IN.5022 — Concurrent and Distributed Computing

Time and Order

Prof. P. Felber

pascal.felber@unine.ch

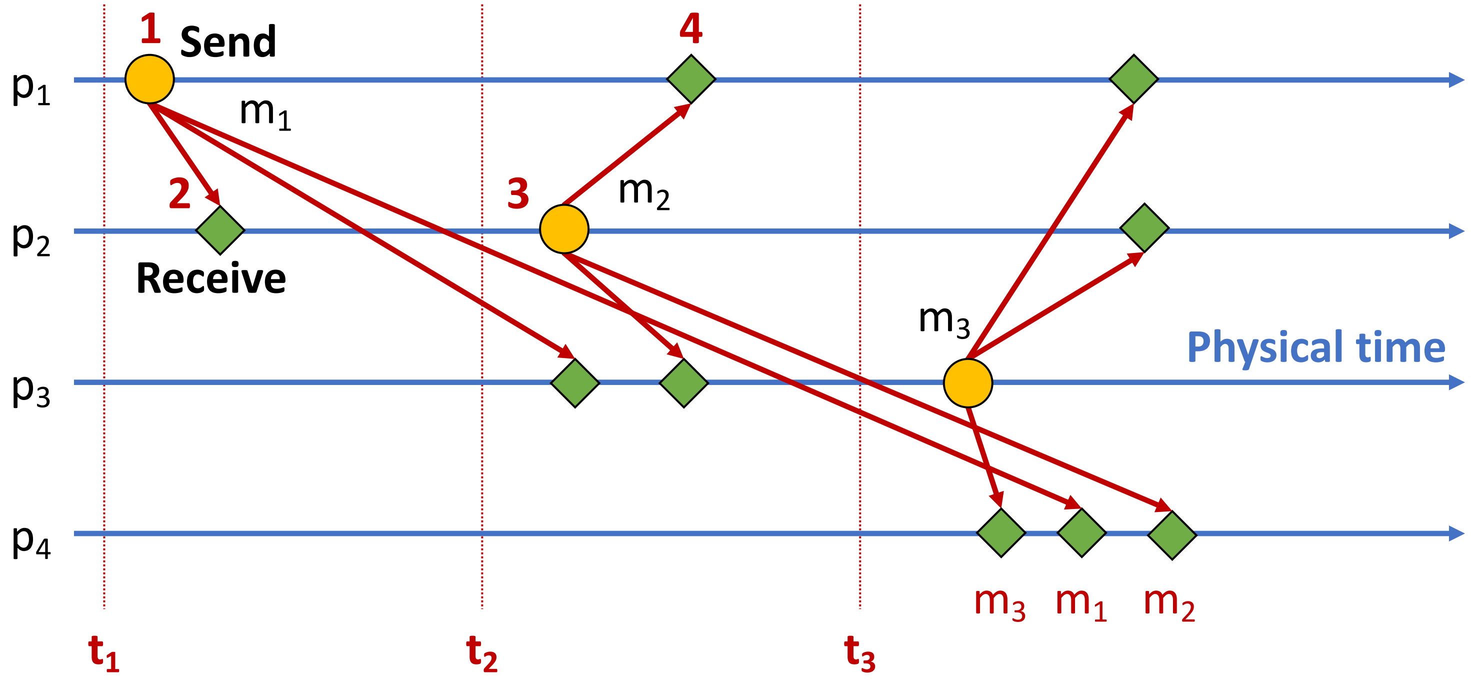
Agenda

- Time and order
- Clocks and physical time
- Logical clocks
- Vector clocks

Time: a major issue

- We casually use temporal concepts, mainly to measure time and order events
  - E.g., “upon timeout, rollback”, “once read lock is granted, acquire write lock” , “p suspect that q has failed”
- Used by many algorithms
  - E.g., distributed synchronization, maintain data consistency, authenticate requests, control concurrency
- In distributed systems, how can we relate local notion of time in a single process to a global notion of time?

Ordering of events?



Three notions of time

- Global clock
  - Time seen by external observer (wall clock time)
  - Hard to implement, limited temporal precision
- Local clocks of individual processes
  - Subject to skew and drift
  - Resynchronization is inaccurate
- Logical notion of time
  - Focus on relative ordering of events (occurred before)
  - No “real-time” clock

Physical time

Logical time

Time vs. ordering

- Time is often *wrongly* used to do ordering
    - E.g., make
- Time is useful for measuring intervals
    - E.g., performance analysis
- Ordering is useful for capturing temporal relationships
    - E.g., debugging (linearize observed set of events)
- How can we determine ordering in truly decentralized systems (many points of serialization)?

Computer clocks

- Each computer in a DS has its own internal clock
  - Used by local processes to obtain the current time value
  - Processes on different computers can timestamp events, but clocks on different computers may give different times

    Clock skew: instantaneous differences between two clocks
  - Computer clocks “drift” from perfect time and their drift rates differ from one another

    Clock drift rate: the relative amount that a computer clock differs from a perfect clock
- Even if clocks on all computers in a DS are set to the same time, their clocks will eventually vary quite significantly unless corrections are applied

Computer clocks

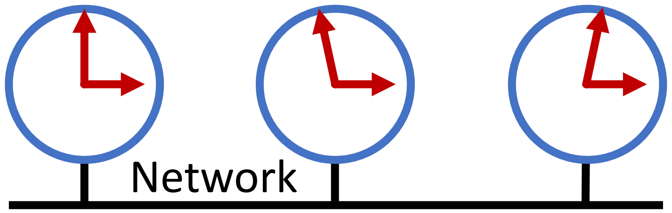
- To timestamp events, we can use the computer's clock
- At real time t, the OS reads the time on the computer's hardware clock $H_i(t)$
  - E.g., a 64-bit number giving nanoseconds since some base time
- It calculates the time on its software clock

$$C_i(t) = \alpha H_i(t) + \beta$$

  - In general, the clock is not completely accurate

How accurate should the clock resolution be?

Clock skew



- Computer clocks are generally *not* in perfect agreement (skew)
- Skew increases with drift
  - Ordinary quartz clocks drift by about 1 second every 11-12 days (10^{-6} s/s)
  - High precision quartz clocks drift rate is about 10^{-7}-10^{-8} s/s

What happens to clocks when batteries become low?

- Computers must periodically synchronize their clocks!

Clock correctness

- A hardware clock H is said to be correct if its drift rate is within a bound $\rho > 0$ (e.g., 10^{-6} s/s)
- This means that the error in measuring the interval between real times t and t' ($t' > t$) is bounded

$$(1 - \rho)(t' - t) \leq H(t') - H(t) \leq (1 + \rho)(t' - t)$$

  - Bounded drift forbids jumps in time readings of hardware clocks

Clock correctness

- Weaker condition of monotonicity on software clocks

  $t' > t \Rightarrow C(t') > C(t)$ (clock value only ever increases)

  - E.g., required by Unix make
- We can achieve monotonicity with a hardware clock that runs fast by updating software clock at a slower rate
  - Adjust the values of α and β in $C_i(t) = \alpha H_i(t) + \beta$
- A faulty clock is one that does not obey its correctness condition

  Crash failure: a clock stops ticking

  Arbitrary failure: any other failure, e.g., jump back in time (Y2K)

Synchronizing physical clocks

- External synchronization
  - A computer clock C_i is synchronized with an external authoritative time source S, so that

    $|S(t) - C_i(t)| < D$ for $i = 1, 2, \ldots, N$ over an interval I of real time
  - The clock C_i is *"accurate to within the bound D"*
- If two processes are synchronized externally within a bound D, then the reading of their clocks does not differ by more than twice D

Synchronizing physical clocks

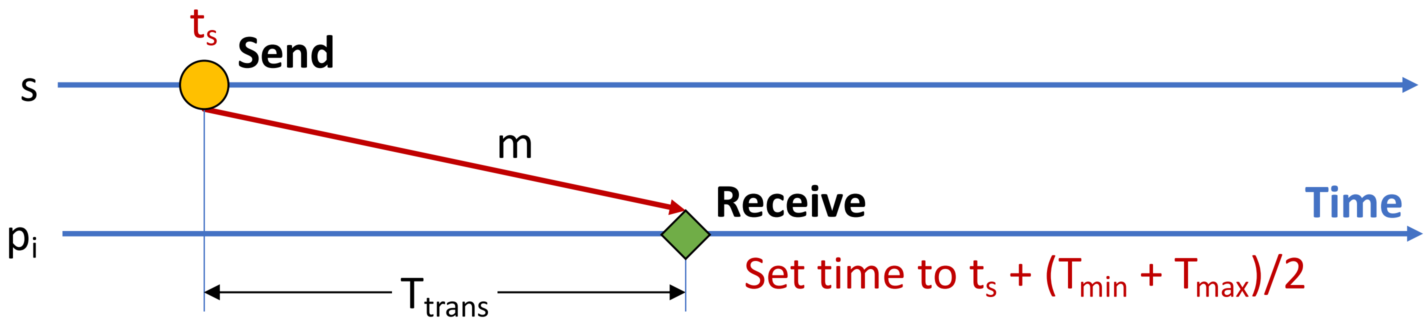
- Internal synchronization
  - The clocks of each pair of computers are synchronized with one another so that

    $|C_i(t) - C_j(t)| < D$ for $i = 1, 2, \ldots, N$ over an interval I of real time
  - The clocks C_i and C_j *"agree within the bound D"*
- Internally synchronized clocks are not necessarily externally synchronized, as they may drift collectively
  - Often, this is not a problem…

Clocks in synchronous systems

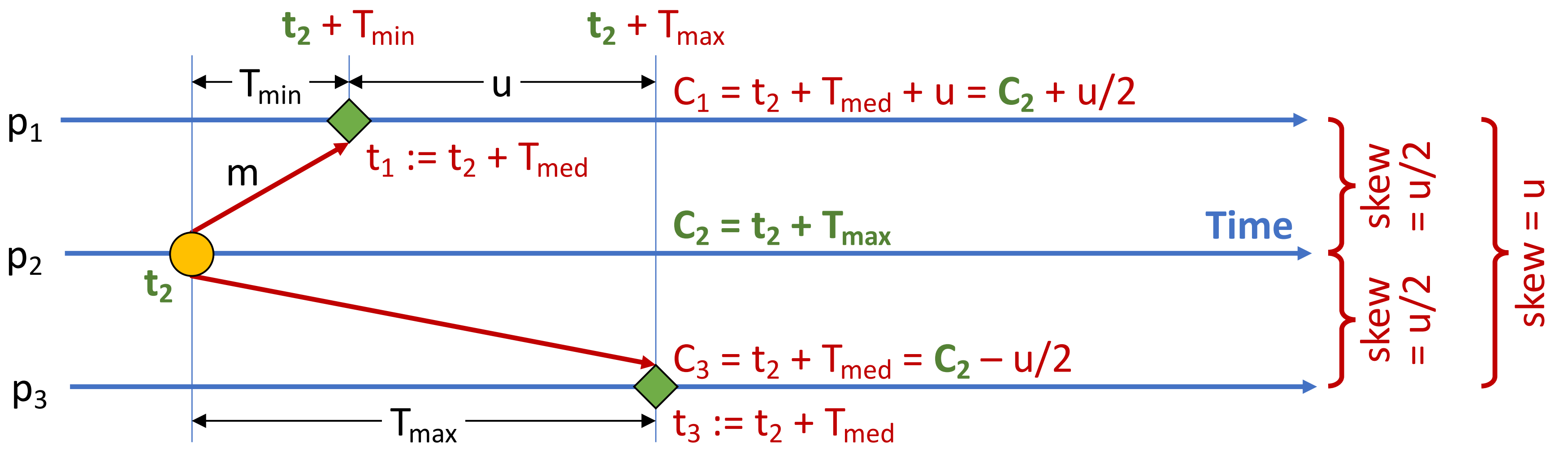
- In a synchronous system
  - We know bounds on message transmission delay

    $T_{min} \leq T_{trans} \leq T_{max}$
- External synchronization with time server *s*
  - Uncertainty $u = T_{max} - T_{min}$
  - Processes synchronized within $u/2$ with time server



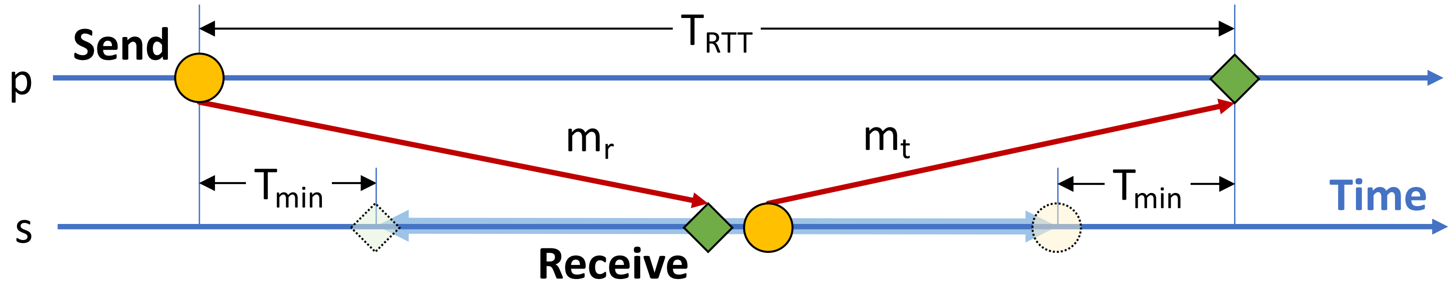
Clocks in synchronous systems

- Internal synchronization
  - Process p_i sends its local time t to process p_j
  - Uncertainty $u = T_{max} - T_{min}$
  - Set clock to $t + (T_{max} + T_{min})/2 = t + T_{med} \Rightarrow$ skew $\leq u/2$
  - With N processes, optimal precision of clocks C_i is $u(1 - 1/N)$



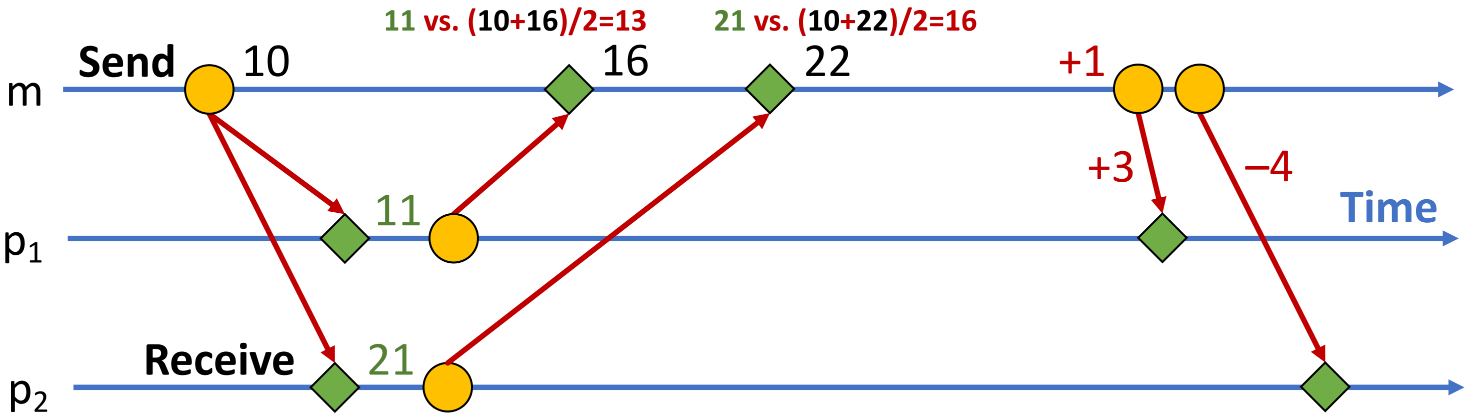
Clocks in asynchronous systems

- [Cristian 89] External synchronization with server s
  - Process p requests time in m_r and receives t in m_t from s
    - Let T_{RTT} be the RTT recorded by p and T_{min} the minimum transmission time
  - Process p sets its clock to $t + (T_{RTT}/2)$
  - Uncertainty is $T_{RTT} – 2T_{min}$ and accuracy is $\pm (T_{RTT}/2 – T_{min})$
    - Earliest time s puts t in message m_t is T_{min} after p sent m_r
    - Latest time was T_{min} before m_t arrived at p
    - Clock of s when m_t arrives is in $[t + T_{min}, t + T_{RTT} – T_{min}]$



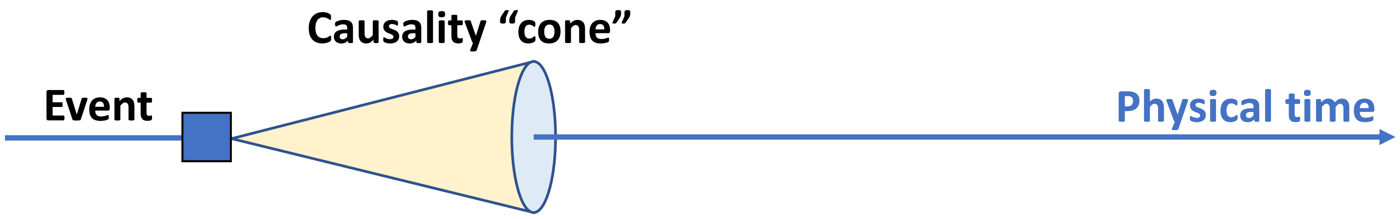
Clocks in asynchronous systems

- [Berkeley 89] Internal synchronization (group of computers)
  - A master m polls to collect clock values from others (slaves)
  - The master uses RTTs to estimate the slaves' clock values
  - It takes an average (eliminating dubious values)
  - It adjusts its clock and sends the required adjustment to the slaves
    - Better than sending the time, which depends on the RTT
  - If m fails, we can elect a new master (not in bounded time)



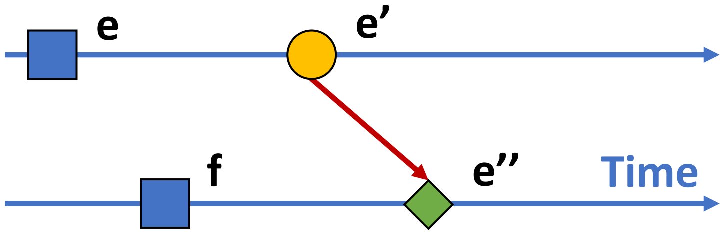
Logical time

- Alternative to synchronising physical clocks
- Event are uniquely ordered in any single process
  - $\rightarrow_i$: total order defined by the order in which p_i observes events
- Distributed events are ordered according to causality
  - When a message m is sent, send(m) occurs before receive(m)
- Note: events propagate at a finite speed



“Potential” causality

- “Happened-before” relation ($\rightarrow$)
  - [HB1]: if $\exists$ process p_i : $e \rightarrow_i e'$, then $e \rightarrow e'$
  - [HB2]: $\forall$ message m, send(m) $\rightarrow$ receive(m)
  - [HB3]: if $e \rightarrow e'$ and $e' \rightarrow e''$, then $e \rightarrow e''$ (transitive)
- Partial order
  - For some events, we do not know which one happened first
- Concurrent events
  - If $e \nrightarrow f$ and $f \nrightarrow e$, then $e \parallel f$



Logical clocks definition

- A logical clock C is a mapping from the set of states S to $\mathbb{N}$ (natural numbers) with the following constraint

$$\forall\ s, t \in S : s \prec t \vee s \leadsto t \Rightarrow C(s) < C(t)$$

$\prec$: locally precedes

$\leadsto$: remotely precedes

Logical clocks algorithm

- Introduced by Leslie Lamport in 1978
  - Orders events globally according to the $\rightarrow$ relation
- L_i: logical clock (counter) used by process p_i to apply logical ("Lamport") timestamp $L(e)$ to event e

  [LC1]

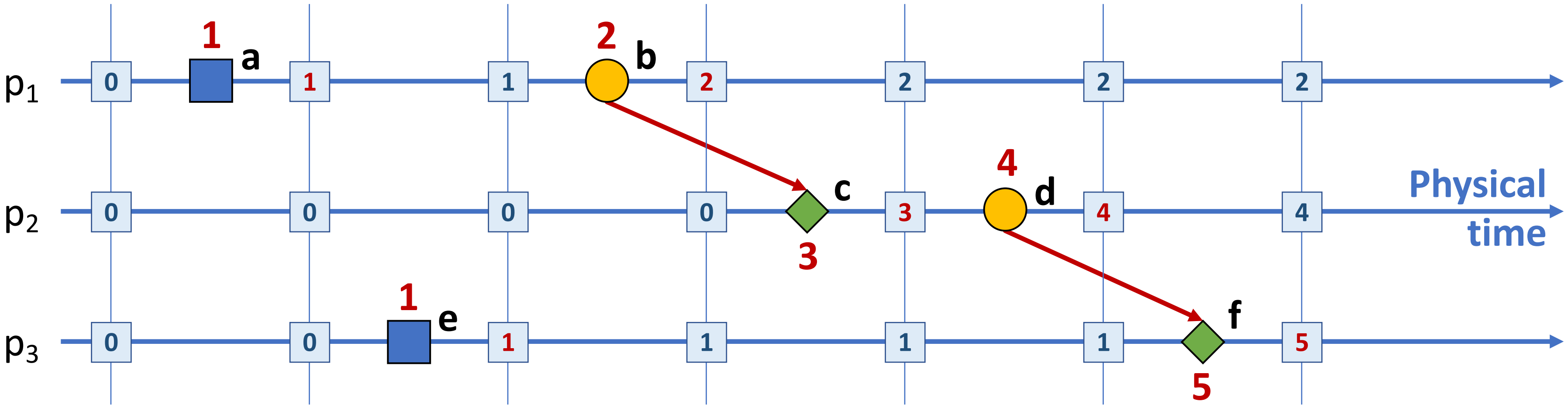
  L_i incremented before each event at p_i: $L_i := L_i + 1$

  [LC2]

  a) When process p_i sends message m, it piggybacks on m the value $t = L_i$
  b) On receiving (m, t), p_j computes $L_j := \max(L_j, t)$ and then applies LC1 before timestamping $receive(m)$

$$e \rightarrow e' \Rightarrow L(e) < L(e')$$

Logical clocks example



Logical clocks and total order

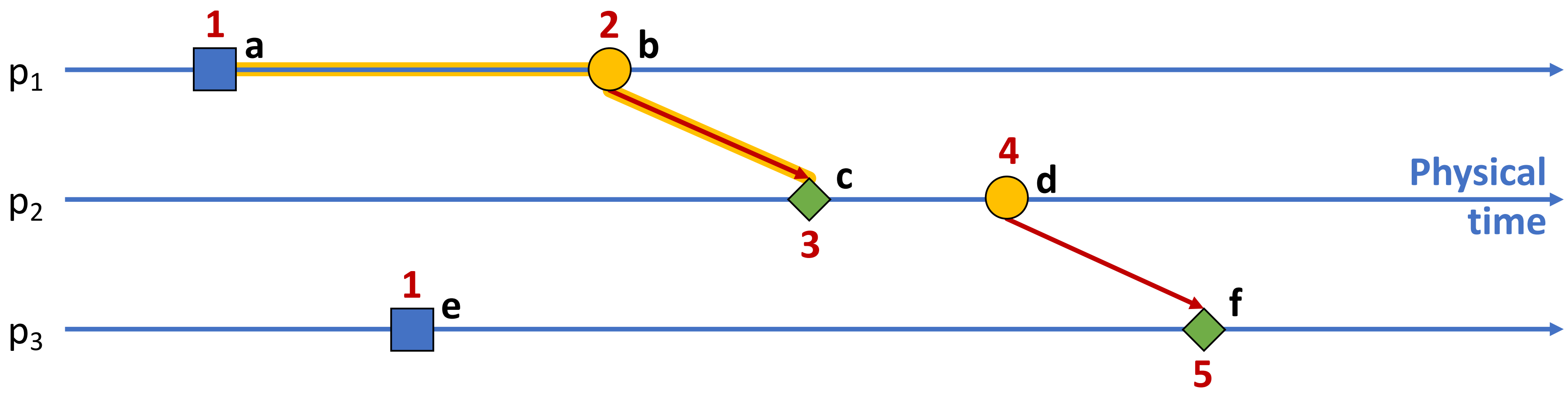
- Some events have the same Lamport timestamp
  - E.g., L(a) = L(e)
- Break ties by using processes ranks
  - Local timestamp T_i at process p_i becomes global timestamp (T_i, i)

    $(T_i, i) < (T_j, j) \Leftrightarrow T_i < T_j \vee (T_i = T_j \wedge i < j)$
  - Global timestamps form a total order

Is that good enough?

Logical clocks limitations

- Are events a and c ordered?
  - Yes, and L(a) < L(c)
- Are events e and c ordered?
  - No, **but** L(e) < L(c)



Logical clocks limitations

- For some pair of events, we do not know which happened first (partial ordering)
  - When ordering is unknown, an arbitrary order is chosen
  - We cannot find true dependencies by looking at ordering
  - Timestamps do not distinguish between causally and arbitrarily ordered events

Vector clocks

- Introduced (independently) by Colin Fidge and Friedemann Mattern in 1988
- Overcome main shortcoming of Lamport's clocks

  $L(e) < L(e') \not\Rightarrow e \rightarrow e'$
- Preserve partial ordering information
- If ordering of events is unknown, leave them unordered (incomparable events)
  - Easier to detect race conditions

Vector clocks concepts

- Each process has an array of logical clocks
  - One clock per process in the system
  - Vector of last known timestamps

    $V_i = (t_{p_0}, t_{p_1}, t_{p_2}, \ldots, t_{p_N})$
- Every event is given a timestamp vector by the process to which it belongs
- The ordering, or lack thereof, of two events can be determined by comparing their timestamps

Vector clocks definition

- A vector clock V is a mapping from the set of states S to $\mathbb{N}^k$ (vector of natural numbers) with the following constraint

$$\forall s, t \in S : s \rightarrow t \Rightarrow V(s) < V(t)$$

$\rightarrow$ is a partial order, thus $<$ must also be a partial order

Vector clocks algorithm

- V_i: vector clock used by process p_i to timestamp events

  [VC1]

  Initially, $V_i[j] = 0$ for $i, j = 1, 2, \ldots, N$

  [VC2]

  Before timestamping e, p_i sets $V_i[i] := V_i[i] + 1$

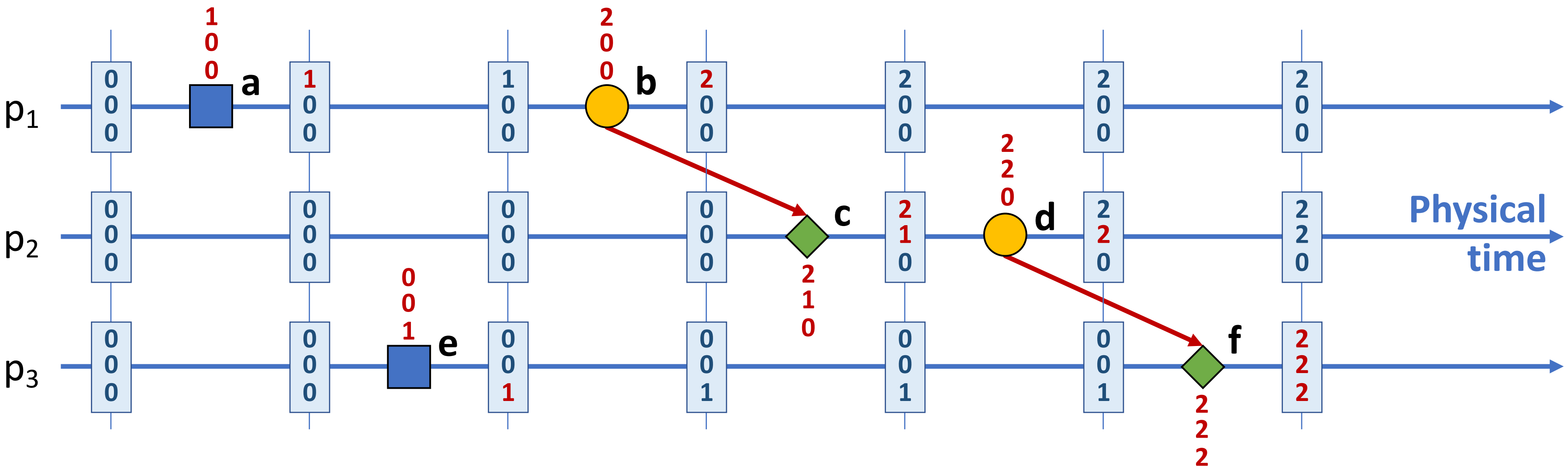
  [VC3]

  When process p_i sends message m, it piggybacks on m the value $t = V_i$

  [VC4]

  On receiving (m, t), process p_i computes $V_i[j] := \max(V_i[j], t[j])$ for $j = 1, 2, \ldots, N$ and then applies VC2 before timestamping the event $receive(m)$

Vector clocks example

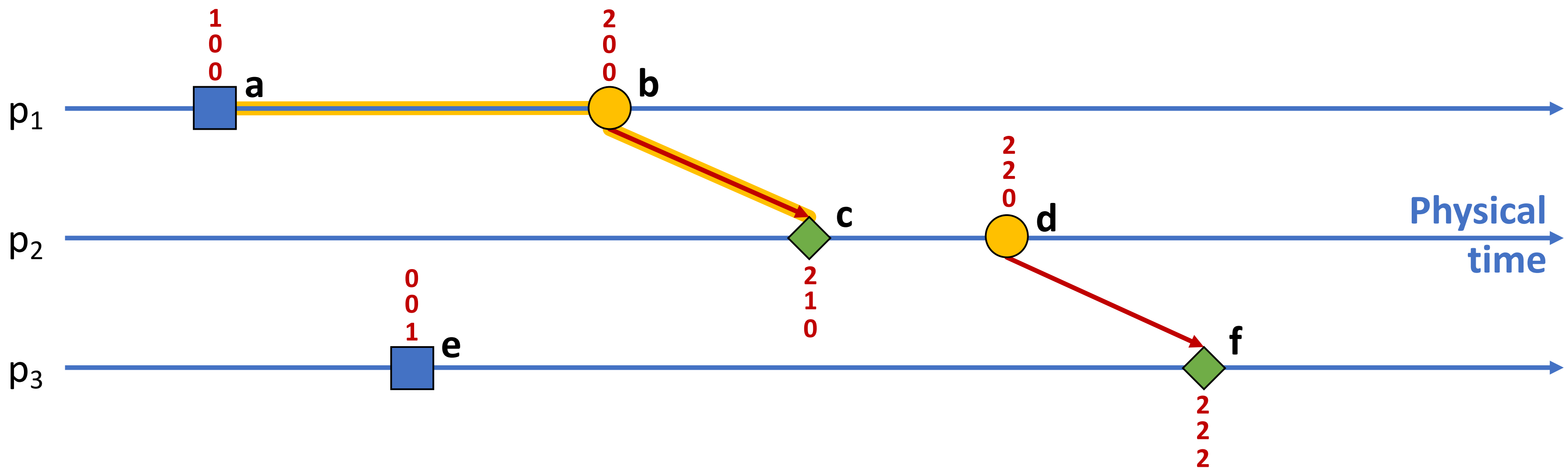


Vector clocks properties

- Interpretation
  - $V_i[i]$: number of events that p_i has timestamped
  - $V_i[j]$ $(j \neq i)$: number of events occurred at p_j that p_i has potentially been affected by
- Comparing vector clocks
  - $V = V' \Leftrightarrow V[j] = V'[j]$ for $j = 1, 2, \ldots, N$
  - $V \leq V' \Leftrightarrow V[j] \leq V'[j]$ for $j = 1, 2, \ldots, N$
  - $V < V' \Leftrightarrow V \leq V' \wedge V \neq V'$
  - E.g., $(2,1,0,4) < (2,3,0,4)$

Vector clocks benefits

- Are events a and c ordered?
  - Yes, **because** V(a) < V(c)
- Are events e and c ordered?
  - No, **because** $V(e) \nleq V(c) \wedge V(c) \nleq V(e)$



Vector clocks pros and cons

- ☺ We have $e \rightarrow e' \Leftrightarrow V(e) < V(e')$
  - Can tell whether e "happened before" e' from vector clocks
- ☺ If $V(e) \nleq V(e')$ and $V(e') \nleq V(e)$, then $e \parallel e'$
  - Events are concurrent when vector clocks are not comparable
  - E.g., $(2,1,0,4) \parallel (2,3,0,2)$
- ☹ Requires static notion of system membership
  - Processes must agree on the number of entries in vectors
  - Vector clocks are useful in systems that deal with membership, e.g., group communication
  - There are techniques to deal with dynamic group membership

Example: atomic broadcast

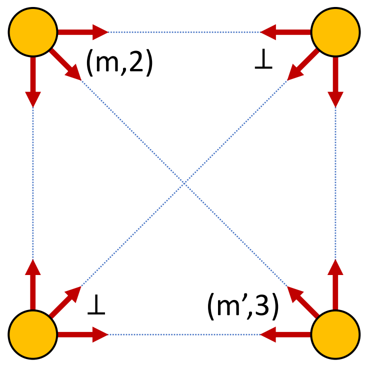
- A process sends a message atomically to all other processes
  - [Agreement]: if some correct process delivers m, then all correct processes deliver m
  - [Ordering]: no two correct processes deliver any two messages in different orders
  - [Termination]: if a correct process broadcasts m, then all correct processes eventually deliver m

How can we implement atomic broadcast using logical time?

System model: reliable channels, asynchronous system, no failures

Atomic broadcast protocol

- Sender adds timestamp in the broadcast
- Receiver waits for full set of messages
  - Orders messages by logical timestamp
  - Breaks ties using sender identifiers
  - Delivers messages in this order
  - Picks new timestamp greater than all seen
- How do we know if we have a full set?
  - Rely upon "membership" to wait for (sets of) messages from *all members*
  - System runs in rounds
  - Send "null" message if nothing to send



Deliver m (timestamp 2)
before
m' (timestamp 3)

Interpretation of logical time

- The relation "a happened before b" means that information can flow from a to b
- The relation "a is concurrent with b" means that no information can flow between a to b
  - Many events can be concurrent with a given event
  - Logical time cannot help detect "simultaneous" events
  - "Real-time" clocks cannot help either, because of limited precision and communication latencies
    - Useful only for "coarse-grain" applications

Things to remember

- Accurate timekeeping is important in a distributed system
  - Algorithms synchronize clocks despite their drift and the variability of message delays
- Time is a tool, typically used to put events in some sort of order
  - E.g., order updates on replicated data
- Often physical time is not necessary and logical time can be used instead
- Logical and vector clocks provide a *partial* order
  - Can be extended to a total order, e.g., by adding clock time or process identifiers to break ties