Distributed Systems

Chapter 2 – Basic Functionality

Chapter 3 – Coordination

- Time and Global States
- Process Synchronization
- Distributed Transactions

Chapter 4 – Quality of Service

Chapter 5 – Middleware

3.1 Time and Global States

- Network Time Protocol (NTP)
- Lamport Timestamps and Vector Timestamps
- Global states: snapshot algorithm and global predicate evaluation

Cooperation and Coordination in Distributed Systems

Communication Mechanisms for the communication between processes

Naming for dynamic binding with communication partners

But... not enough for cooperation:

- ***Synchronization***
- *Coordination algorithms for mutual access, consensus, …*
- *Consistency in transaction processing*
- *…*

- Time measurements for optimization of interactions
- Ordering of events
- Determination of global states

More complicated problems than in central systems!

The Role of Time

A distributed system consists of a number of *processes*

- Each process has a *state* (values of variables)
- Each process takes *actions* to change its state, or to communicate with other processes (send, receive)
- An *event* is the occurrence of an action
- Events *within* a process can be ordered by the time of occurrence
- In distributed systems, also the *time order of events on different machines and between different processes* has to be known

Needed: concept of “global time”, i.e. local clocks of machines have to be synchronized

- Synchronization based on actual (absolute) time
- Synchronization by relative ordering of events
- Distributed global states

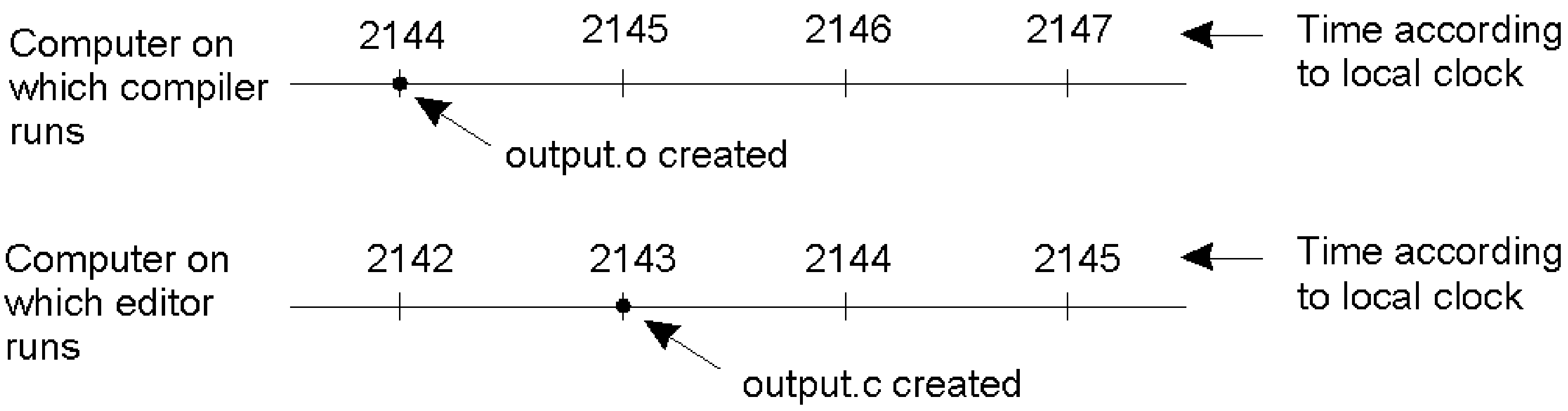
Clock Synchronization

universitätbonn

Clocks in distributed systems are independent

- Some (or even all) clocks are inaccurate
- When each machine has its own clock, an event that occurred after another event may nevertheless be assigned an earlier time.
- How to determine the right sequence of events?

- Example Compiler – synchronization is needed considering the absolute time on all machines:

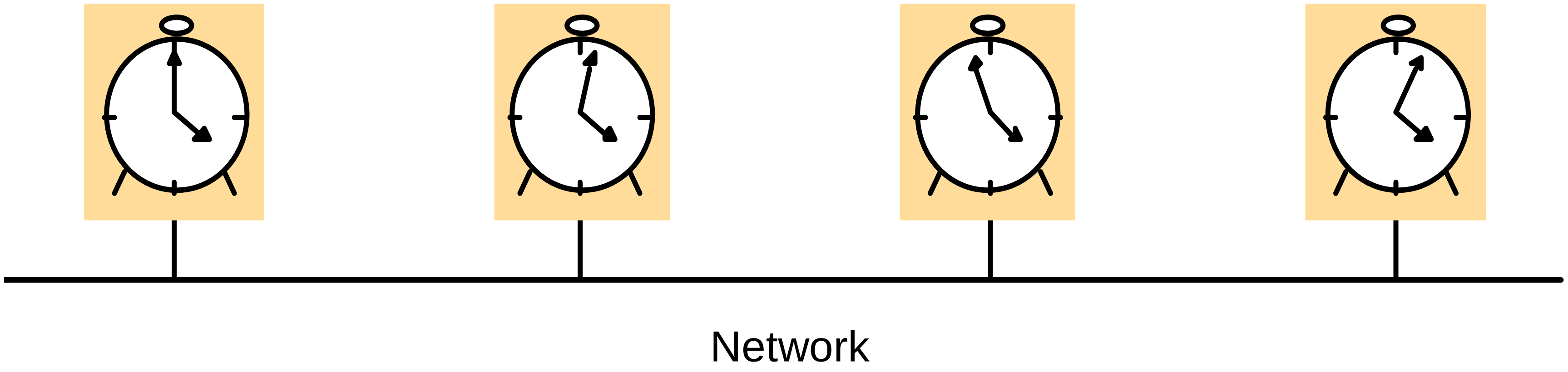


How can we - synchronize clocks with real world?
- synchronize clocks with each other?

Clocks

Necessary for synchronization: assign a *timestamp* with each event

But... how to determine the own resp. all other times in the system?



- *Skew*: the difference between the times on two clocks (at any instant)
- Computer clocks are subject to *clock drift* (they count time at different speeds)
- Clock *drift rate*: the difference per unit of time from some ideal reference clock
- Ordinary quartz clocks drift by about 1 sec in 11-12 days. (10^{-6} secs/sec).
- High precision quartz clocks drift rate is about 10^{-7} or 10^{-8} secs/sec

Universal Coordinated Time (UTC)

International Atomic Time is based on very accurate atomic clocks (drift rate 10^{-13})

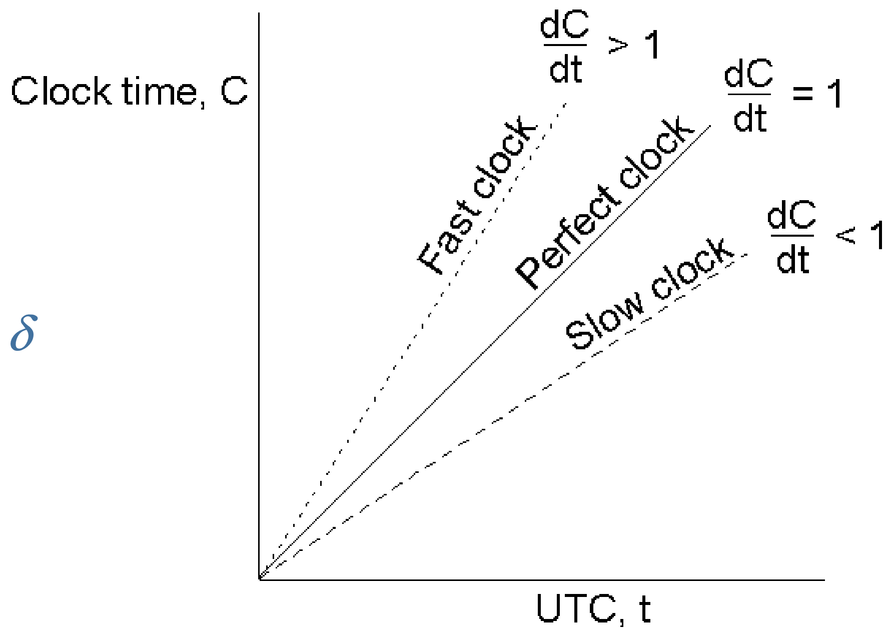
- Problem: "Atomic day" is 3 msec shorter than a solar day
- UTC is an international standard for time keeping solving this problem
- It is based on atomic time, but occasionally adjusted to astronomical time: when the difference to the solar time grows up to 800 msec, an additional leap second is inserted
- UTC is broadcasted from radio stations on land and satellite (e.g. GPS)
- Computers with receivers can synchronise their clocks with these timing signals (*But: only a small fraction of all computers have such receivers!*)
- Problem with received UTC: propagation delay has to be considered
  - Signals from land-based stations (short wave) are accurate up to about 0.1 - 10 milliseconds
  - Signals from land-based stations (long wave) are accurate up to about 30 - 60 milliseconds
  - Signals from satellites (GPS) are accurate up to about 1 microsecond

Clock Synchronization Algorithms

Synchronization principle

- Universal Coordinated Time (as reference time): t
- Clock time on machine p: $C_p(t)$
- Perfect world: $C_p(t) = t$,
  i.e. $dC/dt = 1$

  → Reality: there is a drift in any clock, but based on clock accuracy a maximum drift rate ρ can be specified:
  ρ: $1 - \rho \leq dC/dt \leq 1 + \rho$



- Needed for synchronization: definition of a tolerable skew, the maximum time drift δ
- With this, re-synchronization has to be made in certain intervals: all $\delta/2\rho$ seconds
- How to make such a re-synchronization?

universitätbonn

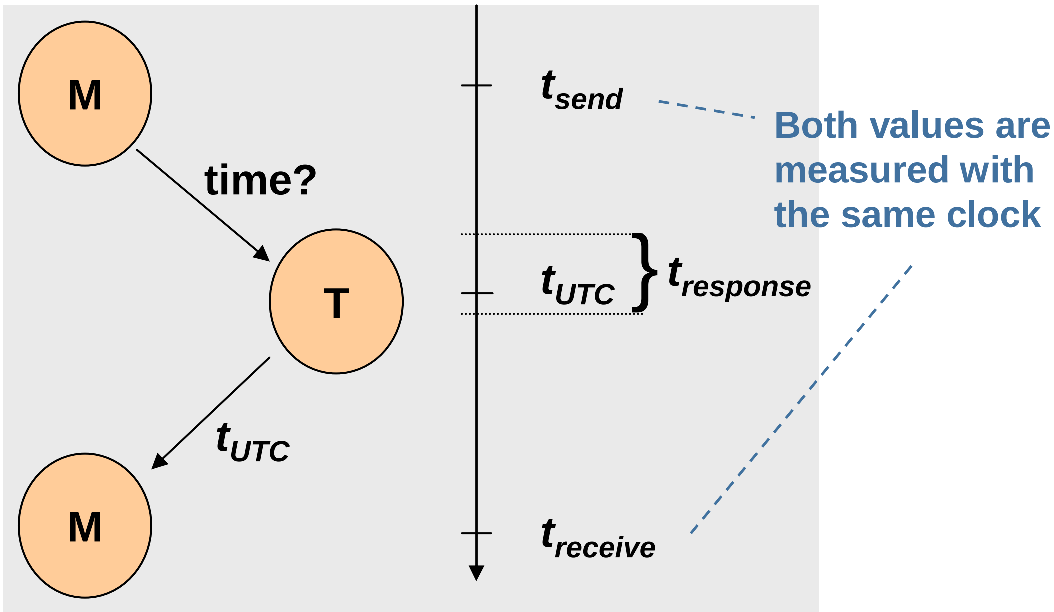
Cristian's Algorithm

There is one central **time server** *T* with a UTC receiver

- All other machines *M* are contacting the time server at least all $\delta/2\rho$ seconds
- *T* responds as fast as it can

M computes current time:

- Hold time t_{send} for sending the request
- Measure time when response with t_{UTC} arrives ($t_{receive}$)
- Subtract service time $t_{response}$ of *T*
- Divide by two to consider only the time since the reply was sent
- Add 'delivery time' to the time t_{UTC} sent by *T*
- Result $t_{synchronous}$ becomes new system time



$$t_{synchronous} = t_{UTC} + \frac{t_{receive} - t_{send} - t_{response}}{2}$$

Consider message run-time, avoid M's time to be moved back

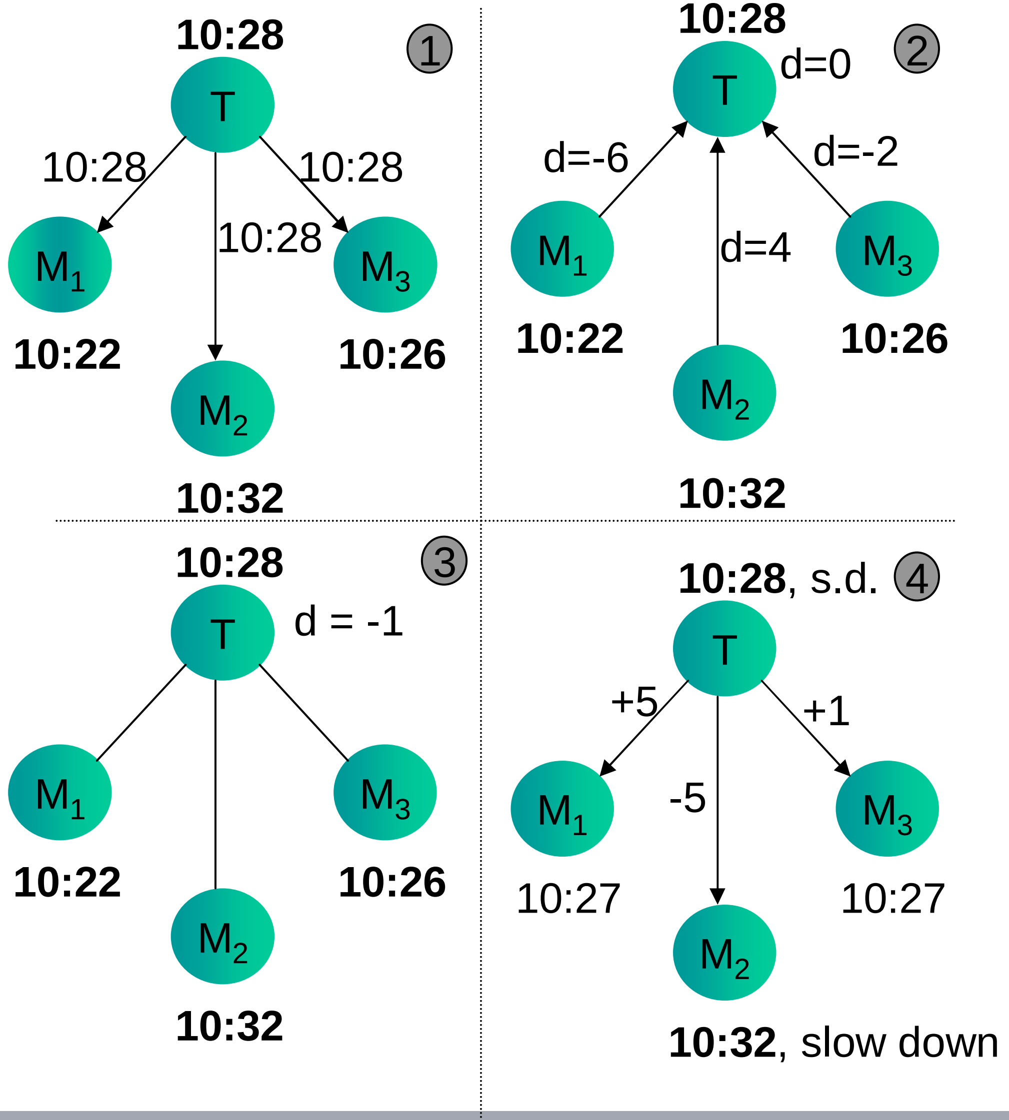
The Berkeley Algorithm

Another approach (Berkeley Unix):

- *Active* time server
- No UTC source needed

1. Time server sends its time to all machines
2. The machines answer with their current deviation from the time server
3. The time server sums up all deviations and divides by the number of machines (including itself!)
4. The new time for each machine is given by the mean time

   Important: fast clocks are not moved back, but instructed to move slower



Distributed Algorithms

Problems with Cristian/Berkeley: use of a *centralized server*, assumption of *symmetric message runtimes*; mainly for use within Intranets

Simple mechanism for decentralized synchronization (based on Berkeley Algorithm):

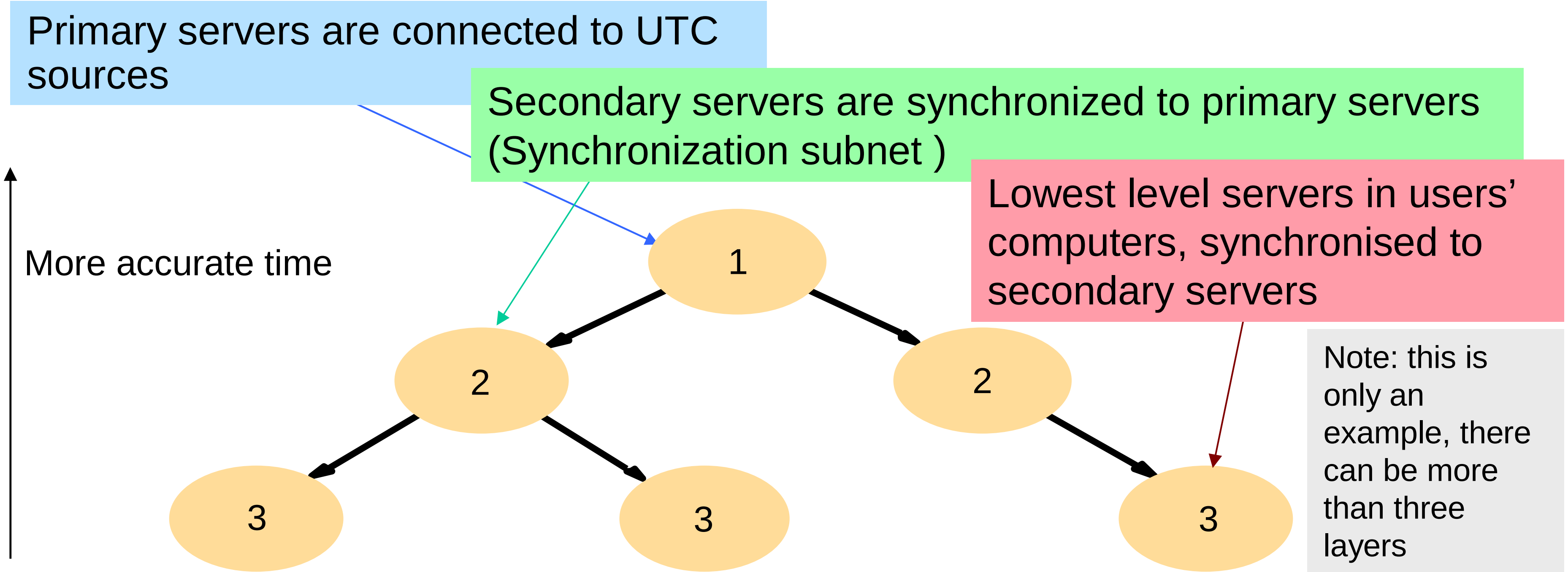
- Divide time into fixed-length synchronization intervals
- At the beginning of each interval all machines
  - Broadcast their current time
  - Collect all values of other machines arriving in a given time span
  - Compute the new time
    - by simply averaging all answers, or
    - by discarding the *m* highest and the *m* lowest answers before averaging (to protect against faulty clocks), or
    - by averaging values corrected by an estimation of their propagation time.
- ... but: in large-scale networks, the broadcasting is difficult, and message runtimes are not considered

→ *Widely used algorithm in the Internet: Network Time Protocol (NTP)*

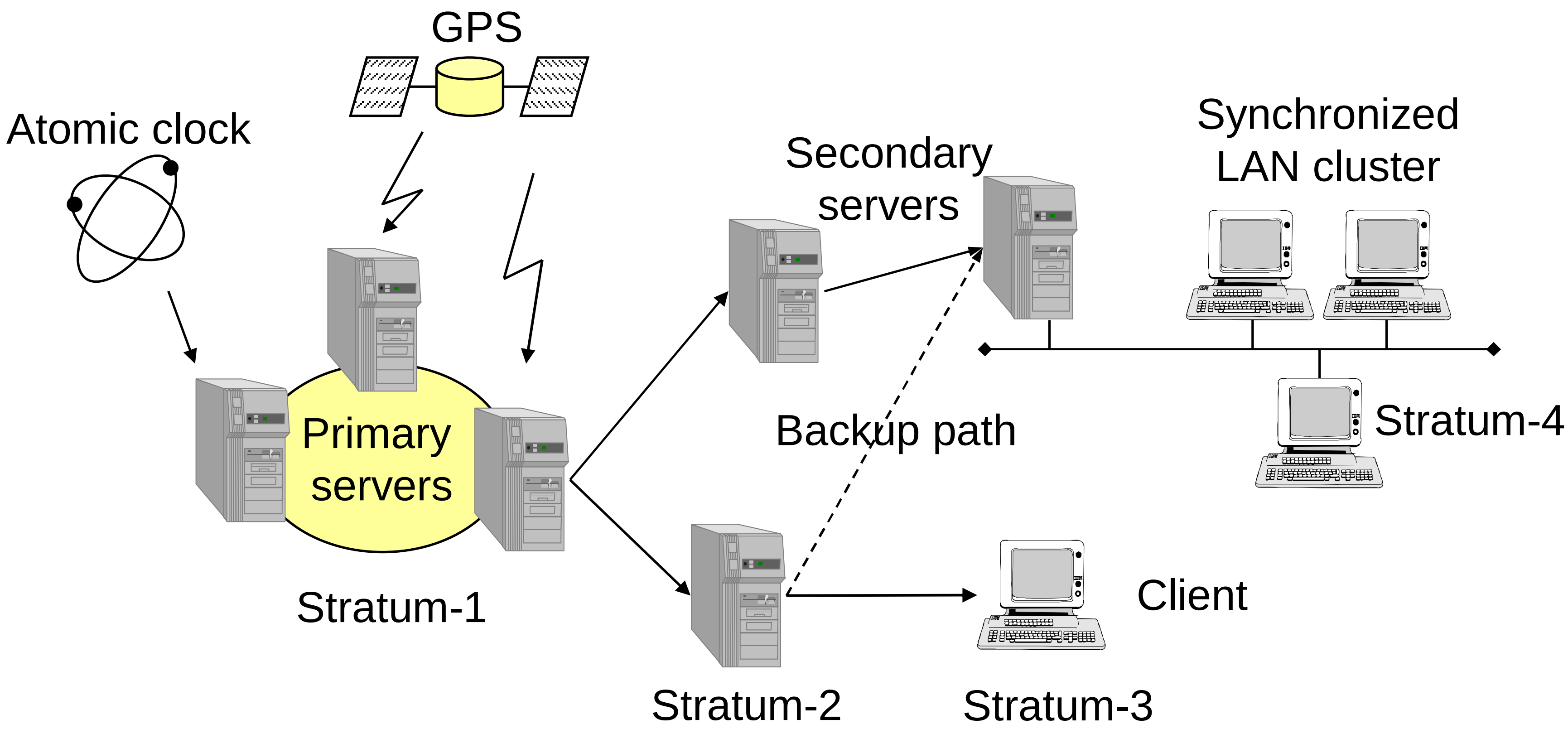
Network Time Protocol (NTP)

NTP is a time service designed for the Internet

- *Reliability* by using redundant paths
- *Scalable* to large number of clients and servers
- *Authenticates* time sources to protect against wrong time data
- NTP is provided by a network of time servers distributed across the Internet
- Hierarchical structure: synchronization subnet tree



Network Time Protocol (NTP)



- Exchange of timestamps between time servers and clients via UDP
- Levels in the synchronization subtree also are called *Stratum*

NTP – Synchronization of Servers

The synchronization subnet can reconfigure if failures occur, e.g.

- A primary that loses its UTC source can become a secondary
- A secondary that loses its primary can use another primary

Modes of synchronization:

- *Multicast*

  A server within a LAN multicasts time to others which set clocks assuming some delay (not very accurate)
- *Procedure call*

  A server accepts requests from other computers (like in Cristiain's algorithm). Higher accuracy than using multicast (and a solution if no multicast is supported)
- *Symmetric*

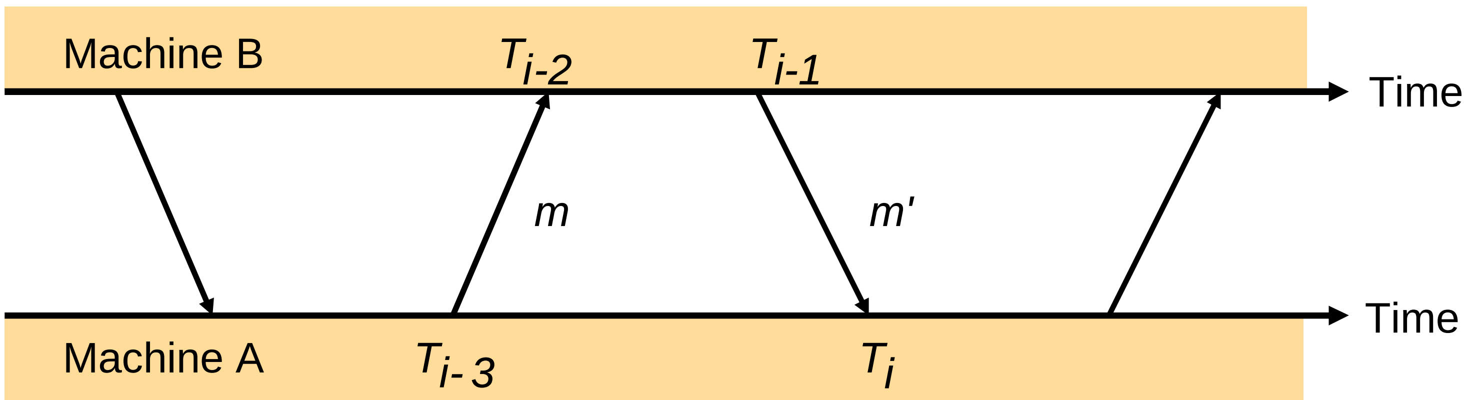
  Pairs of servers exchange messages containing time information; used when very high accuracy is needed (e.g. for higher levels)

All modes use UDP to transfer time data

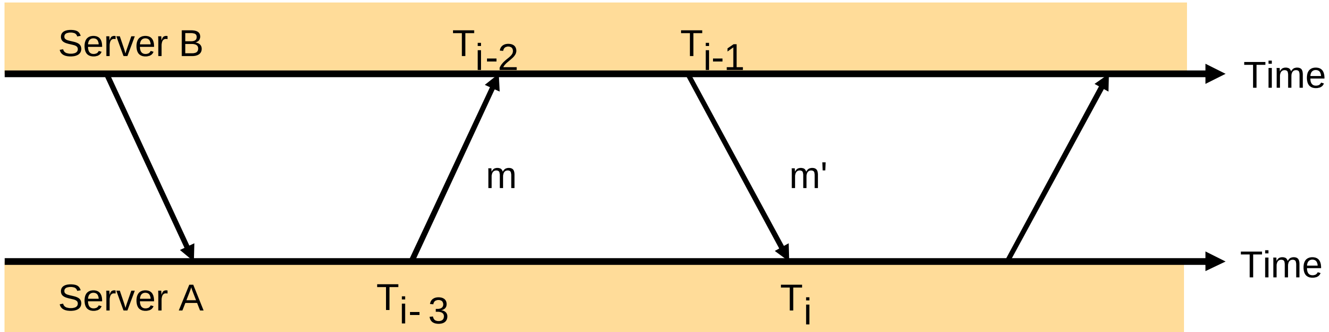
Messages exchanged between a Pair of NTP Peers

UTC is sent in messages between the servers

- Each message contains timestamps of recent events, e.g. for message *m'*:
  - Local times of *Send* (T_{i-3}) and *Receive* (T_{i-2}) of previous message *m*
  - Local time of *Send* (T_{i-1}) of current message *m'*
- Recipient of *m'* notes the time of receipt T_i (it then knows T_{i-3}, T_{i-2}, T_{i-1}, T_i)
- In symmetric mode there can be a non-negligible delay between messages



Accuracy of NTP



For each pair of messages between two servers, NTP estimates

- an offset o_i between the two clocks and
- a delay d_i (total time for transmitting the two messages, which take t and t')
- You have: $T_{i-2} = T_{i-3} + t + o$ and $T_i = T_{i-1} + t' - o$
  for the current offset o between A and B

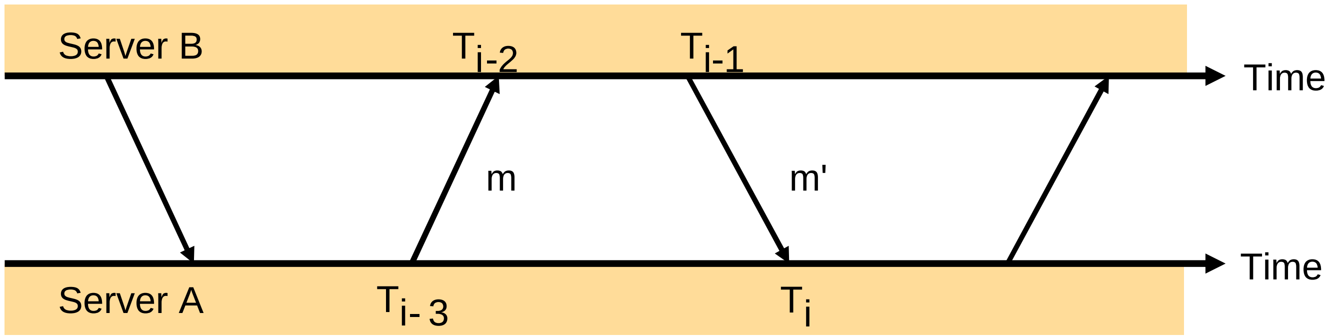
This gives us (by adding the equations) :

$d_i = t + t' = T_{i-2} - T_{i-3} + T_i - T_{i-1}$

Also (by subtracting the equations)

$o = o_i + (t' - t)/2$ where $o_i = (T_{i-2} - T_{i-3} - T_i + T_{i-1})/2$

Accuracy of NTP



- Using the fact that *t*, *t'*>0 it can be shown that

$$o_i - d_i/2 \leq o \leq o_i + d_i/2$$

- Thus o_i is an estimation of the offset and d_i is a measure of the accuracy
- NTP servers filter pairs $<o_i, d_i>$, estimating reliability of time servers from variations in pairs and accuracy of estimations by low delays d_i, allowing them to select peers
- Accuracy of 10s of milliseconds over Internet paths, 1 millisecond on LANs

Lamport Timestamps

The absolute time is not needed in any case. Often enough:
ordering of events only with respect to *logical clocks*

Relation: **happens-before**: $a \rightarrow b$ means that "a happens before b"

(Meaning: *all processes agree* that event a happens before event b)

1. $a \rightarrow b$ is true, when both events occur in the same process
2. $a \rightarrow b$ is true, if one process is sending a message (event a) and another process is receiving this message (event b)
3. $\rightarrow$ is transitive
4. neither $a \rightarrow b$ nor $b \rightarrow a$ is true, if they occur in two processes which do not exchange messages (**Concurrent** Processes/Events, notation: $a||b$)

Needed: assign a (time) value $C(a)$ to an event a on which all processes agree, with $C(a) < C(b)$ if $a \rightarrow b$

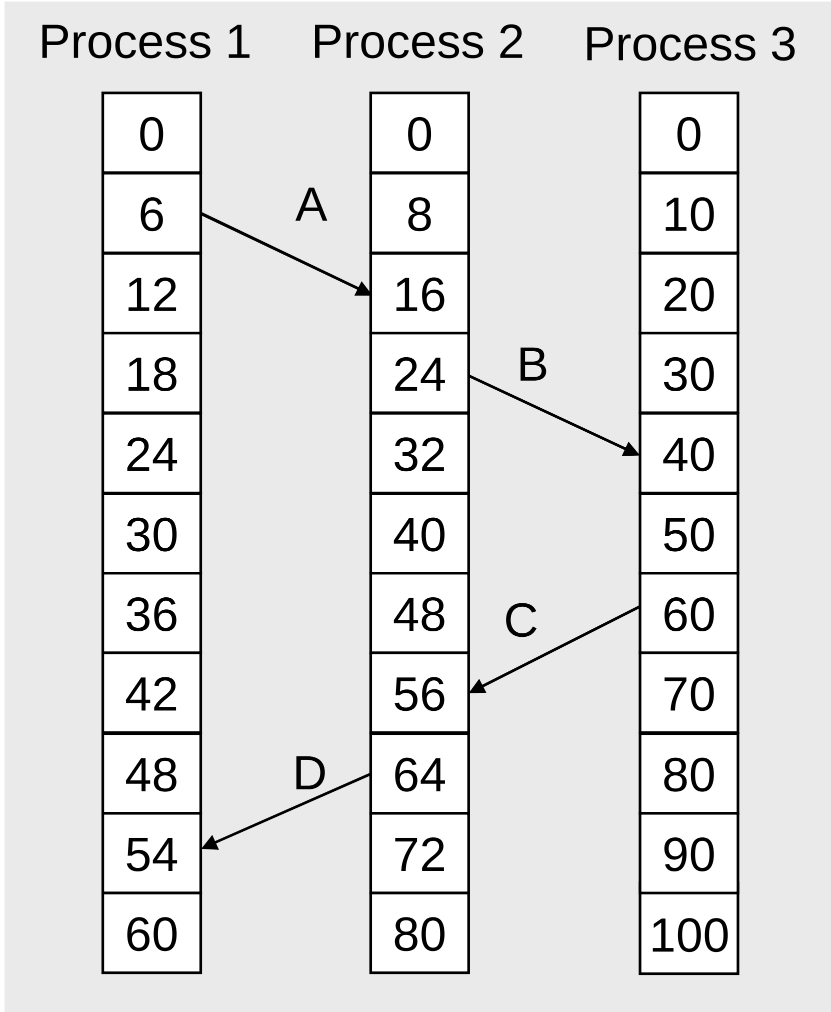
⇨ **Lamport's Algorithm**

Lamport's Algorithm

Dealing with “happens-before”

- Processes count time in different speed
- Events inside a single process can be ordered due to the local time of their occurrence
- Problems only exists when processes interact – e.g. messages C and D arrive before they sent

  → “happens-before” is violated!



Lamport's Algorithm

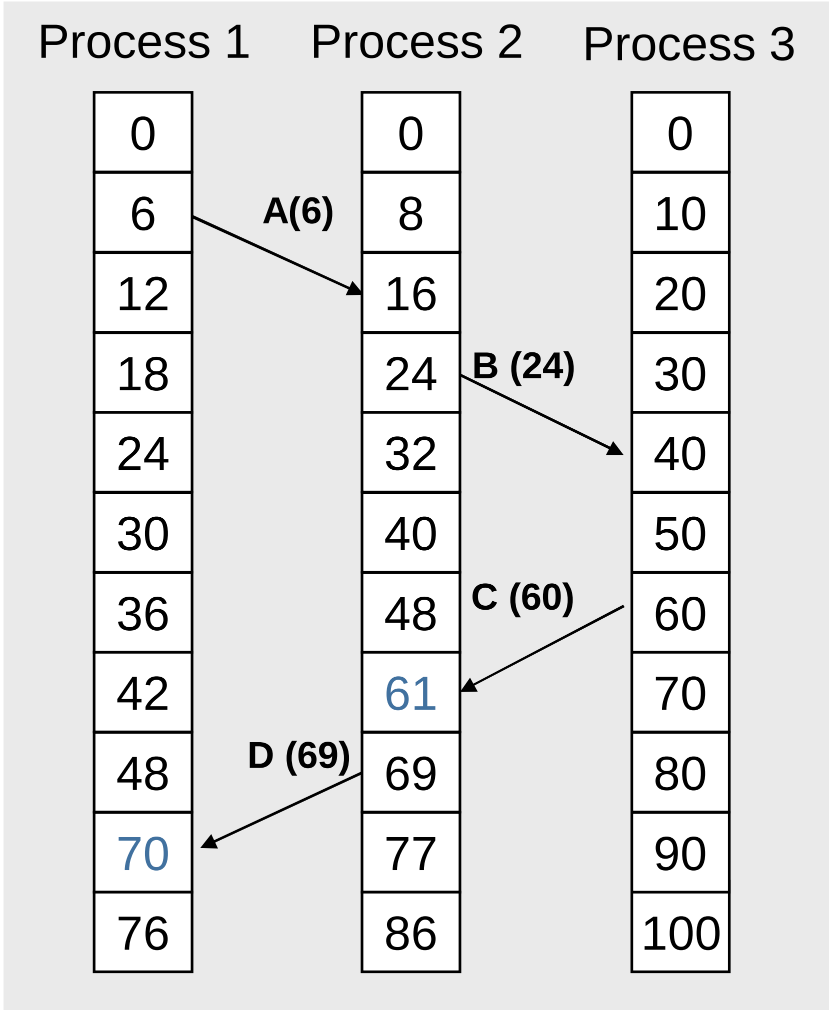
Solution of Lamport:

- Send a timestamp (local time) with every message
- Arriving before sending violates the When receiving a message with a timestamp *C(sent)* higher than the own current time *C(receive)*, forward the local clock to the next higher value:

  C(receive) = *C(sent)* + 1
- Go on counting time with the adjusted value

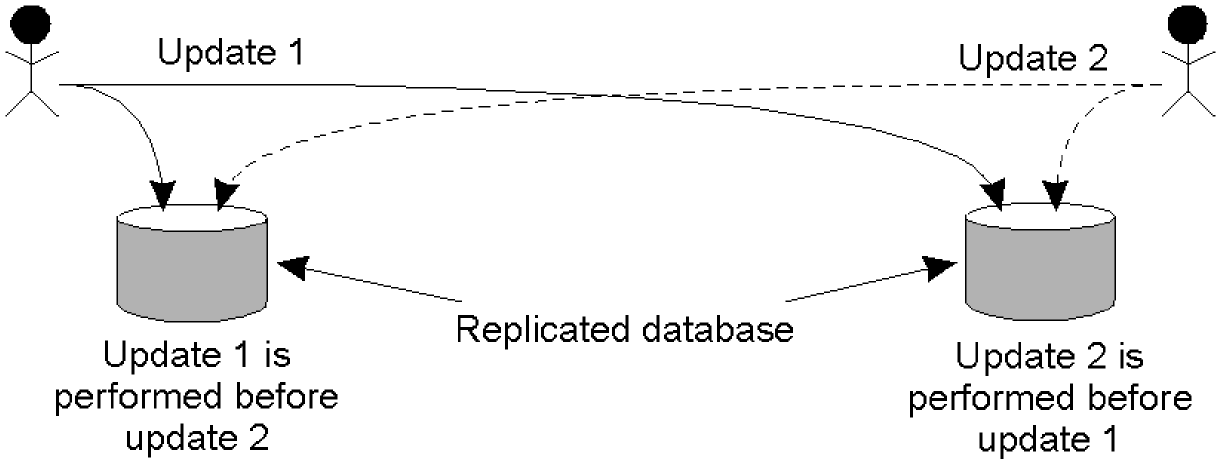
To achieve unique times:

- For all events *a* and *b* it has to be guaranteed that $C(a) \neq C(b)$
- This can be achieved by attaching some process identification to the local time (eg. 14052010141530222.1300)



Application of Lamport Timestamps

Replicated database: updates have to be performed in a certain order



Use Lamport's Timestamps to implement *totally-ordered broadcast*

- Each message is time stamped with the current (Lamport) time of the sender
- The messages are sent to all receivers (and to the sender itself!)
- Received messages are ordered by their timestamps
- Receivers broadcast acknowledgements for the message with the currently smallest timestamp
- Only after receiving acknowledgements from all receivers, the message with the lowest timestamp is read by the processes

Enhancement: Vector Timestamps

Problem with Lamport timestamps: they do not capture **causality**

⇨ Using *vector timestamps*

Definition:

A *vector timestamp VT(a)* for event *a* is in relation *VT(a)* < *VT(b)* to event *b*, if *a* is known to *causally precede b*.

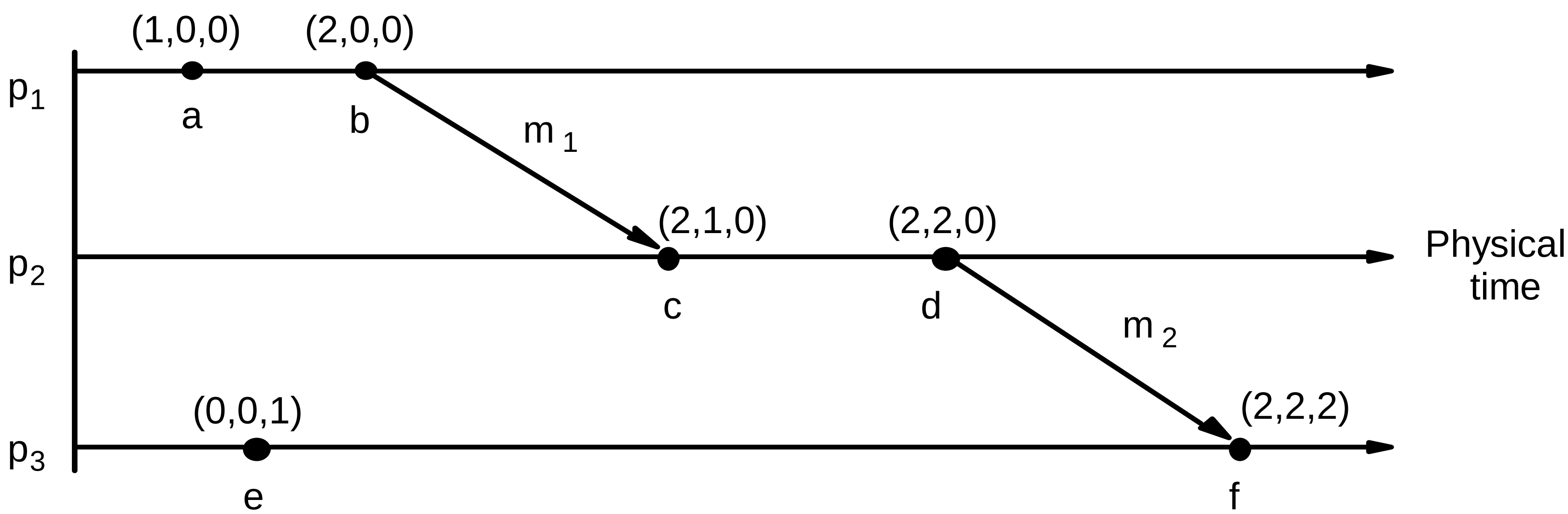
Vector time is constructed by each process P_i as a vector VT_i with:

1. $VT_i[i]$ is the number of events that have occurred so far at P_i
2. If $VT_i[j] = k$ then P_i knows that k events have occurred at P_j
   - When P_i sends a message *m*, then it sends along its current VT_i *(History of the sender)*
   - This timestamp vector tells the receiver P_j how many events in other processes have preceded *m*
   - P_j adjusts its own vector for each k to $VT_j[k] = max\{VT_j[k], VT_i[k]\}$ (*Merging of the own history and the history of the message m*)
   - Add 1 to entry $VT_j[j]$ for the event of receiving *m*

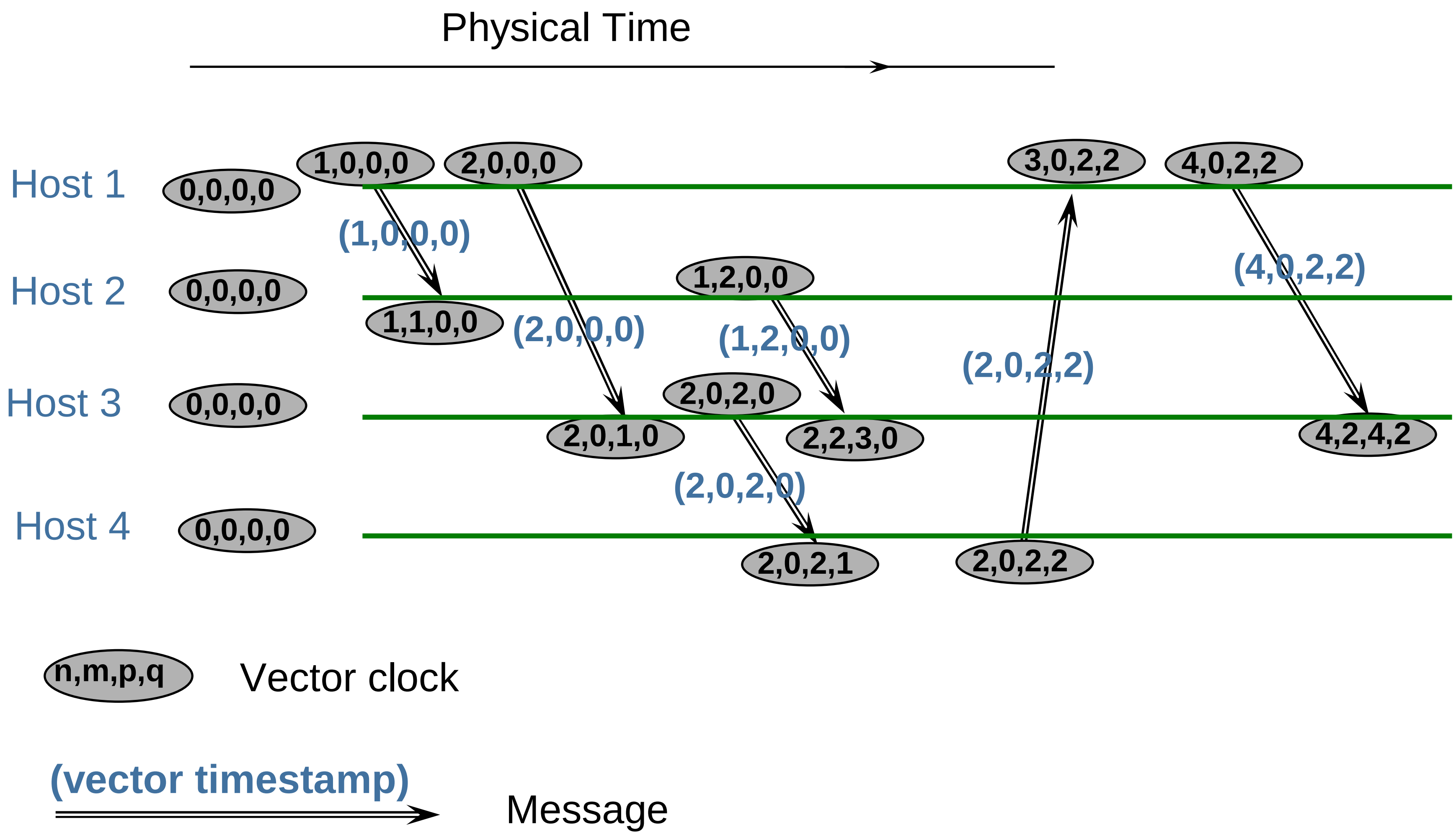
Vector Timestamps - Example

Vector clock VT_i at process p_i is an array of N integers

- Initially $VT_i[j] = 0$ for $i, j = 1, 2, \ldots N$
- Before p_i timestamps an event it sets $VT_i[i] := VT_i[i] + 1$
- p_i piggybacks VT_i on every message it sends
- When p_j receives (m, VT_i) it sets $VT_j[j] = VT_j[j] + 1$ for the receiving event and afterwards $VT_j[k] := \max(VT_i[k], VT_j[k])$ $k = 1, 2, \ldots N$



Vector Timestamps - Example



Vector Arithmetics

Meaning of *VT(a)* ≤ *VT(b)*

- Event *a* belongs to the causal history of event *b*
- "Cone" of *b* contains cone of *a*
- Meaning of "≤":

$$VT_1 \leq VT_2 \Leftrightarrow \forall i : VT_1[i] \leq VT_2[i]$$
$$VT_1 \,||\, VT_2 \Leftrightarrow \neg(VT_1 \leq VT_2) \wedge \neg(VT_2 \leq VT_1)$$
$$VT_1 < VT_2 \Leftrightarrow VT_1 \leq VT_2 \wedge VT_1 \neq VT_2$$

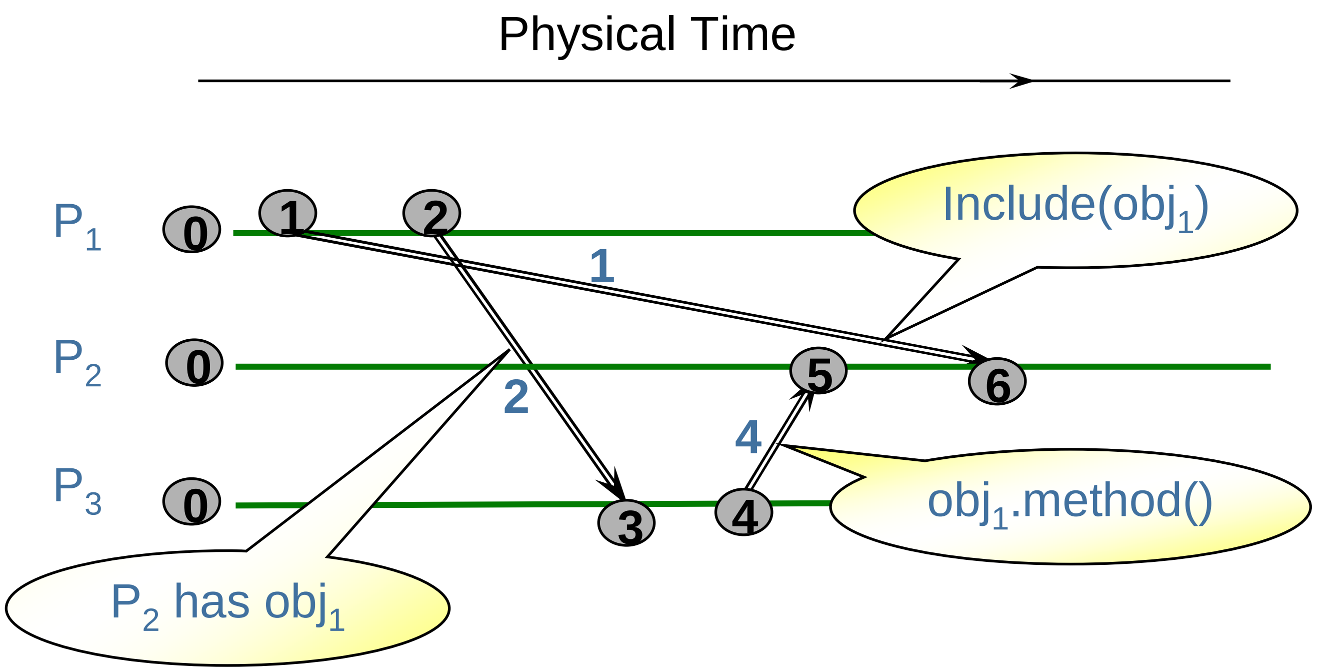
$$\begin{bmatrix} 1 \\ 3 \\ 4 \\ 3 \end{bmatrix} \leq \begin{bmatrix} 2 \\ 3 \\ 6 \\ 4 \end{bmatrix}$$

comparable, component by component

$$\begin{bmatrix} 1 \\ 3 \\ 4 \\ 3 \end{bmatrix} || \begin{bmatrix} 2 \\ 3 \\ 6 \\ 1 \end{bmatrix}$$

concurrent, not comparable component by component

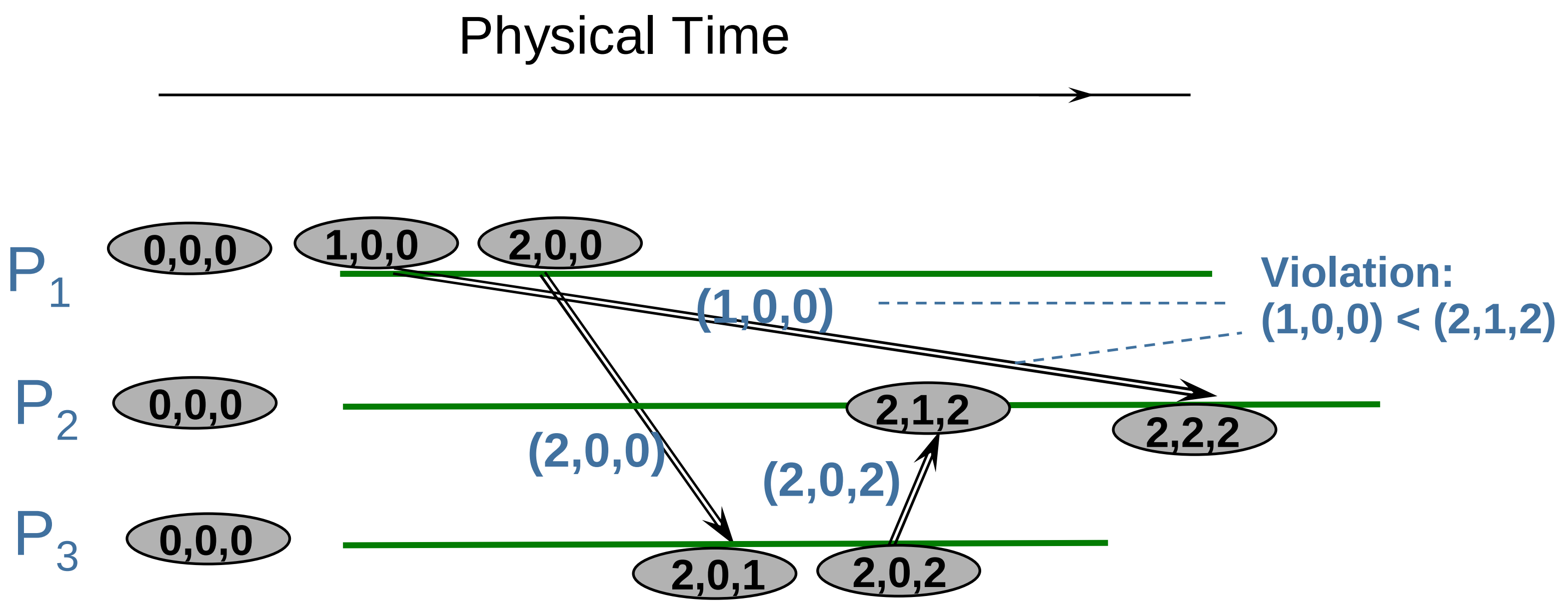
Application: Causality Violation



Vector timestamps can be used for detecting causality violations:

- Causality violation occurs when the order of messages causes an action based on information that another host has not yet received
- In other words: causality violation is not considering causal relationships in message processing

Detecting Causality Violation



Potential causality violation can be detected by vector timestamps

- When P_2 receives the message from P_3, it can see that there is a gap between the own history (0,0,0) and the history of the message (2,0,2)
- The processing could be delayed till we receive another message filling the gap

Causal Broadcast with Vector Timestamps (CBCAST)

Vector timestamps also can be used to implement causal broadcast:

- Each message *m* contains a timestamp *vt(m)* with the vector time of the sender
- By means of *vt(m)* the receiver is able to determine which messages the sender has “seen” before broadcasting *m*
- If the receiver has “seen” less messages than the sender, delivery of *m* is delayed

Algorithm **CB** – executed by every process p_i:

Initialization:

$VT(p_i) := (0,0,\ldots,0)$ /* nothing happened before */

To execute `broadcast(C,m)`:

$VT(p_i)[i] := VT(p_i)[i] + 1$

broadcast(R,m,vt(m)), where vt(m) = VT(p_i)

On receiving a messages *m*:

upon *deliver(R,m,vt(m)) from* p_j do

delay until $\forall k \neq j: vt(m)[k] \leq VT(p_i)[k] \wedge vt(m)[j] = VT(p_i)[j]+1$

`deliver(C,m)`

$VT(p_i) := \max(vt(m), VT(p_i))$

Global State

Often required: not only ordering of events, but determination of the current **global state** of a system, e.g. in

- Deadlock detection
- Garbage collection
- Termination detection

Attempt 1: freeze the system, record all local states of the involved machines and send them to a coordinator who constructs a global state

→ Unpractical – freezing the system is not an option

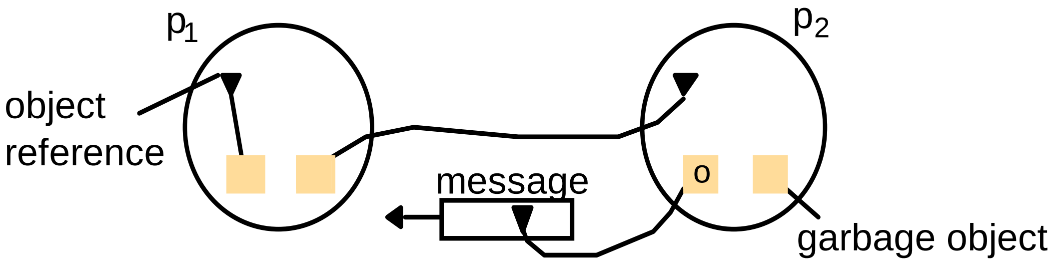
Attempt 2: agree on a time (e.g. NTP time) at which all involved machines do a recording of local state and send the local states to a coordinator

→ Impossible – no accurate clock synchronization given to guarantee all local recordings to occur at exactly the same time

Global State Examples

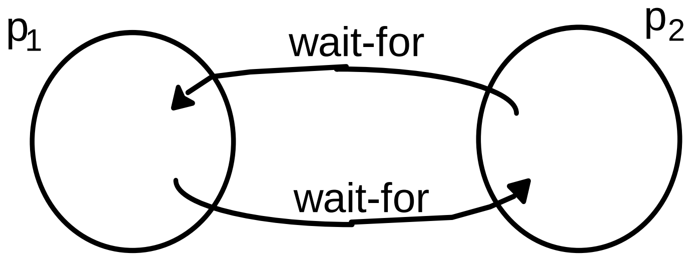
Distributed garbage collection:

Object *o* seems to be garbage, but it has sent a message containing a reference to it



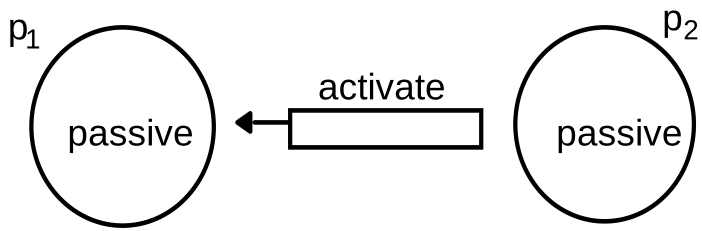
Distributed deadlock detection:

Both processes are waiting for a message from the other process



Distributed termination detection:

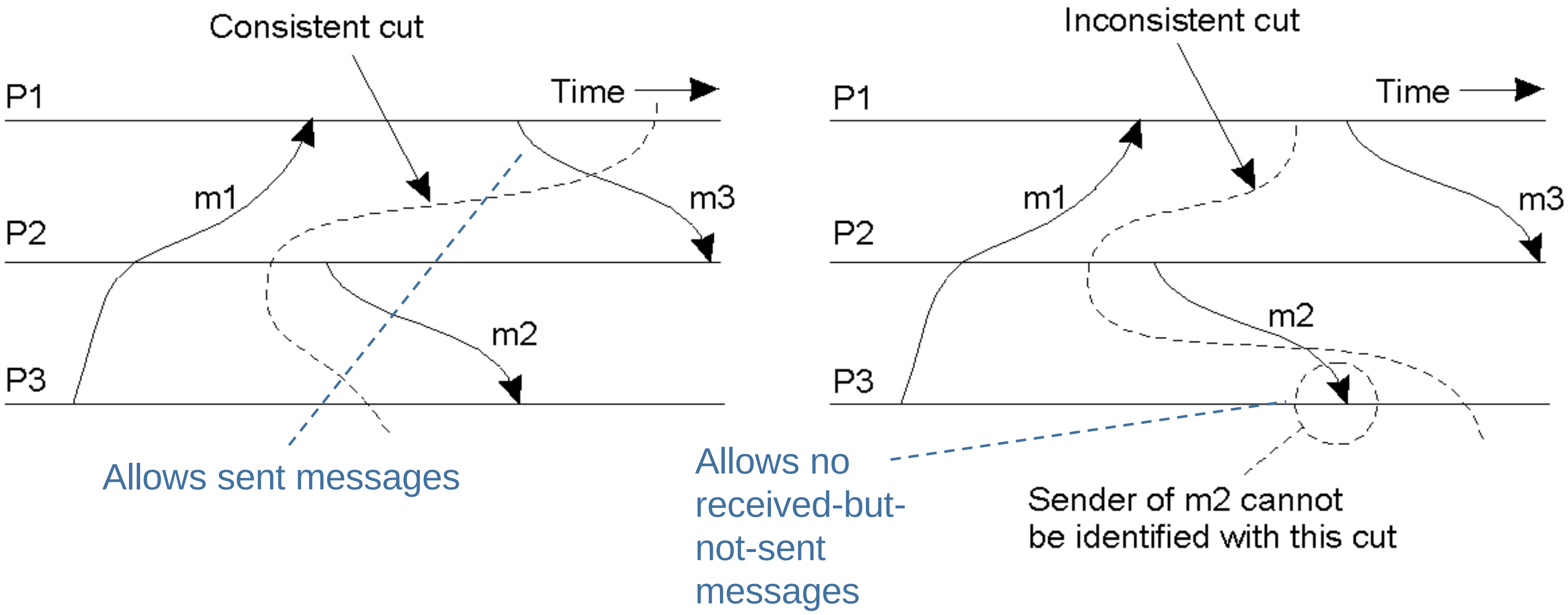
Both processes are passive and seem to be terminated, but in fact there is a message sent by p_2 to activate p_1



Global State

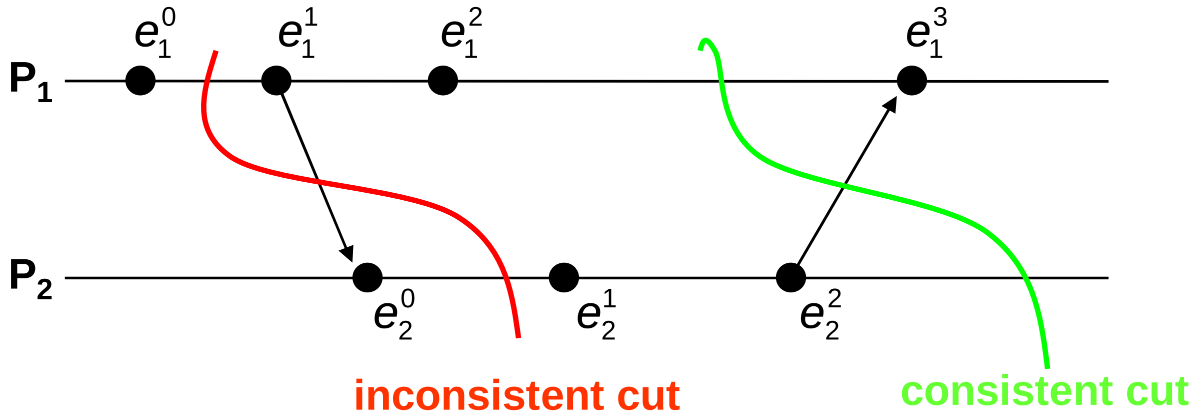
Problem with getting a global state: there is no global time!

- To do: get a consistent global state from lots of local states recorded at different real times
- Needed for global state: **cut**



- A global state is consistent if it corresponds to a consistent cut

Consistent Cuts



What is a consistent cut?

- We have a system of *N* processes p_i (i = 1, … *N*)
- Events *e*: send, receive, or local events
- e_i^j: *j*-th event in process p_i
- Execution of a process is characterized by its *local history*

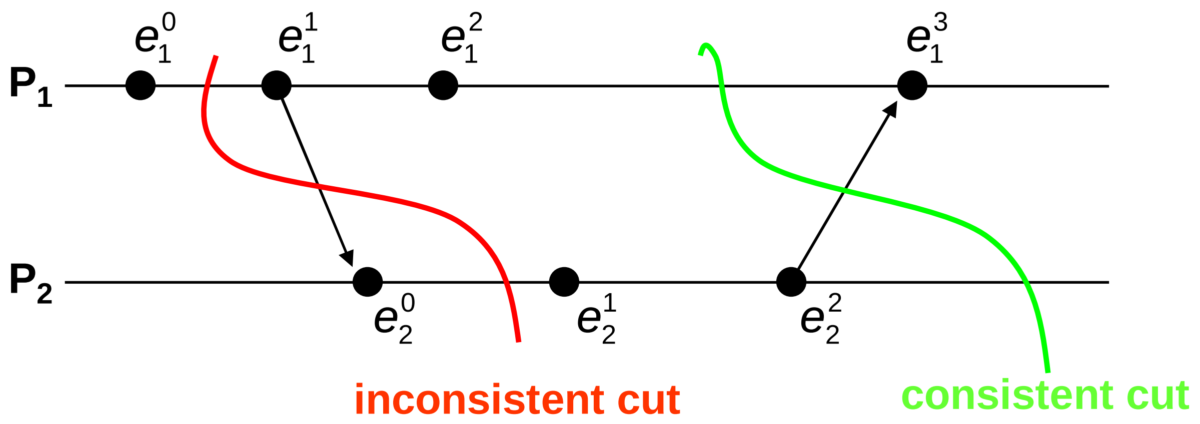
$$history(p_i) = h_i = \langle e_i^0, e_i^1, e_i^2, \ldots \rangle$$

- Finite *prefix* of a history:

$$h_i^k = \langle e_i^0, e_i^1, \ldots, e_i^k \rangle$$

Consistent Cuts

universitätbonn



What is a consistent cut?

- The *global history* is a union of the individual local histories

$$H = h_i \cup h_2 \cup \ldots \cup h_N$$

- A *cut* is a union of prefixes of local process histories

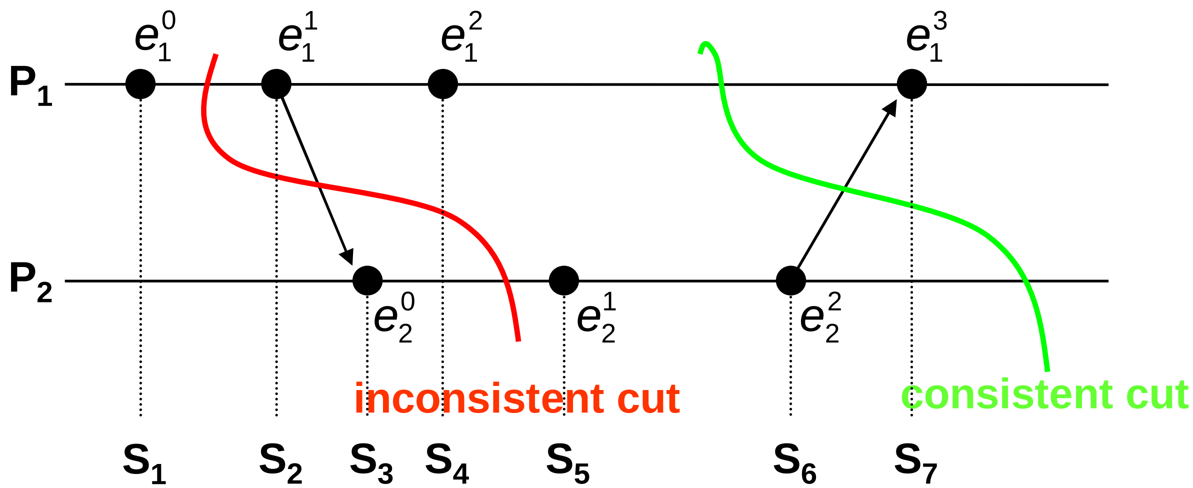
$$C = h_1^{c_1} \cup h_2^{c_2} \cup \ldots \cup h_N^{c_N}$$

- *Frontier* of cut C is defined as

$\{e_i^{Last(i)} : i = 1, \ldots N\}$, where $Last(i)$ is the last event processed by p_i in C

Consistent Cuts and Global State

universitätbonn



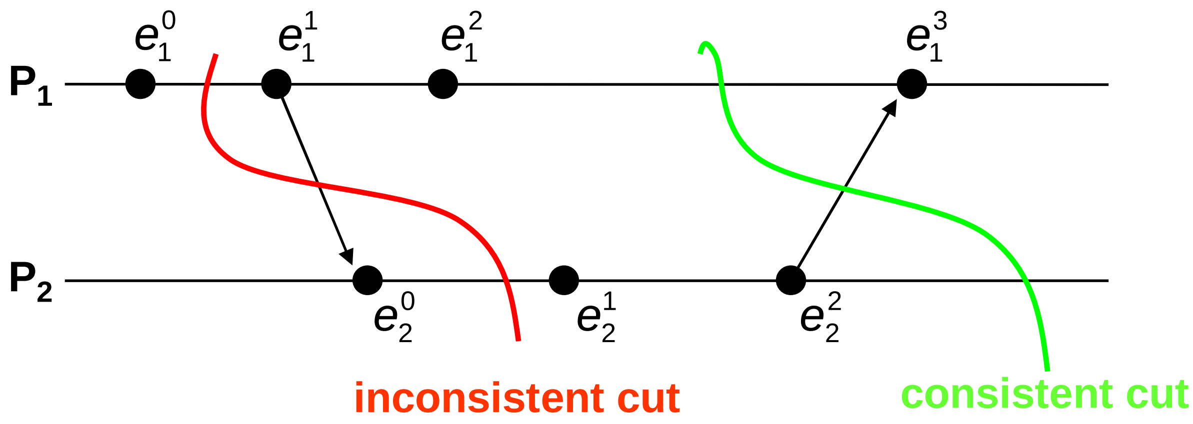
What is a consistent cut?

- *Cut C is consistent* if for all event *e* that *C* contains, *C* also contains all events that happened before *e*: $\forall e \in C : f \rightarrow e \Rightarrow f \in C$

What is a global state?

- *Consistent global state* corresponds to a consistent cut
- Execution may be modeled as a series of state transitions:
  - Each transition relates to one event at some process
  - Simultaneous events could be distinguished with some process identifier

Linearization



Linearization is an ordering of events in a global history that is consistent with the happened-before relationship

→ only passes through consistent global states

Possible linearizations: $\langle e_1^0, e_1^1, e_2^0, e_2^1, e_1^2, e_2^2, e_1^3 \rangle$

$\langle e_1^0, e_1^1, e_1^2, e_2^0, e_2^1, e_2^2, e_1^3 \rangle$

No linearizations: $\langle e_1^0, e_2^0, e_2^1, e_1^1, e_1^2, e_2^2, e_1^3 \rangle$

$\langle e_1^0, e_1^1, e_1^2, e_1^3, e_2^0, e_2^1, e_2^2 \rangle$

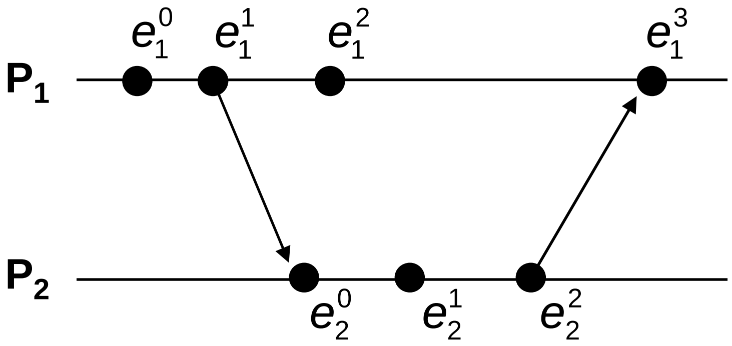
Reachability

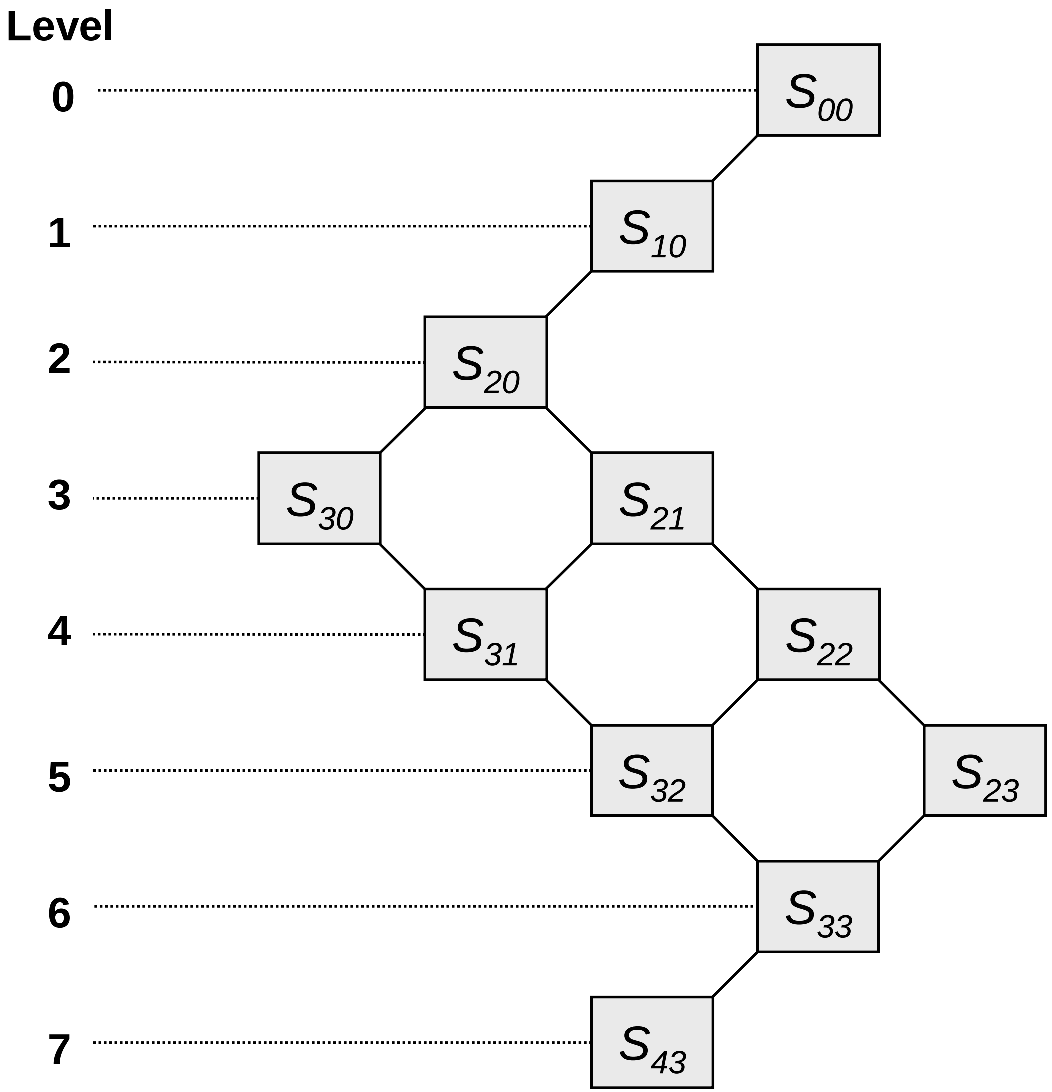
A global state *S'* is *reachable* from a global state *S* if there is a linearization that passes through *S* and afterwards through *S'*

Reachability can be depicted by an *N*-dimensional lattice of global states:

- Nodes: global states
- Edges: possible transitions between states
- Levels: Global states in adjacent levels only differ in one event – linearization traverses from state *S* only to states *S'* on the next level

Reachability





S_{ij}: global state after i events at P_1 and j events at P_2

Global State Predicates

Detecting a condition like “Deadlock exists”, “System is terminated”, “Object is garbage” amounts to evaluating a *global state predicate*:

- *Global predicate*: function mapping from the global states of a system *P* to {`true`, `false`}
- *Stable predicate*: once the system enters a state *S* in which the predicate is `true`, it remains `true` in all future state reachable from *S*

Assume:

- S_0 is the initial state of system *P*
- α is undesirable property of *P*’s global state (e.g. *P* is deadlocked)
- β is desirable property of *P*’s global state (e.g. *P* reaches termination)

Interesting properties:

- *Safety*: “something bad will never happen” – e.g. α evaluates to `false` for all states reachable from S_0
- *Liveliness*: “something good will eventually happen” – for any linearization *L* starting in state S_0, e.g. β evaluates to `true` for some state S_L reachable from S_0

Distributed Snapshot

Chandy/Lamport: **distributed snapshot algorithm**

Record a set of process and channel states, given a set of processes p_i, such that the recorded *global state is consistent* even though the recorded local states have *never appeared at the same time* in execution.

Assumptions:

- No process or channel failures occur, each message is eventually received exactly once
- There is a communication path between any two processes
- Communication channels are unidirectional and FIFO-ordered (each process has unidirectional incoming and outgoing channels)
- Any process may initiate the snapshot algorithm
- Snapshot does not interfere with normal execution
- Each process is able to record its state and the state of its incoming channels (no central collection of state information)

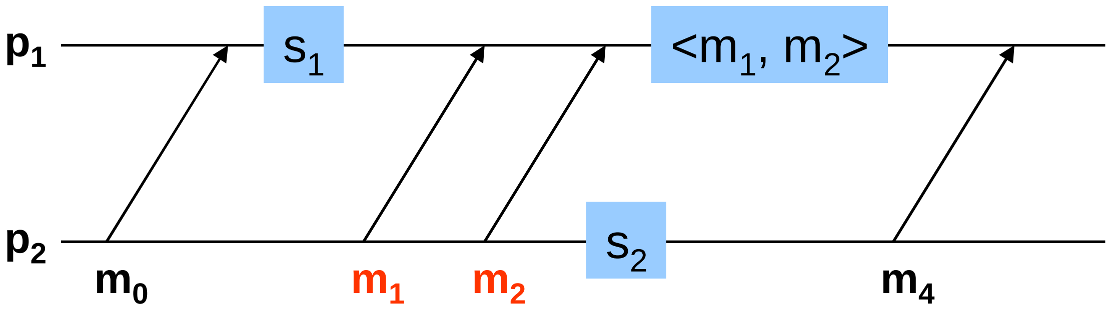
Distributed Snapshot

What is to be achieved?

- Prevent from recording the receiving of a message for which the sending is not recorded
- Messages in transit have to be recorded “in the channel”

Basic idea: each process records…

- its *process state* and
- a *set of messages* sent to it: for each of *p*’s incoming channels, it is recorded any message…
  - ➢ that arrived after *p* recorded it state, and
  - ➢ that was sent before the sender recorded its own state

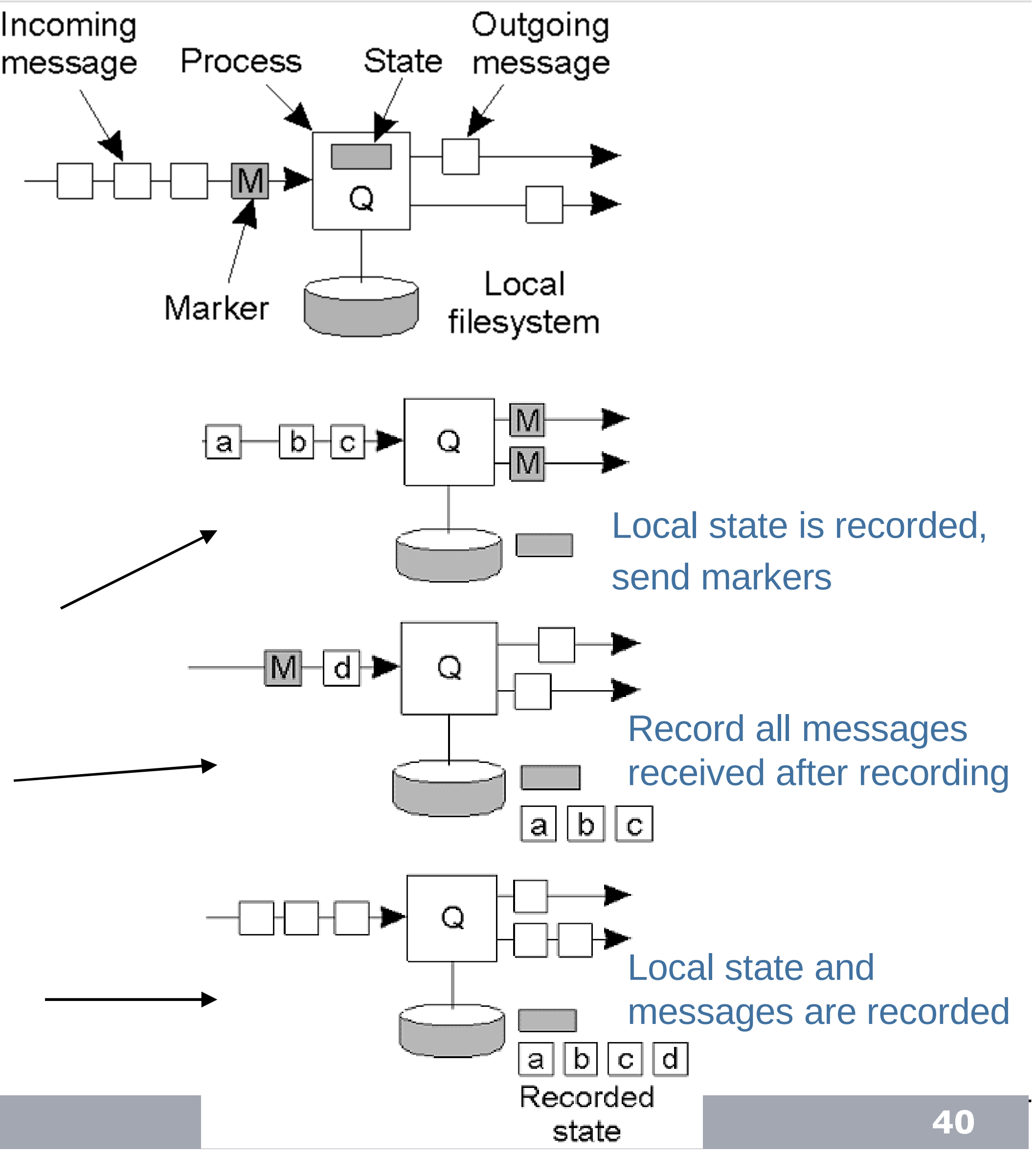


How to implement?
→ use *markers*

Distributed Snapshot

Taking a snapshot:

- Any process *P* can initialize the snapshot by recording its local state
- *P* sends a marker to each process to which he has a communication channel
- *Q* receives marker
    - ➢ First marker received → record local state and send a marker on each outgoing channel
    - ➢ All other markers: record all incoming messages for each channel
    - ➢ One marker for each incoming channel received: stop recording and send results to *P*



Snapshot Algorithm of Chandy/Lamport

Marker receiving rule for process p_i

```
On pi's receipt of a marker message over channel c:
   if (pi has not yet recorded its state) it
      records its process state now;
      records the state of c as the empty set;
      turns on recording of messages arriving over other incoming
      channels;
   else
      pi records the state of c as the set of messages it has
      received over c since it saved its state.
   end if
```

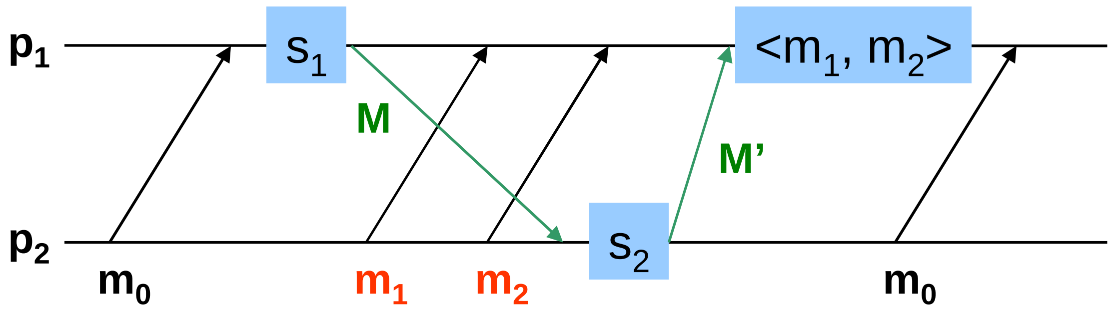
Marker sending rule for process p_i

```
After pi has recorded its state, for each outgoing channel c:
    pi sends one marker message over c
   (before it sends any other message over c).
```

Distributed Snapshot

Algorithm produces a consistent cut:

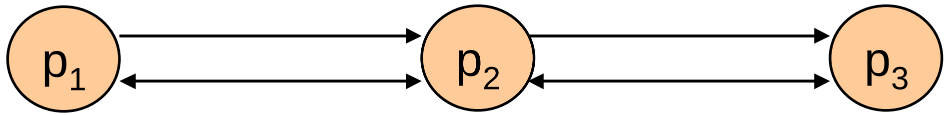
- Maker *M* sent over a FIFO channel after recording of s_1 but before transmitting any other message avoids “messages from the future” produced at p_1
- Marker *M’* sent over a FIFO channel after recording of s_2 but before sending the next message (m_4) avoids producing “messages from the future”

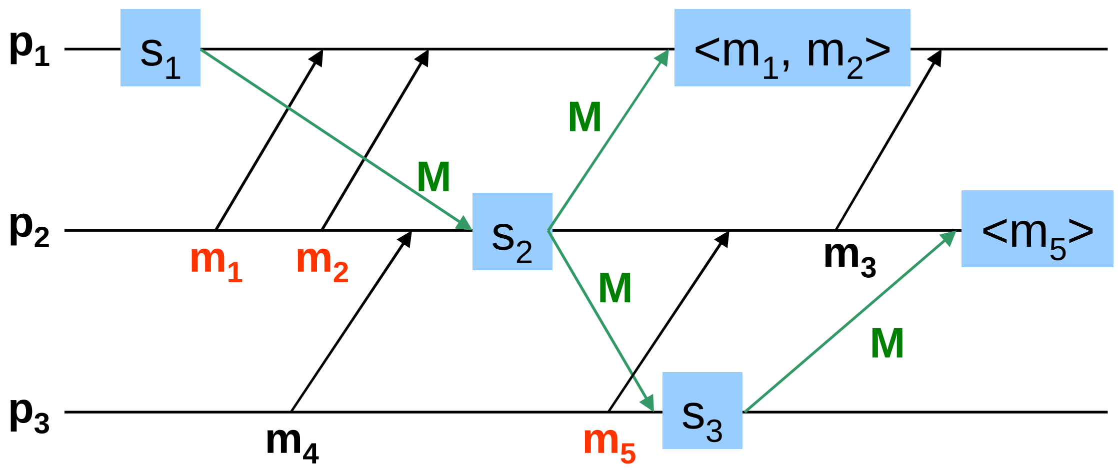


Note: the captured state is not necessarily a valid execution state, but makes a transition to a valid execution state

Distributed Snapshot

It also works for more than two processes:





Correctness of Distributed Snapshot

Assume:

- e_i and e_j occurring at p_i and p_j respectively
- $e_i \rightarrow e_j$
- C is a recorded cut

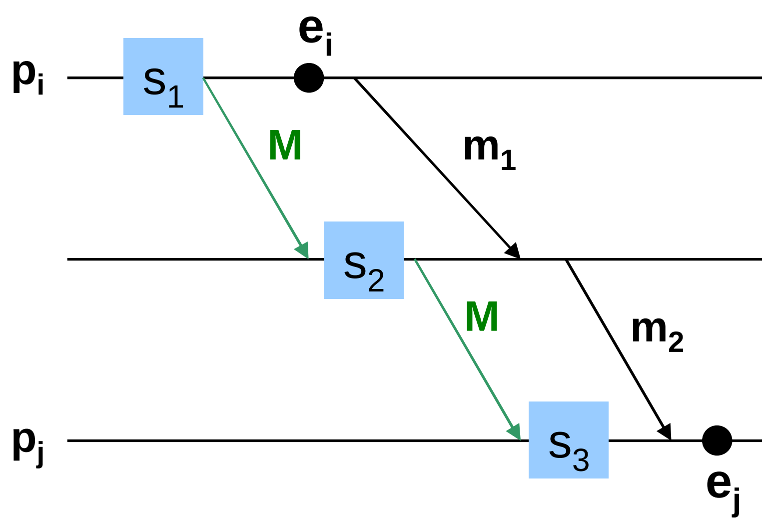
To be shown: $e_j \in C \Rightarrow e_i \in C$

- Case 1: $p_i = p_j$ – obvious
- Case 2: $p_i \neq p_j$ – assume $e_i \notin C$ and $e_j \in C$

  Assume a sequence of messages $m_1, m_2, \ldots, m_H$ giving rise to $e_i \rightarrow e_j$

  By FIFO characteristics of channels and sending and receive rule: marker M would have reached p_j ahead of each $m_1, m_2, \ldots, m_H$

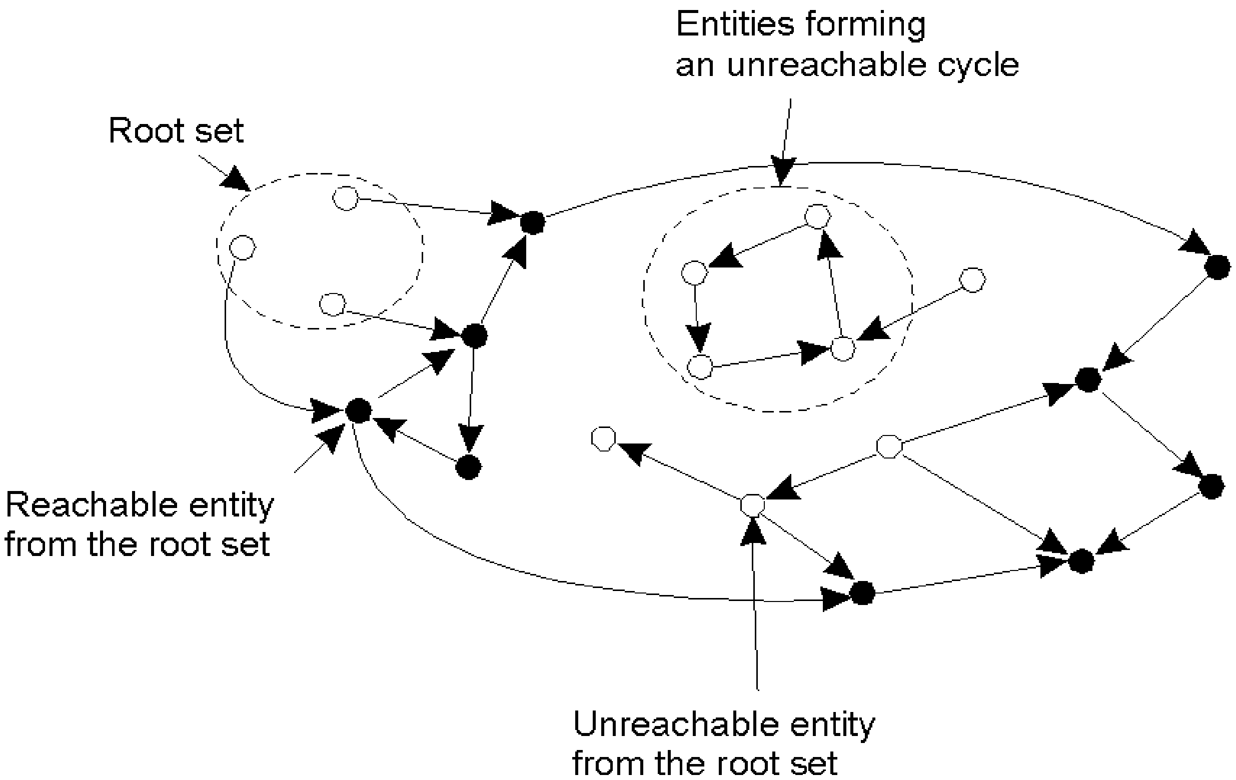
  By maker receiving rule: p_j would have recorded its state before event e_j – contradiction!



The Problem of Unreferenced Objects

Application of snapshot algorithm: *distributed garbage collection*

- Recycling of previously used but now unused memory
- Detection of unreferenced objects without interference with the applications
- Reference graph: there is some root from which any object is reachable

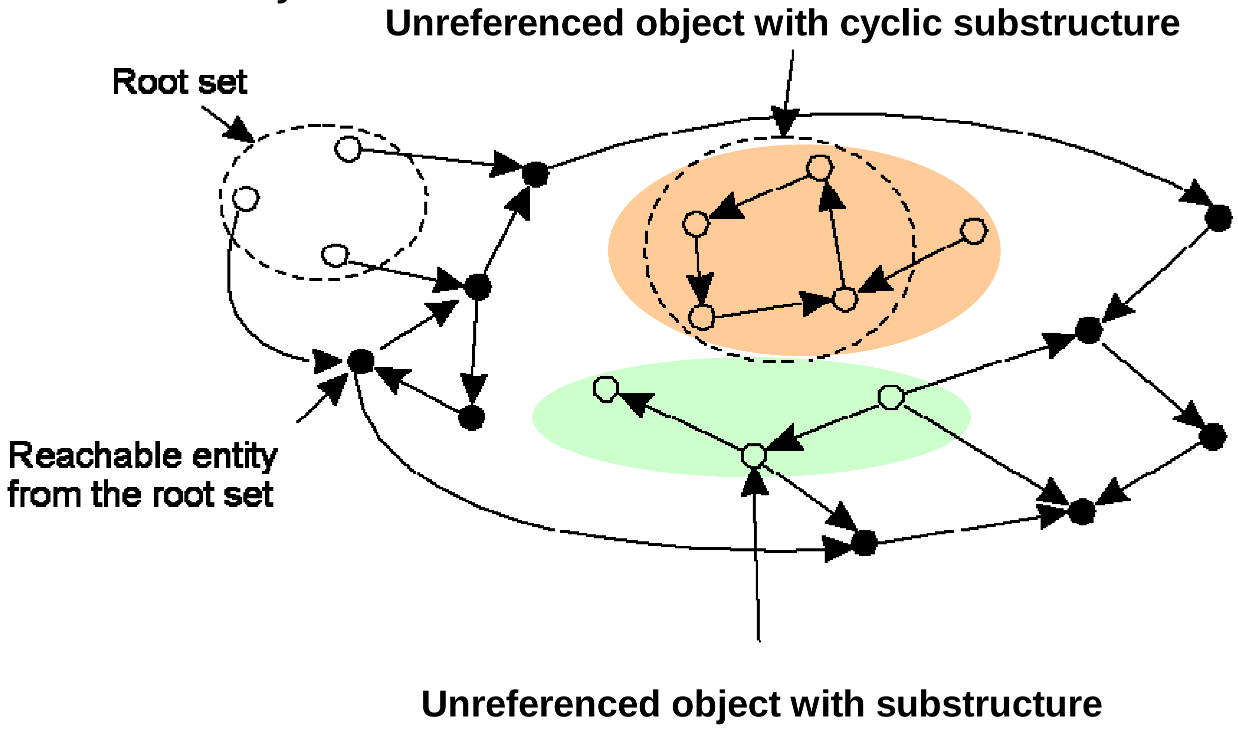


Garbage Collection

An object not reachable from the root is garbage

- Its outgoing references should be deleted
- Thus, substructures without other references can be deleted (how to deal with cycles of objects?)

→ Traversing of all objects necessary



Distributed Garbage Collection

universitätbonn

In distributed systems: large set of objects, remote references

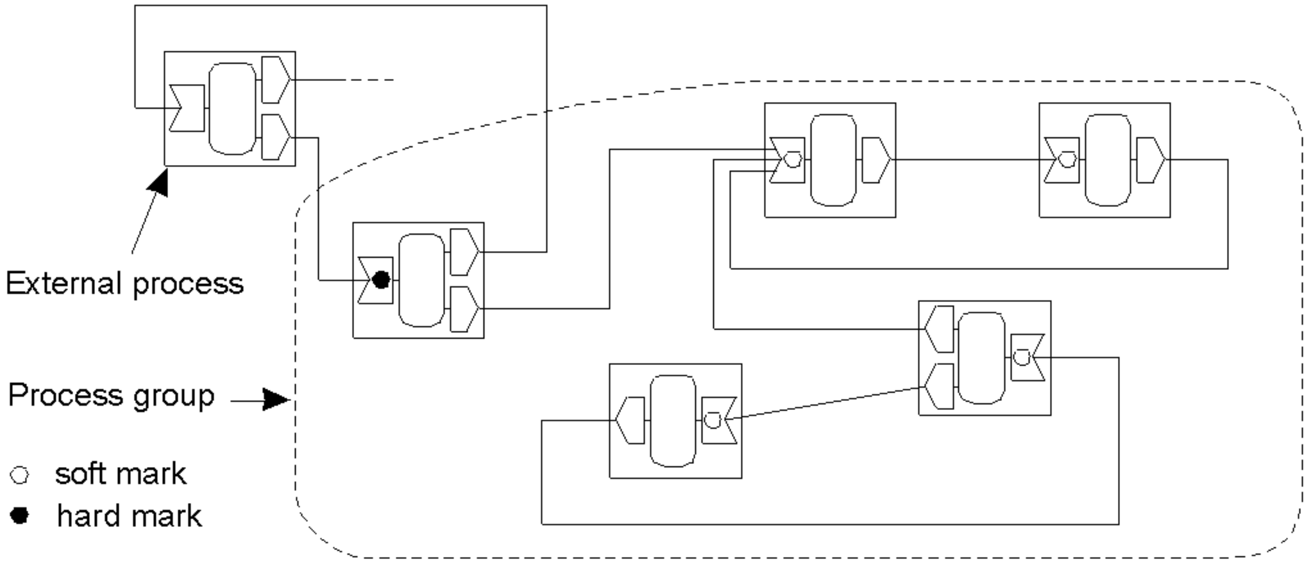
→ Typically decentralized and hierarchical

- *Local* garbage collection

  Assumption: all incoming remote references come from a reachable object

- *Global* garbage collection

  Searching of garbage chains across boundaries by considering local regions as one object with incoming and outgoing references



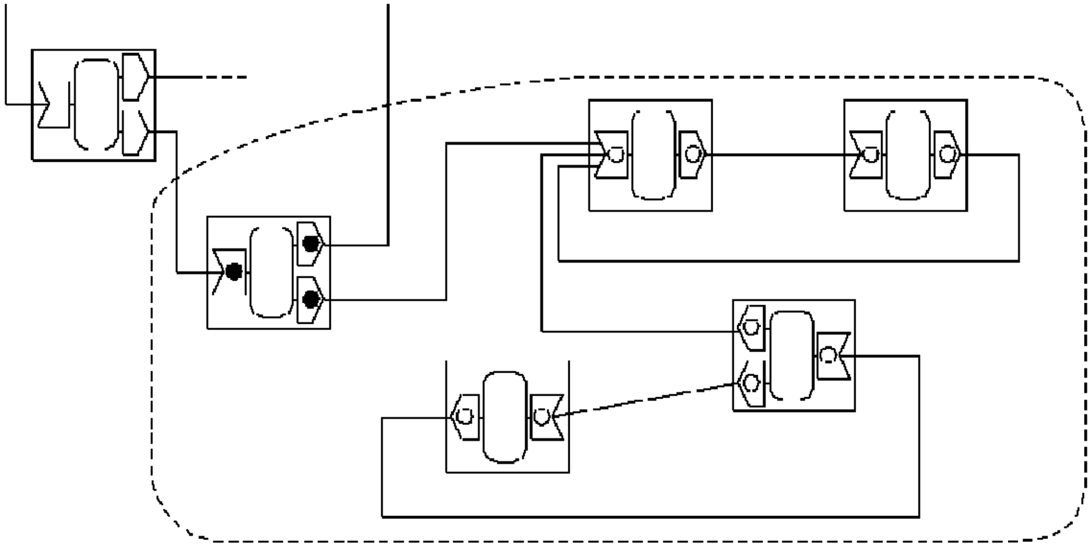
Tracing in Groups

As is snapshot algorithm: use markers with the following initialization:

- *Hard mark*: skeleton reachable from a root object, a hard marked proxy, or an external object
- *Soft mark*: skeleton only reachable from inside the group

Algorithm:

- Propagate hard marks of skeletons to the proxies of an object
- Propagate hard marks of proxies to the referred skeletons
- Repeat these steps till no more change is made



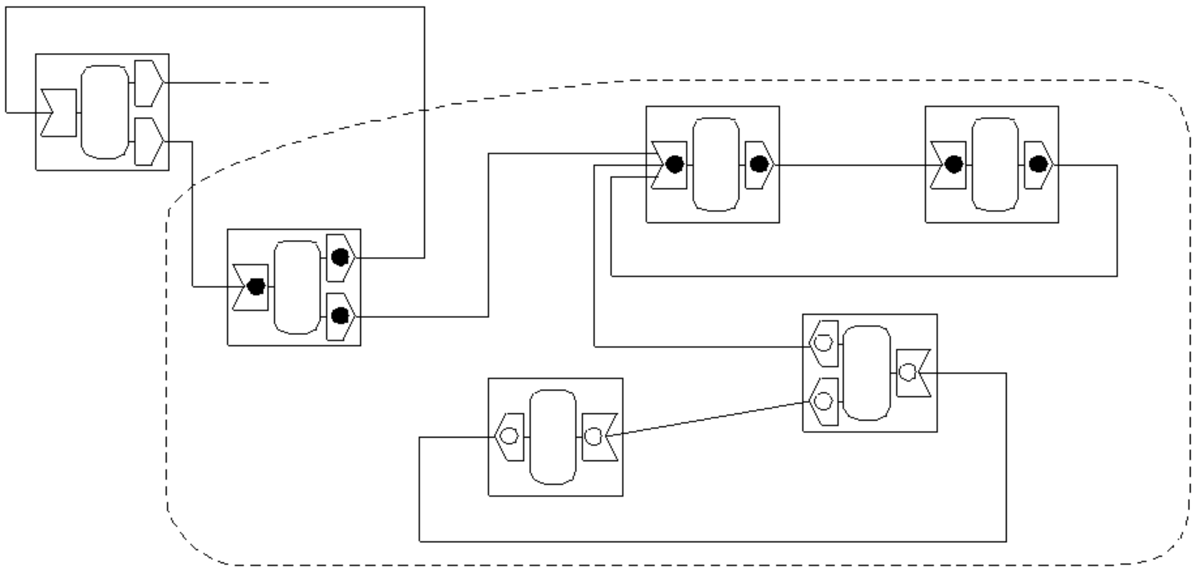
Tracing in Groups

If there are no more changes: deletion of soft-marked objects

- Reduction of objects in the groups
- Relations between groups unconsidered

Afterwards:

- Apply same schema on higher hierarchy level, i.e. consider each group as one object
- Repeat up to the highest hierarchy level and test on reachability from the root set



Distributed Debugging

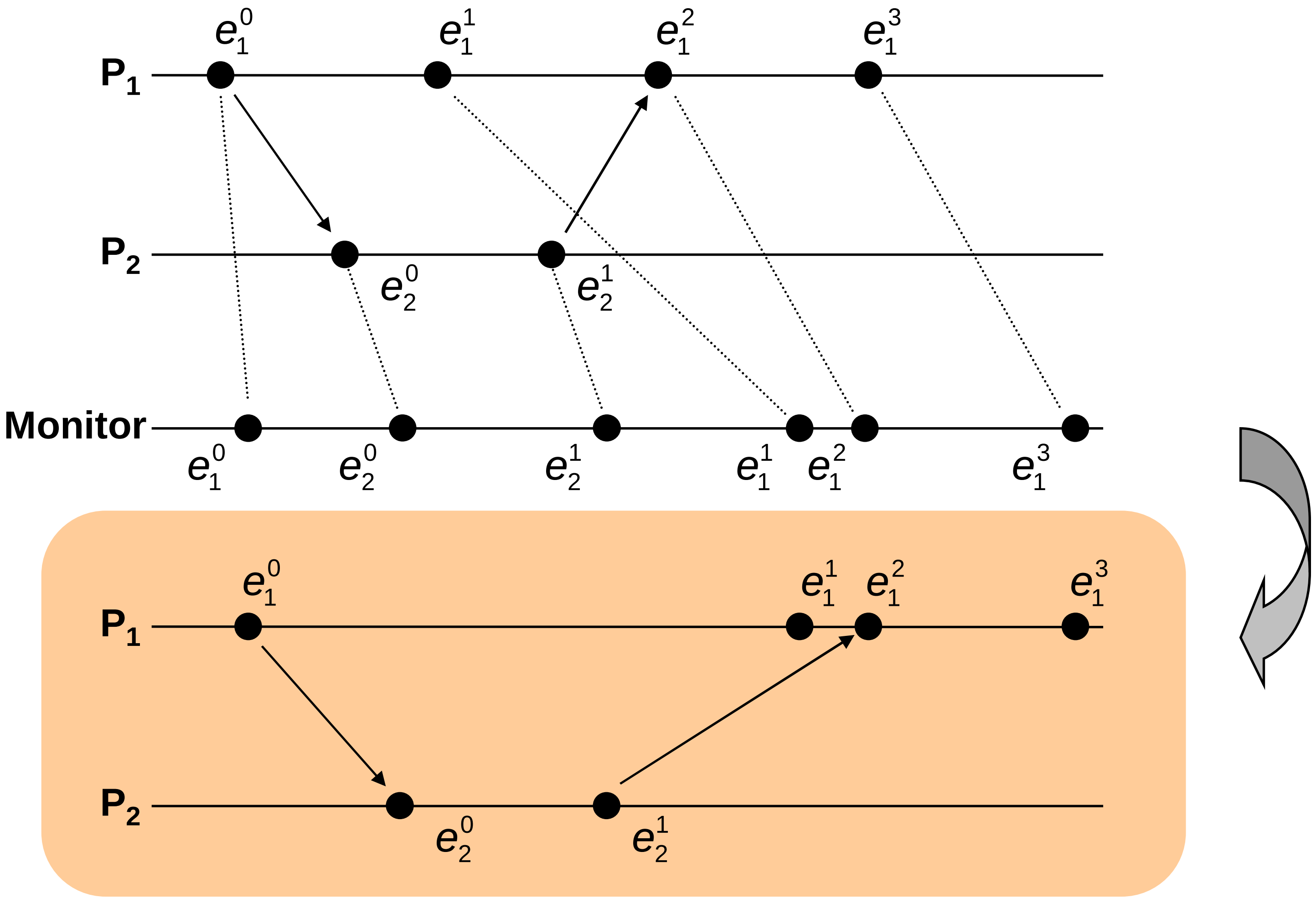
Other use of cuts: *make statements about whether a (transitory) state occurred in a distributed system's execution*

- Does safety condition $|x_i - x_j| < \delta$ for $i,j = 1, \ldots N$ hold for the whole execution although a process my change ist variable any time?
- Is in all points in time at least one valve of my factory‘s pipes open although a process can open/close ist valve at any time?

Approach:

- Make use of global predicates
- Monitor system‘s execution and capture trace information
- Evaluate whether the safety condition was or may have been violated

Monitoring System's Execution



Global State Predicates

More formal: determine whether a global state predicate Φ

- was *definitely* `true` at some point in the observed execution
- was *possibly* `true` in the observed execution

Let H be the history of an observed execution

- $possibly_{\Phi}$: there is a consistent global state S through which a linearization of H passes and $\Phi(S)$ = `true`
- $definitely_{\Phi}$: for all linearizations L of H there is a consistent global state S through which L passes such that $\Phi(S)$ = `true`

Example: anti-collision air control

- Decision variables d_1, $d_2 \in \{\perp, \mathit{fall}, \mathit{rise}\}$ of processes p_1 and p_2
- Safety: $\neg\, possibly_{\Phi}$, with $\Phi = (d_1 = d_2 \wedge d_1 \neq \perp)$
- Liveness: $definitely_{\Phi}$, with $\Phi = (d_i \neq \perp, i = 1,2)$, Φ is stable

Global State Predicates

Using the snapshot algorithm (Φ is non-stable)

- $\Phi(S_{Snap})$ = `true` $\Rightarrow$ $possibly_{\Phi}$
  $\not\Rightarrow$ $definitely_{\Phi}$
- $\Phi(S_{Snap})$ = `false` $\not\Rightarrow$ $possibly_{\Phi}$ or $definitely_{\Phi}$

In general, evaluation of $possibly_{\Phi}$ and $definitely_{\Phi}$ entails search through linearizations derived from an observed execution

- If there is no linearization where Φ evaluates to `true`, $\neg\Phi$ holds in each state of each linearization:

  $\neg\, possibly_{\Phi} \;\Rightarrow\; definitely_{\neg\Phi}$

- Even if Φ evaluates to `false` in some state of each linearization, it may become `true` in some state of one or more linearizations:

  $definitely_{\neg\Phi} \;\not\Rightarrow\; \neg\, possibly_{\Phi}$

Collecting the State

Centralized approach:

- Observed processes p_i (i = 1, 2, …, N) send state information to monitor M
- M records states received from p_i in queue Q_i

Process p_i:

- Send initial state to M
- On state change send new state in a state-message to M

Monitor M:

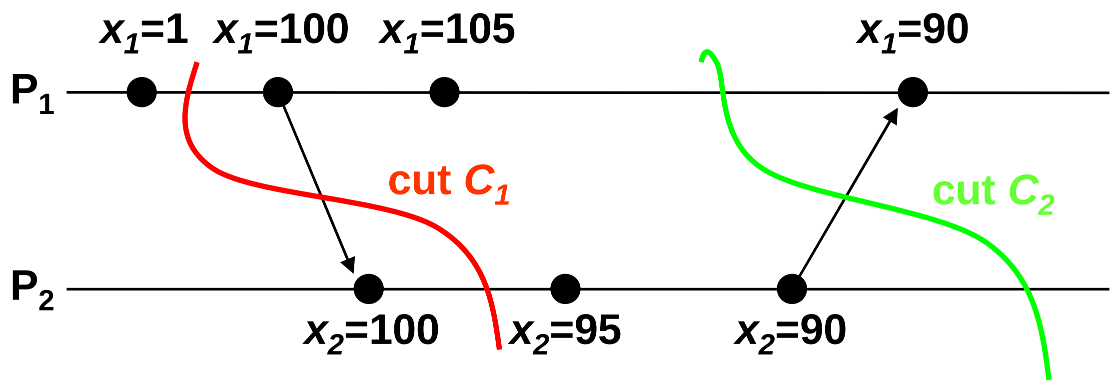
- On receipt of state-message form p_i record received state in queue Q_i

Optimizations to reduce state-message traffic:

- Only send states that may affect the predicate value
- E.g.: x_1 and x_2 are variables of p_1 and p_2 and Φ: $x_1 > x_2$
  - ➢ p_1 only sends state message if x_1 is decreased
  - ➢ p_2 only sends state message if x_2 is increased

Observing Consistent Global States

Example:



Initially, x_1=x_2=0, Φ: $|x_1$-$x_2|$<50

- p_1 and p_2 send state messages when x_1 and x_2 are changed

Monitor evaluates per-process queue

- Cut C_1: x_1=1, x_2=100 (inconsistent)
- Cut C_2: x_1=105, x_2=90 (consistent)

Observing Consistent Global States

Problem: distinguish consistent global states from inconsistent ones

- State information associated with vector timestamps
- Each state message includes value of sender's vector time *vt*
- Each queue Q_i is kept ordered in sending order due to the vector timestamps

Let

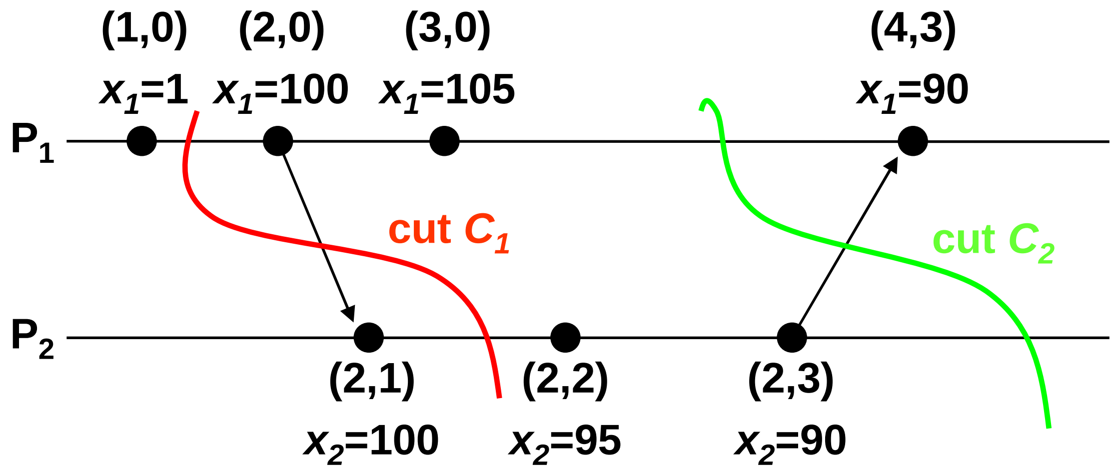
- $S = (s_1, s_2, \ldots, s_N)$ be a global state constructed from received state-messages
- $VT(s_i)$ be the vector timestamp of s_i received form process p_i:

S is a consistent global state if and only if

$VT(s_i)[i] \geq VT(s_j)[i]$ for $i,j = 1, 2, \ldots, N$

Observing Consistent Global States

Example:



Q_1:	**(x_1=1);(1,0)**	Q_2:	**(x_2=100);(2,1)**
	(x_1=100);(2,0)		**(x_2=95);(2,2)**
	(x_1=105);(3,0)		**(x_1=90);(2,3)**
	(x_1=95);(4,3)		

C_1: **(x_1=1);(1,0) / (x_2=100);(2,1)** **not consistent!**

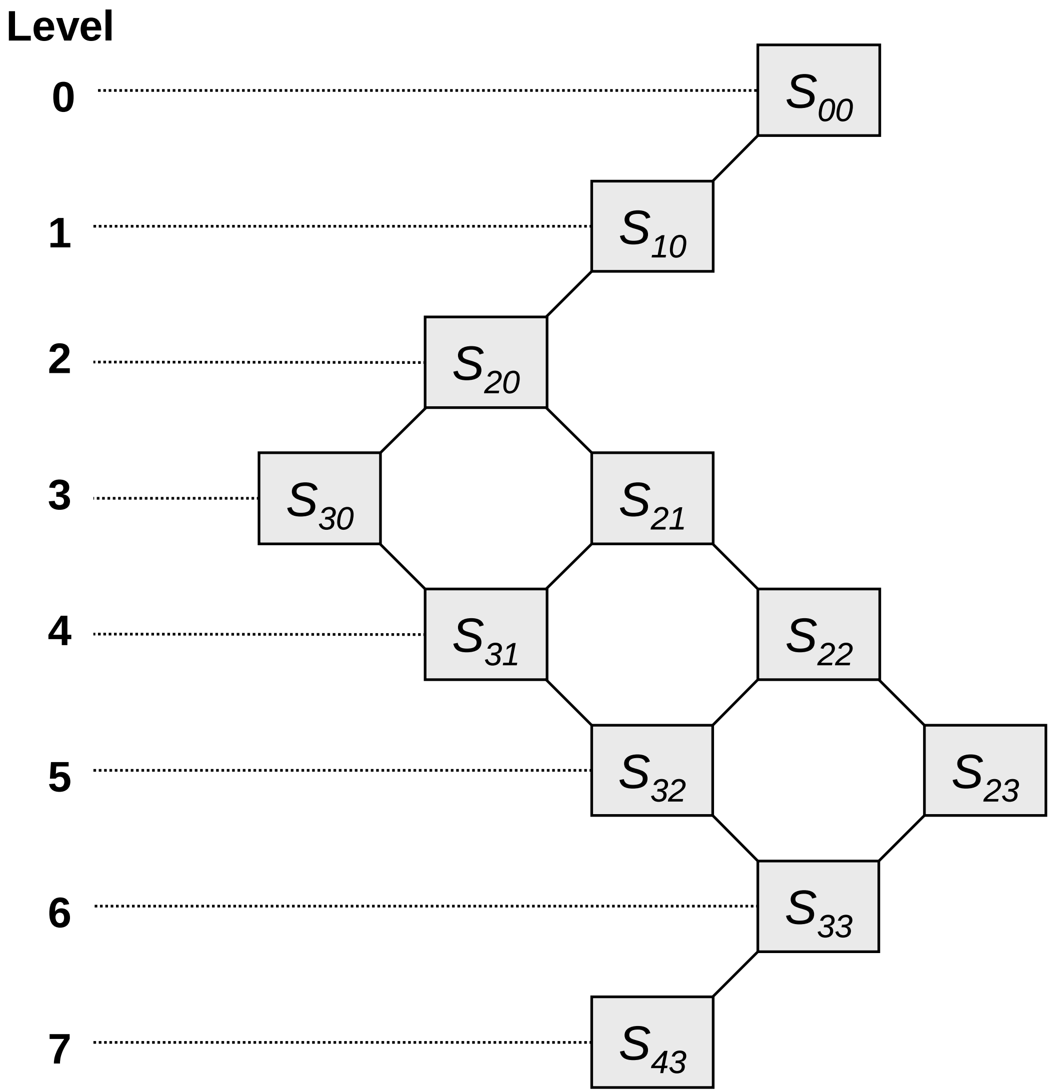
C_2: **(x_1=105);(3,0) / (x_2=90);(2,3)** **consistent**

Observing Consistent Global States

Using vector timestamps and the condition from slide 55, *M* can construct the lattice of consistent global states for a given history – recorded in Q_i

Now evaluate

- *possibly*$_\Phi$: *M* must find a linearization with a state at which Φ evaluates to `true`
- *definitly*$_\Phi$: *M* must find a set of states through which all linearizations must pass, and at each of which Φ evaluates to `true`



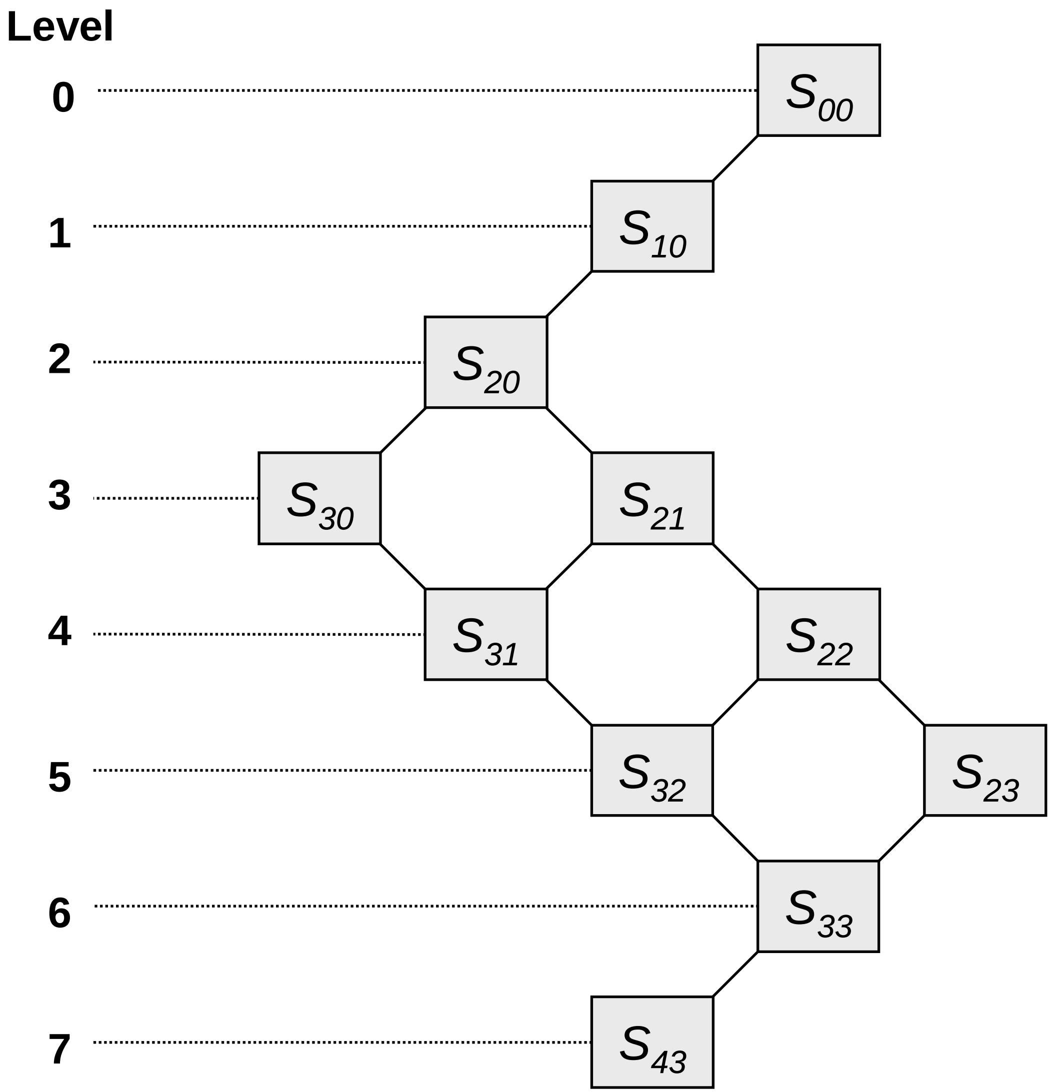
Evaluating *possibly*$_\Phi$

There exists a linearization starting from the initial state that passes through a state for which Φ evaluates to `true`

$\Rightarrow$ *possibly*$_\Phi$

Approach: the monitor

- traverses lattice of reachable states, starting from initial state $(s^0_1, s^0_2, \ldots, s^0_N)$
- stops when it finds a state that evaluates to `true`



Evaluating *possibly*$_\Phi$

Algorithm: evaluating *possibly*$_\Phi$ for global history *H* of *N* processes
(Assumption: execution is infinite)

```
L := 0;
States := {(s⁰₁, s⁰₂, …, s⁰_N)};
while (Φ(S)=false for all S ∈ States)
  L := L + 1;
  Reachable := {S': S' reachable in H from some S ∈ States
                ∧ level(S') = L};
  States := Reachable;
end while
output „possibly_Φ";
```

universitätbonn

Evaluating *possibly* $_\Phi$

How to discover reachable states?

- Let $S = (s_1, s_2, \ldots, s_N)$ be a consisntent state in level L
- Consistent state on level L+1 reachable from S:
  - is of form $S' = (s_1, s_2, \ldots, s'_i, \ldots s_N)$
  - i.e. differs from S only in one position, representing the next state of some process p_i
- Monitor can find all such states by traversing ist queues Q_i

More formal: S' is reachable from S if and only if

for $j = 1, 2, \ldots, N$ and $j \neq i$: $VT(s_j)[j] \geq VT(s'_i)[j]$ and $VT(s'_i)[i] \geq VT(s_i)[i]$

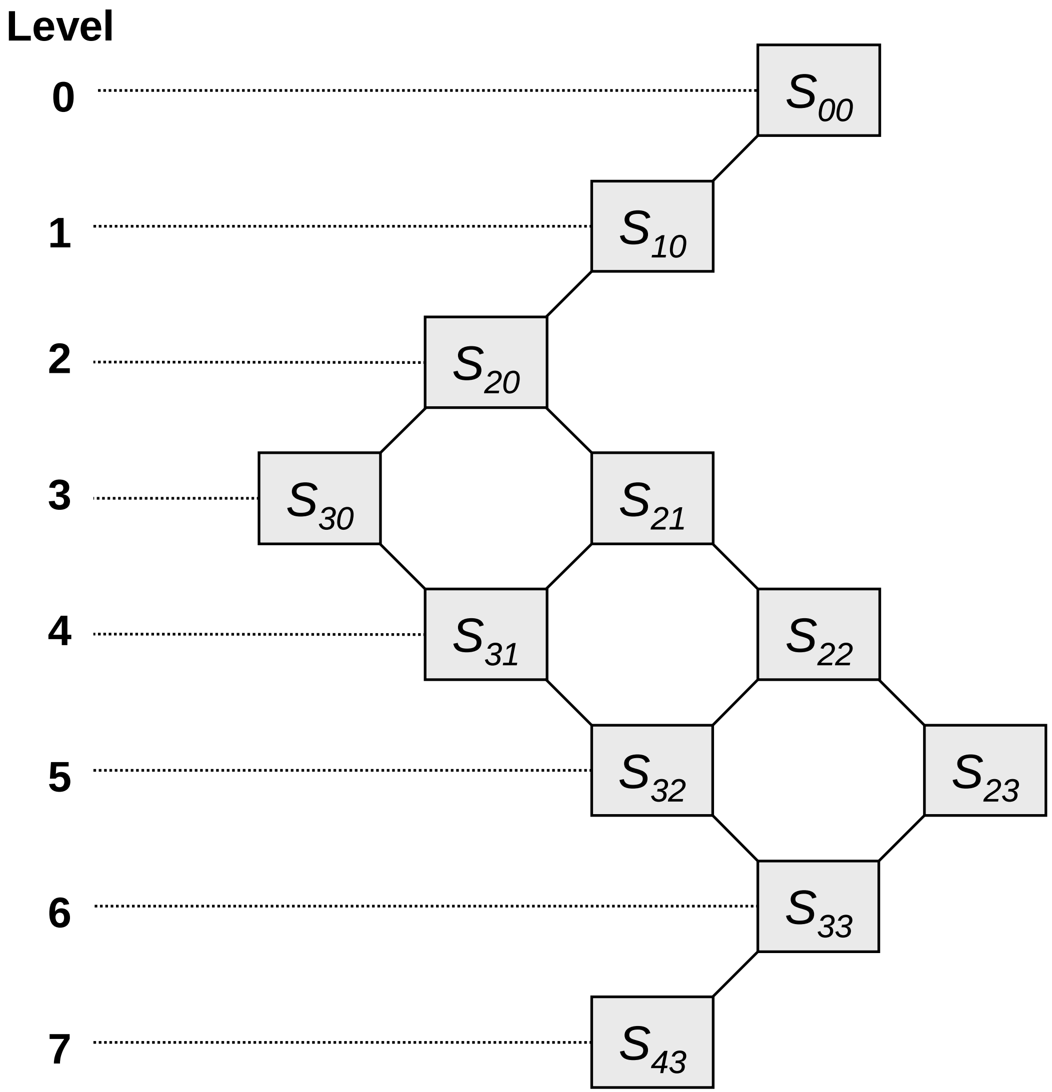
Evaluating *definitely*$_\Phi$

There exists no linearization from the initial to the final state that passes only states for which Φ evaluates to `false`

$\Rightarrow$ *definitely*$_\Phi$

Approach: the monitor

- traverses lattice of reachable states, starting from initial state $S^0 = (s^0{}_1, s^0{}_2, \ldots, s^0{}_N)$
- for the current level records in *States* all states that may be reached on a linearization from S^0 traversing only states for which Φ evaluates to `false`
- reaches a level at which *States* is empty



Evaluating *definitely*$_\Phi$

Algorithm: evaluating *definitely*$_\Phi$ for global history *H* of *N* processes
(Assumption: execution is infinite)

```
L := 0;
if Φ(s⁰₁, s⁰₂, …, s⁰N)
  States := {};
else
  States := {(s⁰₁, s⁰₂, …, s⁰N)};
while (States ≠ ∅)
  L := L + 1;
  Reachable := {S‘: S‘ reachable in H from some S ∈ States
                 ∧ level(S‘) = L};
  States := {S ∈ Reachable: Φ(S) = false};
end while
output „definitelyΦ“;
```

Costs for *k* = maximum number of events in a process

- Time complexity: $O(k^N)$
- Space complexity: $O(kN)$

Conclusion on Time and Global States

Time is an important factor in a distributed system
→ How to synchronize distributed components?

NTP is standard for absolute time synchronization
- Never accurate enough for achieving synchronous systems
- Mainly used to keep computers in the Internet more or less up to date

More commonly used in distributed applications: logical time synchronization
- **Lamport timestamps** for implementing „happen-before“ relationship
- **Vector timestamps** for considering causality

Determination of global states
- Missing global time, thus concept of **consistent cut**
- Algorithm like **Snapshot algorithm** for capturing consistent global states
- **Global predicates** for evaluating system safety and liveness