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\(_j\)(m\(_2\)) then deliver\(_i\)(m\(_1\)) \rightarrow deliver\(_j\)(m\(_2\))

- **Lamport clocks, \( L(e) \)**: an integer
  - Increment to \( \text{max} \) of (sender, receiver)) + 1 on receipt
  - But given \( L(a) < L(b) \), know nothing about order of \( a \) and \( b \)

- **Vector clocks**: list of Lamport clocks, one per process
  - Element \( V[i][j] \) captures #events at \( P_i \) observed by \( P_j \)
  - Crucially: if \( V_i(a) < V_j(b) \), can infer that \( a \rightarrow b \), and
    if \( V_i(a) \sim V_j(b) \), can infer that \( a \sim b \)
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 $a$, $b$, $c$ & $d$

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_ijn to {} (set of all post local snapshot messages received from P_k)
    - Then sends M to all other processes P_k and starts recording C_ijn = {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_ijn, and saves alongside local snapshot
  - Global snapshot comprises all local snapshots & C_ijn
  - 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 \( P_2 \) 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

```java
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 \( \neq (S_{ji}+1) \), 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 \( \text{multicast}(g, m_1) \rightarrow \text{multicast}(g, m_2) \), 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
  - 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

• Need some care with dynamic groups
  – must encode variable-length vector clock, typically using
    positional notation, and deal with joins and leaves
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)
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; event counts and sequencers; monitors; and 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 CS)
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 }
- To obtain lock, a process P, sets state := WANT, and then multicasts lock request to all other processes
- When a process Pj receives a request from Pi:
  - If Pj’s local state is FREE, then Pj replies immediately with OK
  - If Pj’s local state is HELD, Pj queues the request to reply later
- A requesting process Pj 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?