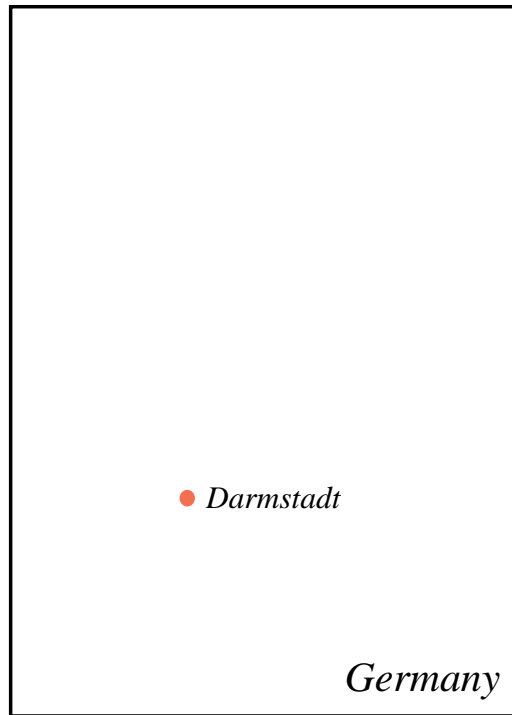


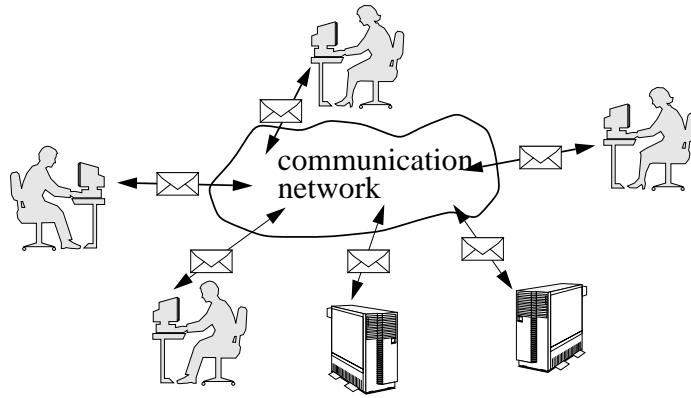
Time in Distributed Systems, Distributed Simulation, and Distributed Debugging

Friedemann Mattern

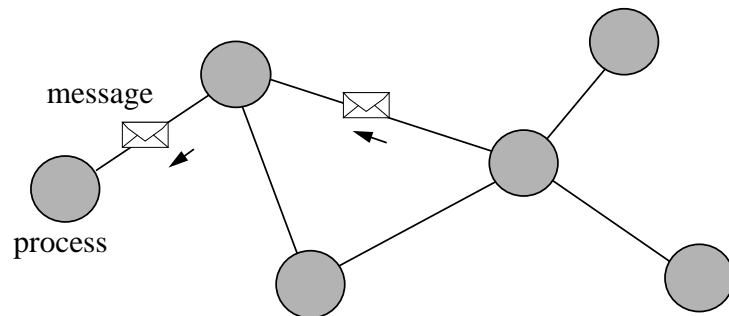
Technical University of Darmstadt, Germany



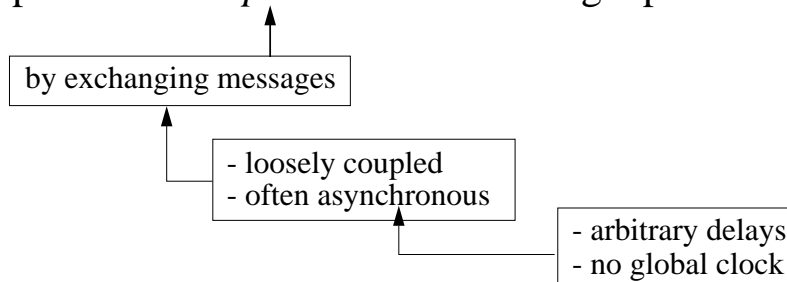
Distributed System



- Machines, persons, processes, “agents”... are located at *different places*.



- The processes *cooperate* to solve a single problem



About the Lectures...

The lectures concentrate on *concepts* (and algorithms)

- they are not about (practical) details
- they are not about (theoretical) formalisms

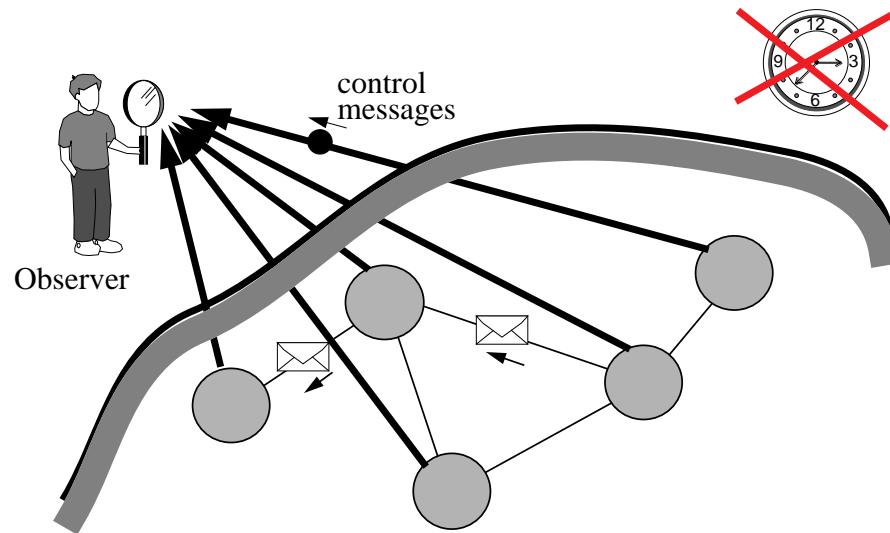
Goal:

Gain insight into the underlying problems, aspects...

- ==> apply this to practical problems
 - ==> formalize the concepts to get nice models
- } “homework exercise”

A Typical Control Problem: Observing Distributed Computations

Deadlock...

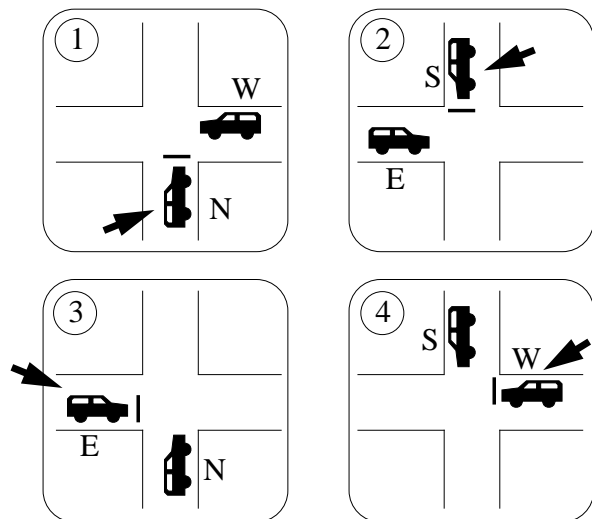


- Observation is only possible via *control messages* (with undetermined transmission times)

"Axiom": Several processes can "never" be observed simultaneously

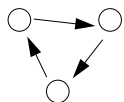
"Corollary": Statements about the global state are difficult

An Example: Phantom Deadlocks

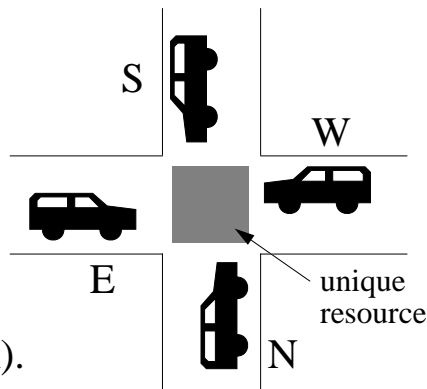


Four single (partial!) observations of the cars N, S, E, W

- 1) N waits for W
- 2) S waits for E
- 3) E waits for N
- 4) W waits for S

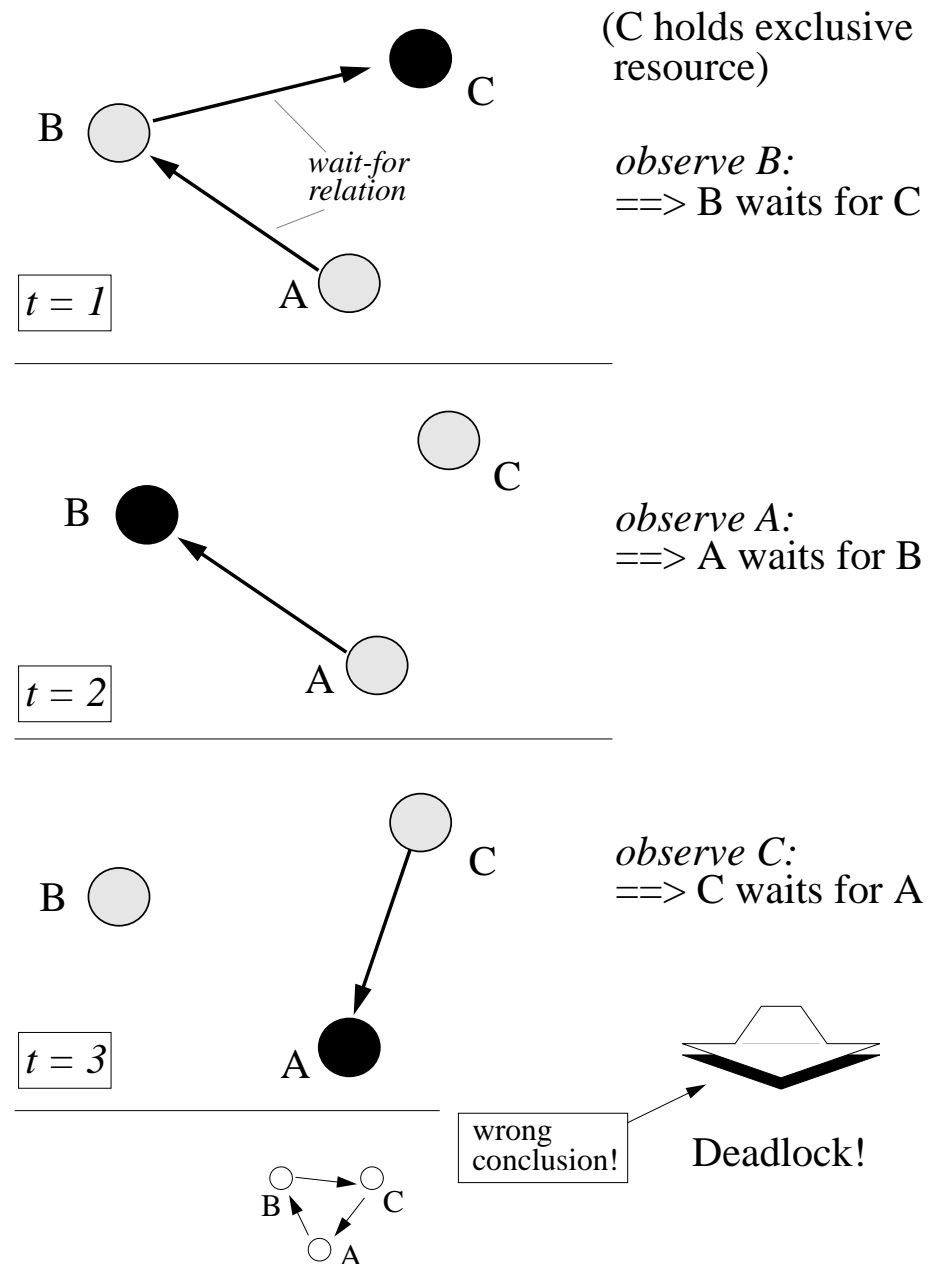


at *different* instants in time yields wrong impression as if there were a cyclic wait condition for a *single* instant in time (--> Deadlock).



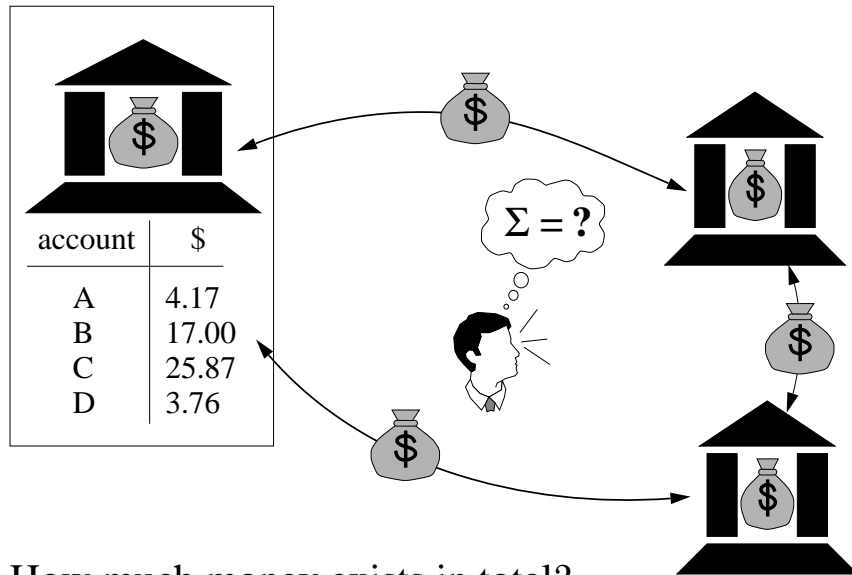
- Required: causal consistency \implies as if simultaneous.

Phantom Deadlocks



An Example: Communicating Banks

- no global view
- no notion of common time

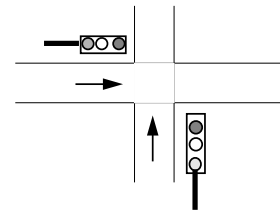


- How much money exists in total?
(if constant; lower bound if monotonically increasing)

- Can this problem be solved? (and if so, how efficiently?)
(Perhaps at least if message transmission is instantaneous?)

- Is it an important problem? (--> consistent snapshots)

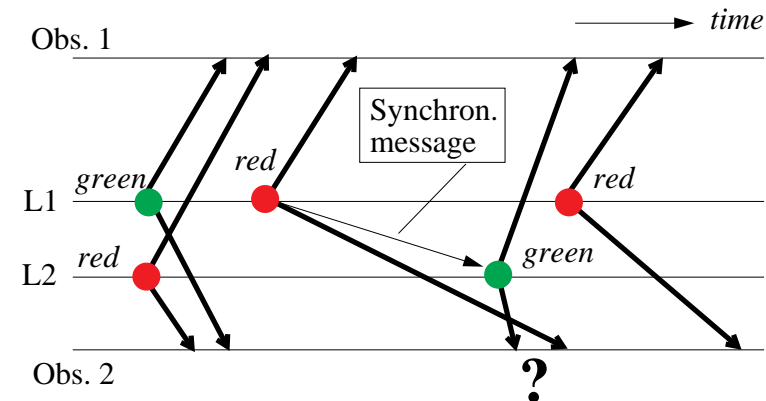
Example: Even More Problems With Many Observers!



Distributed traffic light control

--> safety conditions
(mutual exclusion)

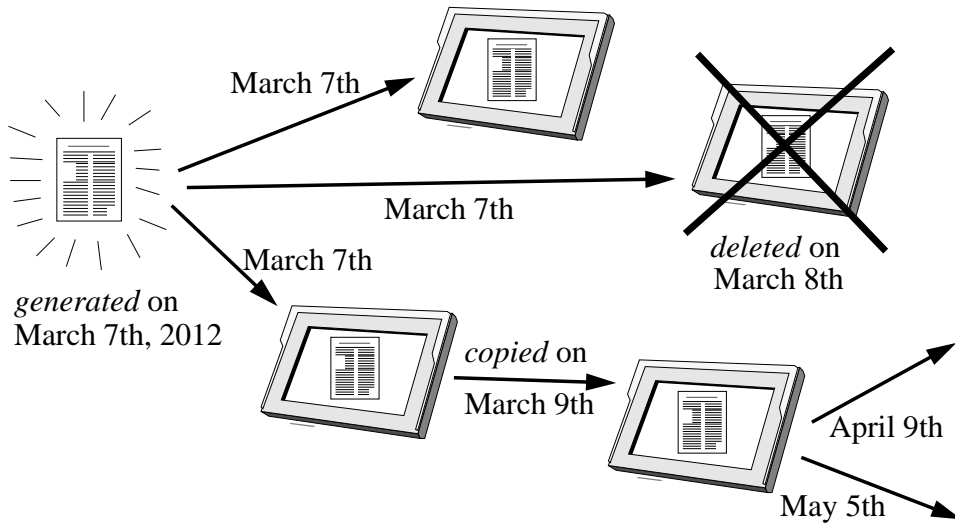
- Each traffic light may switch to red autonomously
- A traffic light may only switch to green if it has learned that the other one is red ("now")
(Token "right to become green" is transmitted by syn. messages)
- State switching is an *event* ●
(*Atomic*: takes no time, action cannot be interrupted)



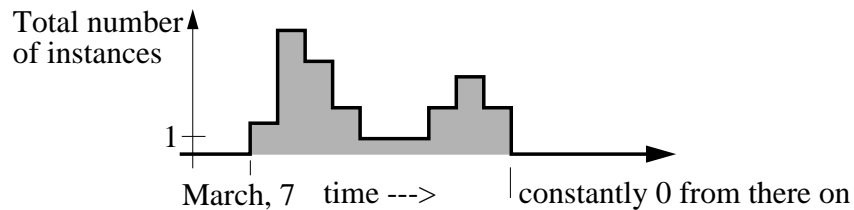
- Which observer is right?

- do we need a notion of global time?
- how can we determine the truth of global predicates?
- in *which sense* is observer 2 wrong?

Copies of an Electronic Newspaper



- New instances (“copies”) might be created from a local instance and then be distributed.
- Instances might be deleted.

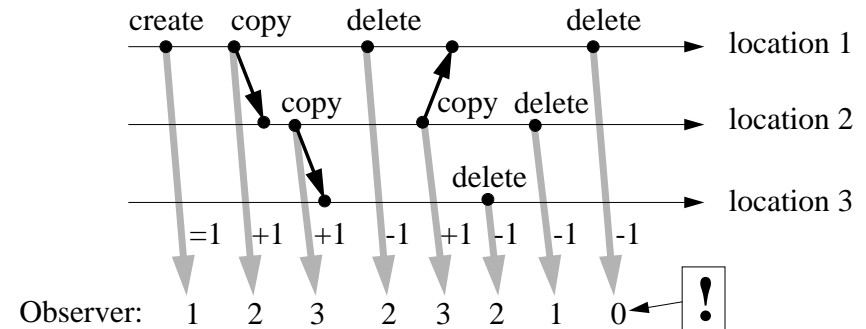


- Interesting question (after March 7, 2012):

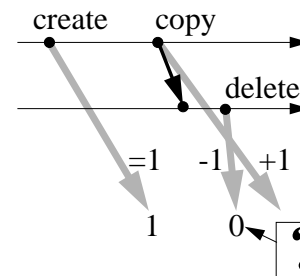
Is the total number of instances = 0 ? } Termination
 ==> newspaper “died out” } detection
 problem

Counting Instances?

- Idea: Observer is informed about
 - unique *create* event
 - each *copy* action
 - each *delete* action



- But: observation is not necessarily causally consistent!



- Note: delete event is a causal consequence of the copy event (“no delete without preceding copy”).

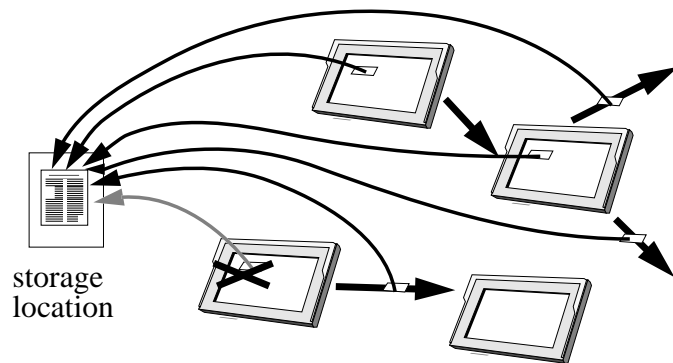
- However: Observer sees consequence before its cause!

- Something (namely “causality”) is *out of order*!

==> Observer may draw wrong conclusions (e.g., “no more instances exist”)

Copying by (Remote) Reference

- With high speed networks "copy by reference" is more sensible than "copy by value".
- Hence: Newspaper instances are read-only, and only a *reference* to the unique storage location is copied
 - Similar to hyperlinks in WWW, e.g. nptp://nyt.ny.us/2012-03-07

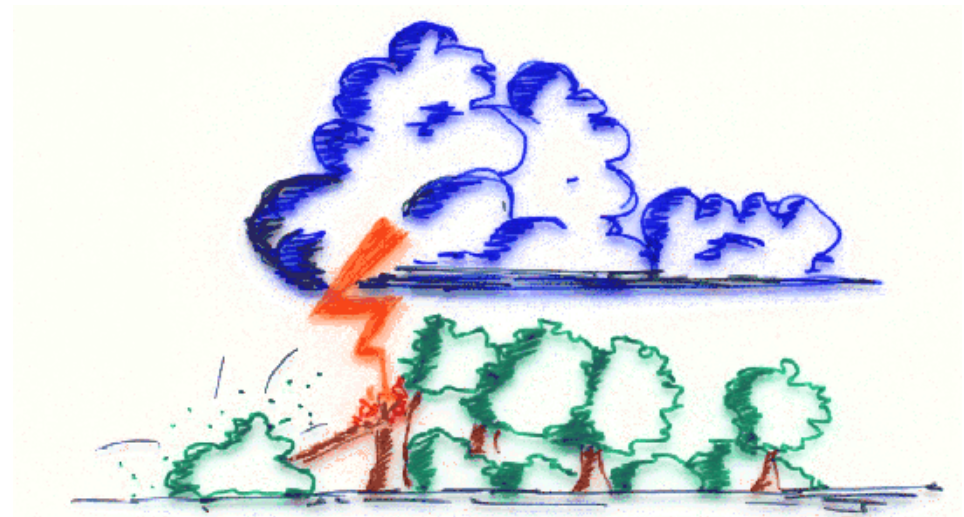


- Copy --> transmit a reference (=address, access path)
- Delete --> remove the reference

-
- Newspaper "died out" if *no more references exist*
 - Reference counter = 0 ==> can no longer be accessed
 - *Garbage collection* problem in distributed systems!
 - Seems to be "related" to the *termination detection* problem!
(In fact, the two problems are equivalent!)
 - *Reference counting* must be done in a causally consistent way! (--> Distributed reference counting)

Example: Prehistoric Society

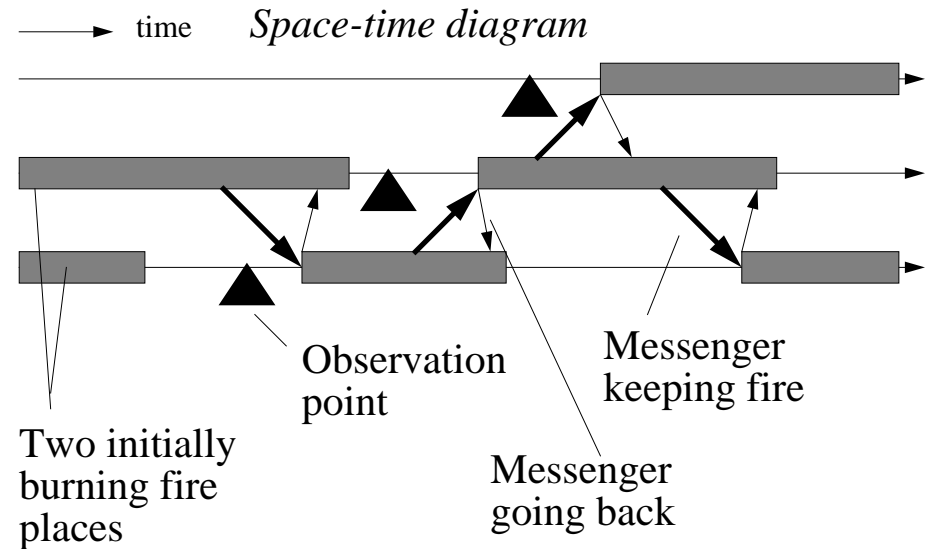
- Organized in local tribes
- Limited technological knowledge
 - > Can't make fire
 - > Keep the fire burning!
- Local fire extinguishes
 - > fetch fire from a remote fireplace with a torch
- Only local view (is there a burning fire somewhere ?)
- If all fireplaces are extinguished and no messenger with a burning torch is in transit --> wait for next thunderstorm (lightning strikes and a tree catches fire...)



- Termination detection is important (no warm meals till next thunderstorm...)



Wrong Observations



For all fire places visited (at some instant in time):

- no fire is burning
- no messenger is in transit

But: There is no *single instant* in time for which no fire is burning.

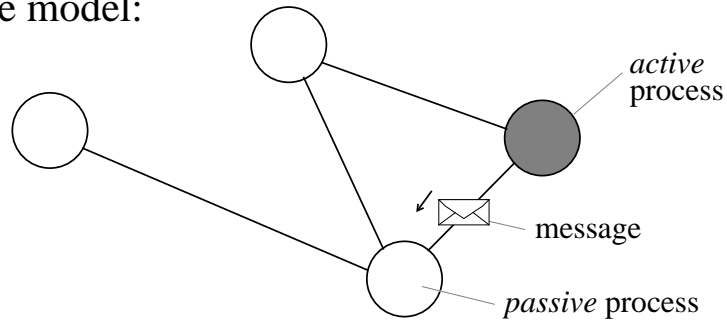
==> Observation is *wrong!*

What can we do to get only correct observations?
(Impossible to observe all processes simultaneously!)

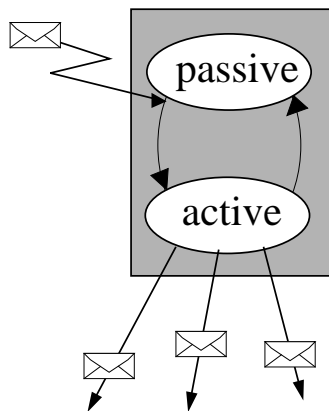
--> General answer later! Now: specific solution.

Distributed Termination Detection

The model:



Message driven distributed (“reactive”) computation:



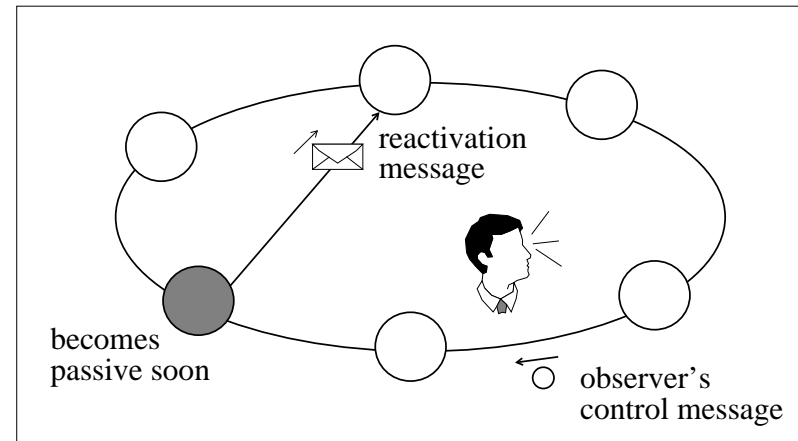
- (1) passive \rightarrow active only on receipt of a message
- (2) active \rightarrow passive spontaneously
- (3) only active processes may send messages

(no spontaneous reactivations!)

Terminated (at t) iff

- (1) no messages in transit
- (2) all processes passive

Behind the Back Activation Problem

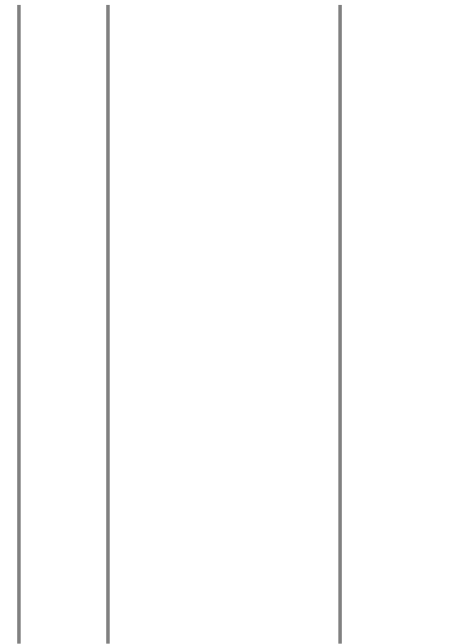
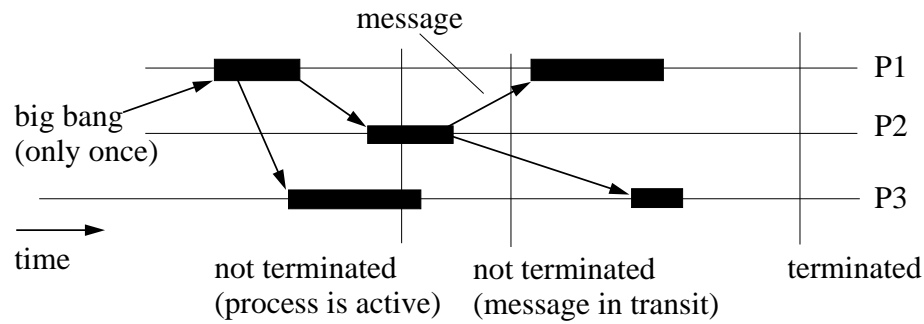


Problem: Implement faithful observer

- using *control messages* (e.g., on a ring) which *visit* the processes and *report* their states
- superimposition of a control algorithm upon the underlying *basic computation*.

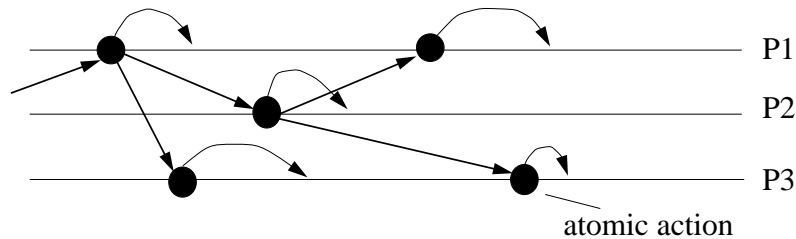
- *Problem:* Determine whether a computation has terminated

The Atomic Model



Idea: Let the duration of activity phases tend to 0.

Model: Process sends (virtual) message to itself when it is activated.
 Message is in transit while process is active.

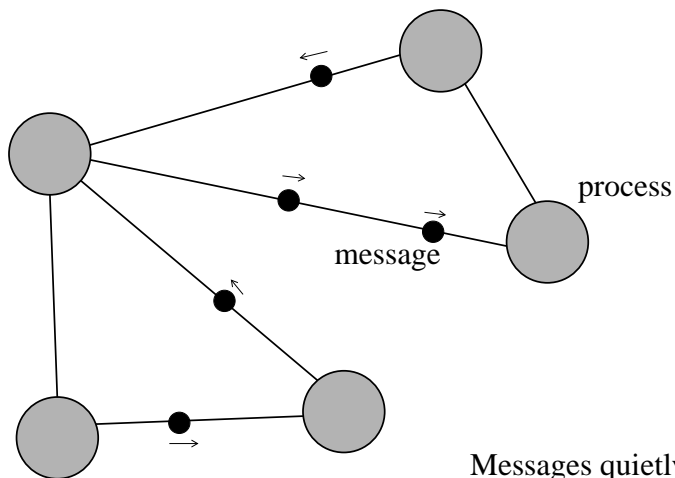


Terminated (atomic model) \Leftrightarrow
 No message is in transit.

\Rightarrow Check whether there are messages in transit

Termination detection problem

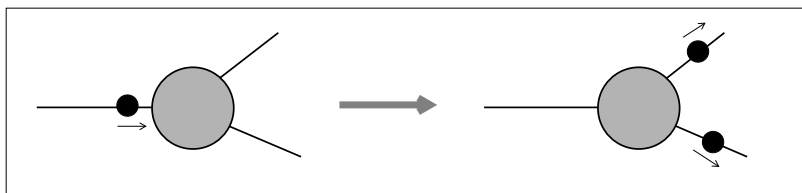
Global Views of Atomic Computations



idealized observer

Messages quietly move towards their targets...

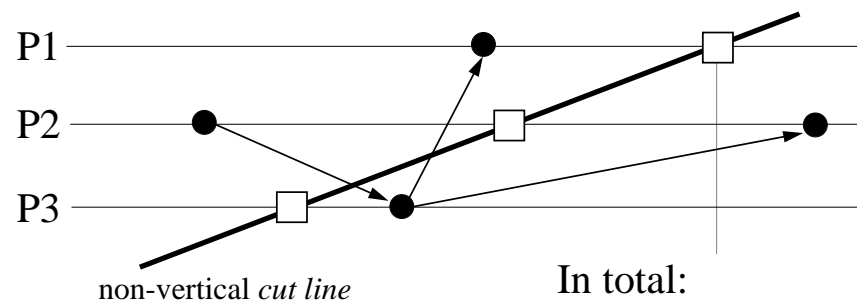
...but suddenly a process "explodes" when it is hit by a message.



Terminated if no ● exists in the global view

Counting Messages?

- Determine whether 0 or >0 messages are in transit.
- Is it correct to count sent and received messages?
- Simple counting is not sufficient! Counter-example:



One does not observe all processes simultaneously

In total:
1 message sent,
1 message received.
But: not terminated!

Reason:

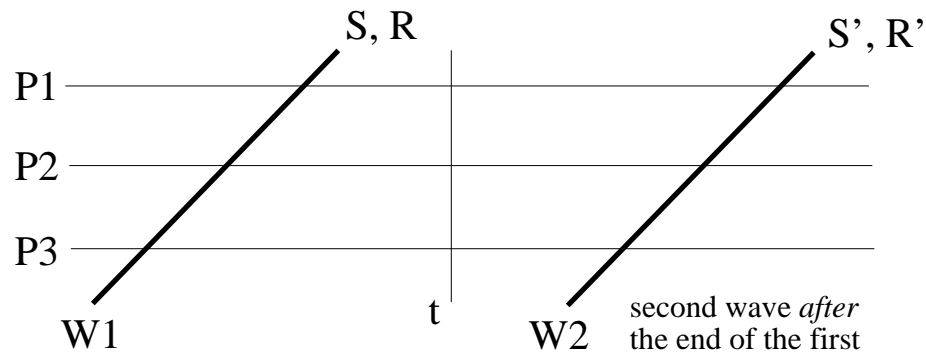
- Message from the "future"
- Inconsistent cut

NB: counting would be correct for a vertical cut!

- Possible strategies to "repair" this defect:

- (1) Detect inconsistent cuts
- (2) Avoid inconsistent cuts

The Four Counter Method



claim: $S=R=S'=R' \implies$ terminated

Proof (sketch):

$S=S' \implies$ no message sent between W1 and W2.

$R=R' \implies$ received

\implies values S and R at t = values of W1.

Hence: $S=R \implies$ at global time instant t:

of messages received = # of messages sent

\implies no message in transit at t

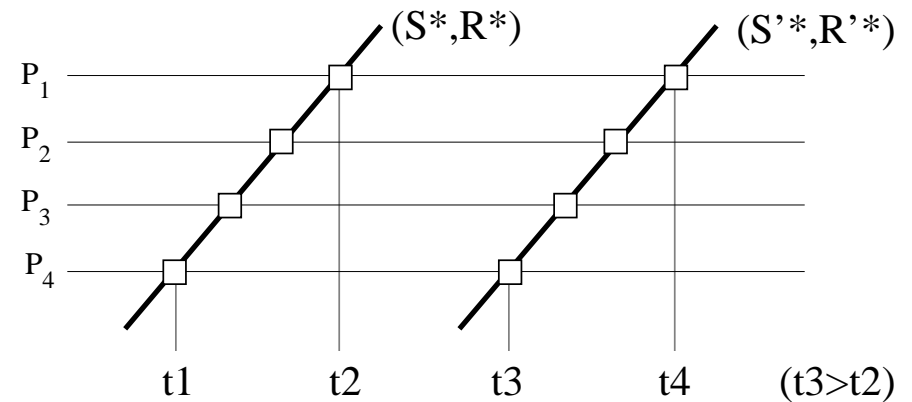
\implies terminated at t

\implies terminated after W1 \square

There exists a more formal proof...

But how does one find such an algorithm?

A Formal Proof



Notation:

- local send counter of process P_i at time t: $s_i(t)$
- local receive counter of process P_i at time t: $r_i(t)$
- $S(t) := \sum s_i(t)$ $R(t) := \sum r_i(t)$

Lemmata:

- (1) $t \leq t' \implies s_i(t) \leq s_i(t'), r_i(t) \leq r_i(t')$ [Def.]
- (2) $t \leq t' \implies S(t) \leq S(t'), R(t) \leq R(t')$ [Def., (1)]
- (3) $R^* \leq R(t_2)$ [(1), r_i is collected before t_2]
- (4) $S'^* \geq S(t_3)$ [(1), s_i is collected before t_3]
- (5) For all t: $R(t) \leq S(t)$ [induction on the number of actions]

Proof:

$R^* = S'^* \implies R(t_2) \geq S(t_3)$ [(3), (4)]

$\implies R(t_2) \geq S(t_2)$ [(2)]

$\implies R(t_2) = S(t_2)$ [(5)]

\implies terminated at t_2

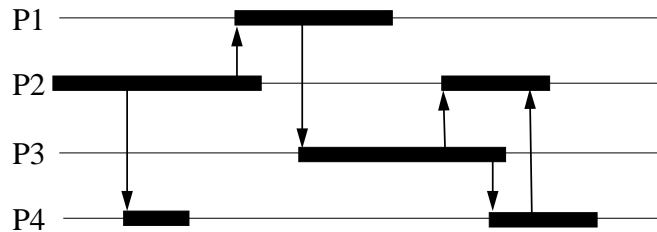
Two
counters
suffice!

Termination Detection for Synchronous Communications

↑ = ? ("same-time": is that possible?)

- Synchronous communication (e.g., CSP or Occam):

- Message arrows can be drawn vertically: (this is indeed justified but it is *not* obvious!) ← messages are never in transit



- Abstract underlying computation modeled with two atomic actions:

$state_p$ takes values *active* or *passive*

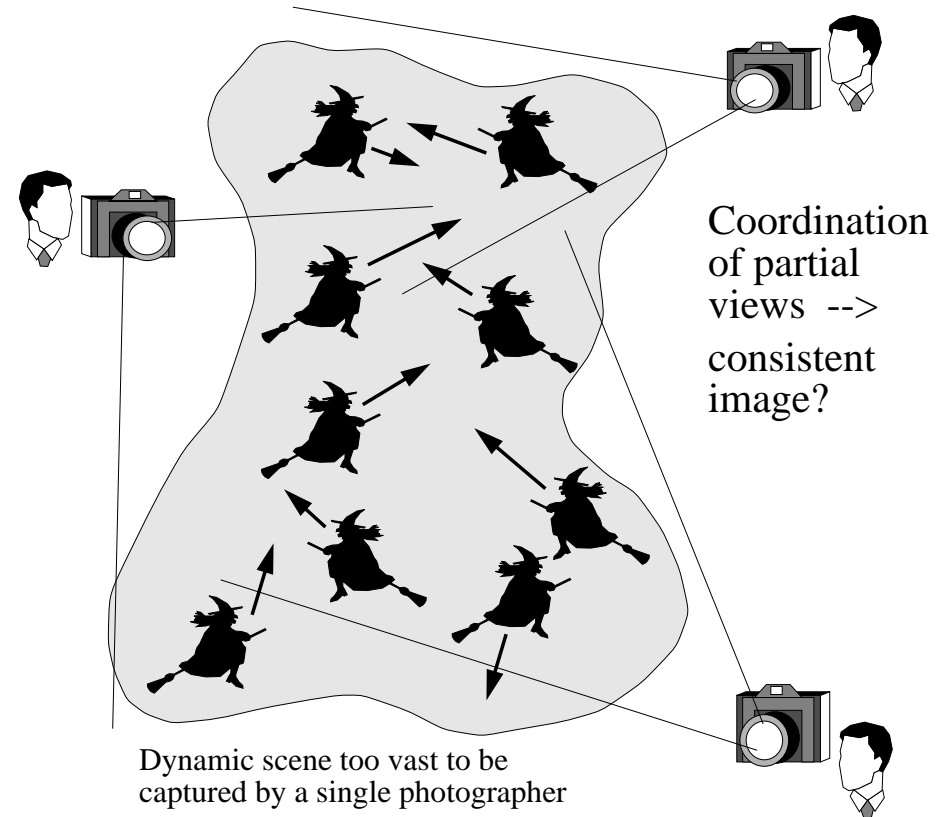
$X_{p/q}$: $\{state_p = active\}$
 $state_q := active$ {"instantaneous" activation}

I_p : $state_p := passive$

messages are of no concern here

- Terminated iff all processes are passive ← "dual" to the atomic model

The Global Snapshot Problem



In reality:

- Population census: fixed time instant (does not work here).
- Inventory: freeze (not practical).

Consistent Snapshots of Global States

Webster:

State = a set of circumstances or attributes characterizing a person or thing at a given time.

But do we have "global time" in a distributed system?

Global state (at a given instant in time)
All local process states + all messages in transit.

Problem: The states of the processes cannot be observed simultaneously!

How can we guarantee consistency?

As if everything were observed simultaneously

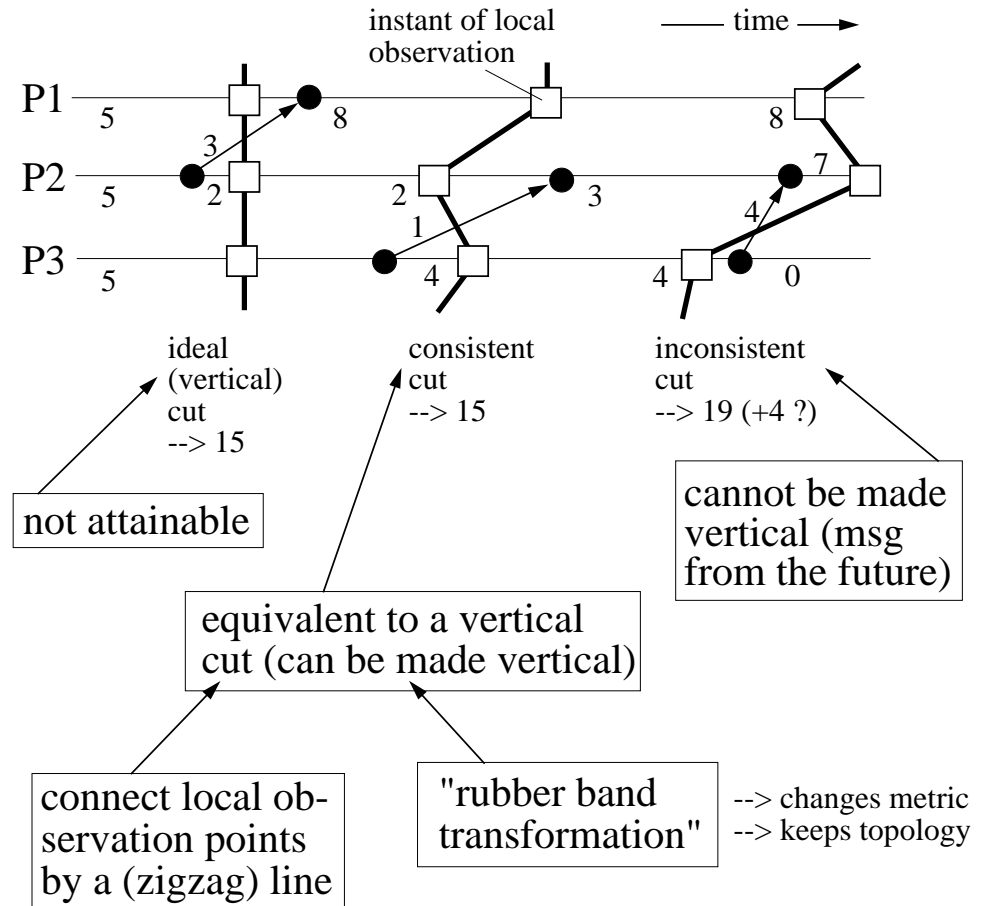
Applications:

- Recovery points for distributed data bases
- Debugging of distributed systems
- ...

Consistent observer: sequence of consistent snapshots

Consistent Snapshots

cf. communicating banks example!



- How can we guarantee that the local observations form a consistent cut?
- How can we observe the messages in transit?

The Snapshot Problem

Goal: "Instantaneous" snapshot of the global state without "freezing" the distributed system.

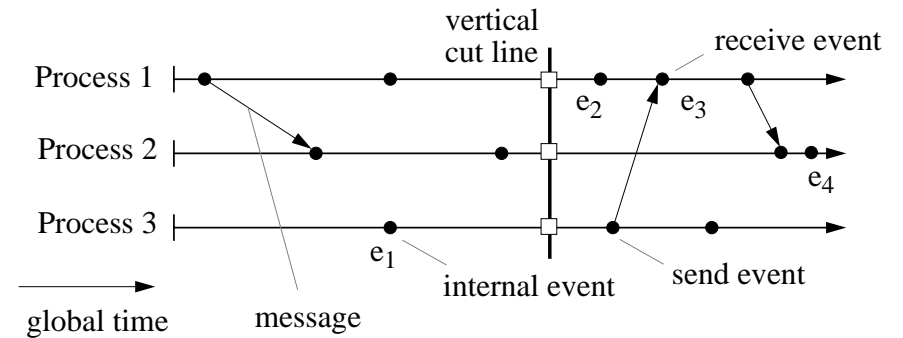
In reality:

- Population census: fixed time instant (does not work here).
- Inventory: freeze (not practical).

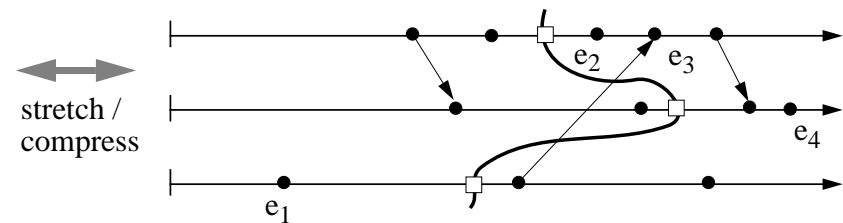
Applications:

- Recovery points for distributed data bases
- Debugging of distributed systems
- ...

Space-Time Diagrams



A different picture of the *same* computation:



Why is it the *same* computation?

Abstract from *real time* -->

Elastic deformations ("rubber band transformation")

Preserves the *causality relation*:

|| $e < e'$ if there is a left-to-right path from e to e'

Message arrows must never go backwards in time! (--> no cycles possible)

|| $e || e'$ ("concurrent", "causally independent")
|| if not ' $<$ ' and not ' $>$ '.

Example: $e_1 < e_3$, but *not* $e_1 < e_2$ ← partial order!

The Causality Relation

- Define the relation ' $<$ ' on the set E of all events:

(causally) precedes

“Smallest” relation on E such that $\mathbf{x} < \mathbf{y}$ if:

- 1) \mathbf{x} and \mathbf{y} happen at the same process and \mathbf{x} comes before \mathbf{y} , *or*
- 2) \mathbf{x} is a send event and \mathbf{y} is the corresponding receive event, *or*
- 3) $\exists \mathbf{z}$ such that: $\mathbf{x} < \mathbf{z} \wedge \mathbf{z} < \mathbf{y}$.



- Why is it a partial *order*?
(i.e., why is it cycle-free?)

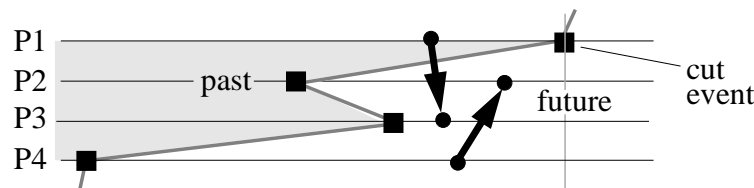
- Terms “happened before” or “causal order” should be avoided (\dashrightarrow confusion)

Consistent and Vertical Cut Lines

such cut lines are called *consistent*

- If no message goes from the “future” to the “past” of a cut line, then this cut line can be drawn vertically in such a way that no messages go from right to left!

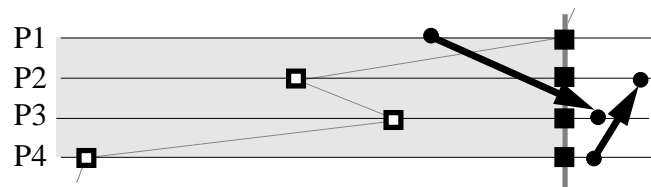
- as if a corresponding wave had visited all processes simultaneously
- obviously useful for termination detection and similar applications



informal graphical proof!



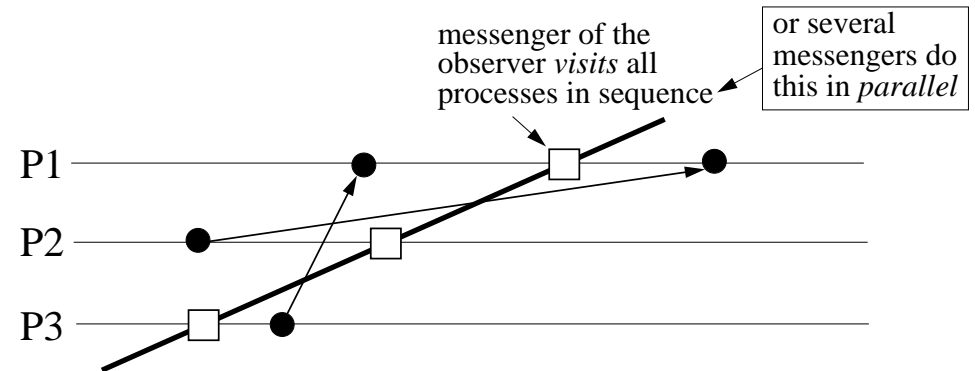
rubber band-transformation



- Move all cut events to the vertical position of the rightmost cut event.
- Events to the left of the cut line keep their position.
- Events between the old and the new cut line are moved just over the new cut line.
 - Corresponding receive events of send events which are moved can also be moved ==> no message arrows go backwards in time!
- Another informal, but “constructive” proof: Cut along the line with a pair of scissors, move right part far to the right; repair cut arrows...
- Formal proof without graphical means: Formally define “cut”...

The Snapshot Algorithm

Yields a consistent view without freezing the system



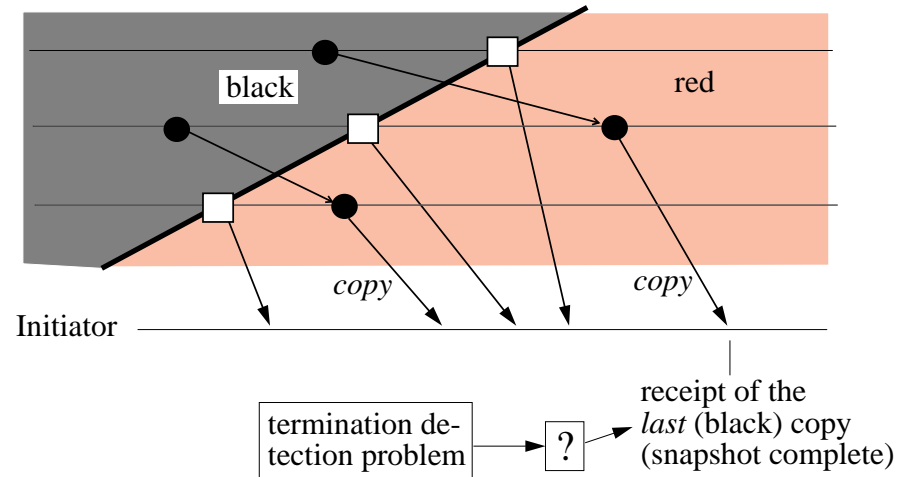
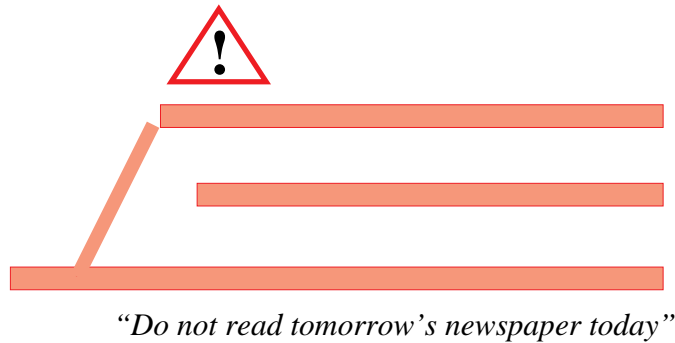
Processes and messages: *black or red*.
 Snapshot instant: black --> red
 then: report local state to the observer.
 Process becomes red if a) it is visited
 b) receives a red message.

Proposition: Snapshot is consistent.
 Proof.: No "message from the future"

The Snapshot Algorithm - Messages

- Messages in transit?

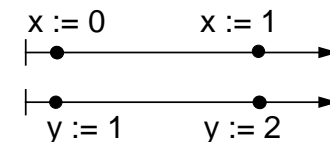
- Black messages received by a red process.
- Send a *copy* of it to the initiator.
- Problem: When does the initiator receive the last copy?



Can we simply *count* the number of sent and received black messages?

But, then: Do we get $x = y$ or $x \neq y$ for our computation?

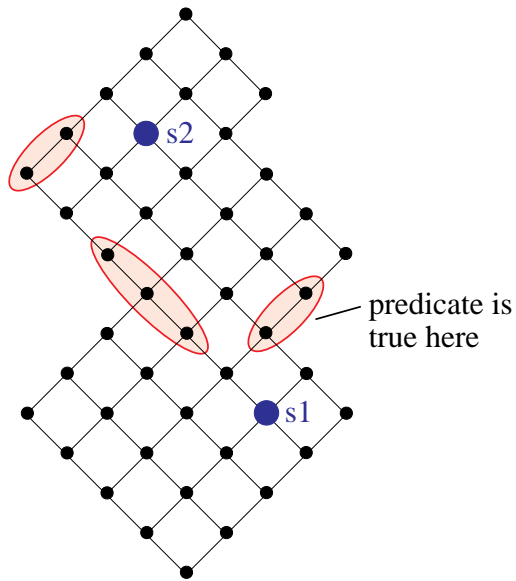
(i.e., which "possible" state do we get with the algorithm?)



How many consistent global states does this computation have?

Detecting Predicates with Snapshots

- Of what value is a (repeated) snapshot algorithm that first yields s1 and then s2?



- Makes sense if the predicate is stable, but otherwise?

NB: The snapshot algorithm is also useful for other purposes, such as determining recovery points, allowing consistent monitoring etc.

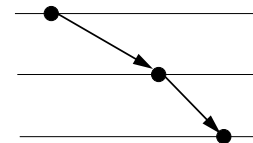
Distributed Computations

- *n-fold distributed computation* (with asynchronous message transmissions) = $(E_1, \dots, E_n, \Gamma, <)$ such that:

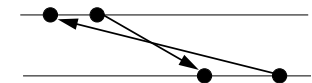
- 1) [Events] All E_i are pairwise disjoint.
- 2) [Messages] Let $E = E_1 \cup \dots \cup E_n$.
For $\Gamma \subseteq S \times R$ with $S, R \subseteq E$ and $S \cap R = \emptyset$ one has:
 - for all $s \in S$ there is *at most one* $r \in R$ s.t. $(s, r) \in \Gamma$
 - for all $r \in R$ there is *exactly one* $s \in S$ s.t. $(s, r) \in \Gamma$
- 3) $<$ is a linear order on each E_i
- 4) $(s, r) \in \Gamma \implies s < r$
- 5) $<$ is an irreflexive partial order on E
- 6) $<$ is the smallest relation which fulfills 3) - 5)
(i.e., there are no other events related by ' $<$ ')

[Causality relation]

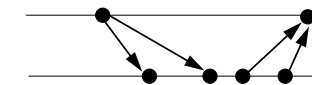
- Counterexamples:



not possible because of (2)



not possible because of (5)



not possible because of (2)

Remarks

- $s \in S$ are called *send events*, $r \in R$ *receive events*
- other events are called *internal events*

Lamport, 1978

- The causality relation ' $<$ ' is often called "*happened before*"

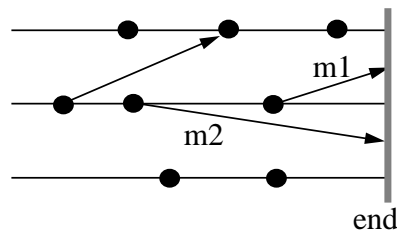
$e < e'$:

- there is a causal chain from e to e'
- e may influence e'
- e' (potentially) causally depends on e
- e' "knows" e

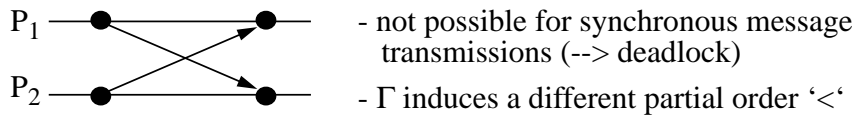
} interpretations

- *Representation* of a computation defined in that way is possible with space-time diagrams

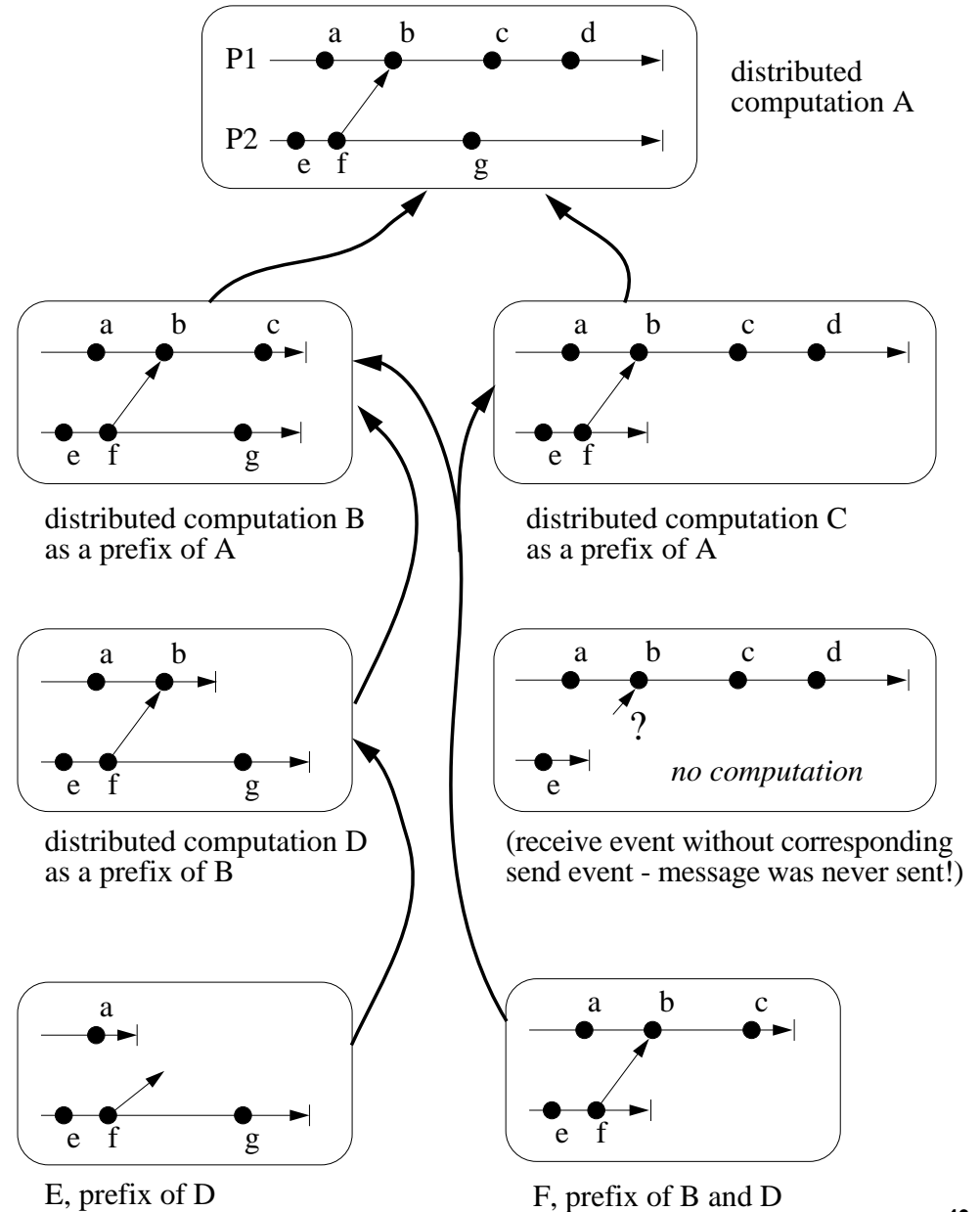
- Definition enables (because of "at most" in item 2) to model *in-transit messages*:



- Distributed computations with *synchronous* message transmissions are modeled in a slightly different way:



Prefixes of Computations

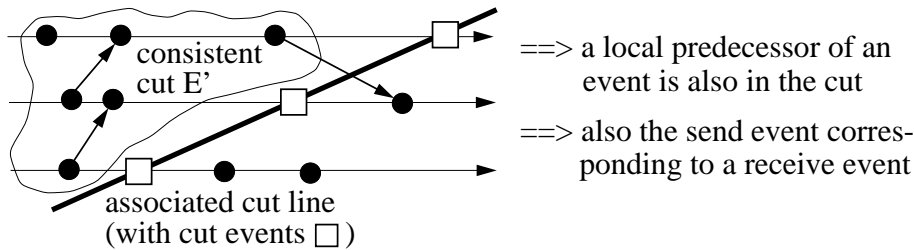


Prefixes and Consistent Cuts

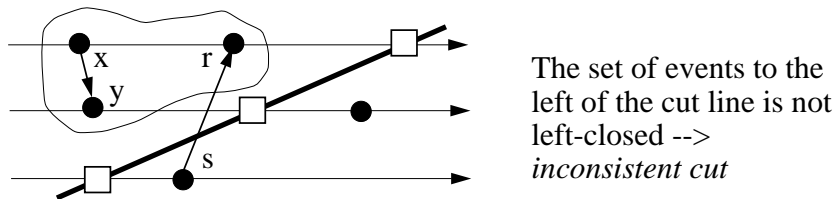
- Prefixes are essentially *left-closed subsets* E' of E with respect to ' $<$ ':

$$\forall x \in E', y \in E: y < x \implies y \in E'$$

- Such subsets are called *consistent cuts*.

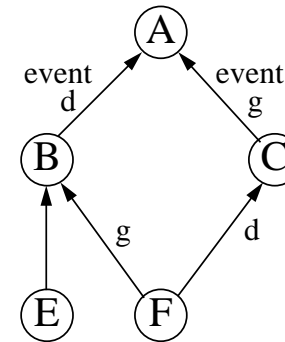


- But: not all lines cutting a time diagram in two parts define a consistent cut!



- General cuts (consistent and inconsistent) are subsets of E which are "*locally left closed*" (' $<$ ' restricted on E_i).
- Cuts can be represented by their *locally rightmost* events.
 - add an initial *dummy event* \perp for each process
 - Example: (r, y, \perp) \leftarrow --> time vectors!

The Prefix Relation



- Graph is directed and contains no cycle.
 - Prefix relation is *transitive*
- \implies Prefix relation is a *partial order*!

- Each consistent cut corresponds to a prefix computation.
- Such a (finite) computation has a final global state.
- Hence one can *associate a global state to a consistent cut*.

- Consider computation A.

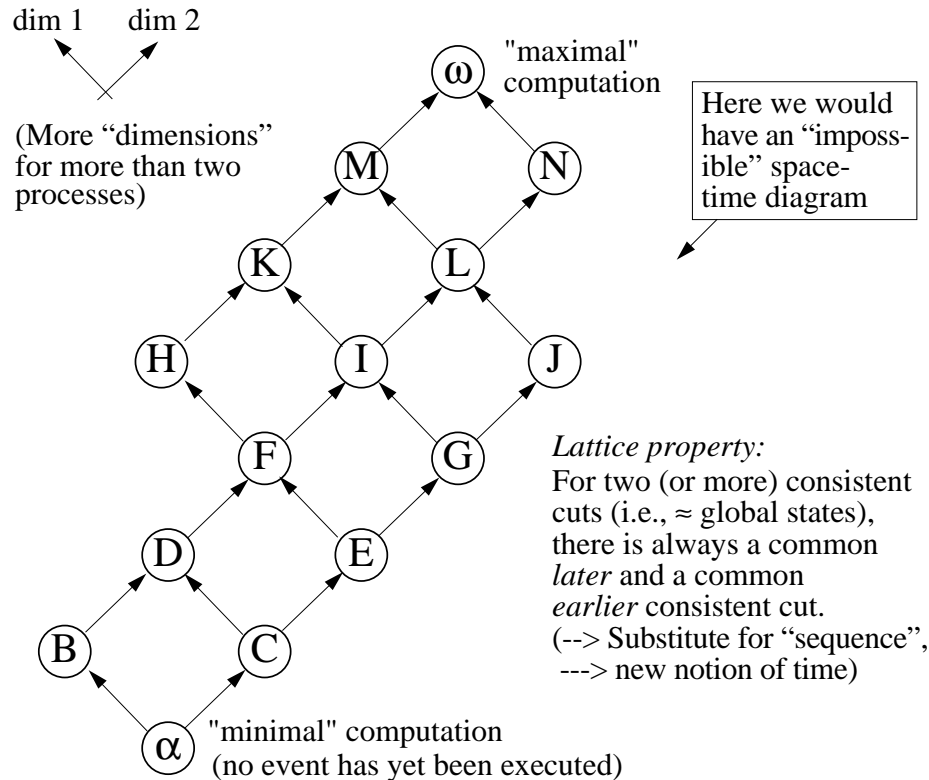
- Was the final state of B or the final state of C an intermediate state of computation A?
- Equivalently: Did d happen before g or vice-versa?
 - Note: Both cases are mutually exclusive (no simultaneous events)

\implies (Executions of) distributed computations are *not sequences of global states* (or of events)! --> no *total* order

- But what then? Is there an adequate *substitute*?

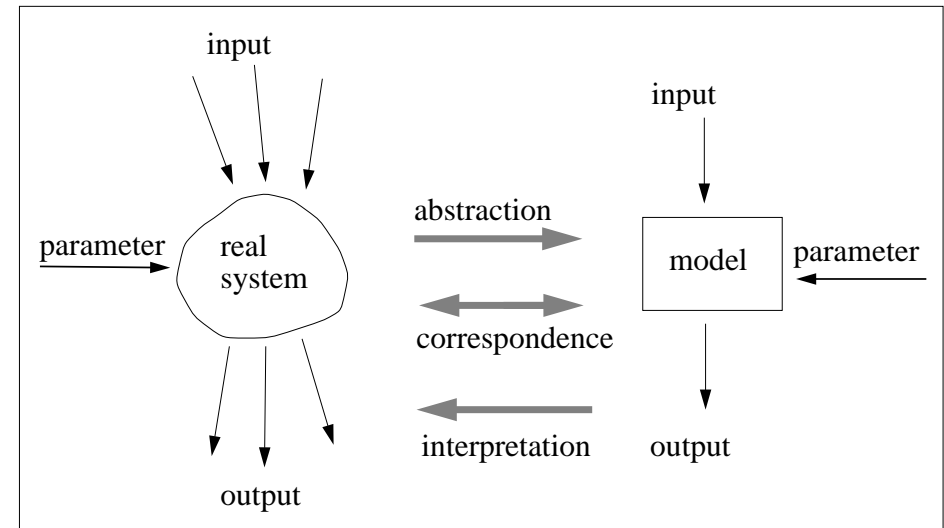
The Prefix Lattice

- Pictorial and mathematical *lattice* of “happened” events.
(i.e., a *partial order* with some additional properties)



Parallel and Distributed Simulation

- Simulation = Experiment with a *model* of the reality.



- Computer based simulation =
Executing a *programmed dynamic model*.

- Used when experiments with reality are

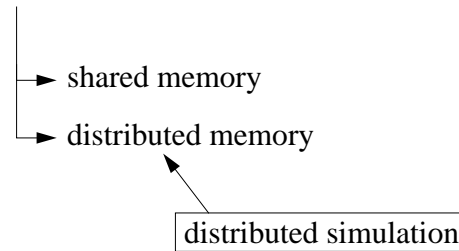
- not possible
- too costly
- too dangerous
- ...

- An intermediate state usually has *several* direct predecessor and successor states!
- Execution moves upwards in a vague and indefinite way!
==> *Uncertainty about the “true” global state!*

Parallel Simulation?

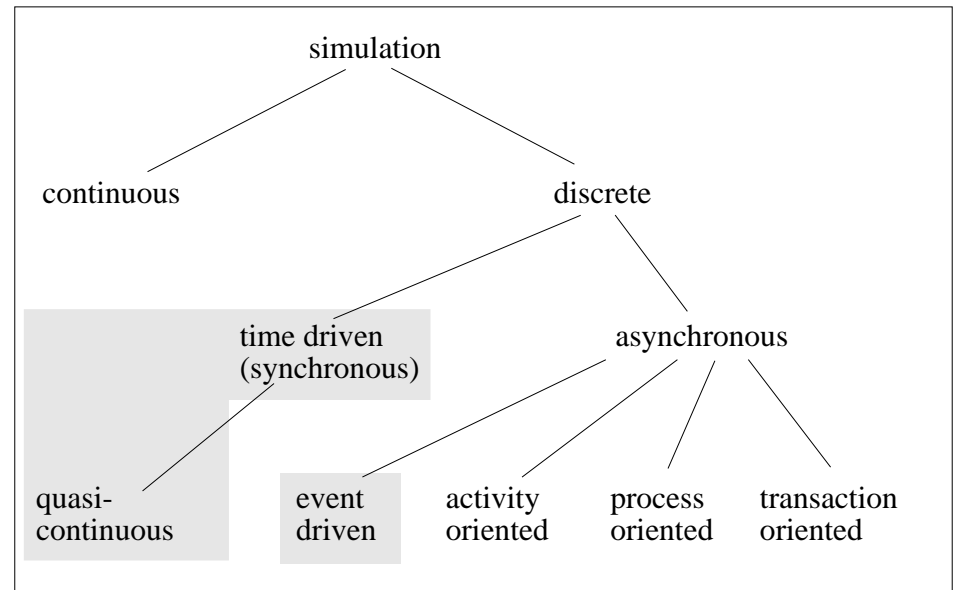
- Simulations are often very time consuming
 - large, complex models
 - many parameters
 - long runs to reduce the variance in stochastic experiments
- Speeding up simulations is very important!
 - many applications in science, engineering ...

- How can one use parallel computers for that?



Simulation Principles

- Usually: analyze development of a system in time
 - > State of the model is advanced "step by step" in simulation time
- *Classification of simulation schemes*



- *Simulation paradigm*

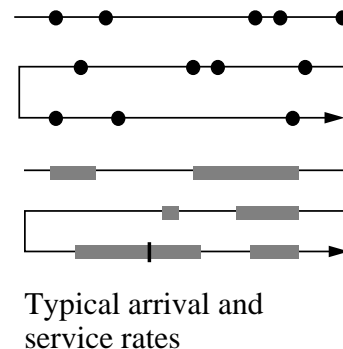
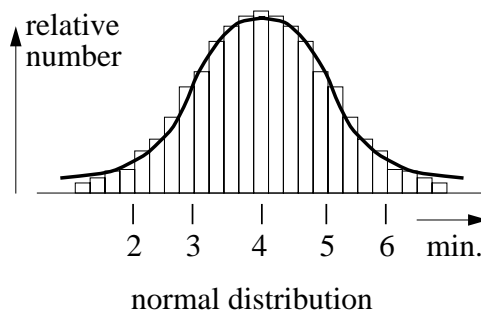
- methods, strategies, modeling styles
- typical simulation languages
- typical application classes
- "world view"

Example of an Event-Driven Simulation

“Booking planes by telephone in a travel agency”

System specification:

1. 5 clerks wait on the phones.
2. 18 phone lines (i.e., at most 13 clients are waiting).
3. “Please wait” when all clerks are busy.
4. Clerk becomes ready --> longest waiting client is served.
5. Clients wait 4 minutes on the average (norm. distrib.).
6. Clients give up if no line is free or if they have been waiting too long.
7. Arrivals are exponentially distributed (mean 20 sec.).
8. Service times are exponentially distributed (mean 1 min for one way, 2 minutes for round trip ticket).
9. Probability for round trip ticket = 0.75.



Simulation Experiments

Analysing the system:

- average waiting time of a client (--> 70 seconds)
- idle times of the clerks (--> 9%)
- utilization of the phone lines (--> 45%)
- percentage of immediately served calls (--> 88%)
- number of clients who gave up (--> 2160 of 18000)
- ...

Possible experiments:

- 1) More clerks--> effects?
- 2) Less clerks --> consequences?
- 3) Consequences of reducing the service time to 55 sec.?
- 4) ...

Event Driven Simulation

Basic assumption:

Model state remains constant between two events

-->

- time “jumps” from event to event
- only events change the state of the model

Typical events

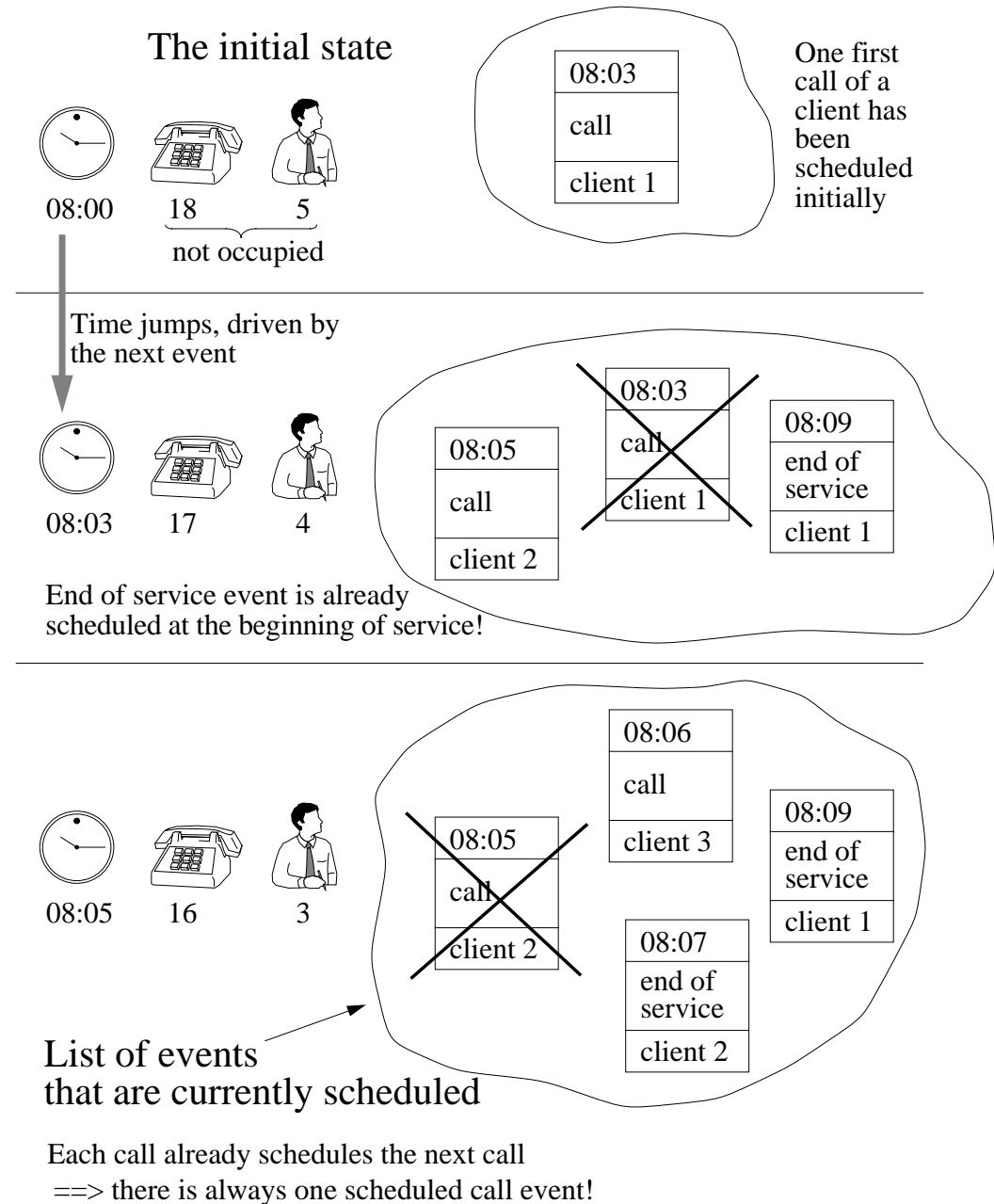
- call of a client
- enqueue at a waiting line
- starting an action
- ...

Event:

- has an associated time (when it will happen)
- if it happens, it “instantaneously” (in simulation time!) changes the state of the model

- Events propel the simulation
- *Events drive simulation time* (i.e., the advancement of the simulation clock)

The Experiment



And so on until:



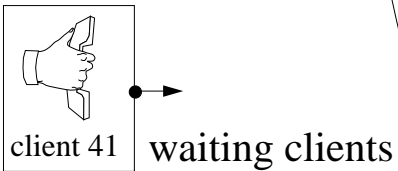
- 5 scheduled end of service events
- 1 scheduled call event



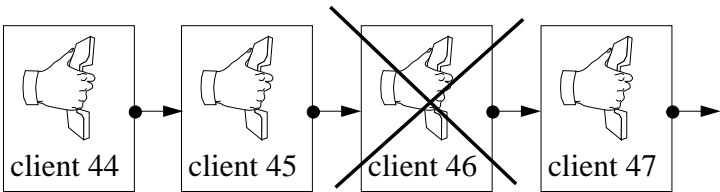
- 5 scheduled end of service events
- 1 scheduled call event

08:55
give up event
client 41

Is scheduled by a call event when all lines were busy



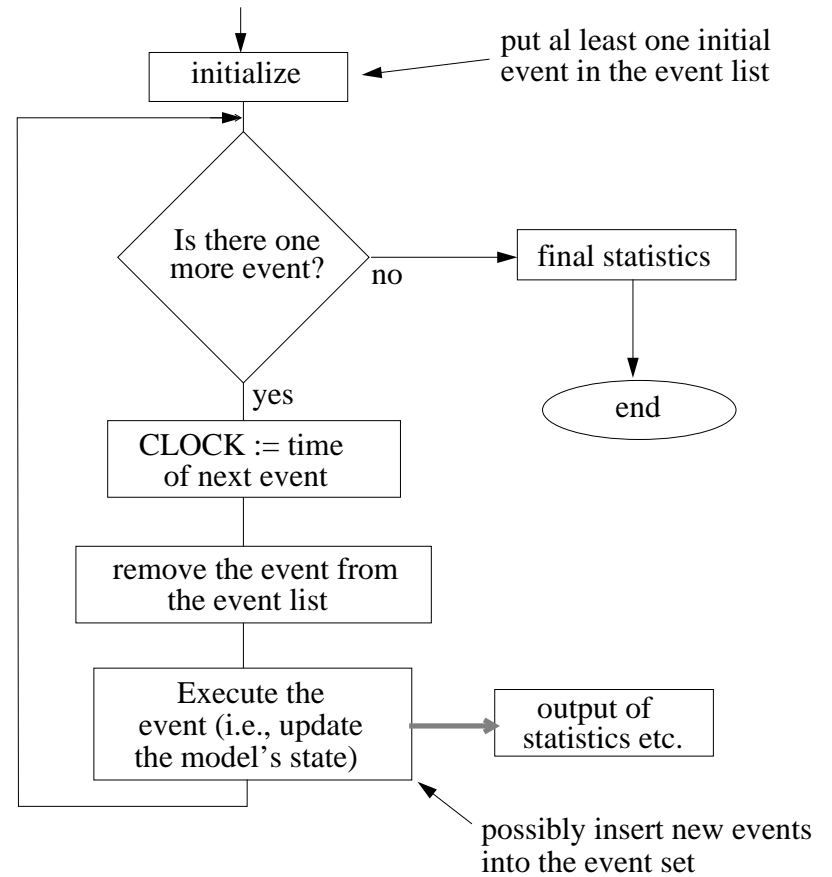
- 3 scheduled give up events
- ...



first client will be served next

client 46 gave up

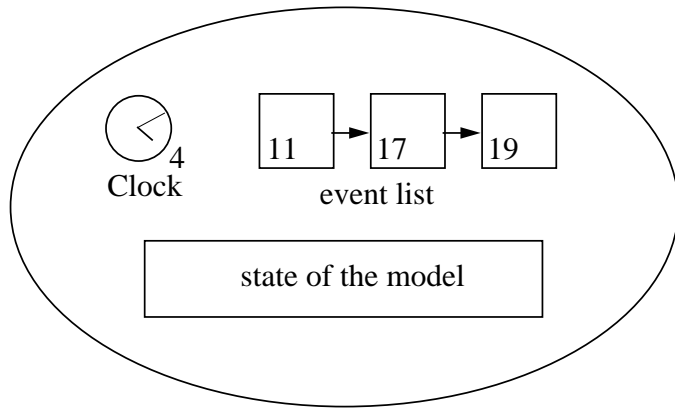
The Simulation Cycle



Idea: - Execute the next event (i.e., the event of the event list with the smallest time).

- This might produce new events which are then inserted into the event list.

Event-Driven Simulation



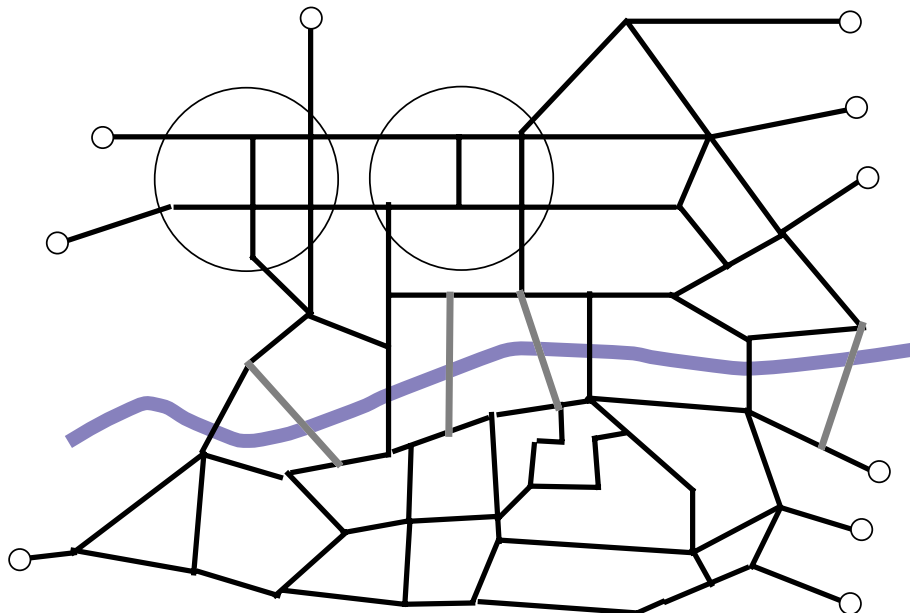
- Simulation time jumps to the next event.
 - Execution of an *event routine*:
 - Changes the model state.
 - Possibly schedules new events (in the future).
- simulation cycle

- *Parallelization* by partitioning the model into autonomous submodels.
- Goal: speedup

Example: Traffic Simulation of a City

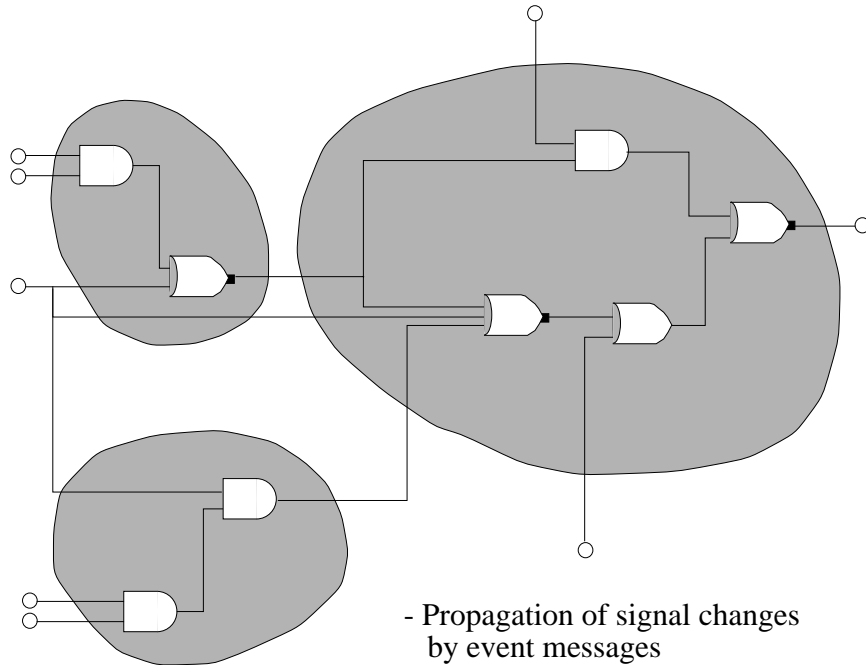
Where should the new bridge be built?

- average time to traverse the city
- various traffic densities



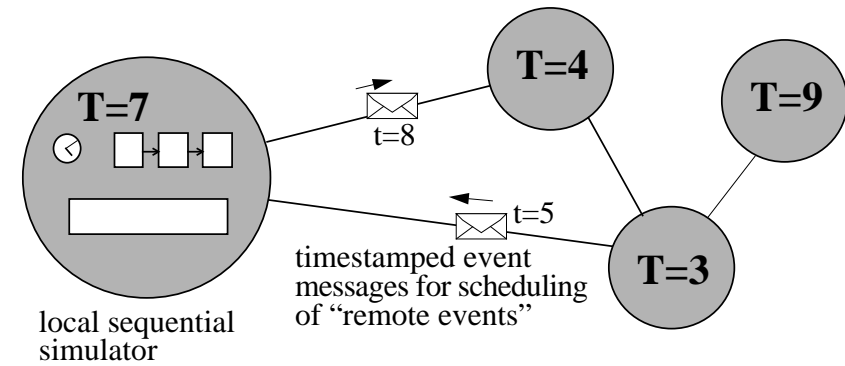
- One *submodel simulator* for each town district.
- *Cooperation* by timestamped event messages. (remote event scheduling)

Example: Logic Simulation



--> Partitioning, mapping, dynamic load balance...
(very important to get significant speed-up values!)

Distributed Simulation



- Clocks have different values --> necessary for speedup!

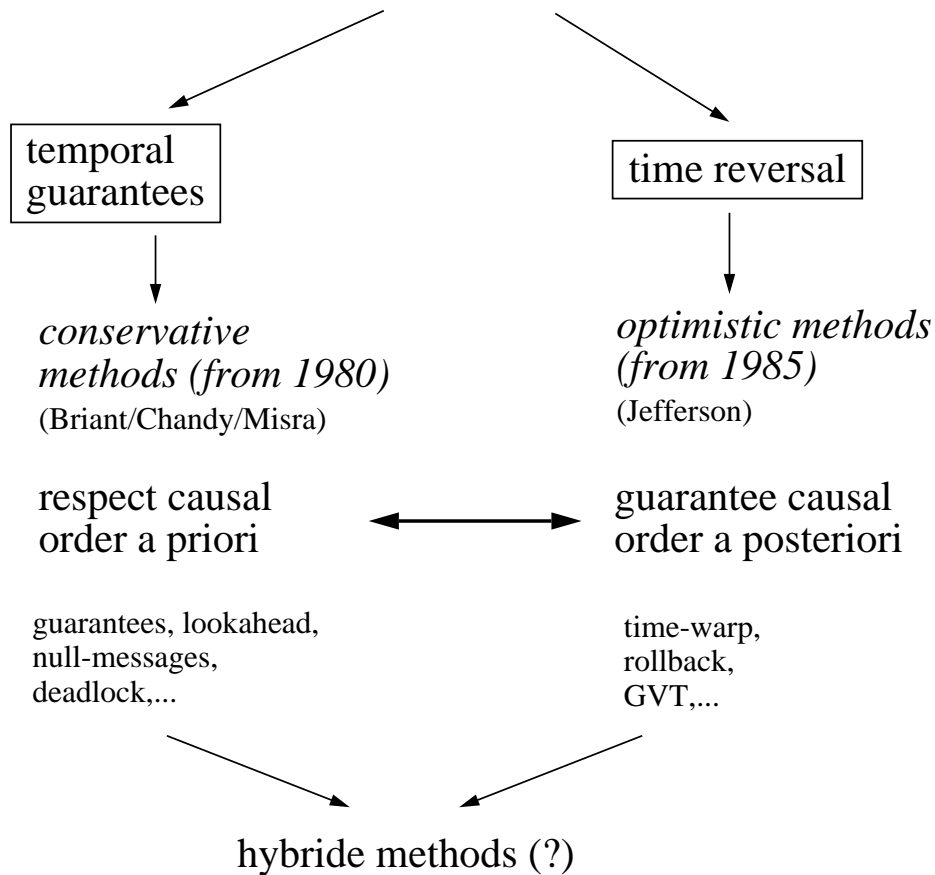
- Timestamp of messages \geq clock of sender.

- But: is timestamp of message $<$ clock of receiver possible?

- When may a simulator advance its local clock?

==> distributed simulation / synchronization schemes

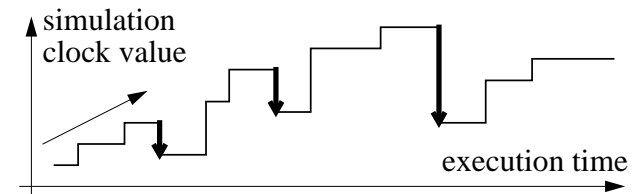
Distributed Simulation Schemes



- Availability of parallel computers --> increased research activities since 1985.
- Many variants of the basic schemes have been designed.
- Many publications on specific aspects.
- But until now no real breakthrough in general speedups.

Optimistic Simulation, Time-Warp

- Each simulator may advance its clock independently.
- If a message with a timestamp < local time of receiver is received: *Rollback*



Rollback:

- set receiver's clock back to timestamp of message
- restore an earlier state (saved checkpoint)
- possibly send out anti-messages

-->Many checkpoints!

-->When are checkpoints obsolete?

- no longer needed
- memory may be freed

Time-Warp - More Aspects

- Simulator may act on illegal local states
=> anything is possible!
- Storage space for saved events and states
=> incremental state saving?
- Overhead (--> speed-up?)
- Many variants, strategies, heuristics..., e.g.:
 - broadcast “all my messages after T=x are invalid” instead of dedicated anti messages
 - lazy cancellation
 - time windows
 - adaptive strategies
 - cancel back

Global Virtual Time (GVT)

$$\text{GVT}(\tau) = \min_i \text{CLOCK}_i(\tau)$$

↑ execution time instant

Minimum of all clocks
(ignore message time-stamps for synchronous communications)

Function of the global state!

- Applications:
 - no rollbacks beyond GVT
 - older checkpoints may be removed
 - unrecoverable output operations may be committed
 - detect end of simulation time period

- GVT approximation:
 - GVT(τ) increases monotonically
 - tight *lower bound* \leq GVT(τ) necessary
↳ “current” GVT value is meaningless

- Modelling of the underlying distributed computation by two types of *atomic actions*:

$$\mathbf{I}_i: \text{CLOCK}_i := \text{CLOCK}_i + d \quad (d > 0)$$

internal action of process i

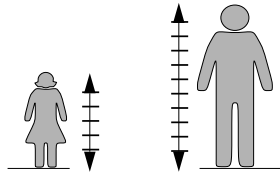
$$\mathbf{X}_{ij}: \text{if } \text{CLOCK}_i < \text{CLOCK}_j \text{ then } \text{CLOCK}_j := \text{CLOCK}_i$$

synchronous remote event scheduling action

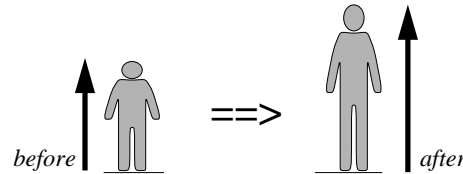
(simplified: message timestamp = sender’s clock)

An Illustration of the GVT Approximation Problem

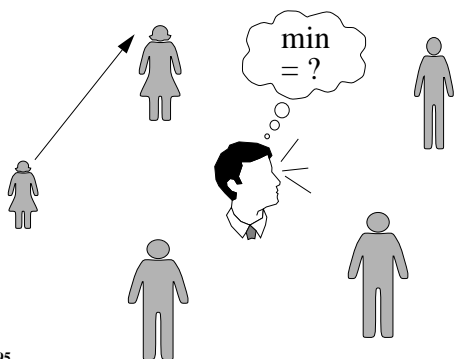
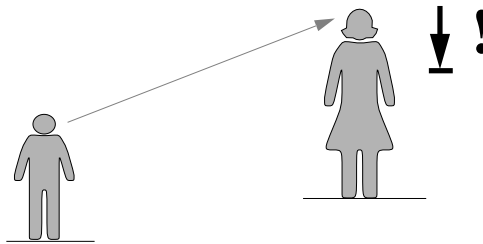
- Each person has a certain height:



- A person may grow spontaneously:



- A person may wink with his eyes at another person --> the other person is reduced to the height of the winking person.



“Axiom” of distributed computing

- Observer has no global view
- Fooling the observer by "behind the back" winking

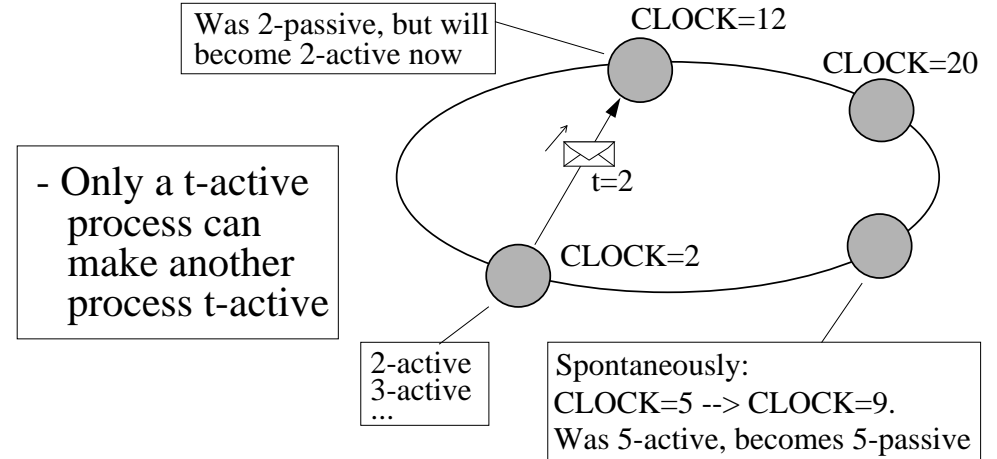
GVT Approximation with Termination Detection Algorithms

Idea: termination detection is binary version of GVT approximation

e.g., 0 and ∞

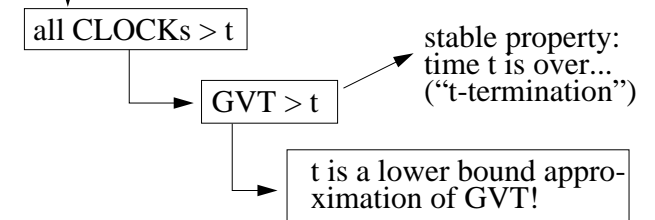
- Fix a *threshold value* $t \in \mathbf{R}$

- Call a process *t-active* if its $\text{CLOCK} \leq t$ (*t-passive* otherwise)



- Only a *t-active* process can make another process *t-active*

- Detect: "no process is *t-active*"



Termination detection problem!

t-Termination as a Bound for GVT

Idea:

- Many termination detection algorithms run in parallel.
- Each algorithm determines a specific lower bound.
- All algorithms are combined into a single algorithm.
(Instead of a single message: transmit a whole *bundle* of messages)

Example: 3 termination detection algorithms with $t_1=5$, $t_2=10$, $t_3=100$ are executed in parallel.
Return $\max t_i$ of those which reported t-termination.

NB: Lower bound is a stable (and hence observer independent) predicate.

==> Why not use a snapshot algorithm?

This is possible. However, it turns out that consistent cuts are not required - inconsistent cuts will also work! Hence, snapshot algorithms are perhaps too “heavy” for that problem!

Speedup ?

- Mapping of simulation objects onto processors
 - minimize communication (remote event scheduling)
 - balance the load (is never perfect!)
 - Message transmission overhead
 - Synchronization overhead
 - No global view --> unavoidable waiting conditions
 - Causal dependencies among events
 - Partitioning the model needs time
-

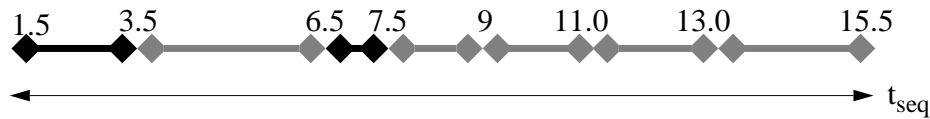
==> Limits the attainable speedup!

Faithful speedup measurements:

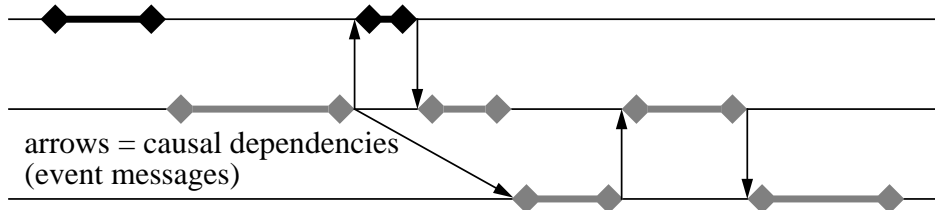
Parallel simulator should be compared to true sequential simulator
(not to the parallel simulator running on a single processor!)

Critical Path Speedup

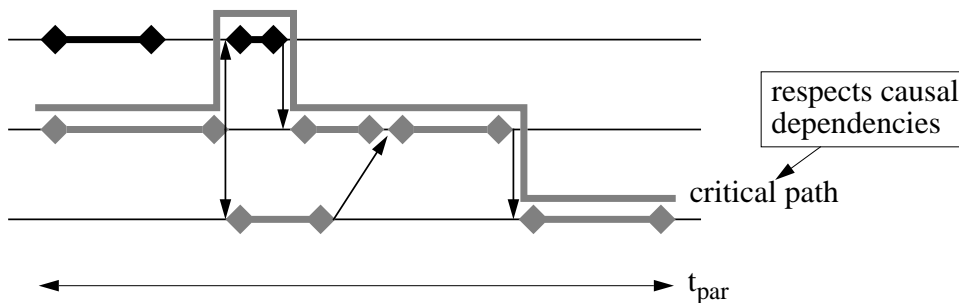
Sequential simulation --> measure the duration of events:



“Distributed sequential” simulation:



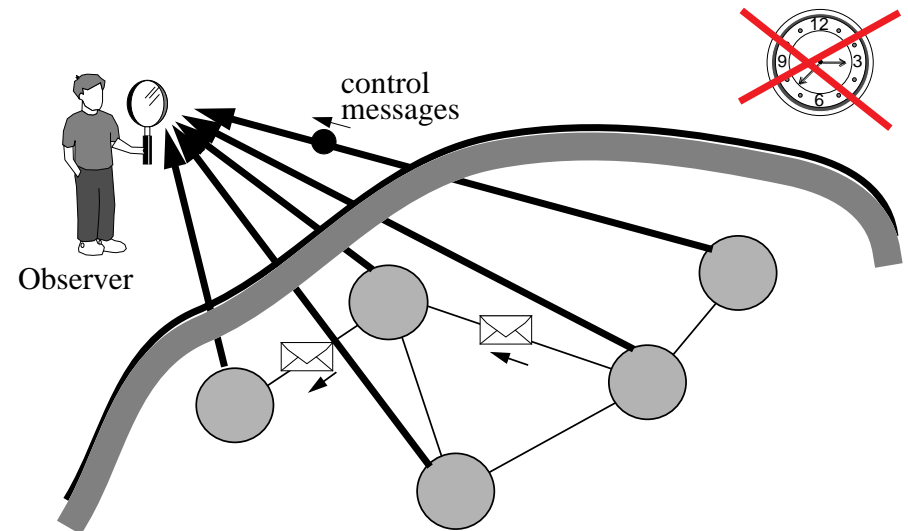
“Optimal” distributed simulation: Push everything as far to the left as possible



$$\text{speedup} = \frac{t_{\text{seq}}}{t_{\text{par}}}$$

Calculated speedup is much too optimistic: It abstracts from communication overhead, from wait conditions, from control overhead...

Observing Distributed Computations



- Observation is only possible via *control messages* (with undetermined transmission times)

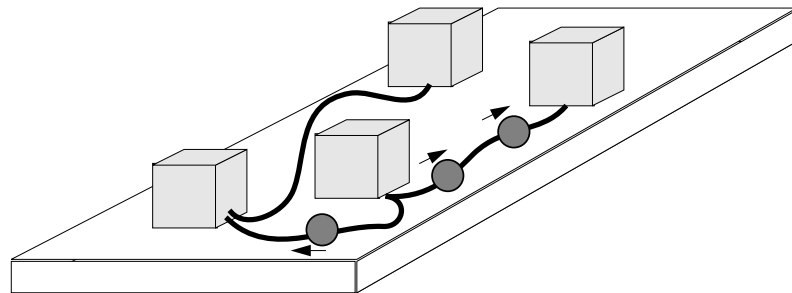
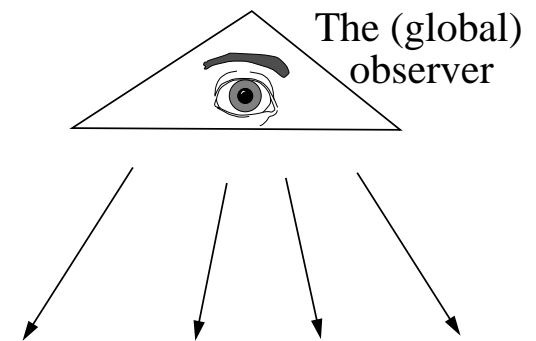
"Axiom": Several processes can "never" be observed simultaneously

"Corollary": Statements about the global state are difficult

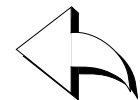
Consequences for monitoring, debugging...?

Observation

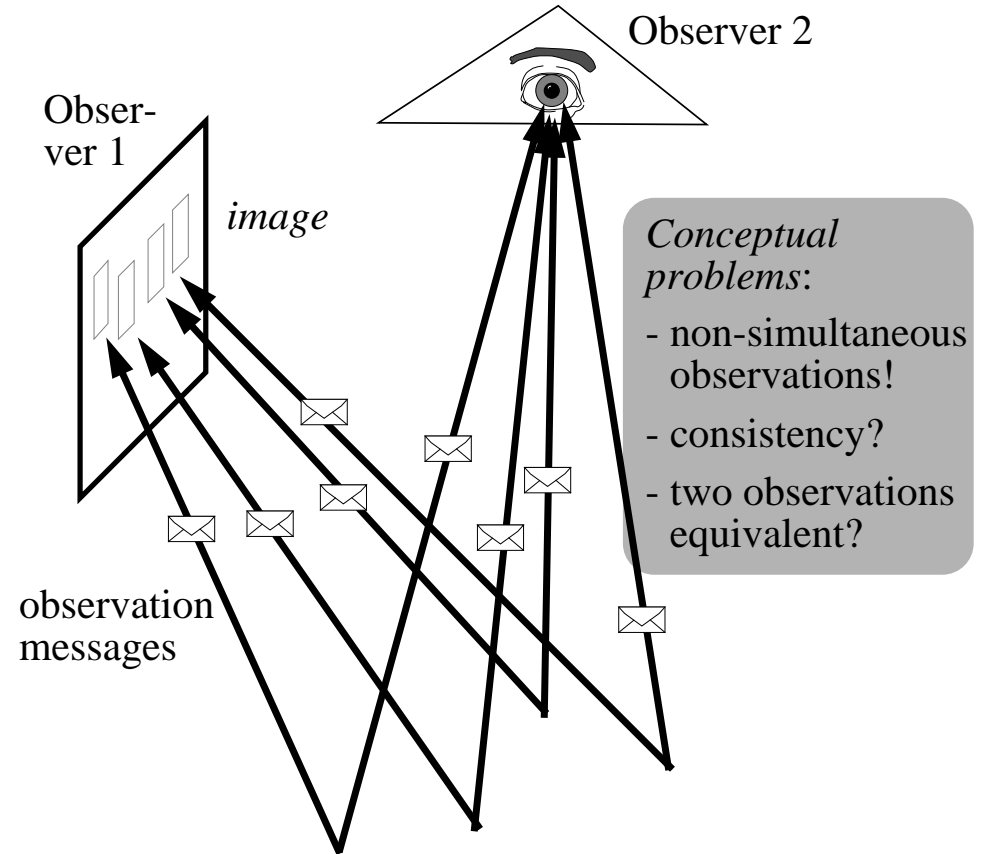
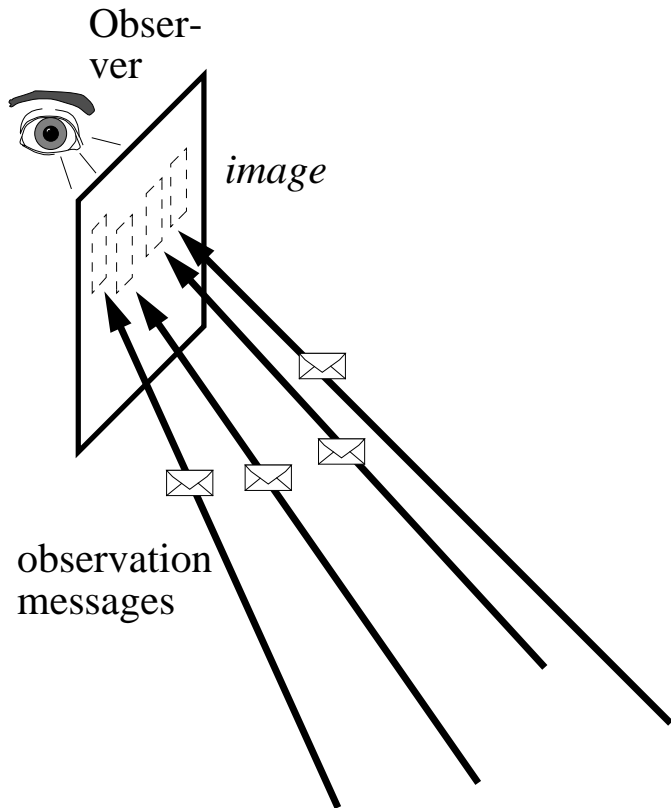
Idealistic view:
global perspective



The real computation

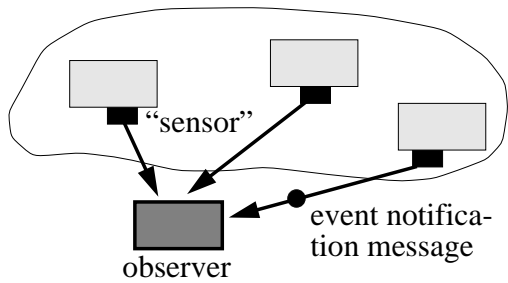


*The object to
be observer*



- Technical problems:*
- instrumentation
 - intrusiveness
 - ...

External Observation

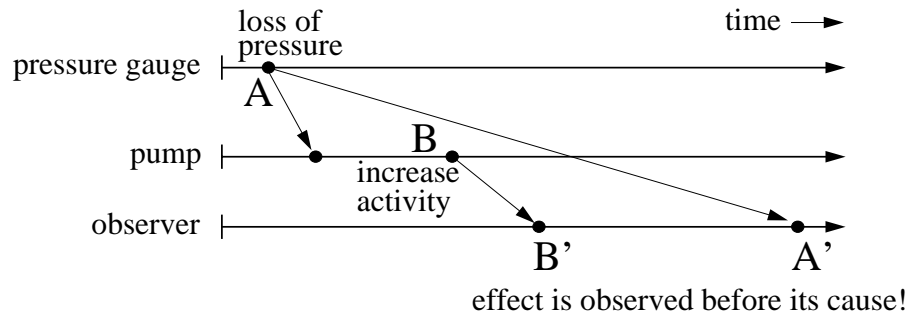
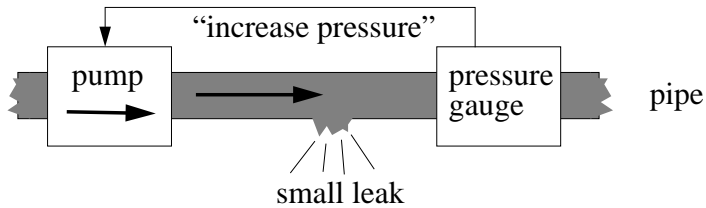


- Debugging
- Monitoring
- Visualization
- Performance analysis

“Internal” Observation

processes within the computation must have a causally consistent view

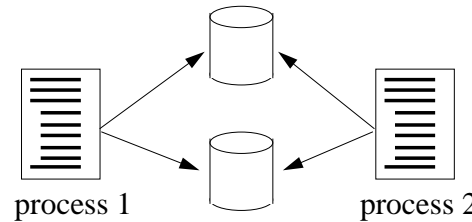
- *Protocols and algorithms*
- Deadlock detection, termination detection...
- Replicated servers (broadcast / multicast protocols)



Wrong conclusion of the observer:

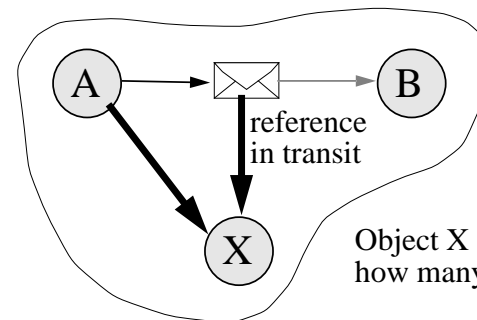
An unmotivated activity by the pump (led to increased pressure and the occurrence of a leak, which) resulted in a loss of pressure

Problem: Realization of causally consistent observers



- causality preserving
- “observations” of the disks should be equivalent (=?)

- *Example: Distributed garbage collection*



Object X must have a consistent “view” of how many references are pointing towards it

Monitoring and Visualization

Capture useful data during execution (for later use...)

Provide an adequate image
Present monitoring data
Snapshot <--> animation

- Application of observation techniques

Motivation

- Parallel and distributed programs are *complex* systems

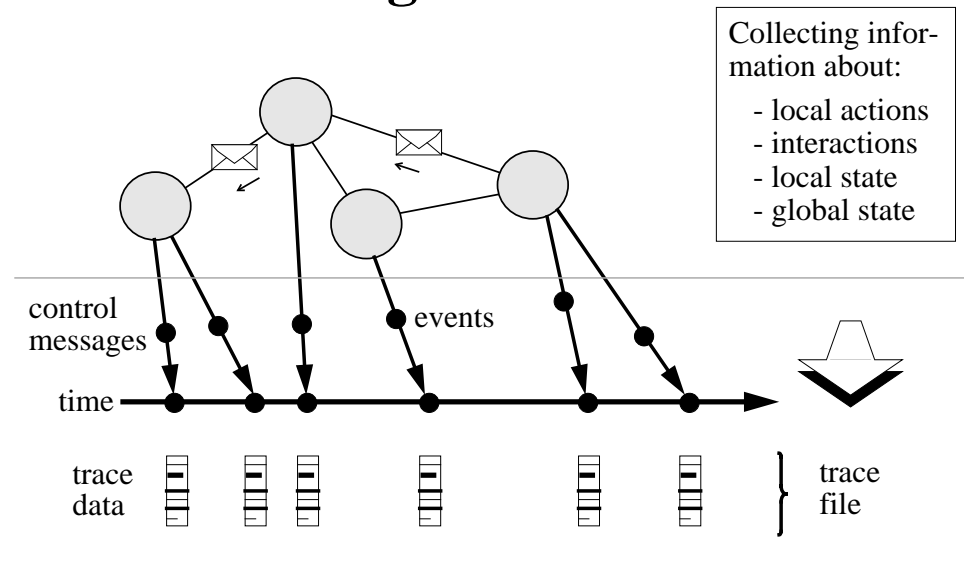
- > difficult to understand
- > error prone
- > difficult to verify

- no central control
- no global time and state
- inherently non-deterministic
- many threads of control
- interaction / synchronization

Purpose

- Knowing (exactly) what is going on...
 - > gain insights, understand complex phenomena
 - > debugging, testing
 - > performance evaluation --> optimization
 - > fault and security management
 - > trend analysis

Monitoring



- *Event-driven* monitoring

- only actions of interest generate information

- *Time-driven* monitoring

- status information is obtained periodically
- sampling rate?
- consistency? (synchronized clocks?)
- information overflow?

Events

What is an event?

Any *atomic* action which significantly affects the *local state* of a process

- sending / receiving a message
- entering / leaving a procedure
- executing a statement / a machine operation
- changing the value of a variable
- ...

What information is associated to an event?

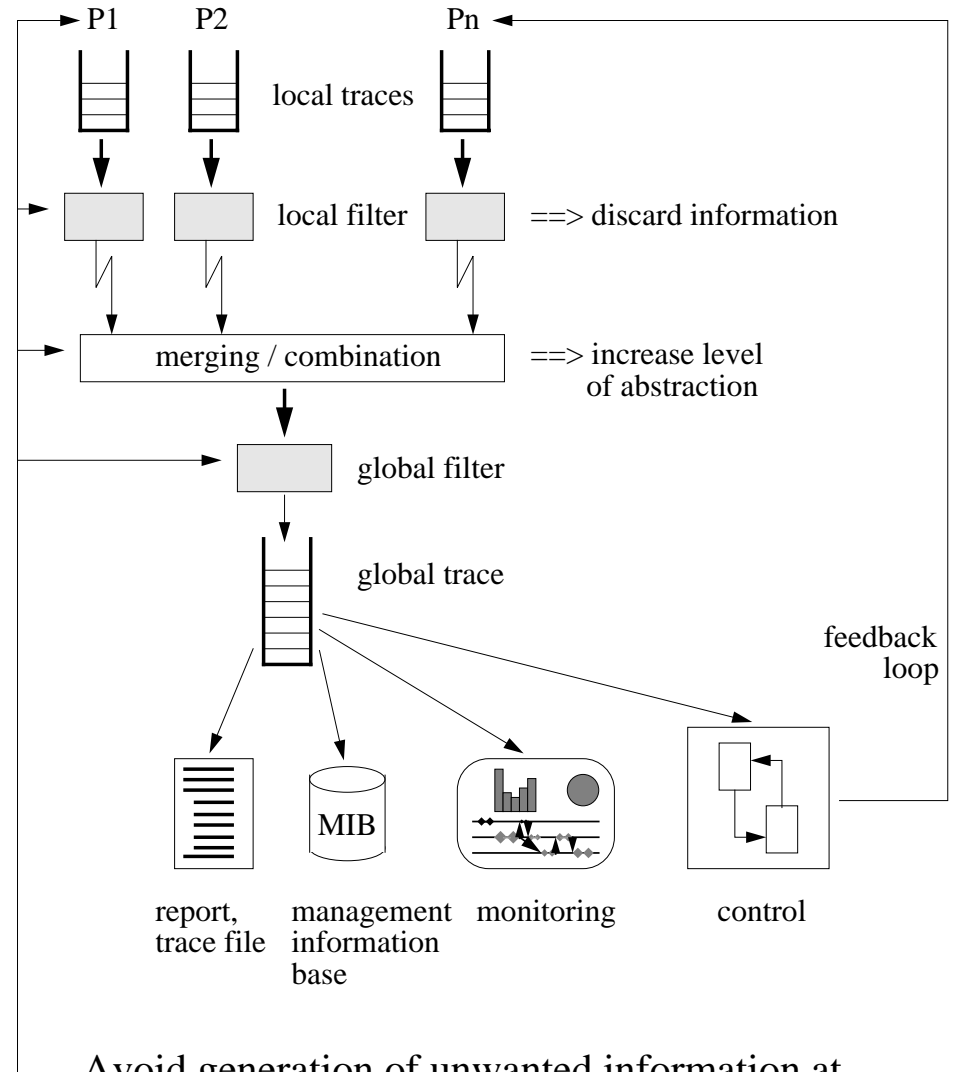
- its type (e.g., “enter procedure”)
- its time of occurrence
- parameters and attributes (e.g., line number)
- ... the whole local state of a process / processor

--> complete information!

Combined events

- grouping of primitive events or other combined events
- there exist various languages to specify combined events
- often: rather complex syntax and unclear semantics; examples:
 - when does “e1 and e2” happen?
 - causal or temporal order in “e1 --> e2”?
 - is negation sensible?
- difficult to “detect”, because components can be located on different processors

Processing of Monitoring Information



Avoid generation of unwanted information at various levels (e.g., activate / deactivate filters)

The Intrusiveness Problem

- Effect of tracing / monitoring / debugging on the behavior of the monitored system
 - monitoring alters the timing of events
 - degrades system performance
 - may change the ordering of events
 - may lead to incorrect behavior / results
 - may mask errors of the unmonitored system
- ==> Result of monitoring is only an approximation of the unmonitored system!

Hardware and Software Monitors

- Hardware monitors
 - nonintrusive
 - physical sensors connected to system buses, processors, memory ports, I/O-channels...
 - typically high-speed comparators for simple bit patterns
 - disadvantages:
 - requires additional hardware
 - very low level
 - not portable
 - problems with caches, pipelining... on the chip
- Software monitors
 - manual or automatic insertion of “probes” into the source code (requires recompilation)
 - instrumented libraries (e.g., communication)
 - insertion into object code
 - instrumentation of the kernel (works for all programs, independent of language or compiler)

Visualization

Systems:

- Balsa II [M. Brown: algorithm animation]
- TOPSYS, VISTOP [Bemmerl (Munic)]
- TMON, TIPS [Univ. of British Columbia]
- SIMPLE, TDL/POET, VISIMON... [U. of Erlangen]
- Jade [Joyce et al.]
- Voyeur [Socha et al.]
- ParaGraph [Heath, Etheridge (Oak Ridge)]
- ...

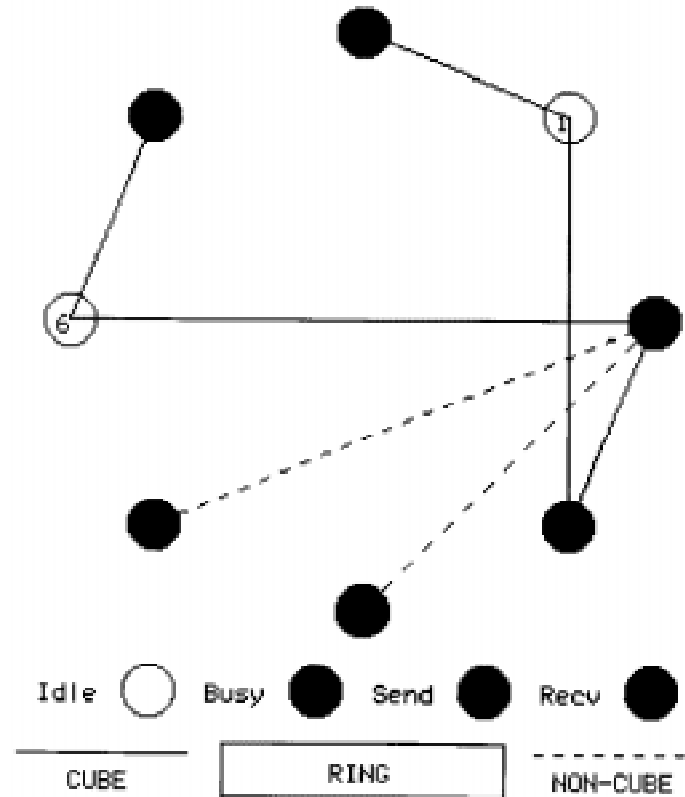


ParaGraph [Heath, Etheridge (Oak Ridge)]

- Trace based graph. display system (portable, available)
- Several different perspectives (color, animation)

Animation ==> Sequence of global snapshots

- Status of each node (idle, active,...)
- Paradigm: “front panel lights” of the system



Consistent view?

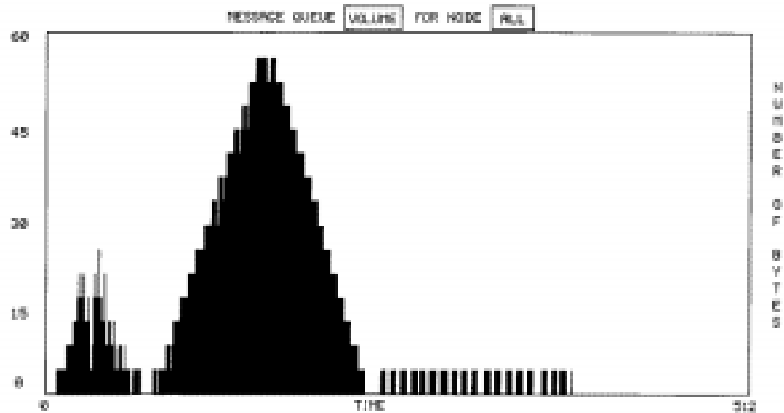
-->

(sufficiently well) synchronized local clocks?

timestamped events!

Message queues

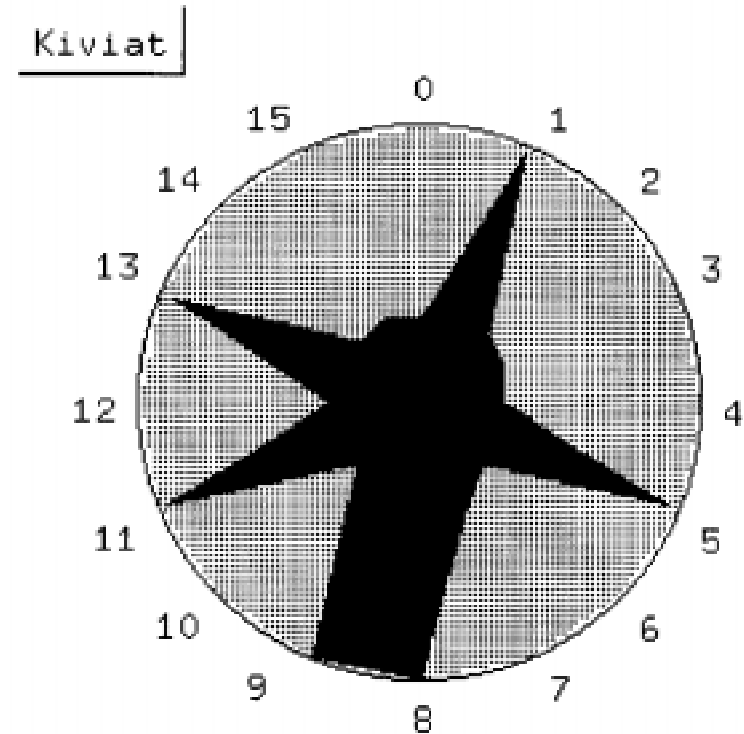
- Number of messages, number of bytes vs. time



--> global time?
(or approximation of global time?)

Kiviat profile

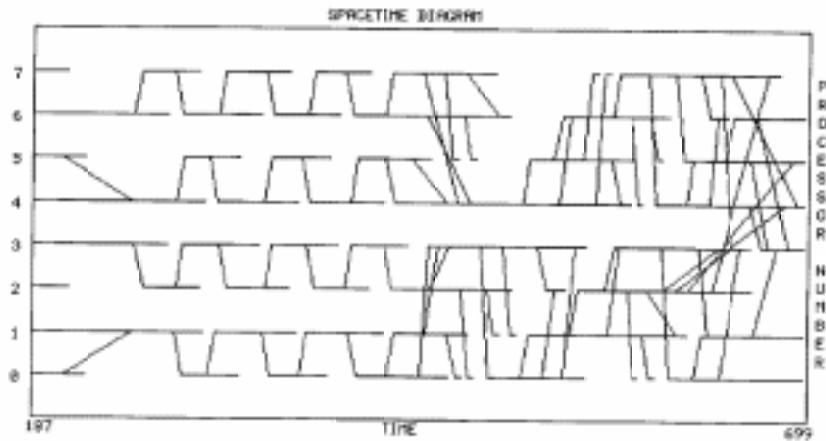
- Recent average fractional utilization of processors
- Each processor represented by a spoke of a wheel
- Size and shape indicate overall load balance



- Is the "snapshot" consistent?
 - > "wrong termination detection" phenomenon would wrongly yield "load 0" for all processors!

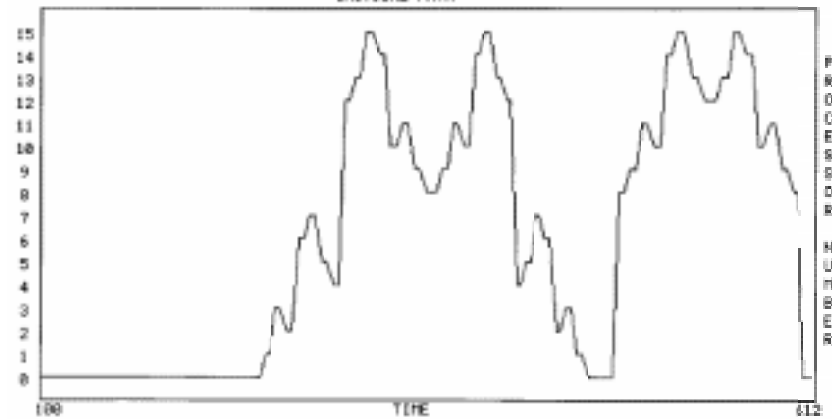
Spacetime diagram

- Processor activity (active/idle) on horizontal lines
- Full detail of message activity (slanted lines)
- Messages “reactivate” idle processors



Critical path

- Longest serial thread (--> limiting performance)
- Identification of bottlenecks

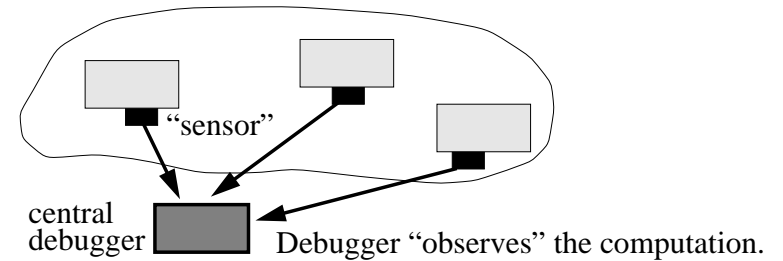


Monitoring and Visualization: Problems

- *Observation problems*
 - Variable message delays
 - Maintaining causality
- *Online / offline?*
 - Real-time monitoring <--> Post mortem analysis
- *Scalability?* (--> massively parallel systems)
- *Volume of data* -->
 - Selective views, abstractions
 - Hierarchies, clustering
 - Filtering, zooming
- *Pragmatics:*
 - Easy to manipulate, easy to understand pictures
 - Layout of items in pictures (problem specific?)
 - Multiple views
 - Standards for graphics and trace data (tool interaction)
- *Technical aspects:*
 - Intrusiveness, probe effect, perturbation, overhead
 - Timestamps, clock synchronization
 - Instrumentation (manual, automatic, code level)
 - Drawing speed, human perception speed
 - Network bandwidth, storage capacity
 - Architecture of event collection

Execution Replay may help with some of the technical problems

Another Application: Debugging



Main focus of a *distributed* debugger:

- Interaction among processes
- Global properties

Use a sequential debugger for purely local errors

Problems:

- Global state is distributed
- No unique time frame
- Error latency (too late when reported...)

Execution Replay helps: (observation of the original run!)

- Reproducing the computation (--> "heisenbug")
- Halting immediately (sequential execution!)

More serious *conceptual* problems: ← Confusion: often not well understood!

- What is a single step? (Next event is not unique!)
- Can we detect global breakpoints? (NB: global halt state is consistent!)
- Observation must be "causally consistent"
- Observations are not unique! ← Relativistic effects

Relativistic effects

Commercial Multiprocess Debugger

TotalView™ multi- process debugger

"... the best debugger in the world."
— Dan Gledhill, Chief Engineer, Advertiser Accounts, Inc.



TotalView™ is a source-level multiprocess debugger in the field, easy, affordable, true-remote for office programmers.

If you write software for a living, don't make any more of your time. Try the TotalView™ debugger. TotalView is a source-level debugger that lets you see all aspects of your application's operation — source, variables and pointers. And it lets you debug multiple processes simultaneously. Yes, programmers can debug both sides of a pipe, both sides of a socket, and both sides of an RPC, under the control of one debugger. TotalView even lets you debug applications that run on multiple workstations, making network distributed debugging a reality.

BBN knows that programmers don't like to wait, especially when they are bug hunting, so TotalView runs fast, single character keyboard accelerators and single-click mouse functions let you jump directly to where you want to be. TotalView is easy to learn, too. You don't have to take a

training course or wade through a manual, and you don't need to change your programming style either. Just write the best program you can and debug it with TotalView software. It's that simple.



BBN "TotalView" Multiprocess Debugger

- TotalView is easy to use. It has a point-and-click interface, online help, easy menus, requires no special make files, and imposes no restrictions on code or symbol table size.
- TotalView is fast. Try it. You have never seen X Windows move so fast.
- TotalView debugs multiple processes simultaneously. The software automatically attaches to newly created processes and follows your fork and exec calls.
- TotalView lets you debug network distributed applications. If you use PVM, RPC or sockets, you need TotalView.

- TotalView shows you what you need to see. Just click on a variable and see data or the underlying structure; click on a function name and see its source.
- TotalView provides source level debugging for C, C++, and FORTRAN.
- TotalView lets you debug at assembly and mixed source/ assembler levels.
- TotalView lets you write conditional breakpoints in C or FORTRAN; why learn "debugger language" when you already know how to program?

- TotalView is available on:
 - Sun SPARCstation running SunOS 4.1.x or Solaris 2.x.
 - Digital Equipment's Alpha running OSF/1.

And here's something your boss will like. TotalView software is affordable.

To find out how you can own "the best debugger in the world," call 1-800-422-2359 to request an evaluation copy of the TotalView multi-process debugger. Or send your request via electronic mail to tv@bbn.com.

BBN SYSTEMS AND TECHNOLOGIES

Execution Replay

- *Reconstruct* the original computation
 - Same initial state --> same “external behavior”
- Computations are usually *non-deterministic*
 - > During the original run of the program:
capture relevant information in a *log-file*
- > Replay using the log-file to direct the scheduler
(e.g., deliver the “right” message to the process)

= ?

- non-deterministic choices
- relative order of significant events

-
- Often, certain requirements are made:
 - Deterministic processes
 - No real-time dependent choices
 - No asynchronous interrupts
 - Usually not applicable to shared memory systems

--> overhead!

-
- Behavior is not changed if during replay:
 - Processes are *slowed down*
 - Processes are *stopped* and examined
 - Graphical *visualization* works in “slow motion”
 - Execution is *sequential* (“step by step”)

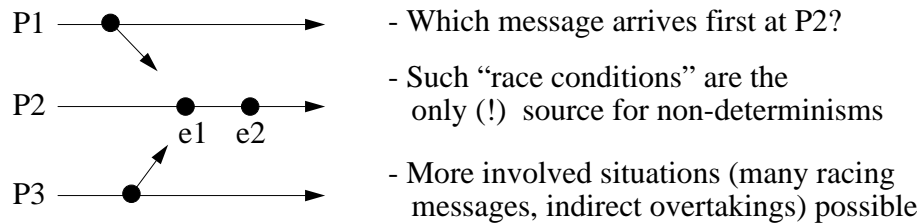
--> debugging!

Applications of Execution Replay

- Reproduce an erroneous run in “slow motion”.
 - add monitoring events
 - add print statement
 - slow motion of a single process
- Global single stepping of the run.
 - NB: next step is not unique!
- Halt immediately and examine the variables of a stopped state.
- Visualize the computation with appropriate speed.

} behavior remains unchanged

Nondeterministic Situations



- Idea:

- During the *original run* P2 logs which message was received at e1 and e2.
- During *replay* P2 consults the log to receive the correct message.

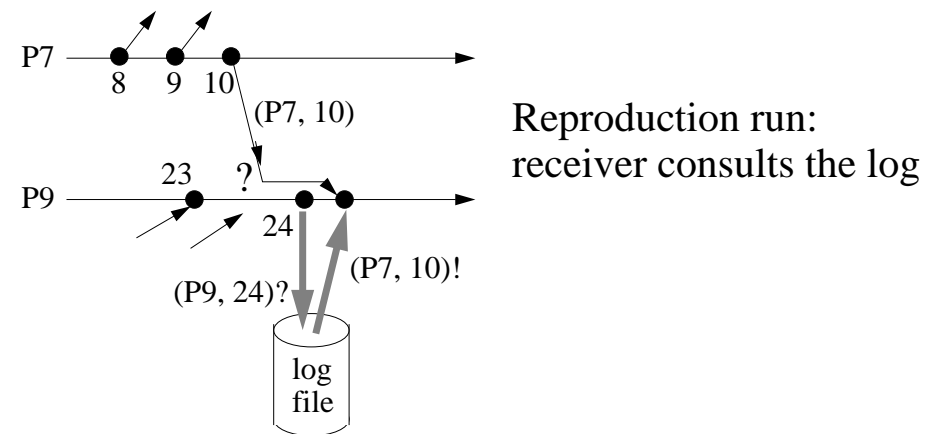
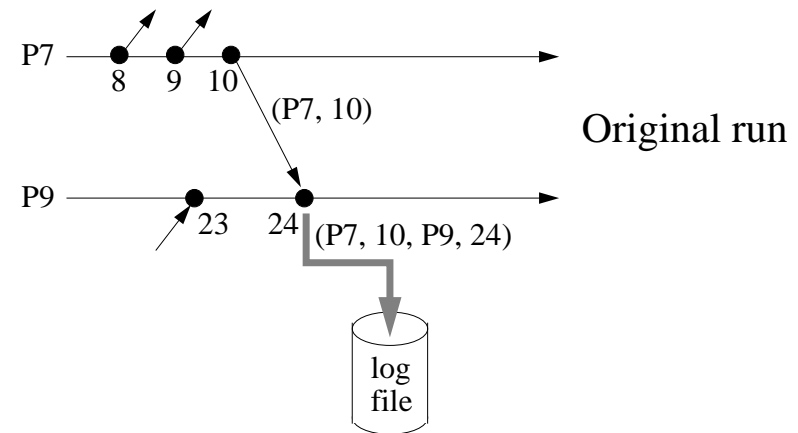
- Messages are uniquely identified by the tuple
(sender, event seq. number of sender, receiver, event seq. number of receiver)

- Only the order of messages is traced, not their contents (“control driven replay”)

- for non-reproducible environments (data input, clock readings etc.) the *contents* of messages must be logged (“data driven replay”)
- further problems: asynchronous interrupts
(expensive solution: register and trigger the instruction counter)

- Replay may start at the beginning or at a checkpoint (= consistent snapshot)

Receiver-Driven Reproduction

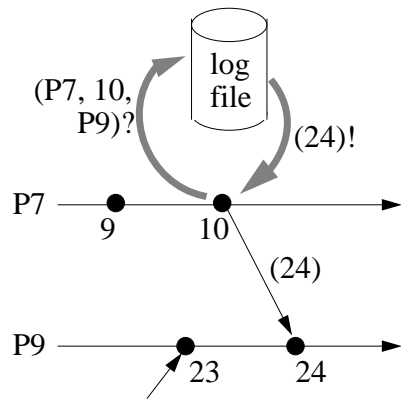


- Is it possible to reduce the log information?

- “P9, 24” is of course unnecessary if each process has its own log file
- but: are further reductions possible?

- Is it possible to omit the message tags?

Sender-Driven Reproduction

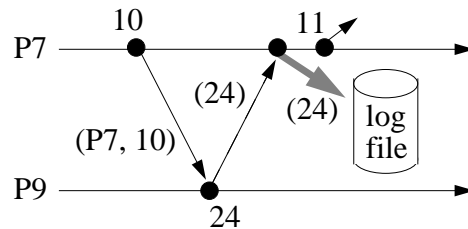


Reproduction run:
 Sender consults the log
 - the key “(P7, 10, P9)” is redundant
 - “(24)” is sufficient for the msg tag

Receiver counts receive events and accepts the message which matches the next receive number

- But how does the sender know the correct event sequence number of the receiver?

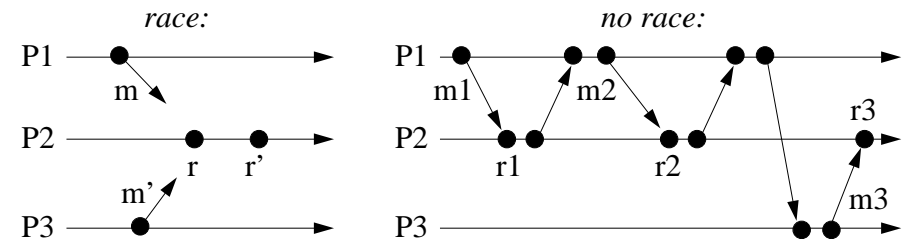
1) Receiver told the sender during the original run



2) Receiver put the information (P7, 10, P9, 24) in its local log file during the original run. All log files are merged, sorted according to the sender, and distributed to the relevant processes (after the run).

Determining Race Conditions

- Idea: Trace only those messages which form a race



- P2 should detect the race condition at r (“on the fly”) during the original run (m and m’ are “concurrent”)

- However, no race condition at receive events r1, r2, r3
 - second computation will be reproduced without further measures
 - messages m1, m2, m3 are “not concurrent” (--> single causal chain)
 - *race condition*: “locally previous receive event does not causally precede the send event of the message currently being received”

- Reduction of the log files

- for example: “accept next 3 messages without consulting the log”
 - or: tag racing messages, untagged messages can always be received

- Use vector timestamps during original run to determine whether two messages are concurrent or not

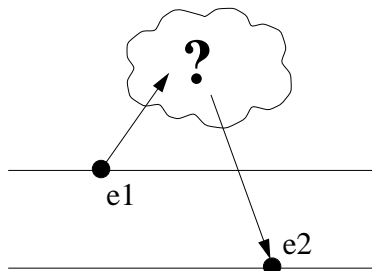
- whole vector is necessary (because of transitive relations)
 - pairwise comparison of two messages suffices for race determination

- For the details see the paper by Netzer and Miller

- claim: log files are typically reduced to 0 - 20 %, run-time overhead between 0 and 8 %

Further Aspects of Execution Replay

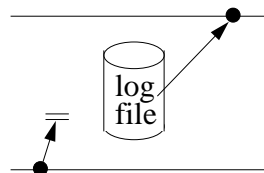
- Reproduction of dynamic systems
- Partial reproduction
 - replay of a subset only (e.g., a single process)
 - replay in an open environment



Problem: Hidden causal dependencies (may e2 be reproduced before e1 ?)

- Pure data-driven reproduction

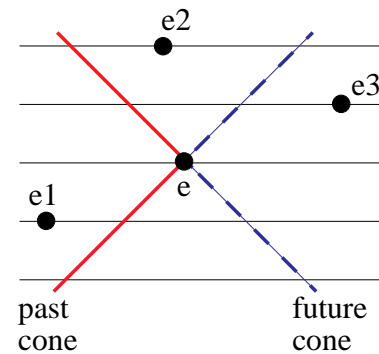
- all messages are received from a log file
- sending of messages is suppressed



==> during replay a message might be received before it is sent (possibly violating causality and causing strange effects)

Concepts Relevant to Distributed Debugging

- Global predicates
- Relativistic effects (multiple observers)
- Causality



- e1 (but not e2 or e3) could be the cause of e
- e potentially affects e3, but not e1 or e2
- realizable with vector time

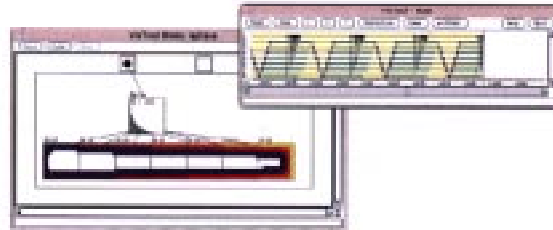
- Concurrent messages --> efficient replay
- Consistent snapshots --> checkpoints (“recovery lines”)
- Causally consistent observers
- ...

TRAPPER Graphical Design Tool for PVM

Visualization Tool

Software Analysis

The Visualization Tool graphically animates the sequential behaviour of each process and the interactions between the processes. Whether you want to see how the contents of a matrix develops in different processes, or whether the exchange of boundary data causes irregularities, or how scalar variables develop in different processes over time, TRAPPER gives you an insight how your application works and helps you to debug it.



Software Monitoring

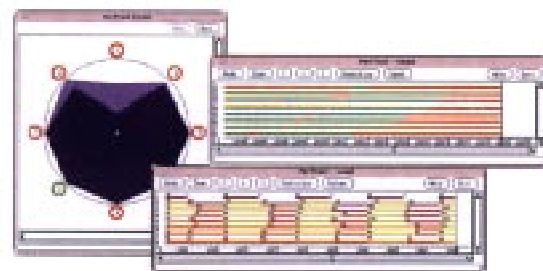
Every event collected by the TRAPPER monitoring system refers to the source code. This enables you to trace communication dead-locks and other programming errors down to source code level. Also, you can refer ill performing parts of your application very quickly to the corresponding lines in the source code.

TRAPPER Performance Tools

Performance Tool

Minimize execution time

TRAPPER has a built-in critical path analysis tool which helps you to find critical computation and communication times in your application. You are able to pinpoint commands in the source code, which are responsible for these critical time periods.

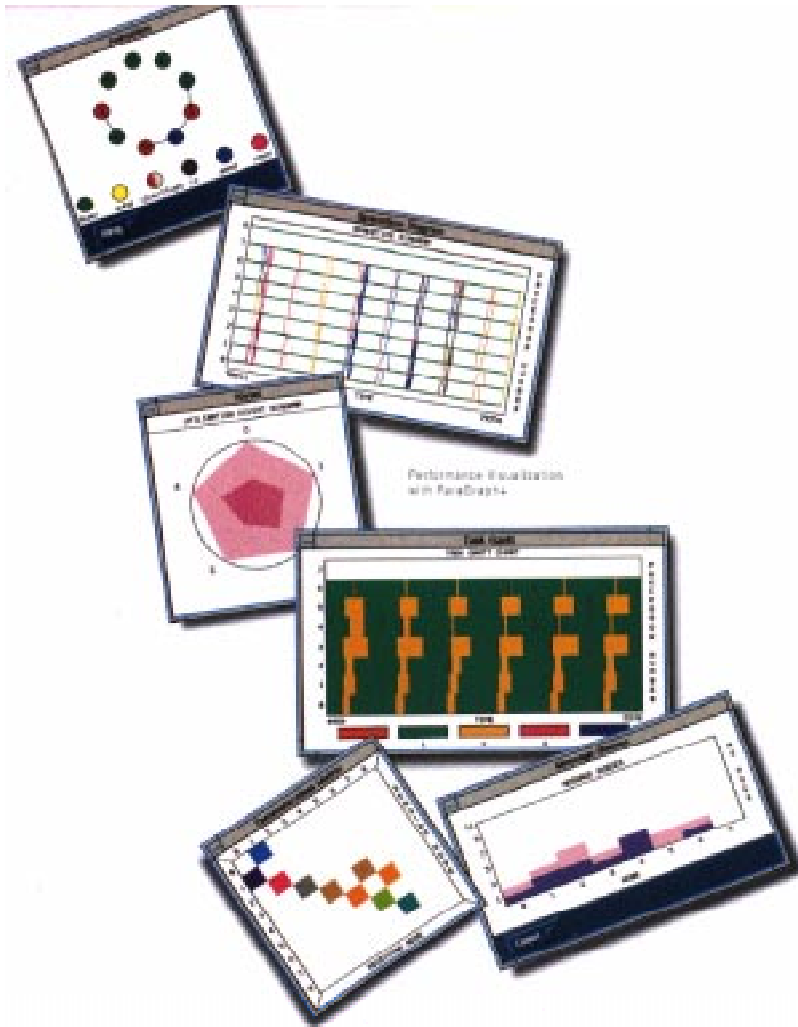


Statistics

For time diagrams, statistics can be generated over any time period. This includes hardware performance diagrams, process states diagrams and user data diagrams. Pressing a button provides you with information such as how much time processes spent waiting for messages or how much time they spent in a particular function.

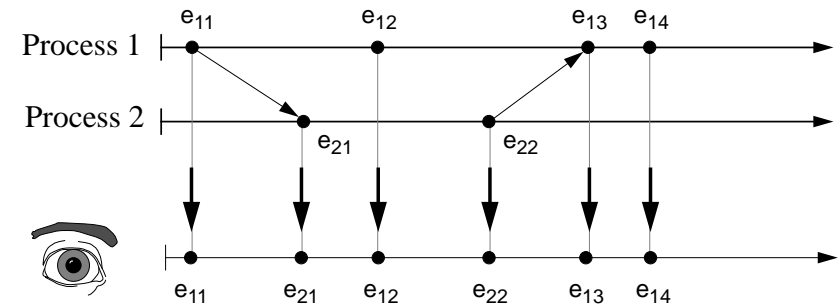


Paragraph+ by PALLAS

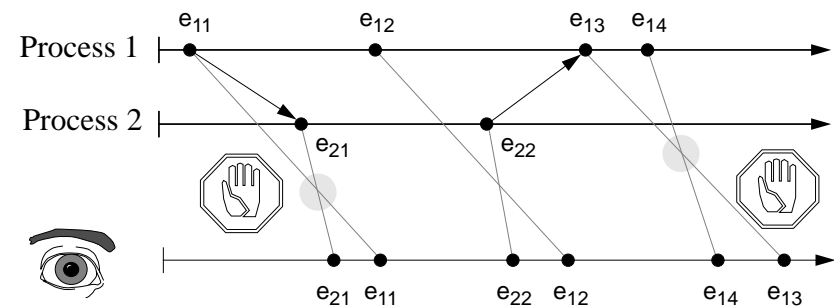


Valid and Invalid Observations

a) *Idealized observation* - instantaneous notification:
 (What we want but can't get)



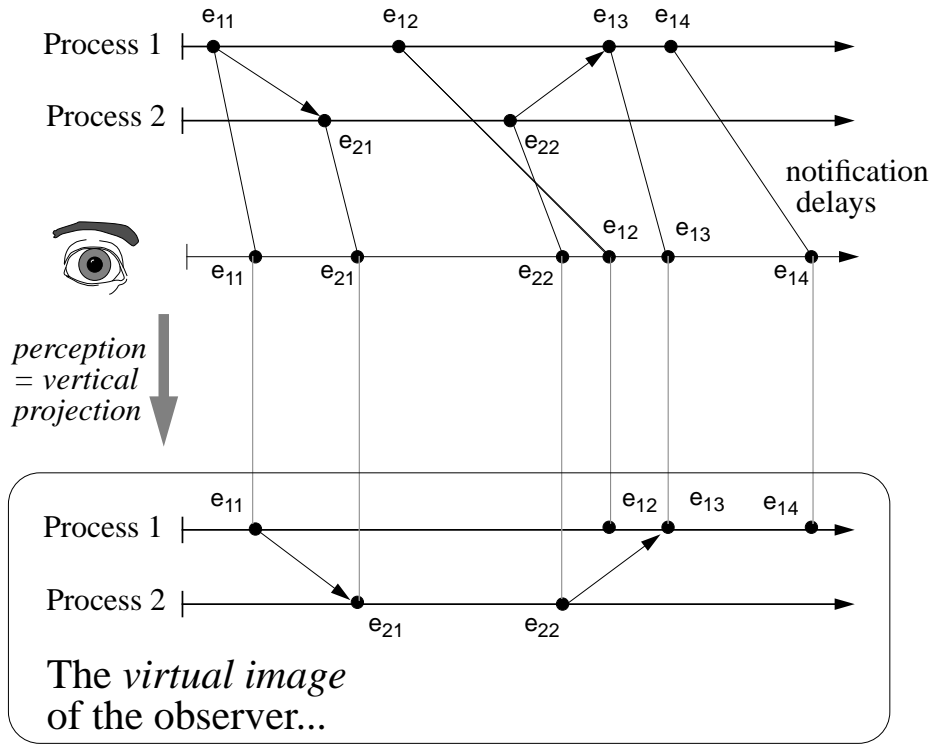
b) *Invalid observations* - violation of causality:
 (What we can get but don't want)



Effect is observed before its cause --> *inconsistent view!*
 - Also: *indirect* effect / causes

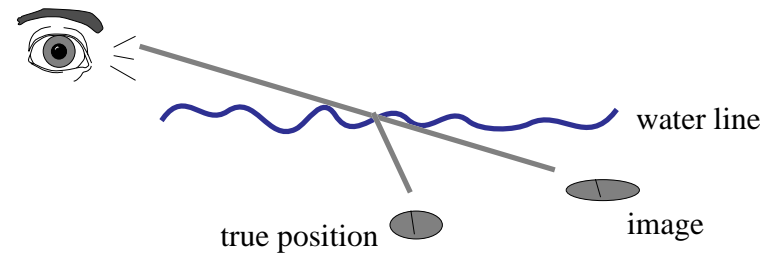
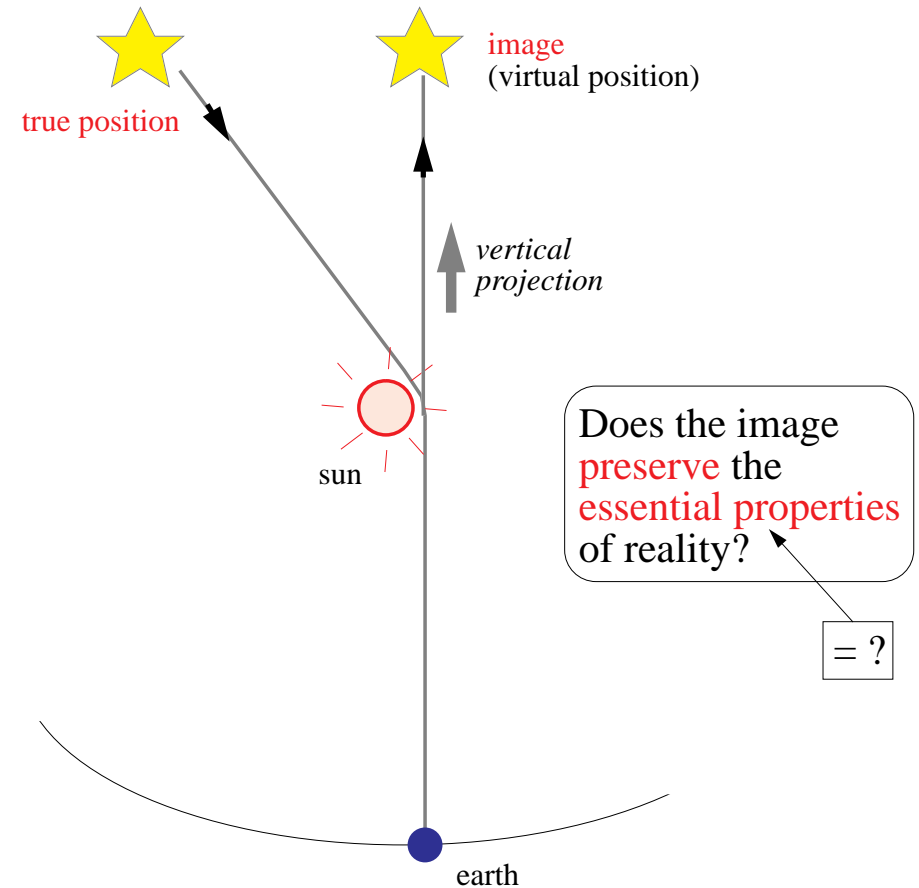
Valid Observations

- Cause always observed before its (possibly indirect) effect
(What we hope to get)



- Virtual image is a valid elastic deformation
- Annotations: "no message backwards in time" (with arrow pointing to the causal structure), "valid interpretation" (in a box with arrow pointing to the virtual image).

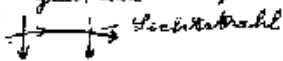
Image and Reality



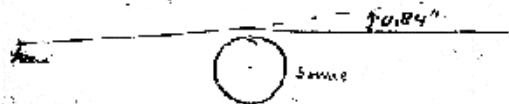
Zürich, 14. I. 13.

Hoch geehrter Herr Kollege!

Eine einfache theoretische Überlegung macht die Annahme plausibel, dass Lichtstrahlen in einem Gravitationsfeld eine Deviation erfahren.



Aus Sonnenstands müsste diese Ablenkung $0,84''$ betragen und wird $\frac{1}{2}$ abnehmen (R = Sonnenradius).



Es wäre deshalb von grösster Interesse, bis zu wie grosser Sonnen-nähe gross Fixsterne bei Anwendung der stärksten Vergrösserungen bei Tage (ohne Sonnenfinsternis) gesehen werden können.

Auf den Rat meines Kollegen, d. Herrn Prof. Maurer bitte ich Sie deshalb, mir mitzutheilen, was Sie nach Ihrer reichen Erfahrung in diesen Dingen für mit den heutigen Mitteln erreichbar halten.

Mit aller Hochachtung
Ihr ganz ergebener

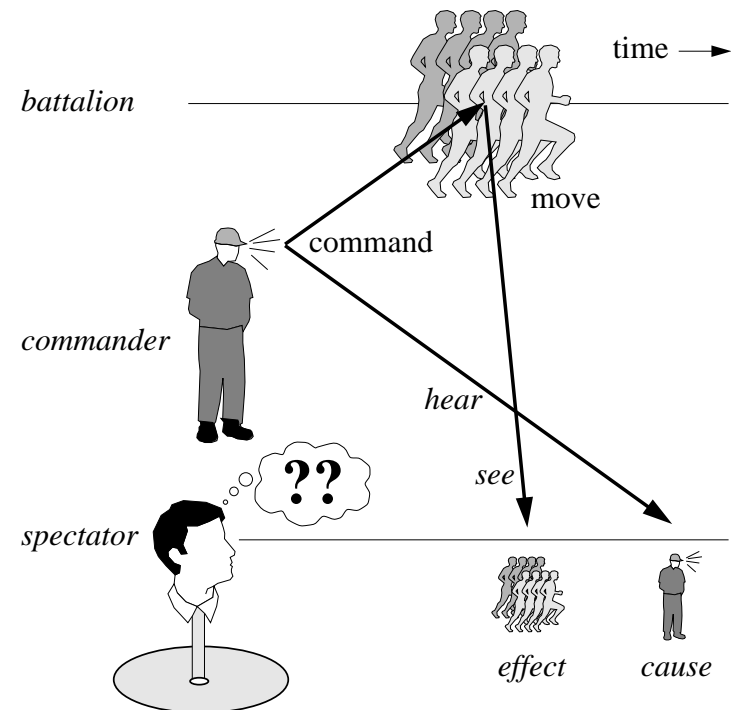
A. Einstein
Technische Hochschule
Zürich.

Causally Consistent Observations

The observation problem if not new...

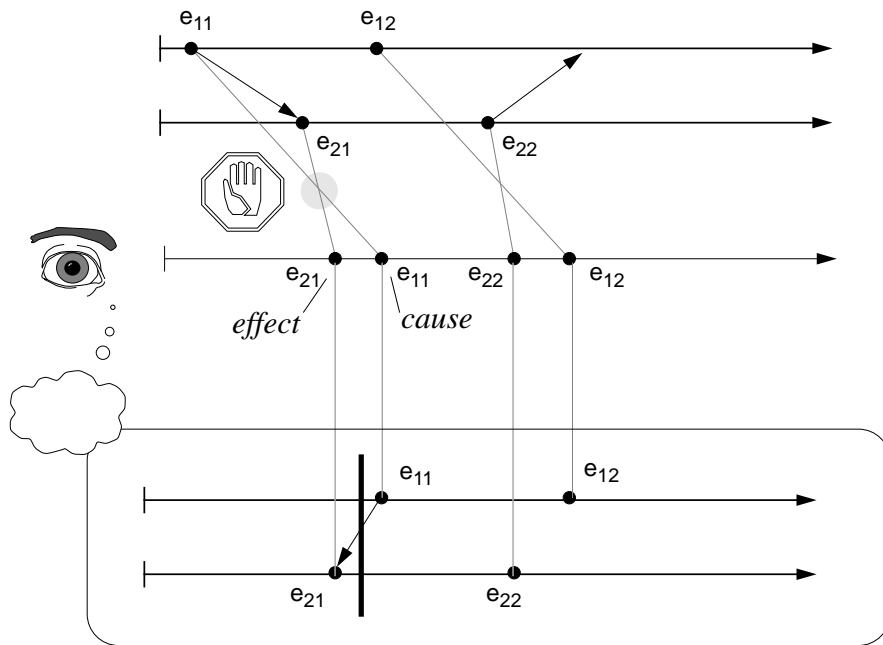
“When a spectator watches a battalion exercising from a distance he sees the men suddenly moving in concert before he hears the word of command or bugle-call, but from his knowledge of causal connections he is aware that the movements are the result of the command, hence that objectively the latter must have preceded the former.”

Christoph von Sigwart (1830-1904) Logic (1889)



Letter to George Hale,
Mount Wilson
Observatory, Pasadena

Images of Invalid Observations



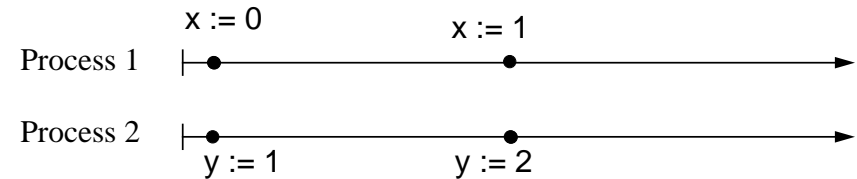
- Message goes backwards in time!
- The global state after e_{21} shows that a message is received which has not yet been sent!
- > Inconsistent cut / global state

Detecting Global Predicates

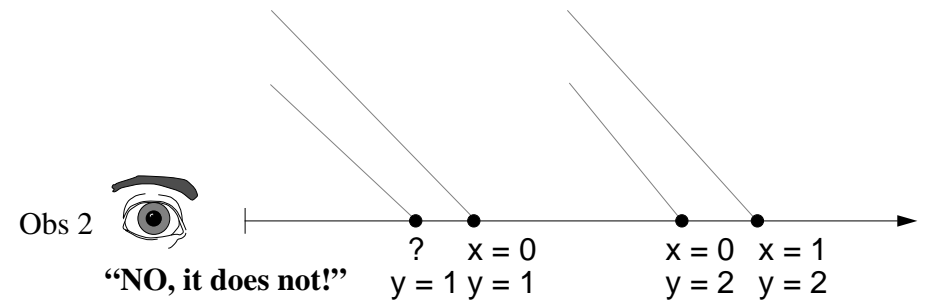
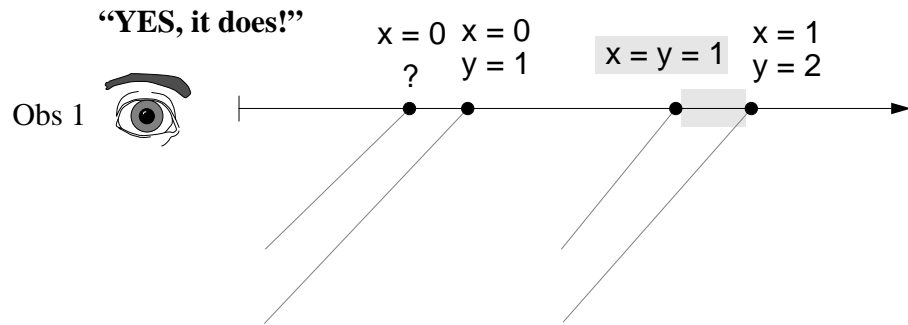
“properties”

Example:

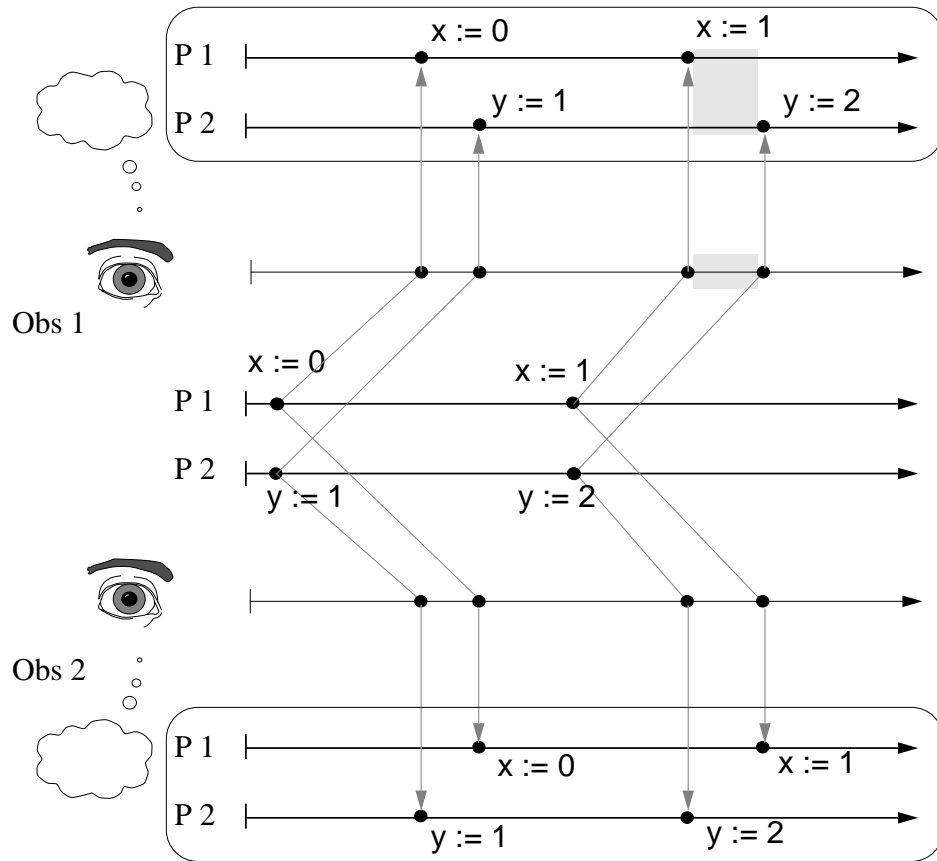
Does $(x=y)$ hold for the following computation?



- How can we *guarantee* causal consistency?



Reconstructing the Views



- Constant transmission speeds (slope)
- Both views are correct (i.e., consistent and equivalent)
- Both time diagrams represent the *same computation*
--> rubber band transformations

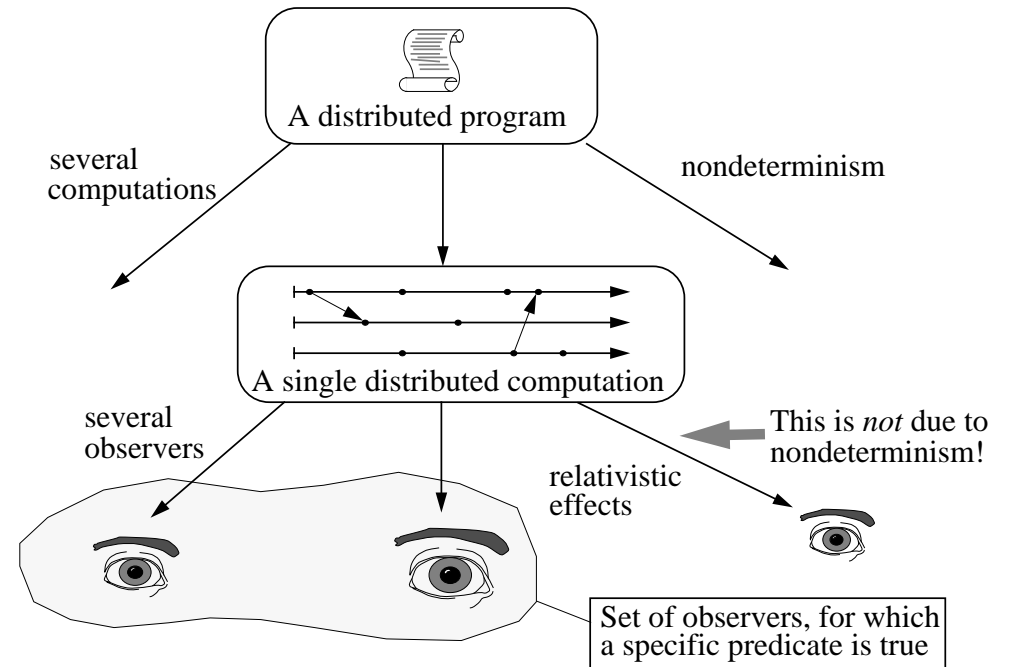
So what?

Do we have $x=y$ or $x \neq y$ for the computation?

Possible Worlds

- Different observers may see *different realities*.
--> Question, whether a specific predicate holds, might be *meaningless!*

No privileged observer



Consequences:

It is naive (i.e., wrong), to try to construct a distributed debugger which can answer such a question. (Which is a "good" question in the traditional sequential case!)

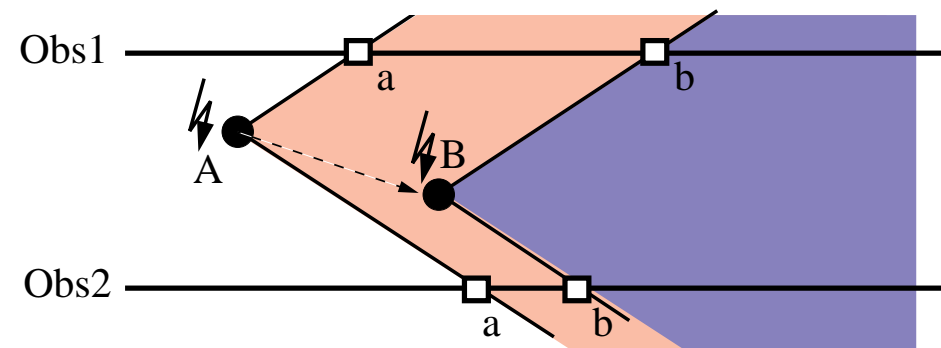
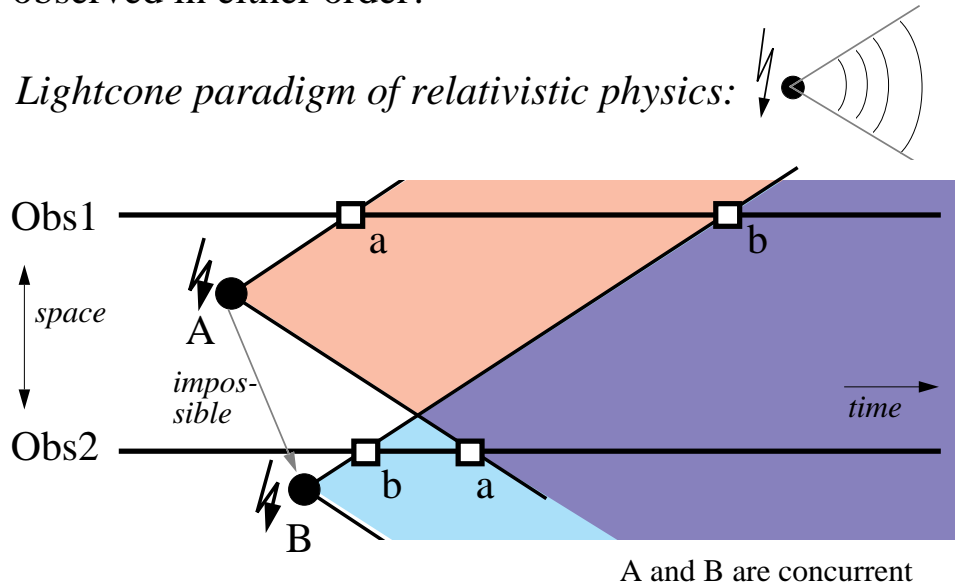
e.g., "stop when $x = y$ "

Reason:

Computation and observation is the same thing in the sequential case. But not for distributed systems!

Relativity of Simultaneity

Two “causally independent” events can be observed in either order!

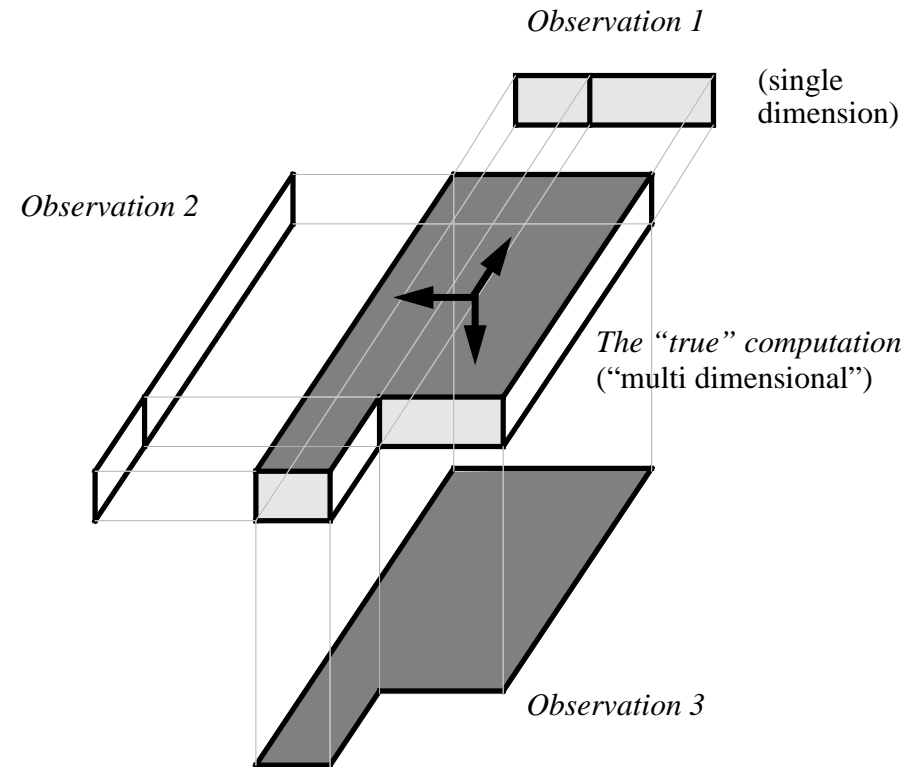


Observer independent
 \implies objective fact

B lies in the cone of A \implies
 B causally depends on A \implies
 All observers see B after A

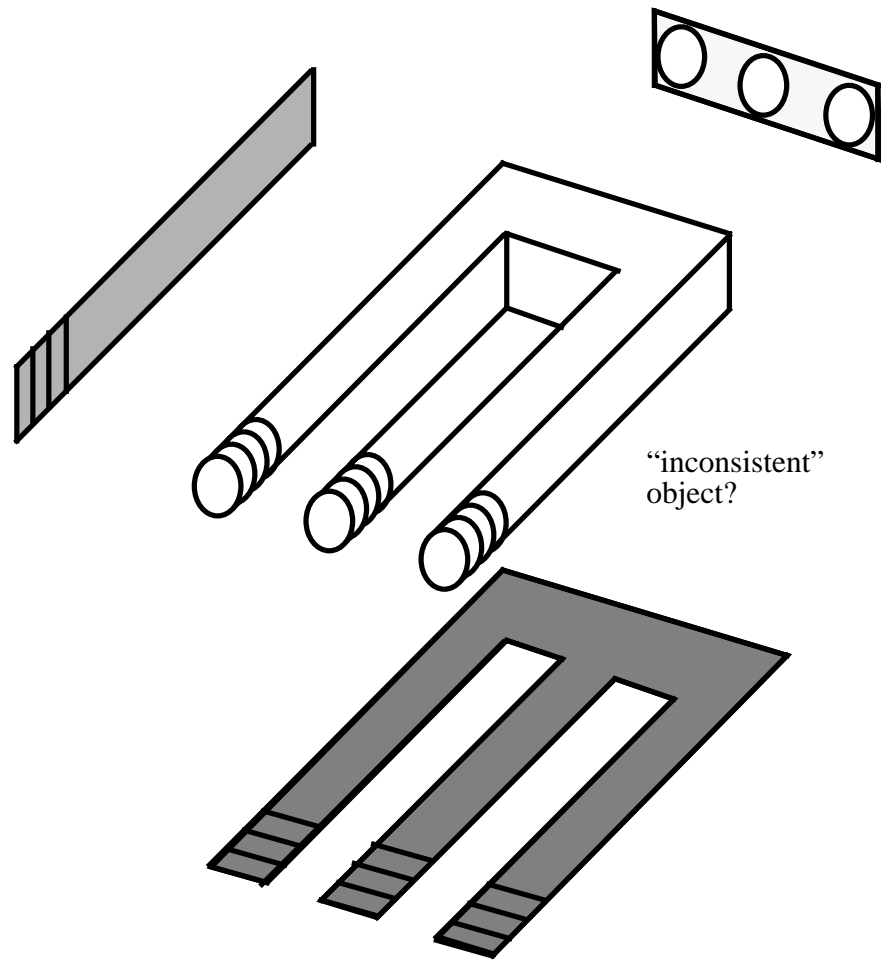
Observations, Images and Reality

- Each observation is necessarily incomplete!



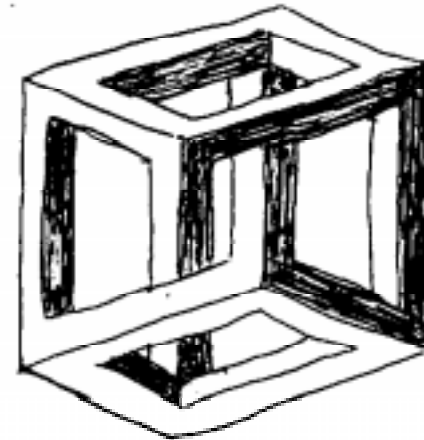
- Observation should preserve "essential properties"
- Some properties are lost, however \rightarrow in our case: causality
- Can we reconstruct the “real thing” from (all) observations?

Incoherent Observations

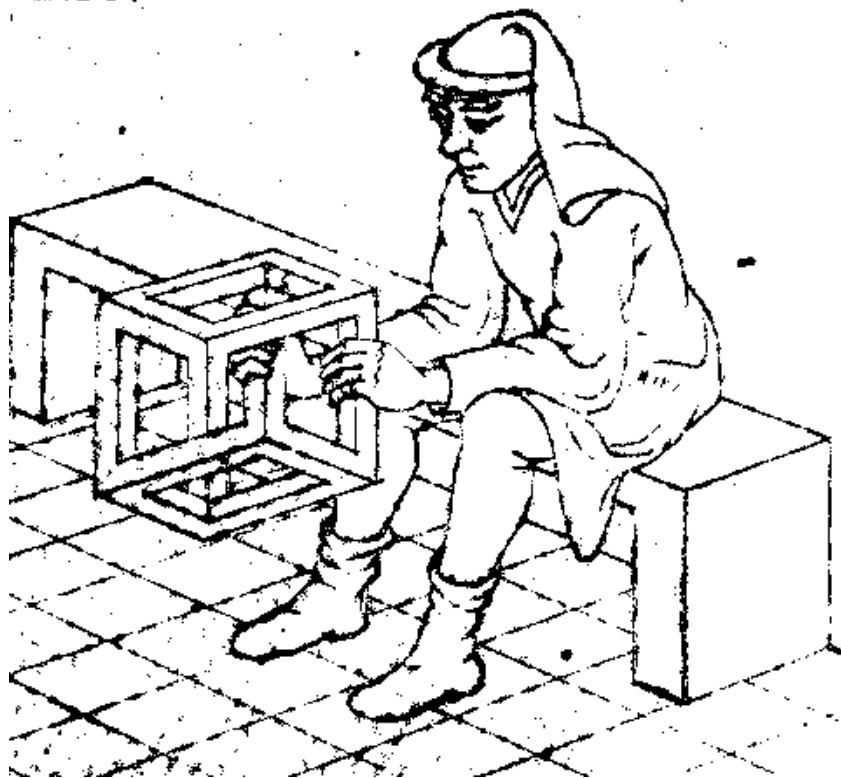


The observed object might be “in reality”
much stranger than we would expect!

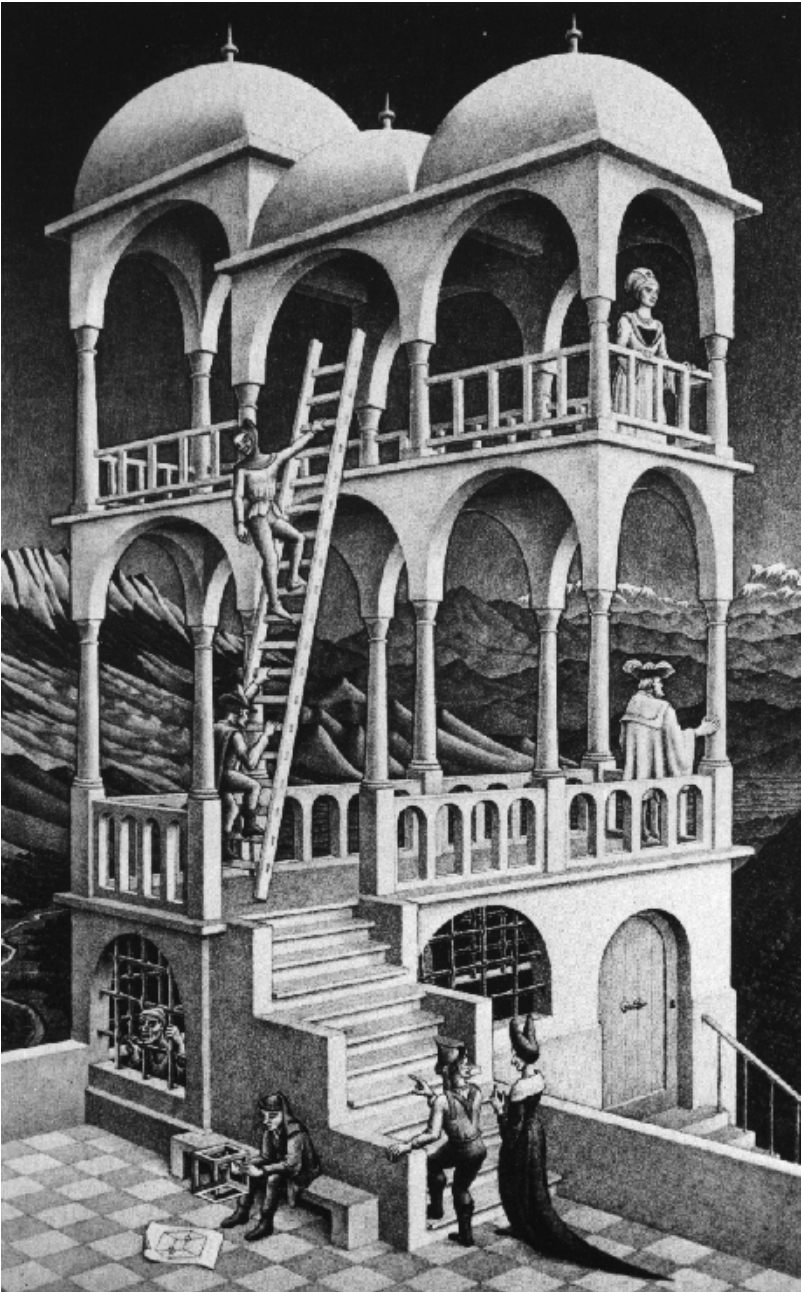
An Inconsistent Image



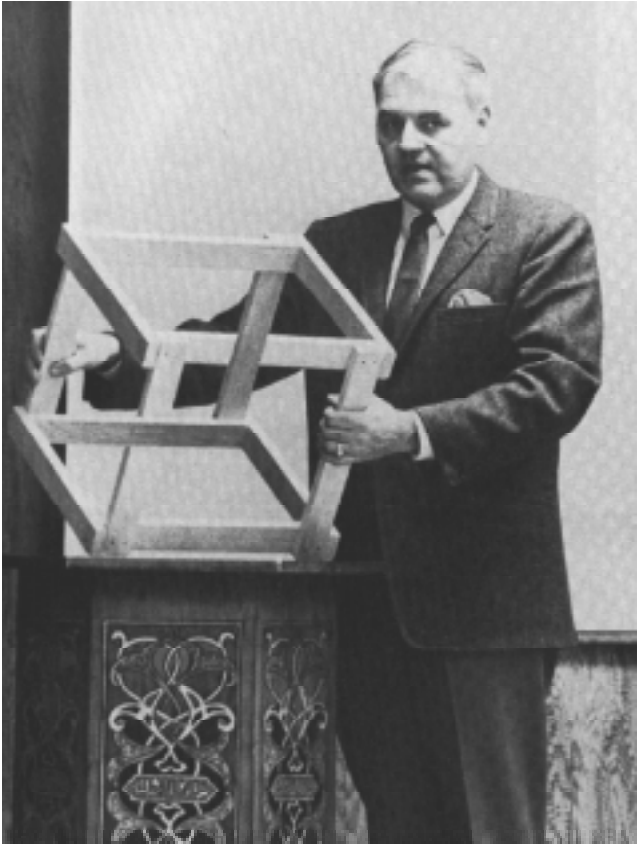
EF-100



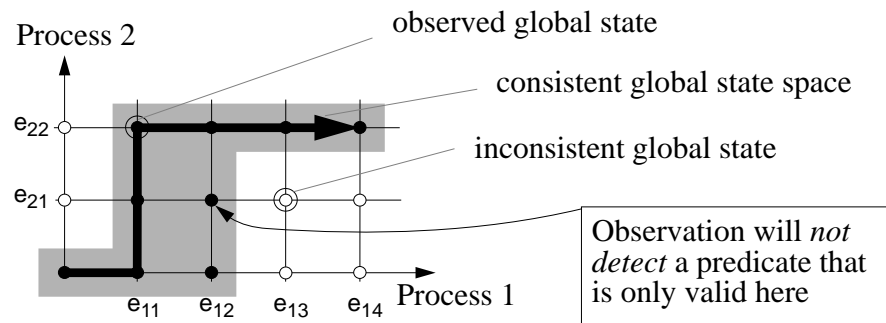
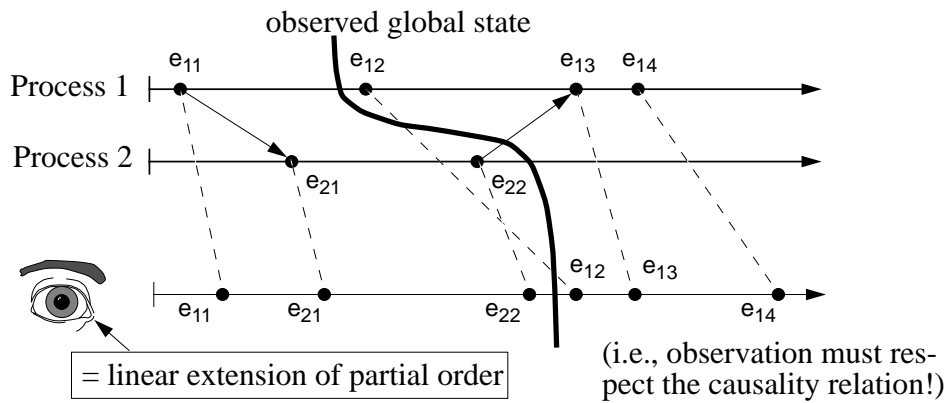
M.C. Escher: Belvedere (1958)



The Evidence!



The Global State Lattice

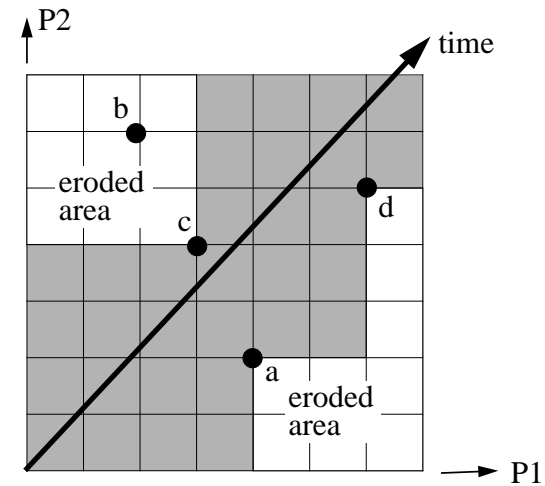
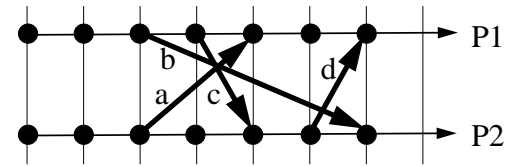


Observation = path in the state lattice (Which remains in the gray area of valid states)

- All observers see *all events* but *different global states!*
- Snapshot algorithm will yield *some* valid global state
- Sequence of snapshots ==> *some* observation

The Eroded State-Hypercube

- Here: 2 processes --> 2 dimensional cube

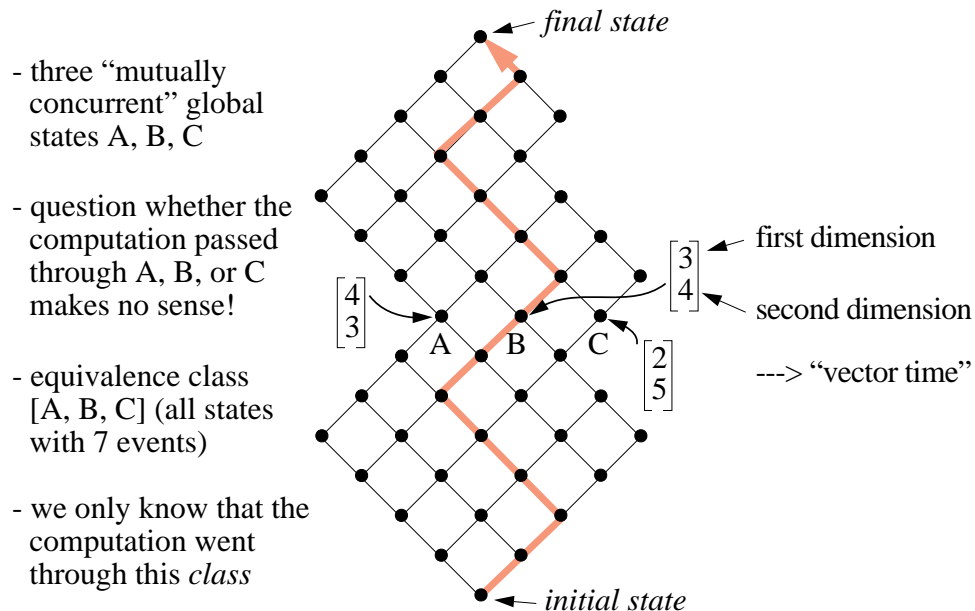


- Inconsistent global states are “eroded away”

- no message is received before it is sent
- messages synchronize the processes
- a process is blocked in a receive event until the message is available (and the corresponding sent has thus been executed)

The Lattice of Consistent States

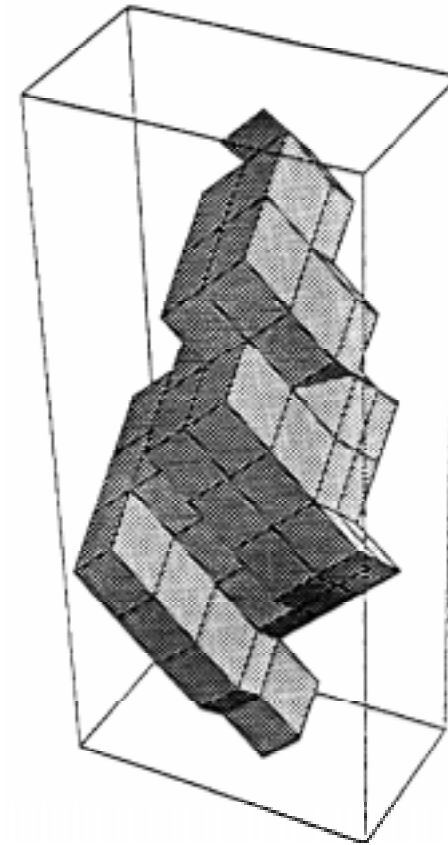
- To each *prefix* corresponds a consistent *cut*.
- To each *cut* corresponds a global (consistent) *state*.
- The “true” sequence of global states is one path through the lattice (but it is unknown if exact global time is unavailable)



- Consistent states form a (mathematical) *lattice*

- earlier, later global state; closed w.r.t. “sup” and “inf”
- visualized as a compact set (no holes)
- sublattice of the lattice of *all* global states

The 3-Dimensional Lattice



[Claude Jard et al., Rennes, France]

- compact set
- synchronization --> edge / crinkle on the surface
- “bottlenecks” become visible

The Dualism of the Diagrams

Serious Consequences...

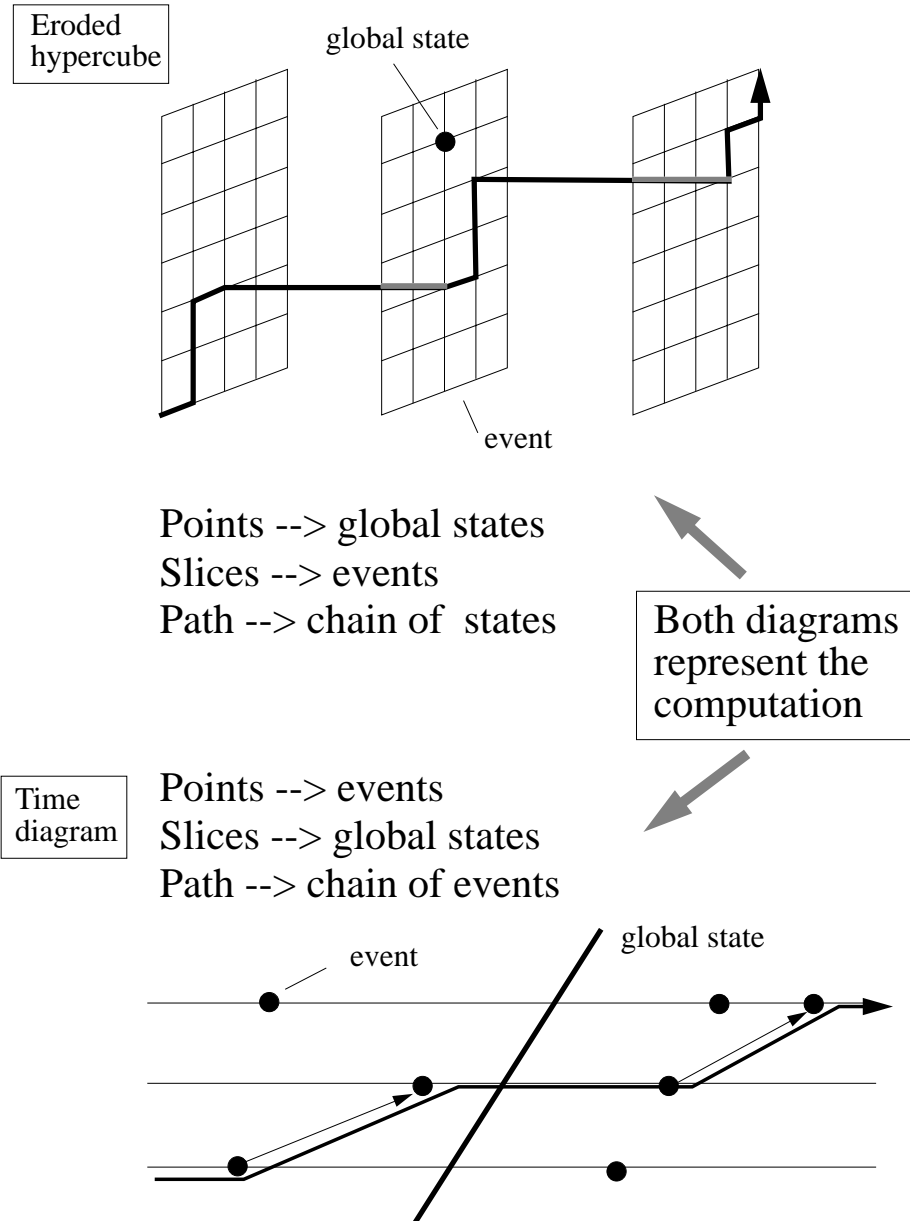
Predicates are satisfied relative to observers only

Debugging: "Next step" is not well-defined

Debugging: "stop when <condition>" *meaningless!*

(Although immediate halting is possible using execution replay!)

- Number of states is of polynomial size
 - Number of observers is of exponential size
- } --> hopeless in general!
- Single observer may miss the state where a certain predicate holds

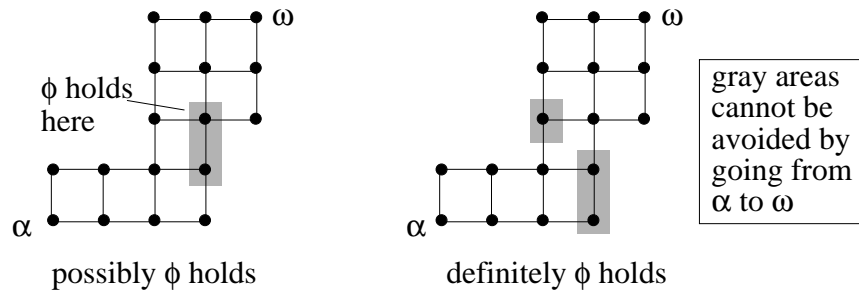


Modal Operators and Observer Independent Predicates

- *Possibly* Φ : "At least one observer sees Φ ."

Example: No observer must observe a state where more than one traffic light shows green: --> *Possibly* Φ should be false.

- *Definitely* Φ : "All observers see Φ ."



number of processes
number of events

- Complexity in general $O(|e|^n)$

More efficient determination of pos / def only for some predicate classes

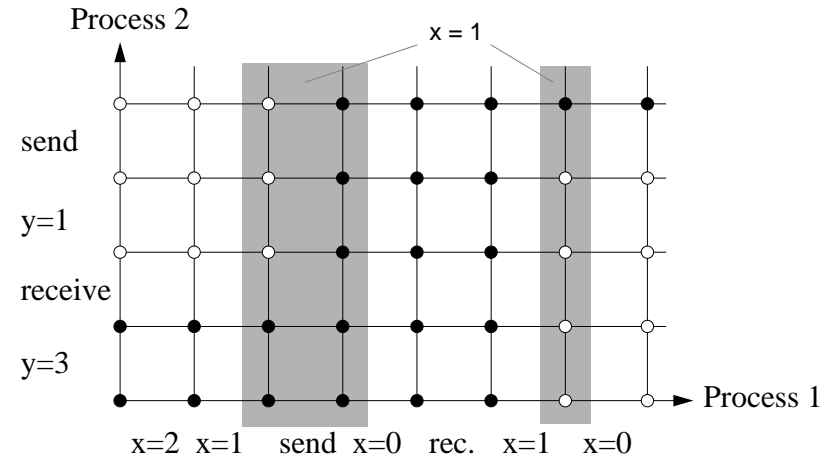
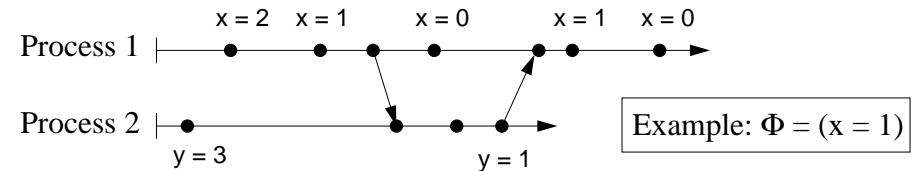
- Predicates Φ , for which *Possibly* $\Phi \Leftrightarrow$ *Definitely* Φ :

- If one observer sees ϕ , then all observers see ϕ .
 - Independent of the specific observer.
 - Efficient detection by a single observer is possible.
- } "good" predicates

- Such predicates can be attributed to the *computation!*

- Examples: *stable* properties (termination, deadlock); *local* predicates

Local Predicates



Whatever events the other processes execute, this does not change the value of Φ .

--> *Hyperplanes* in the n-dimensional lattice

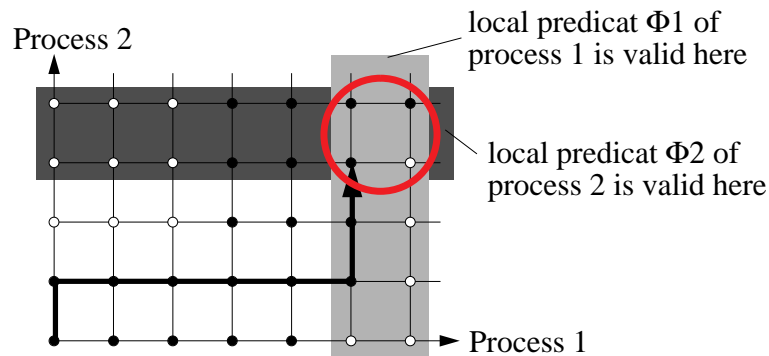
Every path from the initial to the final state necessarily meets all hyperplanes --> *inevitable*

--> *Possibly* $\Phi =$ *Definitely* Φ

Local predicates are not very interesting, however...

Disjunctions of inevitable (i.e., observer independent) predicates are also inevitable...

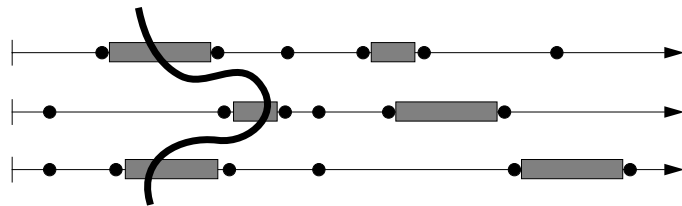
Conjunction of Local Predicates



- How determine whether “possibly $\Phi_1 \wedge \Phi_2$ ” holds?
 - Why is that of interest?
 - Example of traffic lights: possibly “traffic light 1 = green” and “traffic light 2 = green” should be false!

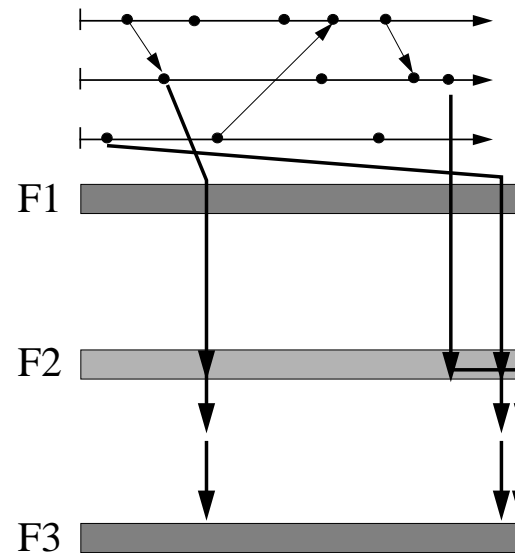
Idea: try to find a rubber band transformation such that there is a vertical line which cuts all processes in a state where the local predicat holds.

NB: Each consistent cut line can be made vertical



Idea for that: All processes execute in parallel, but a process stops as soon as its local predicate holds. Question: Does this idea work?

Determining “possibly $\Phi_1 \wedge \Phi_2 \wedge \dots$ ”

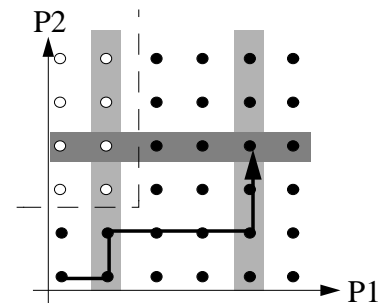
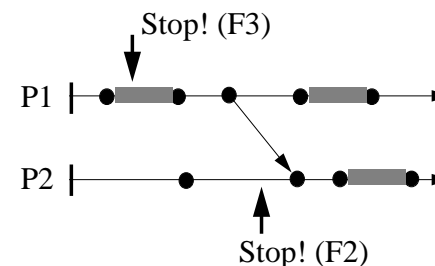


“Semantic filter”:
Only *relevant* events (change of the local predicat) pass.

Filter for *causal consistency*:
An event can only pass, if all causal predecessors of it have already been observed.

Dimension reduction filter:
keeps back all events of a process as soon as the local predicat of that process holds.

- *Idea:* Step by step the search space (n dimensional “cube”) is reduced by one dimension
- *However:* F3 must let pass events if otherwise the observation would block:



- Why is that scheme correct? How efficient is it?

Applications of the Detection Algorithm for “possibly $\Phi_1 \wedge \Phi_2 \wedge \dots$ ”

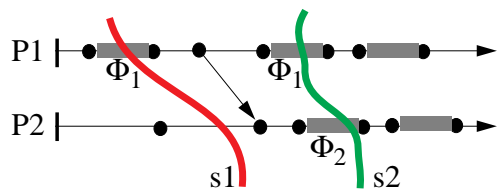
- *Termination* for synchronous communications:

Local predicate Φ_i : *process P_i is passive.*

- If *some* (consistent!) observer sees that all processes are (simultaneously!) passive, the computation has terminated.
- Detect *possibly* ($\forall P_i : P_i$ is passive).
- Detection scheme yields *termination detection algorithm*.

- *Debugging*: STOP WHEN $X1 = 3 \wedge X2 > 0$
(where X_i is a local variable of P_i)

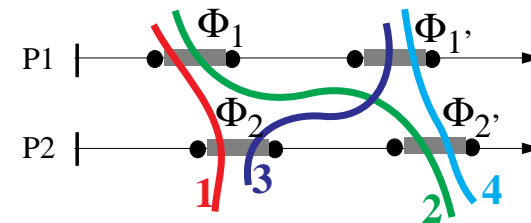
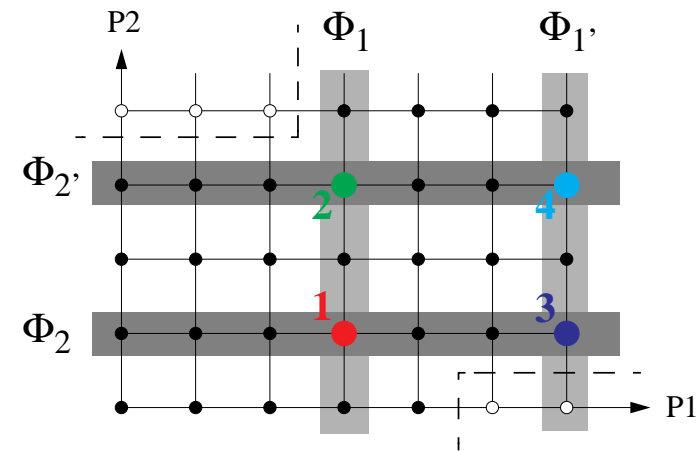
- Useful in *replay mode* (where immediate halting is possible).
- Algorithm yields the “*first*” state where the conjunction is true.



If P1 does not advance after its predicate Φ_1 becomes true, the computation would block in global state s_1 .

- Question: What would be the appropriate semantics of STOP WHEN $X1 = 3$ or $X2 > 0$?

Earliest State “ $\Phi_1 \wedge \Phi_2 \wedge \dots$ ”



- State s is *earlier* than state s' if there exists an observation “... s ... s' ...”.
- For two or more global states with “ $\Phi_1 \wedge \Phi_2 \wedge \dots$ ” there is always a common earliest such state.
 - Take the “process wise” min...
 - For states 2 and 3 in the example, this earliest state is state 1
- The consistent states form a lattice ($\rightarrow \exists$ “earliest”)

Stable Predicates

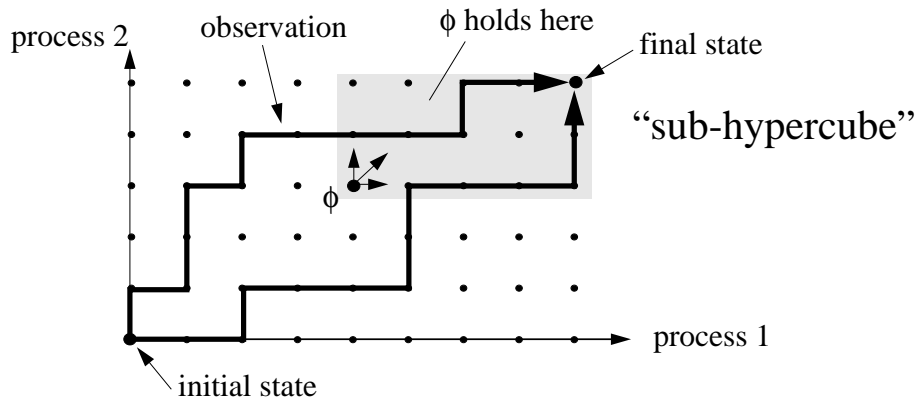
For *some* global predicates

- definition is meaningful (i.e., observer-independent)
- efficient detection is possible

Example: *stable* predicate ϕ on global states

- monotonic: "once true, ever true"
 - if $c1 < c2$ then $\phi(c1) \implies \phi(c2)$

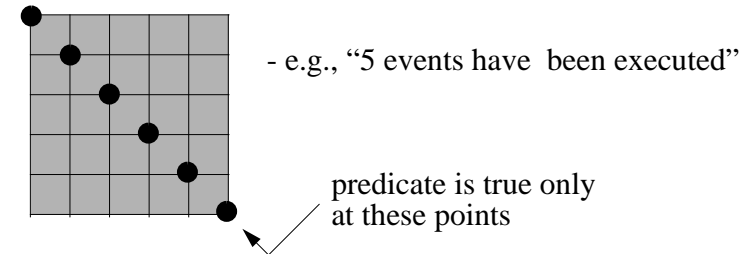
lattice of consistent states



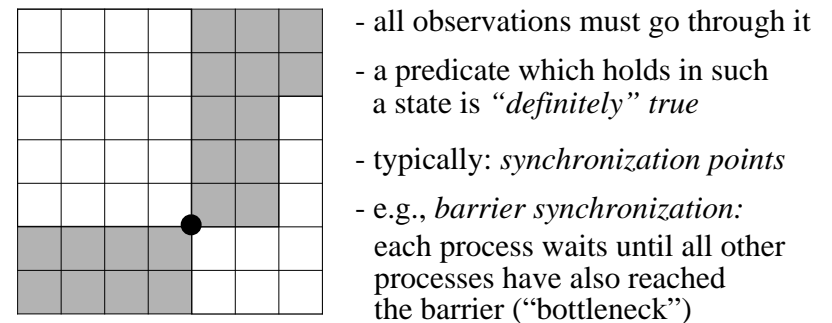
- All observers will inevitably detect the stable predicate (some observers will detect it earlier than others)
- Occasional testing for Φ on some consistent states is sufficient --> *snapshot algorithm makes sense!*
- If the snapshot algorithm establishes the truth of ϕ , ϕ is still true "now"!
- There exist some important stable predicates (e.g., "object is garbage", computation has terminated,...)

Other Observer-Independent Predicates?

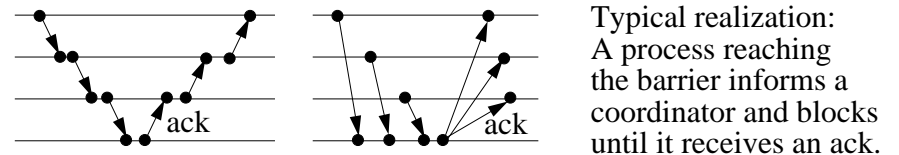
1) Some rather artificial predicates



2) "Inevitable" global states



The problem is not so much to verify whether the predicate holds in this particular state, but to make sure that such a state is eventually reached (before some action is executed)!



Typical realization:
 A process reaching the barrier informs a coordinator and blocks until it receives an ack.

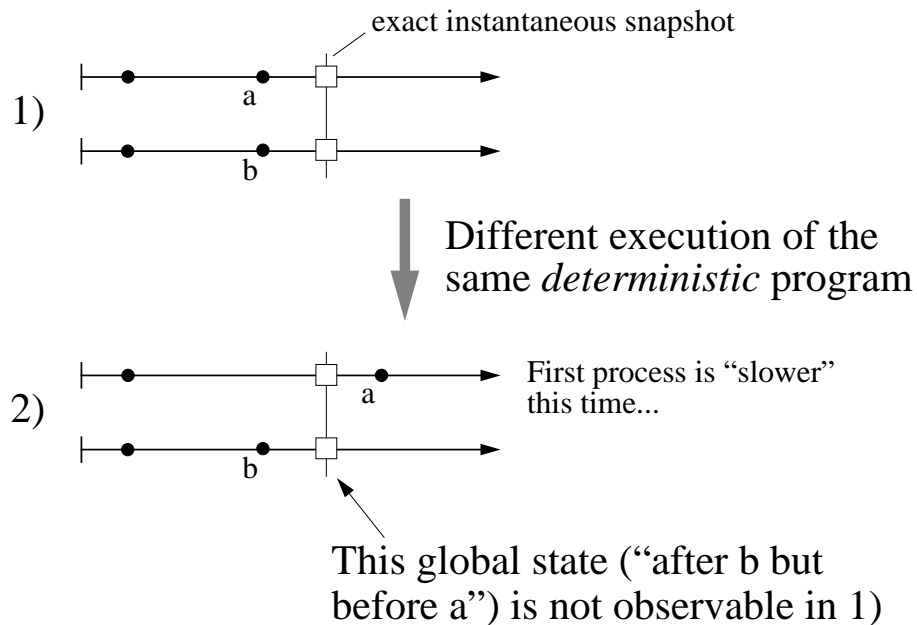
"At" the synchronization point all processes know that all other processes have also reached it (simultaneously?).

What if Global Time Exists?

e.g., perfectly synchronized local clocks
(but how good is “perfect”?)

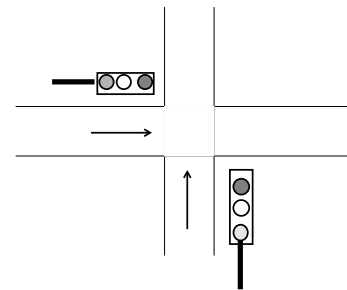
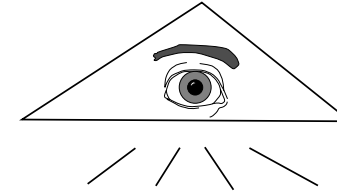
- ==> 1) Obtain “vertical” snapshots
- 2) Virtual image = real computation

Dual problem: *races!*



Hence the observed global state is not “absolute” or “definite”!

Do We Need Consistent Detection of Global Predicates?



Distributed traffic light control:
Do *all* observers see at most one green light?

Sometimes *inconsistent* observations are *acceptable*

Examples:

- 1) Performance debugging
- 2) $\text{load}(P1) + h > \text{load}(P2)$ ← “inherently global”

==> “weakly stable”

==> (slightly) inconsistent views do not harm

But: For deadlock detection, distributed recovery point,... inconsistent views are *not* acceptable!

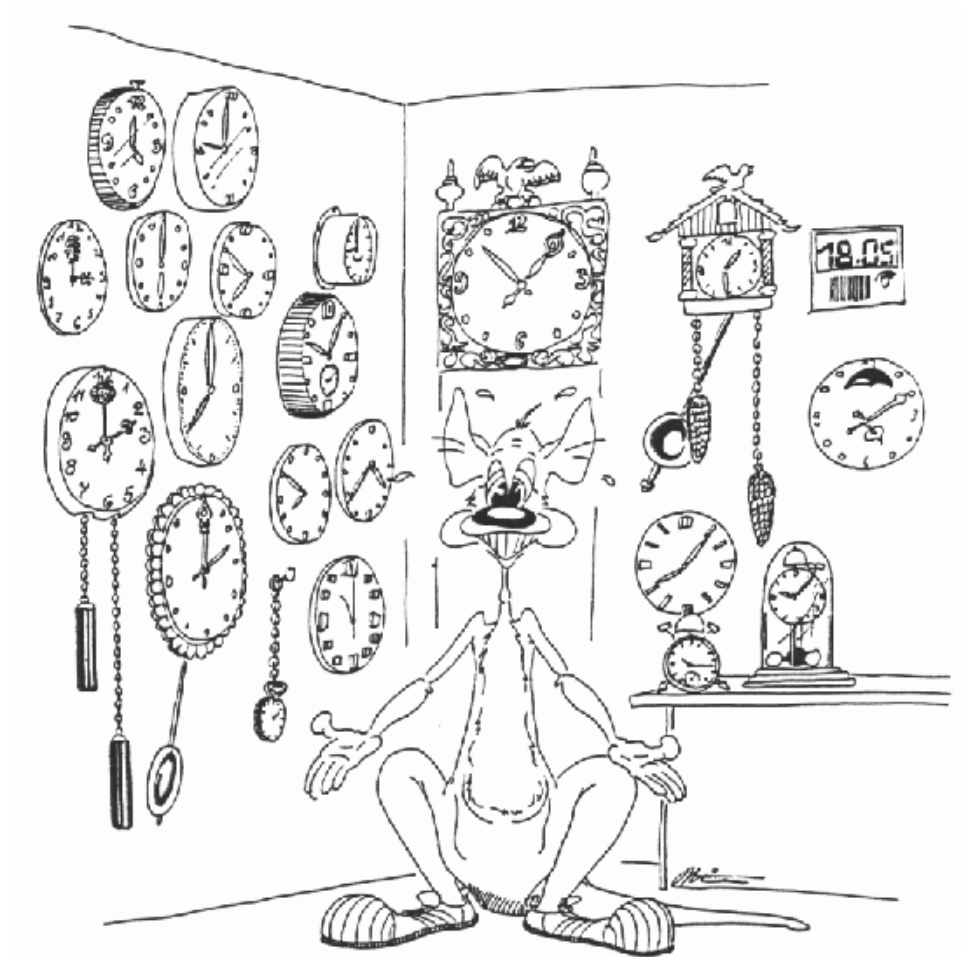
Observations...

- Consistent observation important:
 - Termination detection, deadlock detection,...
 - Debugging, monitoring...
- Predicates are meaningful only relative to an observer
- Huge number of different observers
 - ==> Global property may escape to a debugger!
- Only “few” predicates are observer independent, e.g.
 - *stable* (e.g., termination, garbage, deadlock, GVT-approximation)
 - *local* (rather trivial!)
- Efficient detection schemes exist for those predicates, all other predicates are difficult / impossible to detect

↙ e.g., snapshot algorithm

Observing parallel and distributed programs is much more difficult than observing sequential programs!

Time in Distributed Systems



R. G. Herrtwich, G. Hommel

Time ?

Quid est ergo tempus?
Si nemo ex me quaerat,
scio,
si quaerenti explicare velim,
nescio.

Augustine (354-430)

What then is time?
If no one asks me (what it is),
I know (what it is),
but if I want to explain it to someone,
(I find that) I do not know.

Time is money.

Benjamin Franklin (1706-1790)

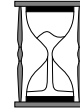
Time is how long we wait.

Richard Feynman (*1918, Nobel prize in physics 1965)

The indefinite continued progress of existence,
events, etc., in past, present,
and future regarded as a whole.

Concise Oxford Dictionary, 8th Ed.

The Arrow of Time: Past, Present, and Future

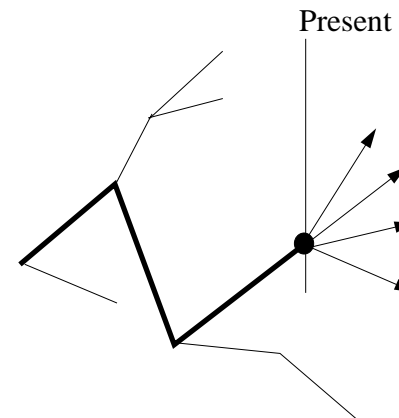


Tempus fugit
(Time flees / flies)

This is the melancholic
dimension of time...

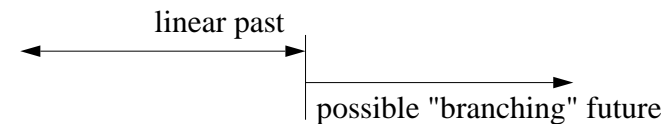
*Time goes, you say?
Ah no! Alas, time stays, we go.*

Austin Dobson, The Paradox of Time



Two roads diverged in a yellow wood,
And sorry I could not travel both.
And be one traveler, long I stood
And looked down one as far as I could
To where it bent in the undergrowth;
...
Then took the other, as just as fair,
...
I shall be telling this with a sigh
Somewhere ages and ages hence:
Two roads diverged in a wood, and I -
I took the one less traveled by,
And that has made all the difference.

Robert Frost (1874-1963)
The Road Not Taken (1916)

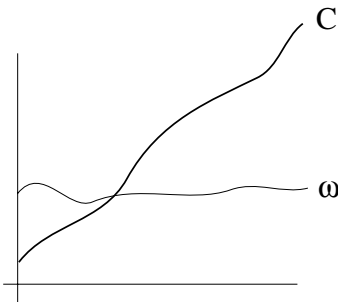


- Looking back, time always seems to be linear...

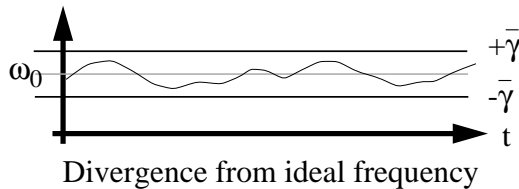
Clocks and Real Time

- *Clock*: Device to measure the physical phenomenon “time”.

$$C(t) = k \int_{t_0}^t \omega(\tau) d\tau + C(t_0) \quad \text{Value of clock C at t}$$



- *Precision* of a clock depends on the stability of its oscillator (with ideal frequency ω_0).



- Ideal clock: $C'(t) = 1$, i.e. $\omega(t) = \text{constant}$.
- Many influencing factors (age, temperature,...) on the stability

- Deviations may accumulate!

--> *Resynchronization* is necessary from time to time

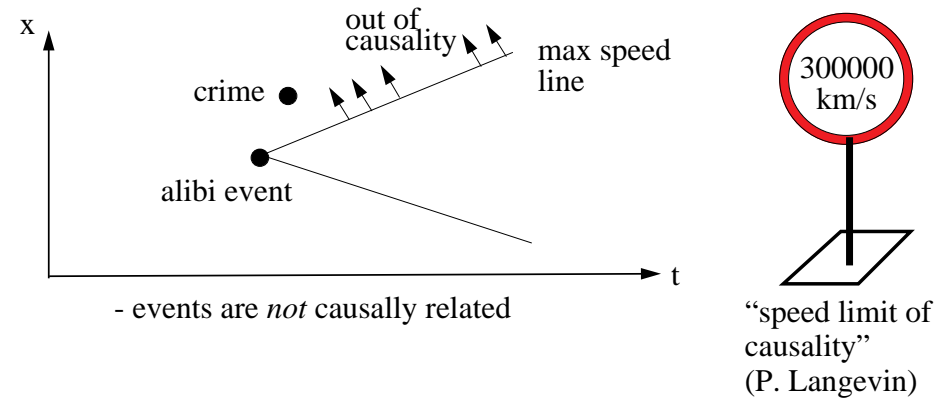
- set clock back / forward (--> $C(t)$ jumps and is non-monotonic)
- increase / decrease oscillator frequency

Time is Powerful

1. *Population census* (consistency by simultaneity)

- agree upon a future date
- everyone gets counted at the same moment

2. Determining *potential causality* (“alibi principle”)



- events are *not* causally related

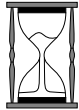
3. *Mutual exclusion* (fairness by linear time order)

- the earliest gets access...

We don't have (real) time in distributed systems

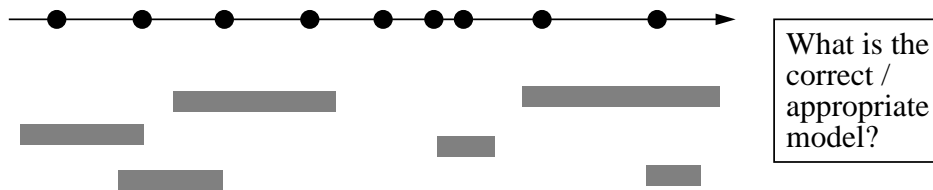
--> look for an adequate substitute (--> *logical time*)

- has most important properties
- is (easily) realizable



Time: Properties and Models

- *Points* “in time” together with a relation “later”
- Or: time *intervals* together with “later”, “overlaps”...



- Are the two models / views “compatible”? (e.g., startpoint and endpoint)
- Structure and properties of time points:

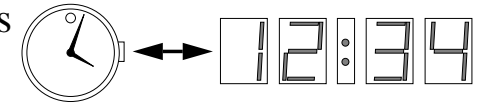
- transitive
- irreflexive } --> lin. order
- linear
- unbounded (“time is eternal”: no beginning and no end)
- dense (there is always a point between two other points)
- continuous
- metric
- homogeneous
- archimedean / inductive (each point will eventually be reached)

- Models: real numbers, rational numbers (?)

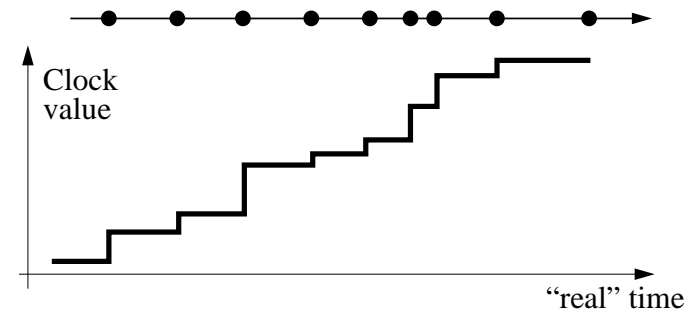
- Are all these properties needed? (when? for what?)
- e.g., discrete (instead of continuous) --> integers suffice!

Time and Clocks in Computer Science

- Hardware counters as clocks --> time becomes *discrete*



- Clock overflow (e.g., long simulation runs) --> time is not eternal but *bounded*
- Event oriented view: nothing happens between two events --> Clocks need not run continuously --> Change clock value only when an event happens
- "World view": Time = Happening of events
- Example of this world view: Event driven simulation



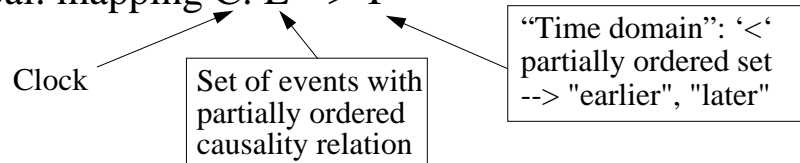
Hence: We call concepts / devices “time” / “clocks” even though they do not have all the ideal properties!

but what are the *essential* properties?

Logical Timestamps

- Purpose: compare events by their timestamps.

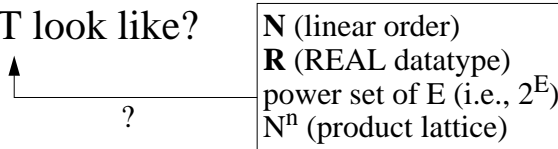
- Goal: mapping $C: E \rightarrow T$



- For $e \in E$ we call $C(e)$ the *timestamp* of e .

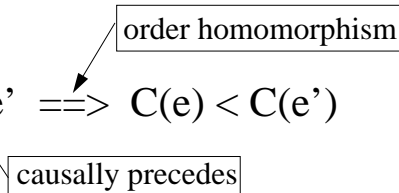
- $\underbrace{C(e')}_{\text{or: } e'}$ later than $\underbrace{C(e)}_{\text{or: } e}$ if $C(e) < C(e')$.

- How should T look like?



- Reasonable requirements:

Clock condition: $e < e' \implies C(e) < C(e')$



Interpretation:

“time respects causality”

If an event e may influence another event e' , then e must get a lower timestamp than e' .

- We would also like to have the converse relation!

Lamport's Logical Clocks

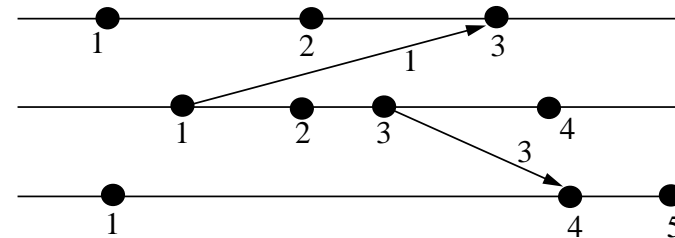
Communications of the ACM 1978:

Time, Clocks, and the Ordering of Events in a Distributed System

$C: (E, <) \rightarrow (N, <)$ Assigns timestamp

causality relation (“potential” causality)

$e < e' \implies C(e) < C(e')$ Clock condition



- “Paths of causality” from left to right

- Protocol for clock implementation:

- local clock ticks for *each* event
- *send* event: timestamp is piggybacked
- *receive* event: $\max(\text{local clock}, \text{timestamp})$

before the clock ticks

- *Proposition:* Protocol guarantees clock condition.

- *Proof:* Causality paths are monotonic.

Properties of Lamport-Timestamps

- What remains from the properties of real time?

- + lin. order, unbounded
- + respects causality (clock condition) ← as does real time!
- discrete
- does not “flow automatically”

- Timestamp = Length of longest preceding chain

- Clock condition ==>

- locally increasing timestamps
- send event has smaller timestamp than receive event
- $C(a) < C(b) \implies \text{not } (b < a)$

Proof.: $b < a \implies C(b) < C(a)$
 $\implies \neg(C(a) < C(b))$

Future cannot influence the past!

“critical path” --> concurrency measure, time complexity

- We have: $C(a) = C(b) \implies a \parallel b$

- Proof left as an exercise...

causally independent
 i.e., $\neg(a < b) \wedge \neg(a > b)$

- Do we have the converse of the clock condition?

- No, $C(e) < C(e') \implies e < e'$ does *not* hold!
- We only have: $C(e) < C(e') \implies e < e' \text{ or } e \parallel e'$ ← see example

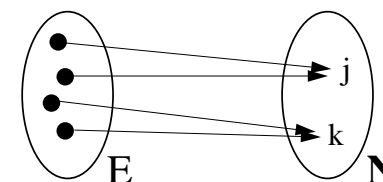
- Hence:

From the timestamps we cannot (always) conclude whether two events are causally dependent or not!

- But wouldn't that be the major goal of timestamps (since causality is the only structure we have in our abstract distributed computations)?
- Yet, Lamport timestamps are useful for some purposes (e.g., mutual exclusion)

Lamport-Timestamps: “Non-Properties”

1) Mapping is not injective:



- Important, e.g., for: "The one who came earliest wins"

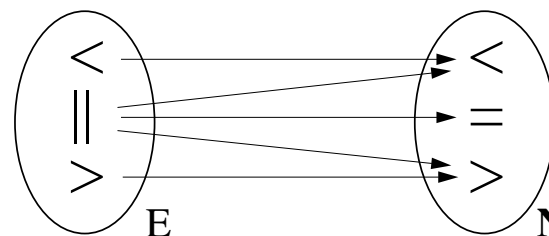
- Solution: Lexikographical order $(C(e), i)$, where i denotes the process number, on which e happens

==> Now - all events have *different* timestamps (i is a “tie breaker”)
 - there is unique *smallest* event for each set of events

- Linear order $(a, b) < (a', b') \iff a < a' \vee a = a' \wedge b < b'$

- Mapping (still) respects causality: $(E, <) \rightarrow (N \times N, <)$
 (only causally independent events are ordered by their second component)

2) Loss of structural information:



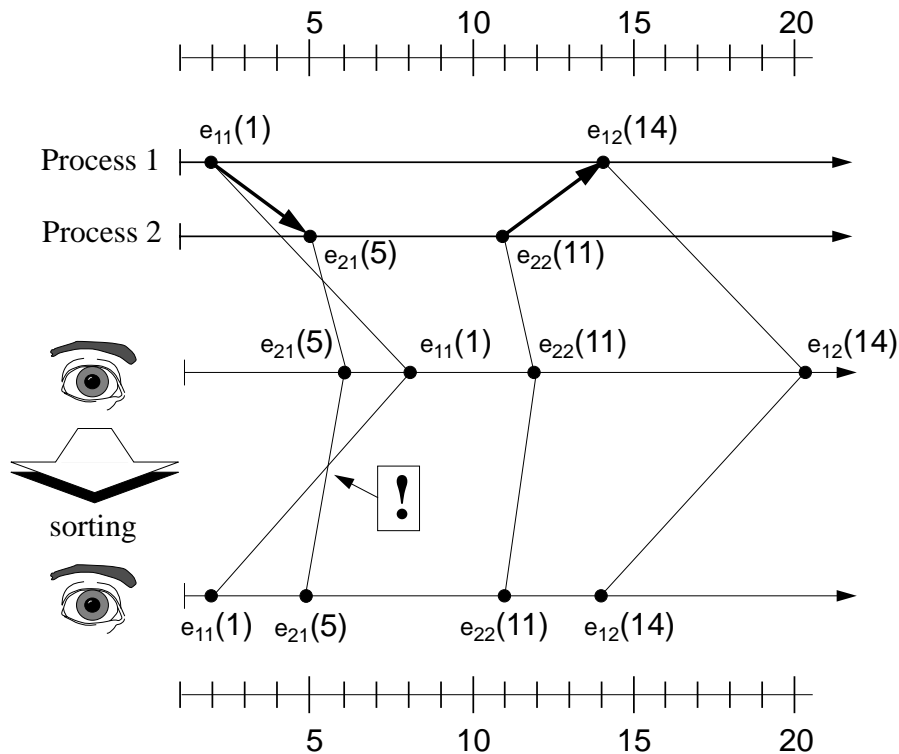
Important defect since one purpose of timestamps is to draw conclusions on the structural relation among events!

- Negation is lost
 - Order homomorphism, but no isomorphism
 - E is a *partial* order, N is a *linear* order (Causally independent events may become comparable!)
- Also note that “=” is transitive, but “||” is not!

Is there a “better” timestamping scheme?

Realizing Causally Consistent Observers with Real-Time

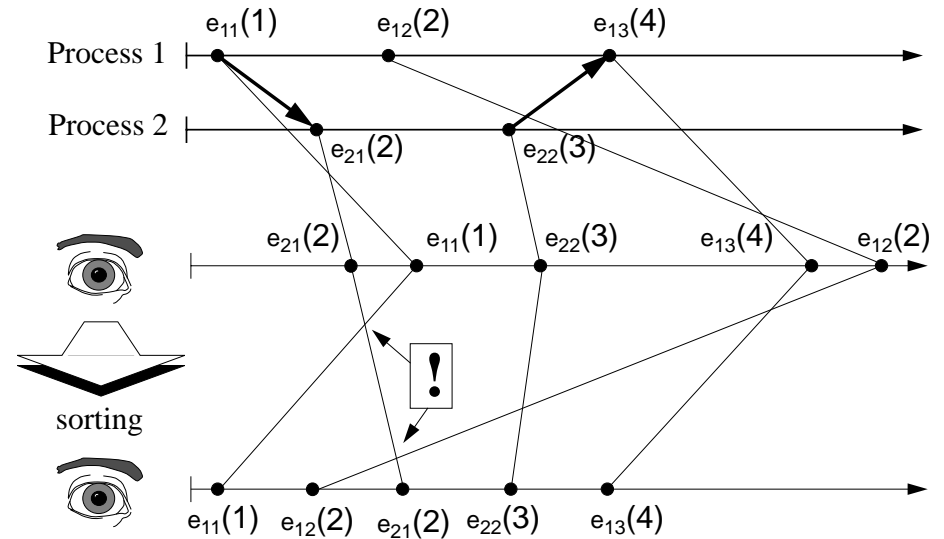
- Basic idea: *Time respects causality*
 ==> *Sorting by global time = "sorting by causality"*
 (--> topological sorting)



- Observer recreates the "true" computation.
- Problem: requires (global) real-time for timestamps!

Realizing Causally Consistent Observers with Lamport Time

- Basic idea: *Lamport time respects causality* ==>
 Sorting yields a *linear extension* of the causality relation.



- Problem: Not well suited for *online monitoring*.
 - Before delivering ("committing") an event, one must be sure that no event with a smaller timestamp will arrive later (see e_{13} and e_{12})!
 - FIFO channels to the observer help, but may still cause *long delays*.
 - Problem also, if only a *subset* of all events is observed.

==> Find a more suitable model of logical time!

Vector Time(stamps)

Quot tempora tot astra.
G. Bruno (1548-1600)

reasonable definition in our model

- $Time :=$ set of past events \implies
- $Timestamp(e) := \{e' \mid e' \leq e\}$

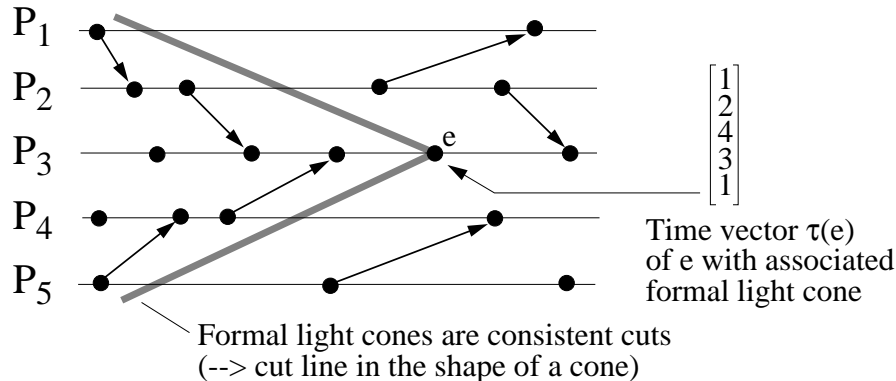
Formal light cone: set of (causally) past events which can affect e

- Light cone can be represented by *locally latest events* (left closed sets)
- There exist n such events ($n =$ number of processes)

\implies Define the n -dimensional vector $\tau(e)$ as follows:

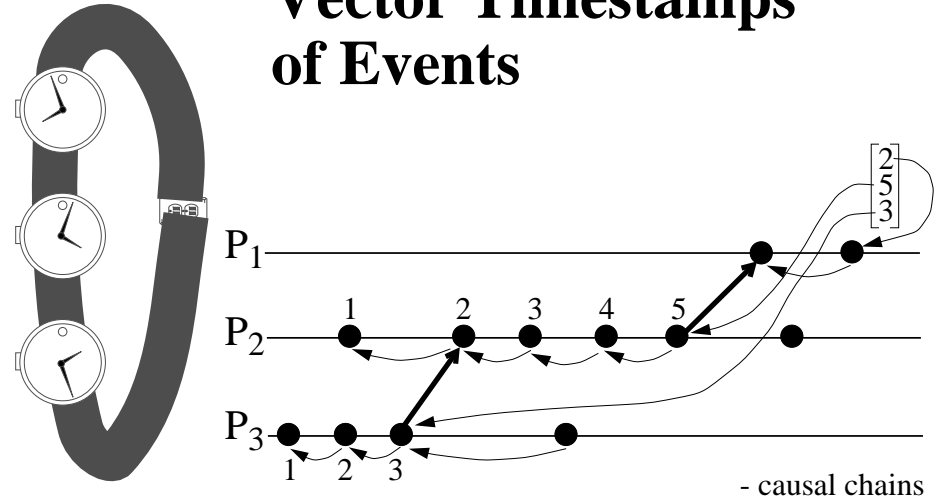
$$\tau(e)[i] := |\{e' \in E_i \mid e' \leq e\}|$$

Set of events on process P_i



- \implies *Timestamp* is an n -dimensional vector
- \implies *Time* is the set of all n -dimensional vectors
- \implies *Clock* is an array $C[1:n]$ (“device” to keep current time)

Vector Timestamps of Events



- Each event has a “vector time stamp”
- Component i points to the most recent causally past event on process i .
- Therefore, because events of a process are totally ordered, it implicitly also “points” to all earlier events.
- \implies Vector represents *whole causal past*.
- \implies Encodes *knowledge* about *each* past event.
- Sometimes some optimizations are possible (omit 0-components, sparse arrays, send only delta-values, use topological knowledge...)

“Vector time”: isomorphic representation of the causality relation (partial order \implies lattice structure)

Timestamp “Arithmetic”

$$\begin{bmatrix} 1 \\ 3 \\ 4 \\ 3 \\ 2 \end{bmatrix} \leq \begin{bmatrix} 1 \\ 7 \\ 4 \\ 6 \\ 2 \end{bmatrix}$$

comparable

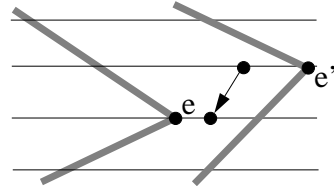
$$\begin{bmatrix} 1 \\ 3 \\ 4 \\ 3 \\ 7 \end{bmatrix} \parallel \begin{bmatrix} 5 \\ 3 \\ 8 \\ 3 \\ 2 \end{bmatrix}$$

concurrent

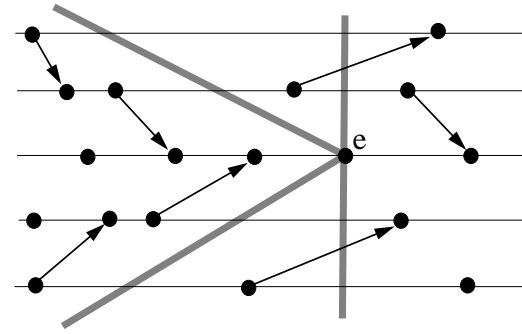
‘<’ is defined as “≤ but ≠”

Interpretation of $\tau(e) < \tau(e')$:

- e lies in the causal past of e'
- cone of e is included in the cone of e'



Vector Time and Ideal Observers



$$\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \dots \begin{bmatrix} 2 \\ 4 \\ 5 \\ 4 \\ 3 \end{bmatrix}$$

Observations of the ideal observer

$$\tau(e) = \begin{bmatrix} 1 \\ 2 \\ 4 \\ 3 \\ 1 \end{bmatrix}$$

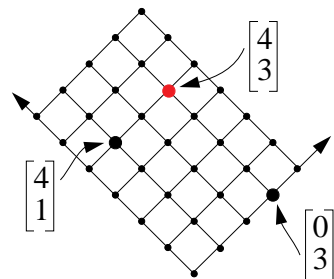
$$\text{id}(e) = \begin{bmatrix} 1 \\ 3 \\ 4 \\ 3 \\ 2 \end{bmatrix}$$

NB: The causal past of an event forms a consistent cut!

- Locally number all events: 1,2,3,...
- Ideal observer sees an event immediately
- Adequate data structure for representing this ideal knowledge: vector / array
- For every causally consistent observer: $\tau(e) \leq \text{id}(e)$ ($\forall e$)
 - a causally consistent observer knows the whole causal past of an event
 - ideal observer typically also knows some other events
- $\tau(e)$ = Infimum of all possible ideal views $\text{id}(e)$
 - Note: $\text{id}(e)$ depends on the specific time diagram!
 - But: $\tau(e)$ is invariant w.r.t. rubber band transformations!

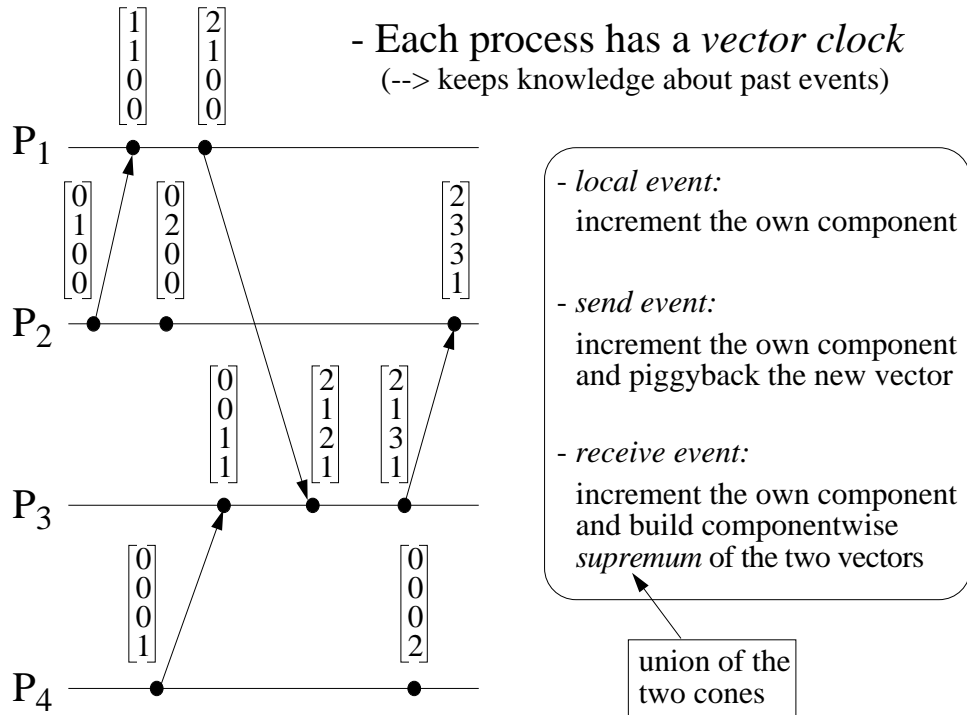
$$\text{sup} \left(\begin{bmatrix} 1 \\ 4 \\ 2 \\ 3 \\ 7 \end{bmatrix}, \begin{bmatrix} 8 \\ 3 \\ 4 \\ 3 \\ 2 \end{bmatrix} \right) = \begin{bmatrix} 8 \\ 4 \\ 4 \\ 3 \\ 7 \end{bmatrix}$$

sup = componentwise maximum

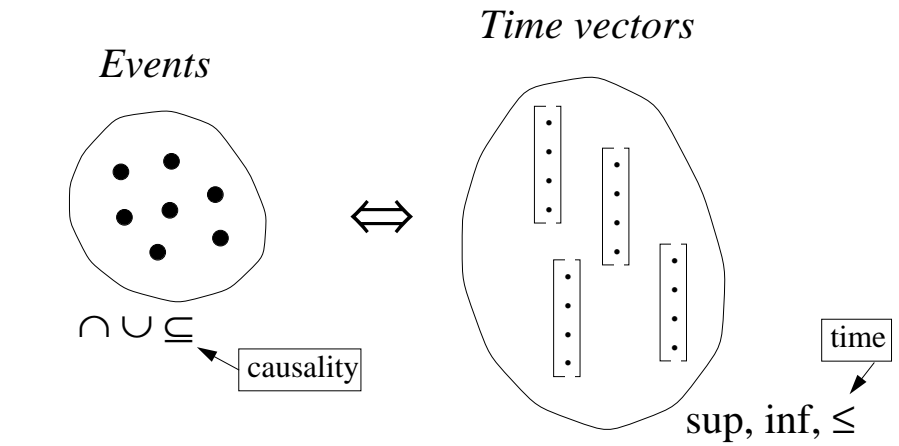


Propagation of Time Knowledge

(--> Implementation of vector time)



Computing with Sets of Events



Set theoretic operations \Leftrightarrow Algebraic operations
 (--> "compute")

Lattice structure on 2^E (ideals) \Leftrightarrow Product lattice on \mathbb{N}^n



Order theoretic properties \Leftrightarrow Algebraic properties

Vector clocks / vector timestamps --> operational "manipulation" of the causality relation

componentwise \rightarrow

- *Claim*: $e < e' \Leftrightarrow \tau(e) < \tau(e')$

- *Interpretation*:
 - $\tau(e) \leq \tau(e') \Leftrightarrow$ there exists a causal chain from e to e'

- *Corollary*: $e \parallel e' \Leftrightarrow \tau(e) \parallel \tau(e')$
 not related

monotonic w.r.t. time vectors!

Isomorphic representation of the causality relation!

Interpretation: Two events do not influence each other iff they are concurrent (w.r.t. the time domain)

Applications of Vector Time

<Momo meets Professor Hora>:

Clocks were standing or hanging wherever Momo looked - not only conventional clocks but *spherical timepieces showing what time it was anywhere in the world...*

“Perhaps one needs a watch like yours to recognize these critical moments,” said Momo. Professor Hora smiled and shook his head. “No, my child, the watch by itself would be no use to anyone. *You have to know how to read it as well.*”

Michael Ende, Momo

- Debugging

- Localising errors (“... can / cannot be the cause...)
- Race conditions (causal independence)
- Efficient replay

- Performance analysis, concurrency measures

- “bottleneck” in the lattice; degree of synchronization
- causally independent events can be executed in parallel

- Implementation of causally consistent observers

- Causal broadcast
- Causal order

- Implementation of consistent snapshots

- Local snapshots at pairwise concurrent events

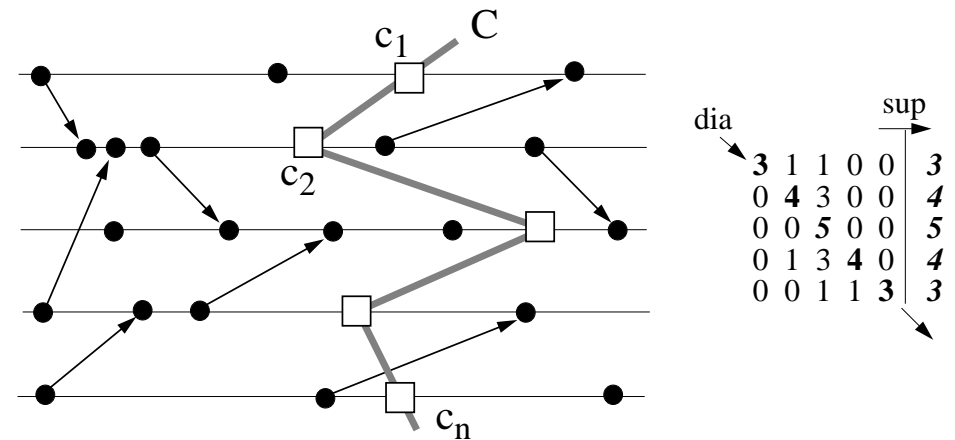
- ... ?

The Cut Matrix

- Cut matrix $\$$ of a cut C (with cut events c_i):

$$\$:= (\tau(c_1), \tau(c_2), \dots, \tau(c_n))$$

(i.e., take time vectors of cut events c_i as the columns)



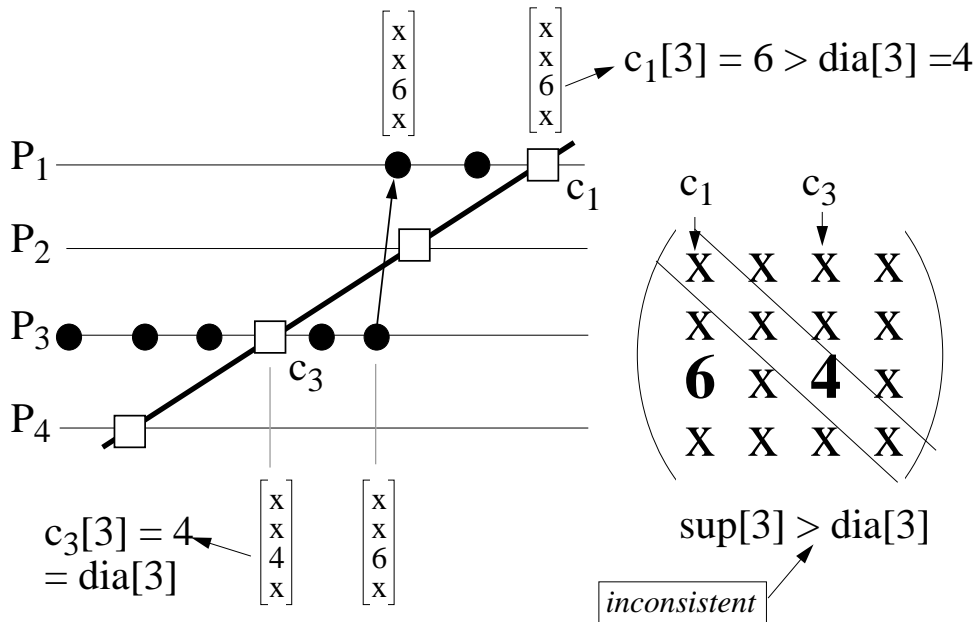
$$C \text{ consistent} \Leftrightarrow \text{dia}(\$) = \text{sup}(\$)$$

diagonal vector

for each line:
maximal value

(i.e., the maximum of a row is the diagonal element)

The “sup = dia” Consistency Criterion



A process (P_1) other than P_3 knows (at cut event c_1) something about local events on P_3 , on which P_3 itself does not yet know anything (i.e., which happen *after* c_3).

\Leftrightarrow

There exists a path from a P_3 -event *after* c_3 to an event *before* c_1 .

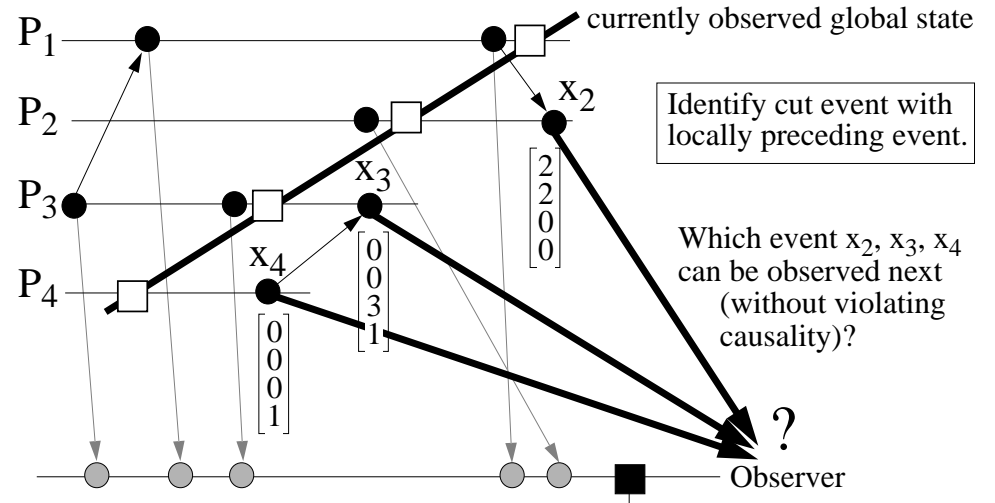
\Leftrightarrow

[generalization over all indices $i \neq j$]

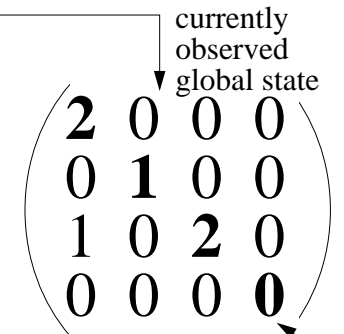
The cut is inconsistent.

Implementing Consistent Observers

- See only *consistent snapshots* in their reconstructed view
- Sequence of observed events *respect the causality order* (i.e., observation is a *linear extension* of the causality relation)

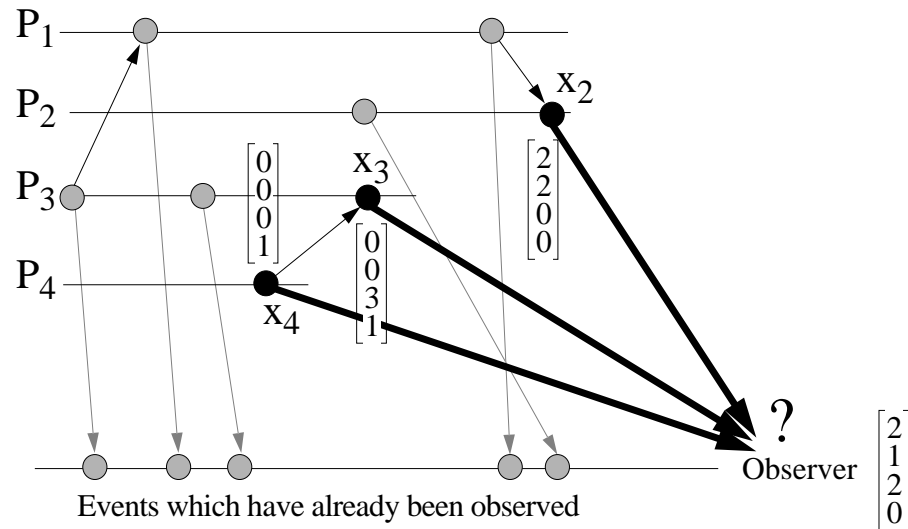


- Which column may be replaced? (x_2, x_4 , but not x_3)
- Observer keeps $dia(\$)$. Timestamp of next observed event must be $\leq dia(\$)$, except diagonal component

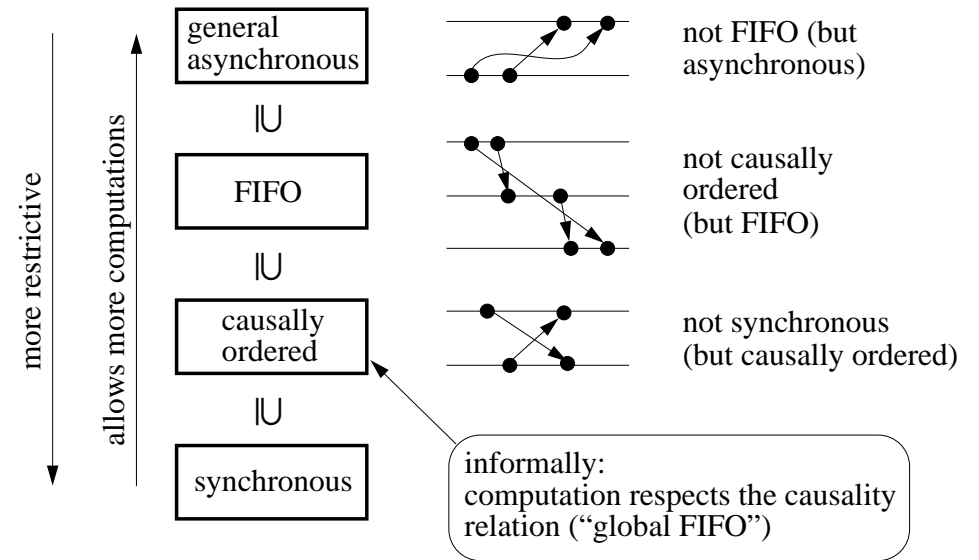


- Goal: Keep always consistent, i.e., $dia(\$) = sup(\$)$!
- NB: Observer needs only a *vector* (dia), not a *matrix* to identify the currently observed state!
- All observation messages do also have a vector.
- Idea: “*This event depends on another event that I should have observed earlier. Hence I better wait until I get notice from the other event...*”
- NB: Does also work if only a *subset* of all events is observed!

Realizing Causally Consistent Observers with Vector Time



The Communication Hierarchy



- Which event x_2 , x_3 , x_4 can be observed next (without violating causality)?
- Compare vector time of event with observer's vector.
- Event x_3 should not be observed, because it "knows" of one event on P_4 which the observer has not yet seen.
 - Idea: "This event depends on another event that I should have observed earlier. Hence I better wait until I get notice from the other event..."
- Realization: Delivery filter which uses message queues.

Typical questions:

- 1) Given a computation with asynchronous communications
 - > can it be realized with FIFO channels? (i.e., does it respect the FIFO property?)
 - > does it respect the causality relation?
 - > is it realizable with synchronous communications (e.g., does it run on a transputer with occam? Or does it block?)
- 2) Is a given algorithm, which is correct for synchronous communications, still correct for a more general model?
 - > e.g., can the algorithm tolerate receiving messages out of order?

Vectors are rather clumsy. Do we really need them to guarantee consistency and to make correct statements about the system?

What are synchronous communications?

(relative to *asynchronous* communications)

- *Naive*: telephone <--> letter

- *Literally*: syn - chronous

↑ same ↑ time

- Does this mean that send and receive happen *simultaneously*?

- But *instantaneous message transmission* is unrealistic!

- NB: There exist distributed programming languages

- which use synchronous message passing (e.g., CSP or Occam)
- which use asynchronous message passing
- which use both (e.g., MPI)

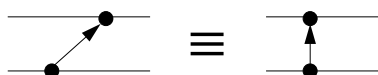
- Restate the headline-question in a more formal way:

- How do we *model* synchronous communication?
- How do we *define* distributed computations with synchronous message passing?

- Proposition:

Synchronous = virtually simultaneous
= as if msg transmission were instantaneous

suitable rubber band transformation?



“As if” Messages were Instantaneous

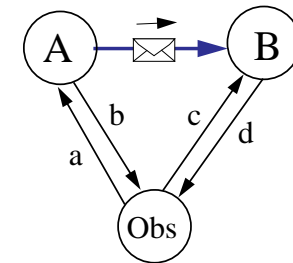
If for a distributed computation a phenomenon can be observed which is impossible with instantaneous messages, the computation must not be realizable with synchronous message passing semantics.

==> message passing should then not be called “synchronous”

Example:

The observer first asks A about the number of messages it sent to B.

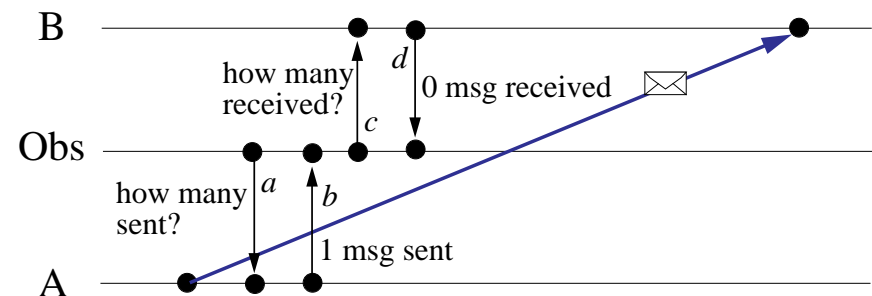
Then it asks B about the number of messages it received from B.



- Restate the headline-question in a more formal way:
- How do we *model* synchronous communication?
- How do we *define* distributed computations with synchronous message passing?

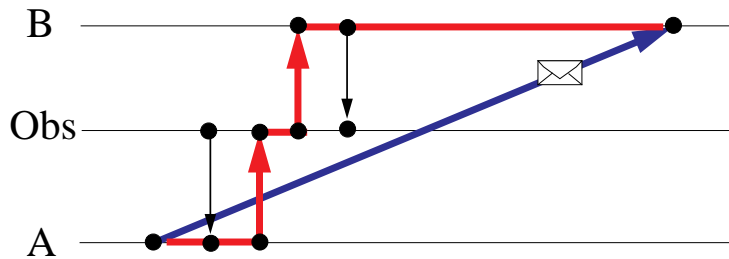
- Proposition:

Synchronous = virtually simultaneous
= as if msg transmission were instantaneous



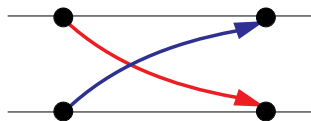
Observer learns that a message from A to B is *in transit* for a certain *duration* ==> not synchronous!

Vertical Message Arrows



- The message from A to B is *overtaken in an indirect way* by a *chain* of other messages.
- The direct message can therefore *not be made vertical* by a rubber band transformation.
(A message of the chain would then go backwards in time)

- Another computation which is not possible with synchronous communications (\implies deadlock):



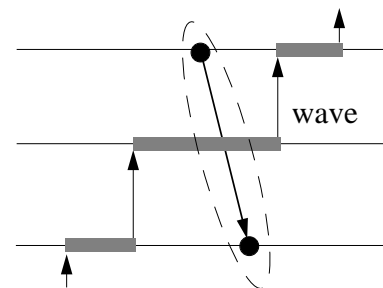
Although each *single* arrow can be made vertical, it is not possible to draw the diagram in such a way that *both* arrows are vertical!



Various Characterizations of Synchronous Communications

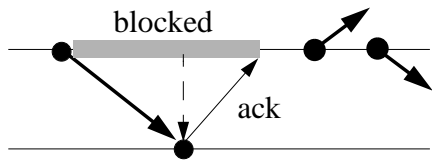
- Question: are they all equivalent?
- Problem: some characterizations are informal or less formal than others

- 1) Best possible approximation of *instantaneous* communications.
(Without clocks, it is not possible to prove that a message was not transmitted instantaneously)
- 2) Space-time diagrams can be drawn such that *all message arrows are vertical*.
- 3) *Communication channels* always appear to be *empty*.
(i.e., messages are never seen to be in transit)
- 4) Corresponding *send-receive events* form one *single atomic action*.



- But what exactly does "atomic" mean?
- Does the combined event happen before or after the wave? Should this be possible with synchronous communication?

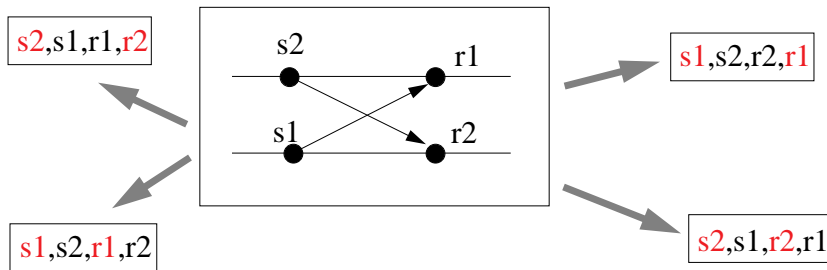
5) Send action *blocks* until an acknowledgement from the receiver is received.



- But can't synchronous communication be implemented (on a system with asynchronous communications) without blocking?

- Motivation: As if the message is sent at the moment it is actually received.

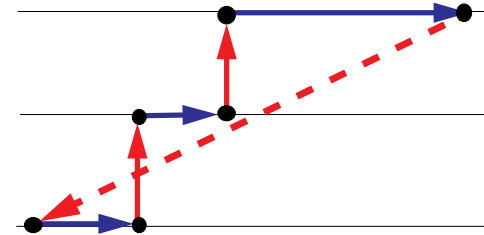
6) \exists linear extension of $(E, <)$ such that \forall corresponding communic. events s, r : r is an immediate predecessor of s .



- The example has 4 different linearizations. In all of them a pair of corresponding send-receive events is separated by other events. Hence this computation *cannot* be realized synchronously.

- Motivation: corresponding events form a single atomic action

7) No cycle is possible by moving along message arrows in either direction, but always from left to right on process lines.



- Interpretation: Ignoring the direction of message arrows \implies

- send / receive is "symmetric"
- "identify" send / receive

- If such a cycle exists \implies no "first" message to schedule

- If no such cycle does exists \implies message schedule exists

7) Define a (transitive) scheduling relation ' $<$ ' on messages:

$$m < n \text{ iff } \text{send}(m) < \text{receive}(n)$$

The graph of ' $<$ ' must be *cycle-free*.

- Then whole messages (i.e., corresponding send-receive events s, r) can be scheduled at once (s before r), otherwise this is not possible.

8) Synchronous causality relation \ll is a *partial order*.

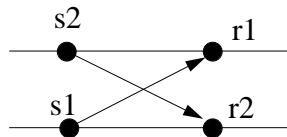
Definition of \ll :

1. If a before b on the same process, then $a \ll b$
 2. $x \ll s$ iff $x \ll r$ (“common past”)
 3. $s \ll x$ iff $r \ll x$ (“common future”)
 4. Transitive closure
- } for all corresp. s, r and for all events x

Interpretation: corresponding s, r are not related, but with respect to the synchronous causality relation they are "identified"

they have the same past and future

Example:



- a) $s1 \ll r2$ (1)
 - b) $r1 \ll r2$ (a, 3)
 - c) $s2 \ll r1$ (1)
 - d) $r2 \ll r1$ (b, 3)
- } cycle, but $r1 \neq r2!$

Compare this characterization to the earlier one "no cycle in the message scheduling relation".

Causally Ordered Computations

Informally: “Globalizing” the FIFO-property

(Similarly as FIFO respects causality on a single channel, causal order respects causality in general)

Formal requirement: $\forall (s,r), (s',r'): s < s' \implies \neg(r' < r)$.

Equivalent characterizations:

1) “Triangle inequality”: No message is bypassed by a chain of other messages.

- NB: This implies FIFO.

2) “Empty interval”: $\forall (s,r): \neg \exists x: s < x < r$.

- Cf. similar property on *linear extensions* for *synchronous* communications.

3) “Weakly instantaneous”: \forall messages $m \exists$ space-time diagram where m is a vertical arrow.

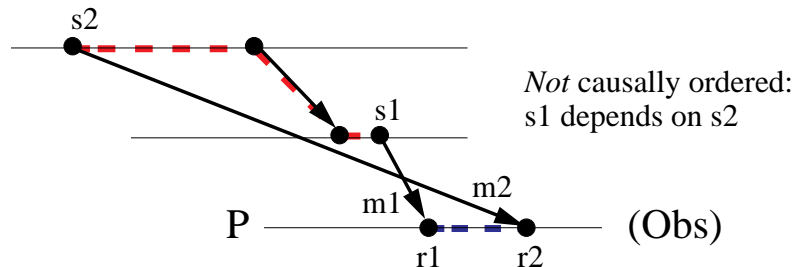
- Cf. “all vertical arrows” property of *synchronous* communications.

- Interpretation: For each (single) message it is possible to claim that this message was transmitted instantaneously.

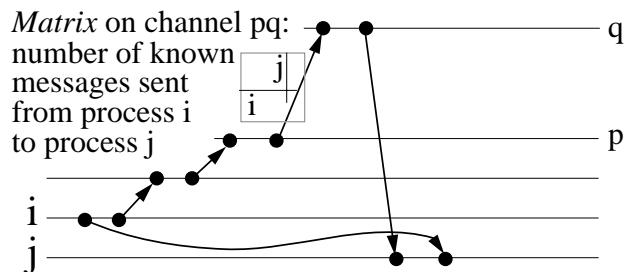
Problem: What are appropriate generalizations for multicast / broadcast?

Causal Order Message Delivery Problem

- Message delivery preserves the causality relation.
- Problem is related to realization of causally consistent observers.

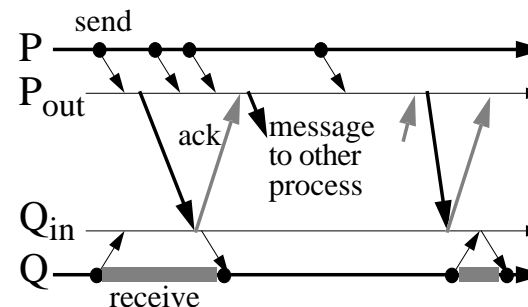


- A message is only delivered to process P if all causally preceding messages (w.r.t. send events) sent to the same process have already been delivered.
- No overtaking of a single message by a chain of messages \implies “Global FIFO property”.
- Canonical realization: *vector of vectors* (“matrix clock”)
 - Each process is a causally consistent observer w.r.t. send events of messages addressed to it.
 - Use scheme for causal observer with n vector timestamps of length n.



Causality Preserving Message Delivery *without* Vector Time

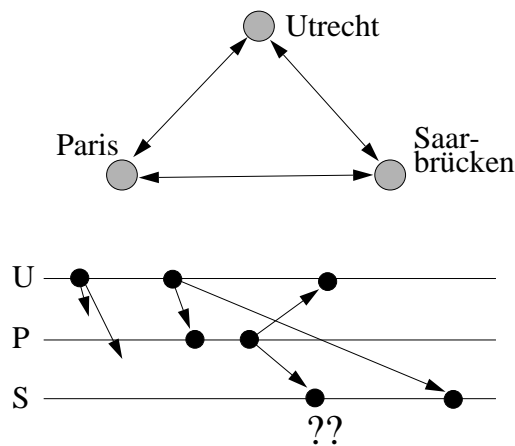
- Each process P has a FIFO-buffer P_{out} and P_{in} .
 - With “send”, the message is handed over to the output buffer P_{out} . P_{out} is then responsible for transmitting the message to the receiver.
 - When executing “receive”, the input buffer P_{in}
 - returns the oldest message if the buffer is not empty,
 - otherwise it blocks P until a message is available.



Rule:
An output buffer waits for an acknowledgment (from the input buffer) before transmitting the next message.

- Sender and receiver are decoupled.
- Because buffers are FIFO and communicate by a handshake protocol, no indirect message overtaking is possible.
 - \implies Correct and efficient implementation of causal order message delivery.


Causal Broadcast



- Confusion because indirect communication was sometimes faster than direct communication.

- Solution: Each participant is a consistent observer of all relevant events.

Date: Fri, 3 Nov 89 16:46:55 +0100
 From: Bernadette Charron <charron@...fr>
 To: mattern

DATE : (101,5,5) ← 
 Bonjour a tous,
 Me revoila...

Au fait, avec vos estampilles vectorielles, les processus ``lents`` sont tout de suite detectes... On ne peut plus dormir en silence, sans etre repere, a moins d'accuser le reseau.

Comme j'ai BEAUCOUP reflechi, je rajoute 100 actions internes pour ma composante.

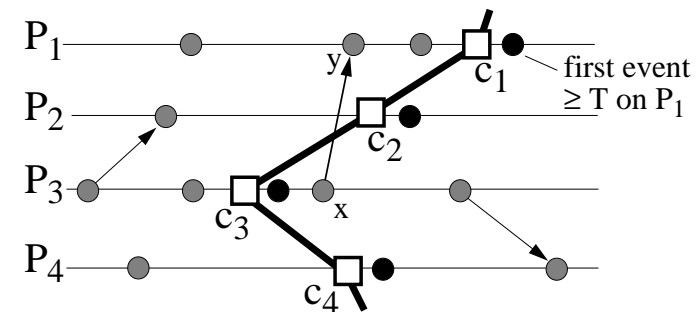
Implementing Snapshots with Vector Time

Idea: Population census paradigm:

- Agree on a common time instant T (well in the future)
- Each process takes its local snapshot at T

==> Does this work with *logical* time?

- Consider the locally first events ● with a timestamp $\geq T$.
- Take a local snapshot □ just before these events.



- A message $x \rightarrow y$ from the “future” to the “past” of the cut line does not exist: $\tau(y) > \tau(x) \geq T$ contradicts the assumption that no event before c_1 has a timestamp $\geq T$.
- Hence the cut is *consistent*!

Choosing the Snapshot Time

Strategy:

- Initiator fixes T and distributes it (wave algorithm, broadcast,...) to all processes.
- Each process takes a local snapshot *just before* its clock jumps to a value $\geq T$.
↑ cf. time leaps when DST starts!

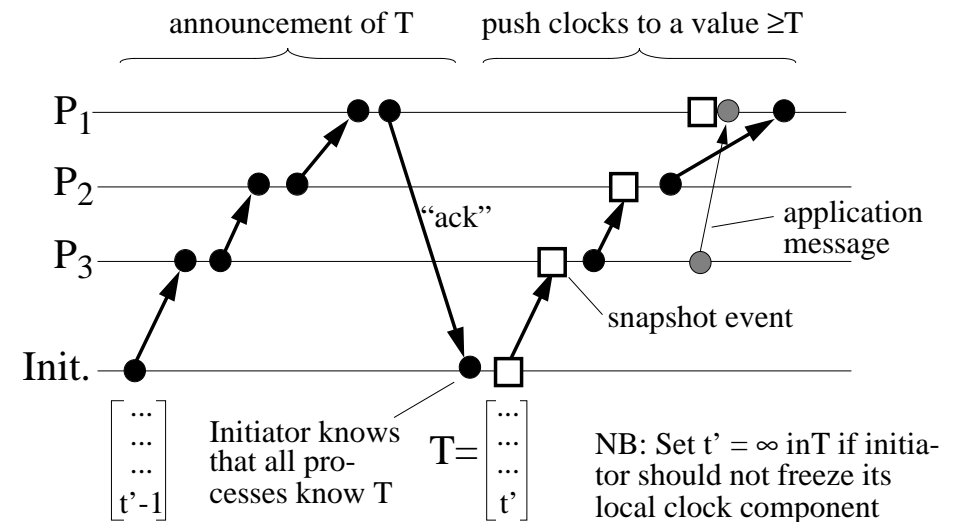
Problems:

- (1) Eventually, each local clock must reach or bypass T . liveness
- (2) Processes must learn about T “in time”. safety
(i.e., before T happens!)

Solutions:

- (1) Initiator increments its clock to vector time T and sends messages (wave...) to all processes. The timestamping scheme automatically pushes all “late” clocks to a value $\geq T$.
- (2) Using vector clocks:
 - Initiator sets $T :=$ timestamp of its *next* event.
(Or it sets its own component in T to ∞ , which will “never” be reached)
 - Initiator announces T to all processes.
 - Initiator does not set its clock to T (according to (1)) until it learns (by acknowledgments, wave...) that all processes know T .

The Snapshot Scheme

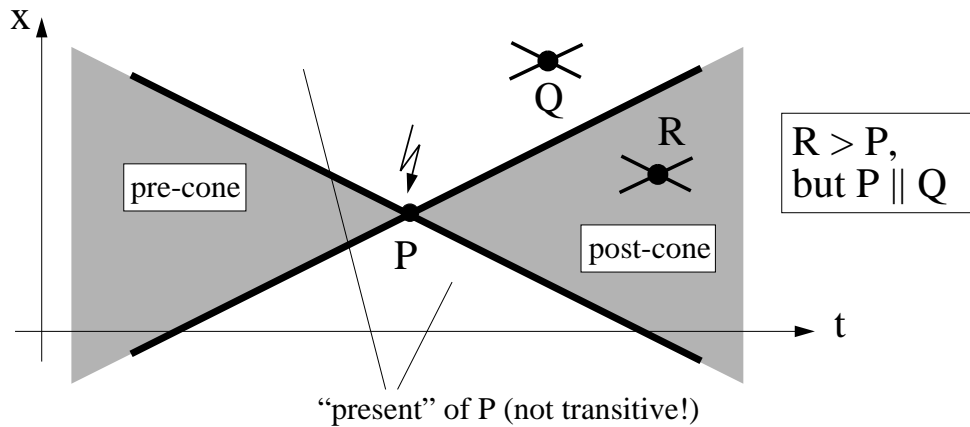


- When a process learns about T , its clock is not yet $\geq T$.
 - Why doesn't it work with Lamport clocks (without freezing)?
 - What about real-time clocks? (Bounds on message delay times?)
 - Scheme can be simplified and optimized!
 - Only last component of vector clocks is relevant.
 - Binary time (black / red) is sufficient.
 - Single wave suffices (if all processes initially know T)
- ==> Yields the snapshot algorithm presented earlier!

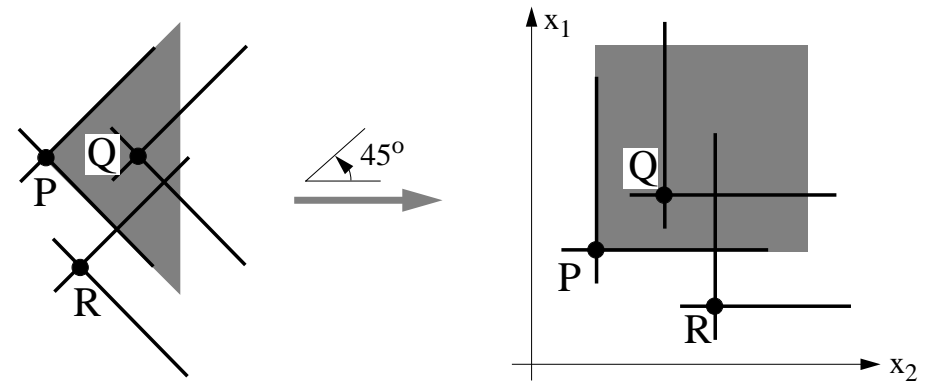
Using vector time and a well known protocol from our “distributed real world” yields a consistent snapshot scheme!

(Vector time is a good substitute for real time)

Vector Time and Minkowski's Space-Time



Lightcone Order and Vector Time Order



90° light cones (normalize the maximum speed to “1 space unit per time unit”, e.g., “light year / year”)

$X=(x_1,x_2)$, $Y=(y_1,y_2)$
vectors = coordinates of the points

space-time

Partial order

2-dimensional cones build a lattice (w.r.t. intersection)

Lorentz-transformation leaves light cone invariant

Space time coordinates enable to test for (potential) causal relationship: with $u=(x_1, t_1)$, $v=(x_2, t_2)$ check $c^2(t_2-t_1)^2 - (x_2-x_1)^2 \geq 0$

vector time

Partial order

Time vectors build a lattice (sup) (cuts also w.r.t. inclusion)

Rubber band transformation leaves causality relation invariant

Time vectors enable a simple test, whether two events are (potentially) causally dependent (check, whether in all components smaller)

- Light cone of Y fully contained in the light cone of X (left picture) $\Leftrightarrow x_1 < y_1 \wedge x_2 < y_2$ (right picture) $\Leftrightarrow (x_1,x_2) < (y_1,y_2) \Leftrightarrow X < Y$.

\implies 2 dimensional cones \approx 2 dimensional cubes

\implies At least for 2 dimensions, space-time and vector time have essentially the same structure!

- “later”
- potential causality
- lattice structure

Space-time / vector time yield a more accurate view of our distributed world than “standard time”!

Bibliography (Selected Items)

Friedemann Mattern
FB 20 - Dept. of Computer Science
Technical University of Darmstadt
Alexanderstr. 6
D 64283 Darmstadt
Germany

email: mattern@informatik.th-darmstadt.de

Most papers (and abstracts) by the author are available at:
<http://www.informatik.th-darmstadt.de/VS/Publikationen.html>

Postscript copies of the slides will be available at:
<http://www.informatik.th-darmstadt.de/VS/pub/slides/siena95.ps>

F. Mattern: Virtual Time and Global States of Distributed Systems. In: Cosnard M. et al. (eds): Proc. Workshop on Parallel and Distributed Algorithms, North-Holland / Elsevier, pp. 215-226, 1989.

F. Mattern: Über die relativistische Struktur logischer Zeit in verteilten Systemen. In: J. Buchmann, H. Ganzinger, W.J. Paul (Eds.): Informatik -Festschrift zum 60. Geburtstag von Günter Hotz, Teubner, pp. 309-331, 1992. English translation "On the Relativistic Structure of Logical Time in Distributed Systems" is available from the author.

R. Schwarz, F. Mattern: Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail. Distributed Computing 7:3, 149-174, 1994.

B. Charron-Bost, F. Mattern, G. Tel: Synchronous, Asynchronous, and Causally Ordered Communication. Technical Report TR-VS-95-02, Department of Computer Science, Technical University of Darmstadt, 1995 (to be published in Distributed Computing).

F. Mattern, H. Mehl, A. Schoone, G. Tel: Global Virtual Time Approximation with Distributed Termination Detection Algorithms. Technical Report RUU-CS-91-32, Department of Computer Science, University of Utrecht, 1991.

F. Mattern: Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation. Journal of Parallel and Distributed Computing 18:4, pp. 423-434, 1993.

"Global States and Time in Distributed Systems", edited by Z. Yang und T.A. Marsland (IEEE Computer Society Press, 1994), contains a collection of reprinted papers and conference contributions.

"Distributed Systems (second edition)", edited by S. Mullender (Addison-Wesley, 1993), contains the paper "Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms" (pp. 55-96) by Ö. Babaoglu and K. Marzullo.

Robert H. B. Netzer and Barton P. Miller: Optimal Tracing and Replay for Debugging Message-Passing Parallel Programs. Brown University, Department of Computer Science, TR CS-94-32, 1994, <ftp://ftp.cs.brown.edu/pub/techreports/94/cs94-32.ps.Z>

"Session Summaries". Proceedings of ACM/ONR Workshop on Parallel and Distributed Debugging, ACM SIGPLAN Notices 18:12, pp. vii-xix, 1993.

D.R. Jefferson: Virtual Time. ACM TOPLAS 7:3, pp. 404-425, 1985.

R. M. Fujimoto: Parallel Discrete Event Simulation. Commun. of the ACM 33:10, pp. 30-53, 1990

*Most of the author's papers are available via WWW:
<http://www.informatik.th-darmstadt.de/VS/Publikationen.html>
(or send an email to mattern@informatik.th-darmstadt.de).*

The End