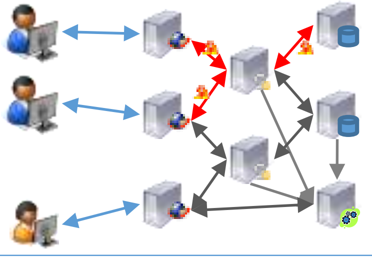


Verteilte Systeme/ Distributed Systems

Artur Andrzejak

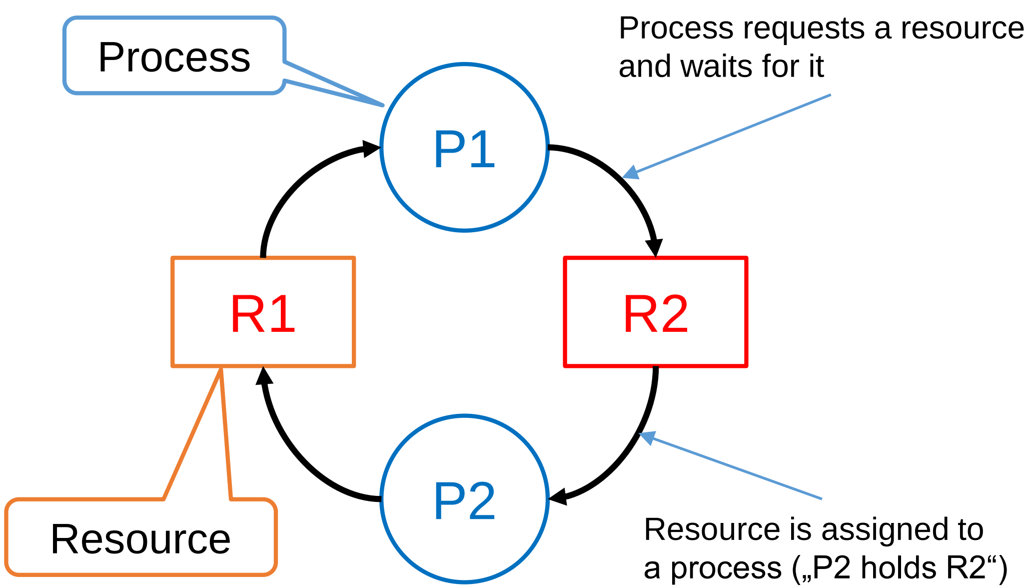
9

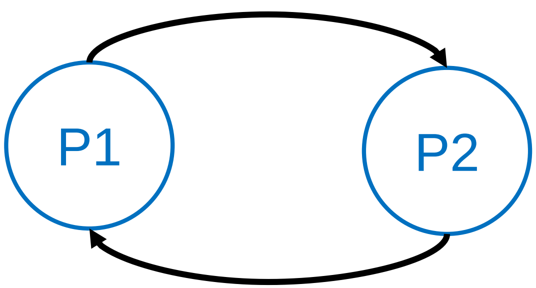
Consistent Global States

Some slides are based on the part 2 of the course:
Distributed Software Systems - Winter 2004/2005
Stefan Leue, Uni Münster

Recording Global States

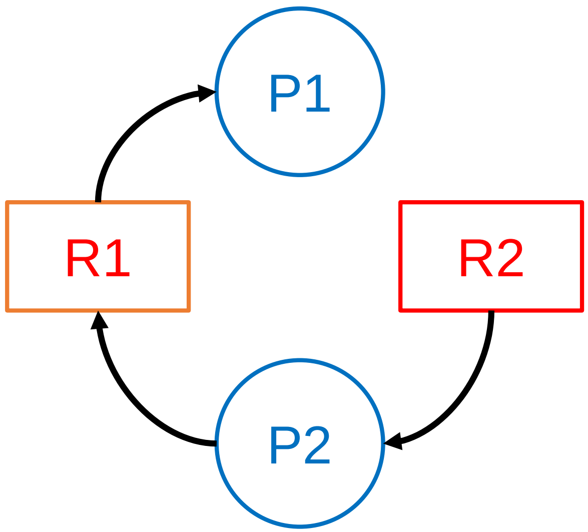
- Knowledge of a global state is useful for:
  - Census (Volkszählung)
  - Inventory in a banking system - how much money is there?
- Technically: detection of a distributed deadlock
  - Is there a cyclic *resource-allocation-graph* between processes and resources in the system?

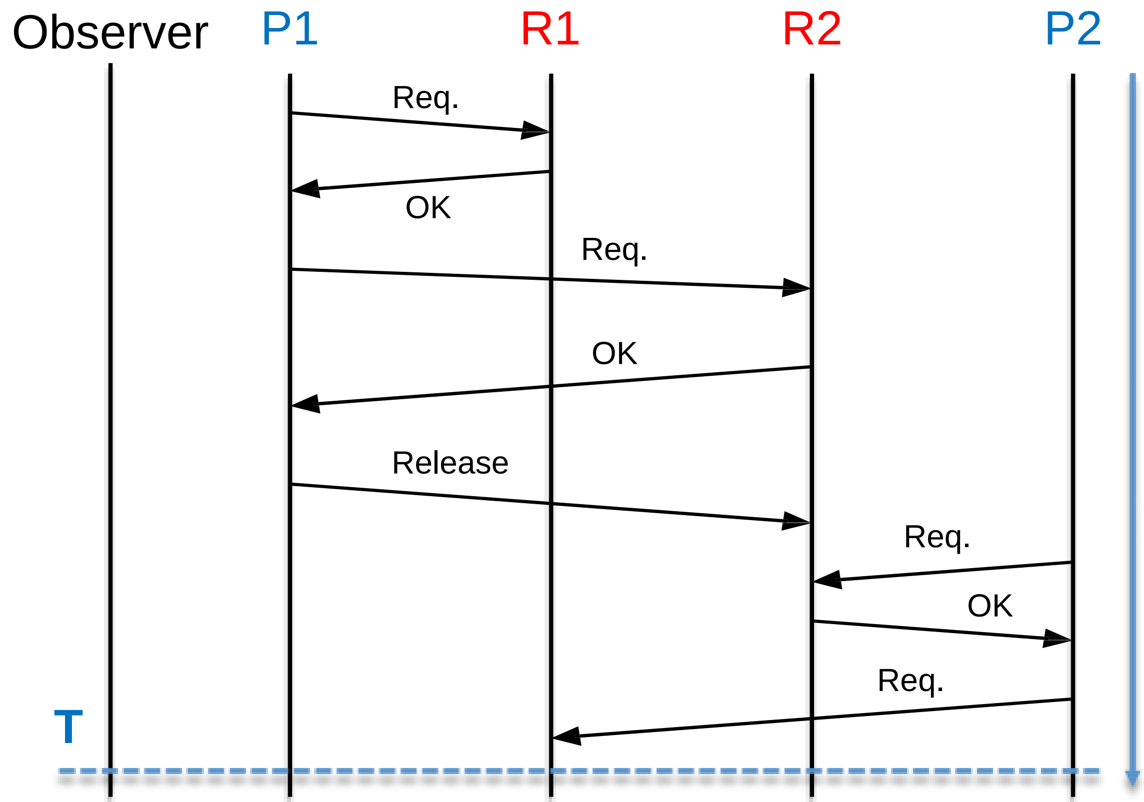




A simplified view

Is there a Deadlock?





Status at **T**:

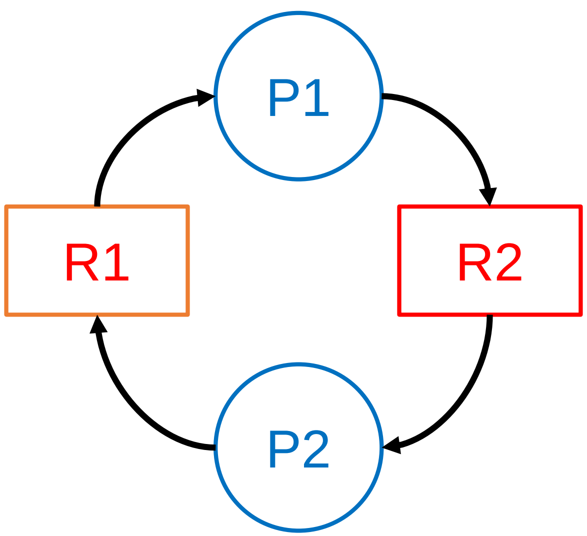
1. P1 holds R1
2. (P1 does <u>not</u> wait for anything)
3. P2 holds R2
4. P2 waits for R1

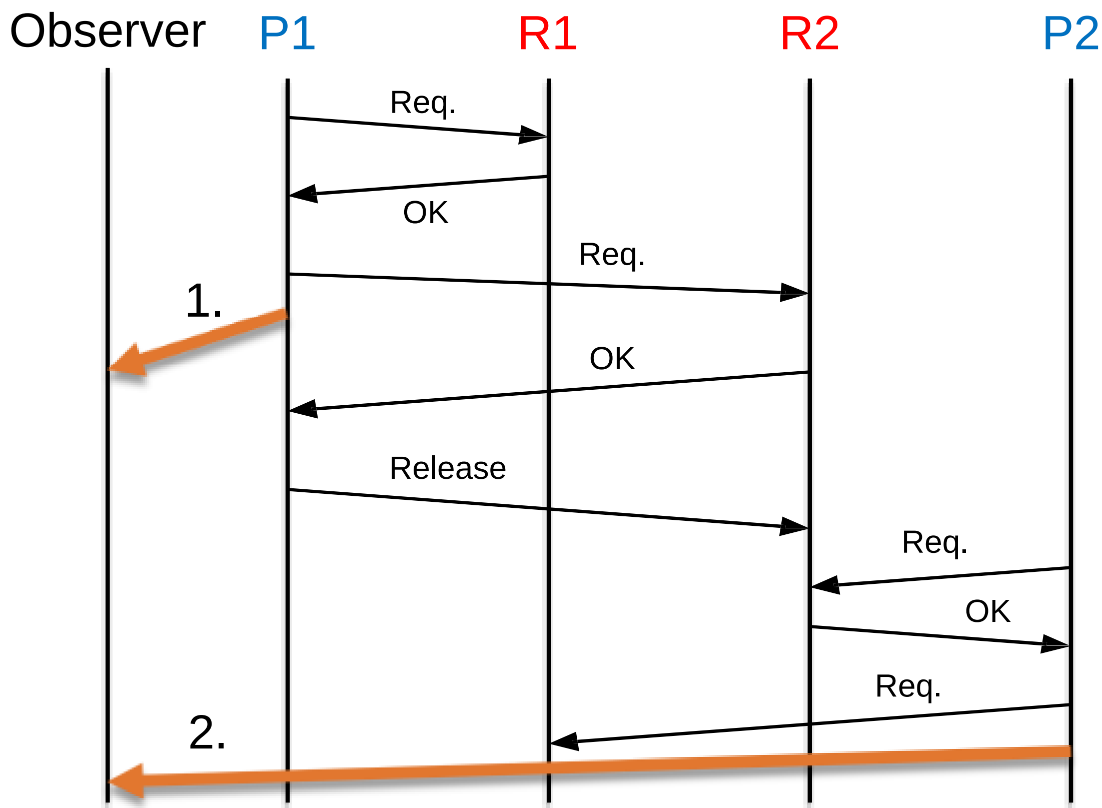
No deadlock!

A Distributed Observation

What does the observer believes?

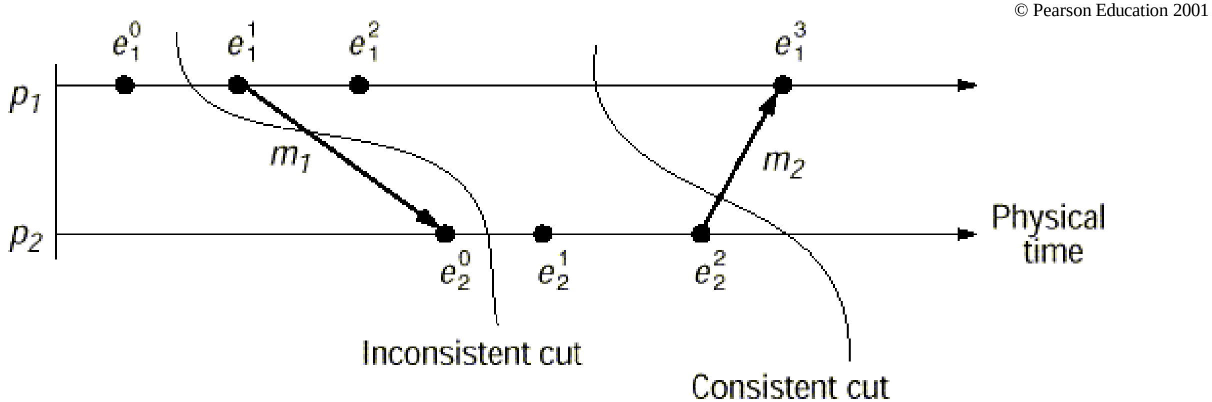
1. P1 holds R1 and *waits for R2*
2. P2 holds R2 and waits for R1





=> Deadlock!

Recall: Consistent Cut and C. Global State



- A a cut C is **consistent** if for all events e and e' holds:

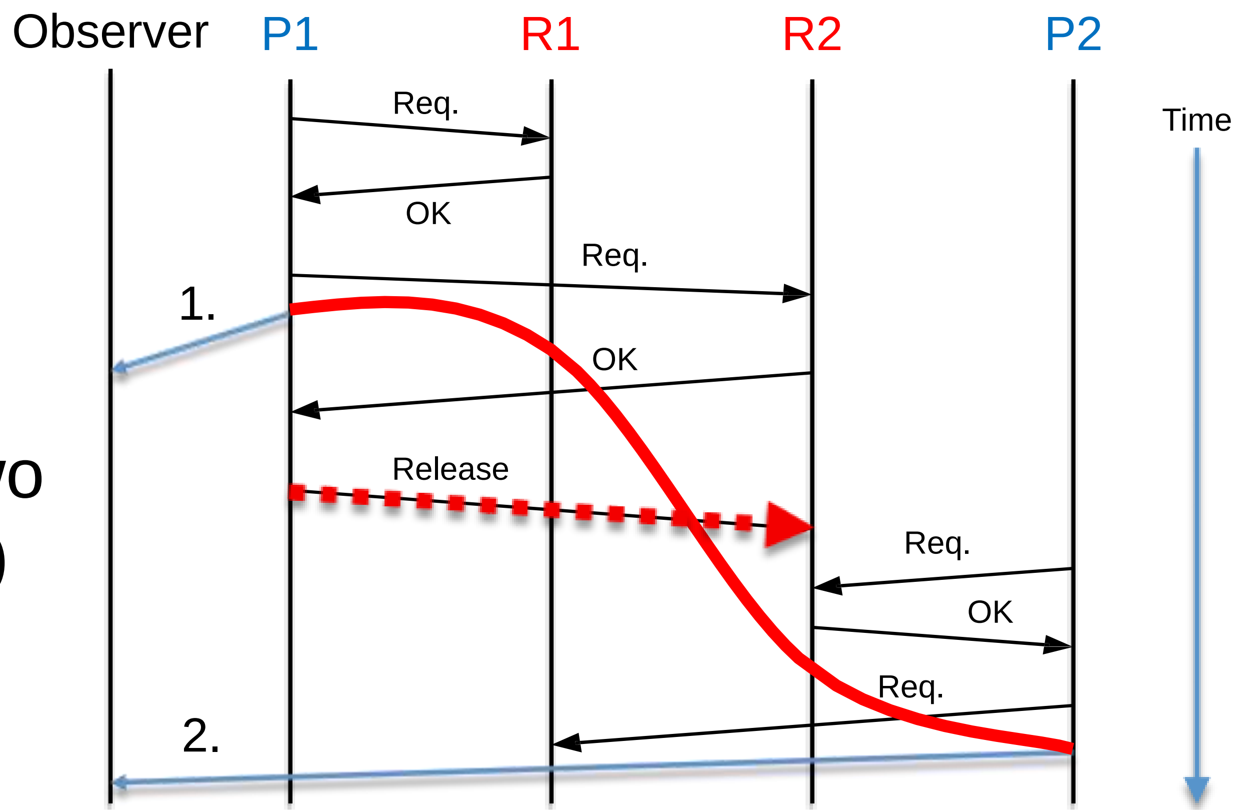
$$(e \in C) \text{ and } (e' \rightarrow e) \Rightarrow e' \in C$$

- Graphically:
  - If all arrows that intersect the cut have their bases to the left and heads to the right of it, then the cut is consistent
- Intuitively: no receiving of messages "from the future"
- A **consistent** global state is one that corresponds to a consistent cut

Deadlock Example: What Went Wrong?

Resolve of the paradox:

The cut (and so the global state) induced by the two messages (1 & 2) is <u>not</u> consistent



- Graphically:
  - If <u>all</u> arrows that intersect the cut have their bases "in the past" and heads "in the future", then the cut is consistent
- Intuitively: no receiving of messages "from the future"

Chandy-Lamport Algorithm

Some slides are based on the part 2 of the course:
Distributed Software Systems - Winter 2004/2005
Stefan Leue, Uni Münster

Chandy-Lamport Algorithmus

- We want to record a global state of an active distributed system
  - Problem: system state changes during the observation
- Snapshot (Schnappschuss):
  - *Dt. „Verteilte Momentaufnahme*, zeichnet den konsistenten Zustand eines verteilten Systems auf“
- One of the first methods proposed in 1985 by K. Mani Chandy and Leslie Lamport

Distributed Snapshots: Determining Global States of Distributed Systems

K. MANI CHANDY
University of Texas at Austin
and
LESLIE LAMPORT
Stanford Research Institute

Chandy-Lamport Algorithm

- For the determination of consistent global states
- Perfect communication: no loss, corruption, reordering or duplication of messages occurs, and messages sent will eventually be delivered
- Unidirectional FIFO channels
- The communication graph consisting of nodes corresponding to processes and directed edges corresponding to the channels is strongly connected
  - i.e. there's a path from every process to every other process
- Any process may initiate a snapshot-taking at any time
- Normal system execution continues during snapshot-taking

Principle of Operation

- We have special messages: **markers**
- There are two rules:
  - A marker sending rule and a marker receiving rule
- An initiator starts with the marker sending rule

Marker sending rule for process P:

- 1. Process P records its state.
- 2. For each <u>outgoing</u> channel C on which a marker has not been sent:
  - P sends a marker along C before it sends any further messages along C

Principle of Operation

Marker Receiving Rule for process P

- On receiving a marker along channel C:
  - **IF** P has not yet recorded its state THEN

    Record the state of C as the empty set

    Follow the “Marker Sending Rule“
  - ELSE

    Record the state of C as:

    set of messages received along C after own state (i.e. of P) was recorded and before P received this marker via C

Principle of operation: Example

Marker Receiving Rule for process P

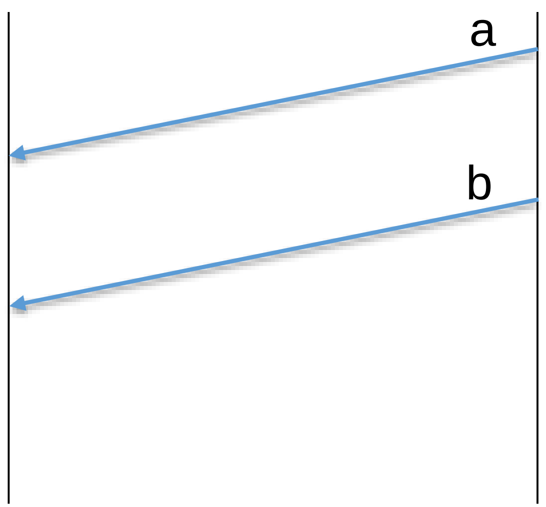
- On receiving a marker along channel C:
  - **IF** P has not yet recorded its state THEN

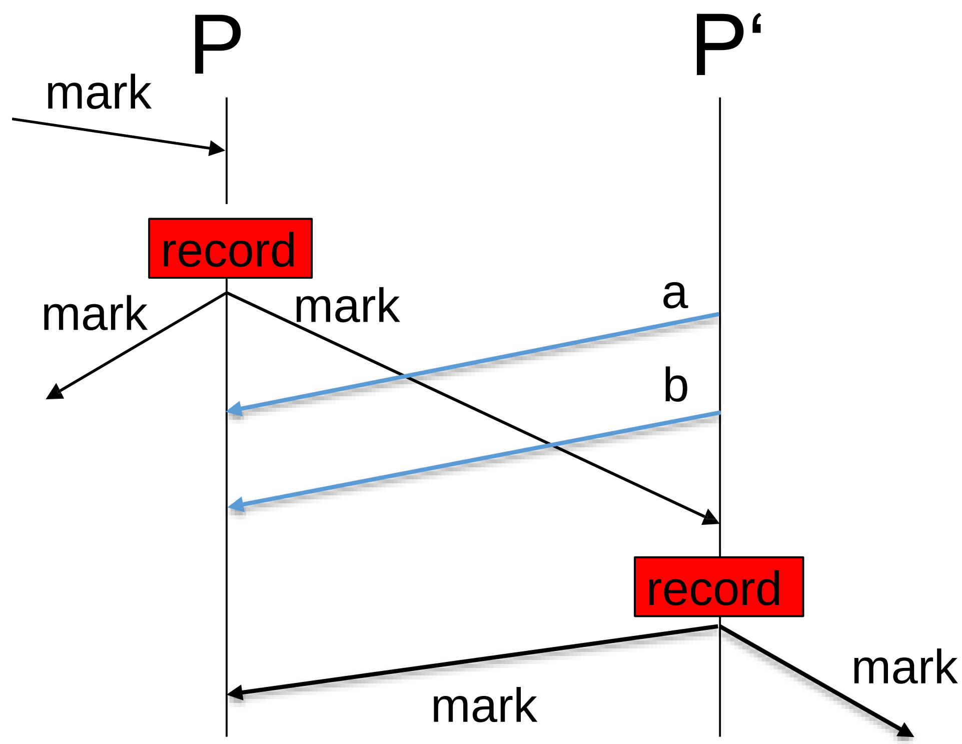
    Record the state of C as the empty set

    Follow the "Marker Sending Rule"
  - ELSE

    Record the state of C as:

    set of messages received along C after own state (i.e. of P) was recorded and before P received this marker via C



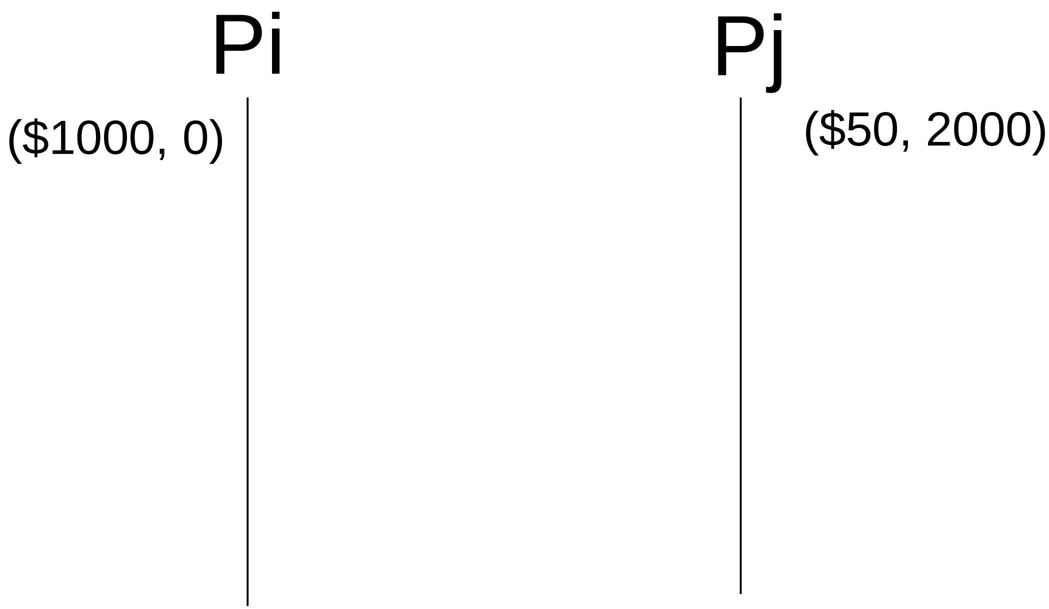


What have P, P' recorded?

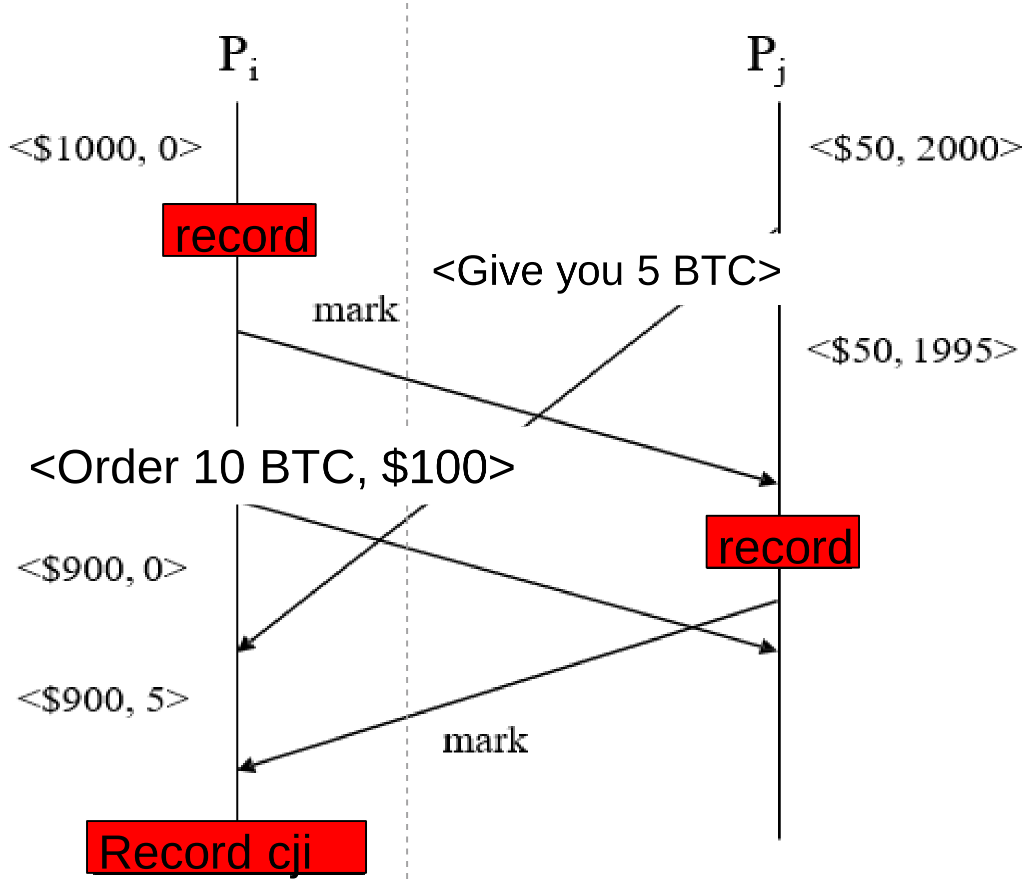
- P : state and messages a, b (in $C_{P',P}$)
- P': only its state and $C_{P',P}=\varnothing$

Chandy-Lamport Algorithm: Example 2

- Two processes exchange bitcoins, $10 per 1 BTC
- Initially:



Chandy-Lamport Algorithm: Example



What have Pi, Pj recorded?

- Pi : state ($1000, 0) and message <Give you 5> (in C_{ji})
- Pj: state ($50, 1995), $C_{ij} = \varnothing$

Chandy-Lamport Algorithm: Termination

- Theorem: The Chandy-Lamport Algorithm terminates

  Proof sketch:
  - Assumption: a process receiving a marker message will record its state and send marker messages via each outgoing channel in finite period of time
  - If there is a communication path from p_i to p_k, then p_k will record its state a finite period of time after p_i
  - Since the communication graph is strongly connected, all process in the graph will have terminated recording their state and the state of incoming channels a finite time after some process initiated snapshot taking

Improvements

Algorithms	Features
Chandy-Lamport [6]	Baseline algorithm. Requires FIFO channels. $O(e)$ messages to record snapshot and $O(d)$ time.
Spezialetti-Kearns [29]	Improvements over [6]: supports concurrent initiators, efficient assembly and distribution of a snapshot. Assumes bidirectional channels. $O(e)$ messages to record, $O(rn^2)$ messages to assemble and distribute snapshot.
Venkatesan [32]	Based on [6]. Selective sending of markers. Provides message-optimal incremental snapshots. $\Omega(n+u)$ messages to record snapshot.
Helary [12]	Based on [6]. Uses wave synchronization. Evaluates function over recorded global state. Adaptable to non-FIFO systems but requires inhibition.
Lai-Yang [18]	Works for non-FIFO channels. Markers piggybacked on computation messages. Message history required to compute channel states.
Li et al. [20]	Similar to [18]. Small message history needed as channel states are computed incrementally.
Mattern [23]	Similar to [18]. No message history required. Termination detection (e.g., a message counter per channel) required to compute channel states.
Acharya-Badrinath [1]	Requires causal delivery support, Centralized computation of channel states, Channel message contents need not be known. Requires 2n messages, 2 time units.
Alagar-Venkatesan [2]	Requires causal delivery support, Distributed computation of channel states. Requires $3n$ messages, 3 time units, small messages.

From: *Distributed Computing, Principles, Algorithms, and Systems* by Ajay D. Kshemkalyani and Mukesh Singhal, 2011, Cabridge Univ Press, https://goo.gl/HwxdZ9

Mututal Exclusion / Wechselseitiger Ausschluss

Wechselseitiger Ausschluss

- Verteilte Prozesse müssen oft ihre Aktivitäten koordinieren
- Ein fundamentaler Fall ist **Verteilter Wechselseitiger Ausschluss** (distributed mutual exclusion) – auch „gegenseitiger" Ausschluss
  - Nur einer der Prozesse darf zugleich auf eine Ressource zugreifen können
  - Z.B. eine Textdatei in NFS
- In OS's bzw. Shared Memory Systemen entspricht das dem Problem der **critical section**
  - Jedoch hier müssen wir das alleine mit Nachrichten lösen, ohne „shared variables"

Wechselseitiger Ausschluss - Anforderungen

- Jeder korrekte Algorithmus für den Wechselseitigen Ausschluss muss drei Bedingungen erfüllen:
- **ME1** (safety): Höchstens ein Prozess ist zugleich in der kritischen Sektion (KS) (hat Privileg)
- **ME2** (liveness): Anfragen, die KS zu betreten und auch zu verlassen sind letztendlich (irgendwann) erfolgreich
- **ME3** ($\rightarrow$ ordering): Falls eine Anfrage R zum Betreten der KS vor einer anderen Anfrage R‘ (im Sinne von happened-before-Relation) stattfindet (d.h. R $\rightarrow$ R‘), dann wird der Zugang zur KS auch in dieser Reihenfolge gewährt

Algorithmen

- Wir werden gleich den Dijkstra‘s Token Ring Algorithmus sehen
  - Setzt einen Ring als Topologie voraus
- Es gibt eine einfachere (nicht stabilisierende) Token-Ring Variante (siehe Tanenbaum Kapitel 6)
  - Ein Token wird einfach an den Nachbarn weitergereicht
  - Der Nachbar muss den Empfang bestätigen
    - Falls tot, wird der übernächste Nachbar genommen
- Der **zentralisierte Algorithmus** ist einfach und praktisch nützlich
  - Wir haben einen Koordinator
  - Alle können mit dem Koordinator kommunizieren

Dijkstra's Token Ring

- Betrachte das Problem des **gegenseitigen Ausschlusses** (**mutual exclusion**) in einem Ring von Prozessen
  - jeder Prozess möchte ab und zu eine **kritische Sektion** (**critical section**) des eigenen Code ausführen – Zugang zu gemeinsamen Ressourcen
  - höchstens einer der Prozesse kann zum gegebenen Zeitpunkt diese Region ausführen
  - jeder Prozess hat ein lokales Prädikat: wenn wahr, hat der Prozess den **Privileg**, seine kritische Region auszuführen
- Problemspezifikation (Prädikat P):
  1. In jeder Konfiguration hat höchstens ein Prozess den Privileg
  2. Jeder Prozess hat den Privileg unendlich oft
- Dieses Problem kann durch die Einführung eines „Tokens“ im Ring gelöst werden

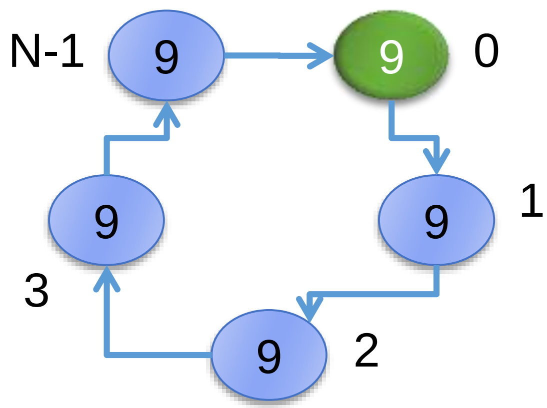
Hier merkt man, warum P auf Folgen von Zuständen definiert ist!

Dijkstra's Token Ring

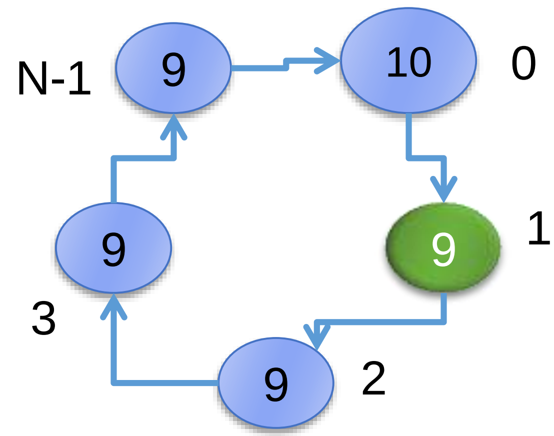
- <u>Systemmodell und Konventionen</u>
  - Der **Zustand s_i** des Prozesses i ist ein Integer in 0,...,K-1,
    - Mit K > N (N = Anzahl der Prozesse)
  - Prozess i>0 kann s_{i-1} (und natürlich s_i) lesen und Prozess 0 kann s_{N-1} lesen
  - Prozess i>0 hat den KS-Privileg, falls $s_i \neq s_{i-1}$
  - Prozess 0 hat den KS-Privileg, falls $s_0 = s_{N-1}$
- <u>Algorithmus</u>
  - Ein privilegierter Prozess kann seinen Zustand ändern (und zugleich den Privileg verlieren) durch:
    - Prozess i>0 setzt $s_i := s_{i-1}$
    - Prozess 0 setzt $s_0 := (s_{N-1}+1) \bmod K$

Dijkstra‘s Token Ring - Beispiel

Fall 1:



Fall 2:



Hier K = 11

- Prozess i>0 hat den KS-Privileg, falls $s_i \neq s_{i-1}$
- Prozess 0 hat den KS-Privileg, falls $s_0 = s_{N-1}$
- Nur ein privilegierter Prozess P (Index i) kann einen **Schritt** ausführen, indem P seinen Zustand so ändert:
  - Falls i>0, P setzt $s_i := s_{i-1}$
  - Falls i = 0, P setzt $s_0 := (s_{N-1}+1) \bmod K$

Thank you.