Distributed Systems

Rik Sarkar
James Cheney

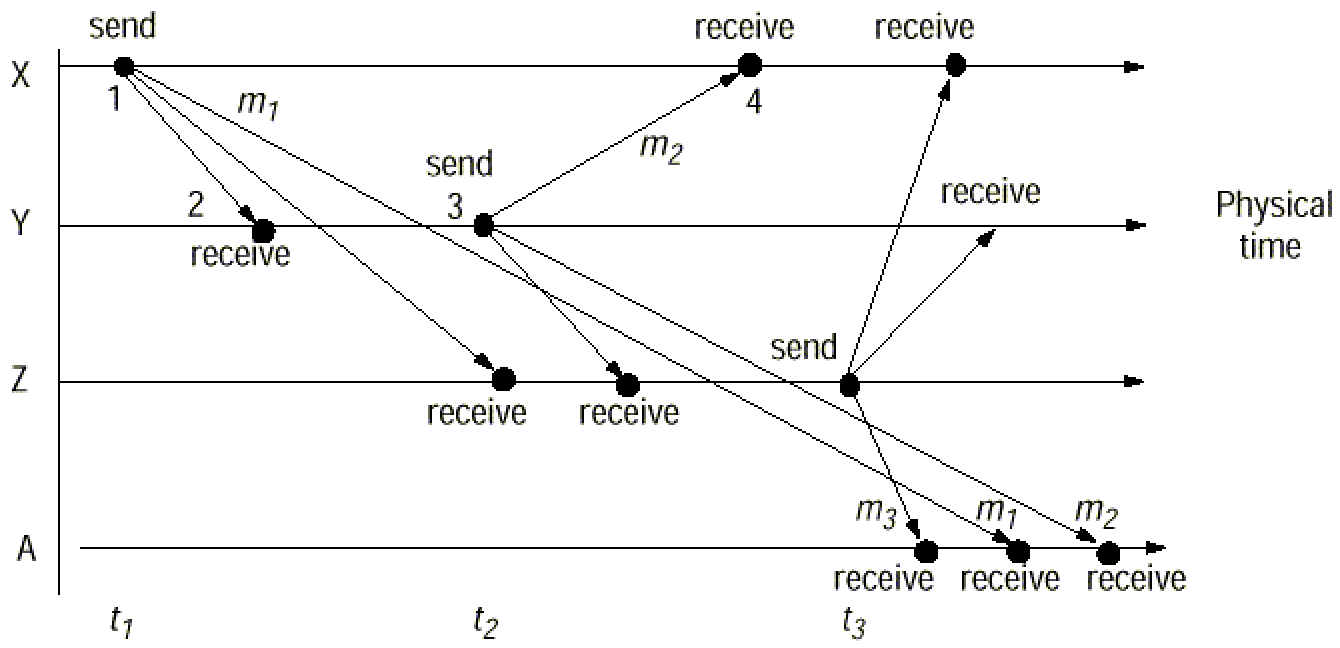
Logical Clocks & Global State

January 30, 2014

Asynchronous event ordering

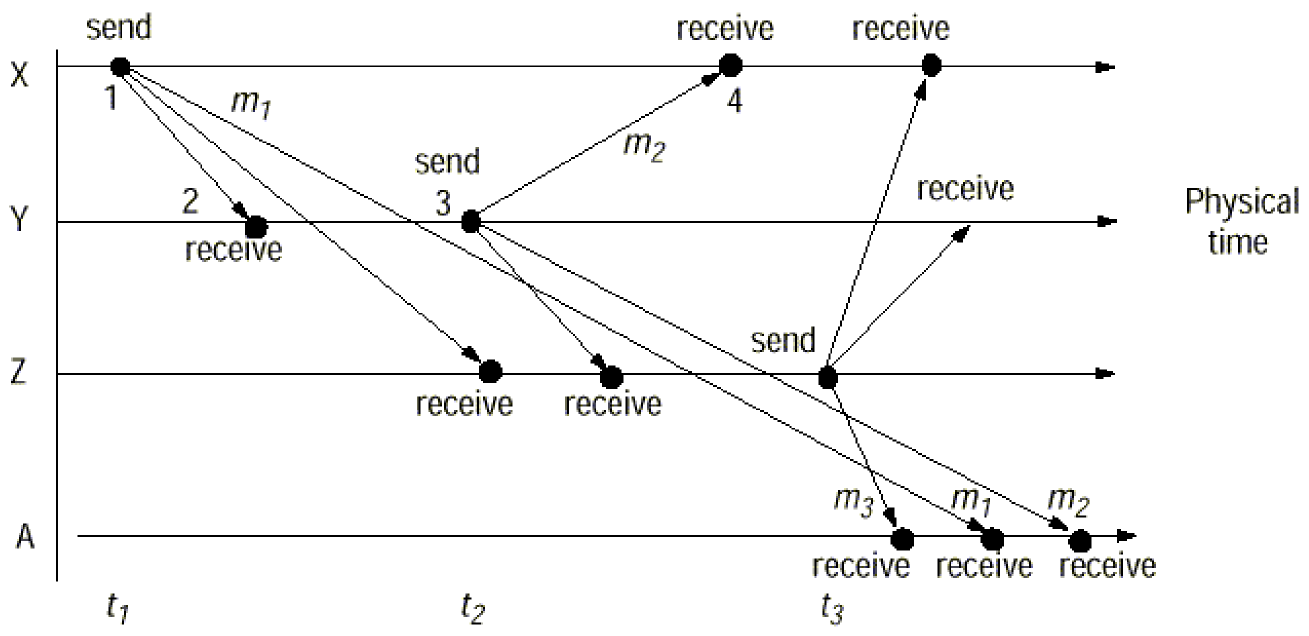
- Goal: achieve some measure of synchronization between processes located at different sites
- Ultimately, we will **never** be able to synchronize clocks to arbitrary precision
  - For some applications low precision is enough, for others it is not.
- Where we cannot guarantee high enough precision for synchronization, we are forced to operate in the asynchronous world
- Despite this we can still provide a **logical ordering** on events, which may useful for certain applications

Logical ordering



- Logical orderings attempt to give an order to events similar to physical causal ordering of reality but applied to distributed processes
- Logical clocks are based on the simple principles:
  - Any process knows the order of events which it observes or executes
  - Any message must be sent before it is received

Happened-before

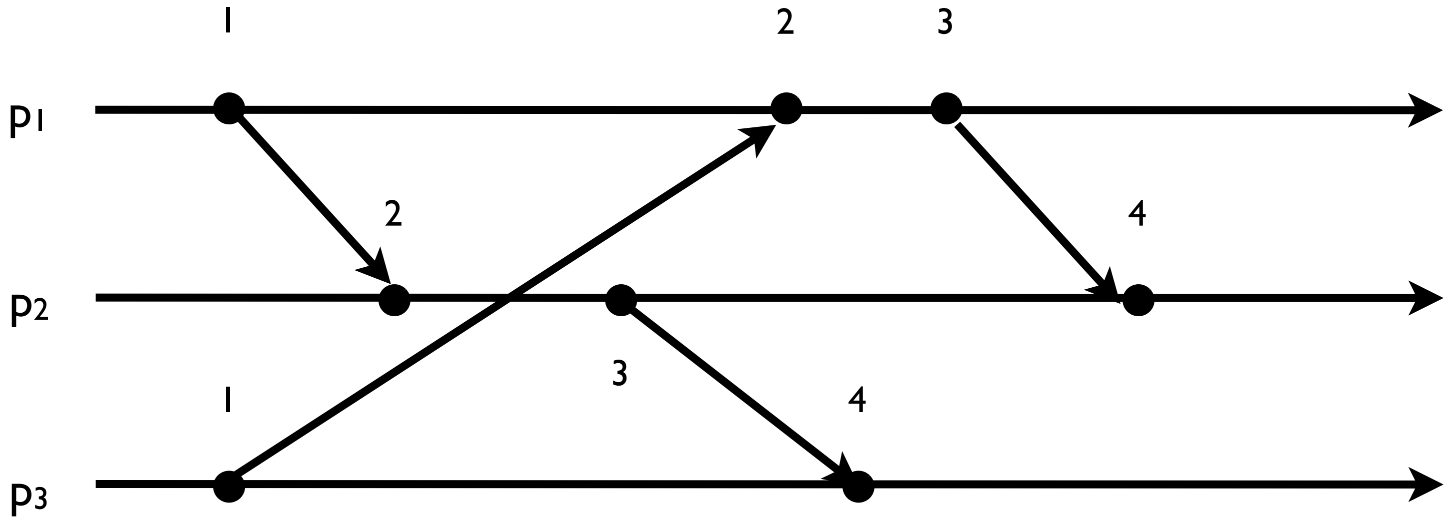


- We define the **happened-before relation** → by the three rules:
  1. If e_1 and e_2 are two events that happen in a single process and e_1 precedes e_2 then $e_1 \rightarrow e_2$
  2. If e_1 is the sending of message m and e_2 is the receiving of the same message m then $e_1 \rightarrow e_2$
  3. If $e_1 \rightarrow e_2$ and $e_2 \rightarrow e_3$ then $e_1 \rightarrow e_3$
- If neither $e_1 \rightarrow e_2$ nor $e_2 \rightarrow e_1$ hold then e_1, e_2 are **concurrent** ($e_1 \| e_2$)

Logical Ordering — A Logical Clock

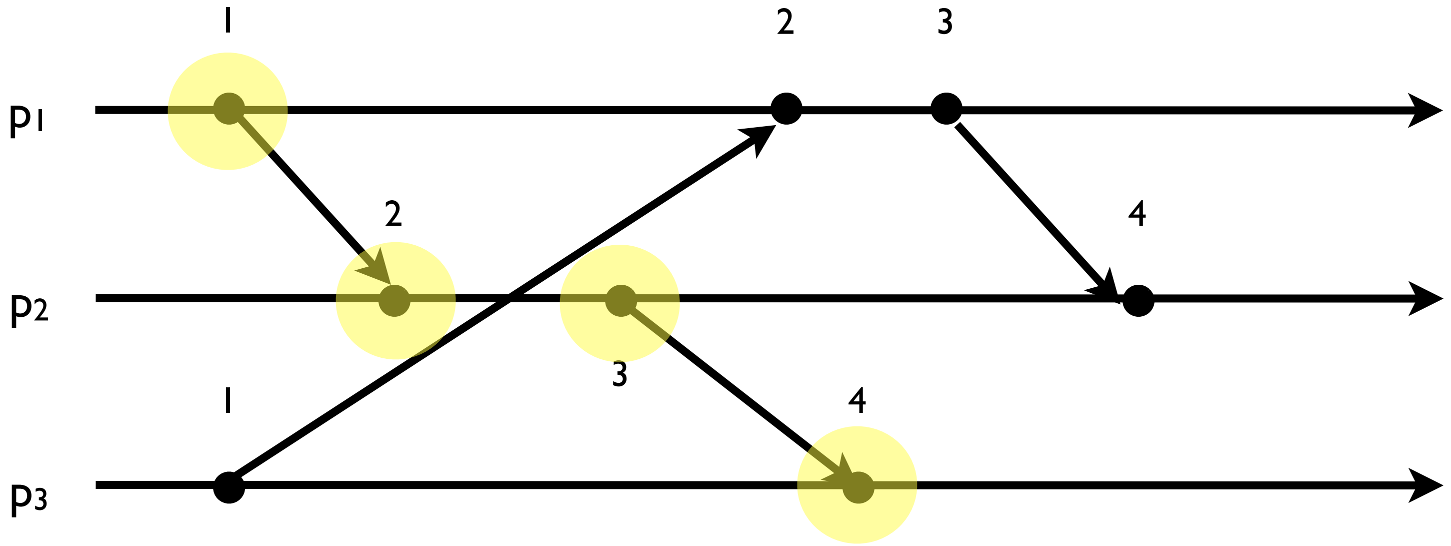
- Lamport designed an algorithm whereby events in a logical order can be given a numerical value
- This is a **logical clock**,
  - similar to a program counter except that there is no backward jumping
  - so it is monotonically increasing
- Each process P_i maintains its internal logical clock L_i
- So in order to record the logical ordering of events, each process does the following:
  - L_i is incremented immediately before each event is issued at P_i
  - When the process P_i sends a message m it piggybacks the value of its logical clock $t = L_i(m)$ - sending (m,t).
  - Upon receiving a message (m,t) process P_j computes the new value of L_j as $max(L_j,t)$ (and then processes m as usual)

Logical clocks: Example



- Note that *e*'s timestamp is the length of the longest chain of events that happened before *e*

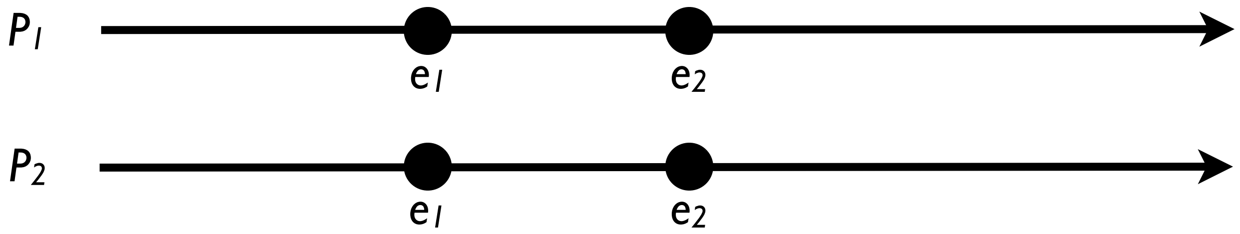
Logical clocks: Example



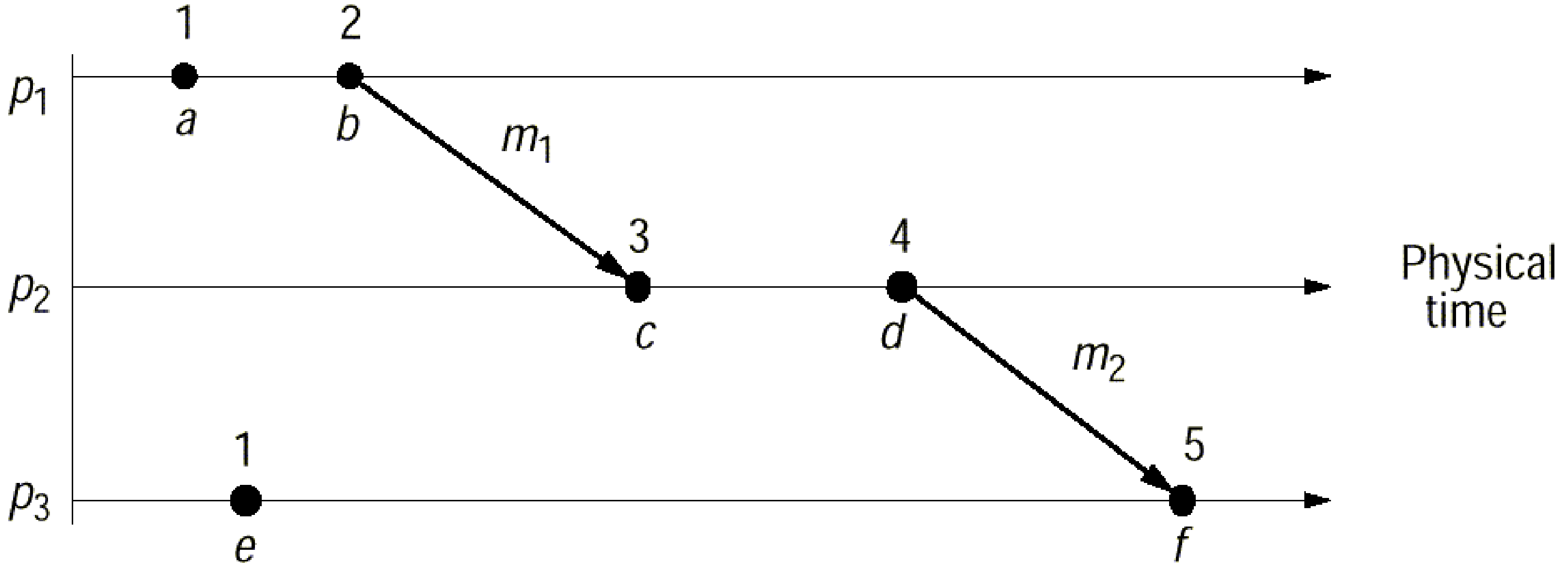
- Note that *e*'s timestamp is the length of the longest chain of events that happened before *e*

Logical Clocks: Properties

- Key point: using induction we can show that:
  - $e_1 \rightarrow e_2$ implies that $L(e_1) < L(e_2)$
- However, the converse is not true, that is:
  - $L(e_1) < L(e_2)$ does not imply that $e_1 \rightarrow e_2$
- It is easy to see why, consider two processes, P_1 and P_2 which each perform two steps prior to any communication.
- The two steps on the first process P_1 are concurrent with both of the two steps on process P_2.
- In particular $P_1(e_2)$ is concurrent with $P_2(e_1)$ but $L(P_1(e_2)) = 2$ and $L(P_2(e_1)) = 1$



No reverse implication



- Clock values *L(e)<L(b)<L(c)<L(d)<L(f)*
- but only *e*→*f*
- while *e* is concurrent with *b*, *c* and *d*.

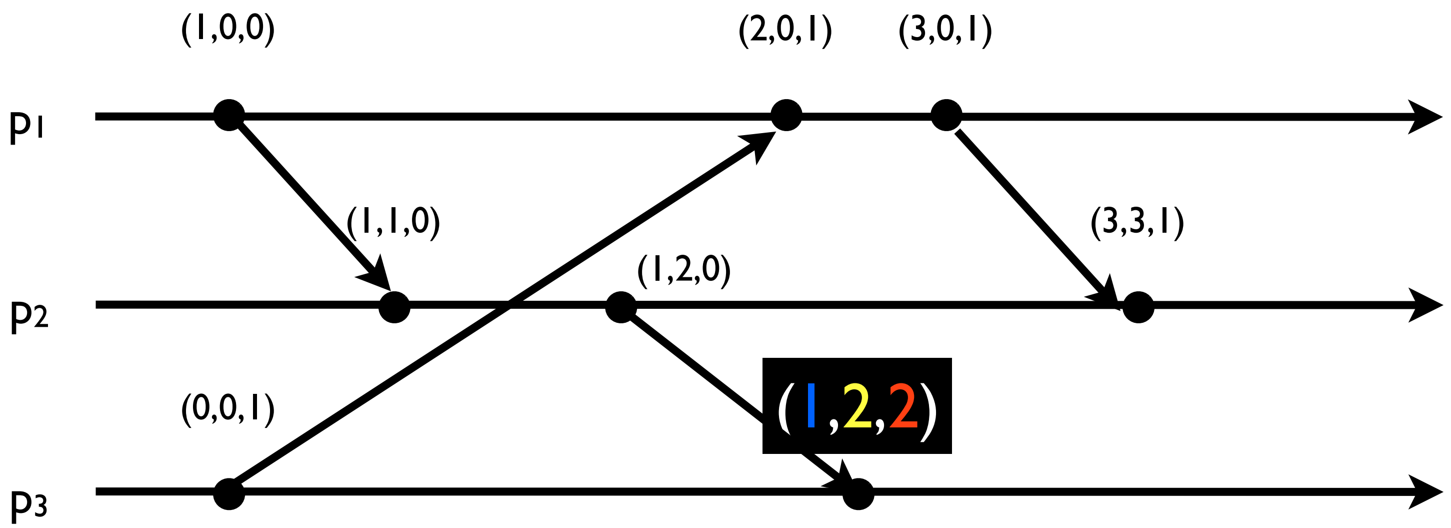
Total ordering

- The happened-before relation is a partial ordering
- The numerical Lamport stamps attached to each event are not unique
  - That is, some (concurrent) events can have the same number attached.
- However we can make it a total ordering by considering the process identifier at which the event took place
- In this case $(L_i(e_1),i) < (L_j(e_2),j)$ if either:
  - $L_i(e_1) < L_j(e_2)$ OR
  - $L_i(e_1) = L_j(e_2)$ AND $i<j$
- This has no physical meaning but can be useful for tie-breaking

Vector Clocks

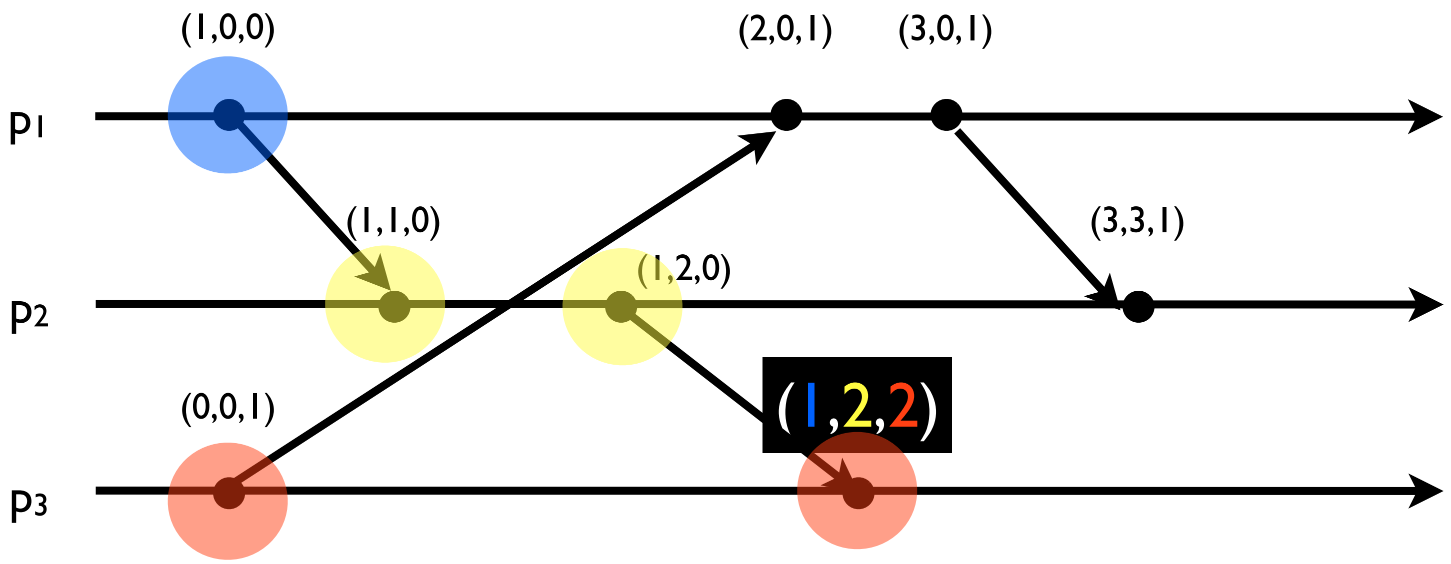
- Vector clocks were developed (by Mattern and Fidge) to overcome the problem of the lack of a reversed implication
- That is: $L(e_1) < L(e_2)$ does not imply $e_1 \rightarrow e_2$
- Each process keeps it own vector clock V_i (an **array** of Lamport clocks, **one for every process**)
- The vector clocks are updated according to the following rules:
  - Initially $V_i = (0,...,0)$
  - As with Lamport clocks before each event at process P_i it updates its own Lamport clock within the vector: $V_i[i] = V_i[i] + 1$
  - Every message P_i sends "piggybacks" its entire vector clock $t = V_i$
  - When P_i receives a timestamp Vx then it updates all of its vector clocks with: $V_i[j] = max(V_i[j],V_x[j])$

Vector Clocks illustrated



Invariant: $V_i[j]$ is the number of events in process P_j that *happened before* current state of process P_i

Vector Clocks illustrated



Invariant: $V_i[j]$ is the number of events in process P_j that *happened before* current state of process P_i

Vector Clocks: correctness

- Vector clocks (or timestamps) are compared as follows:
  - $V_x = V_y$ iff $V_x[i] = V_y[i]$ $\forall i, 1 \ldots N$
  - $V_x \leq V_y$ iff $V_x[i] \leq V_y[i]$ $\forall i, 1 \ldots N$
  - $V_x < V_y$ iff $V_x[i] < V_y[i]$ $\forall i, 1 \ldots N$
- For example (1,2,1) < (3,2,1) but not < (3,1,2)
  - It's not a total order: (1,0,1) and (0,1,0) incomparable!
- As with logical clocks: $e_1 \rightarrow e_2$ implies $V(e_1) < V(e_2)$
- In contrast with logical clocks the reverse is also true: $V(e_1) < V(e_2)$ implies $e_1 \rightarrow e_2$

Vector Clocks

- Vector Clocks augment Logical Clocks
  - Of course vector clocks achieve this at the cost of larger time stamps attached to each message
  - In particular the size of the timestamps grows proportionally with the number of communicating processes
- Summary of Logical Clocks
  - We cannot achieve arbitrary precision of synchronization between remote clocks via message passing
  - We are forced to accept that some events are concurrent, meaning that we have no way to determine which occurred first
  - Despite this we can still achieve a logical ordering of events that is useful for many applications

Global State

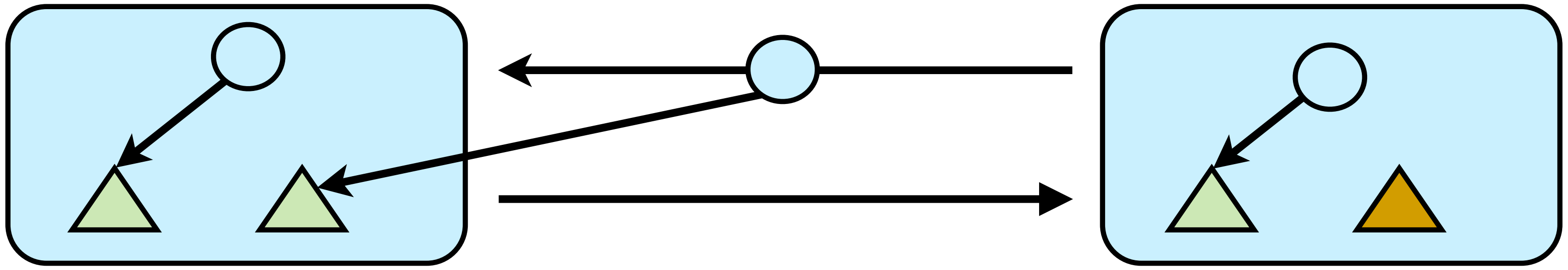
- Correctness of distributed systems frequently hinges upon satisfying some global system invariant
- Even for applications in which you do not expect your algorithm to be correct at all times, it may still be desirable that it is “good enough” at all times
- For example our distributed algorithm may be maintaining a record of all transactions
- In this case it might be okay if some processes are behind other processes and thus do not know about the most recent transactions
- But we would never want it to be the case that some process is in an inconsistent state, say applying a single transaction twice.

Global state: Motivating examples

1. Distributed garbage collection
2. Distributed deadlock detection
3. Distributed termination detection
4. Distributed debugging

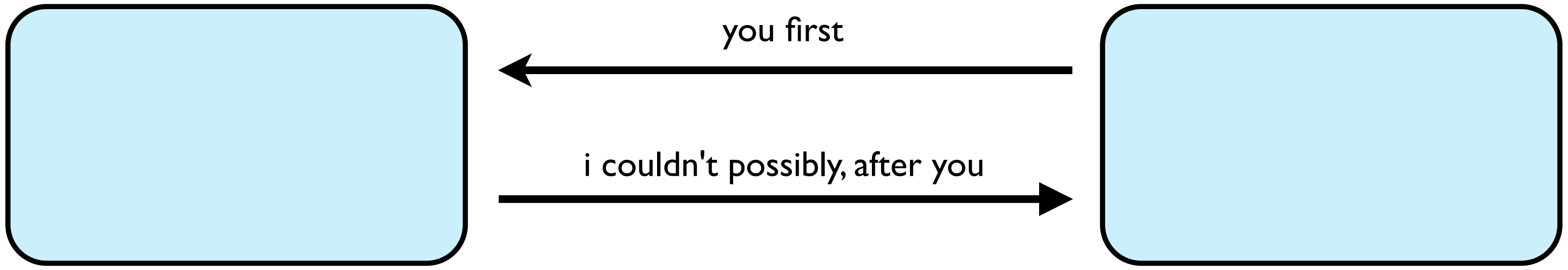
- Let's consider the impact of global time on these problems

Distributed Garbage Collection



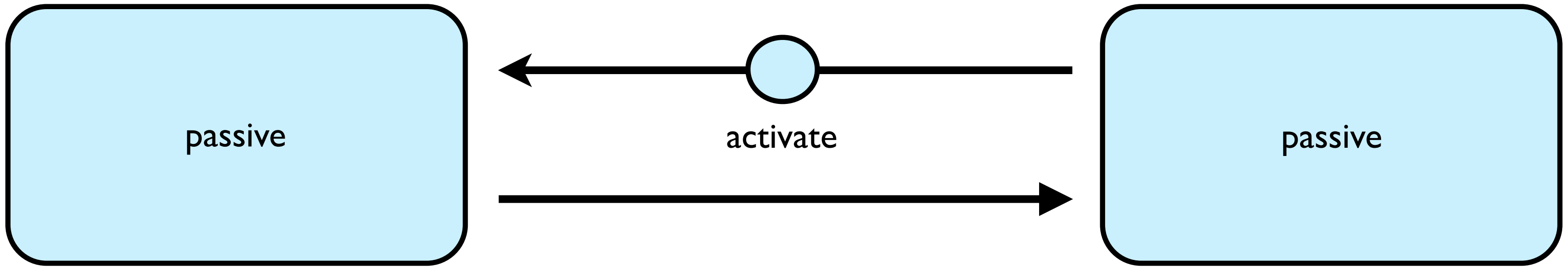
- Determine whether a given resource is "live" (referenced by any processes/messages in transit)
- What if we had a global clock?
  - Agree a global time for each process to check whether a reference exists to a given object
  - This leaves the problem that a reference may be in transit between processes
  - But each process can say which references they have sent before the agreed time and compare that to the references received at the agreed time

Distributed Deadlock Detection



- Determine whether processes are "stuck" waiting for messages from each other.
- What if we had a global clock?
  - At an agreed time all processes send to some master process the processes or resources for which they are waiting
  - The master process then simply checks for a loop in the resulting graph

Distributed Termination Detection



- Determine if all processes are "done" and no messages are in-transit
- What if we had a global clock?
    - At an agreed time each process sends whether or not they have completed to a master process
    - Again this leaves the problem that a message may be in transit at that time
    - Again though, we should be able to work out which messages are still in transit

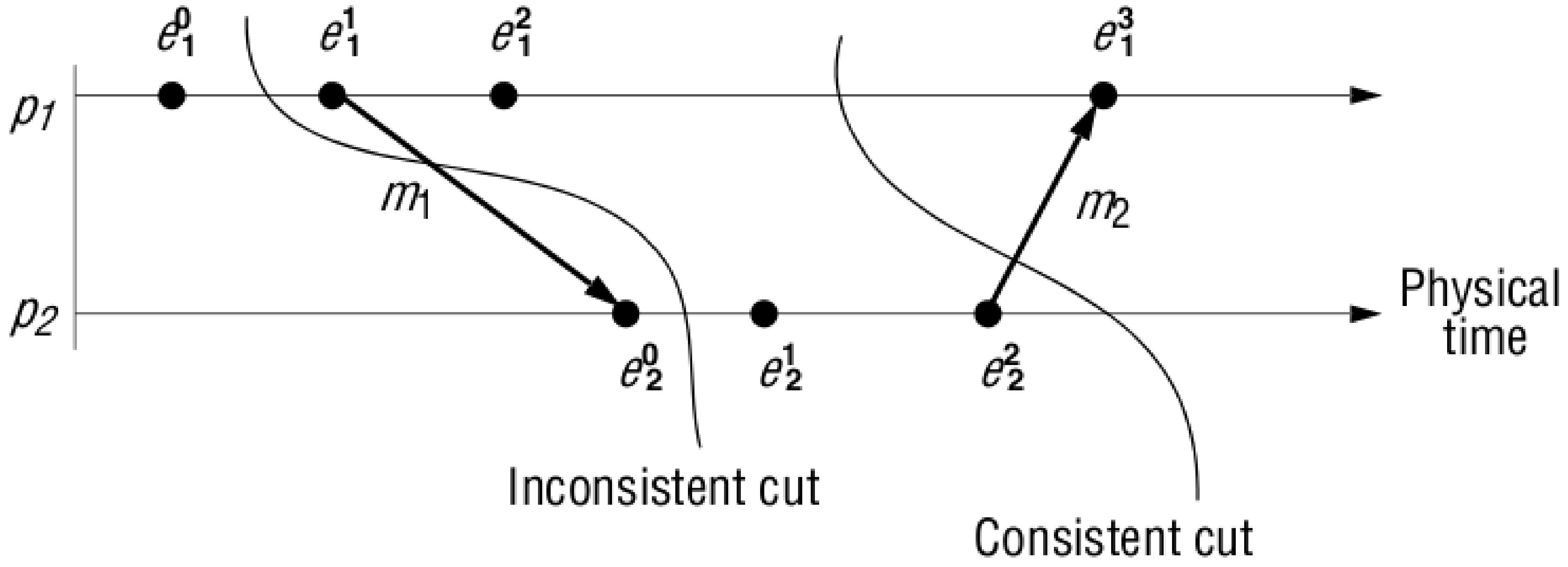
Distributed Debugging

- Compute some property of the combined state of all processes (and channels)
- What if we had a global clock?
  - At each point in time we can reconstruct the global state
  - We can also record the entire history of events in the exact order in which they occurred.
  - Allowing us to replay them and inspect the global state to see where things have gone wrong as with traditional debugging

Global State: Consistent Cuts

- The global state is the combination of all process states and the states of the communication channels at an instant in time
  - So, if we had synchronized clocks, we could agree on a time for each process to record its state
- Since we cannot "stop time" to observe the *actual* global state, we attempt to find *possible* global state(s)
- A **cut** is a collection of prefix of the (combined) histories of the processes
  - partitioning all events into those occurring "before" and "after" the cut
- The goal is to assemble a meaningful global state from the the local states of processes
  - recorded at (possibly) different but concurrent times

Consistent Cuts



- A **consistent cut** is one which does not violate the happens-before relation →
- If $e_1 \rightarrow e_2$ then either:
  - both e_1 and e_2 are before the cut <u>or</u>
  - both e_1 and e_2 are after the cut <u>or</u>
  - e_1 is before the cut and e_2 is after the cut
  - <u>but not</u> e_1 is after the cut and e_2 is before the cut

Summary

- Lamport and Vector clocks were introduced:
  - Lamport clocks $e_1 \rightarrow e_2 \Rightarrow L(e_1) < L(e_2)$
  - Vector clocks additionally satisfying $V(e_1) < V(e_2) \Rightarrow e_1 \rightarrow e_2$
  - But at the cost of message length and scalability
- The concept of a true history of events as opposed to runs and linearizations was introduced
- **Next time:**
  - Chandy and Lamport's algorithm for recording a global snapshot of the system
  - Distributed debugging