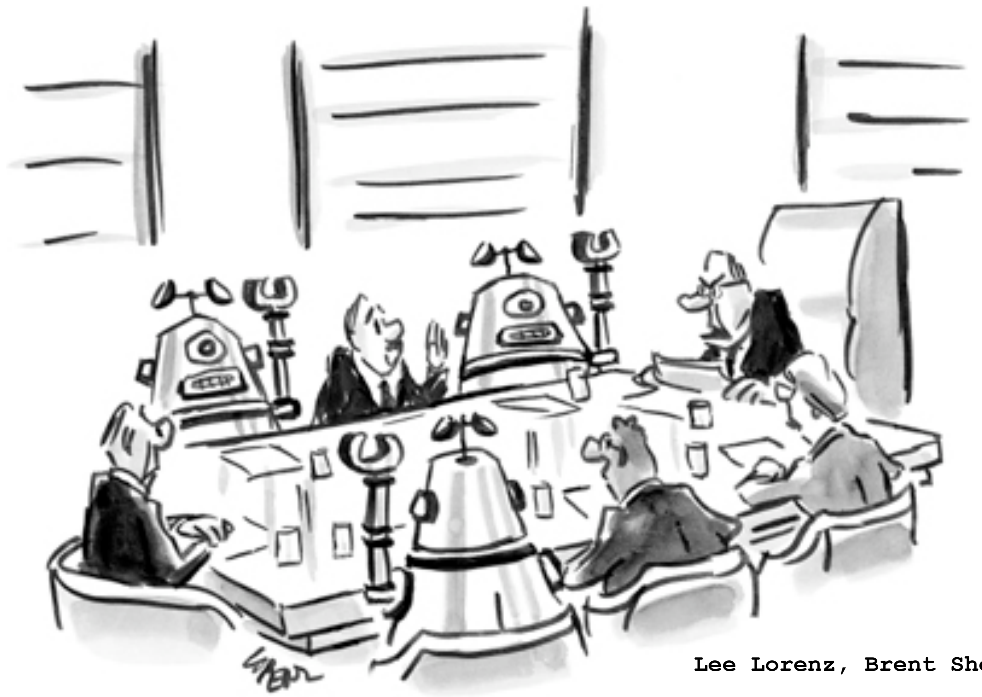


Lee Lorenz, Brent Sheppard

Jenkins, if I want another yes-man, I'll build one!

Versioning, Consistency, and Agreement

COS 461: Computer Networks
Spring 2010 (MW 3:00-4:20 in CS105)

Mike Freedman
http://www.cs.princeton.edu/courses/archive/spring10/cos461/

Time and distributed systems

- With multiple events, what happens first?

A shoots B

B dies

Time and distributed systems

- With multiple events, what happens first?

`B shoots A`

`A dies`

Time and distributed systems

- With multiple events, what happens first?

A shoots B

A dies

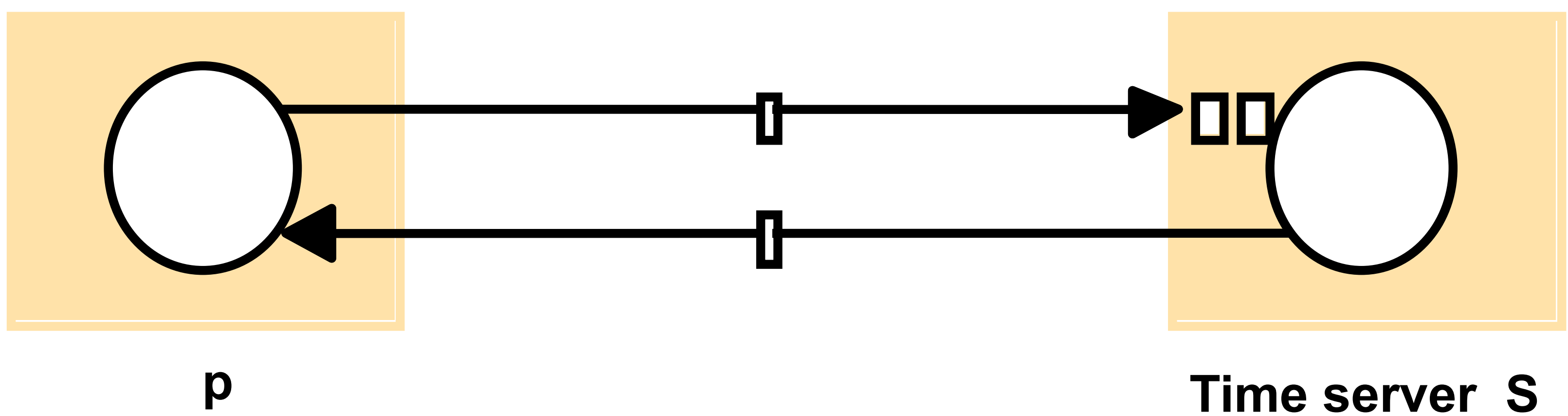
B shoots A

B dies

Just use time stamps?

- Need synchronized clocks

- Clock synch via a time server



Cristian's Algorithm

- Uses a *time server* to synchronize clocks
- Time server keeps the reference time
- Clients ask server for time and adjust their local clock, based on the response
  - But different network latency → clock skew?
- Correct for this? For links with symmetrical latency:

```
RTT = response-received-time – request-sent-time

adjusted-local-time = server-timestamp t + (RTT / 2)

local-clock-error = adjusted-local-time – local-time
```

Is this sufficient?

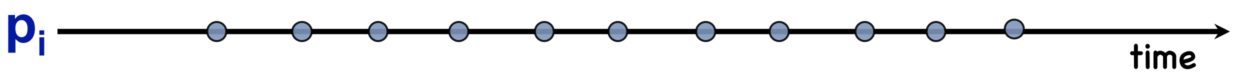
- Server latency due to load?
  - If can measure:
    - adjusted-local-time = server-time t + (RTT+ lag) / 2
- But what about asymmetric latency?
  - RTT / 2 not sufficient!
- What do we need to measure RTT?
  - Requires no clock drift!
- What about “almost” concurrent events?
  - Clocks have micro/milli-second precision

Events and Histories

- Processes execute sequences of events
- Events can be of 3 types:
  - local, send, and receive
- The local history h_p of process p is the sequence of events executed by process

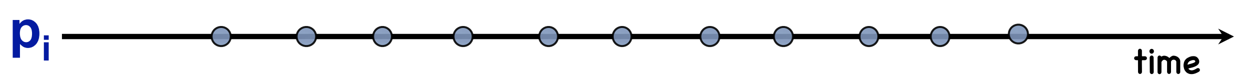
Ordering events

- Observation 1:
  - Events in a local history are totally ordered

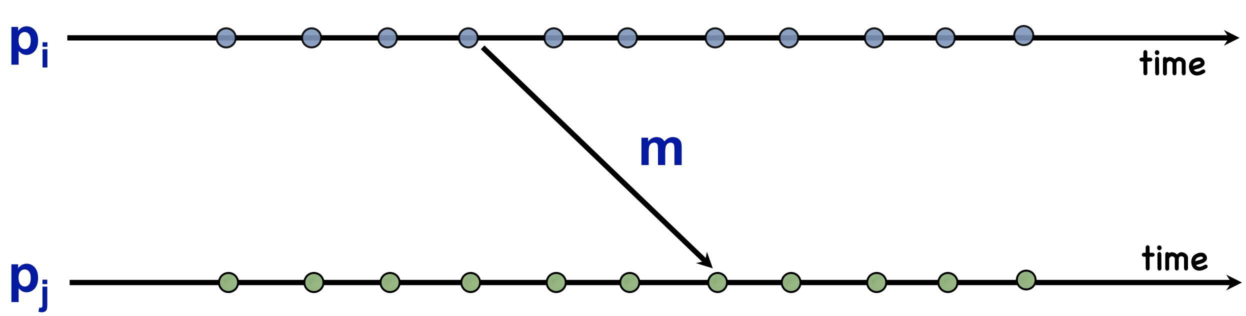


Ordering events

- Observation 1:
  - Events in a local history are totally ordered



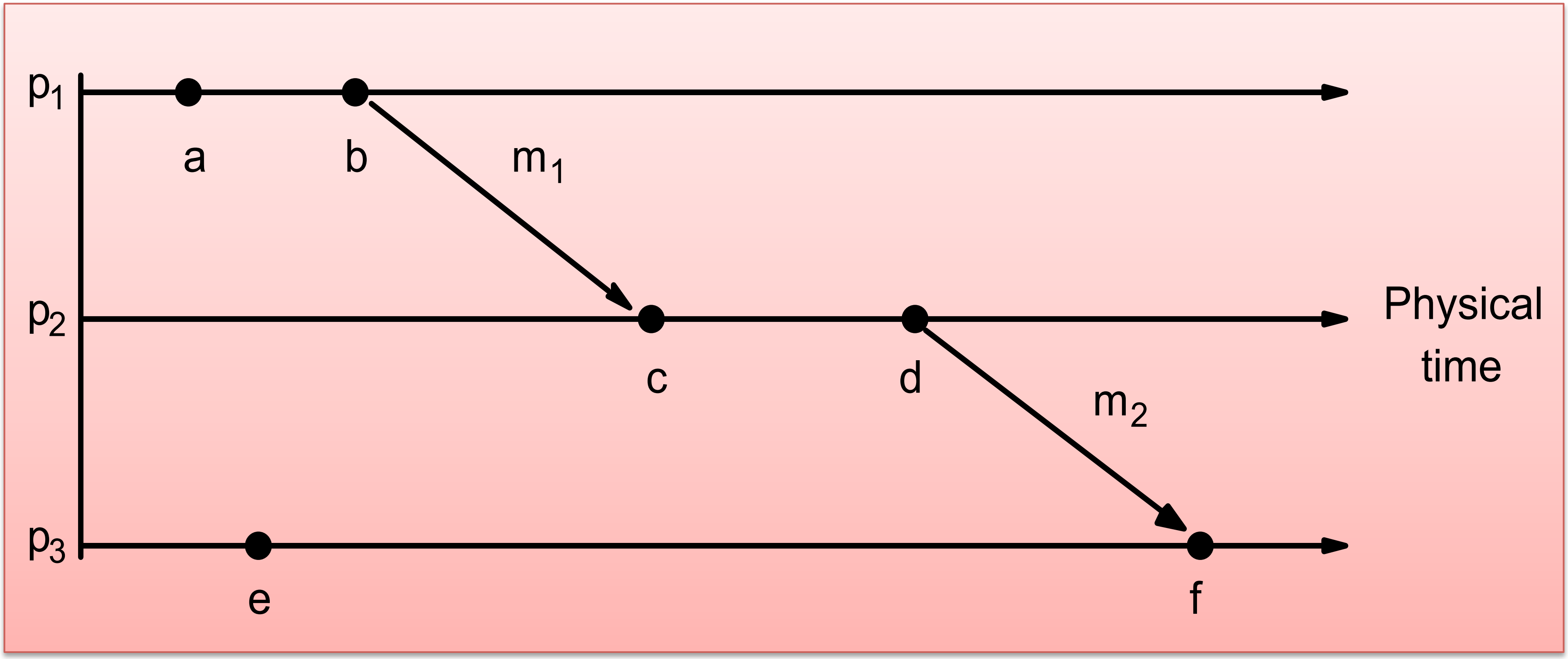
- Observation 2:
  - For every message m, send(m) precedes receive(m)



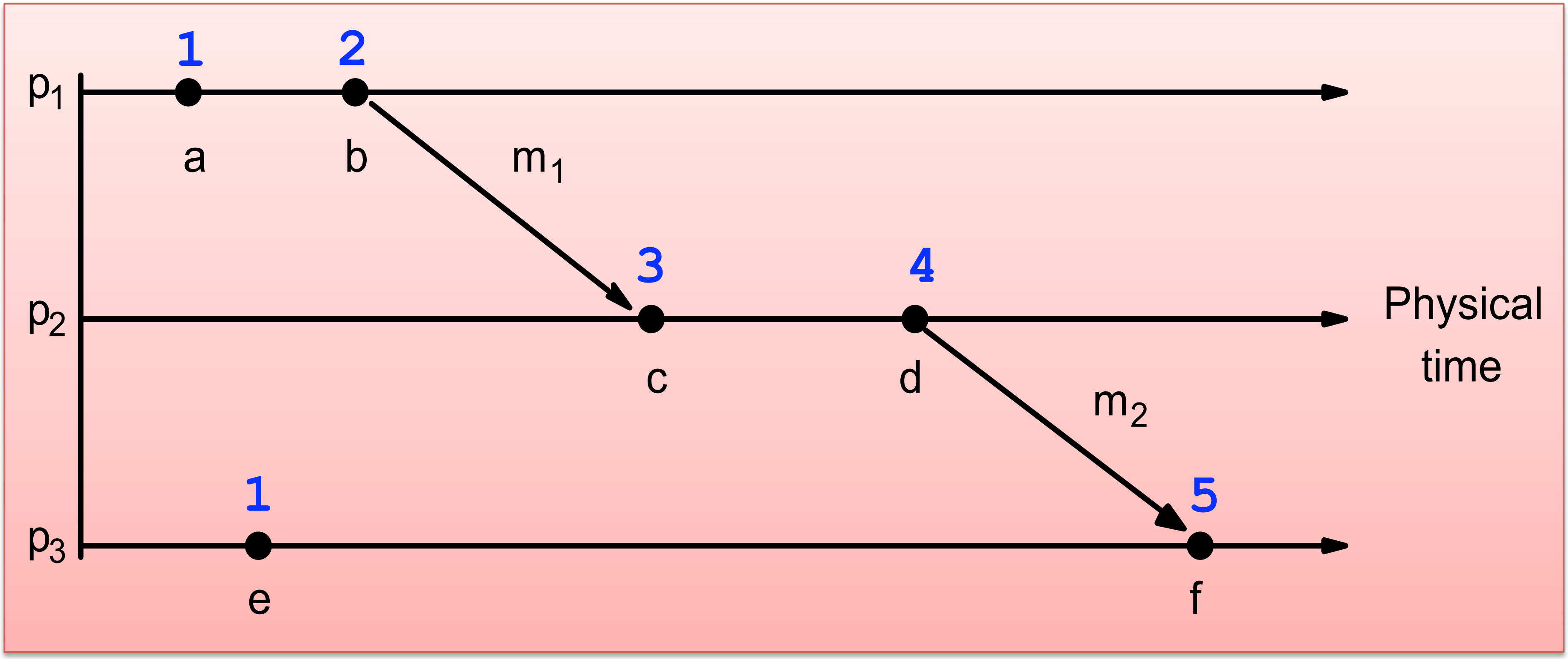
Happens-Before (Lamport [1978])

- Relative time? Define *Happens-Before* ($\rightarrow$) :
  - On the same process: $a \rightarrow b$, if $time(a) < time(b)$
  - If p1 sends *m* to p2: $send(m) \rightarrow receive(m)$
  - If $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$
- Lamport Algorithm uses for partial ordering:
  - All processes use a counter (clock) with initial value of 0
  - Counter incremented by and assigned to each event, as its timestamp
  - A send (msg) event carries its timestamp
  - For receive (msg) event, counter is updated by Max (receiver-counter, message-timestamp) + 1

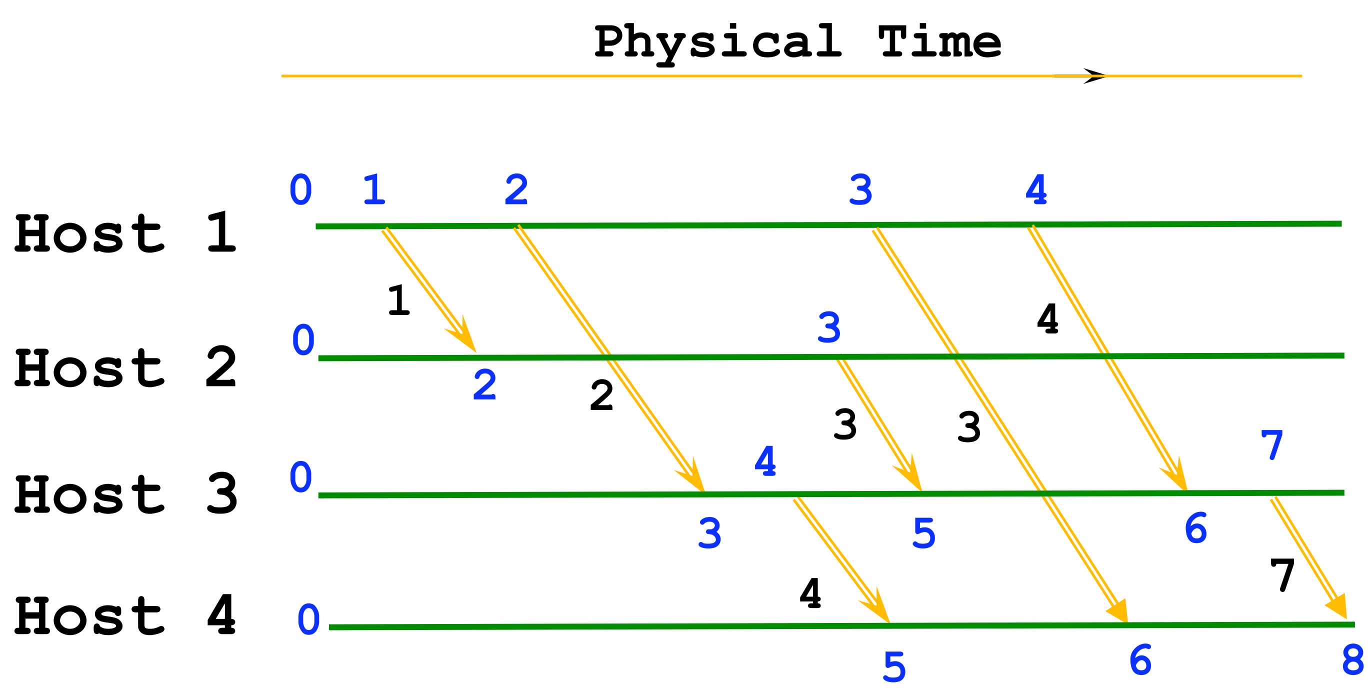
Events Occurring at Three Processes



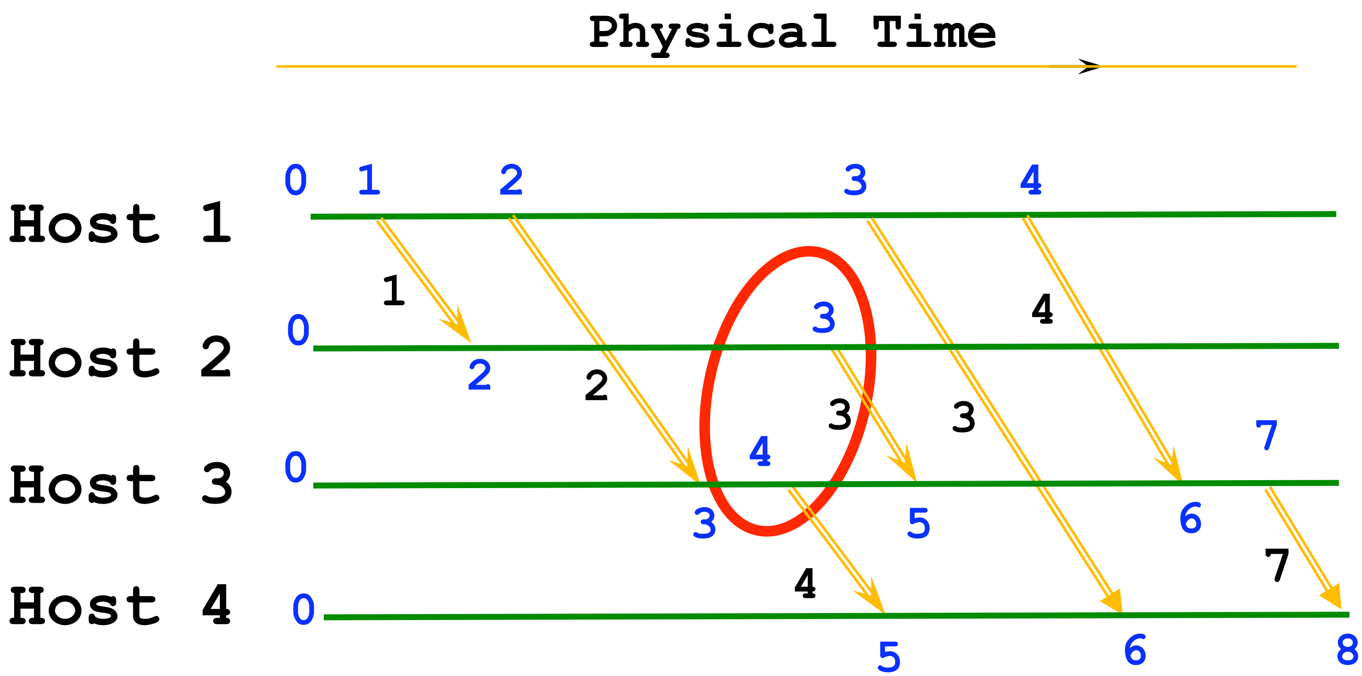
Lamport Timestamps

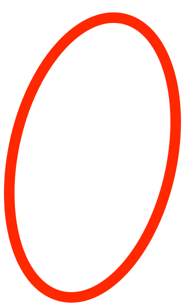


Lamport Logical Time



Lamport Logical Time





Logically concurrent events!

Vector Logical Clocks

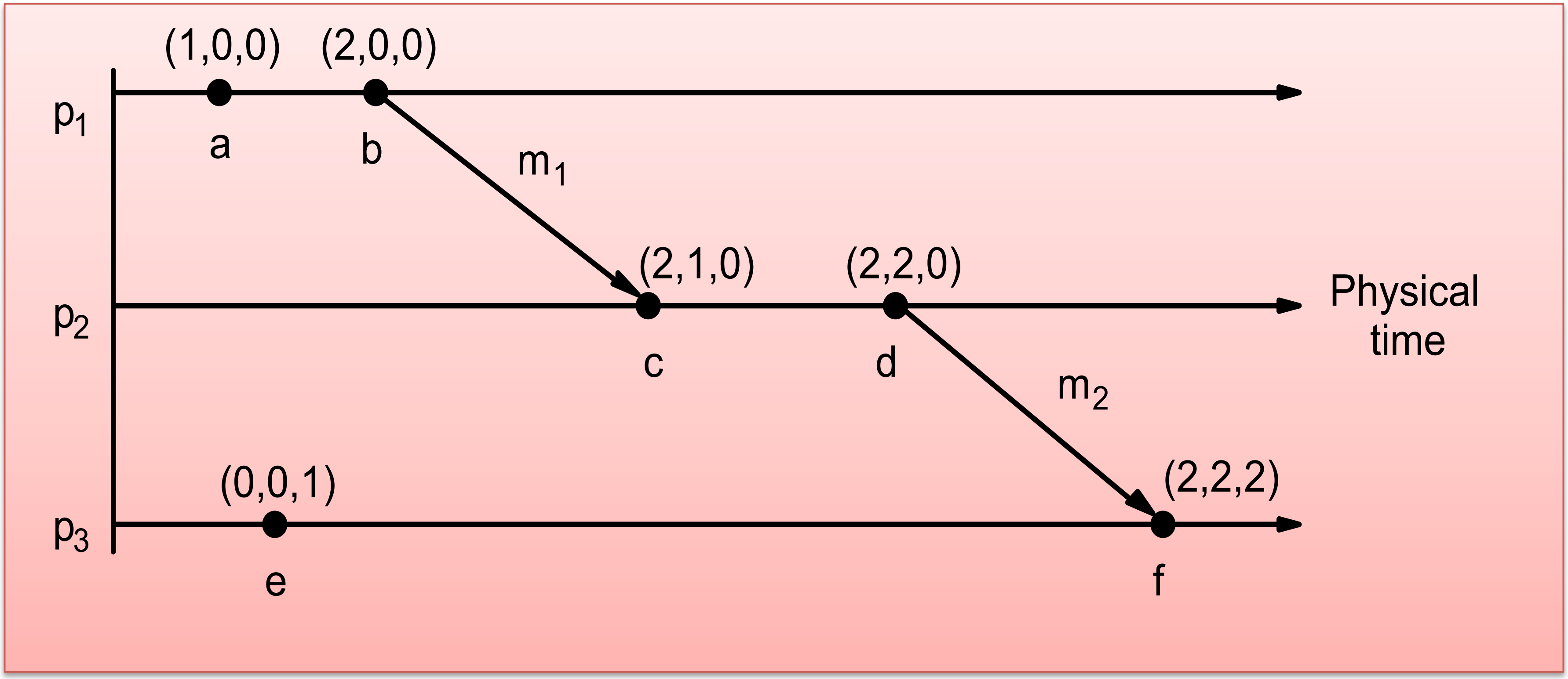
- With Lamport Logical Time
  - e precedes f $\Rightarrow$ timestamp(e) < timestamp (f), but
  - timestamp(e) < timestamp (f) $\Rightarrow$ e ~~precedes~~ f

Vector Logical Clocks

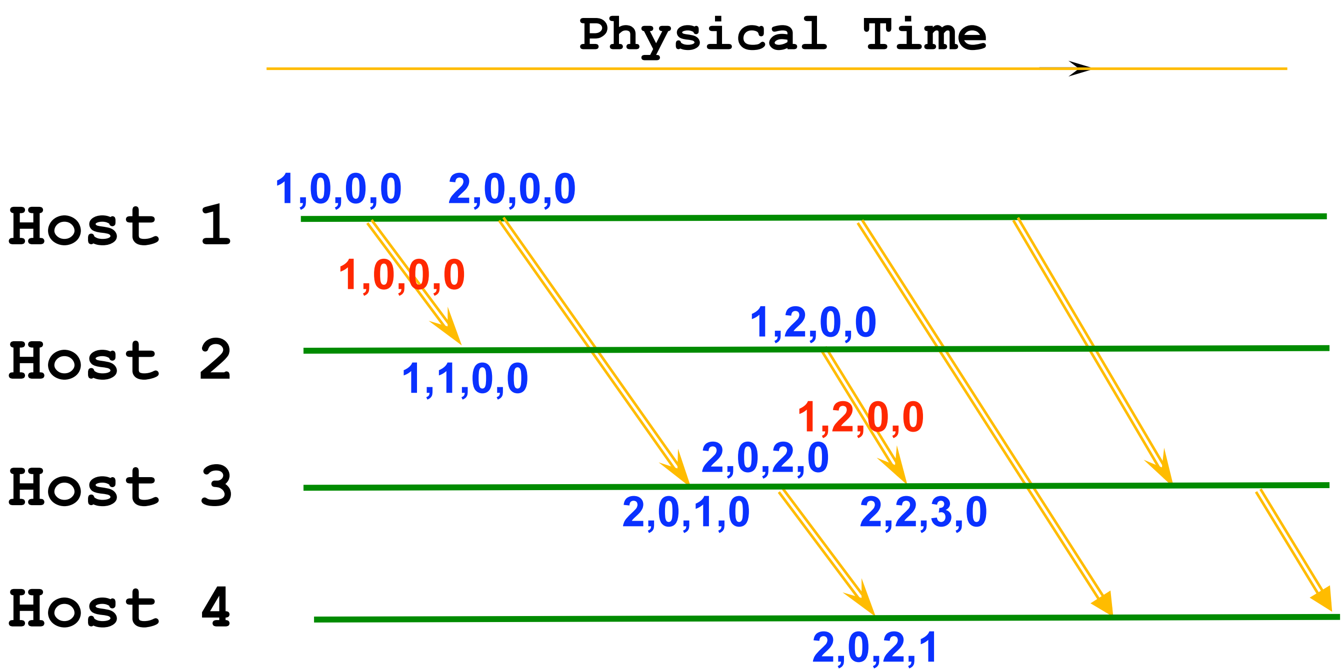
- With Lamport Logical Time
  - e precedes f $\Rightarrow$ timestamp(e) < timestamp (f), but
  - timestamp(e) < timestamp (f) $\Rightarrow$ e ~~precedes~~ f
- Vector Logical time guarantees this:
  - All hosts use a vector of counters (logical clocks), i^{th} element is the clock value for host i, initially 0
  - Each host i, increments the i^{th} element of its vector upon an event, assigns the vector to the event.
  - A send(msg) event carries vector timestamp
  - For receive(msg) event,

$$\mathbf{V_{receiver}[j]} = \begin{cases} \text{Max } (V_{receiver}[j] , V_{msg}[j]), & \text{if } j \text{ is not self} \\ V_{receiver}[j] + 1 & \text{otherwise} \end{cases}$$

Vector Timestamps



Vector Logical Time



$$V_{receiver}[j] = \begin{cases} \text{Max } (V_{receiver}[j]\,,\, V_{msg}[j]), & \text{if j is not self} \\ V_{receiver}[j] + 1 & \text{otherwise} \end{cases}$$

Comparing Vector Timestamps

- a = b if they agree at every element
- a < b if a[i] <= b[i] for every i, but !(a = b)
- a > b if a[i] >= b[i] for every i, but !(a = b)
- a || b if a[i] < b[i], a[j] > b[j], for some i,j (*conflict!*)

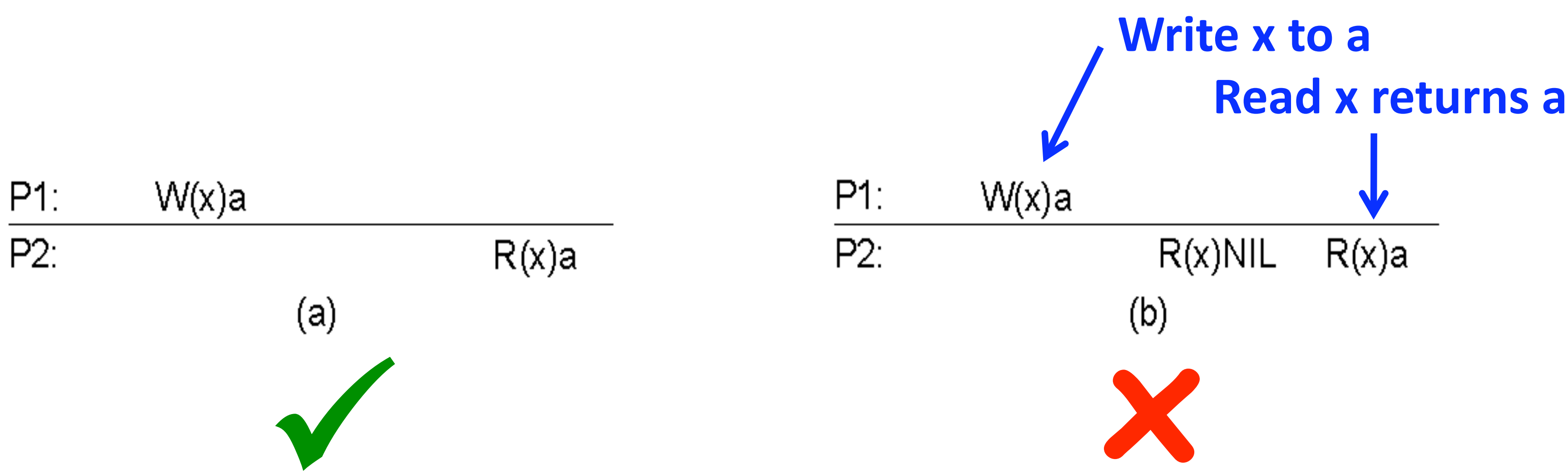
- If one history is prefix of other, then one vector timestamp < other
- If one history is not a prefix of the other, then (at least by example) VTs will not be comparable.

Given a notion of time…

…What's a notion of consistency?

Strict Consistency

- Strongest consistency model we'll consider
  - Any read on a data item X returns value corresponding to result of the most recent write on X
- Need an absolute global time
  - "Most recent" needs to be unambiguous



What else can we do?

- Strict consistency is the ideal model
  - But impossible to implement!
- Sequential consistency
  - Slightly weaker than strict consistency
  - Defined for shared memory for multi-processors

Sequential Consistency

- Definition:

  Result of any execution is the same as if all (read and write) operations on data store were executed in *some* sequential order, and the operations of each individual process appear in this sequence in the order specified by its program

- Definition: When processes are running concurrently:
  - Interleaving of read and write operations is acceptable, but all processes see the same interleaving of operations

- Difference from strict consistency
  - No reference to the most recent time
  - Absolute global time does not play a role

Valid Sequential Consistency?

P1:	W(x)a				
P2:		W(x)b			
P3:			R(x)b		R(x)a
P4:				R(x)b	R(x)a

(a)

P1:	W(x)a				
P2:		W(x)b			
P3:			R(x)b		R(x)a
P4:				R(x)a	R(x)b

(b)

Linearizability

- Linearizability
  - Weaker than strict consistency
  - Stronger than sequential consistency
- All operations (OP = read, write) receive a global time-stamp using a synchronized clock
- Linearizability:
  - Requirements for sequential consistency, plus
  - If $ts_{op1}(x) < ts_{op2}(y)$, then OP1(x) should precede OP2(y) in the sequence

Causal Consistency

- Necessary condition:
  - Writes that are ***potentially*** causally related must be seen by all processes in the same order.
  - Concurrent writes may be seen in a different order on different machines.

- Weaker than sequential consistency

- *Concurrent*: Ops that are not causally related

Causal Consistency

P1:	W(x)a			W(x)c		
P2:		R(x)a	W(x)b			
P3:		R(x)a			R(x)c	R(x)b
P4:		R(x)a			R(x)b	R(x)c

- Allowed with causal consistency, but not with sequential or strict consistency
- *W(x)b* and *W(x)c* are concurrent
  - So all processes don't see them in the same order
- P3 and P4 read the values 'a' and 'b' in order as potentially causally related. No 'causality' for 'c'.

Causal Consistency

P1: W(x)a				
P2:	R(x)a	W(x)b		
P3:			R(x)b	R(x)a
P4:			R(x)a	R(x)b

(a)

P1: W(x)a			
P2:	W(x)b		
P3:		R(x)b	R(x)a
P4:		R(x)a	R(x)b

(b)

Causal Consistency

- Requires keeping track of which processes have seen which writes
  - Needs a dependency graph of which op is dependent on which other ops
  - …or use vector timestamps!

Eventual consistency

- If no new updates are made to an object, after some inconsistency window closes, all accesses will return the last updated value

- Prefix property:
  - If Pi has write w accepted from some client by Pj
  - Then Pi has all writes accepted by Pj prior to w

- Useful where concurrency appears only in a restricted form

- Assumption: write conflicts will be easy to resolve
  - Even easier if whole-"object" updates only

Systems using eventual consistency

- DB: updated by a few proc's, read by many
  - How fast must updates be propagated?
- Web pages: typically updated by single user
  - So, no write-write conflicts
  - However caches can become inconsistent

Systems using eventual consistency

- DNS: each domain assigned to a naming authority
  - Only master authority can update the name space
  - Other NS servers act as “slave” servers, downloading DNS zone file from master authority
  - So, write-write conflicts won’t happen

```
$ ORIGIN coralcdn.org.
        @    IN    SOA    ns3.fs.net.    hostmaster.scs.cs.nyu.edu. (
                  18          ; serial
                  1200        ; refresh
                  600         ; retry
                  172800              ; expire
                  21600  )    ; minimum
```

  - Is this always true today?

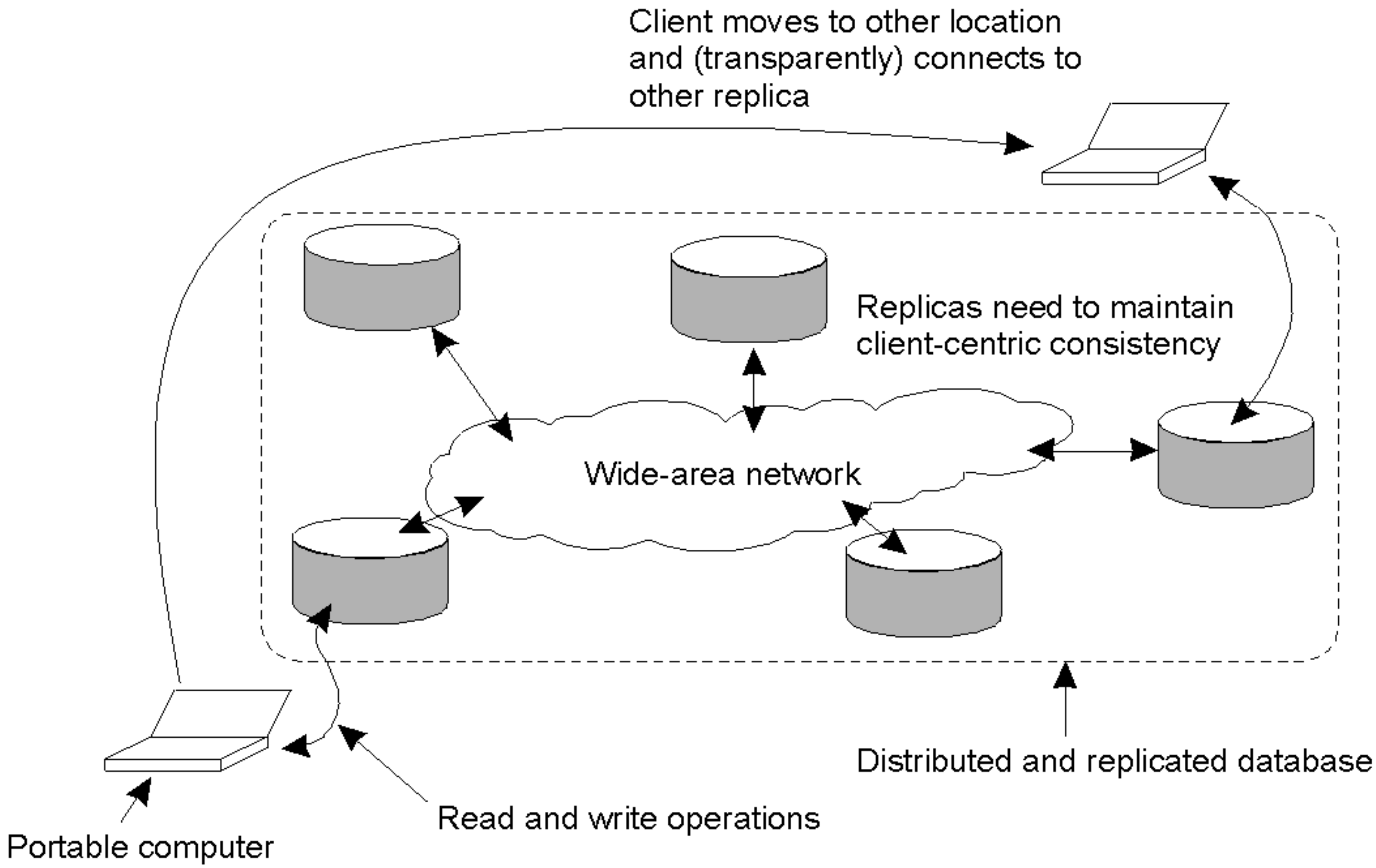
Typical implementation of eventual consistency

- Distributed, inconsistent state
  - Writes only go to some subset of storage nodes
    - By design (for higher throughput)
    - Due to transmission failures
- "Anti-entropy" (gossiping) fixes inconsistencies
  - Use vector clock to see which is older
  - Prefix property helps nodes know consistency status
  - If automatic, requires some way to handle write conflicts
    - Application-specific merge() function
    - Amazon's Dynamo: Users may see multiple concurrent "branches" before app-specific reconciliation kicks in

Examples…

- Causal consistency. Non-causally related subject to normal eventual consistency rules
- Read-your-writes consistency.
- Session consistency. Read-your-writes holds iff client session exists. If session terminates, no guarantees between sessions.
- Monotonic read consistency. Once read returns a version, subsequent reads never return older versions.
- Monotonic write consistency. Writes by same process are properly serialized. Really hard to program systems without this process.

Even read-your-writes may be difficult to achieve



What about stronger agreement?

- Two-phase commit protocol

  - Marriage ceremony

    Do you?
    I do.
    I now pronounce you…

  - Theater

    Ready on the set?
    Ready!
    Action!

  - Contract law

    Offer
    Signature
    Deal / lawsuit

What about stronger agreement?

- Two-phase commit protocol

What about failures?

- If an acceptor fails:
  - Can still ensure linearizability if $|R| + |W| \geq N$
  - "read" and "write" quorums overlap in at least 1 node
- If the leader fails?
  - Lose availability: system not longer "live"
- Pick a new leader?
  - Need to make sure everybody agrees on leader!
  - Need to make sure that "group" is known

Consensus and Paxos Algorithm

- "Consensus" problem
  – N processes want to agree on a value
  – If fewer than F faults in a window, consensus achieved
    - "Crash" faults need 2F+1 processes
    - "Malicious" faults (called Byzantine) need 3F+1 processes
- Collection of processes proposing values
  – Only proposed value may be chosen
  – Only single value chosen
- Common usage:
  – View change: define leader and group via Paxos
  – Leader uses two-phase commit for writes
  – Acceptors monitor leader for liveness. If detect failure, re-execute "view change"

Paxos: Algorithm

View Change from current view

View i: V = { Leader: N2, Group: {N1, N2, N3} }

Phase 1 (Prepare)

- Proposer: Send *prepare* with version# *j* to members of View i
- Acceptor: if *j > vers #* k of any other *prepare* it seen, respond with promise not to accept lower-numbered proposals. Otherwise, respond with k and value v' accepted.

Phase 2 (Accept)

- If majority promise, proposer sends accept with (vers j, value v)
- Acceptor accepts unless it has responded to *prepare* with higher vers # than j. Sends acknowledgement to all view members.

Summary

- Global time doesn't exist in distributed system
- Logical time can be established via version #'s
- Logical time useful in various consistency models
  - Strict > Linearizability > Sequential > Causal > Eventual
- Agreement in distributed system
  - Eventual consistency: Quorums + anti-entropy
  - Linearizability: Two-phase commit, Paxos