Time and Global States

Time is money

Benjamin Franklin

Distributed software systems

CdLM Informatica - Università di Bologna

Time in distributed systems

Time is a challenging issue in distributed computing.

Consider these examples:

I. Suppose we want to build a distributed system to track the battery usage of a set of laptop computers and we'd like to record the percentage of the battery each has remaining at exactly 2pm.

II. Suppose we want to build a distributed, real time auction and we want to know which of two bidders submitted their bid first.

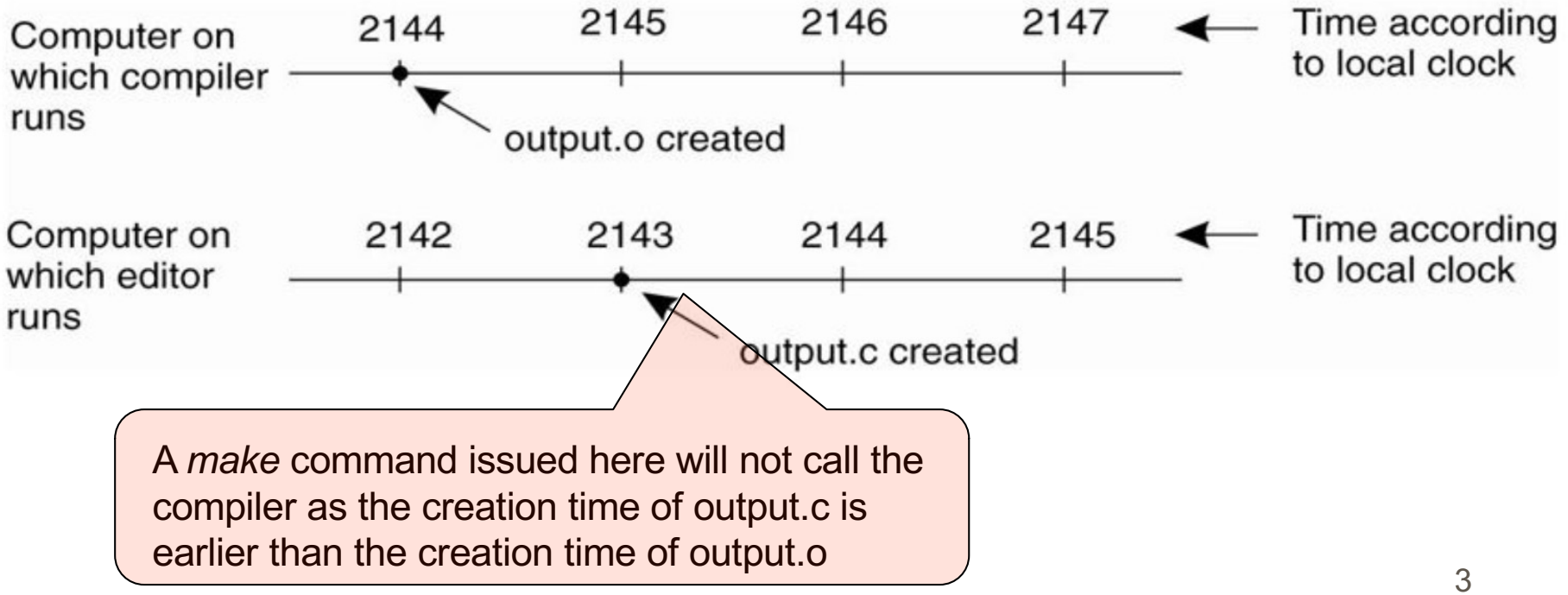
III. Suppose we want to debug a distributed system and we want to know whether variable x_1 in process p_1 ever differs by more than 50 from variable x_2 in process p_2.

In the first example, we would like to synchronize the clocks of all participating computers and take a measurement of absolute time.

In the second and third examples, knowing the absolute time is not as crucial as knowing the order in which events occurred.

Question: what is time in a distributed system?

- Time is not ambiguos in a centralized system: a unique system clock keeps time, all entities use this clock
- Each node in a distributed system has its own local clock: an event that occurred after another event may be assigned an earlier time



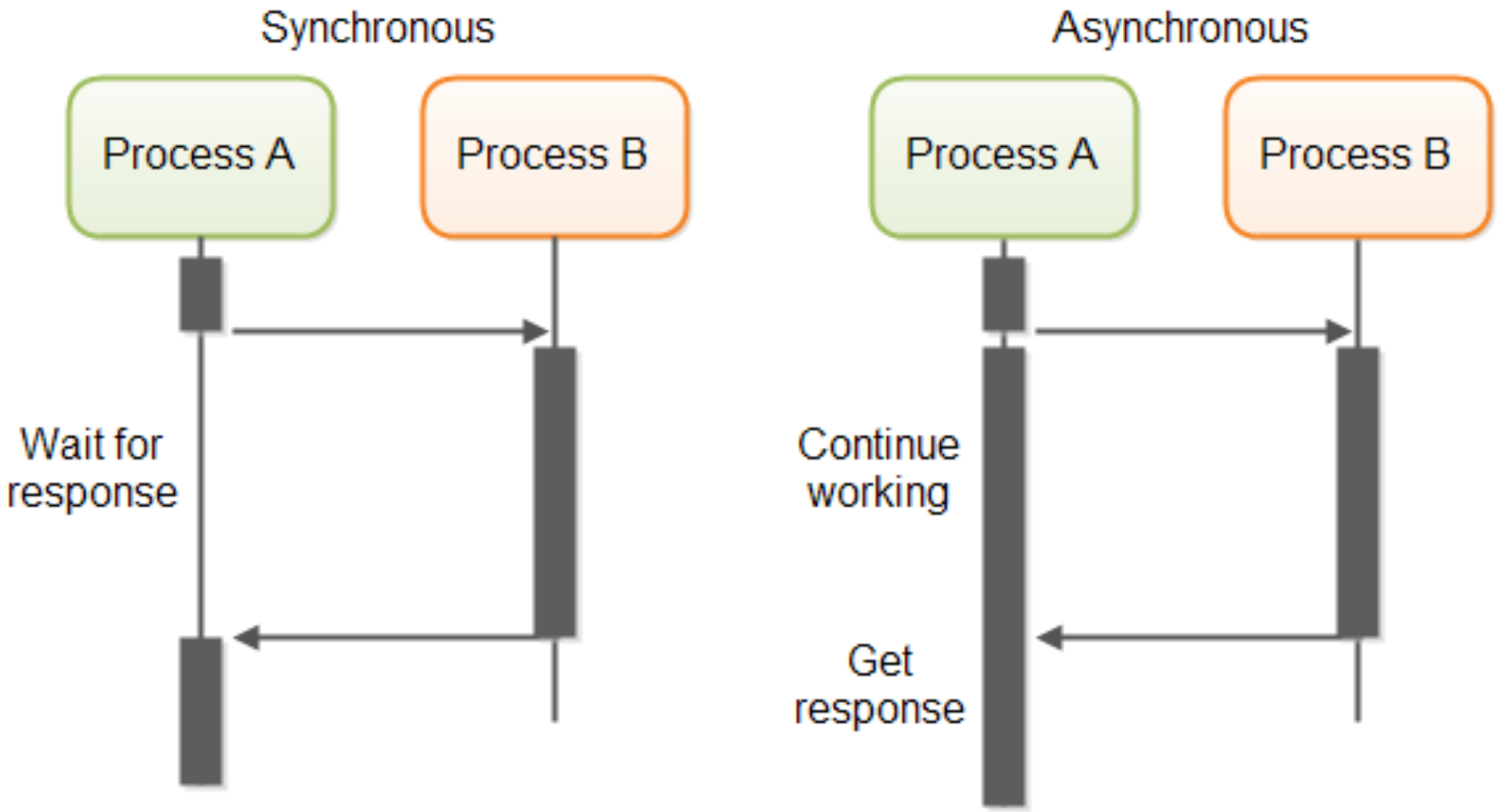
Synchrony and asynchrony

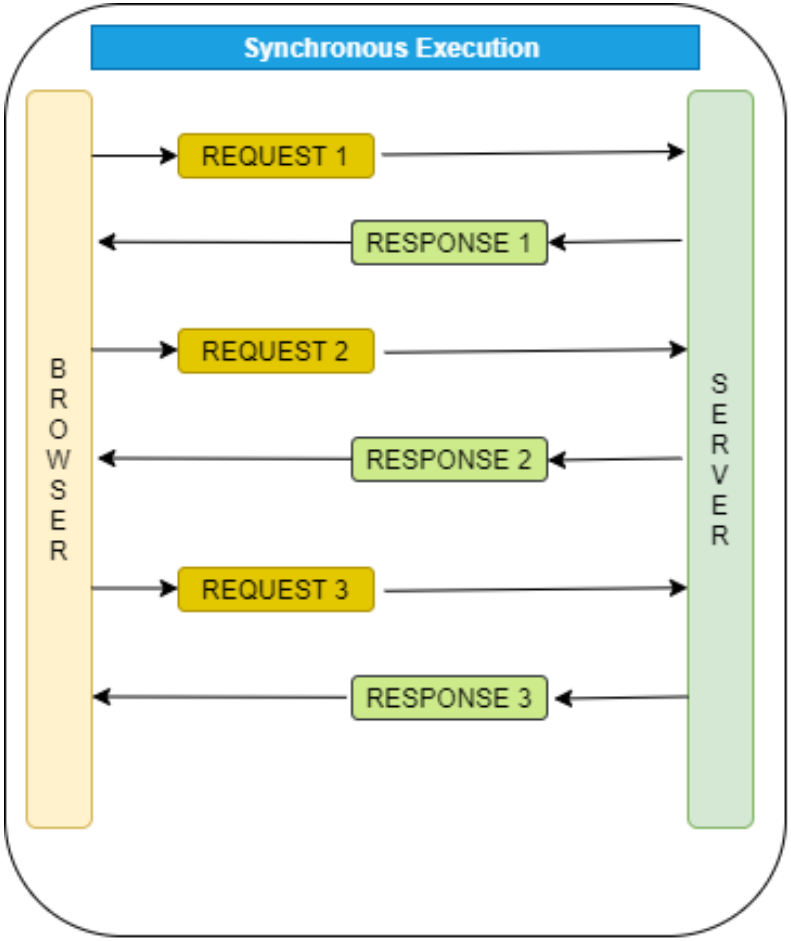
There are two formal models of time for distributed systems:
synchronous and asynchronous.

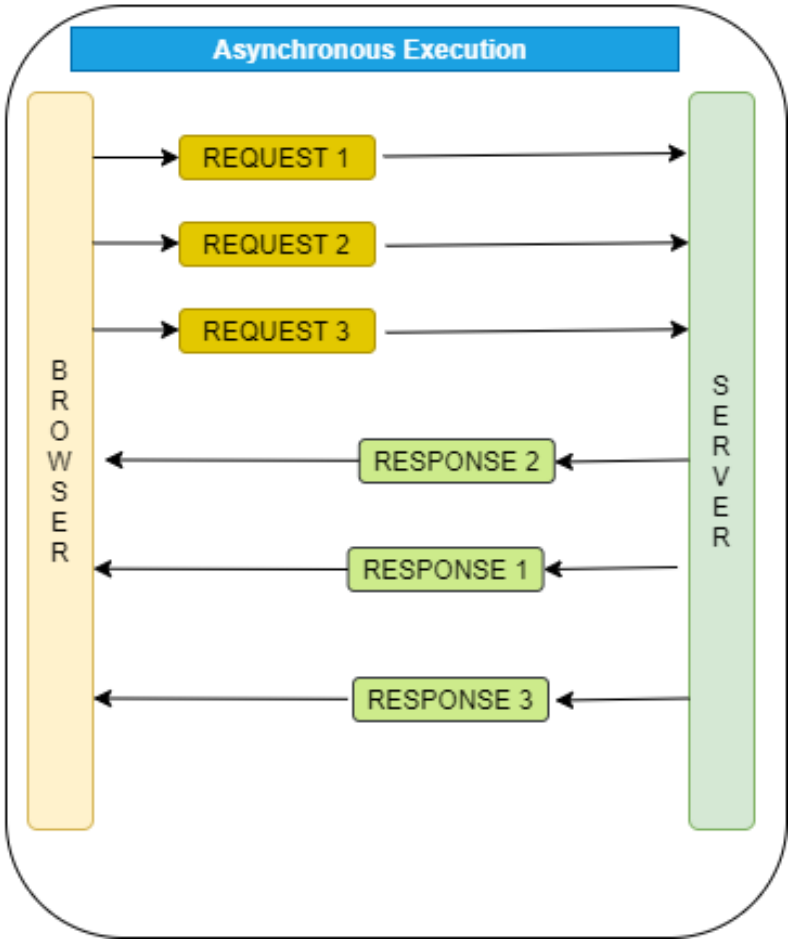
Synchronous means: using the same clock

when two instructions are synchronous they use the same clock and must happen one after the other.

"Asynchronous" means "not using the same clock" so the instructions are not concerned with being in step with each other.


Synchronous
Process A
Process B
Wait for response
Asynchronous
Process A
Process B
Continue working
Get response





In most applications, the UI system displays components which are not dependent on each other in order to be loaded. In such cases, synchronous programming takes more time to load than respective page completely, whereas asynchronous programming loads the page irrespective of remaining or ongoing asynchronous tasks.

Synchrony and asynchrony

Synchronous distributed systems have the following characteristics:

1. the time to execute each step of a process has known lower and upper bounds;
2. each message transmitted over a channel is received within a known bounded time;
3. each process has a local clock whose drift rate from real time has a known bound.

Asynchronous distributed systems, in contrast, guarantee no bounds on process execution speeds, on message transmission delays, or on clock drift rates.

Most distributed systems, including the Internet, are asynchronous systems.

Agenda

Clocks, events and process states

Synchronizing physical clocks

Logical time and logical clocks

Global states

Debugging distributed systems

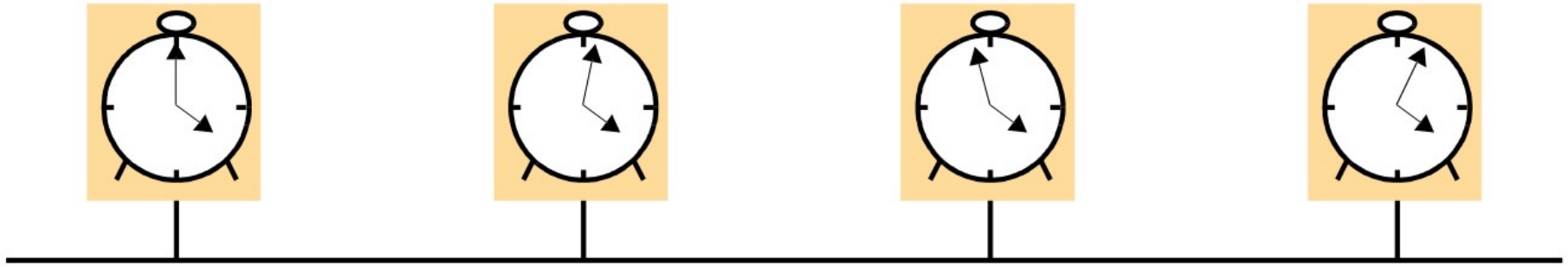
Time in distributed systems

- Time is important in our every day life: for instance, we need to timestamp e-commerce transactions consistently.
- Time is also an important theoretical construct in understanding how distributed executions unfold.
- Time is problematic: each computer has its own physical clock, but these clocks deviate, and we cannot synchronize them perfectly.
- Algorithms for synchronizing physical clocks are approximate
- The absence of global physical time makes it difficult to find out the global state of a distributed program during its execution

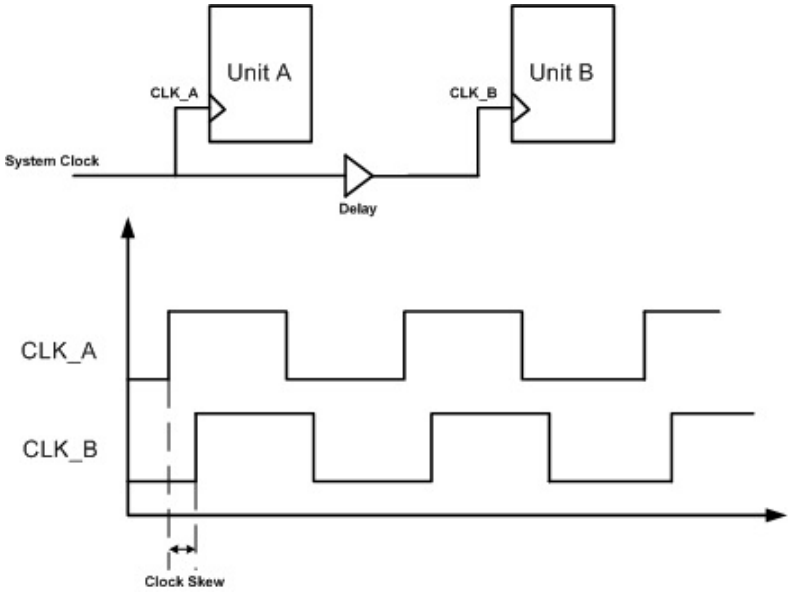
Clock skew

- **Clock skew** happens when a clock signal arrives at different components at different times i.e. the instantaneous difference between the readings of any two clocks is called their *skew*.
- Clock skew can be caused by many different things, such as wire-interconnect length, temperature variations, variation in intermediate devices, capacitive coupling, material imperfections, and differences in input capacitance on the clock inputs of devices using the clock.
- As the clock rate of a circuit increases, timing becomes more critical and less variation can be tolerated if the circuit is to function properly.

Skew between computer clocks in a distributed system



Network

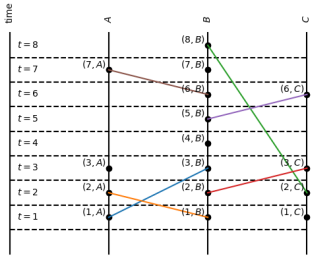


Three notions of time in a distributed system

- Time seen by an external observer: a global clock of ideally perfect accuracy (we call this: **absolute time**)

- Time seen on clocks of individual processes: Each process has its **local time**; local clocks may drift out of sync.

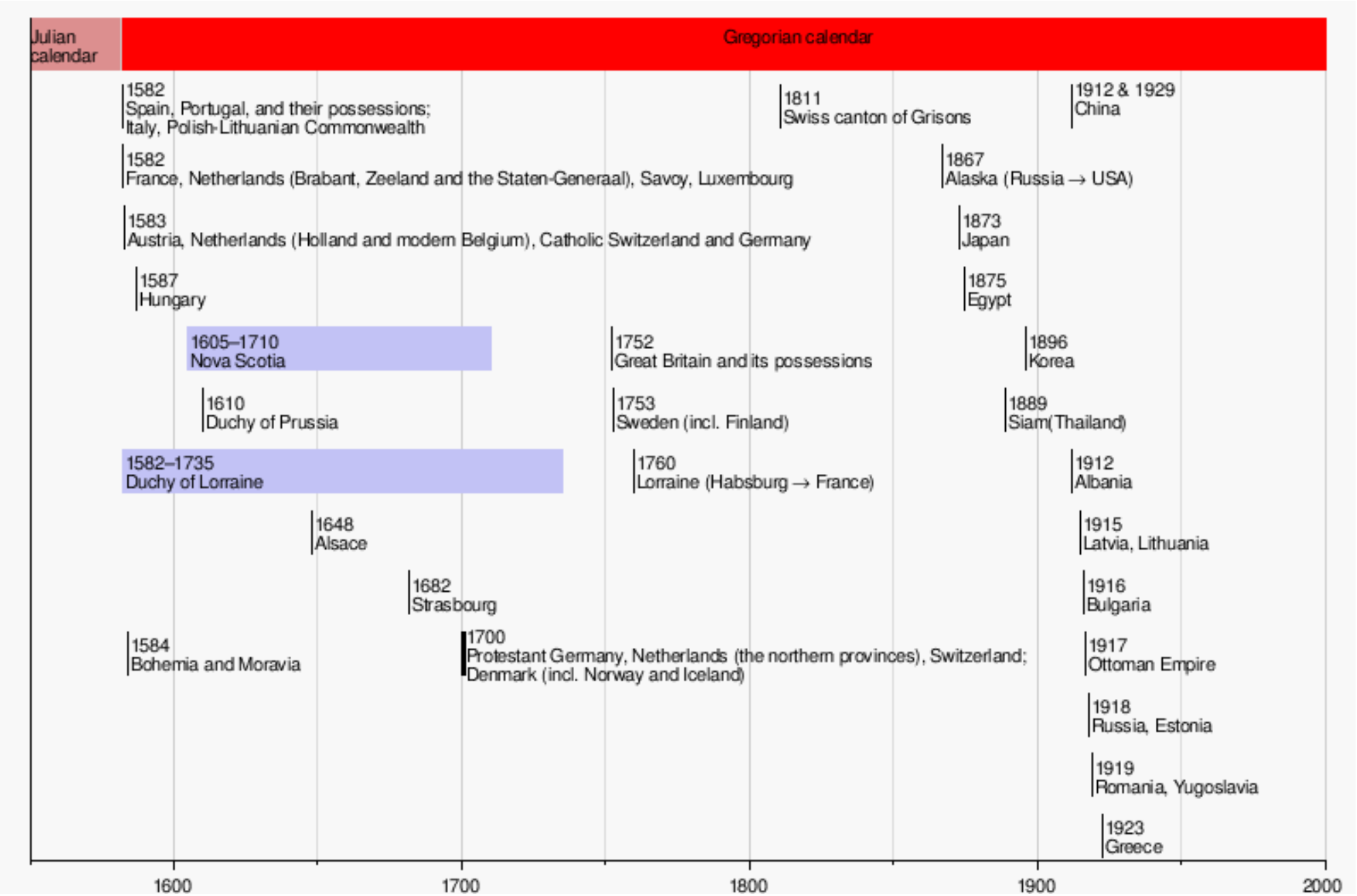
- Logical time: event *a* occurs before event *b* and this ordering is detectable, because information about *a* may have reached *b:* logical clocks are a tool for ordering events without knowing precisely when they occurred.



Absolute time

- According to Newton, **absolute time** (also known as “Newtonian time”) exists independently of any perceiver, progresses at a consistent pace throughout the universe, is measurable but imperceptible, and can only be truly understood mathematically.
- For Newton, time and space were independent and separate aspects of reality, and not dependent on physical events or on each other.
- Einstein denied independence of space and time (see the Theory of Relativity)

Adoption years of the Gregorian calendar in different countries



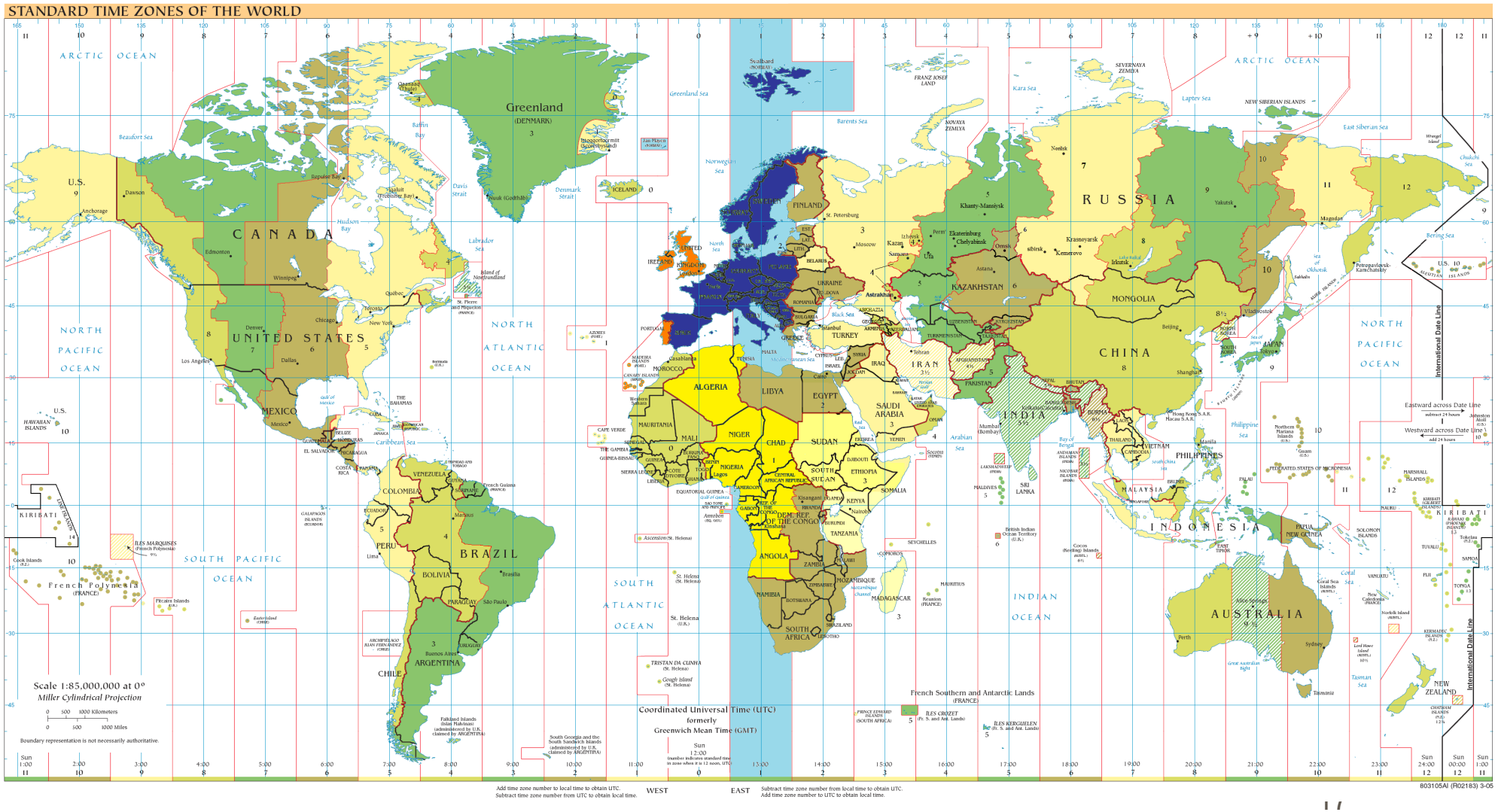
UTC Coordinated Universal Time

- UTC is the primary time standard by which the world regulates clocks and time.
- It is within about 1 second of mean solar time at 0° longitude; it does not observe daylight saving time.
- For most purposes, UTC is considered interchangeable with Greenwich Mean Time (GMT), but GMT is no longer precisely defined by the scientific community

Leap seconds in UTC

- UTC divides time into days, hours, minutes and seconds.
- Days are conventionally identified using the Gregorian calendar
- Each day contains 24 hours and each hour contains 60 minutes.
- Nearly all UTC days contain exactly 86,400 SI seconds with exactly 60 seconds in each minute.
- However, because the mean solar day is slightly longer than 86,400 SI seconds, occasionally the last minute of a UTC day is adjusted to have 61 seconds. The extra second is called a leap second.
- The number of seconds in a minute is usually 60, but with an occasional leap second, it may be 61 or 59 instead.
- Decisions to introduce a leap second are announced at least six months in advance
- The leap seconds cannot be predicted far in advance due to the unpredictable rate of rotation of the Earth.

UTC+1 (from Wikpedia)



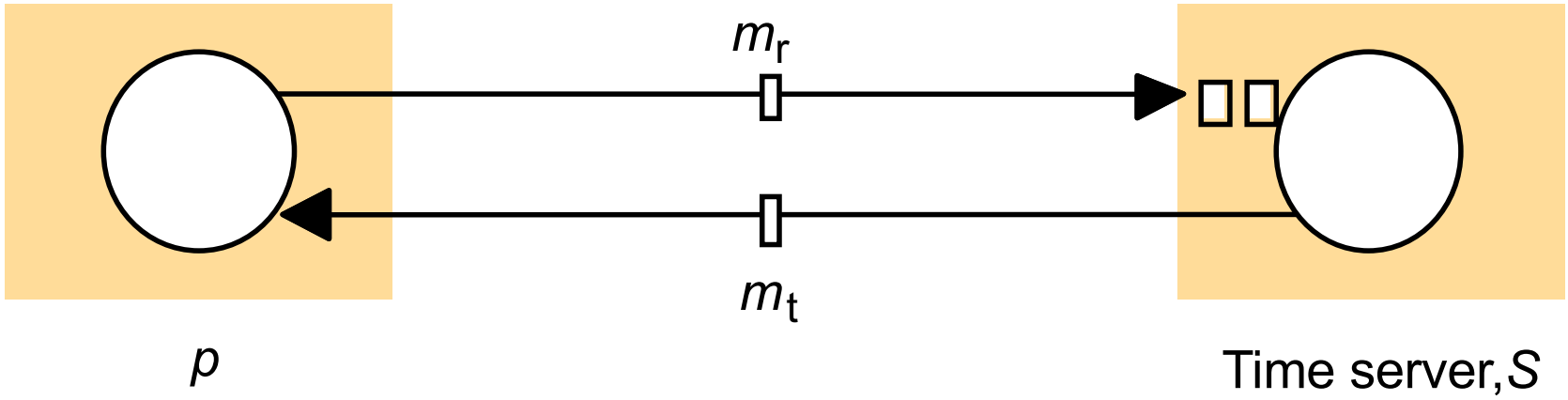
Absolute time is external time for a distributed system

- “Absolute time” is the “gold standard” against which many protocols are defined
- Beware: It is not implementable!
- no system can avoid uncertain details that limit temporal precision
- Moreover, the use of external absolute time is risky: many protocols that require properties defined by external observers are *extremely* costly and, sometimes, are unable to cope with failures

Time seen on internal clocks

- Most computing devices have local clocks
- Clock synchronization among different devices is a big problem: clocks can drift apart and resynchronization, in software, is inaccurate
- Unpredictable speed is a feature of all computing systems, hence cannot predict how long events will take (e.g. how long it will take to send a message and be sure it was delivered to the destination)

Clock synchronization using a time server



Cristian [1989] suggested the use of **a time server**, connected to a device that receives signals from a source of UTC, to synchronize computers externally

Upon request, the server process S supplies to p the current time according to its (S's) clock

Christian's algorithm

- Cristian's algorithm is a method for clock synchronization which can be used in low-latency intranets
- It works between a process P, and a time server S — connected to a source of UTC (Coordinated Universal Time).
  1. P requests S the time
  2. After receiving the request from P, S answers the time T from its own clock.
  3. P then sets its local clock to be T + RTT/2
- This algorithm assumes that the RTT is split equally between request and response, which is a reasonable assumption on a LAN connection
- Further accuracy can be gained by making multiple requests to S and using the response with the shortest RTT
- The algorithm is probabilistic, in that it only achieves synchronization if the round-trip time (RTT) of the request is short compared to the required accuracy
- It also suffers in implementations using a single server, making it unsuitable for many distributed applications where redundancy may be crucial

Network Time Protocol NTP

- The Network Time Protocol (NTP – RFC 5905) for clock synchronization works over packet-switched, variable-latency data networks
- NTP is one of the oldest Internet protocols in current use (used since 1985).
- NTP is intended to synchronize all participating devices to within a few milliseconds of Coordinated Universal Time (UTC).
- It uses the *intersection algorithm*, a modified version of Marzullo's algorithm, to select accurate time servers and is designed to mitigate the effects of variable network latency

NTP uses a hierarchical, semi-layered system of time sources.

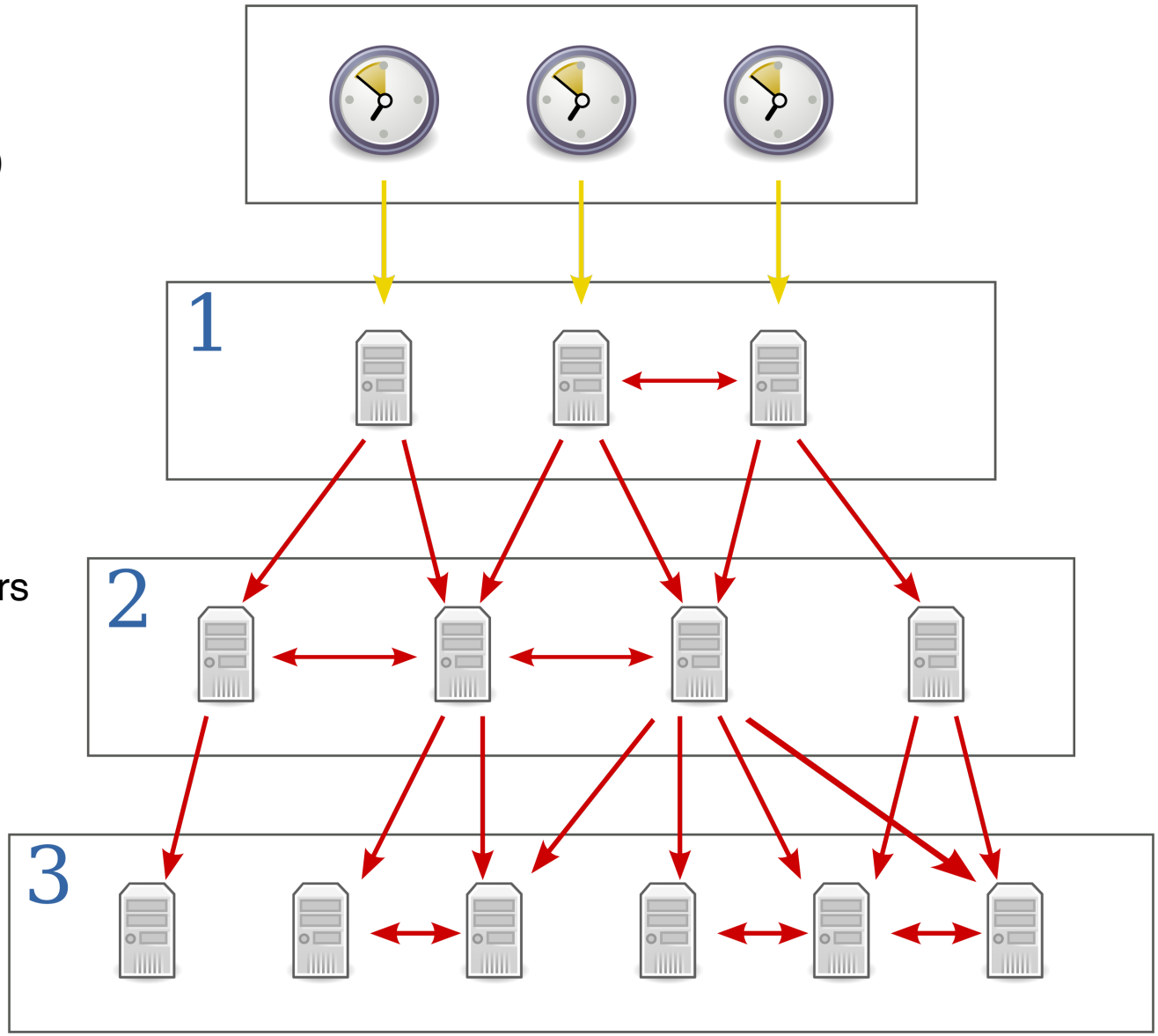
Stratum 0 These are high-precision timekeeping devices such as atomic clocks. Stratum 0 devices are also known as reference clocks.

Stratum 1 These are computers whose system time is synchronized to within a few microseconds of their attached stratum 0 devices. They are also referred to as primary time servers

Stratum 2 These are computers that are synchronized over a network to stratum 1 servers. Often a stratum 2 computer will query several stratum 1 servers

…and so on…

The upper limit for stratum is 15

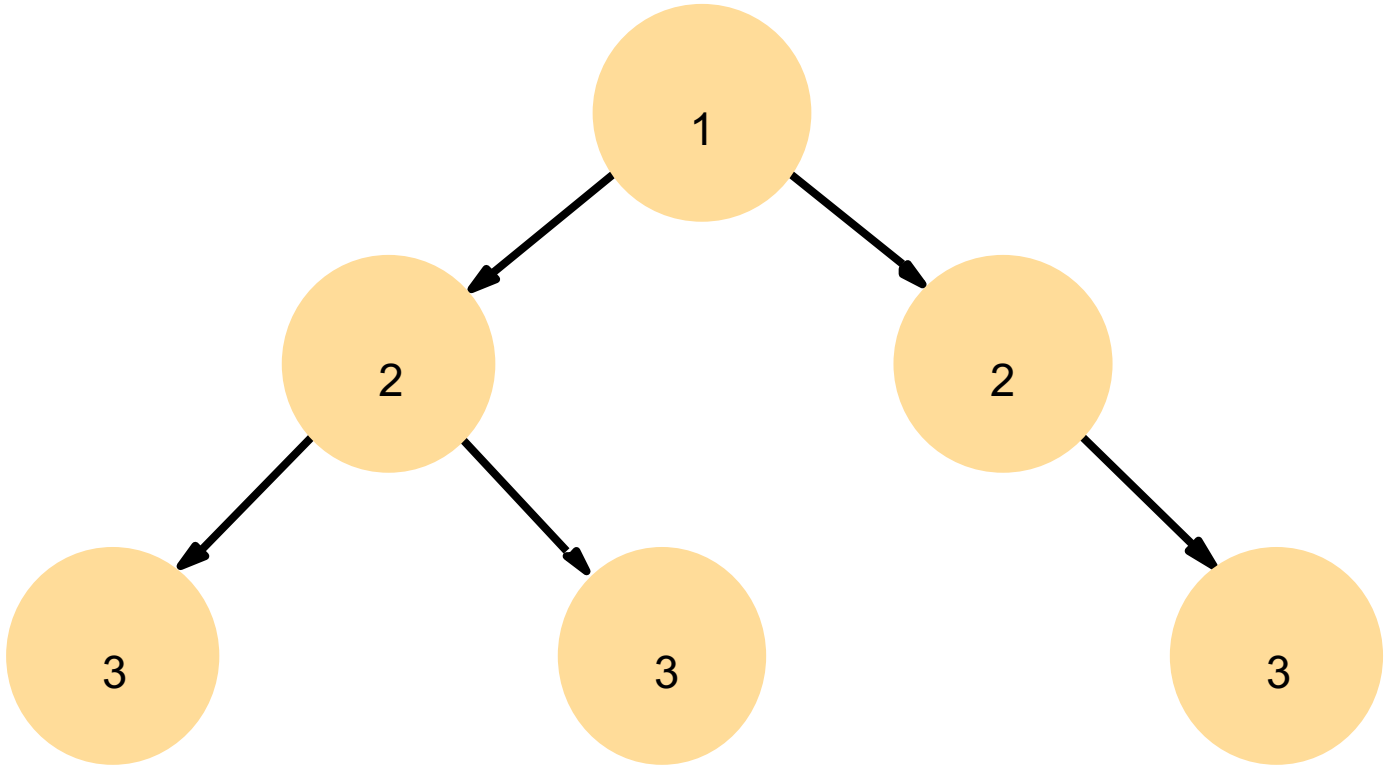


Yellow arrows indicate a direct connection; red arrows indicate a network connection

NTP

- Primary servers are connected directly to a time source such as a radio clock receiving UTC
- Secondary servers are synchronized with primary servers
- The servers are connected in a logical hierarchy called a *synchronization subnet* whose levels are called *strata*
- Primary servers occupy stratum 1: they are at the root.
- Stratum 2 servers are secondary servers that are synchronized directly with the primary servers; stratum 3 servers are synchronized with stratum 2 servers, and so on
- The lowest-level (leaf) servers execute in users' workstations

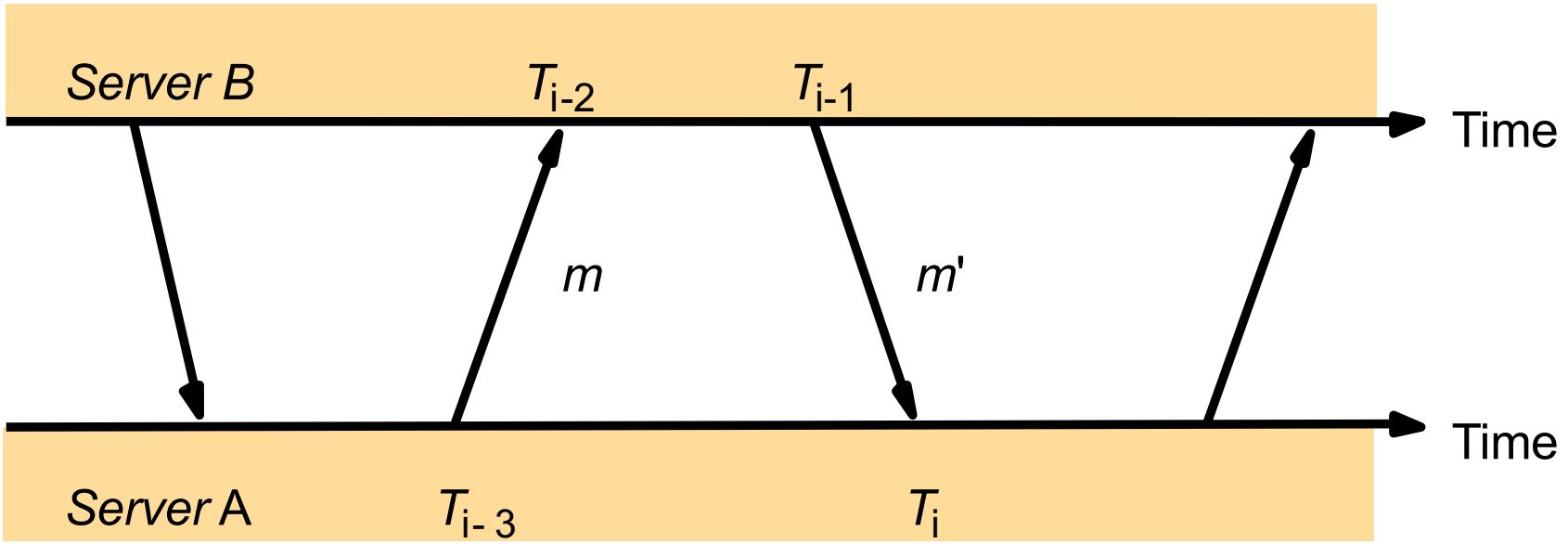
An example synchronization subnet in an NTP implementation



Note: Arrows denote synchronization control, numbers denote strata

Primary servers are connected directly to a time source such as a radio clock receiving UTC; secondary servers are synchronized, ultimately, with primary servers.

Messages exchanged between a pair of NTP peers



Each message bears timestamps of recent message events: the local times when the previous NTP message between the pair was sent and received, and the local time when the current message was transmitted.
The recipient of the NTP message notes the local time when it receives the message.

The mathematics of time in distributed systems

- A distributed system consists of a collection of N processes p_i
- Each process executes on a single processor, and the processors do not share memory; processes can communicate only by sending messages through the network
- The sequence of events within a single process p_i can be placed in a single total ordering, denoted by the relation $\rightarrow_i$ between the events: $e \rightarrow_i e'$ if and only if the event e *occurs before* e' at p_i

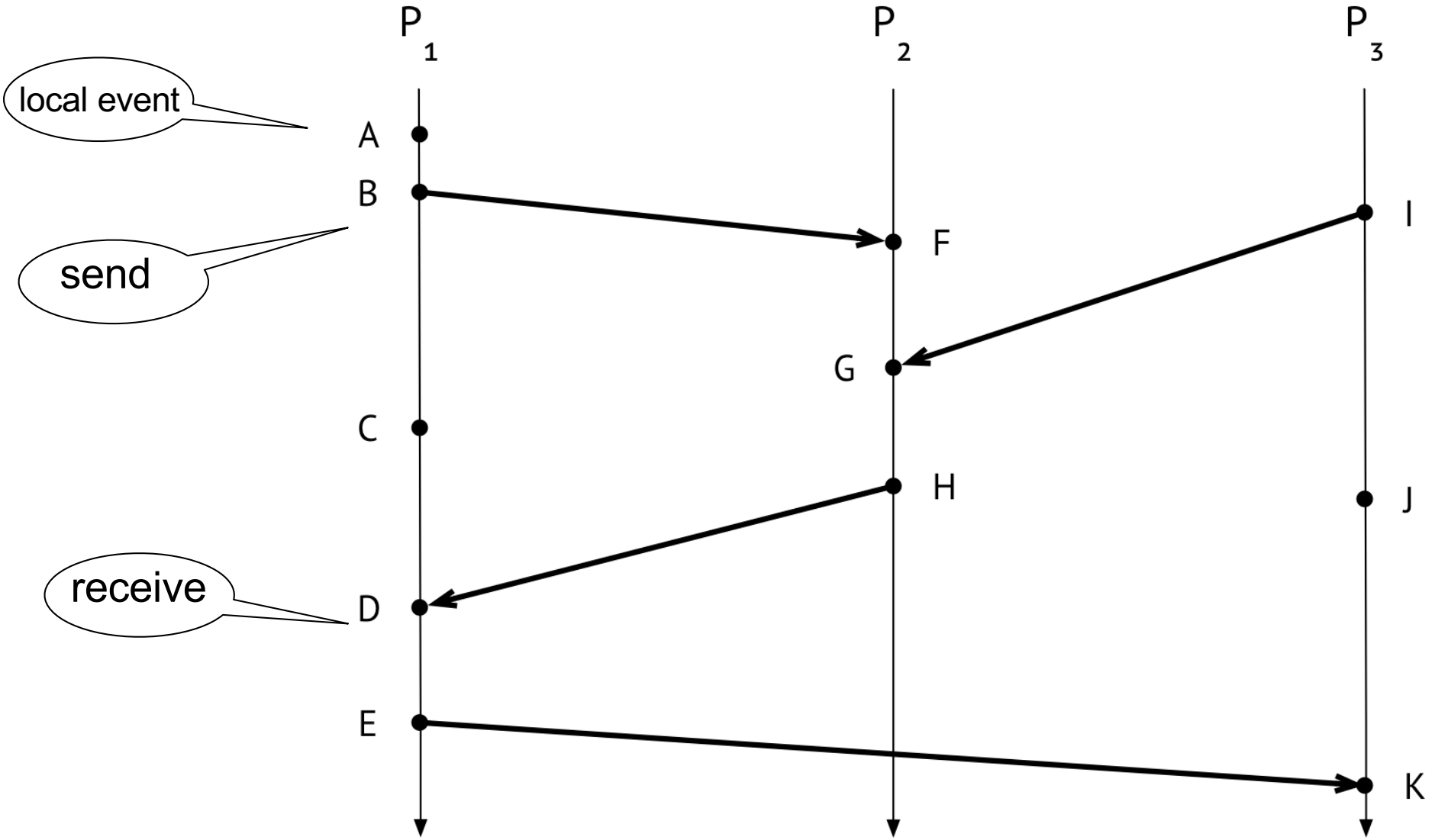
- The *history* of process p_i is the series of events that take place within it, ordered by the relation $\rightarrow_i$:
- *history*(p_i) = $h_i = \langle e_i^0, e_i^1, e_i^2 \ldots \rangle$
- From the point of view of any single process, events are ordered uniquely by the times shown on the local clock.

Lamport's idea

- Every process has a logical clock that is advanced as follows:
  - Every event is assigned a timestamp
  - Timestamps obey the fundamental monotonicity property: if an event e1 causally affects an event e2, then the timestamp of e1 is smaller than the timestamp of e2

- WHAT WE WANT: causality between events can be generally inferred from their timestamps

Example



Causality

Distributed systems are causal, meaning that **the cause precedes the effect**: eg. sending of a message precedes the receipt of the message

Types of events in distributed systems

- local computation
- sending a message
- receiving a message

How to define the *happened-before* relationship between events in a distributed system – in different processes - without using (physical) clocks?

Time as **partial** ordering of events

We can order some of the events that occur at different processes; this partial ordering is based on two rules:

1. If two events occurred at the same process p_i, then they occurred in the order in which p_i observes them – this is the order $\rightarrow_i$.
2. Whenever a message is sent between two processes, the event of *sending* the message occurred before the event of *receiving* the message

this ordering can be thought of as a clock that only has meaning in relation to messages moving between processes: when a process receives a message, it resynchronizes its clock with that sender's clock

happened-before

- the partial ordering obtained by generalizing these two rules is the *happened-before* relation
- we define the *happened-before* relation, denoted by $\rightarrow$:

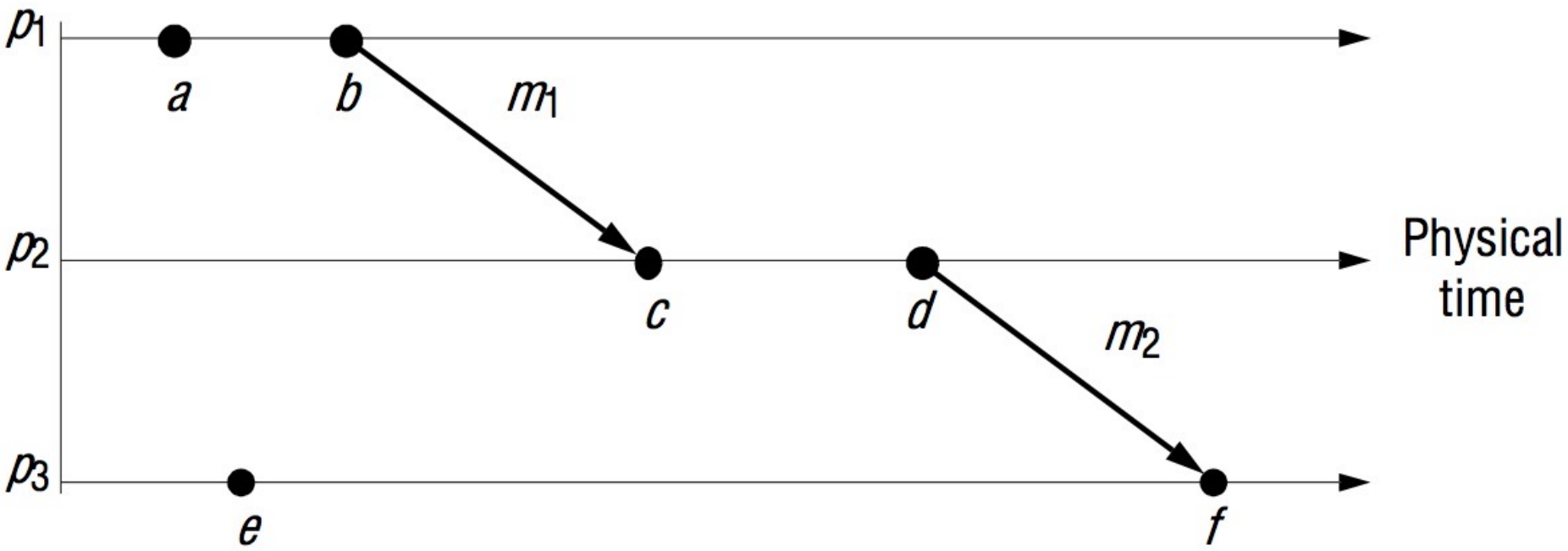
HB1: If $\exists$process $p_i : e \rightarrow_i e'$, then $e \rightarrow e'$.

HB2: For any message m, send(m) $\rightarrow$ receive(m) – where send(m) is the event of sending m, and receive(m) is the event of receiving it.

HB3 (transitivity): If e, e', e'' are events such that

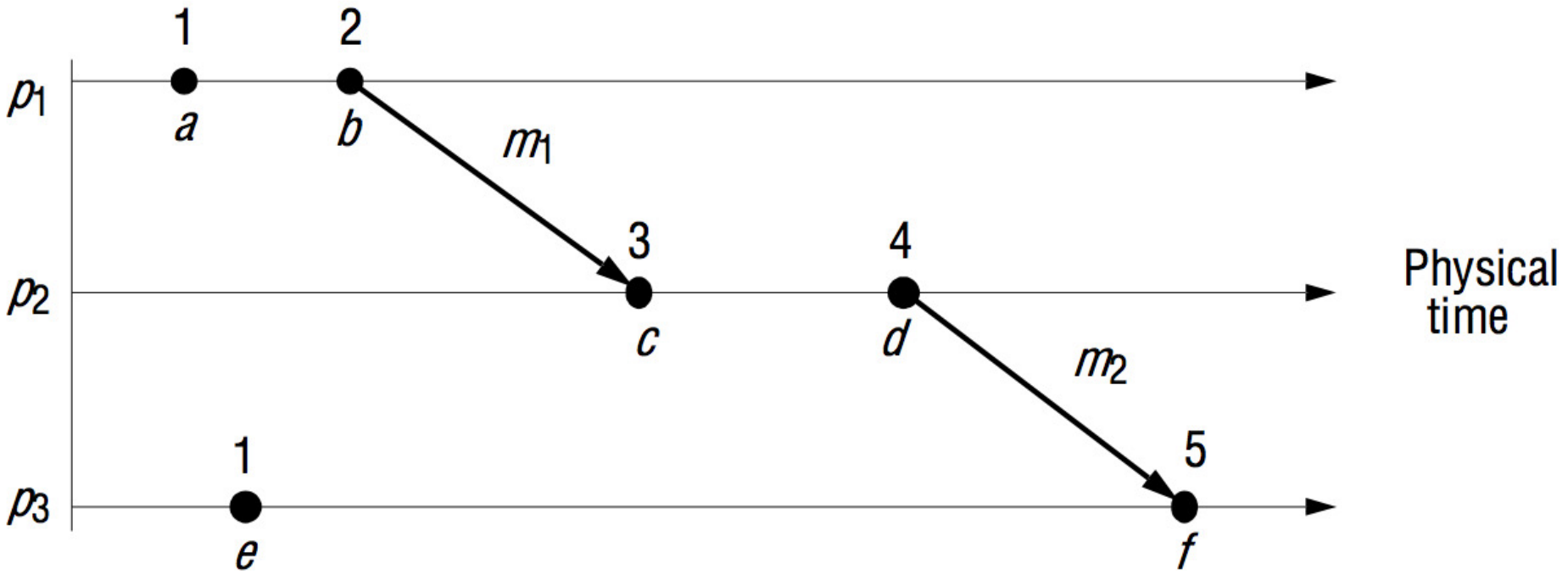
e $\rightarrow$ e' and e' $\rightarrow$ e'' , then e $\rightarrow$ e''

Concurrent events



- Not all events in the system are ordered by the relation $\rightarrow$.
- In the above example, $a \nrightarrow e$ and $e \nrightarrow a$, since they occur at different processes, and there is no chain of messages between them
- We say that events such as *a* and *e* that are not ordered by $\rightarrow$ are concurrent and write this: $a \parallel e$

Lamport timestamps for the events: logical clocks



LC1: L_i is incremented before each event issued at process p_i

LC2 a): When a process p_i sends a message m_i it adds the value $t=L_i$

LC2 b): On receiving (m,t) a process p_j computes $L_j:=\max(L_j,t)$ then applies LC1 before timestamping the event receive(m)

The "Happens-Before" Relation (4)

The question to ask is:

How can some event that "happens-before" some other event possibly have occurred at a later time?

The answer is: it can't!

So, Lamport's solution is to have *the receiving process adjust its clock forward to one more than the sending timestamp value*.

This allows the "happens-before" relation to hold, and also keeps all the clocks running in a synchronised state. The clocks are all kept in sync *relative to each other*.

Vector clocks

- A vector clock for a system of N processes is an array of N integers.
- Each process keeps its own vector clock, V_i , which it uses to timestamp local events.
- Processes add vector timestamps on the messages they send to one another; these are the rules for updating the clocks:

  VC1: Initially, $V_i[j] = 0$, for all processes .

  VC2: Just before p_i timestamps an event, it sets $V_i[i] := V_i[i] + 1$.

  VC3: p_i includes the value $t = V_i$ in every message it sends.

  VC4: When p_i receives a timestamp t in a message, it sets for all j
  $V_i[j] := \max(V_i[j], t[j])$.
- Taking the component-wise maximum of two vector timestamps in this way is known as a *merge* operation

Comparing vector timestamps

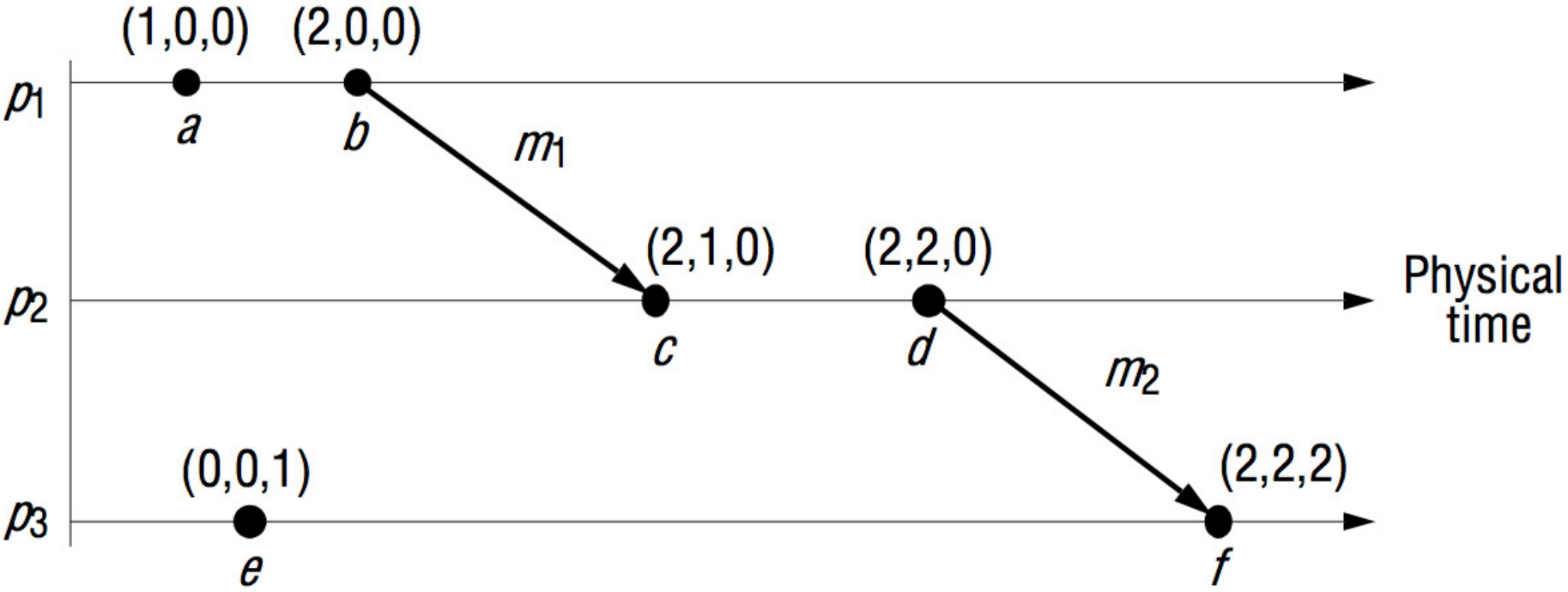
We may compare vector timestamps as follows:

$$V = V' \;\; iff \;\; V[j] = V'[j] \;\; \text{for } j = 1, 2 \ldots, N$$

$$V \leq V' \;\; iff \;\; V[j] \leq V'[j] \;\; \text{for } j = 1, 2 \ldots, N$$

$$V < V' \;\; iff \;\; V \leq V' \wedge V \neq V'$$

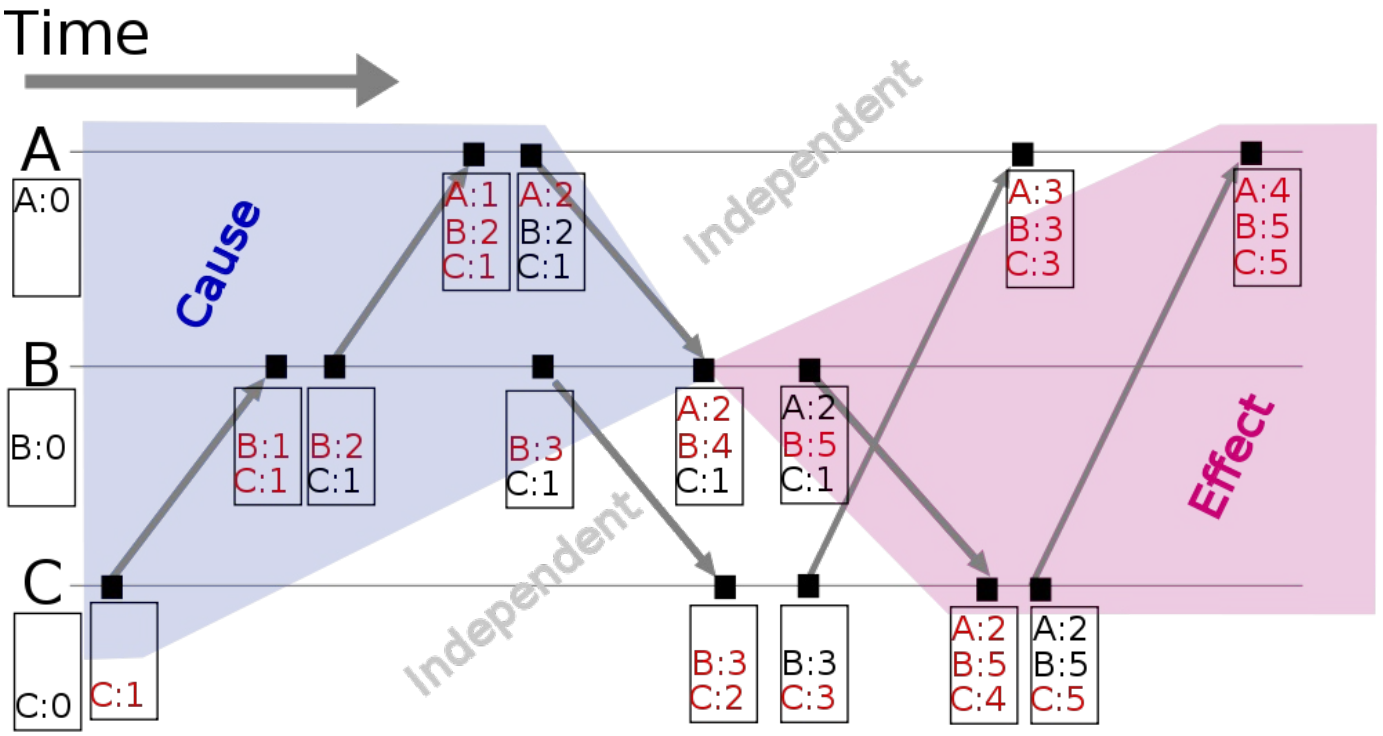
Vector timestamps for the events (slide32)



- We see that V(*a*) < V(*f*), which reflects the fact that $a \rightarrow f$
- We tell when two events are concurrent by comparing their timestamps
- We conclude $c \parallel e$ from the facts that neither $V(c) \leq V(e)$ nor $V(e) \leq V(c)$

Vector clocks (from Wikipedia)

A vector clock of a system of N processes is an array of N logical clocks, one per process
It is used for labeling a partial ordering of events in a distributed system and detecting causality violations.
Interprocess messages contain the state of the sending process's logical clock.



I. Initially all clocks are zero.

II. Each time a process experiences an internal event, it increments its own logical clock in the vector by one.

III. Each time a process sends a message, it increments its own logical clock in the vector by one (as in the bullet above, but not twice for the same event) and then sends a copy of its own vector.

IV. Each time a process receives a message, it increments its own logical clock in the vector by one and updates each element in its vector by taking the maximum of the value in its own vector clock and the value in the vector in the received message (for every element)

Logical clocks vs vector clocks

- Lamport's logical clocks help establish a partial ordering of events
- However, they don't fully capture causality between events in a distributed system where events can occur simultaneously but at different nodes.
- Vector clocks extend Lamport's idea by assigning a vector of integers to each process, allowing them to capture both the ordering and the causal relationships between events.
- They are more powerful than Lamport's clocks because they can resolve concurrent events—those events that occur at different processes but don't have a clear causality relationship.
- Vector clocks can compare the relationships between these events more accurately than Lamport's clocks, providing a more nuanced understanding of the system's state and event ordering

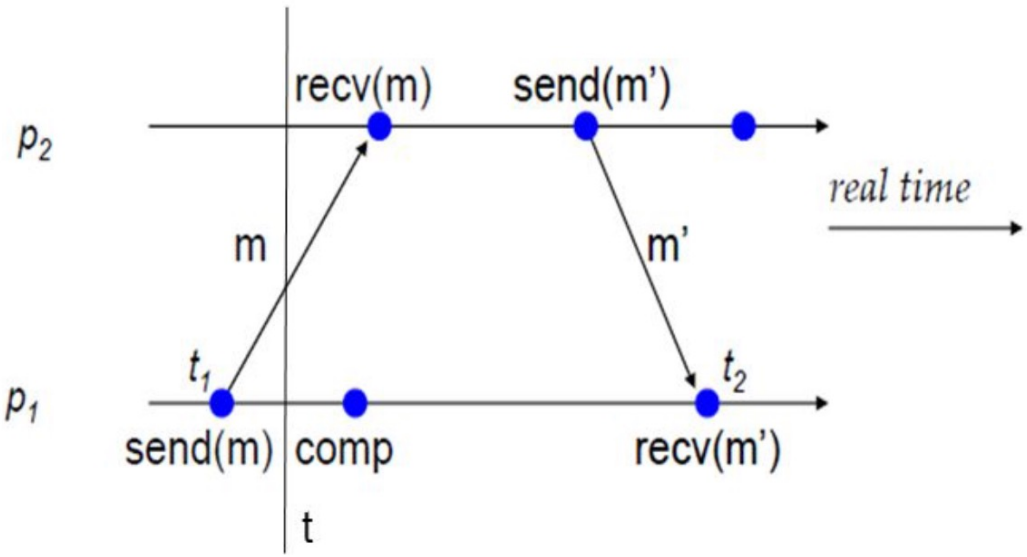
Global states

It is desirable to determine whether a particular property is true of a distributed system as it executes. We'd like to use logical time to construct a global view of the system state and determine whether a particular property is true. A few examples:

- Distributed garbage collection: Are there references to an object anywhere in the system? References may exist at the local process, at another process, or in the communication channel.
- Distributed deadlock detection: Is there a cycle in the graph of the "waits for" relationship between processes?
- Distributed termination detection: Has a distributed algorithm terminated?
- Distributed debugging: For instance, given two processes p_1 and p_2 owning variables x_1 and x_2 respectively, can we determine if the condition $|x_1-x_2| > \delta$ is ever true?

Definition: global state at real time t

The global state of a distributed system at time t is the set of local states of each individual process involved in the system plus the state of all communication channels, with messages in transit



Safety and liveness

Definition: *Global State Predicate*

"A global state predicate is a function that maps from the set of global states of processes in the system to {True, False}."

Safety - a predicate *always* evaluates to *false*. It means that a given undesirable property (e.g., deadlock) never occurs.

Liveness - a predicate *eventually* evaluates to *true*. It means that a given desirable property (e.g., termination) eventually occurs.

Safety and liveness

Liveness is the property which says that a system always makes progress.

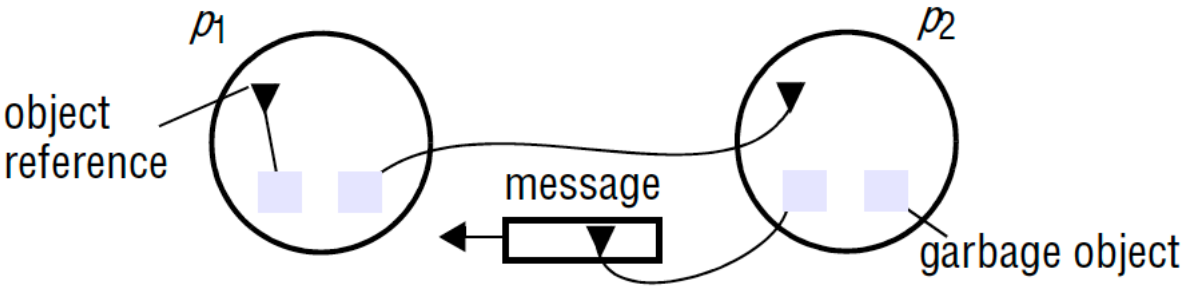
Safety is the property which says that the system is always in the correct state.

If we focus only on safety, then the system as a whole might not make progress.

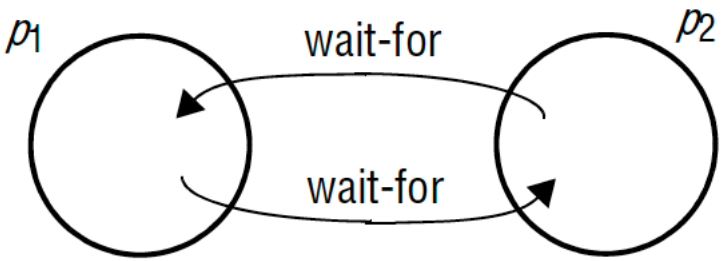
If we focus only on liveness, then safety might be compromised

Detecting global properties

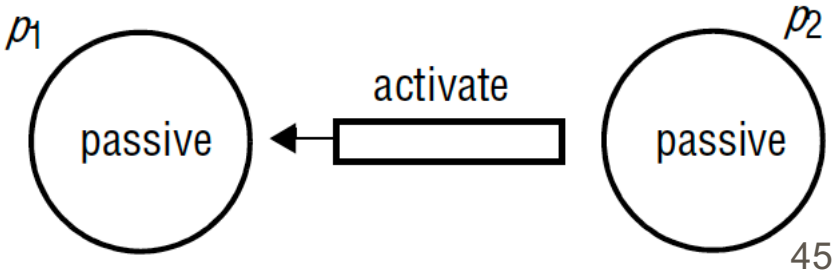
all examples show the need to observe a global state, and so motivate a general approach



(a) Garbage collection



(b) Deadlock



(c) Termination

Global states as cuts in partially ordered histories of events

Cuts provide the ability to "assemble a meaningful global state from local states recorded at different times".

Definitions:

ρ is a system of N processes p_i (i = 1, 2, ..., N)

history(p_i) = h_i = <e_{i0}, e_{i1},...>

h_{ik}=<e_{i0}, e_{i1},..., e_{ik}> - a finite prefix of the process's history

S_{ik} is the state of the process p_i immediately before the k-th event occurs

All processes record sending and receiving of messages. If a process p_i records the sending of message m to process p_j and p_j has not recorded receipt of the message, then m is part of the state of the channel between p_i and p_j.

A *global history* of ρ is the union of the individual process histories: $H = h_0 \cup h_1 \cup h_2 \cup ... \cup h_{N-1}$

A *global state* is the set of states of the individual processes: $S = (s_1, s_2, ..., s_N)$

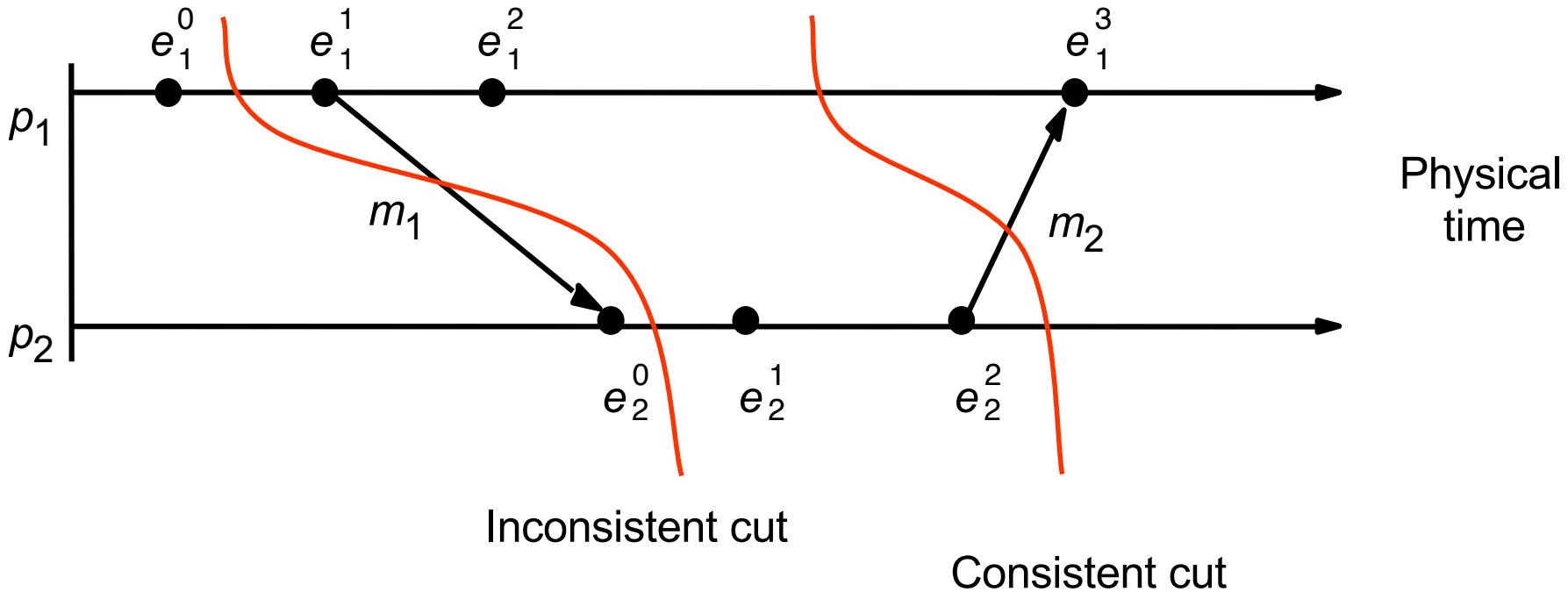
A cut of the system's execution is a subset of its global history that is a union of prefixes of process histories (see next slide).

The *frontier* of the cut is the last state in each process.

A cut C is *consistent* if, for all event pairs *e* and *e'*: $(e \in C \text{ and } e' \rightarrow e) \Rightarrow e' \in C$

A *consistent global state* is one that corresponds to a consistent cut.

Cuts


e_1^0
e_1^1
e_1^2
e_1^3
p_1
m_1
m_2
Physical time
p_2
e_2^0
e_2^1
e_2^2
Inconsistent cut
Consistent cut

Consistent global state

- A **cut** C is **consistent** if, for each event it contains, it also contains all the events that *happened-before* that event:

$$\forall \text{ events } e \in C \text{ , } f \rightarrow e \Rightarrow f \in C$$

- A *consistent global state* corresponds to a consistent cut.
- We characterize the execution of a distributed system as a series of transitions between global states $S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow \ldots$
- In each transition, only one event occurs at some single process in the system: this event is either the sending of a message, the receipt of a message or an internal event.
- (Events that occur simultaneously must be concurrent: neither *happened-before* the other)
- A system evolves in this way through consistent global states.

Some global state predicates are stable

- Detecting a deadlock or termination amounts to evaluating a *global state predicate*.
- A global state predicate maps from the set of global states of processes to {*True*, *False*}.
- One of the useful characteristics of the predicates associated with the state a) of an object being garbage, b) of the system being deadlocked, or c) the system being terminated, is that they are all *stable*:
- Once the system enters a state S in which the predicate is True, it remains True in all future states reachable from S

Representing safety and liveness on global states

- Suppose there is an *undesirable* property P that is a predicate of the system's global state – for example, P could be the property of *being deadlocked*.
- Let S_0 be the original state of the system.
- **Safety** with respect to P is the assertion that P is False for all states S reachable from S_0 .
- Conversely, let Q be a *desirable* property of a system's global state – for example, the property of *reaching termination*.
- **Liveness** with respect to Q is the property that, for any linearization L starting in the state S_0 , Q evaluates to True for some state S_L reachable from S_0

The snapshot algorithm (by Chandy and Lamport)

We describe a ‘snapshot’ algorithm for determining global states of distributed systems

The goal of the algorithm is to record a set of process and channel states (a ‘snapshot’) for a set of processes p_i such that, even though the combination of recorded states may never have occurred at the same time, the recorded global state is consistent.

The algorithm records the state locally at processes; it does not give a method for gathering the global state at one site

The snapshot algorithm (by Chandy and Lamport)

The algorithm assumes that:

- Neither channels nor processes fail – communication is reliable so that every message sent is eventually received intact, exactly once.
- Channels are unidirectional and provide FIFO-ordered message delivery.
- The graph of processes and channels is strongly connected (there is a path between any two processes).
- Any process may initiate a global snapshot at any time.
- The processes may continue their execution and send and receive normal messages while the snapshot takes place

The snapshot algorithm

The algorithm works using marker messages.

Each process that wants to initiate a snapshot records its local state and sends a marker on each of its outgoing channels.

All the other processes, upon receiving a marker, record their local state, the state of the channel from which the marker just came as empty, and send marker messages on all of their outgoing channels.

If a process receives a marker after having recorded its local state, it records the state of the incoming channel from which the marker came as carrying all the messages received since it first recorded its local state.

Some of the assumptions of the algorithm can be facilitated using a more reliable communication protocol such as TCP/IP

The algorithm can be adapted so that there could be multiple snapshots occurring simultaneously

Chandy and Lamport's 'snapshot' algorithm

Marker receiving rule for process p_i

On p_i's receipt of a *marker* message over channel c:

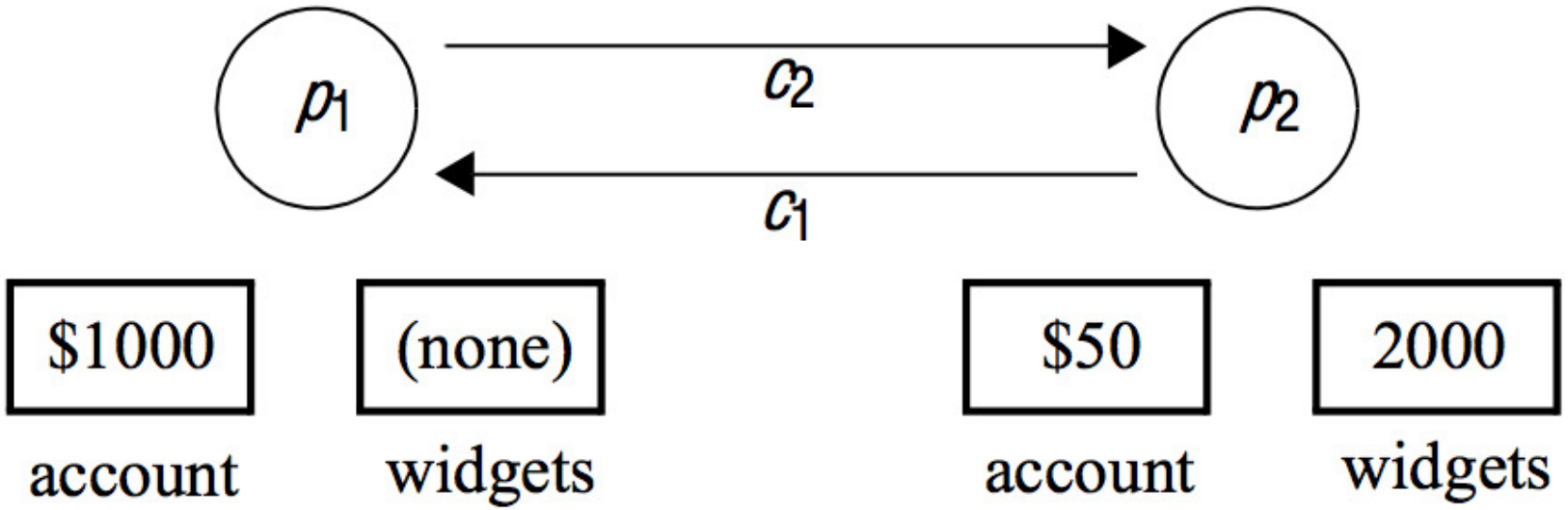
if (p_i has not yet recorded its state) it
records its process state now;
records the state of c as the empty set;
turns on recording of messages arriving over other incoming channels;
else
p_i records the state of c as the set of messages it has received over c since it saved its state.
end if

Marker sending rule for process p_i

After p_i has recorded its state, for each outgoing channel c:

p_i sends one marker message over c
(before it sends any other message over c).

Two processes and their initial states

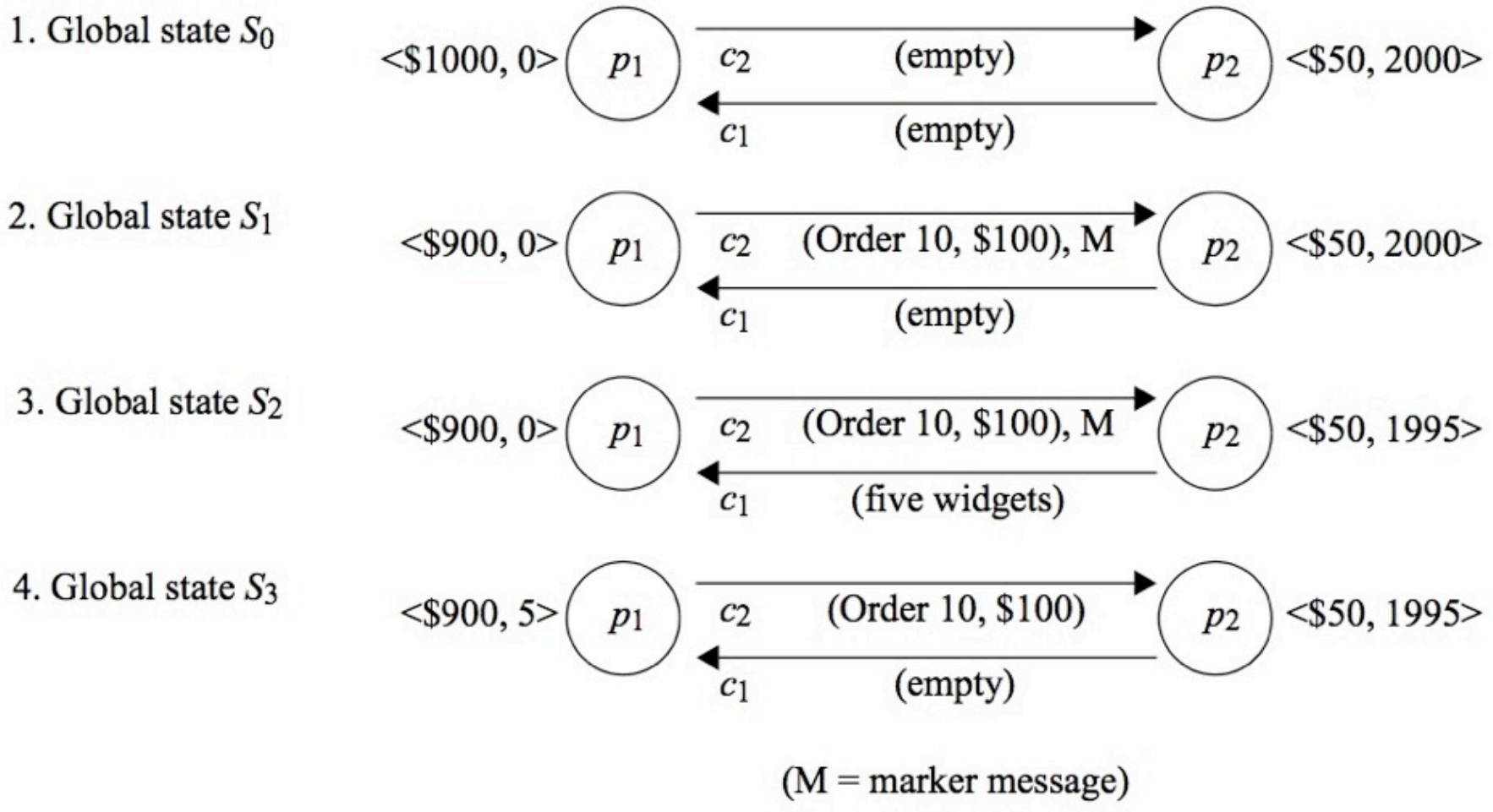


p1 and p2 , connected by two unidirectional channels, c1 and c2
The two processes trade in ‘widgets’.
Process p1 sends orders for widgets over c2 to p2 , enclosing payment at the rate of $10 per widget.
Some time later, process p2 sends widgets along channel c1 to p1

Next slides shows a run of the Chandy Lamport algorithm

The execution of the processes

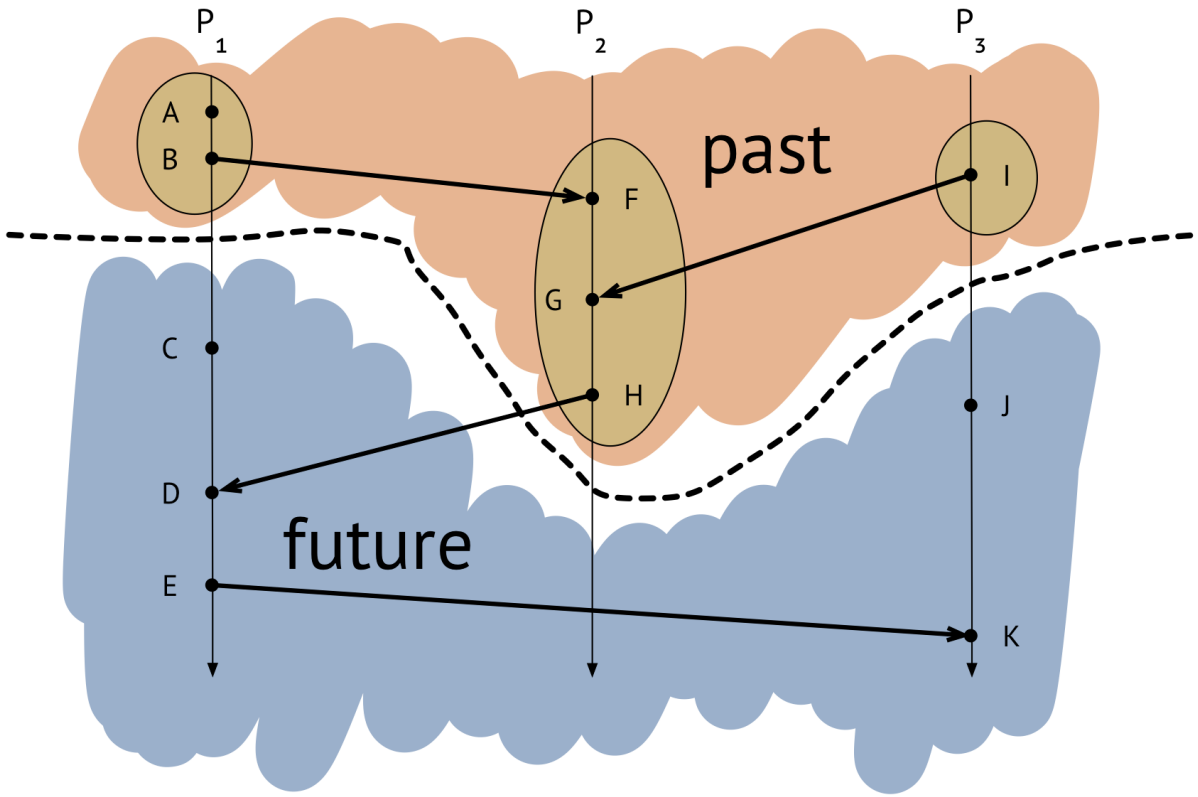
The system enters actual global state S1. Before p2 receives the marker, it emits an application message (five widgets) over c1 in response to a p1 's previous order, yielding a new actual global state S2



The final actual global state is S3 .
The final recorded state is p1 : <$1000, 0>; p2 : <$50, 1995>; c1 : <(five widgets)>; c2 : < >.
Note that this state differs from all the global states through which the system actually passed

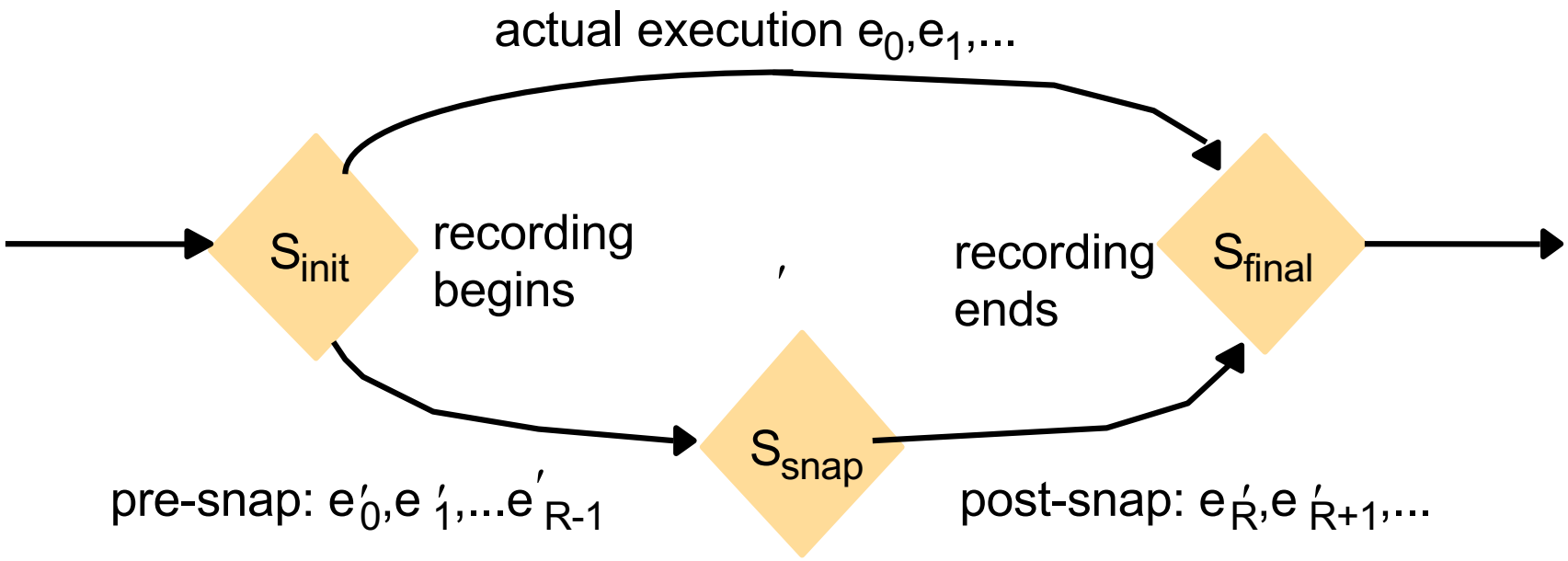
Another example of application of the algorithm

http://decomposition.al/blog/2019/04/26/an-example-run-of-the-chandy-lamport-snapshot-algorithm/



for every circled event, if we look at the events in its causal history, those events are all in the snapshot as well. In other words, the snapshot produces a *consistent cut* in this diagram, where every message received on the “past” side of the cut was sent on the “past” side, and no messages go backwards in time

Reachability between states in the snapshot algorithm



The snapshot algorithm selects a cut from the history of the execution.
The cut, and therefore the state recorded by this algorithm, is consistent.

Predicates on stories

- Our aim is to distinguish a) cases where a given global state predicate φ was *definitely* True at some point in the execution we observed, and b) cases where it was *possibly* True.
- The notion '*possibly*' arises because we may extract a consistent global state S from an executing system and find that φ(S) is True.
- No single observation of a consistent global state allows us to conclude whether a non-stable predicate ever evaluated to True in the actual execution.
- Nevertheless, we may be interested to know whether they might have occurred, as far as we can tell by observing the execution

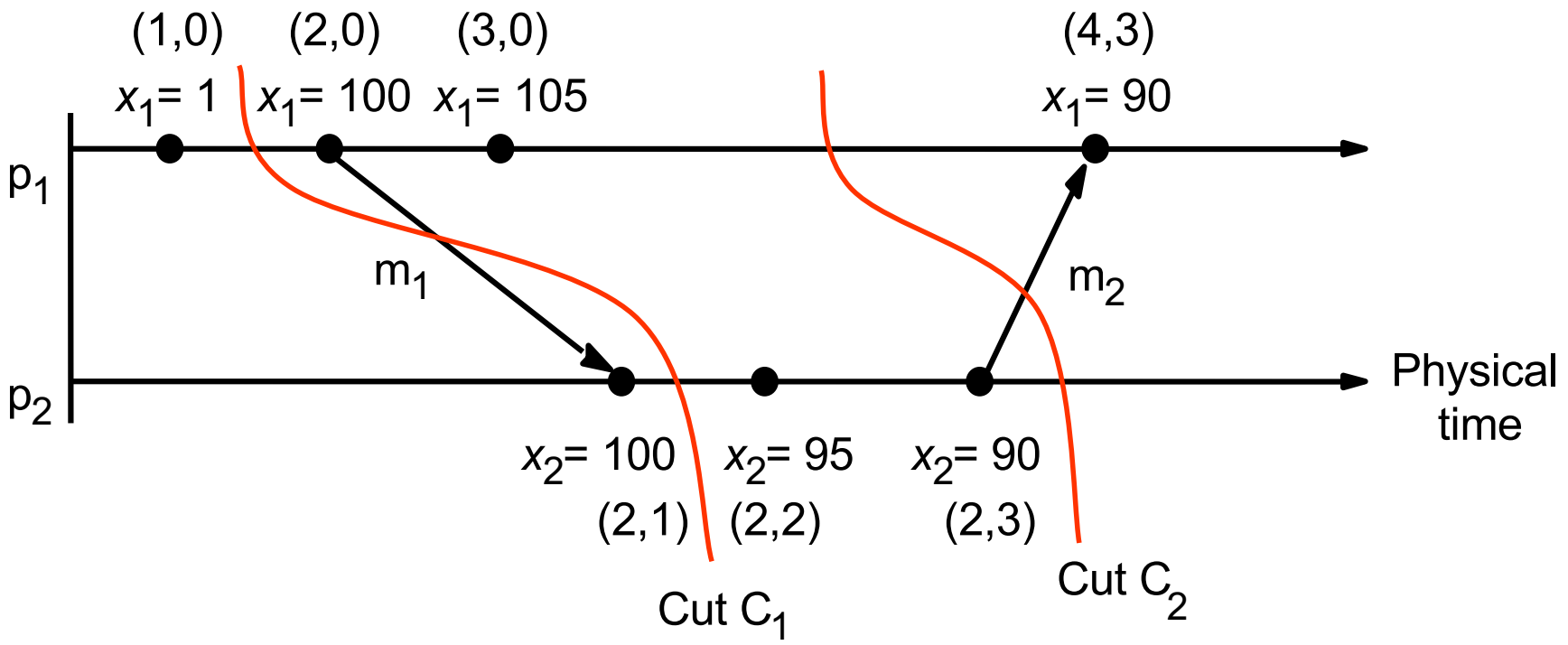
Predicates **possibly** and **definitely**

We define the notions of ***possibly*** φ and ***definitely*** φ for a predicate φ in terms of linearizations of H, the history of the system's execution:

possibly φ The statement *possibly* φ means that there is a consistent global state S through which a linearization of H passes such that φ (S) is True.

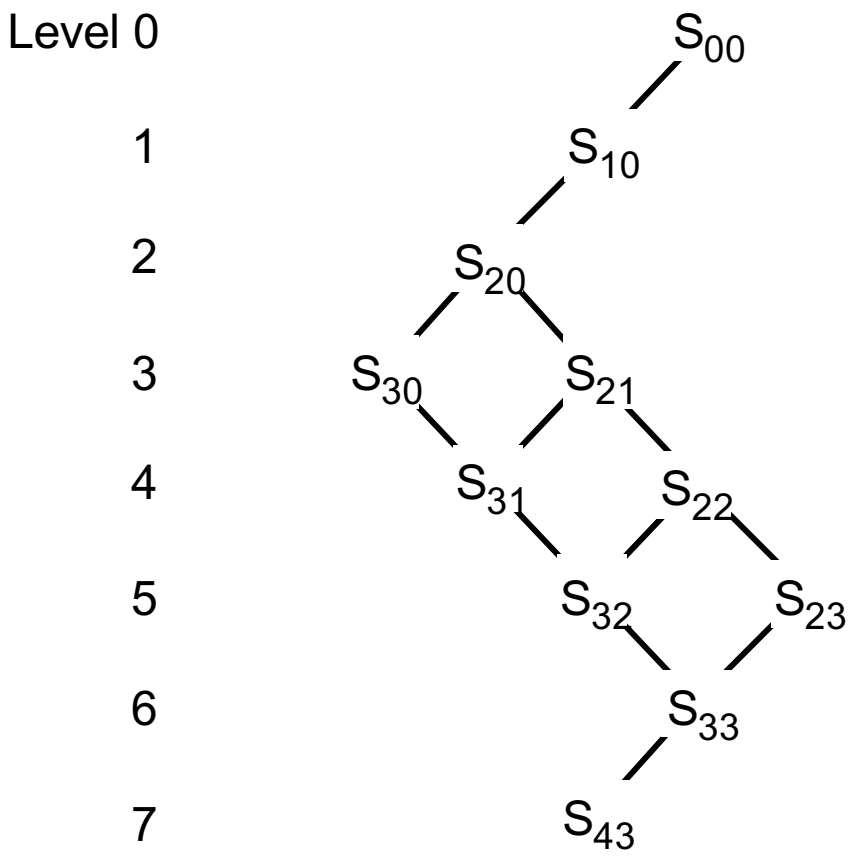
definitely φ :The statement *definitely* φ means that for all linearizations L of H, there is a consistent global state S through which L passes such that φ (S) is True.

Vector timestamps and variable values for the execution of Figure “cuts” slide 46



two processes p1 and p2 with variables x1 and x2 , respectively. The events shown on the timelines (with vector timestamps) are adjustments to the values of the two variables. Initially, x1 = x2 = 0 .
The requirement is x1 – x2 ≤ 50.The processes make adjustments to their variables, but ‘large’ adjustments cause a message containing the new value to be sent to the other process. When either of the processes receives an adjustment message from the other, it sets its variable equal to the value contained in the message

The lattice of global states for the execution of preceding slide



Sij = global state after *i* events at process 1 and *j* events at process 2

This structure captures the relation of reachability between consistent global states. The nodes denote global states, and the edges denote possible transitions between these states. The global state S_{00} has both processes in their initial state; S_{10} has p_2 still in its initial state and p_1 in the next state in its local history. The state S_{01} is not consistent, because of the message m_1 sent from p_1 to p_2 , so it does not appear in the lattice

shows the lattice of consistent global states corresponding to the execution of the two processes in the preceding slide

Algorithms to evaluate *possibly* φ and *definitely* φ

1. *Evaluating possibly* ϕ *for global history H of N processes*

$L := 0$;
States := { $(s_1^0, s_2^0, \ldots, s_N^0)$ };
while ($\phi(S) = False$ for all $S \in$ States)
    $L := L + 1$;
    Reachable := {S': S' reachable in H from some $S \in$ States $\wedge$ level(S') = L};
    States := *Reachable*
end while
output "*possibly* ϕ";

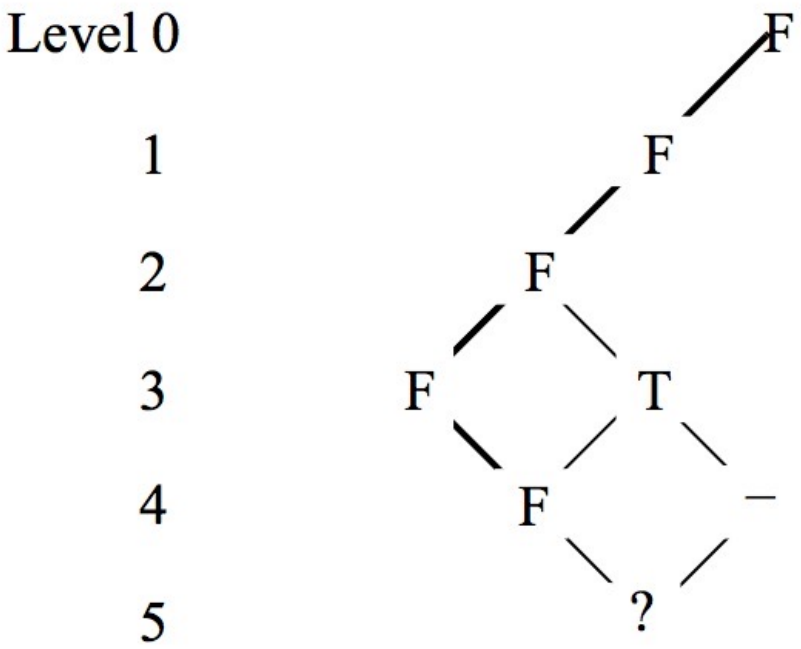
2. *Evaluating definitely* ϕ *for global history H of N processes*

$L := 0$;
if ($\phi(s_1^0, s_2^0, \ldots, s_N^0)$) *then States* := {} *else States* := { $(s_1^0, s_2^0, \ldots, s_N^0)$ };
while (*States* $\neq$ {})
    $L := L + 1$;
    Reachable := {S': S' reachable in H from some $S \in$ States $\wedge$ level(S') = L};
    States := {$S \in Reachable$: $\phi(S) = False$}
end while
output "*definitely* ϕ";

Evaluating definitely φ

- To evaluate *definitely* φ , the monitor again traverses the lattice of reachable states a level at a time, starting from the initial state s_1, s_2,….
- The algorithm assumes that the execution is infinite but may easily be adapted for a finite execution.
- It maintains the set States, which contains those states at the current level that may be reached on a linearization from the initial state by traversing only states for which φ evaluates to False.
- As long as such a linearization exists, we may not assert *definitely* φ: the execution could have taken this linearization, and φ would be False at every stage along it.
- If we reach a level for which no such linearization exists, we may conclude *definitely* φ

Evaluating *definitely* φ



$F = (\phi(S) = \text{False});\ T = (\phi(S) = \text{True})$

at level 3 the set States consists of only one state, which is reachable by a linearization on which all states are False (marked in bold lines).
The only state considered at level 4 is the one marked 'F'.
(The state to its right is not considered, since it can only be reached via a state for which φ evaluates to True.)
If φ evaluates to True in the state at level 5, then we may conclude *definitely* φ. Otherwise, the algorithm must continue beyond this level

Conclusions

- We discussed accurate timekeeping for distributed systems
- We described algorithms for synchronizing clocks despite the *drift* among them and the variability of message delays between computers.
- The degree of synchronization accuracy that is practically obtainable fulfils many requirements but is nonetheless **not sufficient** to determine the ordering of an arbitrary pair of events occurring at different computers.

Conclusions

- The *happened-before* relation is a partial order on events that reflects a flow of information between them – within a process, or via messages between processes.
- Some algorithms require events to be ordered in *happened-before* order: for example, successive updates made to separate copies of data.
- **Lamport clocks** are counters that are updated in accordance with the *happened-before* relationship between events.
- **Vector clocks** are an improvement on Lamport clocks, in that it is possible to determine by examining their vector timestamps whether two events are ordered by *happened-before* or are concurrent.

Conclusions

- We introduced events, local and global histories, cuts, local and global states, runs, consistent states, linearizations (consistent runs) and reachability.
- A consistent state or run is one that is in accord with the *happened-before* relation.
- We considered the problem of recording a consistent global state by observing a system's execution.
- Our goal was to evaluate a predicate on this state.
- An important class of predicates are the *stable* predicates.

Exercise

Why is clock synchronization necessary?

Solution

Why is clock synchronization necessary?

- For example, an eCommerce transaction involves events at a merchant's computer and at a bank's computer: it is important, for auditing purposes, that those events are timestamped accurately.
- Algorithms that depend upon clock synchronization have been developed for several problems in distribution: these include

maintaining the consistency of distributed data (the use of timestamps to serialize transactions),

checking the authenticity of a request sent to a server (see the Kerberos authentication protocol)

eliminating the processing of duplicate updates

- In object oriented systems we need to be able to establish whether references to a particular object no longer exist – whether the object has become garbage (in which case we can free its memory). Establishing this requires observations of the states of processes (to find out whether they contain references) and of the communication channels between processes (in case messages containing references are in transit).

Exercise

Describe the design requirements for a system to synchronize the clocks in a distributed system.

Solution

Describe the design requirements for a system to synchronize the clocks in a distributed system.

Major design requirements:

i. there should be a limit on deviation between clocks or between any clock and UTC;

ii. clocks should only ever advance;

iii. only authorized principals may reset clocks (see Kerberos)

In practice (i) cannot be achieved unless only benign failures are assumed to occur and the system is synchronous.

Exercise

A client attempts to synchronize with a time server. It records the round-trip times and timestamps returned by the server in the table.

- Which of these times should it use to set its clock?
- To what time should it set it?
- Estimate the accuracy of the setting with respect to the server's clock.
- If it is known that the time between sending and receiving a message in the system concerned is at least 8 ms, do your answers change?

Round-trip (ms)	Time (hr:min:sec)
22	10:54:23.674
25	10:54:25.450
20	10:54:28.342

Solution

- The client should choose the minimum round-trip time of 20 ms = 0.02 s. It then estimates the current time to be 10:54:28.342 + 0.02/2 = 10:54:28.352. The accuracy is ± 10 ms.
- If the minimum message transfer time is known to be 8 ms, then the setting remains the same but the accuracy improves to ± 2 ms.