Last time

- 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
  - if $send_i(m_1) \rightarrow send_i(m_2)$ then $deliver_k(m_1) \rightarrow deliver_k(m_2)$

- Lamport clocks, $L(e)$: an integer
  - Increment to (max of (sender, receiver)) + 1 on receipt
  - But given $L(a) < L(b)$, order of $a$ and $b$ is unknown

- The obvious question arises: How can we extend logical time to work “in the other direction”?
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 $V_i[i]$ copied into packet
  – If $P_i$ receives a message from $P_j$ then, for all $k = 0, 1, \ldots,$
    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 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 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

• 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 – 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 $P_i$ 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

- **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 $P_1$ must be received at each process $P_j$ in the order they were sent
  - E.g. in the above, each receiver must see $m_1$ before it sees $m_3$.
  - But other relative delivery orders are unconstrained – e.g., $m_1$ vs $m_2$, $m_2$ vs. $m_4$, etc.
- Looks easy, but is non-trivial on delays/retransmissions
  - E.g. what if message $m_1$ to $P_2$ takes a loooong time?
- Receivers may need to **buffer** messages to ensure order
  - Must “hold back” $m_3$ until $m_1$ has been delivered to $P_2$.

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

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 \( \neq (S_{ji}+1) \), 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

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

messages consumed by application

delivery queue

\( S_j \)

hold-back queue

held back message delivered

anything deliverable?

can't deliver — hold back

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 \( \text{multicast}(g, m_1) \rightarrow \text{multicast}(g, m_2) \), 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 \( \rightarrow \)

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

• Same example as before, but causal ordering requires:
  (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 $P_2$ sees $m_2$ before $m_1$
  – To be correct, must hold back (delay) delivery of $m_2$ at $P_2$
  – 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

• Some care needed with dynamic groups
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 transitive 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)
    - Key insight: allow applications to define ordering operator(s)
Distributed mutual exclusion

• In first part of course, saw need to coordinate concurrent processes / threads
  – In particular considered how to ensure mutual exclusion: allow only 1 thread in a critical section

• A variety of schemes possible:
  – test-and-set locks; semaphores; monitors; active objects

• But most of these ultimately rely on hardware support (atomic operations, or disabling interrupts...)
  – not available across an entire distributed system

• Assuming we have some shared distributed resources, how can we provide mutual exclusion in this case?

Solution #1: central lock server

• Nominate one process C as coordinator
  – If \( P_i \) wants to enter critical section, simply sends lock message to \( C \), and waits for a reply
  – If resource free, \( C \) replies to \( P_i \) with a grant message; otherwise \( C \) adds \( P_i \) to a wait queue
  – When finished, \( P_i \) sends unlock message to \( C \)
  – \( C \) sends grant message to first process in wait queue
Central lock server: pros and cons

• Central lock server has some good properties:
  – **Simple** to understand and verify
  – **Live** (providing delays are bounded, and no failure)
  – **Fair** (if queue is fair, e.g. FIFO), and easily supports priorities if we want them
  – **Decent performance**: lock acquire takes one round-trip, and release is ‘free’ with asynchronous messages

• But **C** can become a performance bottleneck...
• ... and can’t distinguish crash of **C** from long wait
  – can add additional messages, at some cost

Solution #2: token passing

• Avoid central bottleneck
• Arrange processes in a logical ring
  – Each process knows its predecessor & successor
  – Single token passes continuously around ring
  – Can only enter critical section when possess token; pass token on when finished (or if don’t need to enter critical section)
Token passing: pros and cons

• Several advantages:
  – Simple to understand: only 1 process ever has token => mutual exclusion guaranteed by construction
  – No central server bottleneck
  – Liveness guaranteed (in the absence of failure)
  – So-so performance (between 0 and N messages until a waiting process enters, 1 message to leave)

• But:
  – Doesn’t guarantee fairness (FIFO order)
  – If a process crashes must repair ring (route around)
  – And worse: may need to regenerate token – tricky!

• And constant network traffic: an advantage???

Solution #3: Totally ordered multicast

• Scheme due to Ricart & Agrawala (1981)
• Consider N processes, where each process maintains local variable state which is one of { FREE, WANT, HELD }
• Invariant: At most one process is in HELD state at a time.
• To obtain lock, a process P_i sets state := WANT, and then multicasts lock request to all other processes
• When a process P_j receives a request from P_i:
  – If P_j’s local state is FREE, then P_j replies immediately with OK
  – If P_j’s local state is HELD, P_j queues the request to reply later
• A requesting process P_i waits for OK from N-1 processes
  – Once received, sets state := HELD, and enters critical section
  – Once done, sets state := FREE, & replies to any queued requests

• What about concurrent requests?
  – By concurrent we mean: P_j is already in the WANT state when it receives a request from P_i.
Handling concurrent requests

- Need to decide upon a **total order**:  
  - Each processes maintains a Lamport timestamp, $T_i$  
  - Processes put current $T_i$ into request message  
  - Insufficient on its own (recall that Lamport timestamps can be identical) => use process ID (or similar) to break ties  
  - Note: may not be “fair” as the same process always “wins”

- Hence if a process $P_j$ receives a request from $P_i$ and $P_j$ is also acquiring the lock (i.e. $P_j$’s local state is WANT)  
  - If $(T_j, P_j) < (T_i, P_i)$ then queue request from $P_i$  
  - Otherwise, reply with **OK**, and continue waiting

- Note that using the total order ensures **correctness**, but not **fairness** (i.e. no FIFO ordering)  
  - Q: can we fix this by using vector clocks?

**Totally ordered multicast: example**

- Imagine $P_1$ and $P_2$ simultaneously try to acquire lock...  
  - Both set **state** to WANT, and both send multicast message  
  - Assume that timestamps are 17 (for $P_1$) and 9 (for $P_2$)

- $P_3$ has no interest (**state** is FREE), so replies **Ok** to both

- $9 < 17$: $P_1$ replies **OK**; $P_2$ stays quiet & enqueues $P_1$

- $P_2$ enters the critical section and executes...

- and when done, replies to $P_1$ (to enter critical section)
Additional details

- Completely decentralized solution ... but:
  - Lots of messages (1 multicast + N-1 unicast)
  - OK for most recent holder to re-enter CS without any messages
- Variant scheme (Lamport) - multicast for total ordering
  - Processes each maintain (and collectively agree on) an ordered queue of requests and ACKs, relying on total ordering
  - To enter, process $P_i$ multicasts request($P_i, T_i$) [same as before]
  - On receipt of a message, $P_j$ replies with an ack($P_i, T_i$)
  - Processes adds all requests and ACKs to the queue in order
  - If process $P_i$ sees his request is earliest and ACK’d by all, can enter CS ... and when done, multicasts a release($P_i, T_i$) message
  - When $P_j$ receives release, removes $P_i$’s request from queue
  - If $P_j$’s request is now earliest in queue, can enter CS...
- Both Ricart & Agrawala and Lamport’s scheme have N points of failure: doomed if any process dies :-(

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