Verteilte Systeme
Betriebssysteme II

Kapitel 8: Synchronisation

Prof. Dr. Wolfgang Küchlin

Dipl.-Inform., Dr. sc. techn. (ETH)

Arbeitsbereich Symbolisches Rechnen
Wilhelm-Schickard-Institut für Informatik
Fakultät für Informations- und Kognitionswissenschaften

Universität Tübingen

Steinbeis Transferzentrum
Objekt- und Internet-Technologien (OIT)

Wolfgang.Kuechlin@uni-tuebingen.de
http://www-sr.informatik.uni-tuebingen.de

Einleitung

- Grundproblem: Verteilung in Raum und Zeit
- eng gekoppelte Systeme:
  Alle Prozesse haben dieselbe Systemzeit
  - Synchronisation über gemeinsamen Hauptspeicher
  - Gemeinsame Nutzung von Mutex-, Semaphor- und Condition-Variablen
- verteilte Systeme: Keine gemeinsame Systemzeit
  - spezielle Algorithmen für resultierende Probleme
- Problemklassen sind:
  1. Uhr-Synchronisation
  2. Gegenseitiger Ausschluss
  3. Wahlverfahren
  4. Atomare Transaktionen
  5. Verklemmungen

Einleitung

- Randbedingungen für verteilte Algorithmen
  1. Information über verschiedene Maschinen verteilt
  2. keine synchrone Systemzeit
  3. Entscheidungen der Prozesse auf Grund lokaler Information
  4. Vermeidung zentraler Abhängigkeiten

- Punkte 1 & 2:
  Wichtige Feststellungen über verteilte Systeme

- Punkte 3 & 4:
  Charakteristika guter, verteilter Algorithmen

Uhr-Synchronisation

- unabhängig laufende Uhren können nicht jederzeit völlig synchron sein
  - Laufzeit der Synchronisationspulse größer als 0
  - unabhängige Uhren driften immer (*clock skew*)

- Bedeutung global synchroner Zeit
  - Beispiel: `make`
    - compiliert nur, wenn die Quelldatei jünger als Objektcode
    - Situation: Editor und Compiler auf unterschiedlichen Maschinen mit unterschiedlicher Zeit
    - Problem: Alter Objektcode einer neu editierten Datei kann jüngeres Datum tragen als der später editierte Quellcode

Uhr-Synchronisations-Algorithmen

- internationaler Zeitstandard
  - Universal Coordinated Time - UTC) vom BIH (Bureau International de l'Heure) in Paris
- Ideal:
  Uhren in verteilten System empfangen UTC-Zeitimpuls
- Problem
  - bei räumlich weit verteilten Systemen und kleinen Zeitquanten (ns)
  - nicht alle Maschinen haben UTC-Empfänger

Uhr-Synchronisations-Algorithmen

- Architektur
  - Jede Maschine hat Uhr (*timer*), C, die sie H mal pro Sekunde inkrementiert
  - UTC Standardzeit t
  - lokale timer C_p der Maschine p mit Wert $C_p(t)$
  - Idealfall: $C_p(t) = t$ und $\frac{d}{dt} C_P = 1$
- jedes C_p *driftet* mit Maximalrate ρ, d.h. $\left|\frac{d}{dt} C_P\right| \leq 1 + \rho$
- Sollen Uhren maximal um δ Sekunden divergieren
  $\Rightarrow$ Synchronisation alle $\delta/2\rho$ Sekunden
  - denn

$$C_{P1}(t_0 + x) = t_0 \pm \rho \cdot x \quad \text{und} \quad C_{P2}(t_0 + x) = t_0 \pm \rho \cdot x$$

$$\Rightarrow \left|C_{P1} - C_{P2}\right| < 2\rho x \leq \delta \quad \text{und} \quad x \leq \frac{\delta}{2\rho}$$

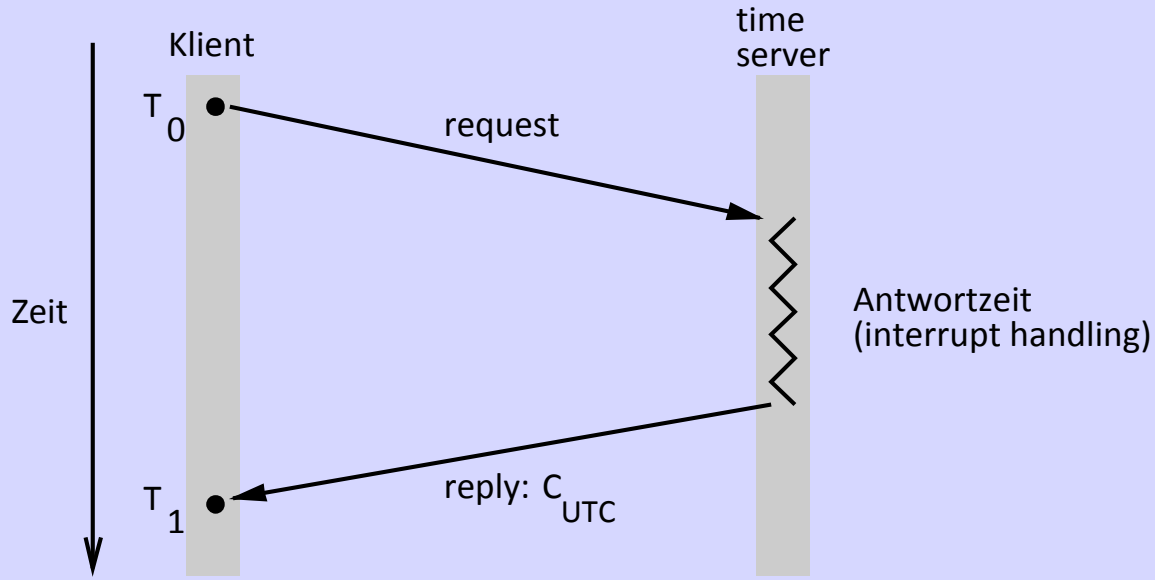
Cristian's Algorithmus (passiver time-server mit UTC)

- time-server (UTC Empfänger)
  - wird von allen anderen Maschinen periodisch nach der Zeit gefragt

- **Problem 1:**
  - Falls $C_{UTC} < C_{Klient}(T_1) \rightarrow$ langsame Anpassung
    - Uhren dürfen nie rückwärts gehen (wegen `make` etc)

Cristian's Algorithmus (passiver time-server mit UTC)

- **Problem 2:**
  - Wegen Signallaufzeit der Antwort ist C_{UTC} bei Ankunft im Klienten veraltet
  - Klient weiß: $T_1 - T_0 = t_{request} + t_{in} + t_{reply}$
  - wenn t_1 bekannt: $t_{reply} \approx \frac{T_1 - T_0 - t_{in}}{2}$
  - Schätzung durch mehrere Proben verfeinern



Uhr-Synchronisations-Algorithmen

- **Berkeley UNIX Algorithmus (aktiver time-server ohne UTC)**
  - **time daemon** fragt jede Maschine periodisch nach der Zeit
  - Mittlung der Antworten
  - Durchschnittszeit als neue Zeit für Clients
  - Dezentralisierung: Jeder schickt seine Zeit an jeden anderen und bildet Durchschnitt selbst
- **Verschiedene externe Quellen**
  - verschiedene UTC Empfänger im Netz
  - alle verschicken ihre Zeit und ein Intervall, innerhalb dessen die Zeit korrekt ist
  - Jeder empfangende Rechner errechnet sein Intervall als den Durchschnitt aller empfangenen Intervalle
    - Ausreißer, die Durchschnitt leer machen, werden ignoriert

Logische Uhren

- Anforderung: **Uhren *relativ* synchron**
  - Übereinstimmung der absoluten Zeit nicht so wichtig
  - Ereignis hat (irgendwann) *vor* einem anderen stattgefunden, wenn sein **Zeitstempel** kleiner ist als der andere
  - **logische Zeit** statt absolute UTC-Zeit
- Motivation
  - bei verteilter Berechnung tritt bei Ereignis e_g Fehler auf
  - Welche Ereignisse können zu Fehler geführt haben?
    → Nur Ereignisse, die logisch vor e_g stattgefunden haben
- Frage: Wie kann man relativ synchrone Zeitstempel berechnen?

Logische Uhren

- Def.: b ist **unmittelbar abhängig** von a (schreibe: **a** → **b**) gdw
  1. Falls a und b Ereignisse im gleichen Prozess sind und a unmittelbar vor b stattfindet, so gilt a → b
  2. Ist a das Ereignis des Sendens einer Nachricht *m* durch einen Prozess und b das Ereignis des Empfangens von *m* durch einen (anderen) Prozess, so gilt a → b

- Def.: Relation „**logisch abhängig**“ oder „geschieht logisch vor“ ist die (reflexive) transitive Hülle →* von →
  - →* ist gleichzeitig die von → induzierte Partialordnung

- Sind zwei Ereignisse in der Ordnung unvergleichbar so heißen sie **nebenläufig** (*concurrent*)
  - unvergleichbar: Es gilt weder a → b noch b → a

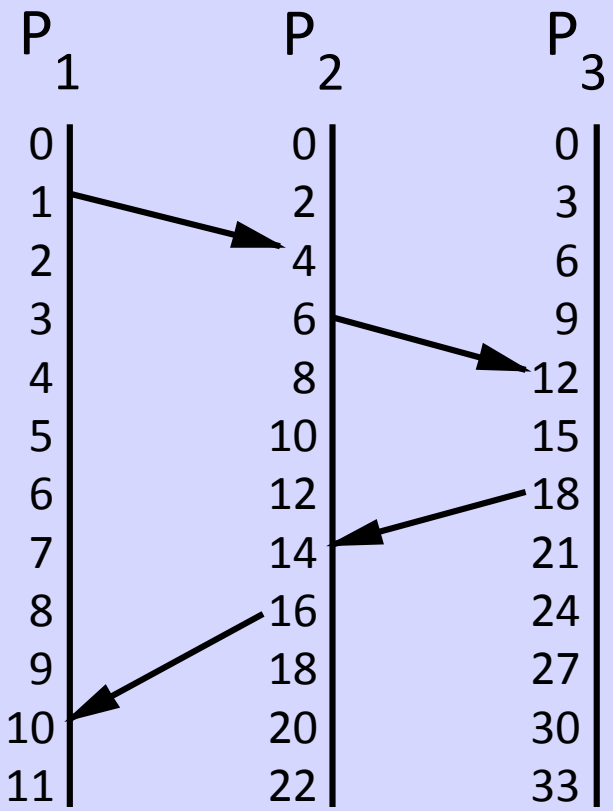
Logische Uhren

- Ziel: ”logische Uhr“ C, für die gilt $C(a) \leq C(b) \Leftrightarrow a \rightarrow^* b$
  - bzw. $C(a) < C(b) \Leftrightarrow a \rightarrow^+ b$
- **Lamports logische Uhren**: Ereigniszähler $C_L(a) < C_L(b) \Rightarrow a \not\rightarrow^+ b$
  - $a \rightarrow^+ b \Rightarrow C_L(a) < C_L(b)$ oder äquivalent
- **Vektoruhren** von Mattern und Fidge: Vektoren von Ereigniszählern
  - $a \rightarrow^+ b \Leftrightarrow C_V(a) < C_V(b)$

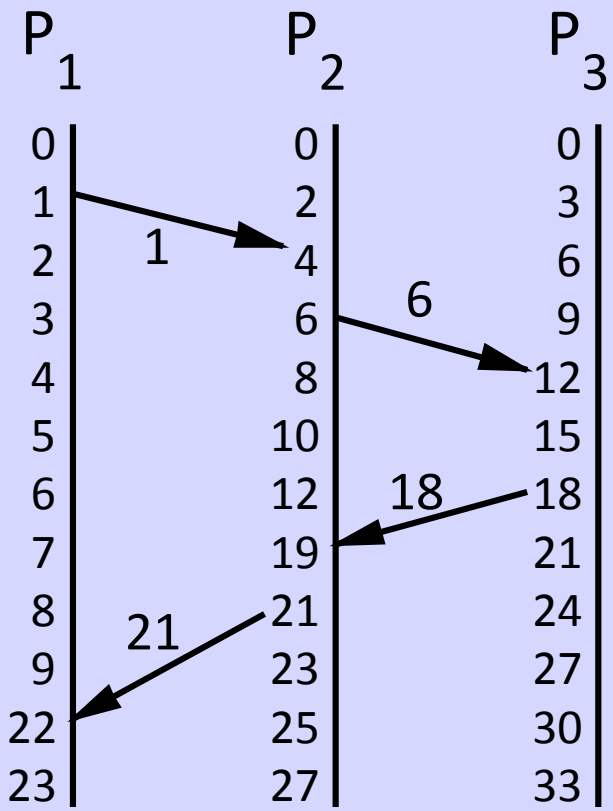
Regeln für Lamports logische Uhren

- Voraussetzung: Jeder Prozess P_i benutzt eigene Uhr C_i
- **Regel 1: $C_i(b) := C_i(a) + d; d > 0$**
  - C_i wird zwischen je zwei aufeinander folgenden Ereignissen a und b in Prozess P_i inkrementiert.
  - Falls $a \rightarrow^+ b$ in P_i so folgt also $C_i(b) > C_i(a)$
- **Regel 2: $C_j(b) = \max (C_j(b), t_m + d) ; d > 0$**
  - Versenden einer Nachricht (Ereignis a) mit Zeitstempel $t_m = C_i(a)$
  - Korrektur des Zeitstempels des Empfängers beim Empfang einer Nachricht (Ereignis b) mit Zeitstempel $t_m > 0$
- Es gilt: $a \rightarrow b \Rightarrow C_L(a) < C_L(b)$
  - Da < transitiv, gilt auch $a \rightarrow^+ b \Rightarrow C_L(a) < C_L(b)$
- Ausschluss absolut gleichzeitiger Ereignisse
  - logische Zeit (lexikographisch) durch Prozessnummer erweitern zu $C_i.i$
  - Für $i \neq j$, $C_i = C_j$, gilt dann $C_i.i \neq C_j.j$

Beispiel für Lamports logische Uhren



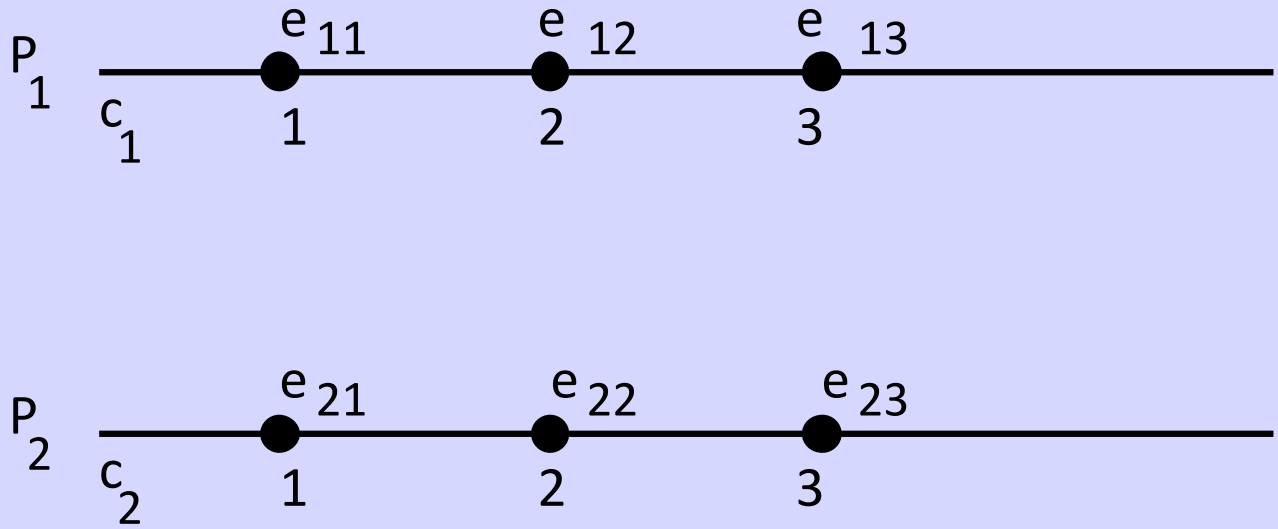
Uhren ohne Lamport



Lamports logische Uhren

Vektor-Uhren – Motivation

- Lamports Uhren spiegeln Ereignisabfolge nur grob wider
- Es gilt $C(a) < C(b) \Rightarrow b \not\rightarrow a$
  aber man erkennt nicht, ob $a \rightarrow b$
- Beispiel: $C(e_{22}) < C(e_{13})$ aber $e_{22} \not\rightarrow e_{13}$
  $C(e_{22}) < C(e_{23})$ und $e_{22} \rightarrow e_{23}$

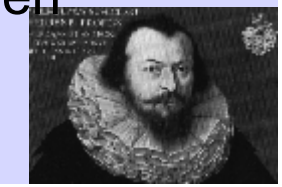


Vektor-Uhren – Implementierung

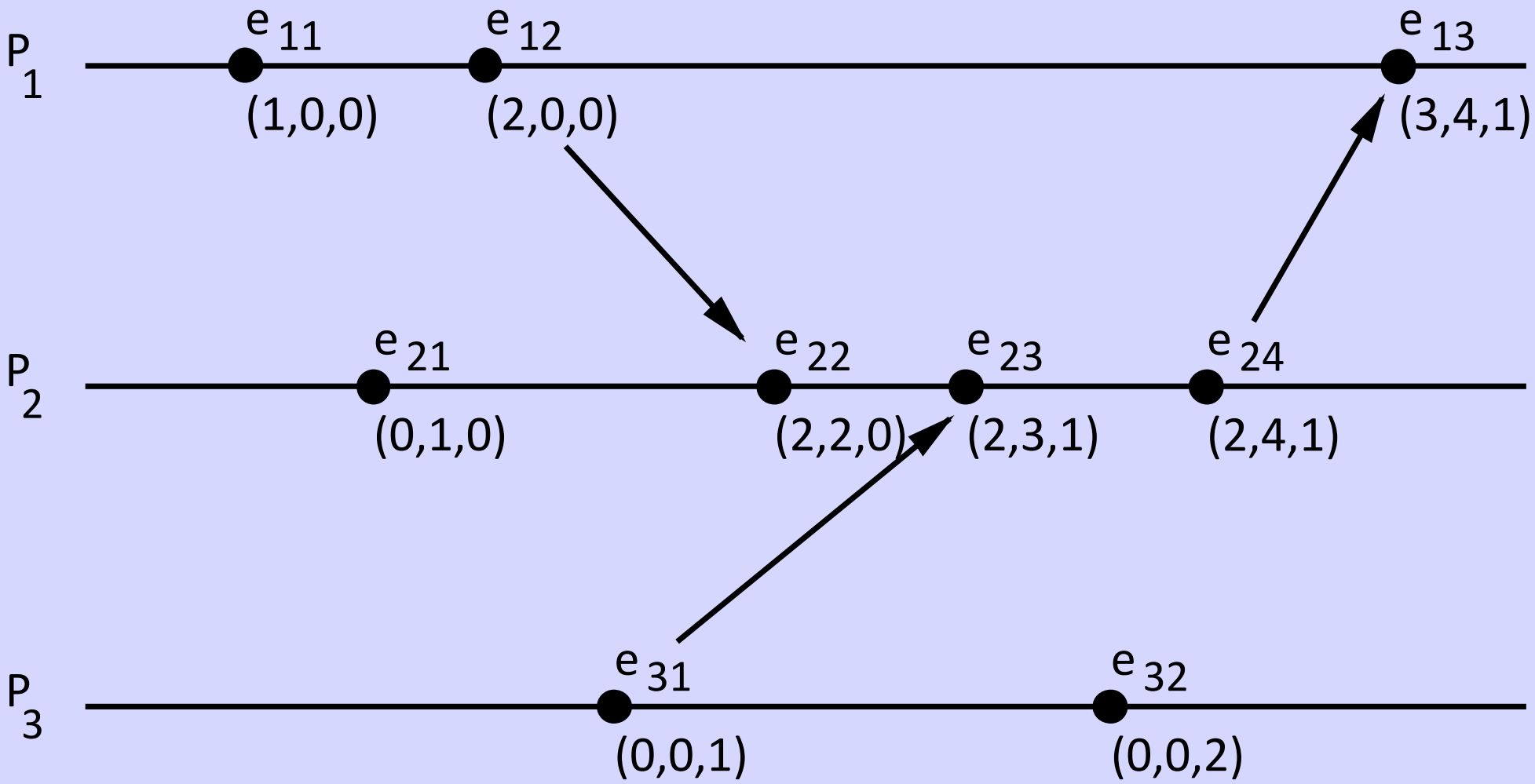
- Zeitstempel ist nicht einzelne Zahl, sondern Vektor
  - $C_i[\ ]$ – Vektor in Prozess i.
  - Komponente j stellt jeweilige „lokale Zeit“ in Prozessor j dar
  - $C_i[i]$: tatsächliche lokale Zeit in Prozess i
  - $C_i[k]$ mit $k \neq i$: Letzte Info über die lokale Zeit in P_k die P_i erfahren hat
    - Zeit des letzten Ereignisses a in P_k, das kausal vor b geschah

- **Regel 1: $C_i(b)[i] = C_i(a)[i] + d$** ($d > 0$)
  - falls $a \rightarrow b$ in P_i
- **Regel 2: $C_j(b)[k] = \max (C_j(b)[k], t_m[k])$**
  - Versenden einer Nachricht (Ereignis a) mit Zeitstempel $t_m = C_i(a)$
  - Korrektur des Zeitstempels bei Empfangen einer Nachricht (Ereignis b) mit Zeitstempel $t_m > 0$

- P_k erfährt durch Regel 2 auch über transitive Abhängigkeiten
  - Zeiten, an denen zuletzt Meldungen an den Sender geschickt wurden



Vektor-Uhren – Beispiel



Vektor-Uhren

- Es gilt

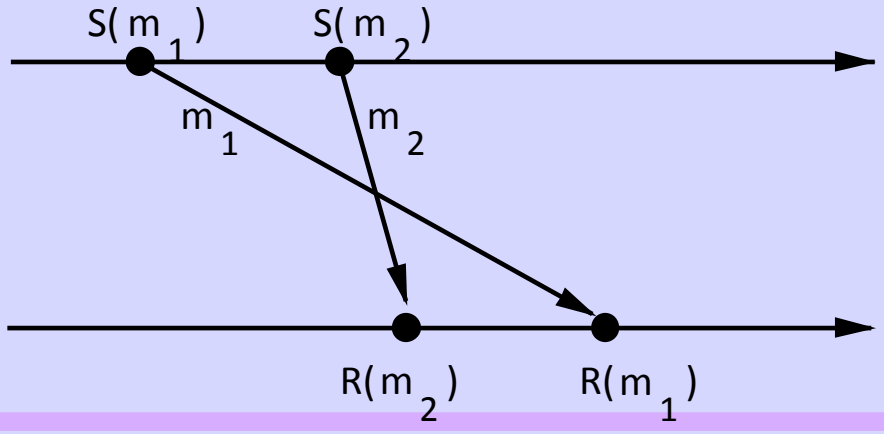
$$C(a) = C(b) \quad \text{iff} \quad \forall i \quad C(a)[i] = C(b)[i]$$
$$C(a) \leq C(b) \quad \text{iff} \quad \forall i \quad C(a)[i] \leq C(b)[i]$$
$$C(a) < C(b) \quad \text{iff} \quad C(a) \leq C(b), \neg(C(a) = C(b))$$
$$C(a) \,||\, C(b) \quad \text{iff} \quad \neg(C(a) \leq C(b)), \neg(C(a) \geq C(b))$$

- a und b sind **nebenläufig**, falls C(a)||C(b)
- a und b sind **kausal abhängig**, falls C(a)<C(b) oder C(b)<C(a)

- Es gilt: **a → b iff C(a)<C(b)**
- Über Vektor-Uhren lässt sich Ereignisabfolge genau rekonstruieren.

Kausale Nachrichtenordnung

- Kausale Nachrichtenordnung
  - Nachrichten sind kausal geordnet, falls $S(m_1) \rightarrow^* S(m_2) \Rightarrow R(m_1) \rightarrow^* R(m_2)$
  - Empfangs-Ereignisse haben dieselbe kausale Ordnung wie Sende-Ereignisse
  - Problem in Packet Switched Networks, da Nachrichten unterschiedliche Pfade nehmen können
- Raum-Zeit-Diagramm
  - Verletzung bei sich überkreuzenden Abhängigkeitspfeile *in die gleiche Richtung*



Birman-Schiper-Stephenson Protokoll

- Dieses Protokoll sorgt dafür, dass eine Meldung garantiert erst dann ausgeliefert wird, wenn die vorausgehende Meldung ausgeliefert wurde.
- Vektor-Zeitstempel werden benutzt, um Senden und Empfangen von Nachrichten zu zählen
  - Entscheidung, ob vorausgehende Meldung existiert, auf die gewartet werden muss
- Anwendungen
  - Finanztransaktionen, Börsenhandel

Birman-Schiper-Stephenson Protokoll

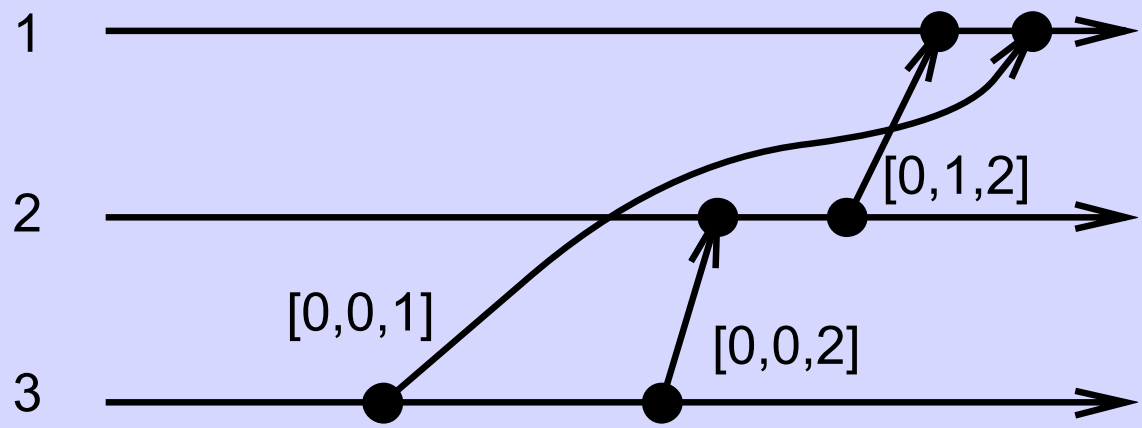
- Grundlagen
  - Broadcasts
  - $C_{Pi}[i]$ zählt Meldungen, die P_i verbreitet
  - $C_{Pi}[j]$ zählt die von P_i empfangenen Meldungen mit Absender P_j
- **Regel 1:**
  Bevor Prozess P_i die Meldung *m* verbreitet, inkrementiert er $C_{Pi}[i]$ und stempelt *m*.
- **Regel 2:**
  Erhält Prozess P_j von P_i Meldung *m* mit Zeitstempel C_m, so wird *m* erst ausgeliefert, wenn folgende Bedingungen erfüllt sind:
  - $C_{Pj}[i] = C_m[i] - 1$
  - $C_{Pj}[k] \geq C_m[k] \quad \forall k \in \{1,2,\ldots,n\} - \{i\}$ wobei n = Anzahl der Prozesse
- **Regel 3:**
  Wird Meldung an P_j ausgeliefert, so wird C_{Pj} nach Regel 2 für Vektoruhren nachgeführt.

Birman-Schiper-Stephenson Protokoll

- (2a) P_j hat alle Meldungen von P_i gesehen, die *m* vorangehen.
  - Vermeidung unmittelbarer Überkreuzungen
- (2b) P_j hat alle Meldungen gesehen, die P_i vor dem Verbreiten von *m* gesehen hat
  - Vermeidung mittelbarer Überkreuzungen
- Durch Verbreitung per broadcast kommt es zu keinen Verklemmungen

Birman-Schiper-Stephenson Protokoll

- Beispiel
  - [0,0,2] darf erst an P_2 ausgeliefert werden, nachdem auch der broadcast von [0,0,1] bei P_2 angekommen ist.
  - [0,1,2] darf erst an P_1 ausgeliefert werden, nachdem [0,0,1] bei P_1 angekommen ist.
  - P_1 muss vor [0,1,2] noch den broadcast [0,0,2] empfangen.



Wechselseitiger Ausschluss – Zentralistische Lösung

- zentraler Serverprozess vergibt Zugangsberechtigung zu kritischem Abschnitt
  - Server nimmt Anforderungen entgegen
  - Server beantwortet erste Anforderung durch OK und reiht andere in Warteschlange
  - Klienten sind dadurch blockiert
  - Tritt Klient aus dem kritischen Abschnitt wieder aus, so meldet er sich beim Server ab
  - Server sendet dem nächsten Wartenden OK
- Lösung ist einfach, hat aber Server als zentralen Knackpunkt

Vert. wechsels. Ausschluss – Ricart-Lamport-Agrawala

- Algorithmus von Ricart und Agrawala sowie Lamport
- Voraussetzungen:
  - Lamports verteilte logische Uhr mit → als Totalordnung
  - zuverlässige Kommunikation
- Ablauf:
  - Prozess P schickt Nachricht an alle beteiligten Prozesse im System mit
    - Namen des kritischen Abschnitts
    - eigenem Namen
    - Uhrzeit
  - P wartet auf das OK aller Prozesse
  - P betritt kritischen Abschnitt, sobald er alle OKs hat

Verteilter wechselseitiger Ausschluss – Algorithmus

- Empfang einer Anfrage wegen kritischem Abschnitt
  - Angleichen der logischen Uhrzeit nach Lamport
  - Außerdem:
    - nicht am kritischen Abschnitt interessiert: Sende OK
    - selbst im kritischen Abschnitt: Beantwortung zurückstellen
    - wartet selbst: Vergleiche Zeit der Nachricht mit der Zeit seiner eigenen Anforderung.
      - eigene Zeit kleiner: Gewonnen und Beantwortung zurück stellen
      - Andernfalls: (trete von eigenem Wunsch zurück und) Sende OK
- Verlassen des kritischen Abschnitts:
  - Sende OK an alle zurückgestellten Anforderungen

Verteilter wechselseitiger Ausschluss

- Kein deadlock
  - da → Totalordnung und wohl fundiert
- Keine starvation
  - da logisch synchrone Uhren
    - eine Uhr kann nicht permanent hinter allen nachgehen
    - jede request Nachricht stellt Uhr vor
- Kritik
  - 2·(n-1) Nachrichten pro Eintritt in kritischen Abschnitt
  - Jeder Prozess muss jede Anforderung genehmigen
    ⇒ n Knackpunkte statt eines zentralen Knackpunkts
  - Jeder Prozess muss gesamte Verwaltungslast tragen können
  - Variante mit Mehrheit von Genehmigungen:
    - Nicht jeder Prozess ist Knackpunkt, dafür aber komplizierterer Algorithmus
- Algorithmus lohnt sich nicht gegenüber Algorithmus mit zentraler Vergabe

Token-Ring-Algorithmus

- Statt Token immer wieder an Zentrale zurückzugeben, Token in einem Ring weiterreichen
- vorteilhaft, wenn viele in den kritischen Abschnitt wollen
- nicht vorteilhaft, wenn nur wenige interessiert sind
- unbegrenzt vielen Token Operationen möglich, ohne dass jemand am KA interessiert ist

Wechselseitiger Ausschluss – Vergleich

Algorithmus	Nachrichten pro Monitor-Durchgang	Verzögerung vor Eintritt	Probleme
Zentral	3	2	Manager Crash
Verteilt	2(n-1)	2(n-1)	Crash irgendeines Prozesses
Token Ring	unbegrenzt	max: (n-1) min: 0	Crash irgendeines Prozesses verlorenes Token

Wahlalgorithmen

- Viele verteilte Algorithmen benötigen irgendeinen speziellen Prozess als Koordinator
  - i.A. jeder Prozess kann diese Rolle übernehmen (SPMD-Paradigma)
- Annahmen:
  - Alle Prozesse durchnummeriert
  - Nummern der Prozesse sind bekannt
  - größter Prozess soll Koordinator werden
  - nicht bekannt:
    Welche Prozesse laufen zu gegebener Zeit?
    Welche Prozesse sind abgestürzt?

Rüpel-Algorithmus

- Wenn Prozess P bemerkt, dass Koordinator nicht antwortet, so initiiert P Neuwahl wie folgt
  1. P schickt eine WAHL-Nachricht an jeden Prozess mit höherer Nummer
  2. Falls niemand antwortet, gewinnt P und wird Koordinator
  3. Falls einer der Höheren antwortet, gibt P auf.
     (Der Höhere setzt Wahl fort).
- Falls P WAHL-Nachricht empfängt
  - beantwortet Nachricht mit OK und
  - hält Wahl ab → *Gibt es noch lebende größere?*
- letzter Prozess, der nicht aufgibt, wird Koordinator
  - informiert alle anderen Prozesse mit Spezialnachricht
- Nach Neustart eines abgestürzten Prozesses, sofort Wahl
  → *Bin ich der derzeit Größte?*

Ring-Algorithmus

- Idee: Lineare Ordnung des Rüpel-Algorithmus zu Ring schließen
- Prozess P zirkuliert WAHL Nachricht auf dem Ring
  - nicht ansprechbare Prozesse werden übergangen
  - lebende Prozesse tragen ihre Nummer in Nachricht ein
- Kommt Nachricht zu P zurück
  - Ermittlung des größten lebenden Prozesses
  - weiteres Rundschreiben mit Ausgang der Wahl
- Nebenbemerkung
  - P_1 und P_2 können gleichzeitig Wahl abhalten, ohne dass es Konflikte gibt.

Transaktionen

- Transaktionen sind höherer Synchronisationsmechanismus
  - Ursprung: Datenbankbereich
- Eine Transaktion bündelt eine Menge von Einzelaktionen sodass das Bündel nach außen wie eine Einzelaktion erscheint
  - Beispiel: Überweisung = {Abbuchung, Gutschrift}.
- Im Erfolgsfall
  - werden alle Einzelaktionen durchgeführt
  - stellt sich ein permanenter Effekt ein (*the transaction commits*)
- Transaktionen können scheitern (Misserfolgsfall)
  - falls eine der Einzelaktionen scheitert
  - falls eine Integritätsbedingung verletzt wird
- Im Misserfolgsfall
  - werden die bereits gelungenen Einzelaktionen zurückgenommen (*roll-back*)
  - als sei insgesamt nichts geschehen (*the transaction aborts*)

ACID Eigenschaften von Transaktionen

- Transaktionen haben die ACID Eigenschaften
  - Atomicity, Consistency, Isolation, Durability
- **Atomicity**: das Bündel wird ganz oder garnicht ausgeführt
- **Consistency**: Integritätsbedingungen bleiben erhalten. Gibt es eine Bedingung I, die vor T gilt und bewahrt werden muss, so wird T nicht ausgeführt, falls I danach verletzt wäre.
- **Isolation**: Die Einzelaktionen und Zwischenzustände sind von der Umwelt isoliert. Es ist unmöglich, von außen einen Zwischen-zustand zu beobachten. Dies bedeutet auch, dass zwei sich zeitlich überlappende Transaktionen logisch isoliert bleiben, als ob sie hintereinander ausgeführt würden (**serializability**).
- **Durability**: Im Erfolgsfall hat die Transaktion einen permanenten Effekt, der auch einen System-Crash überlebt.

Beispiel zu Transaktionen

- Buchung einer Reise (Hinflug, Hotels, Mietwagen, Rückflug)
- Atomicity: Reise nur gesamthaft buchen
  - Teile vorläufig buchen
  - Vorläufige Buchungen zurücknehmen (roll-back), wenn eine Buchung scheitert.
- Scheitern der Transaktion (**abort**)
  - z.B. keine Plätze mehr buchbar
  - saubere Rücknahme alle Teilvorgänge
  - keinerlei permanente Seiteneffekte
- Erfolgsfall (Festschreibung, commit)
  - Festschreibung der vorläufigen Buchungen, mindestens als log.
  - Ergebnisse werden permanent wirksam
    - z.B. endgültiger Grundbucheintrag
  - Typischerweise auf permanentem sicherem Speicher
    - redundante Magnetplatten, Bändern, WORM optische Platten etc.

Transaktions-Befehle

- BEGIN_TRANS
  - folgende Befehle bilden zusammen Transaktion
- END_TRANS
  - Beende Transaktion und versuche Festschreibung
- ABORT_TRANS
  - Abbruch Transaktion; Rückkehr zum status quo ante
- READ
  - Lese von Datei
- WRITE
  - Schreibe auf Datei
- Beispiel

```
BEGIN_TRANS
        if (reserve(ZRH,JFK) < 0) ABORT_TRANS;
        if (reserve(JFK,CMH) < 0) ABORT_TRANS;
END_TRANS
```

Transaktions-Eigenschaften

- **Serialisierbarkeit**
  - Nebenläufige Transaktionen beeinflussen sich höchstens so, wie bei (irgend-)einer Nacheinanderausführung
  - Sind n nebenläufige Transaktionen gegeben, darf das Transaktions-System (Transaktions-Monitor, z.B. CICS) ihre Einzelaktionen nach einem **Ablaufplan** (schedule) verschränkt ausführen
  - Bedingung: Effekt der Transaktionen insgesamt äquivalent zu seriellem Plan mit nacheinander ausgeführten Transaktions-Bündeln
  - Ablaufpläne sind aus Effizienzgründen nötig
    - Alle Buchungen im selben Hotel aus mehreren Transaktionen nacheinander ausführen, dann alle Mietwagen-Buchungen etc.

Implementierung – Privater Arbeitsraum

- einfachste Lösung:
  - Ausführung der Transaktion zunächst auf privatem Arbeitsraum
    - ähnlich Schmierzettel, Konzeptpapier
  - commit-Phase: Endergebnisse auf permanentes Medium schreiben
- nicht den ganzen Systemzustand kopieren
  - kopieren der zu beschreibenden Teile
- Beispiel:
  - Aufbau von *Schattenblöcken* (die zu beschreibenden Blöcke einer Datei)
  - Kopieren des Index (I-Nodes) und Schreiben der neue Blöcke auf Platte
  - `commit`: Neuen Index zurück schreiben
  - `fail`: Originaldatei bleibt unbeschadet

  → größere Effizienz

  → exklusiver Zugriff auf Index kürzer
  → Minimierung der Zeit, innerhalb derer Crash fatal wäre, auf überbrückbares Intervall

Implementierung – Vorausgeführtes Logbuch

- Logbuch auf sicherem Speicher
  - Notieren der Auswirkungen des Updates bei nachfolgendem Commit
    - z.B. Ausgabe von UNIX `diff alt neu`
- Logbuch korrekt geschrieben
  - Durchführen des eigentlichen File-Updates und Commit im Log notieren
- Crash beim Update
  - Restauration aller Updates seit Generalsicherung aus Log-Einträgen
  - Bei geeigneten Log-Einträgen auch Rücknahme der Änderungen möglich (*rollback*)
- Konzeptuell:
  - Logbuch als Permanentspeicher
  - Datenbank und gesamter normaler Hauptspeicher als Arbeitsraum

Implementierung – Zwei-Phasen Festschreibung

- Problem: Atomare Festschreibung
- Prozess, der Transaktion durchführt, als Koordinator
- Phase 1:
  - Koordinator notiert `PREPARE COMMIT` ins Logbuch und benachrichtigt alle beteiligten Prozesse
  - Prozesse entscheiden, ob zum Commit bereit oder nicht
  - Prozesse notieren Entscheidung ins Logbuch und senden Entscheidung an Koordinator
- Phase 2:
  - nicht alle bereit: Koordinator notiert `FAILURE` ins Logbuch
  - alle bereit: Koordinator notiert `COMMIT` ins Logbuch und informiert Prozesse
    - Prozesse notieren `COMMIT` ins Logbuch und führen Festschreibung durch
    - Prozesse informieren Koordinator über Ende der Transaktion
- 1. Phase → keine unkoordinierten Commits einzelner Prozesse
- Verwendung des Logfiles → Schutz gegen Abstürze

Verteilte Verklemmungen

- Problem prinzipiell gleich wie in Einzelprozessor-Systemen
- schwerer zu lösen, da relevante Information über Prozessabhängigkeiten verteilt ist
- vier Lösungsansätze
  1. Problem ignorieren
     (im Zweifel Maschine neu booten)
  2. Deadlocks erkennen und dann (ohne reboot) beseitigen
  3. Deadlocks statisch vermeiden
     (durch geschickte statische Allokationsstrategien)
  4. Deadlocks dynamisch vermeiden
     (durch geschickte dynamische Allokation).
- Wir diskutieren 2 und 3.

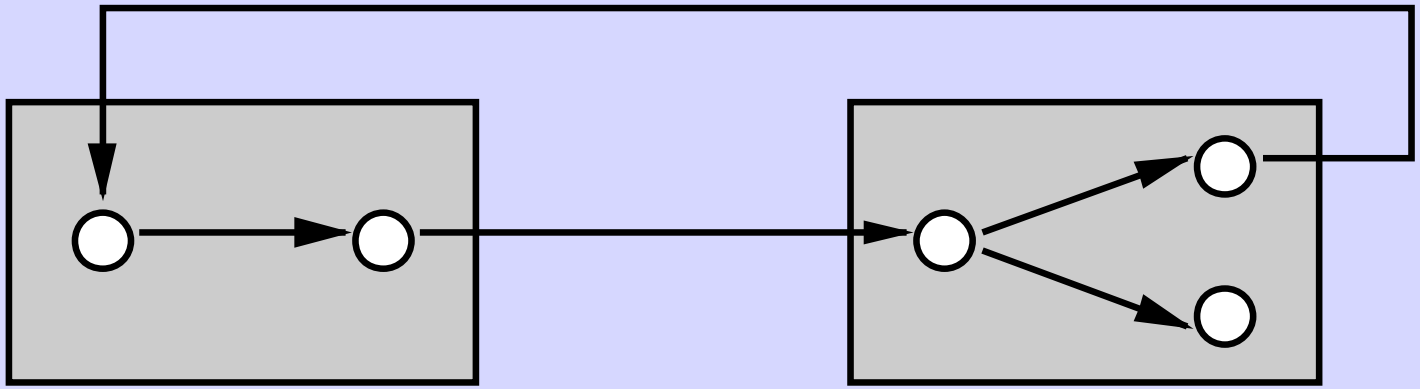
Zentrale Erkennung verteilter Verklemmungen

- Koordinator hält zentralen Abhängigkeitsgraphen
  - Systemteilnehmer informieren ihn durch Nachrichten über Anforderung und Freigabe von Ressourcen
- Problem: Pseudo-Verklemmungen
  - Auslieferung der Nachrichtensequenz (Freigabe, Anforderung) in falscher Reihenfolge
  - Lösung: Synchronisiertes Ausputzen (*flush*) der Nachrichtenpuffer
  - Alternativ: Kausale Nachrichtenordnung
  - aber: Teuer!

Verteilte Erkennung verteilter Verklemmungen

- Problem: Abhängigkeiten über Maschinengrenzen hinweg möglich
- Chandy-Misra-Haas-Algorithmus
  - verschickt Nachricht, wenn Prozess wegen eines anderen blockiert
  - Nachricht besteht aus Tripel (Originator, Sender, Empfänger)
- erste Nachricht: (Originator, Originator, Empfänger)
- Empfänger blockiert: Nachricht wird weitergereicht
  - Verschicken an den oder die Prozesse, auf die er wartet
  - (Originator, Empfänger_n, Empfänger_n+1)
- Kommt Nachricht zum Originator zurück → Verklemmung gefunden
  - (Originator, Empfänger n, Originator)

Verteilte Erkennung verteilter Verklemmungen



- Beseitigung der Verklemmungen
  - einer der Prozesse nimmt Anforderung zurück
    - Prozess terminieren
    - Transaktion terminieren, die Anforderung auslöst
  - Alle am Zyklus beteiligten Prozesse bekannt
    → Bestimmung des zurücktretenden Prozesses durch Wahl
- Problem: Blockierte Prozesse müssen Interrupt (Ankunft einer Nachricht) behandeln
  - `upcall:` Betriebssystem ruft bestimmten Handler-thread auf
    → nicht-trivialer Mechanismus

statische Verklemmungsvermeidung

- alle Ressourcen linear aufsteigend ordnen
- Akquisition nur in aufsteigender Ordnung möglich
  → Zyklen unmöglich
- Problem
  - Prozess muss wissen, welche Ressourcen er insgesamt braucht

Verklemmungsvermeidung mit Lamports verteilter Uhr

- Jeder Prozess bekommt eindeutigen Zeitstempel
- Vereinbarung: Nur ältere dürfen auf jüngere Prozesse warten oder umgekehrt
- *wait-die* Variation
  - alter Prozess wartet auf Betriebsmittel des jüngeren Prozesses
  - jüngerer Prozess wird terminiert, wenn älterer Prozess ein gewünschtes Betriebsmittel hält

  → *restart-kill* Sequenzen möglich, wenn der jüngere sofort wieder startet
- *wound-wait* Variation
  - älterer Prozess entzieht jüngerem Prozess das Betriebsmittel
  - jüngerer Prozess wartet auf das Betriebsmittel des älteren Prozesses

  → keine *restart-kill* Sequenzen

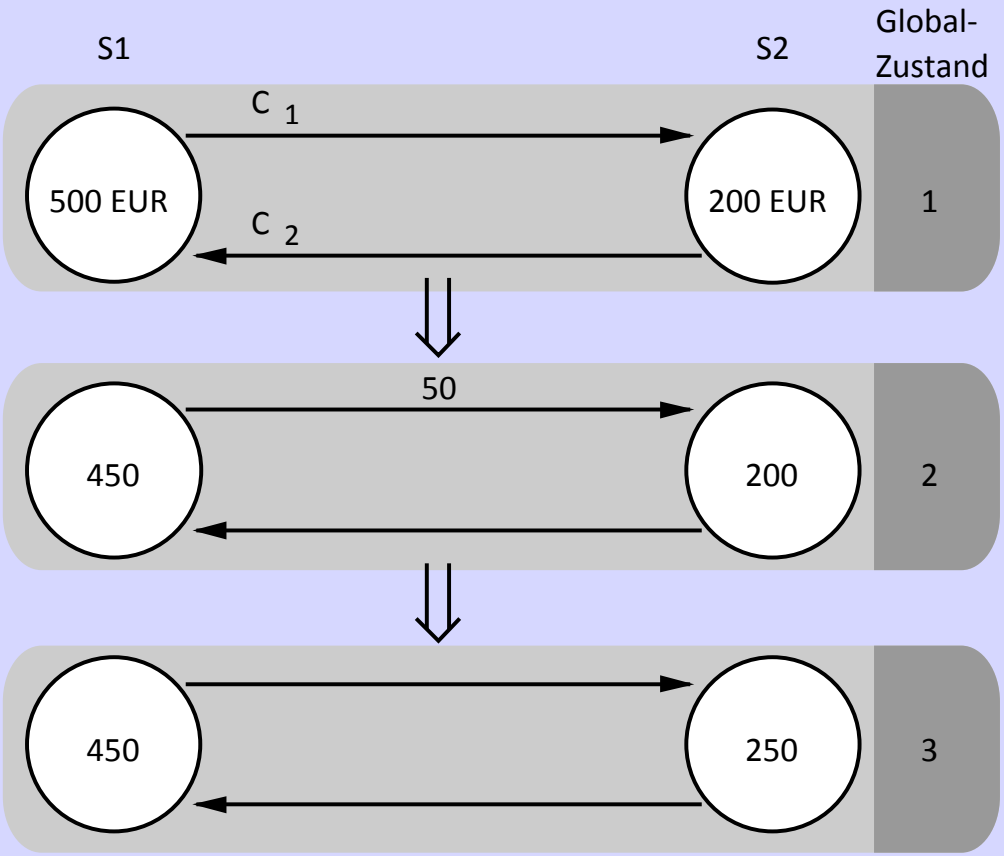
Globaler Systemzustand (*Global State*)

- Problem: Garantiert gleichzeitiges Aufnehmen des Zustands
  - da keine globale Zeit

- Bemerkung
  - im verteilten System gehört Zustand der Kommunikationskanäle zum Systemzustand

Globaler Systemzustand – Beispiel: Bank-Transfers

- Fixierung des Gesamtzustands:
  - Inventur bei S1; S2; C1; C2
- möglicher Fehler:
  - Inventur bei S1 in Global-Zustand 1
  - Inventur bei C1; C2 und S2 in Globalzustand 2
  - Gesamtkontostand stimmt nicht



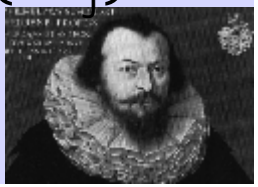
Globaler Systemzustand – formale Definitionen

- Ziel: Aufnehmen eines konsistenten Globalzustands aus lokalen Zuständen einzelner Rechner und Kanäle

- Definition: **Lokalzustand**
  - Der Lokalzustand LZ_i jedes Rechenorts S_i (*site; computer; process*) ist der lokale Kontext der verteilten Anwendung.

- `send(`m_{ij}`)` / `rec(`m_{ij}`)` : Ereignis des Sendens / Empfangens einer Meldung m_{ij} von S_i nach S_j

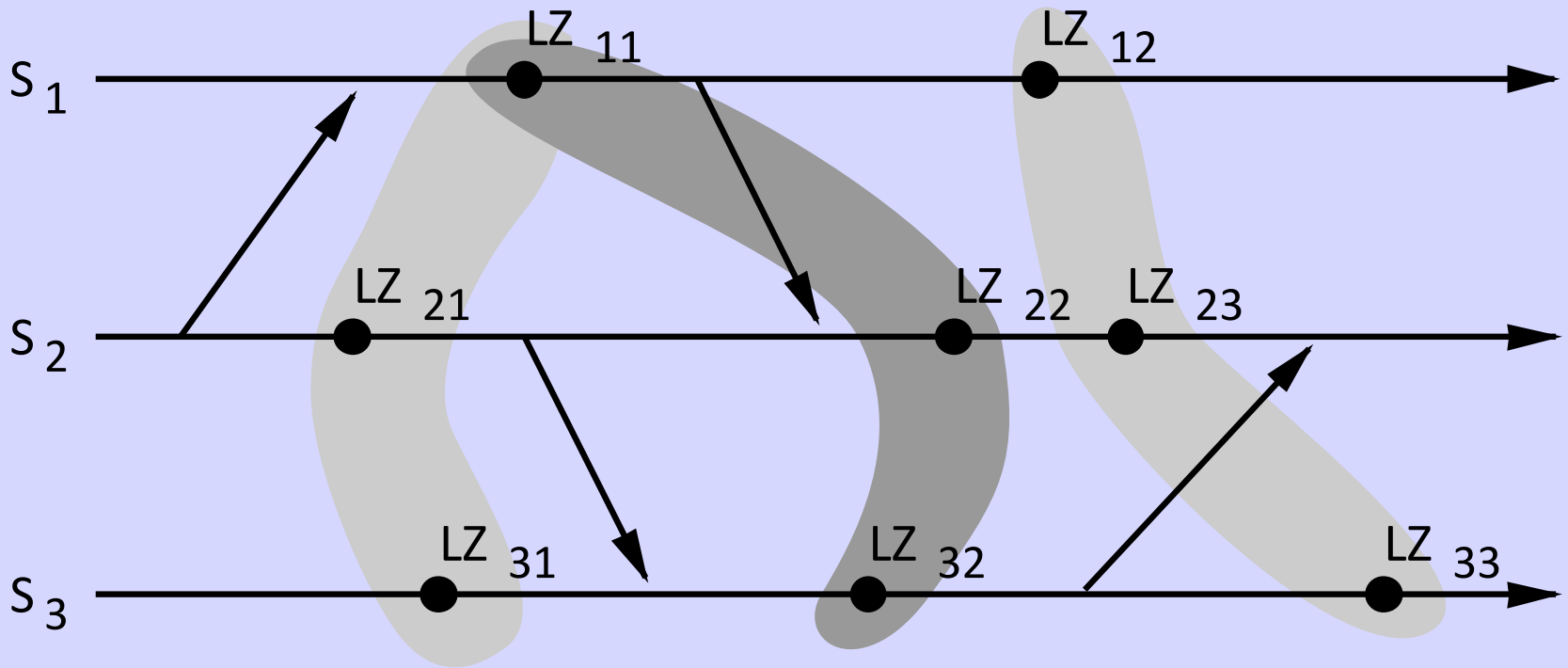
- Es gilt:
  - `send(`m_{ij}`)` ∝ LZ_i iff time(send(m_{ij})) < time(LZ_i)
  - `rec(`m_{ij}`)` ∝ LZ_j iff time(rec(m_{ij})) < time(LZ_j)



Globaler Systemzustand – formale Definitionen

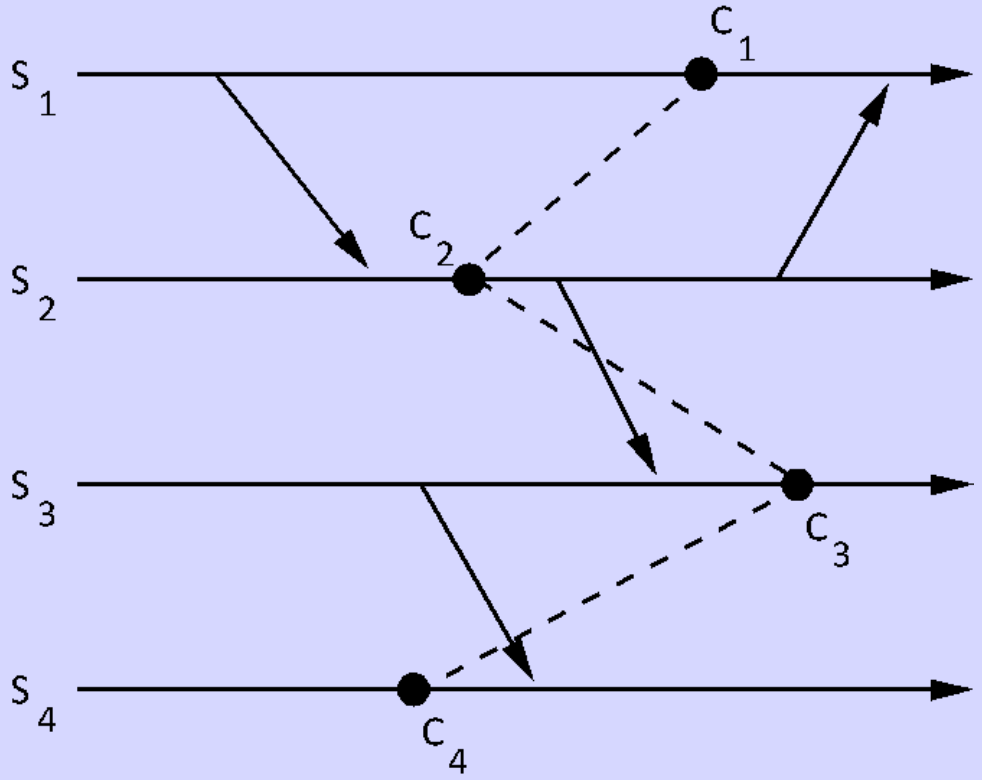
- Def.: Nachrichtenmenge **In-Transit**:
  - $transit(LZ_i;LZ_j) = \{m_{ij} | send(m_{ij}) \in LZ_i \wedge rec(m_{ij}) \notin LZ_j\}$
- Def.: Nachrichtenmenge **Inkonsistent**:
  - $inconsistent(LZ_i;LZ_j) = \{m_{ij} | send(m_{ij}) \notin LZ_i \wedge rec(m_{ij}) \in LZ_j\}$
  - Sammlung von Lokalzuständen ist inkonsistent, wenn sie inkonsistente Nachrichtenmengen enthält
- Globalzustand ist **transitfrei** (*transitless*) iff
  $transit(LZ_i;LZ_j) = \emptyset \ \forall i,j$
- Globalzustand ist ***stark konsistent*** (*strongly consistent*) gdw. er transitfrei und konsistent ist.

Globaler Systemzustand – Beispiel



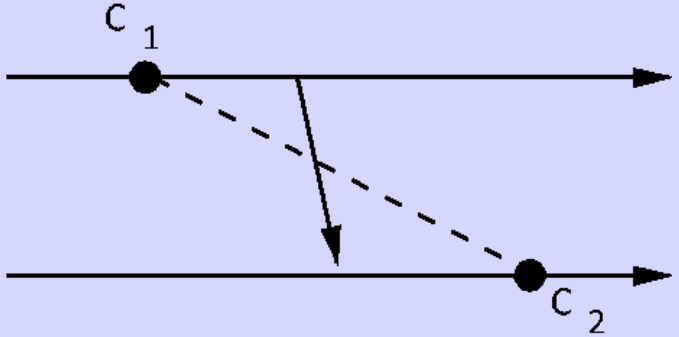
Schnitte (Cuts) einer verteilten Rechnung

- **Schnitt**: Graphische Repräsentation eines Globalzustands im Raum-Zeit-Diagramm
  - besteht aus Linie, die Schnitt-Ereignisse $C = \{c_1, \ldots, c_n\}$ verbindet



Schnitte (Cuts) einer verteilten Rechnung

- Schnitt C = {c_1; c_2; … ; c_n} ist ***konsistent***, gdw es zwischen je zwei Schnitt-Ereignissen keine inkonsistenten Nachrichten gibt.
  - Empfang von Nachricht m_{ij} vor Ereignis c_j auf S_j
    ⇒ Senden von Nachricht m_{ij} vor Ereignis c_i auf S_i
- Theorem:
  Schnitt C = {c_1; c_2; … ; c_n} ist konsistent gdw es gibt in C keine kausal abhängigen Ereignisse (keine Abhängigkeiten $c_i \rightarrow^+ c_j$)
- Beweis (Skizze):
  - Beispiel abhängiger Ereignisse



  - Ereignisse abhängig ⇒ es gibt inkonsistente Nachricht
  - keine inkonsistente Nachricht ⇒ Ereignisse können nicht kausal abhängig sein.

Globaler Systemzustand – Chandy-Lamport Algorithmus

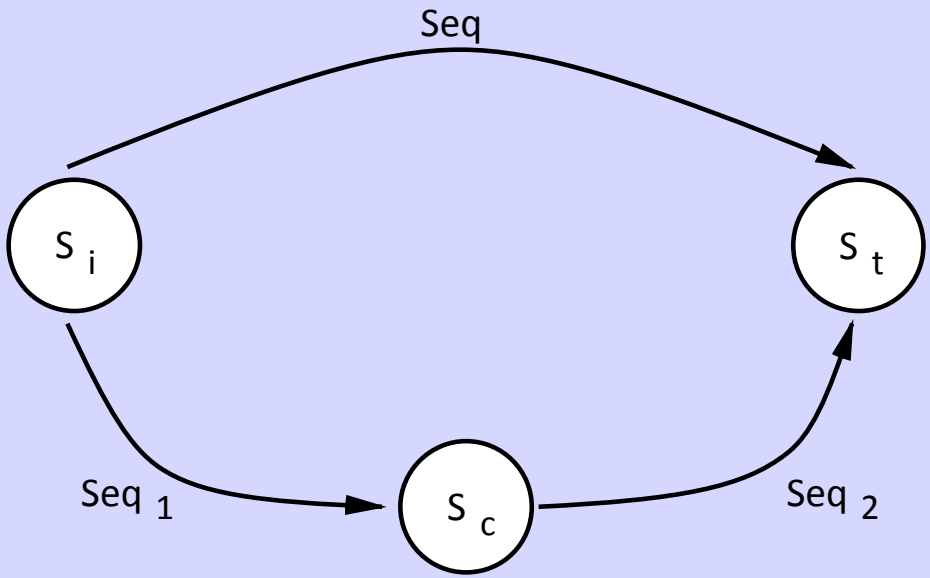
- Algorithmus nimmt konsistenten Globalzustand (auch *snapshot*) auf
- Nutzung von Marken
  - Marken haben keinen Einfluß auf die zugrundeliegenden Rechnungen
- Kommunikationskanäle sind FIFO
- **Senderegel für Marken** (für Prozess P)
  1. P nimmt seinen Zustand auf
  2. P schickt eine Marke auf jeden Kanal, auf den er noch keine geschickt hat
- **Empfangsregel für Marken** (für Prozess Q)
  - Wenn Marke in Kanal C empfangen wurde
    1. Falls Q seinen Zustand noch nicht aufgenommen hatte
       (a) (reset C) Nimm Zustand von C als leer auf (leere Sequenz von Nachrichten)
       (b) Wende Senderegel von Marken an
  - else
    2. Falls Q seinen Zustand schon aufgenommen hatte
       (a) Nimm Zustand von C auf als Sequenz der Nachrichten, die seither eingegangen sind.

Globaler Systemzustand – Chandy-Lamport Algorithmus

- Algorithmus terminiert, sobald über jeden Kanal eine Marke geflossen ist
- Sammlung des Globalzustands
  - jeder Teilzustand wird an Urheber des Schnappschusses geschickt
- Sinn der Marken:
  - Trennung der bereits aufgenommene Meldungen von den noch aufzunehmenden
- Mehrere Prozesse können gleichzeitig globalen Schnappschuss initiieren
  - Verwendung verschiedener Marken

Globaler Systemzustand – Chandy-Lamport Algorithmus

- aufgenommener Globalzustand muss nicht realem Systemzustand zu bestimmter Zeit entsprechen
- Es gilt lediglich:
  - S_c: aufgenommener Globalzustand zwischen Zuständen S_i und S_t,
  - Seq: Sequenz der Aktionen, die S_i in S_t überführt
  - Es ex. Permutation Seq ' = Seq '$_2$ ⊗ Seq '$_1$ von Seq so dass Seq '$_1$ S_i nach S_c überführt



Globaler Systemzustand – Chandy-Lamport Algorithmus

- Nützlichkeit des Globalzustands
  - Erkennen *stabiler* Systemeigenschaften
    - stabil: Eigenschaft gilt unter jeder Permutation von Aktionen
  - Beispiel:
    - Terminierung und Deadlock.
    - Gesamte Geldmenge in geschlossenem System
  - Gilt Eigenschaft in S_i, so gilt sie auch in S_c und wird bei der Aufnahme des Globalzustands erfasst

Verteilte Terminierung

- System-Modell
  - Prozesse aktiv oder inaktiv (*idle*)
  - Berechnungsnachrichten und Kontrollnachrichten
- Terminierung des verteilten Algorithmus
  - falls alle Prozesse inaktiv sind und
  - falls keine Berechnungsnachrichten in Transit
- Algorithmus mit Chandy-Lamport Algorithmus
  - Prozess inaktiv → Startet Aufnahme des Globalzustands
  - Prozess hat über jeden Kommunikationskanal Marke empfangen
    - Mitteilung an Originator, ob Prozess selbst inaktiv
  - Originator stellt Terminierung fest, falls alle Prozesse sich inaktiv melden
  - Algorithmus ist aufwändig, da i.a. mehrmals Aufnahme des Globalzustands

Verteilte Terminierung – Huangs Algorithmus

- Idee
  - Gewicht der noch zu erledigenden Arbeit protokollieren
  - Versenden einer Nachricht → Teil des Arbeitsgewichts geht auf Empfänger über
  - Arbeit erledigt → Rückgabe des Gewichtsanteils
  - Terminierung wenn alle Gewichtsteile zurückgegeben
- Algorithmus
  - Koordinator mit Anfangsgewicht 1
  - andere Prozesse mit Anfangsgewicht 0.
  - **B(DW):** Berechnungsnachricht mit Gewicht DW (etwa RPC).
  - **C(DW):** Kontrollnachricht mit Gewicht DW.

Verteilte Terminierung – Huangs Algorithmus

- Regel 1: Aktiver Prozess mit Gewicht W schickt Berechnungsnachricht an P
  - Finde $W_1 > 0$; $W_2 > 0$, so dass $W_1 + W_2 = W$
  - $W := W_1$
  - Schicke $B(W_2)$ an P
- Regel 2: Empfang von B(DW) durch P mit W
  - $W := W + DW$
  - Ist P inaktiv, so wird er aktiv
- Regel 3: Aktiver Prozess P mit W wird inaktiv
  - Schicke C(W) an Koordinator
  - P wird inaktiv
- Regel 4: Koordinator empfängt C(DW)
  - $W := W + DW$
  - Ist $W = 1 \rightarrow$ Terminierung
- Bemerkung:
  - Regel für Finden von $W_1 > 0$; $W_2 > 0$: $W_1 = W_2 = W/2$
  - Fork kann durch Aufteilung der Gewichte realisiert werden