Technische Universität Darmstadt

Telecooperation Lab
Prof. Dr. Max Mühlhäuser

TK1: Distributed Systems - Programming & Algorithms

Chapter 3: Distributed Algorithms

Section 2: Synchronization – Time and State

Lecturer: **Prof. Dr. Max Mühlhäuser**

Types of Distributed Systems

Two types of distributed systems relevant for distributed algorithms:

  - Synchronous
  - Asynchronous

- **Synchronous** distributed systems are systems with known bounds on:
  1. **Message transmission delay**
  2. **Execution time (for each step of system)**
  3. **Clock drift rate**
  - Typically specify maximum and minimum bound (can be 0)
  - Measured or conservatively estimated
- **Asynchronous** system is a system where at least one of the above three conditions does not hold

Types of Systems: Remarks

Are **assumptions** about synchronous distributed systems **reasonable**?

Yes and maybe no

- Yes: synchronous distributed systems (as defined) *can be (and are!) built*
  - Some video-on-demand services, other special purpose / realtime systems
- Maybe no:
  - Uncertainty wrt. bounds (max - min) may be so large that system becomes *impractical*
  - *and:* the Internet as a whole **is asynchronous**
    - And still: there are many useful applications in the Internet ☺
    - why: synchronous solution may be "provably correct", lack of properties may be mitigated

- Conclusion: One can live with asynchronous systems
- But: many distributed algorithms first built for synchronous case, then extended
  - i.e. we will hear about these again later

- For now, we will focus on condition 3: "clock drift rate"
  - Leads to issues of physical clocks → HW clock synchronization, below

Hardware Clocks

- Physical clocks in computers are realized as crystal oscillation counters @ hardware level
  - Correspond to counter register H(t)
  - Used to generate interrupts
- Usually scaled → approximate physical time t yielding software clock C(t): $C(t) = \alpha H(t) + \beta$
  - C(t) measures time relative to some meaningful reference event
    - E.g., 64 bit counter for # of nanoseconds since last boot
    - Add “boot time” (stored somewhere) → get approximate actual time T (via OS call)
  - Programmer/user: tempted to assume C(t) = t
    - Never 100% achieved; plus: *resolution usually much lower that that of H(t)*
    - Note: values given by two consecutive clock queries will differ only if clock resolution is sufficiently smaller than processor cycle time
    - Interesting effects if one tries to measure, e.g., “execution time” of code block

Problems with Hardware Clocks

- **Skew:** Disagreement in the reading of two clocks
- **Drift:** Difference in the rate at which two clocks count the time
  - Due to physical differences in crystals, plus heat, humidity, voltage, etc.
  - Accumulated drift can lead to significant skew
- **Clock drift rate:** difference in precision between a prefect reference clock and a physical clock,
  - Usually, 10^{-6} sec/sec; 10^{-7} to 10^{-8} for high precision clocks

Measuring Time

- Traditionally, time measured astronomically
  - Dasis: transit of the sun (highest point in the sky)
  - Yields: solar day and solar second (a 86400th of mean solar day)
- Problem: Earth's rotation is slowing down
  - Days get longer and longer
  - 300 million years ago there were 400 days in the year ;-)
- Modern way to measure time: atomic clock
  - Based on transitions in Cesium-133 atom
  - Still need to correct for Earth's rotation: leap seconds
- Result: Universal Coordinated Time (UTC)
  - UTC available via radio signal, telephone line, satellite (GPS)
  - Typical sending frequency: once per second (rarely, for a 1GHz computer!)

Synchronizing Clocks

- **External** synchronization
  - synchronize process clock with an authoritative external reference clock S(t) by limiting its skew to a delay bound D > 0
    - $|S(t) - C_i(t)| < D$ for all t
  - For example, synchronization with a UTC source
- **Internal** synchronization
  - Synchronize the local clocks within a distributed system to disagree by not more than a delay bound D > 0, without necessarily achieving external synchronization
    - $|C_i(t) - C_j(t)| < D$ for all i, j, t
- Obviously:
  - For a system with external synchronization bound of D, the internal synchronization is bounded by 2D

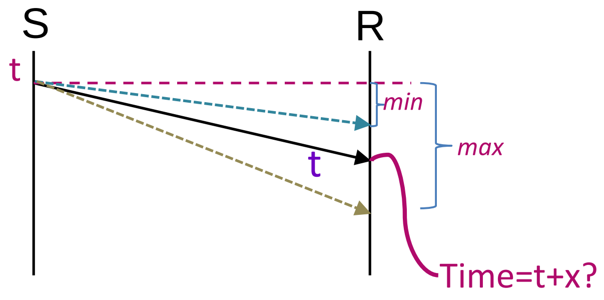
Clock Correctness

When is a clock correct?

1. If drift rate falls within a bound r > 0, then for any t and t' with t'>t the following error bound in measuring t and t' holds:
   - $(1-r)(t'-t) \leq H(t') - H(t) \leq (1+r)(t'-t)$
   - But consequence: no jumps in HW clocks allowed → how to synch. clocks?
2. Sometimes monotonically increasing clock is enough:
   - $t'>t \Rightarrow C(t') > C(t)$ --- but, this is insufficient for 'time keeping matters'
3. Hence: **frequently used condition** …
   - Monotonically increasing +
   - Drift rate bounded between synchronization points +
   - Clock may jump ahead (!) at synchronization points

Synchronization in Synchronous Systems

- Recall synchronous system:
  - Known bounds for: i) clock drift rate, ii) max. message transmission delay *max*, and iii) time to execute each step
- Synchronization procedure:



1. Sender piggybacks own time t on message m
2. Receiver sets own clock to: $t + T_{trans}$

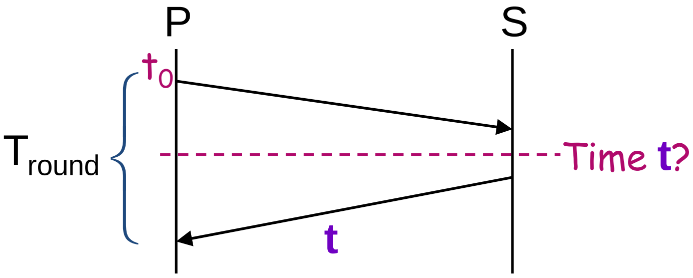
Problem: How to estimate T_{trans} ?

- Usually, it is possible to estimate minimum transmission delay *min* (or: set to 0)
  - Make conservative assumptions, also assume: no cross traffic interferes
- Let *u = max - min*
  - If receiver sets own clock to t + ½(*max*+*min*), then the skew is bounded by u/2
  - When synchronizing **N clocks**: it is possible to show that the optimum achievable bound is $u * (1- {}^{1}/_{N})$
- Is this usable for, e.g., the Internet?

Synchronization in Asynchronous Systems

Christian’s Algorithm (F. Christian 1989)

- Observations:
  - Round trip times between processes are often reasonably short in practice, yet theoretically unbounded
  - Practical estimate possible if round-trip times are sufficiently short in comparison to required accuracy



- Principle:
  - (assume, e.g., UTC-synchronized time server S)
  - Process P sends requests to S
  - Measures round-trip time T_{round}
    - In LAN, T_{round} should be around 1 ms
    - During 1ms, a clock with a 10^{-6} sec/sec drift rate varies by at most 10^{-9} sec
    - Hence the estimate of T_{round} is reasonably accurate
  - Naive estimate: Set clock to t + ½T_{round}

Christian's Algorithm: Analysis

Accuracy of estimate?

- Assumptions:
  - requests and replies via same net (same delay characteristics)
  - *min* is either known or can be estimated conservatively
- Calculation:
  - Earliest time that S can have sent reply: $t_0 + min$
  - Latest time that S can have sent reply: $t_0 + T_{round} – min$
  - Total time range for answer: $T_{round} - 2 * min$
  - Accuracy is $\pm$ ($½T_{round} - min$)
- Discussion:
  - Really only suitable for deterministic LAN environment or Intranet
  - Problem of failure of S
    - Redundancy through group of servers, multicast requests
    - How to decide if replies vary? (byzantine agreement problems)
  - Imposter providing false clock readings? (authentication)

Alternative Algorithm

Berkeley algorithm (Gusella&Zatti ‘89)

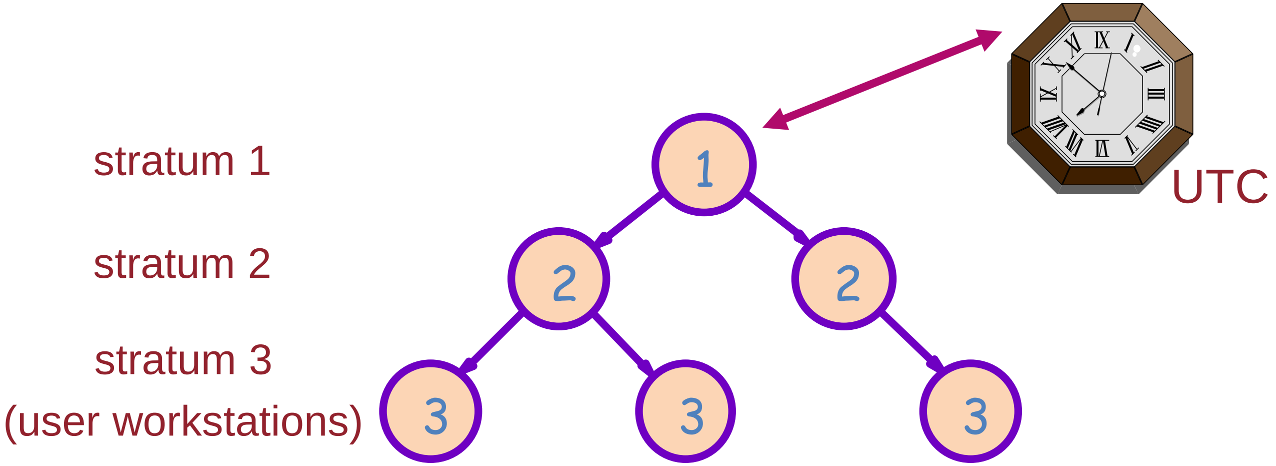
- Principle:
  - No external synchronization, but one master server (time daemon)
  - Master polls slaves periodically about **their** clock readings
  - Estimate of local clock times using Christian-like round trip estimation
  - Averages the values obtained from a group of processes
    - Cancels out individual clock’s tendencies to run fast
  - Tells slave processes by which amount of time to adjust local clock
  - Master failure → find new master via election algorithm (see later)
- Experiment published:
  - 15 computers, local drift rate < $2x10^{-5}$, max round-trip 10 ms
  - Clocks were synchronized to within 20-25 ms

- Note: neither algorithm is really suitable for Internet where RTTs vary a lot

Clock Synchronization: NTP

Internet Network Time Protocol (NTP)

- Goals:
  - Ability to externally synchronize clients via Internet to UTC
  - Provide reliable service tolerating lengthy losses of connectivity
  - Enable clients to resynchronize sufficiently frequently to offset typical HW drift rates
  - Provide protection against interference
- Synchronization subnets



Note: Arrows denote synchronization control, numbers denote strata.

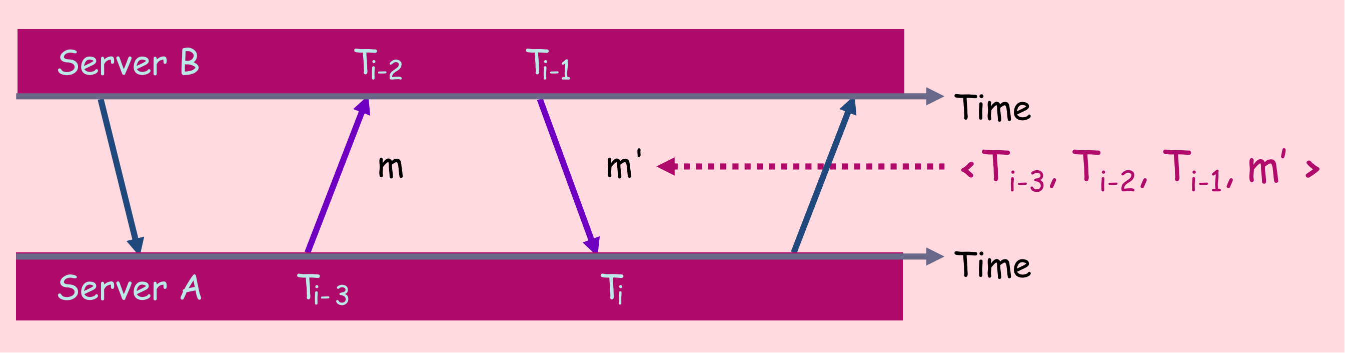
Clock Synchronization: NTP (cont.)

- Layered client-server architecture, based on UDP message passing
- Synchronization at clients with higher strata number less accurate due to increased latency to strata 1 time server
- Failure robustness: if a strata 1 server fails, it may become a strata 2 server that is being synchronized though another strata 1 server
- Modes
  - Multicast:
    - One computer periodically multicasts time info to all other computers on net
    - These adjust clock assuming a very small transmission delay
    - Only suitable for high speed LANs; yields low but usually acceptable sync.
  - procedure-call: similar to Christian's protocol
    - Server accepts requests from clients
    - Applicable where higher accuracy is needed,
      or where multicast is not supported by the network's hard- and software
  - Symmetric:
    - Each node is client and server (higher accuracy, more overhead, no hierarchy)

Clock Synchronization: NTP (cont.)

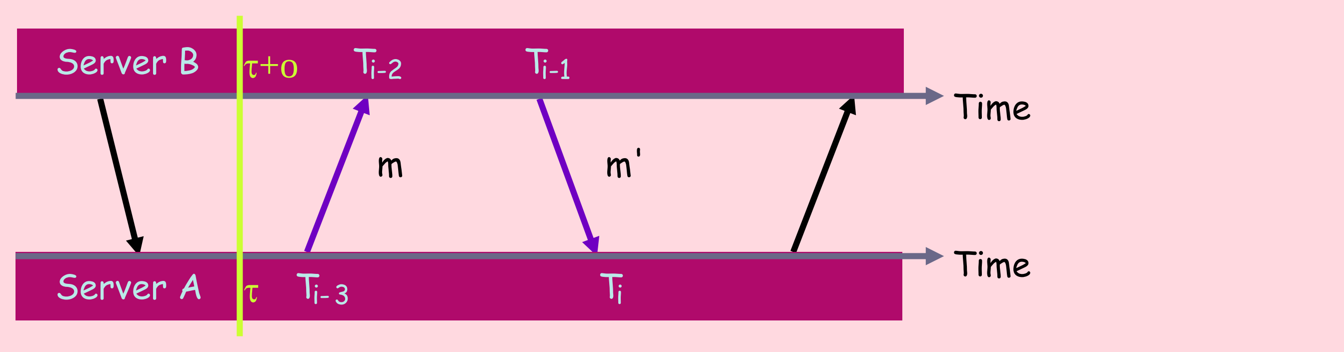
Here: protocol mechanism for procedure-call mode (symmetric mode: similar)

- All messages carry timing history information
  - Local timestamps of send and receive of *previous* NTP message
  - Local timestamp of send of *current* message



- For each pair i of messages (m, m’) exchanged between two servers the following values are being computed
  (based on 3 values carried w/ msg and 4th value obtained via local timestamp):
  - Offset o_i: estimate for the actual offset between two clocks
  - Delay d_i: true **total transmission time** for the pair of messages

Clock Synchronization: NTP (cont.)



- Let **o** the true offset of B's clock relative to A's clock (which can not be calculated), and let **t** and **t'** the true transmission times of m and m' (note: T_i, T_{i-1} ... are **not** true time!)
- Then, obviously: $T_{i-2} = T_{i-3} + t + o$ (1) and
  $T_i = T_{i-1} + t' - o$ (2)
- Adding equations (1) + (2) leads to

  delay d_i $= t + t' = T_{i-2} - T_{i-3} + T_i - T_{i-1}$ (clock errors zeroed out → **true** d)
- Offset (can't get rid of T's → estimate); e.g., subtract equations (1) – (2):

  offset **o** $= o_i + ½ (t' - t)$, where offset **estimate o_i** $= ½ (T_{i-2} - T_{i-3} + T_{i-1} - T_i)$
- Accuracy (note: ½ (t - t') is error in estimate):
  for t, t' ≥ 0 it can be shown that

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

  hence, o_i is an estimate for o, and d_i is a measure for the estimate's accuracy

Clock Synchronization: NTP (cont.)

Internet Network Time Protocol (NTP): implementation, discussion:

- Statistical algorithms based on 8 most recent <o_i, d_i> pairs:
  → determine quality of estimates
- The value of o_i that corresponds to the minimum d_i is chosen as an estimate for o
- Time server communicates with multiple peers,
  eliminates peers with unreliable data,
  favors peers with lower strata number
  (e.g., for primary synchronization partner selection).
- NTP phase lock loop model: modify local clock in accordance with observed drift rate
- (at times of slower LANs), experiments achieved synchronization accuracies of 10 msecs over Internet, and 1 msec on LAN using NTP

DistSys Middleware (Platforms) usually contains „time service“ (Corba, DCE, ...), based on similar methods

Hardware Clocks: Summary

- Remember Problems, Terms
  - Clock register: has limited resolution, access from programming language restricted
  - Skew: disagreement in the reading of two clocks
  - Drift: difference in the rate at which two clocks count the time
    - Due to phys. differences in crystals, plus heat, humidity, voltage etc.
    - Accumulated drift can lead to significant skew
  - Clock drift rate: difference in precision: “prefect reference clock” vs. physical clock
    - Usually, 10^{-6} sec/sec, 10^{-7} to 10^{-8} for high precision clocks
  - External synchronization (wrt. “external clock”, such as UTC) vs. internal sync.
  - Jumps in hardware clock? → what bounds can be guaranteed?
    - Usually, monotonicity of clock required + bounds in drift rate & jumps
- Remember Algorithms
  - In synchronous distributed systems, skew between N clocks can be bound
    - … by an “optimal algorithm” to u (1 - 1/N) where u = max – min (upper / lower transmission delay bounds)
  - iI asynchronous distributed systems (Internet)
    - Algorithms by Christian and by Gusella&Zatti, both are not satisfying → NTP ‘best result we can get’

Idea: what do we really need? Exact time or ordering of events?

Logical Clocks

- remember **why** we wanted global time: causalities, ordering of events
  - with physical clocks, skew may still be, e.g., 10^{-3} sec
- idea: use ‘provable’ causalities to order events, hope to infer other causalities
  - within a single process: ordering of events directly observable
  - between different processes, simplest ‘provable’ causality:
    sending of a message happens before receiving the same message
- is this a total or partial order?
  - actually, total order is possible (with significant overhead!)
  - often, partial order is sufficient
- underlying rules:
  - If two events happen in the same process p_i, then they occurred in the order in which p_i observed them: this introduces a **local happened before relation** $\rightarrow_i$
  - for any message passing, the send event occurs before receive event

Happened Before Relation

Lamport's "happened before" relation

is the basic building block for ordering events

- This "happened before" relation $\rightarrow$ is defined based on $\rightarrow_i$ and on the send/receive causality:

HB1: for any pair of events e and e',
if there is a process p_i such that e $\rightarrow_i$ e', then e $\rightarrow$ e'

HB2: for any pair of events e and e' and for any message m,
if e = send(m) and e' = receive(m), then e $\rightarrow$ e'

HB3: if e, e' and e'' are events and
if e $\rightarrow$ e' and e' $\rightarrow$ e'', then e $\rightarrow$ e''
(in other words, HB is identical to its transitive closure)

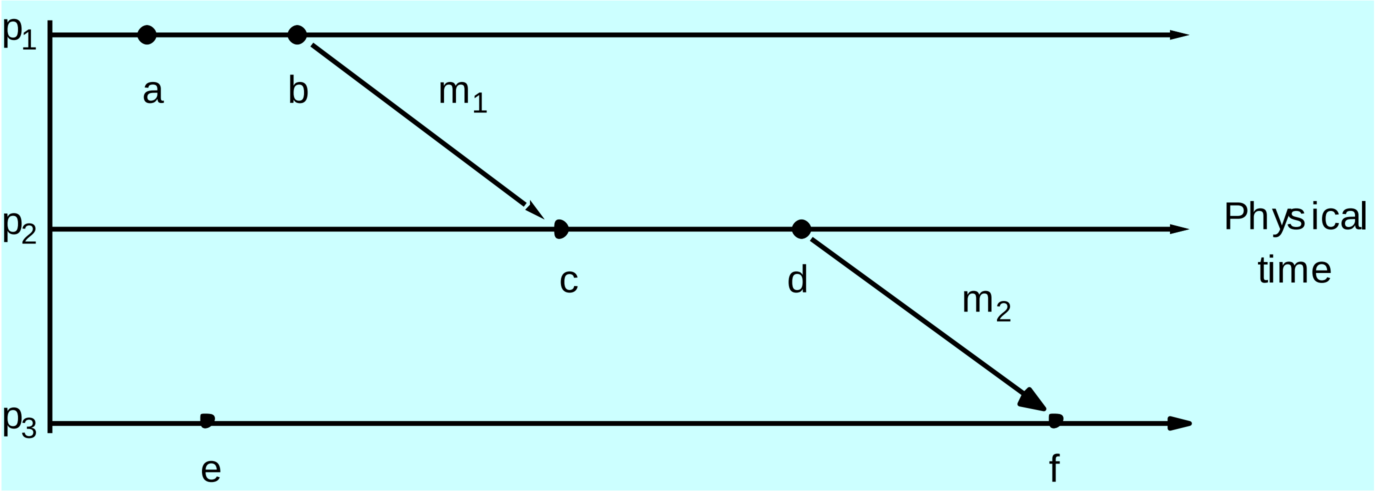
- Happened before defines a partial order

Happened Before

HB1/2/3 lead to the following:

- ‘meaning’ of HB: for any tuple e → e’ …
  - **either** e and e’ are (direct or indirect) successors *in the same process*,
  - **or** there are events e’’ and e’’’ and a message m such that: e → e’’ and e’’’ → e’ and e’’ = send(m) and e’’’ = receive(m)
    - note the recursive definition, i.e.: e → e’’ may itself involve a message
- note: e → e’ does not necessarily express causality between e and e’
  - causality means both *e happened before e’* AND *e was the reason for e’*
- concurrency:
  - for all e, e’, if *not(e → e’)* and *not(e’ → e)* holds,
    then we say that e and e’ are concurrent (also written as e || e’)

Happened Before: Example



- in the above example, following holds:
  - Because of HB1: a → b, c → d, e → f
  - Because of HB2: b → c, d → f
  - … and their transitive closure
- the following events are concurrent:
  - e is concurrent to all other events except f

Lamport's Logical Clocks

Logical clocks …

- … are the concept & distributed algorithm corresponding to the HB relation,
- … permit inference of local event order (rather trivial)
  - monotonically increasing counters impose total order on observed events
- … and some reasoning about global event orders
  - more information comes later

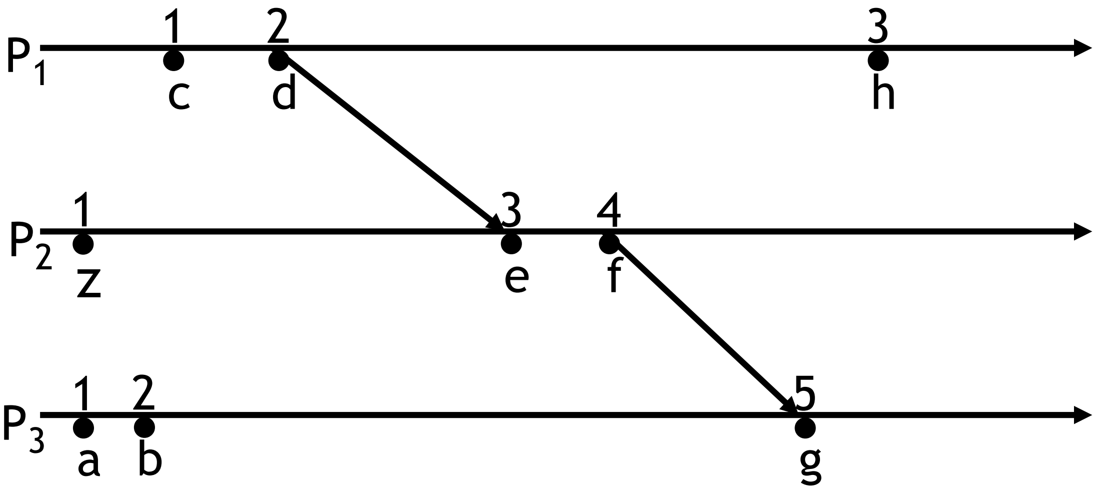
concept & distributed algorithm:

- every process p_i maintains a logical clock L_i
- $L_i(e)$ denotes the timestamp of event *e* i.e. the L_i value at its occurrence
  - Messages piggyback the timestamp of the *send* event
- algorithm ≈ rules to update logical clocks and message time stamps:

1. LC1: L_i is incremented before each event at p_i
2. LC2: A process i piggybacks L_i on every message sent, i.e. sends (m, L_i)
3. LC3: On receiving (m, t), p_j first computes $L_j = \max(L_j, t)$,
   then increments L_j, and then timestamps event *receive(m, t)*

Lamport's Clocks: Example

- at the beginning, timestamps are initialized (convention: to zero)
- clocks are updated according to rules LC1, LC2, and LC3
  - local events increase local timestamp
  - messages piggyback local timestamp
  - receiver of message calculates max of message and local timestamps, increases this max by 1, uses that for local timestamp



Problems

above algorithm has one big problem (cf. *global event orders* above)

- if e → e' then L(e) < L(e')
- **but:** L(e) < L(e') does not imply e → e'
  - for example, events b and c in figure above

- Note: It is possible to augment the partial order of **timestamps** to a total order
  - simple solution: each timestamp is concatenated with process number: $L_i(e).p_i$
  - since p_i's are unique, this gives unique timestamps
  - but: this is completely artificial
  - does not solve the above problem

Logical Vector Clocks

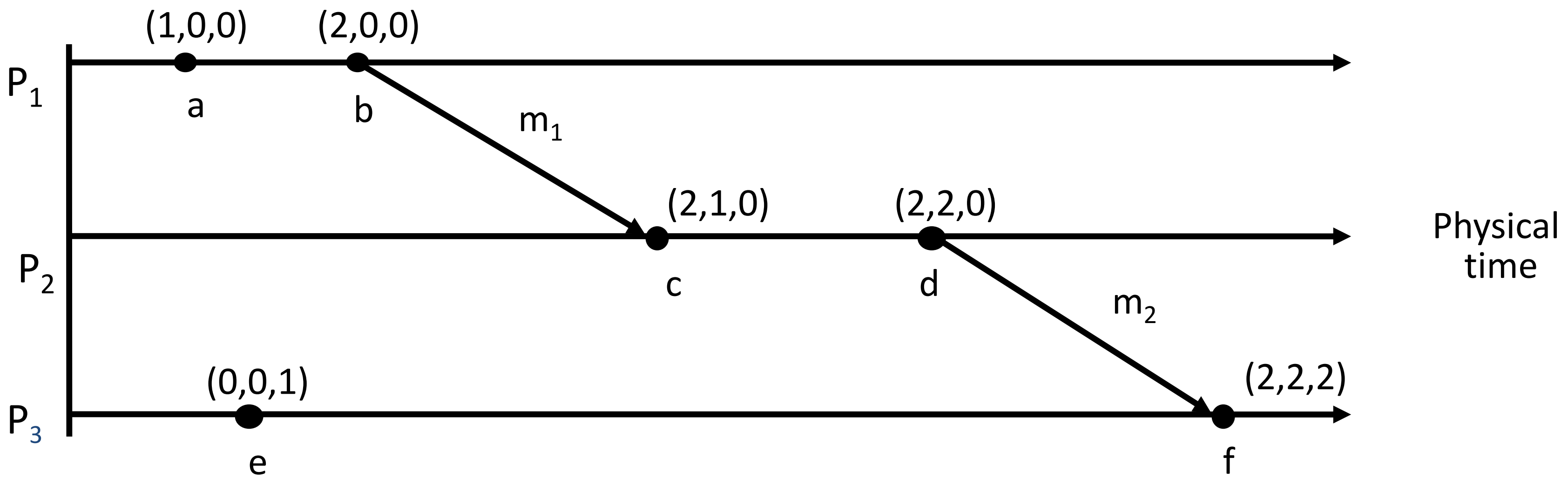
logical vector clocks solve the above problem

- vector clocks are realized as array of N (#of processes) integers
- each process i keeps its own vector clock V_i [1, ..N]
- timestamps are piggybacked as in Lamport's protocol
  - note: Timestamp is now the complete vector

again, distributed algorithm ≈ clock update rules:

1. VC1: initially, all V_i are set to [0, …, 0] (convention)
2. VC2: i sets $V_i[i] = V_i[i] + 1$ before timestamping an event
3. VC3: i includes V_i in every message (piggybacking)
4. VC4: if i receives a timestamp vector t, then i updates its V_i by means of a component-wise 'merge' similar to LC3: $V_i[j] = \max(V_i[j], t[j]), \forall j=1,..,N$

Vector Clocks: Example



- Vector clocks behave similarly to Lamport's clocks
- difference: vector clocks contain more detailed information about events in other processes
  - $V_i[j]$ is the # of events in p_j that p_i has potentially been affected by
  - Lamport's clocks only *summarize* this information over all processes p_j
  - note: on the other hand, VC has higher communication complexity

Vector Clock Comparisons

- Comparison of timestamp vectors is as follows

  $V = V'$ iff $V[j] = V'[j]\ \forall j=1,..,N$

  $V \leq V'$ iff $V[j] \leq V'[j]\ \forall j=1,..,N$

  $V < V'$ iff $V \leq V'$ and $V \neq V'$
- Note: Possible to have unordered vectors
  - These imply concurrent events
- In our example:
  - $V(b) < V(d)$
  - $V(e)$ unordered to $V(d)$, i.e., $e \,||\, d$
- Theorem: $e \rightarrow e' \Leftrightarrow V(e) < V(e')$
  - This can be proven …
  - … and means: the big problem of LC is solved

now, we will move to the second corner of the ‚Bermuda Triangle'

Global States: Intro

Remember sample problem 'total amount of money in circulation' from intro chapter

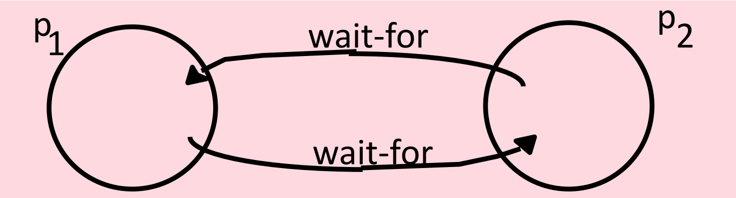
- Another sample problem that would require the view on a global state:

  Distributed deadlock detection:

  is there a cyclic 'wait-for' graph

  amongst processes and resources

  in the system?



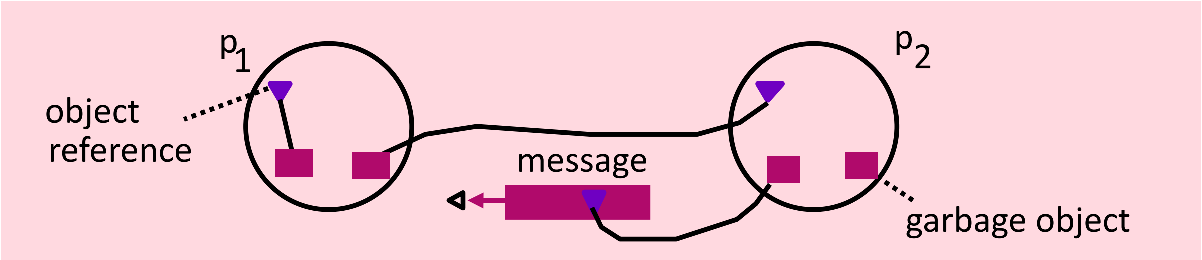
- many other examples exist, both of general (see next slide) and of domain specific nature (e.g., in distributed simulation, …)
- remember (from intro) why a fully consistent, up-to-date global state is infeasible:
  - system state changes while we conduct observation
  - observation msgs are outdated
  - messages suffer timestamp (clock) problems

  → we may get inaccurate, inconsistent, even contradicting / false, outdated observation results

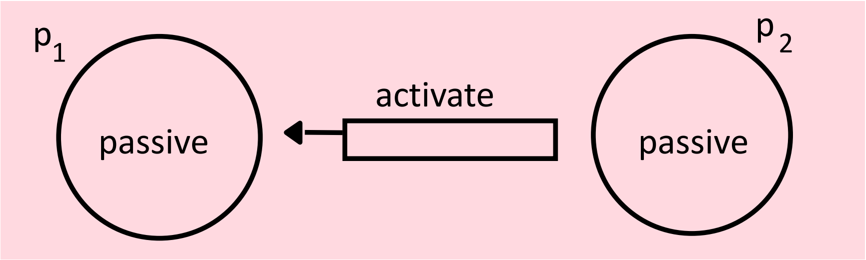
Global States: Intro (cont.)

Further (general) problems that would require the view on a global state

- distributed garbage collection: any reference to an object left?



- distributed termination detection:
  any active process left or any process activation message in transit?



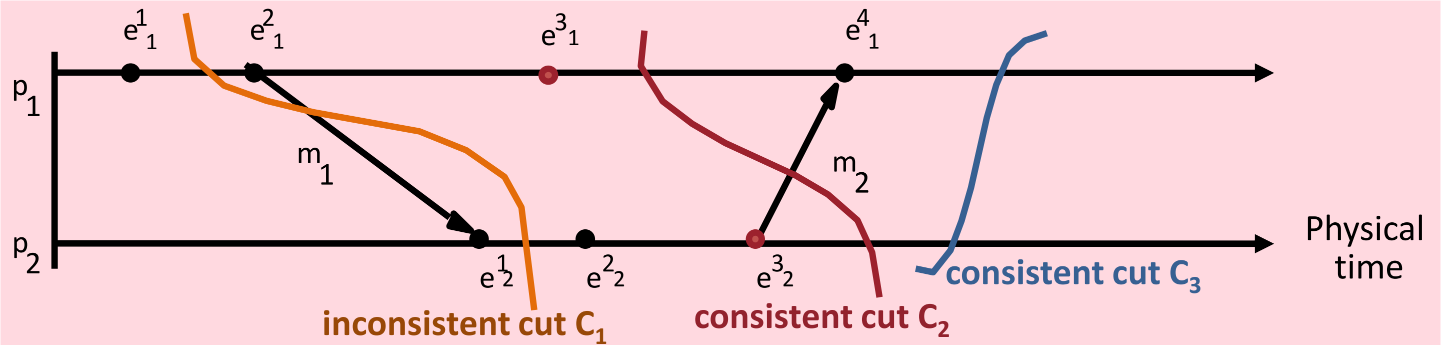
Cuts as Approximation of Global States

compute an up-to-date global state - how close can we get? remember *observation issues:*

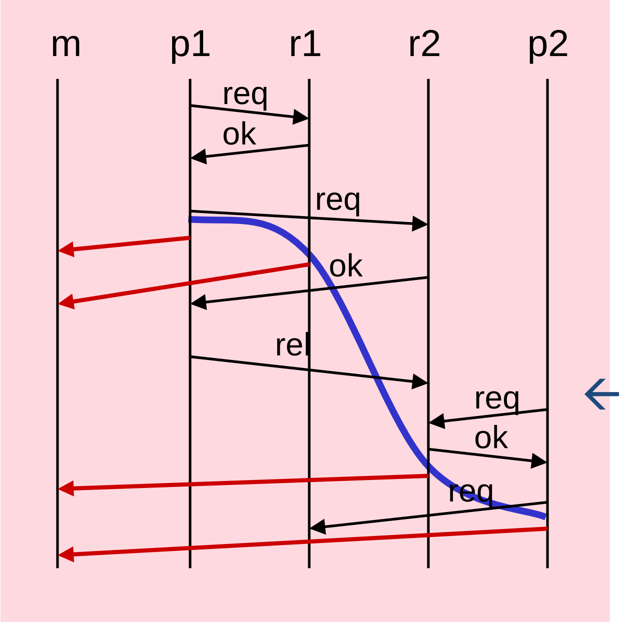
- what can/should be observed? *process events → states* and *communication events → channel states*
- why is it difficult? interrelated 'Bermuda triangle' problems
  - an important sub-issue, see distributed termination & garbage collections: messages-in-transit? (cf. 'channel states')
- a first idea: **observe** events & states of local processes, **infer** 'channel' states i.e. messages-in-transit accept that only **out-dated states** can be 'reconstructed' – but **how well?**

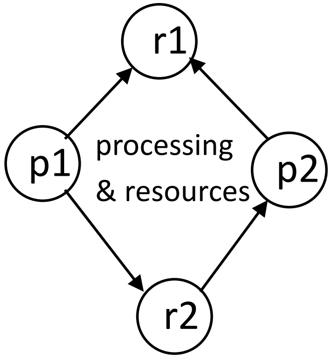
this first idea leads to: **cuts**

- compute an assumed former global state from state information of the processes, such that the resulting "cut" through the system is **consistent:**
  - the cut includes no events that are the effect of another event without that the 'causing events' be also part of the cut (cf. „→" relation), see C_2 below
  - the idea: only events that could have happened 'simultaneously' appear as the-most-recent-event-per-process
    - 'concurrent events' in the sense of HB, such as e^3_1 and e^3_2 for C_2, or
    - 'causal events' such as e^3_2 and e^4_1 for C_3



Global States: Cuts (cont.)



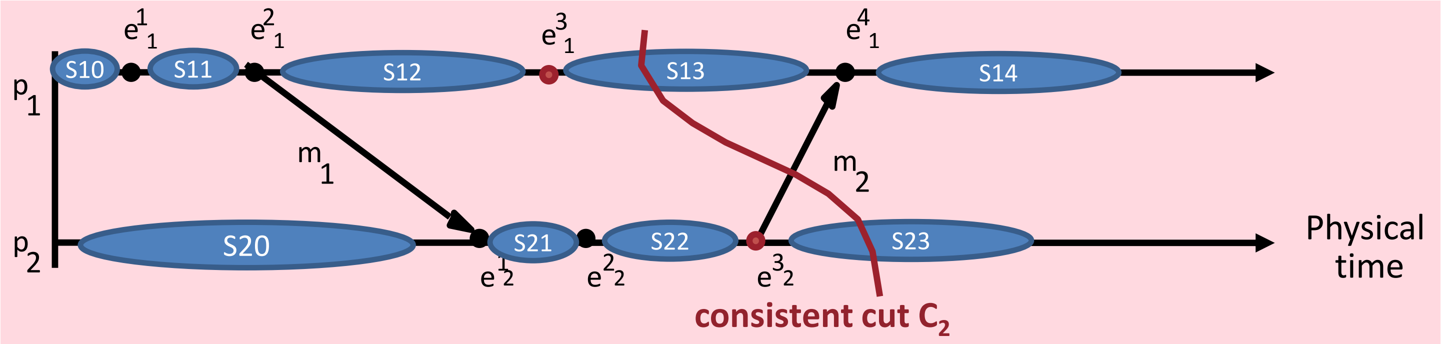


← another example of an inconsistent cut, involving 5 processes:
2 processing (p), resource mgmt. (r), 1 monitoring
why inconsistent? „rel“ (p1→r2):
receive event ∈ cut, but send event ∉cut
in this case, a „phantom deadlock“ would be detected

Inlet about the concept *history:*

- history of process p_i : $h_i = < e_i^1, e_i^2, \ldots>$
  - each e_i^k corresponds to either a send, receive, or internal action
- s_i^k describes the state of process i immediately after the k-th event e_i^k
- (remember idea: record send/receive events as part of state ⇒ channel info recoverable)
- global history: $H = (h_1, h_2, .., h_N)$

Global States: Cuts (cont.)



local states (blue ellipses) indicated here for explanation purposes

Definitions:

1. cut C:== $(h_1^{c1}, h_2^{c2}, .., h_N^{cN})$ such that $(\forall i$: h_i^{ci} a prefix of $h_i)$

- prefix: history 'that may be outdated', i.e. 'except 0…n *last* events'
- this means: each process' history is just divided into past and future

2. consistent cut: cut C for which $\forall$ e $\in$ C: f → e $\Rightarrow$ f $\in$ C

- state of s_i in C is that of p_i immediately succeeding last event processed by p_i in C (i.e., e_i^{ci})
- global system state S = $(s_1, s_2, .., s_N)$ – set of local states
- consistent global state: corresponds to consistent cut
  - considered potentially possible [past] global state
  - e.g., state (S12, S22) could have happened (but did not, in the picture above – which a DistSys cannot see)

Global States

Global state sequence

- consider a system as evolving in a sequence of global state transitions

  $S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow \ldots$

  - precisely 1 process performs local transition in every step of sequence
  - turns the partial order of all events in the system into a total order,
    concurrent events can be thought of as having happened in some total order (linearization) that is consistent with the partial order described through $\rightarrow$
  - S' called reachable from state S if there is a linearization passing thru S then S'
  - possible to further formalize these concepts
- Definitions:
  - Run = ordering of all events (in a global history)
    consistent with $\rightarrow_i$ for each p_i i.e. with each local history's ordering
  - Linearization (also known as 'consistent run') =
    Run *and* consistent with "$\rightarrow$"
    (in addition to $\rightarrow_i$, all related send and receive events are in order)
  - S' reachable from S:
    $\exists$ linearization: …$\rightarrow$ S $\rightarrow$ … $\rightarrow$ S' $\rightarrow$…

Global States vs. Properties

remember properties from intro chapter! here: → state properties

- express syst. properties by defining predicates π evaluated on states
  - e.g., $x(S_i) = 5$
  - e.g., $\text{num\_crit}(S_i) \leq 1$
- safety properties: “No bad thing will ever happen”, e.g.
  - system never deadlocked
    - (note: a *required* deadlock would be a *liveness* and even a *stability* predicate: once deadlocked, sys. will remain so forever)
  - the sum of money in the electronic payment system is constant i.e. never changes
  - there is never more than one process in critical section
- liveness properties: “Eventually something good will happen”, e.g.
  - the system will eventually make progress
  - the system will eventually terminate
  - every sent message will eventually be received (leadsto)
  - if a process requests access to the critical section infinitely often, it will be granted access infinitely often (strong fairness)

Global States (cont.)

Kinds of global state predicates:

- Stability

ω= true in S

$\forall S', S \rightarrow \ldots \rightarrow S' \Rightarrow \omega$= true in S'

- Safety

α = undesirable property

S_0 = initial state of system

$\forall S, S_0 \rightarrow \ldots \rightarrow S \Rightarrow \alpha$ = false in S

- Liveness

β = desirable property

S_0 = initial state of system

$\exists S, S_0 \rightarrow \ldots \rightarrow S \Rightarrow \beta$ = true in S

Snapshots

Definitions ok! Goal now: observation of global states
i.e. distributed algorithms for computing global states

- impossibility to obtain up-to-date global state information →
- goal: take "snapshots" i.e. observe
  consistent global states: states that correspond to consistent cuts
  - no ordering violation within processes
  - no causality violation for send/receive message events
  - local states comprised in consistent global state *could* have occurred all at the same time
- naive logical time based snapshot algorithm
  - given time t known to all processes 1.. N for which snapshot is sought
    - determine state of each process
      after events with timestamp less than or equal to t
    - for every pair (x, y) of processes determine, according to the local event history for x and y, which messages have been sent but are not yet received at time t - those messages are part of the state
  - problem: requires logical clocks

Snapshots

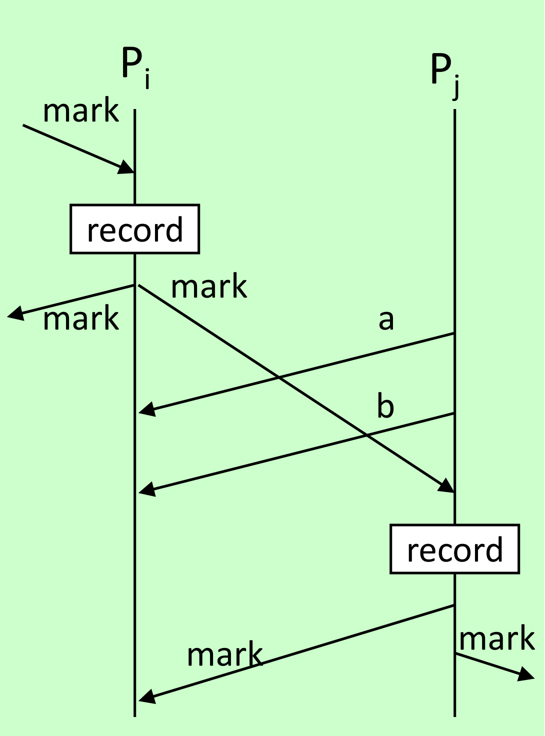
Chandy-Lamport algorithm: determination of consistent global states

- goal:
  observe consistent global states w/o maintenance of logical clocks
- idea:
  - use marker messages broadcasted in the (fully connected) distributed system to distinguish messages sent before or at t from those after t
  - causes processes to locate local and channel state information, does not (yet) include gathering of local state info to form global state
- assumptions:
  - perfect communication: no loss, corruption, reordering or duplication of messages occurs, and messages sent will eventually be delivered
  - unidirectional FIFO channels
  - processes fully meshed, virtually
    (there’s a „path“ from every process to every other process; e.g., TCP)
  - any process may initiate a snapshot-taking at any time (!!)
  - normal system execution continues during snapshot-taking

Snapshots: Chandy-Lamport

First sketch of Chandy-Lamport snapshot algorithm:

- Elements:
  - Players: processes P_i with
    - Incoming channels
    - Outgoing channels
  - Concept: „flood“ channels with (broadcast) marker messages
  - 2 Rules: Marker sending / receiving rule; rough sketch:
    - upon receipt of 1st marker:
      - record own state
      - turn on recording of incoming msgs on every channel; for channel P_j→P_i: record msgs until P_j has recorded its state → sends marker → we receive it (these msgs are considered „in transit“ in channel P_j→P_i for cut)
      - send markers on all outbound channels
    - upon receipt of further markers:
      - stop to record incoming msgs on corresp. channel (as above for P_j→P_i)
      - if markers rcvd. on all channels: send all info to initiator (its ID travels in marker msg)
    - processes may record their state at different points in time, but the differential is always accounted for by the state of the corr. channel
- Start of algorithm
  - a process acts as if it received a marker message



Snapshots: Chandy-Lamport (cont.)

Marker receiving rule for process p_i

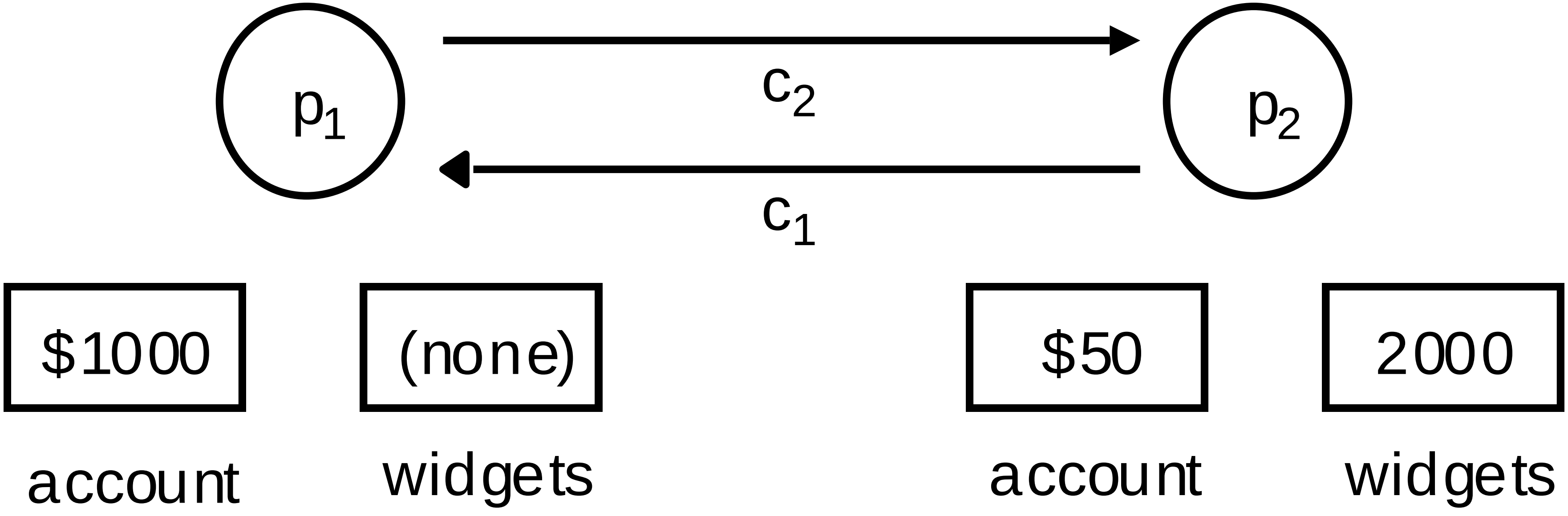
```
On $p_i$'s receipt of a marker message over channel $c$:
    if    ($p_i$ has not yet recorded its state) then it
          records its process state now;
          records the state of $c$ as the empty set;
          turns on recording of messages arriving over other incoming
          channels;
    else
          $p_i$ 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 $p_i$ has recorded its state, for each outgoing channel $c$:
     $p_i$ sends one marker message over $c$
    (before it sends any other message over $c$).
```

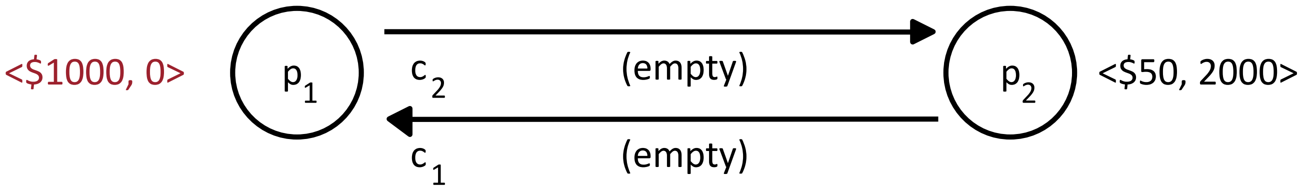
Snapshots: Chandy-Lamport (cont.)

- Example: „electronic commerce“

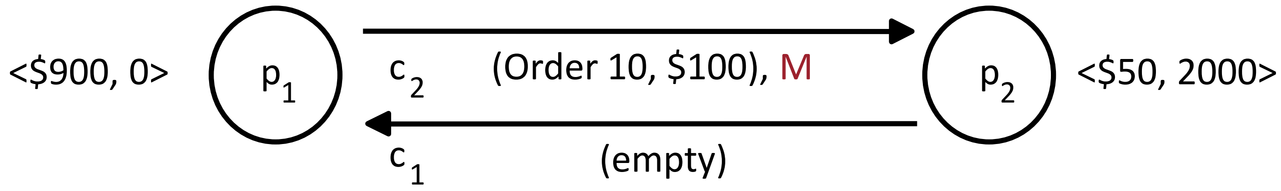


Snapshots: Chandy-Lamport (cont.)

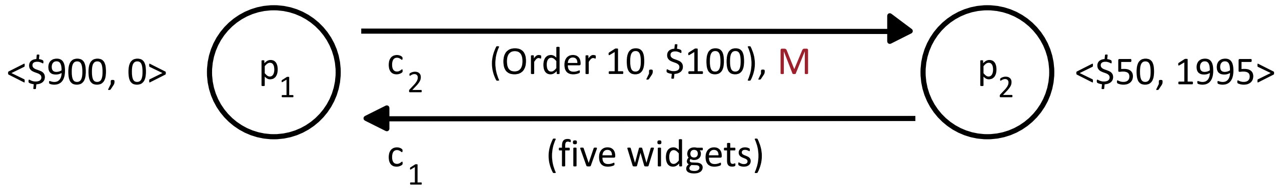
1. Global state S_0



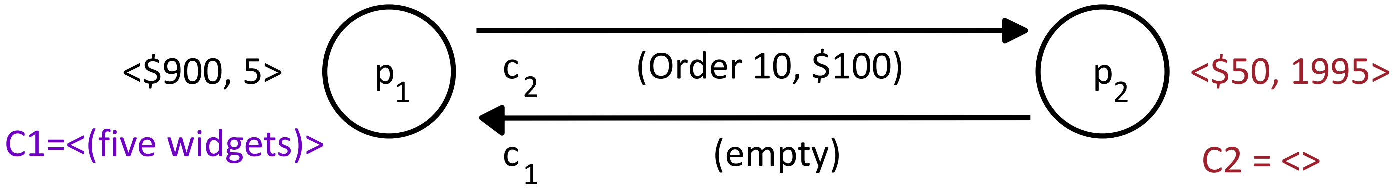
2. Global state S_1



3. Global state S_2



4. Global state S_3

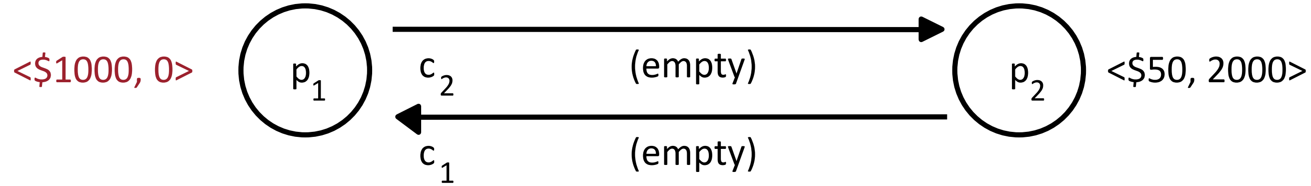


(M = marker message)

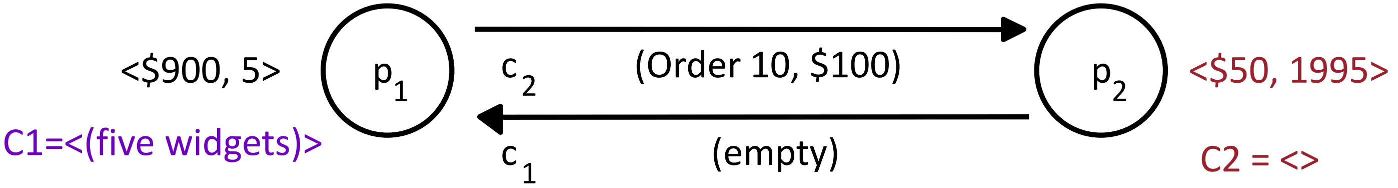
Snapshots: Chandy-Lamport (cont.)

1. Global state S_0

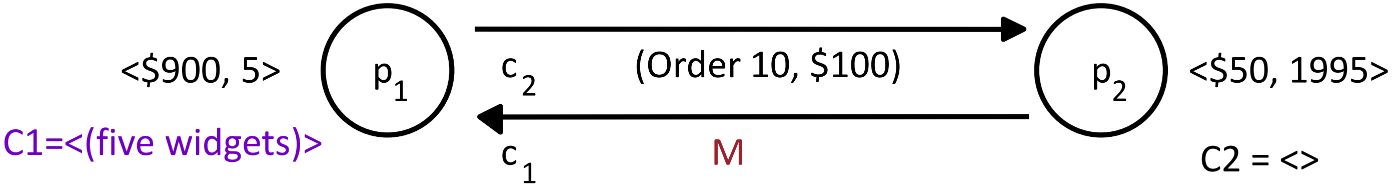
⋮



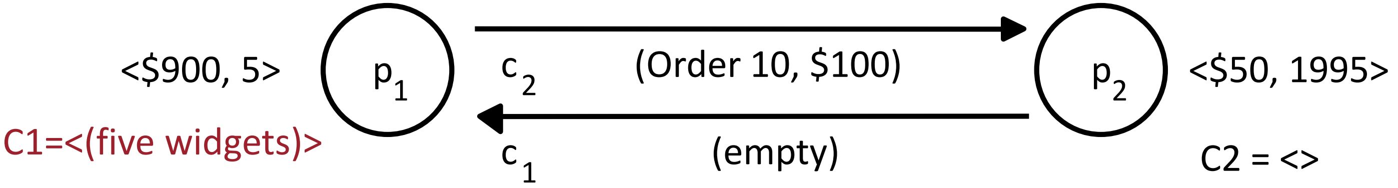
4. Global state S_3



5. Global state S_4



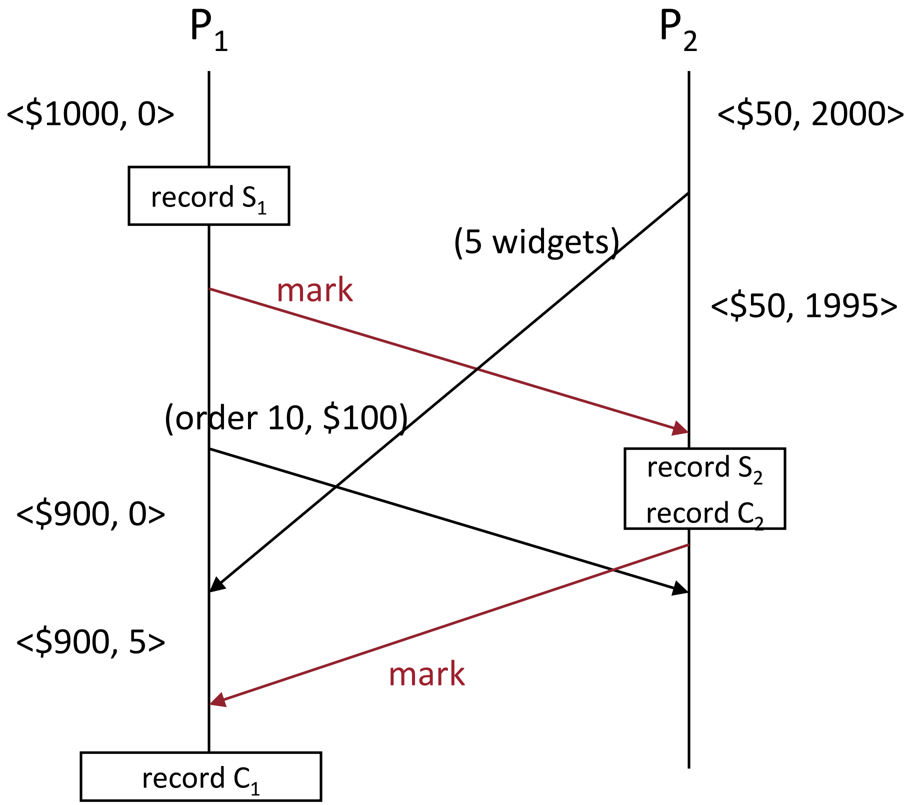
6. Global state S_5



(M = marker message)

Snapshots: Chandy-Lamport (cont.)

MSC (message sequence chart) for example:



- recorded state

  P_1: <$1000, 0>, P_2: <$50, 1995>, c_2: <>, c_1: <(5 widgets)>

Snapshots: Chandy-Lamport (cont.)

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 processes in the graph will have terminated recording their state and the state of incoming channels a finite time after some process initiated snapshot taking.

Snapshots: Chandy-Lamport (cont.)

Theorem: Snapshots taken by the Chandy-Lamport Algorithm correspond to consistent global states

- Proof:

Let e_i and e_k be events at P_i and P_k, and let $e_i \rightarrow e_k$.

Then (we have to prove), if e_k is in the cut, so is e_i.

That means, if e_k occurred before P_k recorded its state, then e_i must have occurred before P_i recorded its state

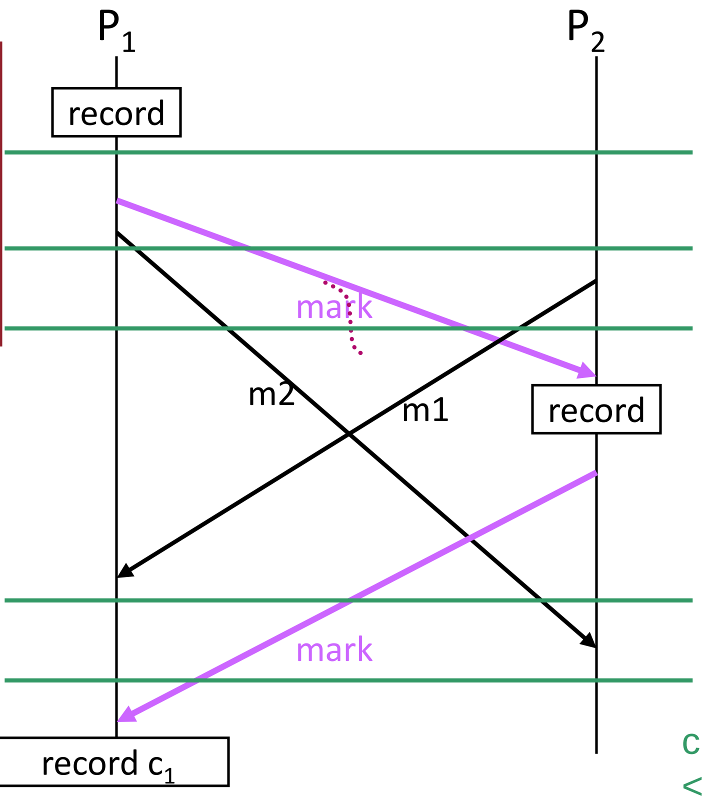
k=i: obvious.

$k \neq i$: assume P_i recorded its state before e_i occurred

- as $k \neq i$ there must be a finite sequence of messages $m_1,\ldots, m_n$ that induced $e_i \rightarrow e_k$
- then, before any of the $m_1,\ldots, m_n$ had arrived, a marker must have arrived at the corresponding receiver (at P_k before m_n), and P_k must have recorded it's state before e_k occurred, hence a contradiction

Snapshots: Chandy-Lamport (cont.)

- as expected: Chandy-Lamport algorithm records a possible global system state, but the actual execution of the system that ‘took’ the snapshot may never have reached it.
- shown here by example: (s_{mi}/r_{mi} denotes send/receive event for msg m_i)



observed snapshot state:

(<>,<s_{m1}>)

→any linearization will have s_{m1} before any of r_{m1}, s_{m2}, r_{m2}

actual computation

S_i (<>,<>)

S_{i+1} (<s_{m2}>, <>)

S_{i+2} (<s_{m2}>,<s_{m1}>)

S_{i+3} (<s_{m2},r_{m1}>,<s_{m1}>)

S_{i+4} (<s_{m2},r_{m1}>,<s_{m1} ,r_{m2} >)

corresponds to linearization <s_{m2},s_{m1}, r_{m1}, r_{m2} >

Snapshots: Chandy-Lamport (cont.)

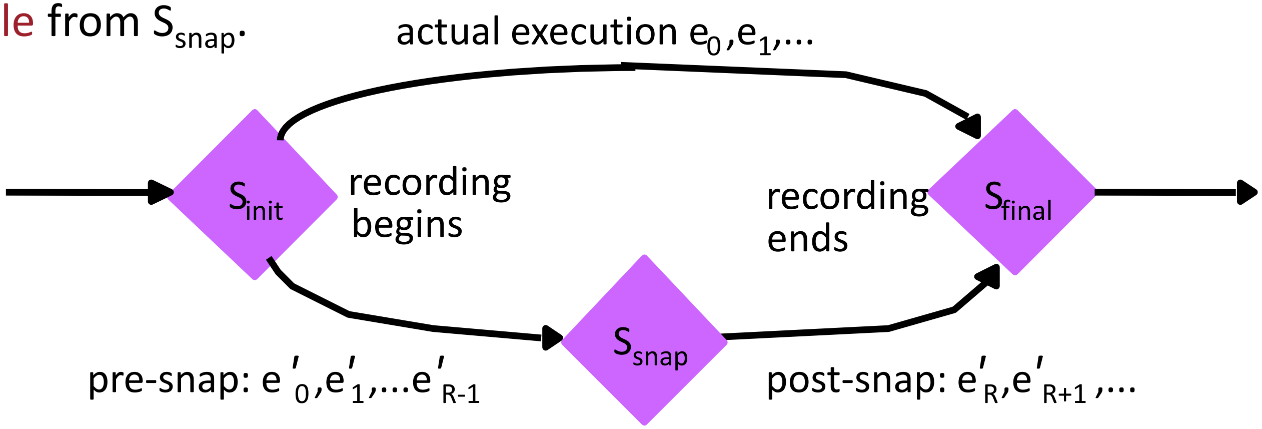
Reachability Theorem: let Sys = e_0, e_1, .. the linearization of a sys. execution

Let:

- S_{init} the initial global state of the system immediately before Chandy-Lamport snapshot-taking was initiated by the first process,
- S_{snap} the recorded snapshot state, and
- S_{final} the global system state after the algorithm terminated.

Then: there is a permutation (and still: linearization) Sys'= e'_0, e'_1, .. of Sys such that

- S_{init}, S_{snap} and S_{final} occur in Sys' and
- S_{snap} is reachable from S_{init}, and
- S_{final} is reachable from S_{snap}.


actual execution e_0,e_1,...
S_{init}
recording begins
recording ends
S_{final}
S_{snap}
pre-snap: e'_0,e'_1,...e'_{R-1}
post-snap: e'_R,e'_{R+1},...

Snapshots: Chandy-Lamport (cont.)

Proof:

- split events in Sys in
  - pre-snap events: occurred before respective process (where event occurred) recorded its state
  - post-snap events: all other events
- how to order events in order to obtain Sys'?
  - assume e_j post-snap event at one process, e_{j+1} pre-snap in *different* process
    - *not*($e_j \rightarrow e_{j+1}$), since otherwise they would be send and receive of same msg, $\Rightarrow$ they would be either both post- or both pre-snap

      $\Rightarrow$ e_j and e_{j+1} may be swapped in Sys'
    - swap adjacent events, if necessary and possible, until Sys' is so that all pre-snap events precede all post-snap events
  - let e'_0, e'_1, .. e'_{R-1} denote the prefix of pre-snap events in Sys', hence the set of events prior to state recording for each process, hence all events leading from S_{init} up to the state recorded as S_{snap}
  - S_{init} & S_{final} unchanged $\Rightarrow$ reachability relationship established

Snapshots: Chandy-Lamport (cont.)

Conclusion:

- Reachability property and system property specification
  - S_{snap} may never have occurred in an actual system execution
  - hence, for an arbitrary state predicate P, $P(S_{snap})$ = true says nothing about the actual system run
- However, let P a stable predicate (stability property), then

  $P(S_{snap}) \Rightarrow P(S_{final})$; $\neg P(S_{snap}) \Rightarrow \neg P(S_{init})$

Consequences for Distributed Debugging:

- Properties
  - typically, user is interested in invariant safety properties
    - the system does not reach a deadlock state
    - the difference between variables x and y is always non-zero
    - the valves v_1 and v_2 may never be open at the same time
  - above snapshot algo. can at best prove violation of these properties
    - works well if the violation is a stability property:
      example again: deadlock - once true (as a debugging issue), it remains true
  - interested in a monitoring algorithm that records system traces in order to decide whether safety properties were, or may have been, violated in a given system run