Distributed systems

Lecture 12: Logical time, vector clocks, process groups, and ordered broadcast

Michaelmas 2019

Dr Martin Kleppmann

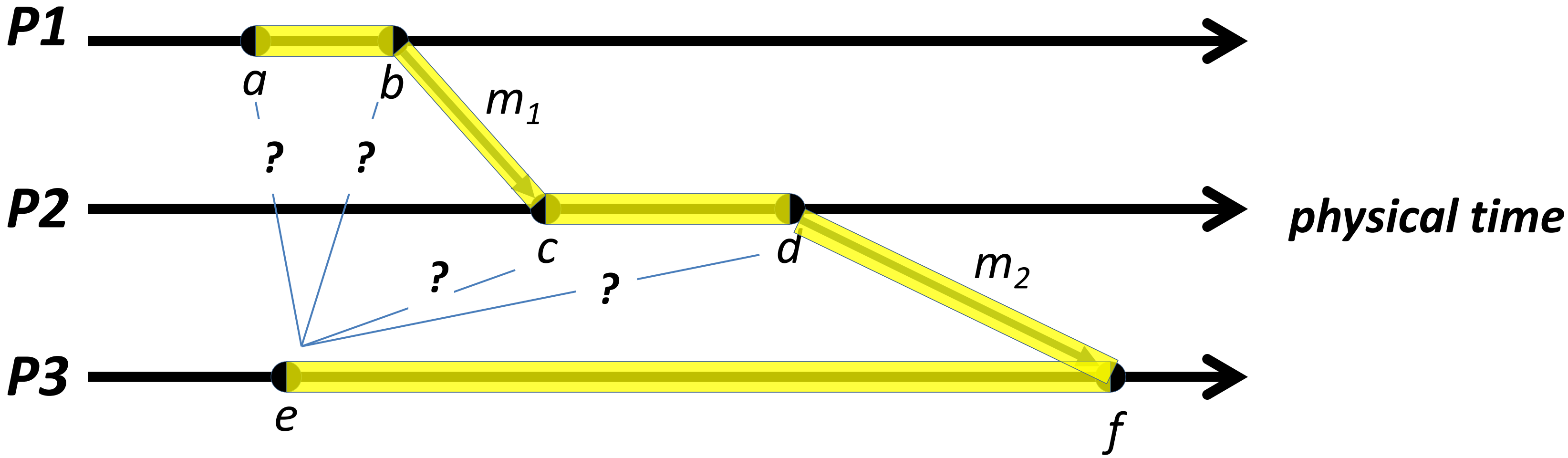
(With thanks to Dr Robert N. M. Watson
and Dr Steven Hand)

Last time

- Clock skew and drift
- The clock synchronization problem
- Cristian's Algorithm, Berkeley Algorithm, NTP
- The happens-before relation

- Saw physical time can't be kept exactly in sync; instead, use **logical clocks** to track ordering between events:
  - Defined ***a*** $\rightarrow$ ***b*** to mean '***a* happens-before *b***'
  - Easy inside single process, & use causal ordering (*send* $\rightarrow$ *receive*) to extend relation across processes

Example



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

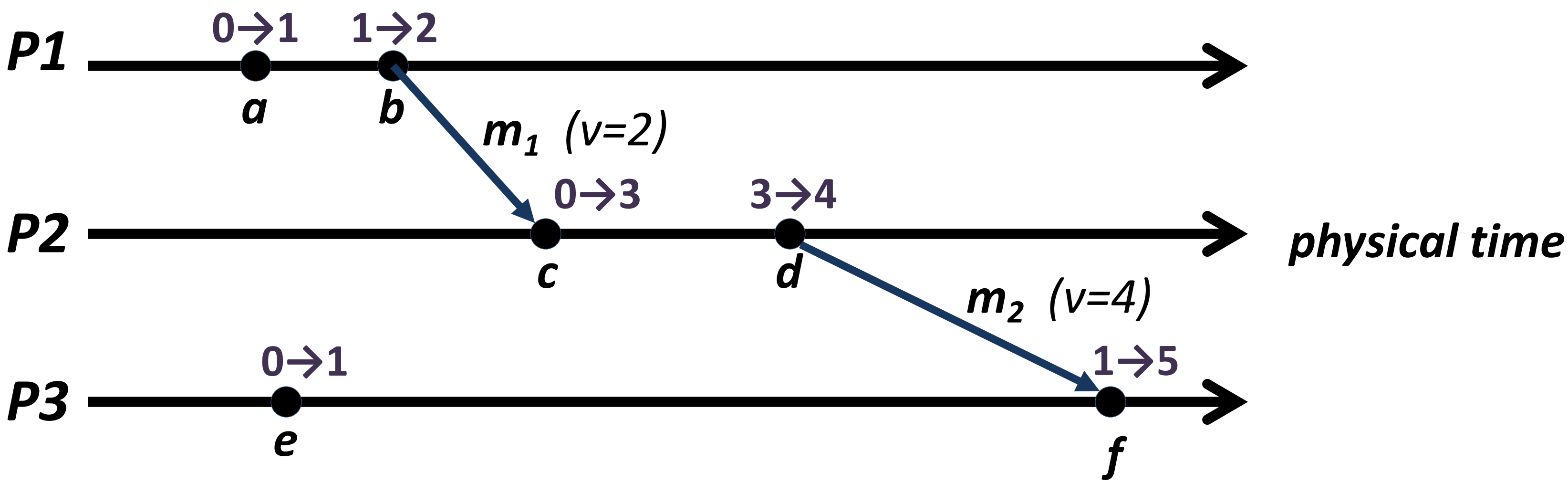
Causal ordering

- NB. "**causal**" ≠ "**casual**"!
  - As in "cause and effect"
- e.g. P1 sends message m, P2 receives m
  - receipt of m is caused by sending of m
  - so sending event **causally precedes** receipt event
- e.g. Alice asks a question, Bob answers it
  - observer would be confused if they hear the answer before hearing the question
- Causal order is any order that is compatible with happens-before relation

Logical time

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

Lamport Clocks: Example

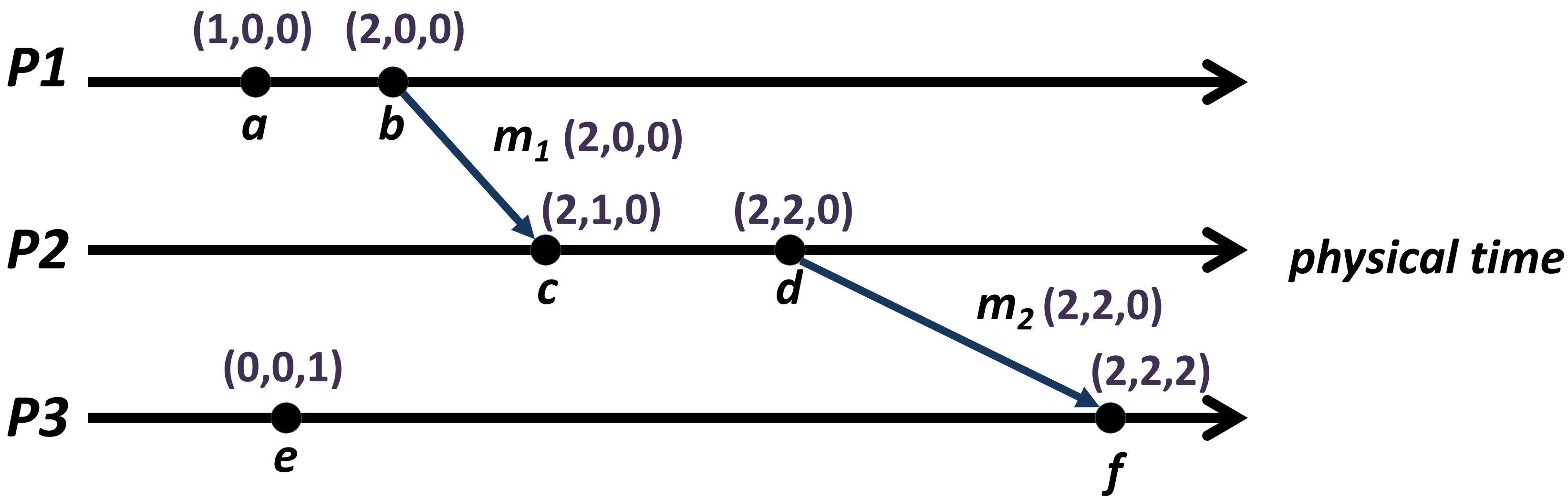


- When $\mathbf{P_2}$ receives $\mathbf{m_1}$, it extracts timestamp 2 and sets its clock to **max(0, 2)** before increment
- Event timestamps are not unique
  - E.g., event ***e*** has the same timestamp as event ***a***
- Break ties by looking at process IDs, IP addresses, …
  - This gives a **total order** and **globally unique timestamps** (assuming process IDs are globally unique)
  - Concurrent events are ordered arbitrarily

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* ‖ *b***
- One solution is **vector clocks**:
  - An **ordered list of logical clocks**, one per-process
  - Each process $\mathbf{P_i}$ maintains $\mathbf{V_i[\,]}$, initially all zeroes
  - On a local event ***e***, $\mathbf{P_i}$ increments $\mathbf{V_i[i]}$
    - If the event is message send, new $\mathbf{V_i[\,]}$ copied into packet
  - If $\mathbf{P_i}$ receives a message from $\mathbf{P_j}$ then, for all **k** = 0, 1, …, it sets $\mathbf{V_i[k] := max(V_j[k], V_i[k])}$, and increments $\mathbf{V_i[i]}$
- Intuitively $\mathbf{V_i[k]}$ captures the number of events at process $\mathbf{P_k}$ that have been observed by $\mathbf{P_i}$

Vector clocks: example



- When **P2** receives $\mathbf{m_1}$, it **merges** entries from **P1**'s clock
  - choose the maximum value in each position
- Similarly when **P3** receives $\mathbf{m_2}$, it merges in **P2**'s clock
  - this incorporates the changes from **P1** that **P2** already saw
- Vector clocks ***explicitly track transitive causal order***: timestamp of ***f*** captures the history of ***a***, ***b***, ***c*** & ***d***

Using vector clocks for ordering

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

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

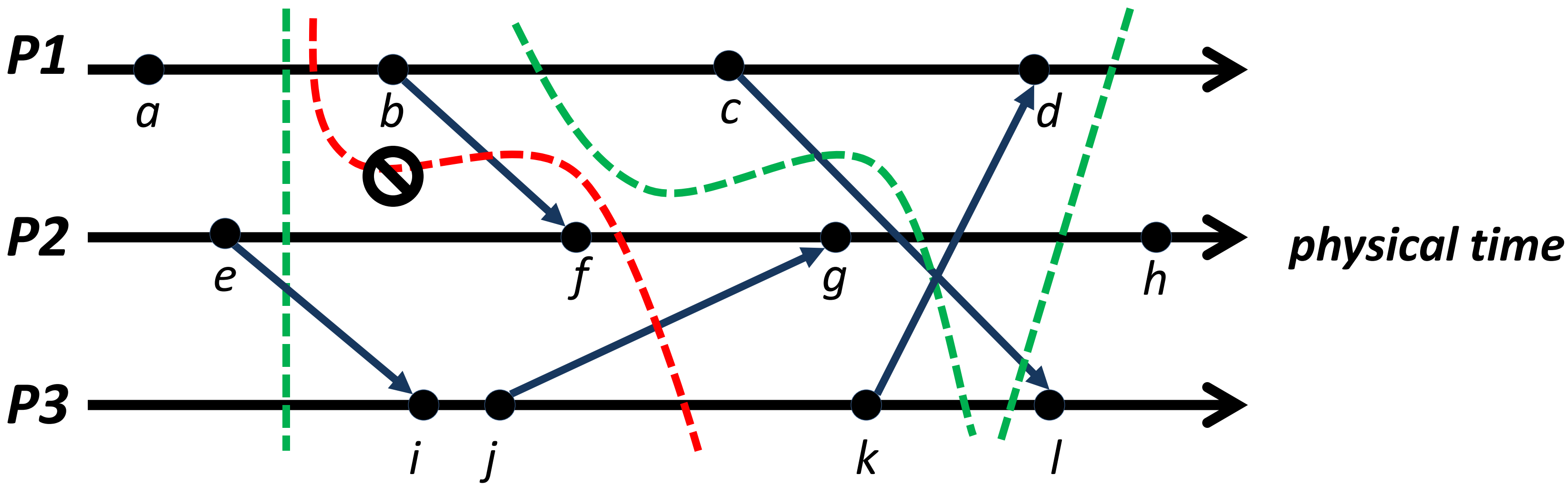
- For any two event timestamps **T(a)** and **T(b)**
  - if $a \rightarrow b$ then **T(*a*) < T(*b*)** ; **and**
  - if **T(*a*) < T(*b*)** then $a \rightarrow b$
- Hence can use timestamps to determine if there is a **causal ordering** between any two events
  - i.e. determine whether $a \rightarrow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

Does this seem familiar? Recall **Time-Stamp Ordering** and **Optimistic Concurrency Control** for transactions

Consistent global state

- We have the notion of "***a*** happens-before ***b***" ($a \rightarrow b$) or "***a*** is concurrent with ***b***" ($a \parallel b$)
- 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 – **sketch**

We will skip this material in lecture and it is not examinable – but it is helpful in thinking about distributed algorithms:

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

Process groups

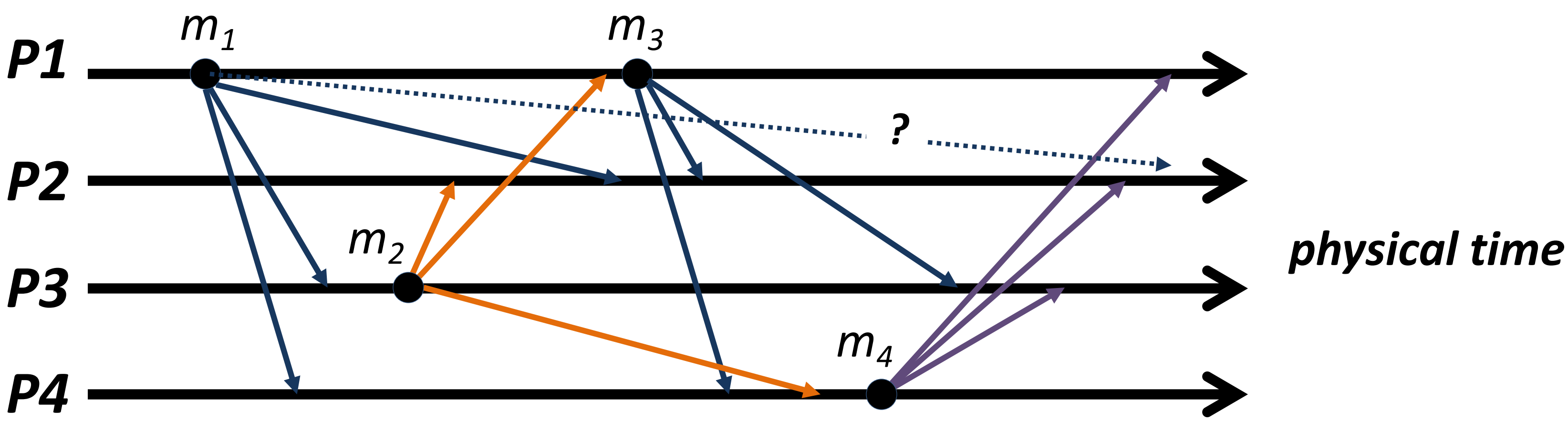
- **Process groups** are a key distributed-systems primitive:
  - Set of processes on some number of machines
  - Possible to **multicast** messages to all members
  - Allows fault-tolerant systems even if some processes fail
- Membership can be **fixed** or **dynamic**
  - If dynamic, have explicit `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

When we use "**multicast**" in distributed systems, we mean something stronger than conventional network datagram multicasting – do not confuse them

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



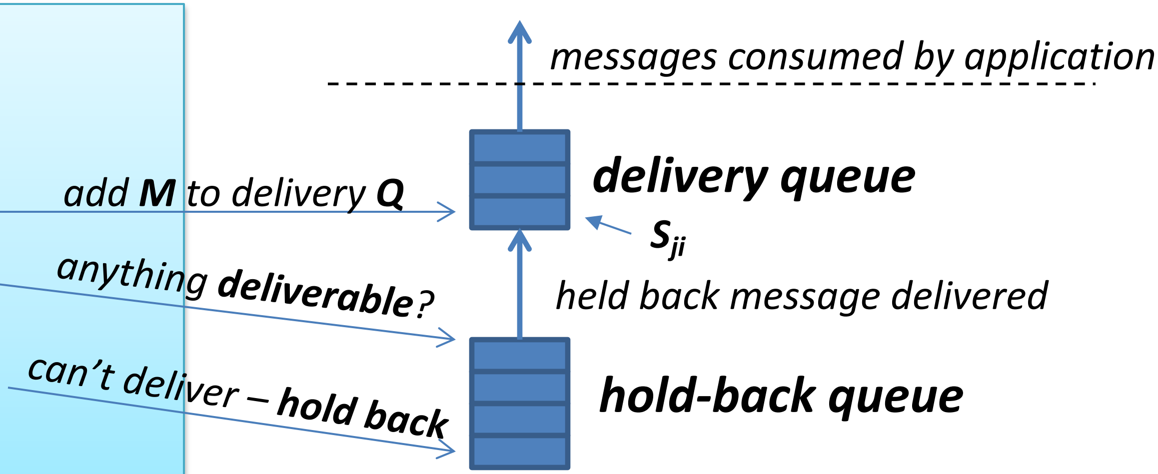
- With **FIFO ordering**, messages from process $\mathbf{P_i}$ must be received at each process $\mathbf{P_j}$ in the order they were sent
  - E.g. in the above, each receiver must see $\mathbf{m_1}$ before it sees $\mathbf{m_3}$
  - But other relative delivery orders are unconstrained – e.g., $\mathbf{m_1}$ vs $\mathbf{m_2}$, $\mathbf{m_2}$ vs. $\mathbf{m_4}$, etc.
- Looks easy, but is non-trivial on delays/retransmissions
  - E.g. what if message $\mathbf{m_1}$ to **P2** takes a loooong time?
- Receivers may need to **buffer** messages to ensure order
  - Must "hold back" $\mathbf{m_3}$ until $\mathbf{m_1}$ has been delivered to **P2**

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);
}
```

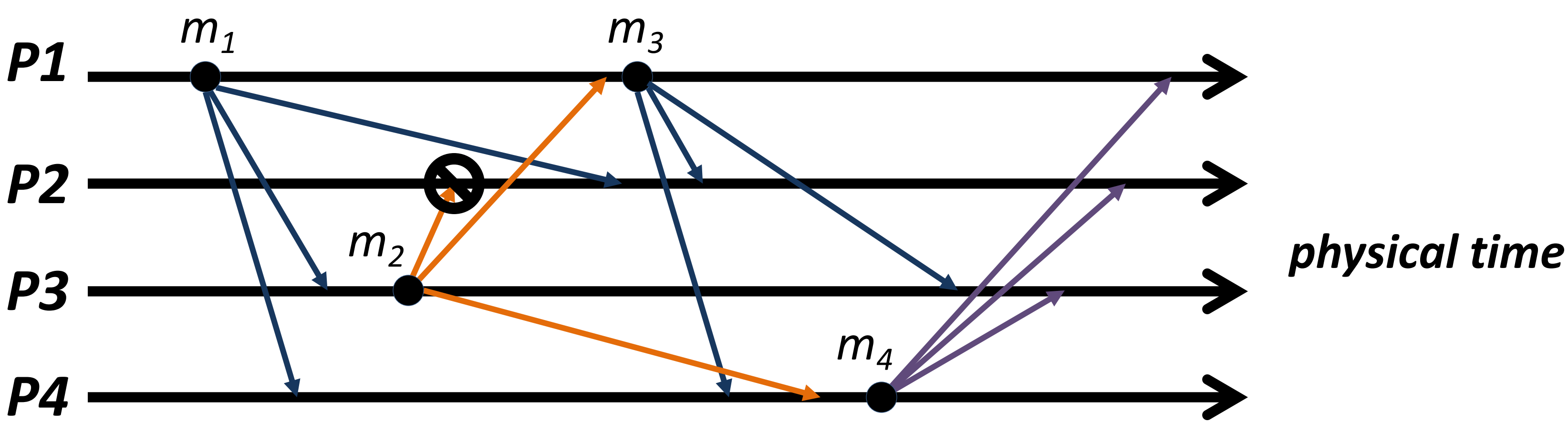


- Each process P_i maintains sequence number (SeqNo) S_i
- New messages sent by P_i include S_i, incremented after each send
  - Not including retransmissions, which retransmit with the same SeqNo!
- P_j maintains S_{ji}: the SeqNo of the last ***delivered*** message from P_i
  - If receive message from P_i with SeqNo ≠ (S_{ji}+1), **hold back**
  - When receive message with SeqNo = (S_{ji}+1), **enqueue for delivery**
  - Also **deliver consecutive messages** in hold-back queue (if present)
  - **Update S_{ji}**
- Apps. **receive** asynchronously as they read from delivery queue

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 *send(**g**, **m_1**)* $\rightarrow$ *send(**g**, **m_2**)*, then all processes will see $\mathbf{m_1}$ before $\mathbf{m_2}$
  - **Total ordering**: if any process delivers a message $\mathbf{m_1}$ before $\mathbf{m_2}$, then all processes will deliver $\mathbf{m_1}$ before $\mathbf{m_2}$
- Causal ordering implies FIFO ordering, since any two multicasts by the same process are related by $\rightarrow$
- Total ordering (as defined) does *not* imply FIFO (or causal) ordering, just says that all processes must agree
  - Sometimes want **FIFO-total** ordering (combines the two)

Causal ordering



- e.g. order of messages in chat app (question→answer)
- Same example as before, but causal ordering requires:
  (*a*) everyone must see $\mathbf{m_1}$ before $\mathbf{m_3}$ (as with FIFO), **and**
  (*b*) everyone must see $\mathbf{m_1}$ before $\mathbf{m_2}$ (due to happens-before)
- Is this ok?
  – No! $\mathbf{m_1} \rightarrow \mathbf{m_2}$, but **P2** sees $\mathbf{m_2}$ before $\mathbf{m_1}$
  – To be correct, must hold back (delay) delivery of $\mathbf{m_2}$ at **P2**

Causal order and happens-before

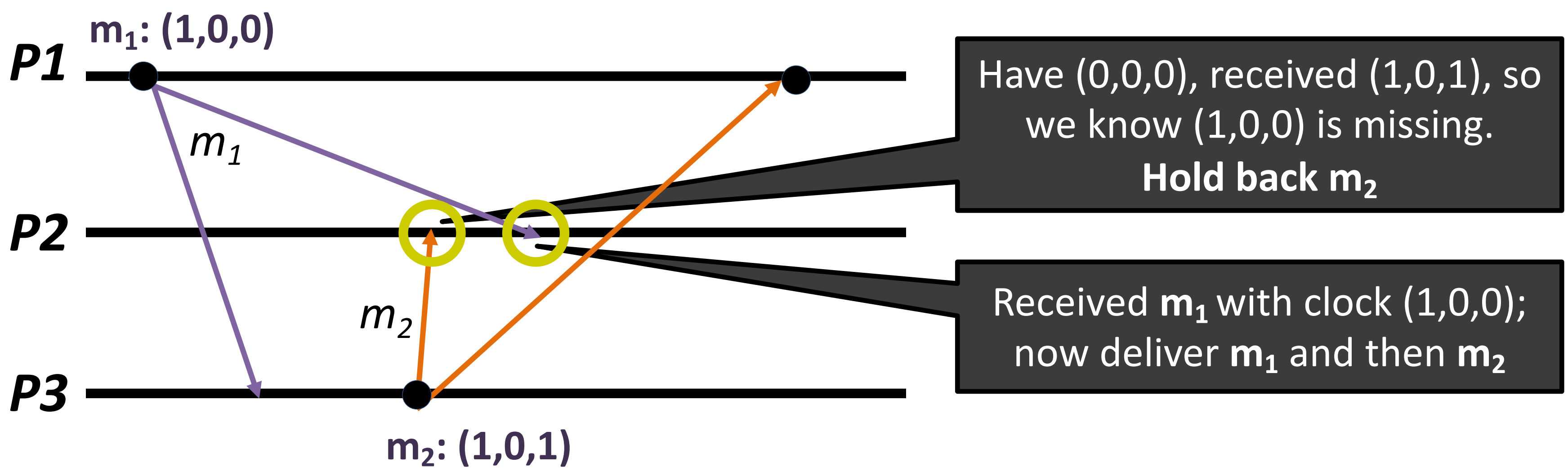
- Happens-before is a **strict partial order**
  - Irreflexive, transitive, asymmetric
- Any **linear extension** of happens-before is a causal order
  - The order is ***consistent with causality***
- For a given partial order, there may be many possible linear extensions
  - Concurrent events can be ordered arbitrarily

Causal order message delivery

- When message ***m*** is received, need to decide:
  - Does a message ***m'*** exist that we have not yet received, such that ***m'*** $\rightarrow$ ***m***?
  - If yes, wait for ***m'*** to be received and deliver it first
  - If no, deliver ***m*** to the application now
- Solution: variant of vector clocks
  - Increment only on ***message send***, not on every event
  - Detects relative ordering of ***messages***, not events
  - Gap in number sequence $\Rightarrow$ wait for message

Implementing causal ordering

- Like FIFO multicast, but with vector clocks instead of sequence numbers



- Some care needed with dynamic groups

Total ordering

- Sometimes we want all processes to see exactly the same sequence of messages, in the same order
  - particularly for **state machine replication** (see later)
- One option: 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
  - Problem: what if sequencer crashes/is unreachable?
- Another option: order by **Lamport timestamp**
  - Problem: how do you know if you have seen all messages with timestamp $< T$?
  - Need to wait for ≥ 1 message with timestamp $\geq T$ from **every other process**

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!
- Performance of total-order multicast not so good:
  - Since every message delivery transitively depends on every prior one, delays holds up the entire system
  - Instead tend to an (almost) synchronous model, but this performs poorly, particularly over the wide area
  - Insight: total order multicast is **equivalent to consensus** [Chandra and Toueg 1996]

Summary + next time

- Vector clocks
- Consistent global state + consistent cuts
- Process groups and reliable multicast
- Implementing order

- Distributed mutual exclusion
- Leader elections and distributed consensus
- Distributed transactions and commit protocols
- Replication and consistency