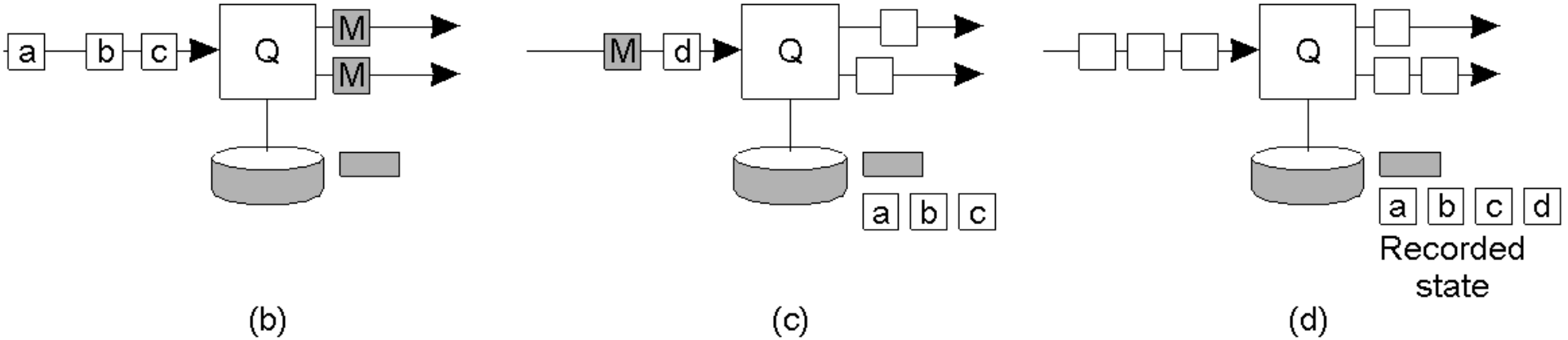
- Ein Prozeß, der einen Schnappschuß initiiert, folgt der Markierungs-Senderegel
- **Markierungs-Senderegel** für Prozeß p_i
  a) Nehme eigenen Zustand auf
  b) Versende Marker per Broadcast
  c) Nehme alle erhaltenen Nachrichten von allen eingehenden Kanälen auf
  a) und b) muß vor jedem anderen lokalen Ereignis geschehen
- **Markierungs-Empfangsregel** für Prozeß p_i
  - Wenn p_i seinen Zustand noch nicht aufgezeichnet hat (erster Marker wurde erhalten):
    - Zeichne eigenen Zustand auf
    - Zeichne Kanal, über den die Markernachricht kam, als leer auf.
    - Nehme alle erhaltenen Nachrichten von allen anderen eingehenden Kanälen auf
    - Versende Marker per Broadcast
  - Wenn p_i seinen Zustand schon aufgezeichnet hat:
    - Speichere den Zustand des Kanals, über den der Marker erhalten wurde
    - Stoppe die Aufzeichnung dieses Kanals

Verteilter Schnappschuß: Algorithmus Ablauf



Marker ist **Grenze** zwischen **Vergangenheit** und **Zukunft**

Verteilter Schnappschuß: Algorithmus Ablauf

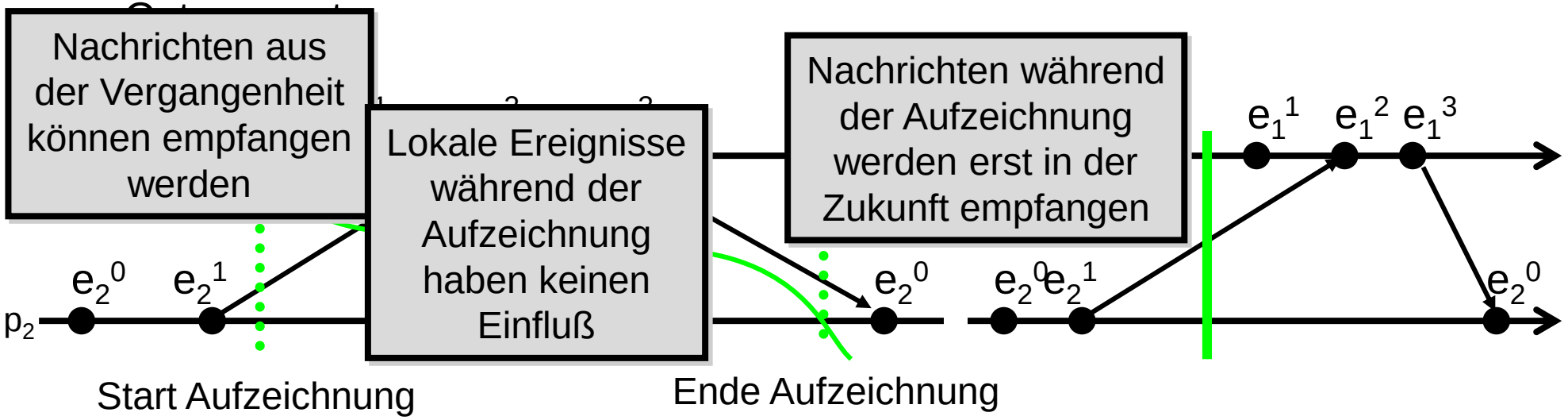


Marker-**Empfangsregel** für Prozeß Q

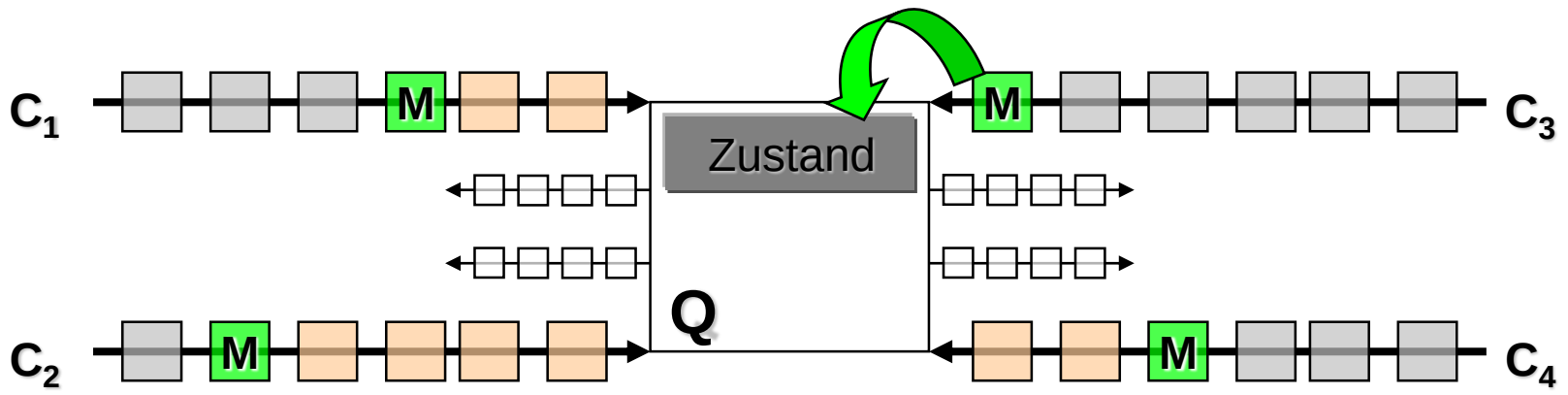
- (b): Prozess Q initiiert einen Schnappschuß und
  - hält seinen lokalen Zustand fest;
  - Marker-**Senderegel** für Prozeß Q: Q sendet über alle ausgehenden Kanäle einen Marker vor jeder anderen Nachricht;
- (c): Q hält alle ankommenden Nachrichten (auch anderer Eingangskanäle) fest.
- (d): Q erhält einen Marker auf seinem Eingangskanal und stoppt die Aufzeichnung dieses Kanals;

Verteilter Schnappschuß: Algorithmus Ende

- Wenn Q **auf allen Eingangskanälen einen Marker** erhalten und **verarbeitet hat** ist der Algorithmus für ihn beendet.
- Q sendet dann seinen **lokalen Zustand** sowie die jeweils **aufgezeichneten Nachrichten für alle Eingangskanäle** an den initiierenden Prozeß.
- Dieser wertet schließlich das Ergebnis entsprechend aus, analysiert also z.B. bestimmte Zustandsprädikate.
- Man kann beweisen, dass dieser Algorithmus immer einen konsistenten



Verteilter Schnappschuss: Der Cut



Nachricht im Kommunikationskanal zum Zeitpunkt des Schnappschuss

M Zeitpunkt des Schnappschuss

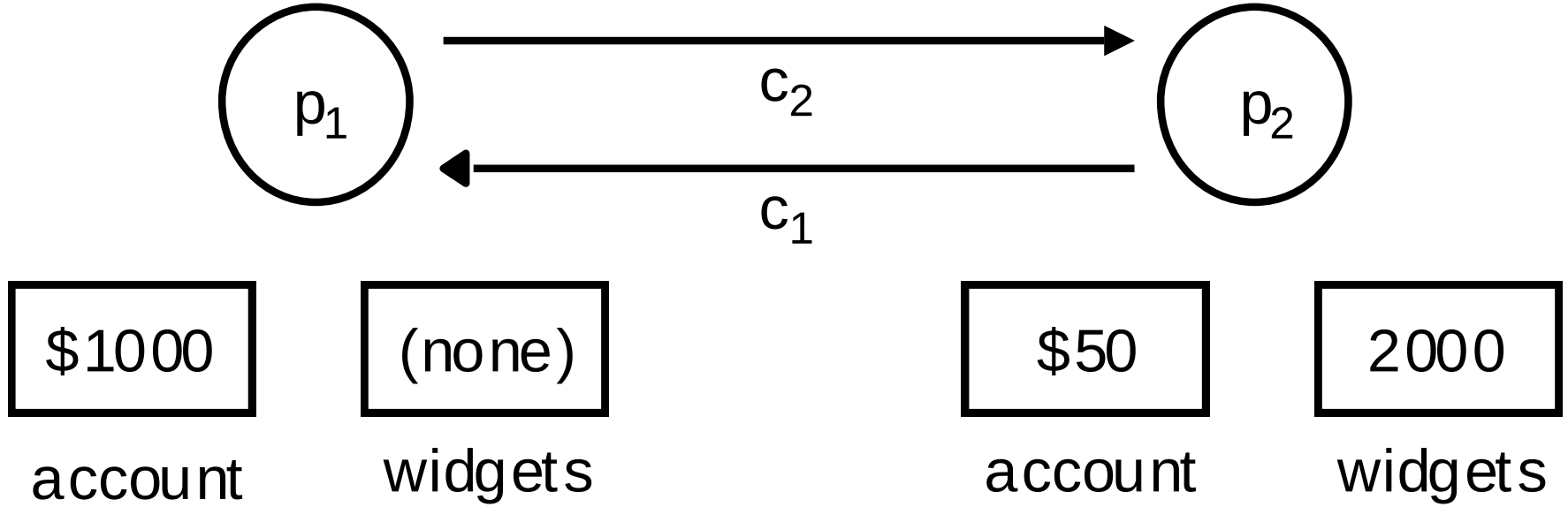
Nachricht im Kommunikationskanal nach dem Schnappschuss

Zustand Zustand von Prozess Q zum Zeitpunkt des Schnappschuss

Ausgehende Kanäle werden von anderen Prozessen protokolliert

Verteilter Schnappschuß: Beispiel

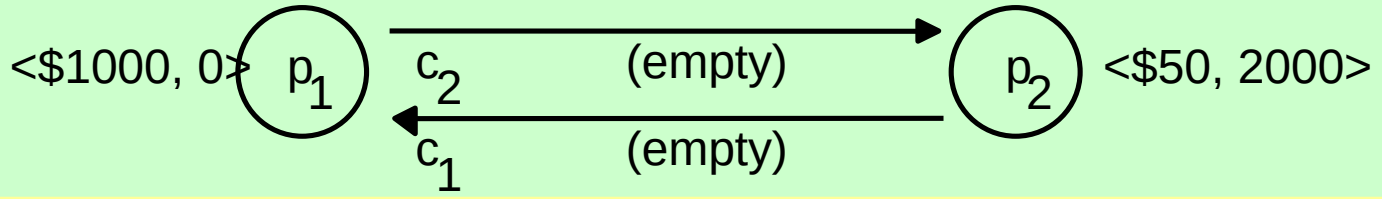
- 2 Prozesse p_1 und p_2 kommunizieren.
- p_2 verkauft Produkte zum Preis von \$10 das Stück.
- p_1 kauft Produkte.
- Der **Zustand** der beiden Prozesse wird durch die **Zahl der Produkte** und den **Kontostand** bestimmt.
- Zwischen den Prozessen gibt es **zwei Kommunikationskanäle** c_1 und c_2.



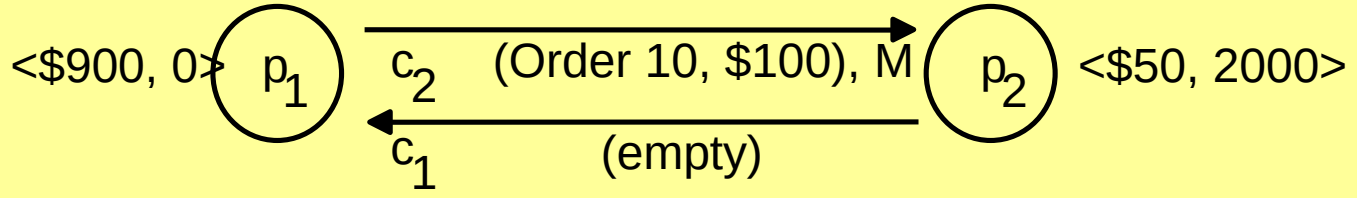
Verteilter Schnappschuß: Beispiel

- Sei der so gegebene Zustand s_0, in dem p_1 nun die Aufzeichnung initiiert.
- p_1 sendet eine Marker-Nachricht an p_2 vor der Kaufanfrage für 10 Produkte.

1. Global state S_0

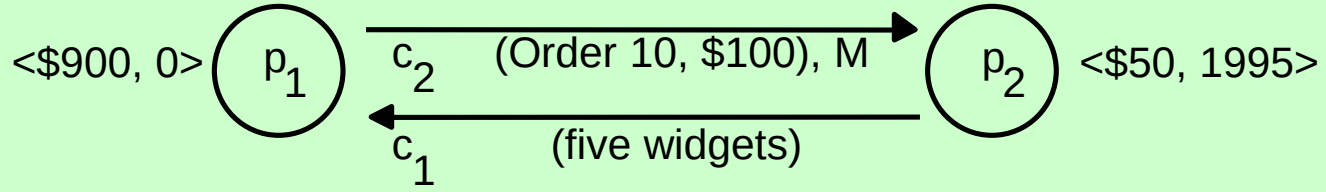


2. Global state S_1



- Bevor p_2 den Marker erhält, sendet er die Bestätigung über einen früheren Produktkauf an p_1. Dadurch ergibt sich Zustand s_2 wir folgt:

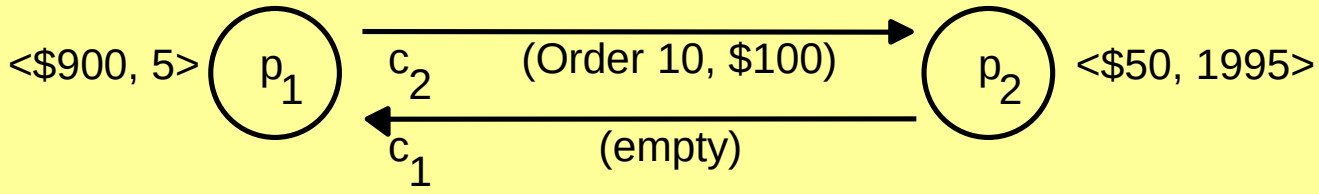
3. Global state S_2



Verteilter Schnappschuß: Beispiel

- Nun empfängt p_1 die Nachricht von p_2 und p_2 empfängt den Marker.
  - p_2 zeichnet den Zustand <50,$1995> auf. Schließlich schickt er selbst einen Marker an p_1.
  - p_1 zeichnet die Nachricht von p_2 auf.
- Nach Erhalt des Markers zeichnet p_1 den Zustand von c_1 auf (eine Nachricht). Resultierender Zustand:

4. Global state S_3

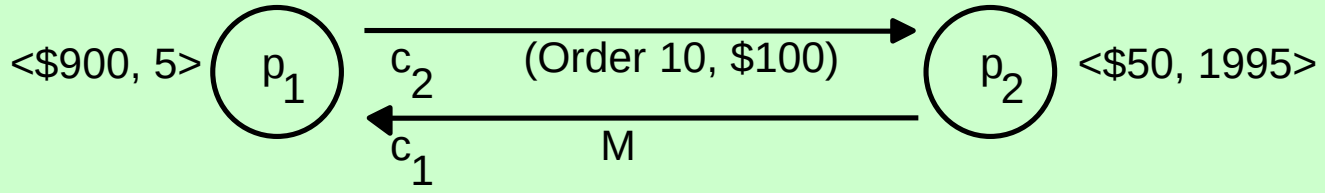


- Nach Erhalt des Markers ist der **aufgezeichnete Zustand** dann:
  - **Prozesse**: p1: <$1000,0>; p2: <$50,1995>;
  - **Kanäle**: c1: <five widgets>; c2: < >;
  - Dies entspricht keinem der wirklichen globalen Zustände, ist aber ein möglicher Zustand des Systems.

Verteilter Schnappschuß: Beispiel

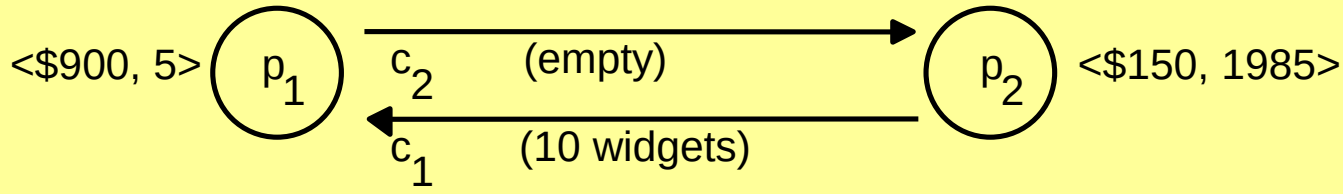
- Das Ende der Aufzeichnung: p_2 sendet eine Marker-Nachricht an p_1

5. Global state S_4



- Für p_2 ist die Beteiligung am Algorithmus beendet: Der lokale Zustand ist festgehalten und alle Eingangskanäle sind komplett aufgezeichnet.

6. Global state S_5



- Für p_1 ist der Algorithmus beendet: Der lokale Zustand ist festgehalten und alle Eingangskanäle sind komplett aufgezeichnet.
- Um den **Gesamtzustand** nun zu erhalten, sind zu dem Marker alle Informationen der beteiligten Prozesse einzusammeln.

Verteilter Schnappschuss: Beispiel