Distributed Systems
8L for Part IB

Handout 2

Dr. Steven Hand

Clocks

- Distributed systems need to be able to:
  - order events produced by concurrent processes;
  - synchronize senders and receivers of messages;
  - serialize concurrent accesses to shared objects; and
  - generally coordinate joint activity
- This can be provided by some sort of “clock”:
  - **physical clocks** keep time of day
    - (must be kept consistent across multiple nodes)
  - **logical clocks** keep track of event ordering

Physical Clock Technology

- Quartz Crystal Clocks (1929)
  - resonator shaped like a tuning fork
  - laser-trimmed to vibrate at 32,768 Hz
  - standard resonators accurate to 6ppm at 31°C... so will gain/lose around 0.5 seconds per day
  - stability better than accuracy (about 2s/month)
  - best resonators get accuracy of ~1s in 10 years
- Atomic clocks (1948)
  - count transitions of the caesium 133 atom
  - 9,192,631,770 periods defined to be 1 second
  - accuracy is better than 1 second in 6 million years...

Coordinated Universal Time (UTC)

- Physical clocks provide ‘ticks’ but we want to know the actual time of day
  - determined by astronomical phenomena
- Several variants of universal time
  - **UT0**: mean solar time on Greenwich meridian
  - **UT1**: UT0 corrected for polar motion; measured via observations of quasars, laser ranging, & satellites
  - **UT2**: UT1 corrected for seasonal variations
  - **UTC**: civil time, tracked using atomic clocks, but kept within 0.9s of UT1 by occasional leap seconds

Computer Clocks

- Typically have a real-time clock
  - CMOS clock driven by a quartz oscillator
  - battery-backed so continues when power is off
- Also have range of other clocks (PIT, ACPI, HPET, TSC, ...), mostly **higher frequency**
  - free running clocks driven by quartz oscillator
  - mapped to real time by OS at boot time
  - programmable to generate interrupts after some number of ticks (~= some amount of real time)

The Clock Synchronization Problem

- In distributed systems, we'd like all the different nodes to have the same notion of time, but
  - quartz oscillators oscillate at slightly different frequencies (time, temperature, manufacture)
- Hence clocks tick at different rates:
  - create ever-widening gap in perceived time
  - this is called **clock drift**
- The difference between two clocks at a given point in time is called **clock skew**
- Clock synchronization aims to minimize clock skew between two (or a set of) different clocks

Clock Skew and Clock Drift

08:00:00

08:00:00

February 18, 2012
08:00:00

Clock Skew and Clock Drift

08:01:24

Skew = *84 seconds*
Drift = *84s / 34 days*
= +2.47s per day

March 23, 2012
08:00:00

08:01:48

Skew = *108 seconds*
Drift = *108s / 34 days*
= +3.18s per day

Dealing with Drift

- A clock can have positive or negative drift with respect to a reference clock (e.g. UTC)
  - Need to [re]synchronize periodically
- Can’t just set clock to ‘correct’ time
  - Jumps (particularly backward!) can confuse apps
- Instead aim for gradual compensation
  - If clock fast, make it run slower until correct
  - If clock slow, make it run faster until correct

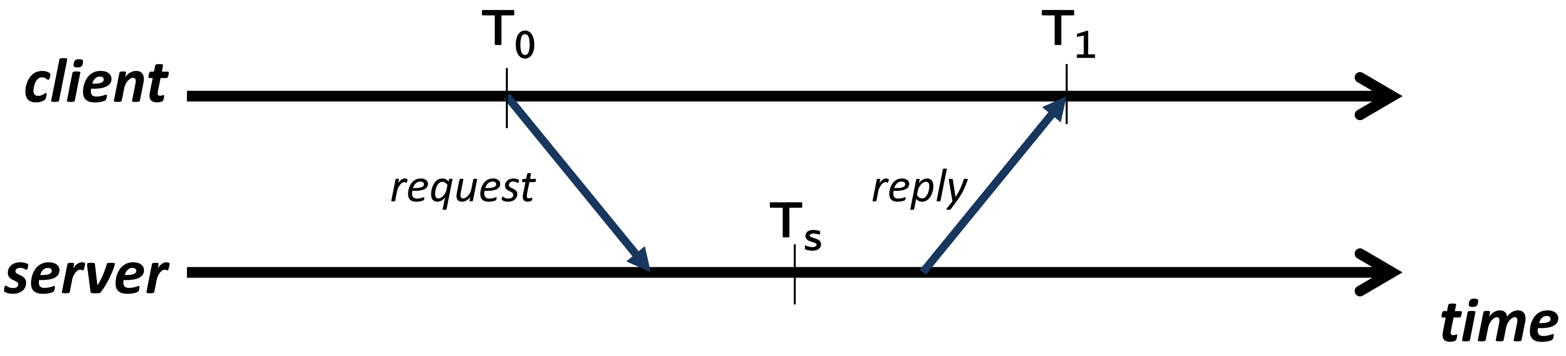
Compensation

- Most systems relate real-time to cycle counters or periodic interrupt sources
  - e.g. calibrate TSC against CMOS RT clock at boot, and compute scaling factor (e.g. cycles per microsecond)
  - can now convert TSC differences to real-time
  - similarly can determine how much real-time passes between periodic interrupts: call this **delta**
  - on interrupt, add delta to software real-time clock
- Making small changes to delta gradually adjusts time
  - Once synchronized, change delta back to original value
  - (or try to estimate drift & continually adjust delta)

Obtaining accurate time

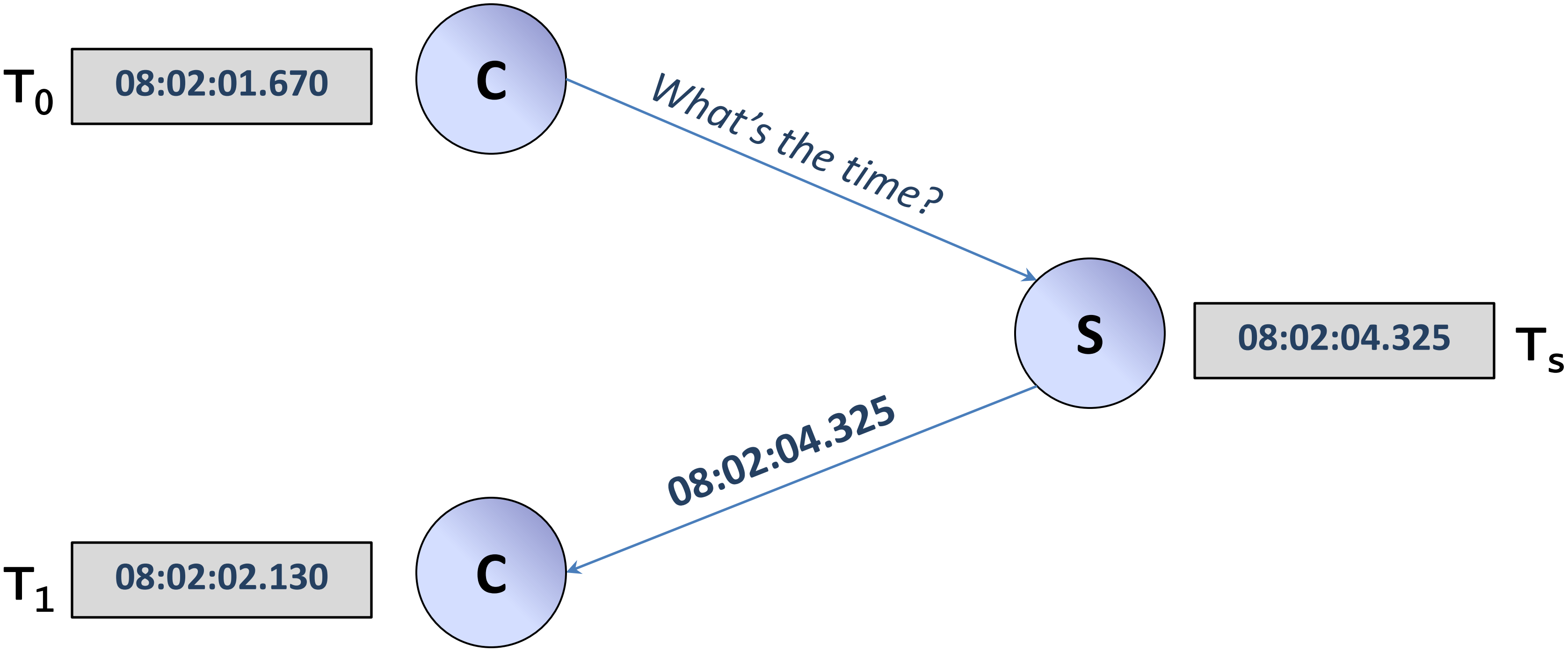
- Of course, need some way to know correct time (e.g. UTC) in order to adjust clock!
  - could attach a GPS receiver (or GOES receiver) to computer, and get ±1ms (or ±0.1ms) accuracy…
  - …but too expensive/clunky for general use
- Instead can ask some machine with a more accurate clock: a **time server**
  - e.g. send RPC getTime() to server
  - What's the problem here?

Cristian’s Algorithm (1989)



- Attempt to compensate for network delays
  - Remember local time just before sending: T_0
  - Server gets request, and puts T_s into response
  - When client receives reply, notes local time: T_1
  - Correct time is then approximately ($T_s + (T_1 - T_0) / 2$)
  - (assumes symmetric behaviour...)

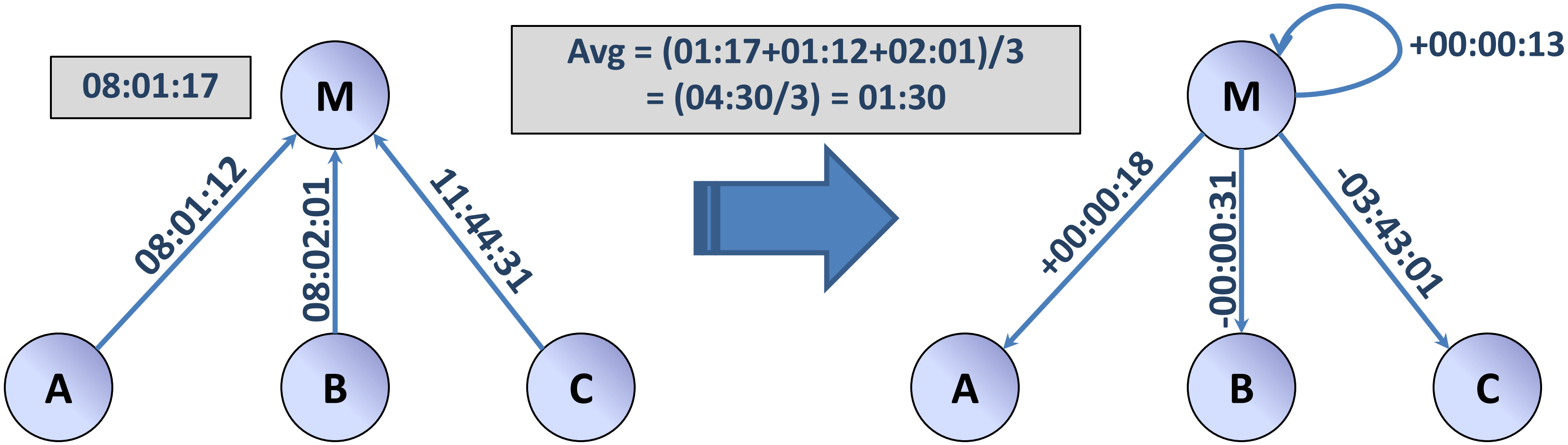
Cristian's Algorithm: Example



- RTT = 460ms, so one way delay is [approx] 230ms.
- Estimate correct time as (08:02:04.325 + 230ms) = 08:02:04.555
- Client gradually adjusts local clock to gain 2.425 seconds

Berkeley Algorithm (1989)

- Don’t assume have an accurate time server
- Try to synchronize a set of clocks to the average
  - One machine, M, is designated the master
  - M periodically polls all other machines for their time
  - (can use Cristian’s technique to account for delays)
  - Master computes average (including itself, but ignoring outliers), and sends an adjustment to each machine

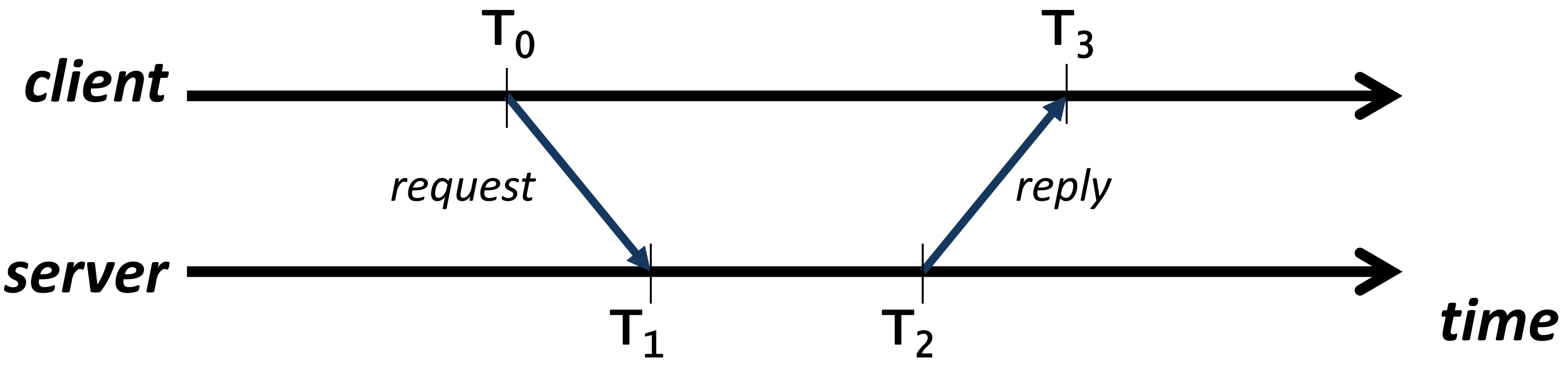


Avg = (01:17+01:12+02:01)/3
= (04:30/3) = 01:30

Network Time Protocol (NTP)

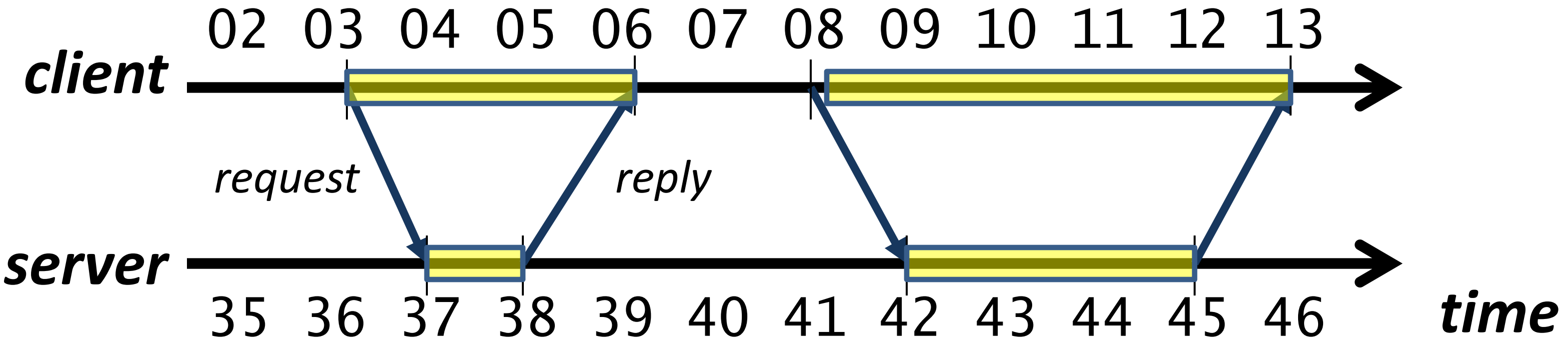
- Previous schemes designed for LANs; in practice today’s systems use NTP:
  - Global service designed to enable clients to stay within (hopefully) a few ms of UTC
- Hierarchy of clocks arranged into strata
  - Stratum0 = atomic clocks (or maybe GPS, GEOS)
  - Stratum1 = servers directly attached to stratum0 clock
  - Stratum2 = servers that synchronize with stratum1
  - … and so on
- Timestamps made up of seconds and ‘fraction’
  - e.g. 32 bit seconds-since-epoch; 32 bit ‘picoseconds’

NTP Algorithm



- UDP/IP messages with slots for four timestamps
  - systems insert timestamps at earliest/latest opportunity
- Client computes:
  - Offset O = $((T_1-T_0) + (T_2-T_3)) / 2$
  - Delay D = $(T_3-T_0) - (T_2-T_1)$
- Relies on symmetric messaging delays to be correct (but now excludes variable processing delay at server)

NTP Example



- First request/reply pair:
  - Total message delay is ((6-3) - (38-37)) = 2
  - Offset is ((37-3) + (38-6)) / 2 = 33
- Second request/reply pair:
  - Total message delay is ((13-8) - (45-42)) = 2
  - Offset is ((42-8) + (45-13)) / 2 = 33

NTP: Additional Details

- NTP uses multiple requests per server
  - Remember <offset, delay> in each case
  - Calculate the **filter dispersion** of the offsets & discard outliers
  - Chooses remaining candidate with the smallest delay
- NTP can also use multiple servers
  - Servers report **synchronization dispersion** = estimate of their quality relative to the root (stratum 0)
  - Combined procedure to select best samples from best servers (see RFC 5905 for the gory details)
- Various operating modes:
  - **Broadcast** (“multicast”): server advertises current time
  - **Client-server** (“procedure call”): as described on previous
  - **Symmetric**: between a set of NTP servers

Physical Clocks: Summary

- Physical devices exhibit **clock drift**
  - Even if initially correct, they tick too fast or too slow, and hence time ends up being wrong
  - Drift rates depend on the specific device, and can vary with time, temperature, acceleration, …
- Difference between clocks is called **clock skew**
- **Clock synchronization algorithms** attempt to minimize the skew between a set of clocks
  - Decide upon a target correct time (atomic, or average)
  - Communicate to agree, compensating for delays
  - In reality, will still have 1-10ms skew after sync ;-(

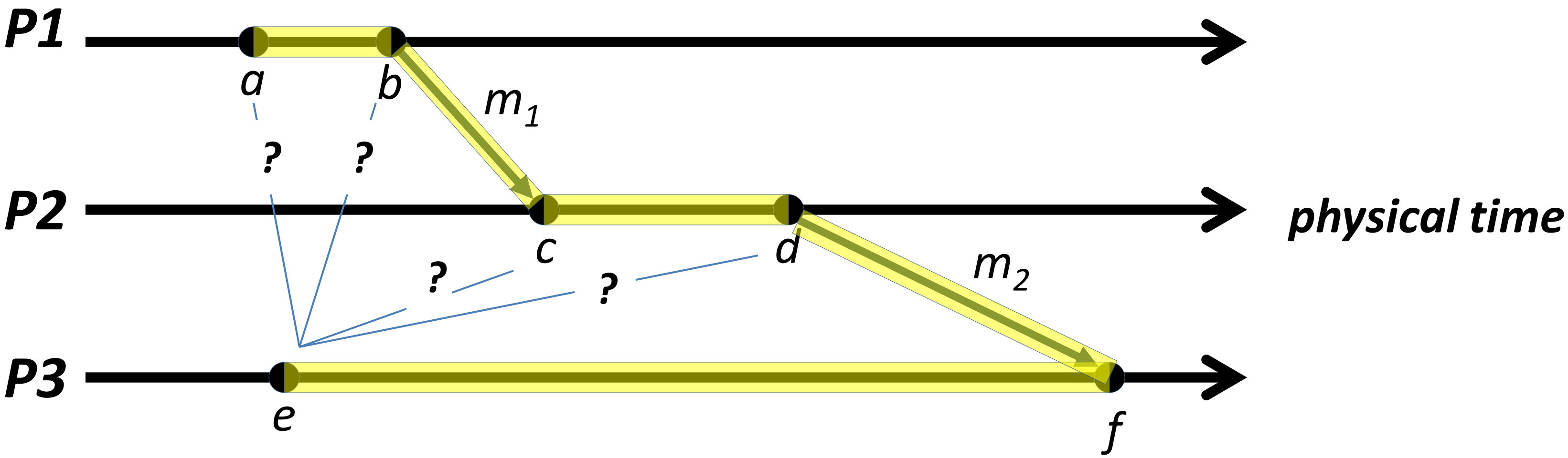
Ordering

- One use of time is to provide ordering
  - If I withdrew £100 cash at 23:59.44…
  - And the bank computes interest at 00:00.00…
  - Then interest calculation shouldn’t include the £100
- But in distributed systems we can’t perfectly synchronize time => cannot use this for ordering
  - Clock skew can be large, and may not be trusted
  - And over large distances, relativistic events mean that ordering depends on the observer
  - (similar effect due to finite ‘speed of Internet’ ;-)

The "happens-before" relation

- Often don't need to know <u>when</u> event *a* occurred
  - Just need to know if *a* occurred before or after *b*
- Define the **happens-before** relation, $\boldsymbol{a \rightarrow b}$
  - If events *a* and *b* are within the same process, then $a \rightarrow b$ if *a* occurs with an earlier local timestamp
  - Messages between processes are ordered ***causally***, i.e. the event *send(m)* → the event *receive(m)*
  - Transitivity: i.e. if $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$
- Note that this only provides a partial order:
  - Possible for neither $a \rightarrow b$ nor $b \rightarrow a$ to hold
  - We say that *a* and *b* are **concurrent** and write $a \sim b$

Example

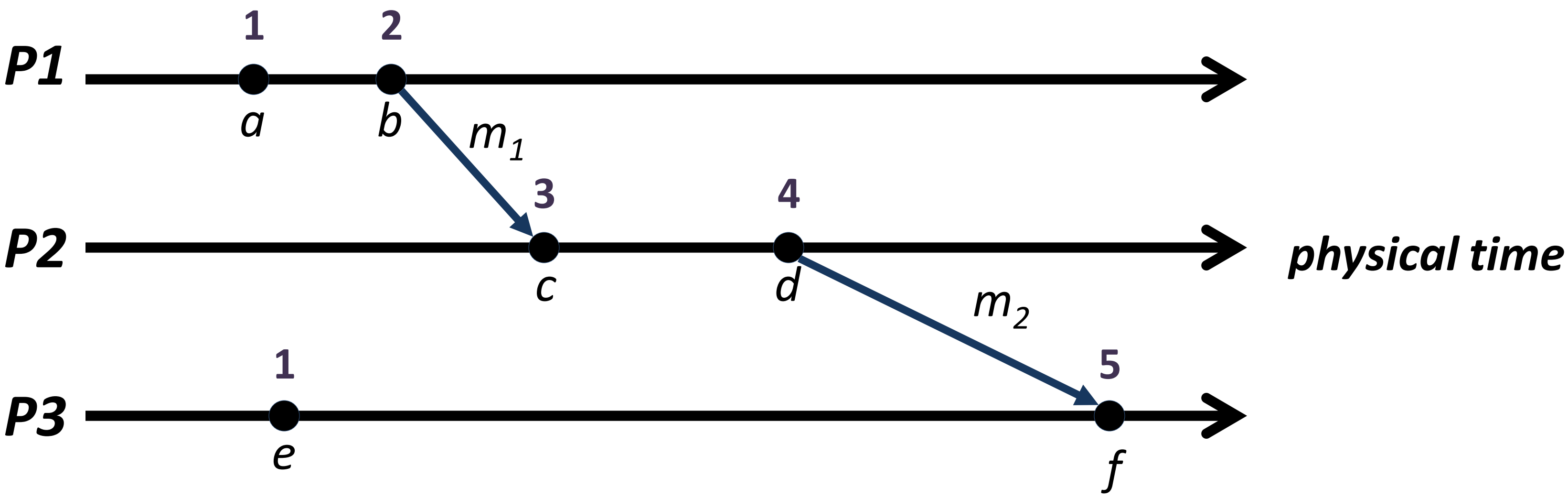


- Three processes (each with 2 events), and 2 messages
  - Due to process order, we know $a \rightarrow b$, $c \rightarrow d$ and $e \rightarrow f$
  - Causal order tells us $b \rightarrow c$ and $d \rightarrow f$
  - And by transitivity $a \rightarrow c$, $a \rightarrow d$, $a \rightarrow f$, $b \rightarrow d$, $b \rightarrow f$, $c \rightarrow f$
- However event *e* is **concurrent** with *a*, *b*, *c* and *d*

Implementing Happens-Before

- One early scheme due to Lamport [1978]
  - Each process P_i has a logical clock L_i
    - L_i can simply be an integer, initialized to 0
  - L_i is incremented on every local event *e*
    - We write $L_i(e)$ or $L(e)$ as the timestamp of *e*
  - When P_i sends a message, it increments L_i and copies the value into the packet
  - When P_i receives a message from P_j, it extracts L_j and sets L_i := **max**(L_i,L_j), and then increments L_i
- Guarantees that if $a \rightarrow b$, then $L(a) < L(b)$
  - However if $L(x) < L(y)$, this doesn't imply $x \rightarrow y$!

Lamport Clocks: Example

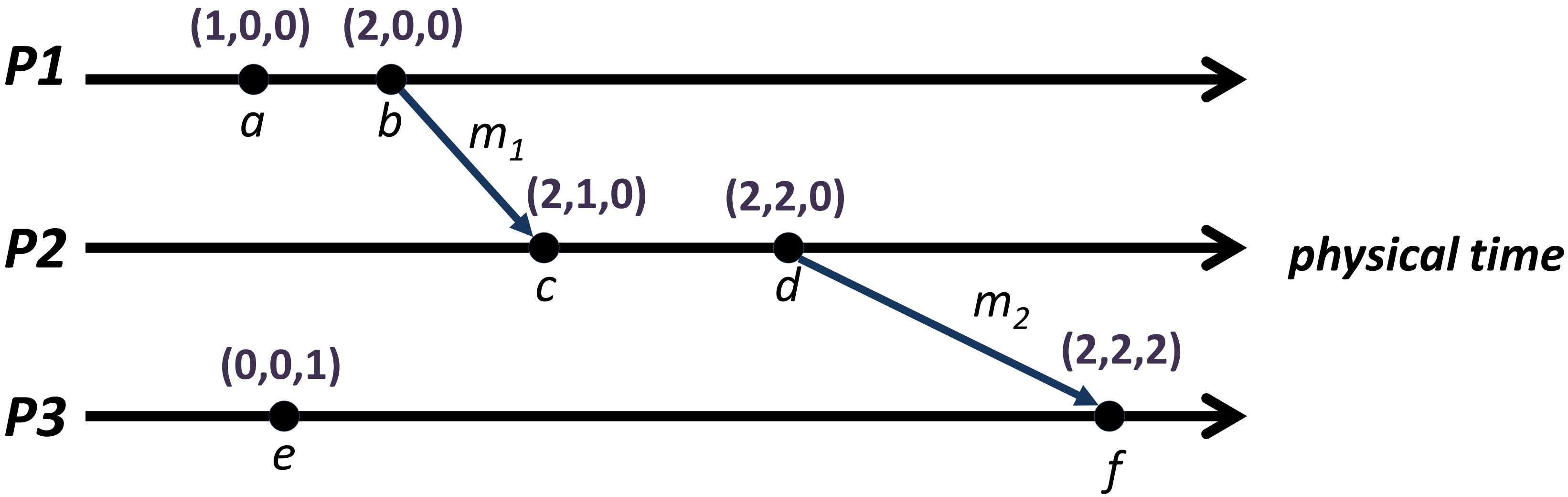


- When P_2 receives m_1, it extracts timestamp 2 and sets its clock to max(0, 2) before increment
- Possible for events to have duplicate timestamps
  - e.g. event *e* has the same timestamp as event *a*
- If desired can break ties by looking at pids, IP addresses, …
  - this gives a **total order**, but doesn't imply happens-before!

Vector Clocks

- With Lamport clocks, given L(*a*) and L(*b*), we can't tell if $a \rightarrow b$ or $b \rightarrow a$ or $a \sim b$
- One solution is **vector clocks**:
  - An ordered list of logical clocks, one per-process
  - Each process P_i maintains $V_i[]$, initially all zeroes
  - On a local event *e*, P_i increments $V_i[i]$
    - If the event is message send, new $V_i[]$ copied into packet
  - If P_i receives a message from P_j then, for all k = 0, 1, …, it sets $V_i[k] := \max(V_j[k], V_i[k])$, and increments $V_i[i]$
- Intuitively $V_i[k]$ captures the number of events at process P_k that have been observed by P_i

Vector Clocks: Example



- When P_2 receives m_1, it **merges** the entries from P_1's clock
  - choose the maximum value in each position
- Similarly when P_3 receives m_2, it merges in P_2's clock
  - this incorporates the changes from P_1 that P_2 already saw
- Vector clocks ***explicitly track the transitive causal order***: *f*'s timestamp captures the history of *a*, *b*, *c* & *d*

Using Vector Clocks for Ordering

- Can compare vector clocks piecewise:
  - $V_i = V_j$ iff $V_i[k] = V_j[k]$ for $k = 0, 1, 2, \ldots$
  - $V_i \leq V_j$ iff $V_i[k] \leq V_j[k]$ for $k = 0, 1, 2, \ldots$
  - $V_i < V_j$ iff $V_i \leq V_j$ and $V_i \neq V_j$
  - $V_i \sim V_j$ otherwise

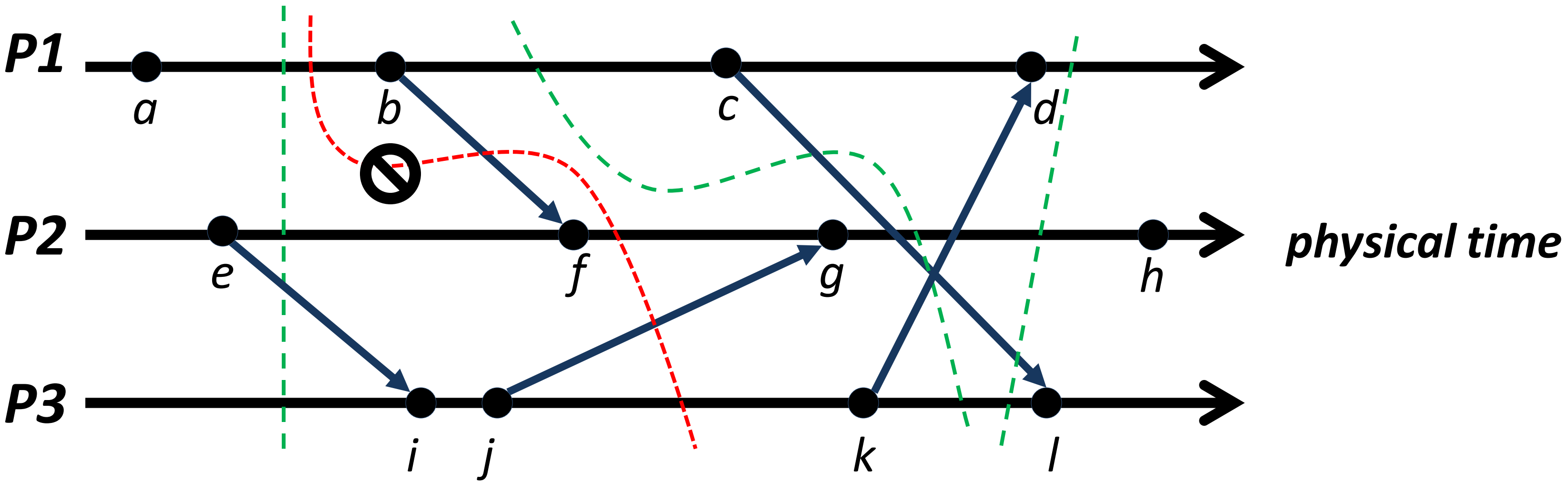
e.g. [2,0,0] versus [0,0,1]

- For any two event timestamps T(a) and T(b)
  - if $a \rightarrow b$ then $T(a) < T(b)$; **and**
  - if $T(a) < T(b)$ then $a \rightarrow b$
- Hence can use timestamps to determine if there is a causal ordering between any two events
  - i.e. determine whether $a \rightarrow b$, $b \rightarrow a$ or $a \sim b$

Consistent Global State

- We have the notion of "*a* happens-before *b*" ($a \rightarrow b$) or "*a* is concurrent with *b*" ($a \sim b$)
- What about 'instantaneous' system-wide state?
  - distributed debugging, GC, deadlock detection, ...
- Chandy/Lamport introduced **consistent cuts**:
  - draw a (possibly wiggly) line across all processes
  - this is a consistent cut if the set of events (on the lhs) is closed under the happens-before relationship
  - i.e. if the cut includes event *x*, then it also includes all events *e* which happened before *x*
- In practical terms, this means every *delivered* message included in the cut was also *sent* within the cut

Consistent Cuts: Example



- Vertical cuts are always consistent (due to the way we draw these diagrams), but some curves are ok too:
  - providing we don’t include any receive events without their corresponding send events
- Intuition is that a consistent cut *could* have occurred during execution (depending on scheduling etc),

<< Observing Consistent Cuts >>

- Chandy/Lamport Snapshot Algorithm (1985):
  - Distributed algorithm for generating a ‘snapshot’ of relevant system-wide state (e.g. all memory, locks held, …)
  - Based on flooding special marker message M to all processes; causal order of flood defines the cut
  - If P_i receives M from P_j and it has yet to snapshot:
    - It pauses all communication, takes local snapshot & sets C_{ij} to {}
    - Then sends M to all other processes P_k and starts recording C_{ik} = { *set of all post local snapshot messages received from* P_k }
  - If P_i receives M from some P_k *after* taking snapshot
    - Stops recording C_{ik}, and saves alongside local snapshot
  - Global snapshot comprises all local snapshots & C_{ij}
  - Assumes reliable, in-order messages, & no failures

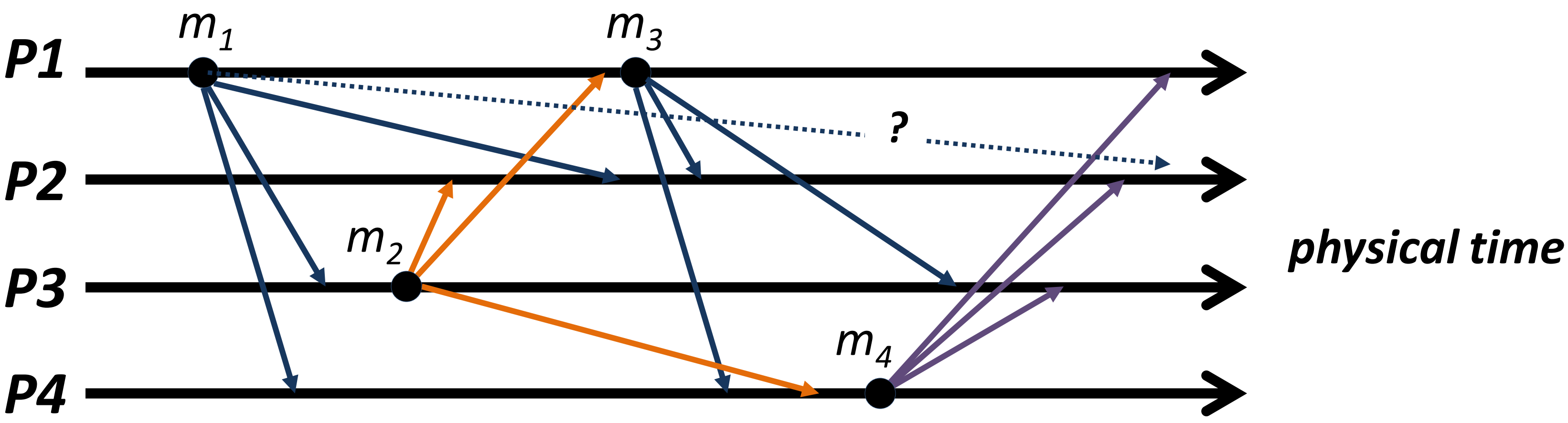
Process Groups

- Often useful to build distributed systems around the notion of a **process group**
  - Set of processes on some number of machines
  - Possible to **multicast** messages to all members
  - Allows fault-tolerant systems even if some processes fail
- Membership can be **fixed** or **dynamic**
  - if dynamic, have explicit join() and leave() primitives
- Groups can be **open** or **closed**:
  - Closed groups only allow messages from members
- Internally can be structured (e.g. coordinator and set of slaves), or symmetric (peer-to-peer)
  - Coordinator makes e.g. concurrent join/leave easier…
  - … but may require extra work to **elect** coordinator

Group Communication: Assumptions

- Assume we have ability to send a message to multiple (or all) members of a group
  - Don’t care if ‘true’ multicast (single packet sent, received by multiple recipients) or “netcast” (send set of messages, one to each recipient)
- Assume also that message delivery is reliable, and that messages arrive in bounded time
  - But may take different amounts of time to reach different recipients
- Assume (for now) that processes don’t crash
- What delivery *orderings* can we enforce?

FIFO Ordering



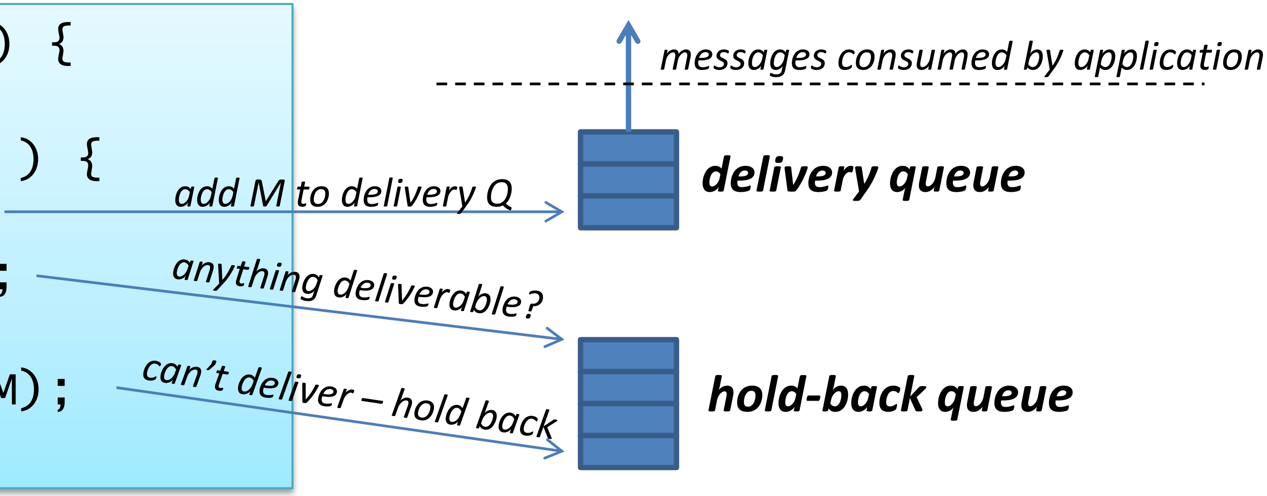
- With **FIFO ordering**, messages from a particular process P_i must be received at all other processes P_j in the order they were sent
  - e.g. in the above, everyone must see m_1 before m_3
  - (ordering of m_2 and m_4 is not constrained)
- Seems easy but not trivial in case of delays / retransmissions
  - e.g. what if message m_1 to P2 takes a loooong time?
- Hence receivers may need to **buffer** messages to ensure order

Receiving versus Delivering

- Group communication middleware provides extra features above ‘basic’ communication
  - e.g. providing reliability and/or ordering guarantees on top of IP multicast or netcast
- Assume that OS provides receive() primitive:
  - returns with a packet when one arrives on wire
- Received messages either **delivered** or **held back**:
  - “delivered” means inserted into delivery queue
  - “held back” means inserted into hold-back queue
  - held-back messages are delivered later as the result of the receipt of another message…

Implementing FIFO Ordering

```
receive(M from Pi) {
 s = SeqNo(M);
 if (s == (Sji+1) ) {
   deliver(M);
   s = flush(hbq);
   Sji = s;
 } else holdback(M);
}
```

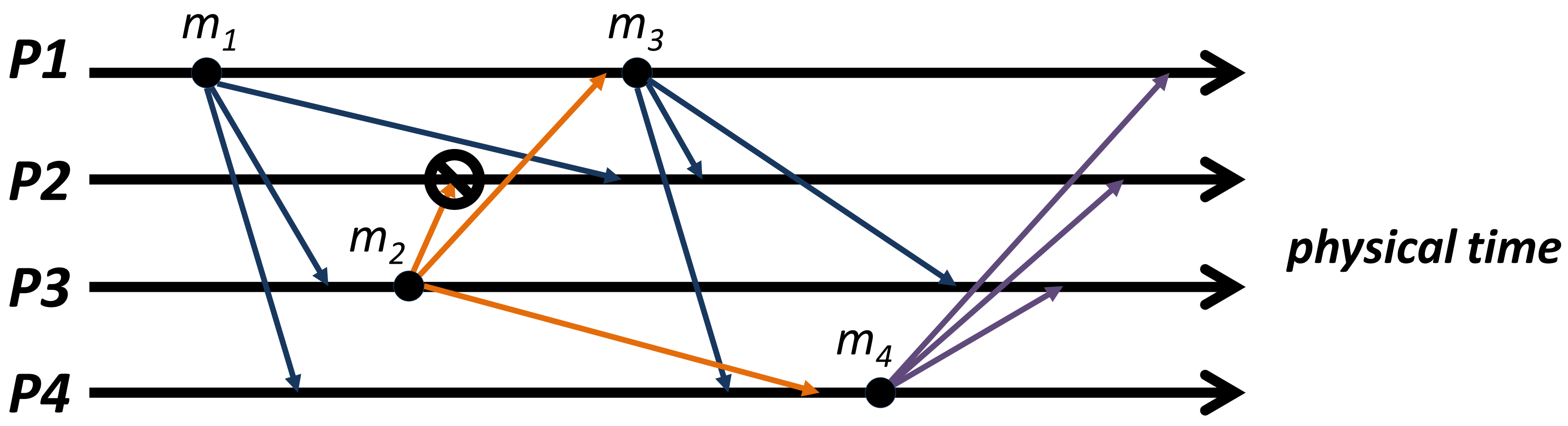


- Each process P_i maintains a message sequence number (SeqNo) S_i
- Every message sent by P_i includes S_i, incremented after each send
  - not including retransmissions!
- P_j maintains S_{ji} : the SeqNo of the last ***delivered*** message from P_i
  - If receive message from P_i with SeqNo ≠ (S_{ji}+1), hold back
  - When receive message with SeqNo = (S_{ji}+1), deliver it … and also deliver any consecutive messages in hold back queue … and update S_{ji}

Stronger Orderings

- Can also implement FIFO ordering by just using a reliable FIFO transport like TCP/IP ;-)
- But the general 'receive versus deliver' model also allows us to provide **stronger** orderings:
  - **Causal ordering**: if event $multicast(g, m_1) \rightarrow multicast(g, m_2)$, then all processes will see m_1 before m_2
  - **Total ordering**: if any processes delivers a message m_1 before m_2, then all processes will deliver m_1 before m_2
- Causal ordering implies FIFO ordering, since any two multicasts by the same process are related by →
- Total ordering (as defined) does *not* imply FIFO (or causal) ordering, just says that all processes must agree
  - In reality often want **FIFO-total** ordering (combines the two)

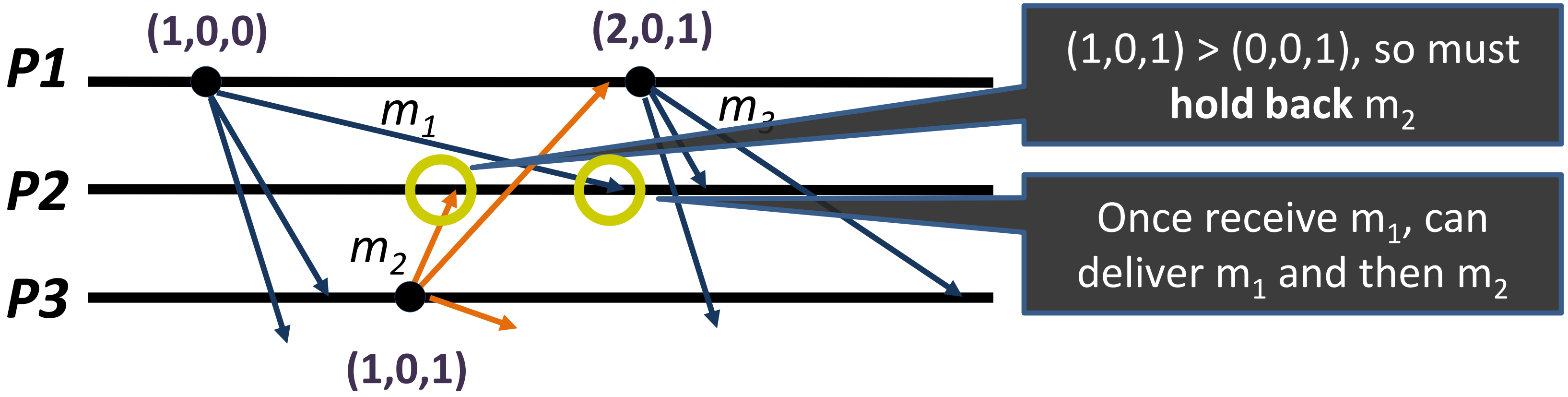
Causal Ordering



- Same example as previously, but now causal ordering means that
  (*a*) everyone must see m$_1$ before m$_3$ (as with FIFO), **and**
  (*b*) everyone must see m$_1$ before m$_2$ (due to happens-before)
- Is this ok?
  - No! m$_1$ $\rightarrow$ m$_2$, but P2 sees m$_2$ before m$_1$
  - To be correct, must hold back (delay) delivery of m$_2$ at P2
  - But how do we know this?

Implementing Causal Ordering

- Turns out this is pretty easy!
  - Start with receive algorithm for FIFO multicast…
  - and replace sequence numbers with vector clocks



(1,0,1) > (0,0,1), so must **hold back** m_2

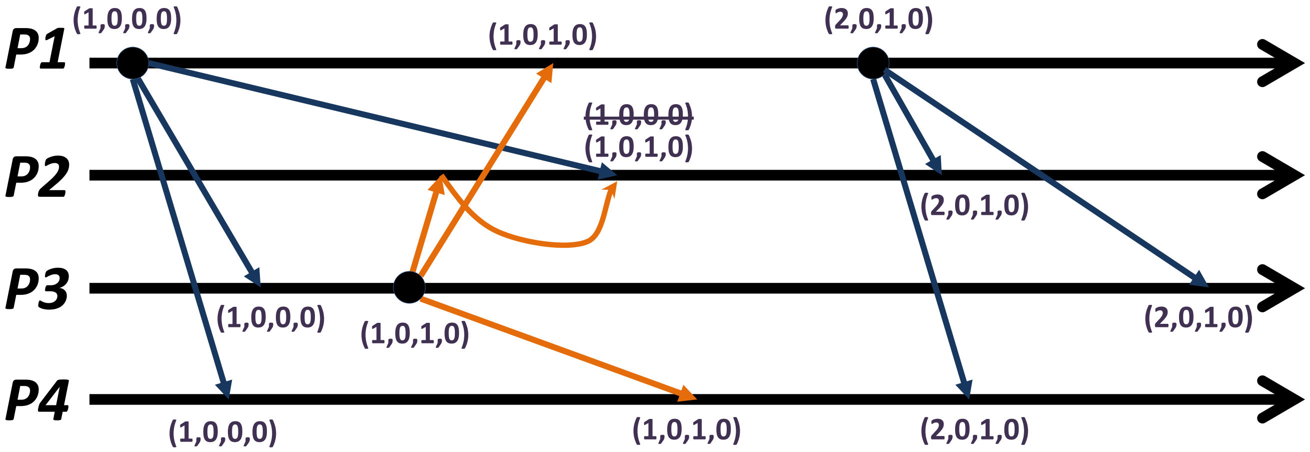
Once receive m_1, can deliver m_1 and then m_2

- Need some care with dynamic groups
  - must encode variable-length vector clock, typically using positional notation, and deal with joins and leaves

In more detail

- Each process P_i has vector V_i[] to ensure causal order
  - don’t use this vector to track *other* process-internal events
- To send message m, P_i first increments its local vector V_i[i], and copies the result into message as a timestamp
- On receipt of message m from P_j we only deliver if
  - **$V_j[j] = V_i[j] + 1$** (i.e. m is the *next* message from P_j) **and**
  - **$V_j[k] <= V_i[k]$** for all **$k \neq j$** (i.e. P_i has seen at least as many other messages as P_j)
  - If these conditions **do not** hold, m must be held back
  - Otherwise we increment V_i[j] and deliver the message... and check if we can now deliver any held-back messages
- Note that we do not increment V_i[i] on receive

Example:



- P1 increments first element, and sends message w/ timestamp [1,0,0,0]
- P3 and P4 receive it and compare local (0,0,0,0) to [1,0,0,0]
  - ok, so both set their local vectors to (1,0,0,0)
- P3 increments third element, and sends message w/ timestamp [1,0,1,0]
  - P1, P4 compare (1,0,0,0) to [1,0,1,0] => ok, so both update to (1,0,1,0)
  - P2 receives and compares (0,0,0,0) to [1,0,1,0] – **cannot deliver!**
- P2 receives P1's message and compares (0,0,0,0) to [1,0,0,0] – **ok**
- After delivery, P2 checks held-back queue, and now can deliver P3's message

Total Ordering

- Sometimes we want all processes to see exactly the same, FIFO, sequence of messages
  - particularly for state machine replication (see later)
- One way is to have a **‘can send’ token**:
  - Token passed round-robin between processes
  - Only process with token can send (if he wants)
- Or use a **dedicated sequencer process**
  - Other processes ask for **global sequence no**. (GSN), and then send with this in packet
  - Use FIFO ordering algorithm, but on GSNs
- Can also build *non-FIFO* total order multicast by having processes generate GSNs themselves and resolving ties

Ordering and Asynchrony

- FIFO ordering allows quite a lot of **asynchrony**
  - e.g. any process can delay sending a message until it has a batch (to improve performance)
  - or can just tolerate variable and/or long delays
- Causal ordering also allows some asynchrony
  - But must be careful queues don't grow too large!
- Traditional total order multicast not so good:
  - Since every message delivery transitively depends on every other one, delays holds up the entire system
  - Instead tend to an (almost) synchronous model, but this performs poorly, particularly over the wide area ;-)
  - Some clever work on **virtual synchrony** (for the interested)