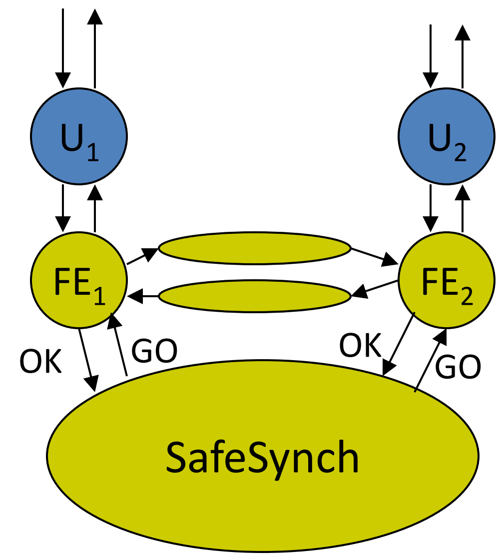
6.852: Distributed Algorithms
Fall, 2015

Lecture 13

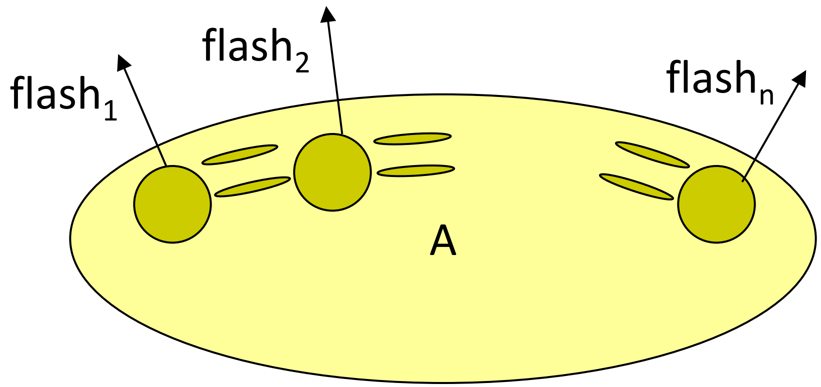
Last time



- Synchronizers, which allow us to run synchronous distributed network algorithms in an asynchronous network, with comparatively low costs in time and communication.
- Applications: BFS, Shortest Paths,...

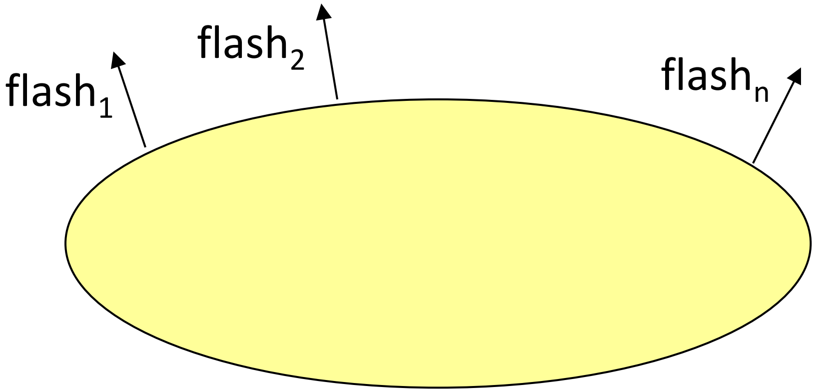
- Synchronizers achieve local synchronization.
- Now we show that achieving stronger, global synchronization must take considerable more time.

Lower Bound on Time for Synchronization



k-Session Problem

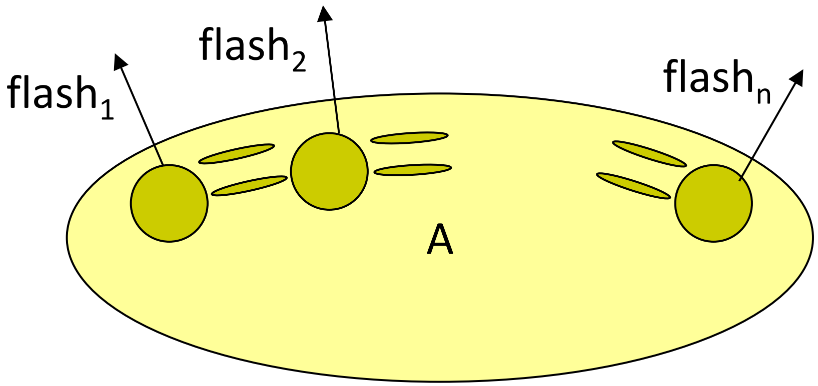
- Session:
  - Any sequence of $flash$ events containing at least one $flash_i$ event for each location i.



- k-Session problem:
  - Perform at least k separate sessions (in every fair execution), and eventually halt.

Asynchronous lower bound

- Consider a distributed algorithm A that solves the k-session problem.



- Time measure:
  - Timed execution: Fair execution with times labeling events, subject to upper bounds d and l.
  - $T(\alpha)$ = time of last flash in timed execution α.
  - $T(A)$ = supremum, over all timed executions α, of $T(\alpha)$.

Lower bound

- Theorem 2: If A solves the k-session problem then
  $T(A) \geq (k-1)\, diam\, d$.

- Proof:
  - By contradiction.
  - Suppose $T(A) < (k-1)\, diam\, d$.
  - Consider any slow timed execution α (all messages take time exactly d, the worst case).
  - α contains no flash event at a time $\geq (k-1)\, diam\, d$.
  - Decompose $\alpha = \alpha_1\, \alpha_2\, \ldots \alpha_{k\text{-}1}\ \alpha''$, where:
    - Time of last event before α'' is $< (k-1)\, diam\, d$.
    - No flash events occur in α''.
    - In each α_r the difference between the times of the first and last events is $< diam\, d$.

Lower bound

- Reorder events in α, while preserving dependencies:
  - Events of same process.
  - Send and corresponding receive.
- Consider processes j_0 and j_1, $dist(j_0, j_1) = \ diam$.
- Reorder within each α_r separately:
  - For α_1: Reorder to $\beta_1 = \gamma_1 \, \delta_1$, where:
    - γ_1 contains no event of j_0, and
    - δ_1 contains no event of j_1.
  - For α_2: Reorder to $\beta_2 = \gamma_2 \, \delta_2$, where:
    - γ_1 contains no event of j_1, and
    - δ_1 contains no event of j_0.
  - And alternate thereafter.

Lower bound, cont'd

- If the reordering yields a fair execution of A (can ignore timing), then we get a contradiction, because it contains at most $k-1$ sessions:
  - No session entirely within γ_1, (no event of j_0).
  - No session entirely within $\delta_1 \gamma_2$ (no event of j_1).
  - No session entirely within $\delta_2 \gamma_3$ (no event of j_0).
  - …
  - Thus, every session must span some γ_r - δ_r boundary.
  - But, there are only $k-1$ such boundaries.

- It remains only to construct the reordering…

Constructing the reordering

- E.g., consider α_r for r odd.
- Need $\beta_r = \gamma_r\, \delta_r$, where γ_r contains no event of j_0, δ_r no event of j_1.
- If α_r contains no event of j_0, don't reorder, define $\gamma_r = \alpha_r$, $\delta_r = \lambda$.
- If α_r contains no event of j_1, don't reorder, define $\gamma_r = \lambda$, $\delta_r = \alpha_r$.
- Now assume α_r contains events of both j_0 and j_1.

- Claim: No event of j_1 depends on any event of j_0.
- Why: Insufficient time for messages to travel from j_0 to j_1:
  - Execution α is slow (message deliveries take time d).
  - Time between first and last events in α_r is < $diam\ d$.
  - j_0 and j_1 are $diam$ hops apart.

- Reorder α_r to β_r, in which all events of j_0 follow all events of j_1.
- γ_r is the part ending with the last event of j_1, δ_r the rest.

Today’s plan

- Logical time
- Applications of logical time
- Weak logical time and vector timestamps
- Reading:
  - Chapter 18
  - [Lamport 1978] Time, Clocks, and the Ordering of Events in a Distributed System
  - [Mattern]
- Next:
  - Consistent global snapshots
  - Stable property detection
  - Reading: Chapter 19

Logical Time

[Lamport] Time, clocks,…

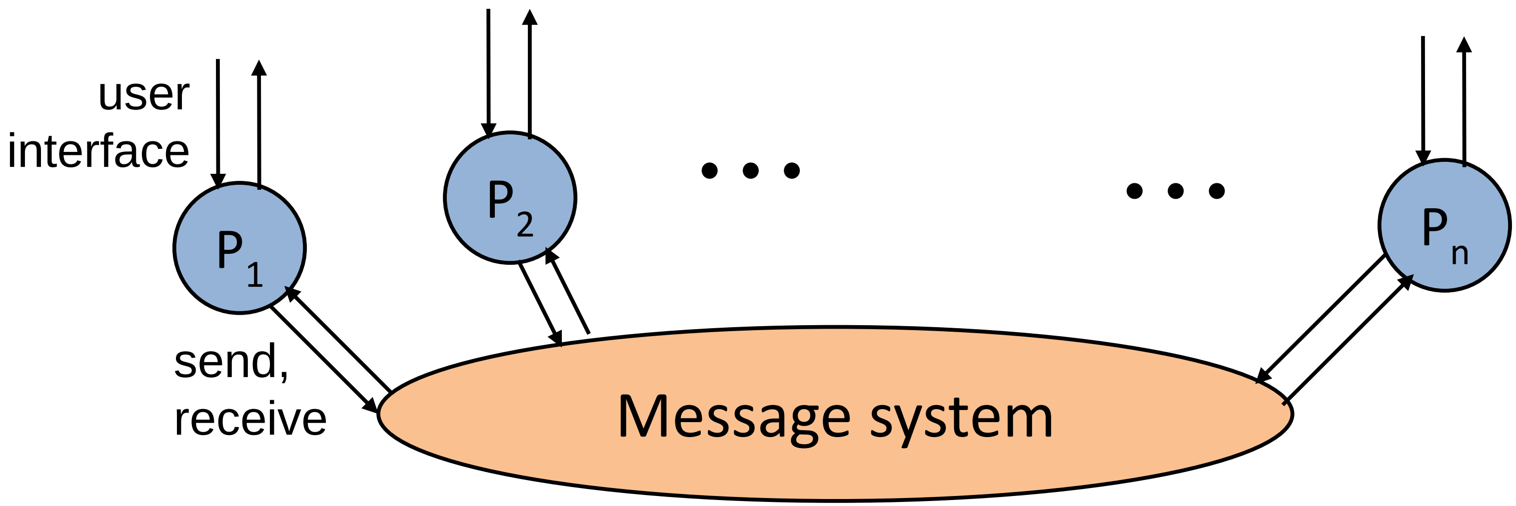
- Winner of first Dijkstra Prize, 2000.

> “Jim Gray once told me that he heard two different opinions of this paper: that's it trivial and that it's brilliant. I can't argue with the former, and I'm disinclined to argue with the latter.” –Lamport

Logical time

- An important abstraction, which simplifies programming for asynchronous networks
- Imposes a single total order on events occurring at all locations.
- Processes know the order.
- Assign logical times (elements of some totally ordered set T, such that the real numbers) to all events in an execution of an asynchronous network system, subject to some properties that make the logical times “look like real times”.
- Applications:
  - Global snapshot
  - Replicated state machines
  - Distributed mutual exclusion
  - …

Logical time



- Consider a send/receive system A with FIFO channels, based on a strongly connected digraph.
- Events of A:
  - User interface events
  - Send and receive events
  - Internal events of process automata
- Q: What conditions should logical times satisfy?

Logical time

- For execution α, function $ltime$ from events in α to totally-ordered set T is a logical time assignment if:
  1. $ltimes$ are distinct: $ltime(e_1) \neq ltime(e_2)$ if $e_1 \neq e_2$.
  2. $ltimes$ of events at each process are monotonically increasing.
  3. $ltime$(send) < $ltime$(receive) for the same message.
  4. For any t, the number of events e with $ltime(e) < t$ is finite. (No "Zeno" behavior.)
- Properties 2 and 3 say that the $ltimes$ are consistent with dependencies between events.
- Under these conditions, $ltime$ "looks like" real time, to all the processes individually:
- Theorem: For every fair execution α with an $ltime$ function, there is another fair execution α' in which the events appear in $ltime$ order such that $\alpha \mid P_i = \alpha' \mid P_i$ for every i.

Logical time

- For execution α, $ltime$ is a logical time assignment if:
  1. $ltimes$ are distinct: $ltime(e_1) \neq ltime(e_2)$ if $e_1 \neq e_2$.
  2. $ltimes$ of events at each process are monotonically increasing.
  3. $ltime$(send) < $ltime$(receive) for the same message.
  4. For any t, the number of events e with $ltime(e) < t$ is finite.
- Theorem: For every fair execution α with an $ltime$ function, there is another fair execution α' with events in $ltime$ order such that $\alpha | P_i = \alpha' | P_i$ for every i.
- Proof:
  - Use properties of $ltime$.
  - Reorder actions of α in order of $ltimes$; a unique such sequence exists, by Properties 1 and 4.
  - By Properties 2, and 3, this reordering preserves all dependencies, so we can fill in the states to give the needed execution α'.
  - Indistinguishable to each process because we preserve all dependencies.

Logical time

- For execution α, $ltime$ is a logical time assignment if:

  1. $ltimes$ are distinct: $ltime(e_1) \neq ltime(e_2)$ if $e_1 \neq e_2$.

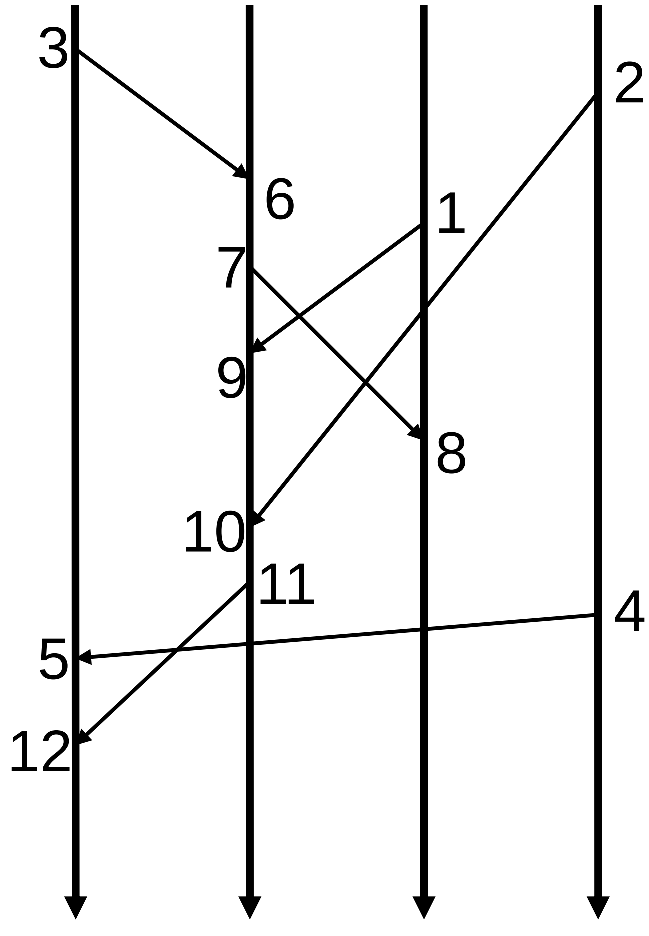
  2. $ltimes$ of events at each process are monotonically increasing.

  3. $ltime$(send) < $ltime$(receive) for the same message.

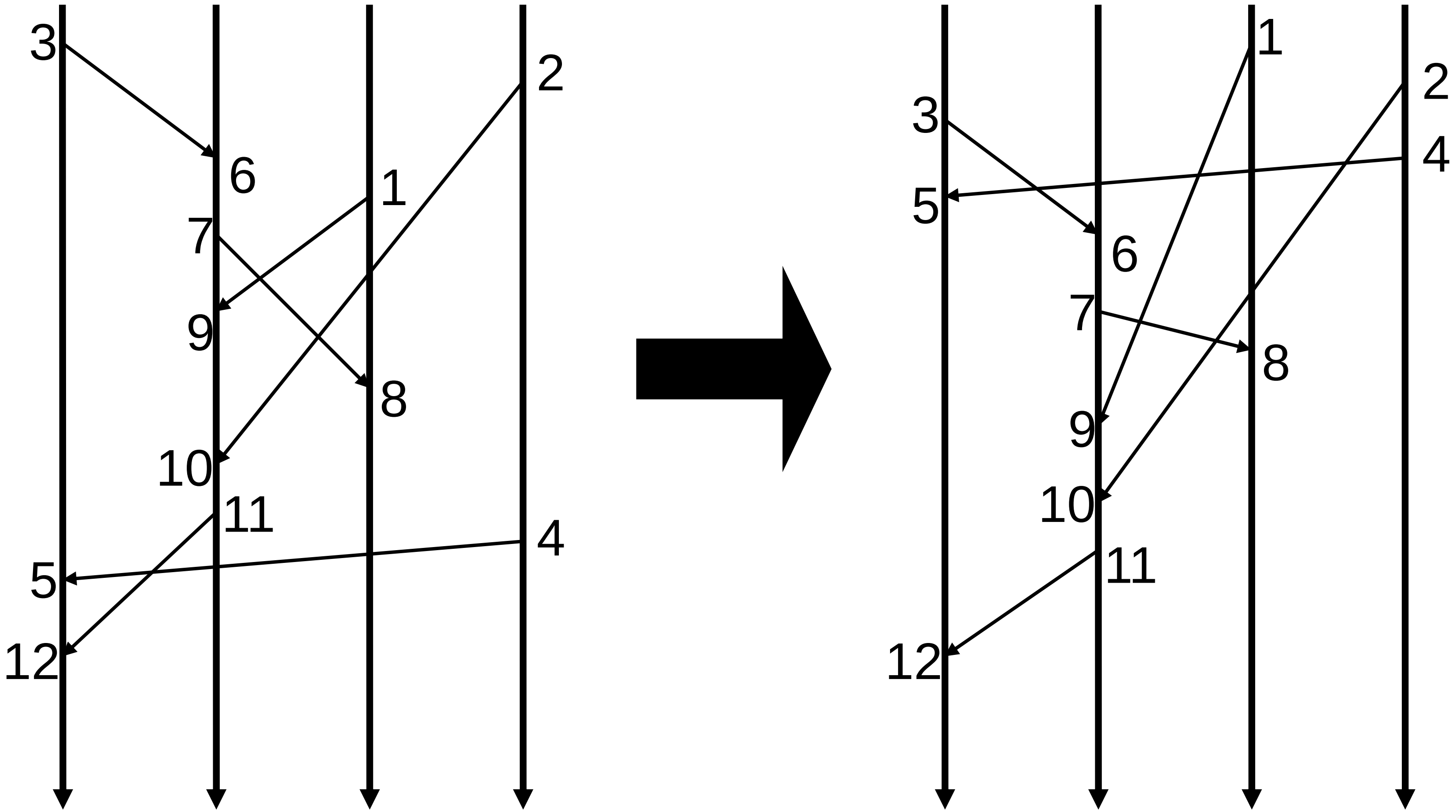
  4. For any t, the number of events e with $ltime(e) < t$ is finite.

- Combination of dependencies described in Properties 2 and 3 is often called causality, or Lamport causality.
- Common way to represent dependencies: a Causality Diagram:

Logical time



Logical time



Lamport's algorithm for generating logical times

- Based on timestamping idea of Johnson and Thomas.
- Each process maintains a local nonnegative integer $clock$ variable, used to count steps.
- $clock$ is initially 0.
- Every event of the process (send, receive, internal, or user interface) increases $clock$:
  - When process does an internal or user interface step, it increments $clock$.
  - When process sends, it first increments $clock$, then piggybacks the new value c on the message, as a $timestamp$.
  - When a process receives a message with timestamp c, it increases $clock$ to $\max(clock, c) + 1$.
- Using the clocks to generate logical time for events:
  - $ltime$ of an event is (c, i), where
    - $c = clock$ value immediately after the event
    - $i = $ process index, to break ties
  - Order the (c, i) pairs lexicographically.

Lamport’s algorithm generates logical times

1. Events’ $ltimes$ are unique.
   - Because the $clock$ at each process is increased at every step and we use process indices as tiebreakers.
2. Events of each individual process have strictly increasing $ltimes$.
   - The rules ensure this.
3. $ltime$(send) < $ltime$(receive) for same message.
   - By the way the receiver determines the clock after the receive event.
4. Non-Zeno.
   - Because every event increases the local clock by at least 1 and there are only finitely many processes.

Welch's algorithm

- What if we already have clocks?
  - Monotonically non-decreasing, unbounded.
  - Can't change the clock (e.g., maintained by a separate algorithm, or arrive from some external time source).
- Welch's algorithm:
  - Idea: Instead of advancing the $clock$ in response to received timestamps, simply delay the receipt of "early" messages.
  - Each message carries the $clock$ value from the sender.
  - Receiver puts incoming messages in a FIFO buffer.
  - At each locally-controlled step, first remove from the buffer all messages whose timestamps < current $clock$, and process them, in the same order in which they appear in the buffer.
  - Logical time of event is (c, i, k), order lexicographically.
    - c = local clock value when event "occurs"
      - receive event is said to "occur" when message is removed from buffer, not when it first arrives.
    - i = process index, first-order tiebreaker
    - k = sequence number, second-order tiebreaker

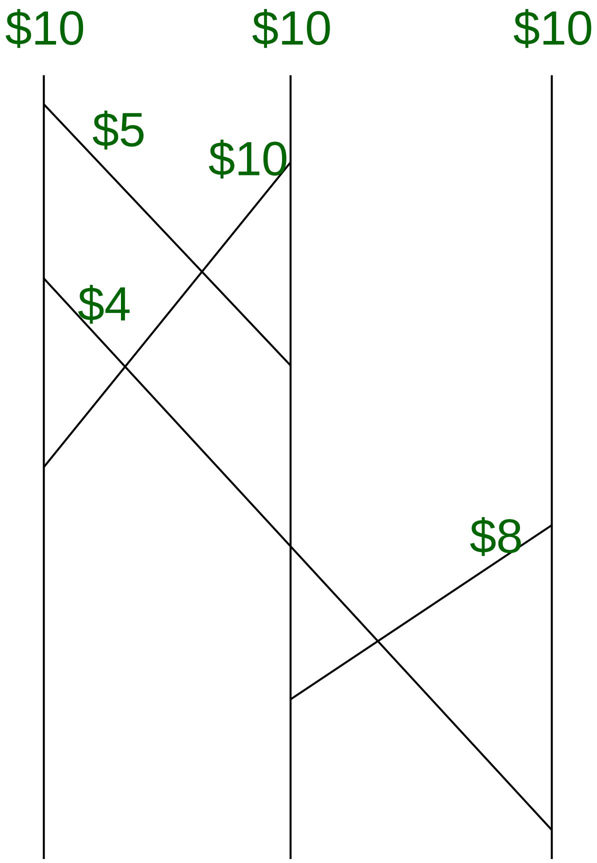
Logical time in broadcast systems

- Analogous definition and theorem:
- For execution α, function *ltime* from events in α to totally-ordered set T is a logical time assignment if:
  1. *ltimes* are distinct: $ltime(e_1) \neq ltime(e_2)$ if $e_1 \neq e_2$.
  2. *ltimes* of events at each process are monotonically increasing.
  3. *ltime*(bcast) < *ltime*(receive) for the same message.
  4. For any t, the number of events e with $ltime(e) < t$ is finite. (No "Zeno" behavior.)
- Theorem: For every fair execution α with an *ltime* function, there is another fair execution α' with events in *ltime* order such that $\alpha | P_i = \alpha' | P_i$ for every i.

Applications of Logical Time

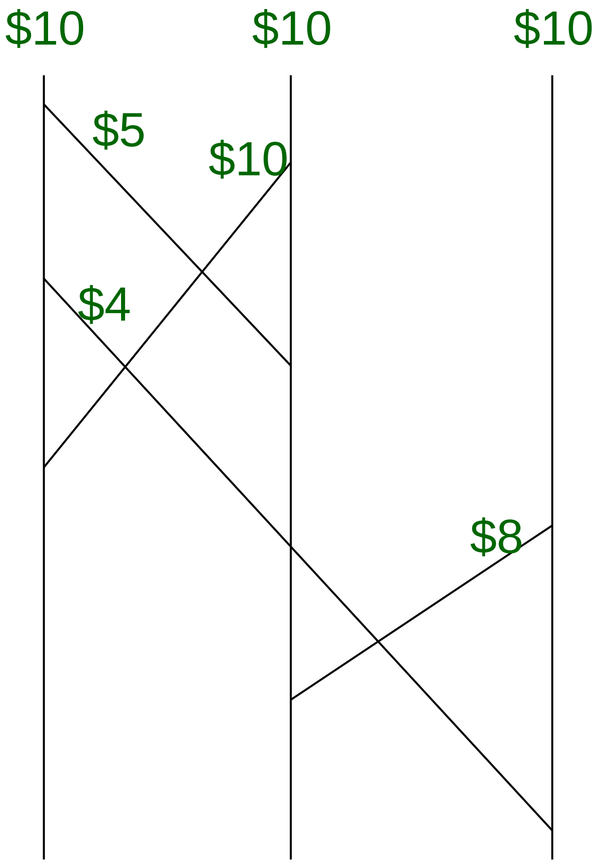
Banking System

- Distributed banking system with transfers (no external deposits or withdrawals).
- Assume:
  - Asynchronous send/receive system.
  - Each process has an $account$ with an integer amount of money ≥ 0.
  - Processes can send money at any time to anyone.
    - Send message with value, subtract value from $account$.
    - Add value received in message to $account$.
  - Add “dummy” $0 transfers (heartbeat messages).



Banking System

- Algorithm triggered by input signal to one or more processes; processes awaken upon receiving either such a signal or a message from another process.
- Require:
  - Each process should output local balance, so that total of the balances = correct amount of money in the system.
    - Well-defined because there are no deposits/withdrawals.
  - Don't "interfere" with underlying money transfer, just "observe" it.



Banking system algorithm

- Assume logical-time algorithm, which assigns logical times to all banking system events.
- Algorithm uses an agreed-upon logical time value t.
- Each process determines the value of its $account$ at logical time t.
  - Specifically, after all events with $ltime \leq t$ and before all events with $ltime > t$.
- Each process determines, for each incoming channel, the amount of money in transit at logical time t.
  - Specifically, money in messages sent at $ltime \leq t$ and received at $ltime > t$.
  - Attach $ltime$ of send event to each message as a timestamp.
  - Start counting from when local $ltime$ first becomes $> t$, stop when message timestamp $> t$.
- Q: What if local $ltime > t$ when a node wakes up?
  - Keep logs just in case, or
  - Keep retrying with different values of t.

Applications of logical time: Global snapshot

- Generalizes banking system.
- Assume:
  - Arbitrary asynchronous send/receive system A that sends infinitely many messages on each channel.
- Require:
  - Global snapshot of system state (nodes and channels) at some point after a triggering input.
  - Should not interfere with the system's operation.
- Useful for debugging, system backups, detecting termination.
- Use same strategy as for bank audit:
  - Select logical time, all snap at that time (nodes and channels).
  - Combining all these results give global snapshot of an "equivalent" execution.

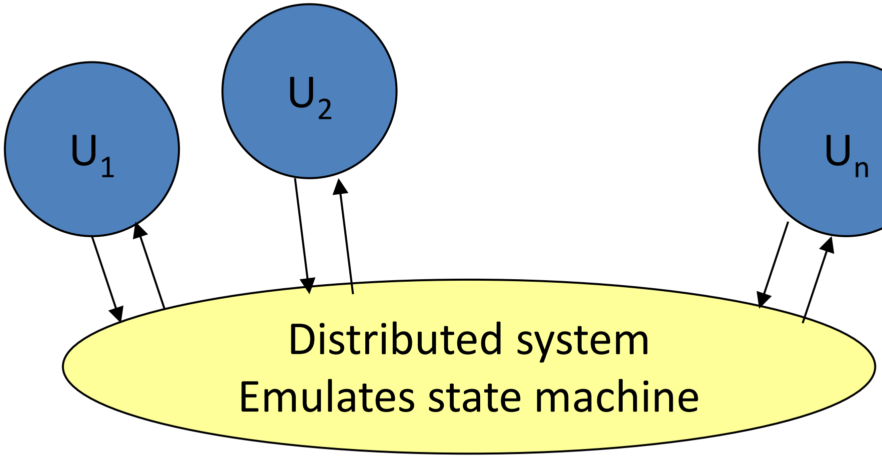
Another application:
Replicated State Machines (RSMs)

Replicated State Machines (RSMs)

- Important use of logical time.
- A focal point of Lamport's paper.
- Allows a distributed system to simulate a single centralized state machine.
- Centralized state machine:
  - V: Set of possible states
  - v_0: Initial state
  - $invs$: Set of possible invocations
  - $resps$: Set of possible responses
  - $trans$: $invs \times V \rightarrow resps \times V$: Transition function
- Same formal definition as shared variable, defined in Chapter 9 (next week).

Replicated State Machines

- Users of distributed system submit invocations, get responses in well-formed manner (blocking invocations).



- Want system to look like “atomic” version of the centralized state machine (defined in Chapter 13).
- Allows possible delays before and after actually operating on the state machine).
- Could weaken requirement to “serializability”, same idea but allows reordering of events at different nodes.

RSM algorithm

- Assume broadcast network.
- First attempt:
  - Originator of an invocation broadcasts the invocation to all processes (including itself).
  - All processes (including the originator) perform the transition on their copies when they receive the messages.
  - When the originator performs the transition, it determines the response to pass back to the user.
- Not quite right---all processes should perform the transitions in the same order.
- So, use logical time to order the invocations.

RSM algorithm

- Assume logical times.
- Originator of an invocation $bcast$s the invocation to all processes, including itself; attaches the logical time of the $bcast$ event.
- Each process maintains state variables:
  - X: Copy of the machine state.
  - $invbuffer$: Invocations it has heard about and their timestamps
    - Timestamp = logical time of $bcast$ event.
  - $knowntime$: Vector giving largest known logical time for each process
    - For itself: Logical time of last local event.
    - For each other node j: Timestamp of last message received from j.
- Process may perform invocation π from its $invbuffer$, on its copy X of the machine state, when:
  - $timestamp(\pi)$ is the smallest timestamp of any invocation in $invbuffer$, and
  - $knowntime(j) \geq timestamp(\pi)$ for every j.
- After performing π, remove it from $invbuffer$.
- If π originated locally, then also respond to the user.

Correctness

- Liveness: Termination for each operation
  - LTTR. Depends on logical times growing unboundedly and all nodes sending infinitely many messages.
- Safety: Atomicity (each operation "appears to be performed" at a point in its interval, as in a centralized machine):
  - Each process applies operations in the same (logical time) order.
    - FIFO channels ensure that no invocations are "late".
  - Each operation "appears to be performed" at a point in its interval:
    - Define a serialization point for each operation π---a point in π's interval where we can "pretend" π occurred.
    - Here, serialization point for π can be the earliest point when all processes have reached the logical time t of π's bcast event.
    - Claim this point is within π's interval:
      - It's not before the invocation, because the originating process doesn't reach time t until after the invocation arrives.
      - It's not after the response, because the originator waits for all $knowntime$s to reach t before applying the operation and responding to the user.

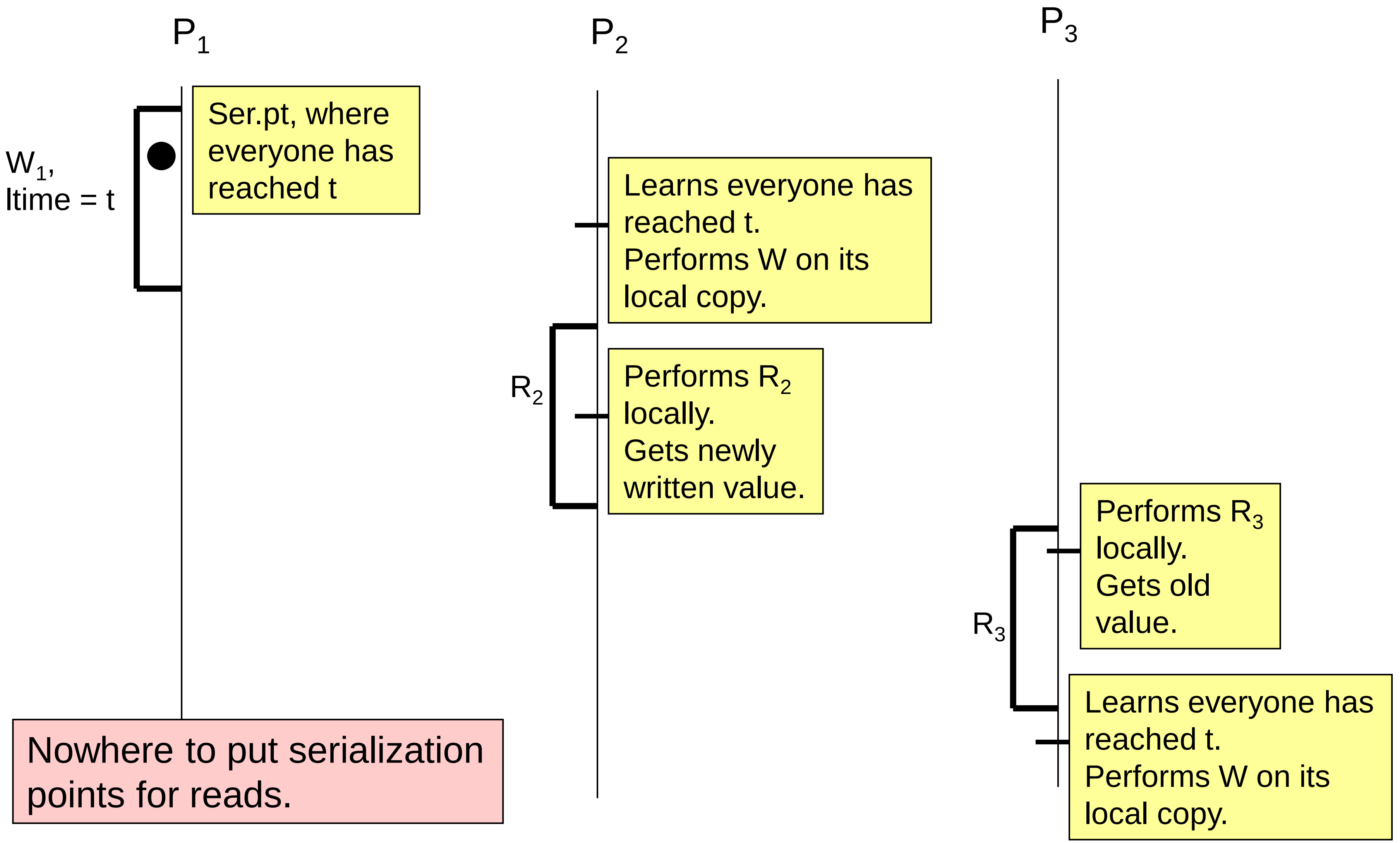
Safety, cont'd

- Safety: Atomicity (each operation "appears to be performed" at a point in its interval, as in a centralized machine):
  - Each process applies operations in the same (logical time) order.
  - Define serialization point for each operation π to be the earliest point when all processes have reached the logical time t of π's bcast event.
  - This point is within π's interval.
  - The order of the serialization points is the same as the logical time order, which is the same as the order in which the operations are performed on all copies.
  - So, responses are consistent with the order of serialization points.
  - That is, it looks to all the users as if the operations occurred at their serialization points---as in a centralized machine.

Special handling of reads

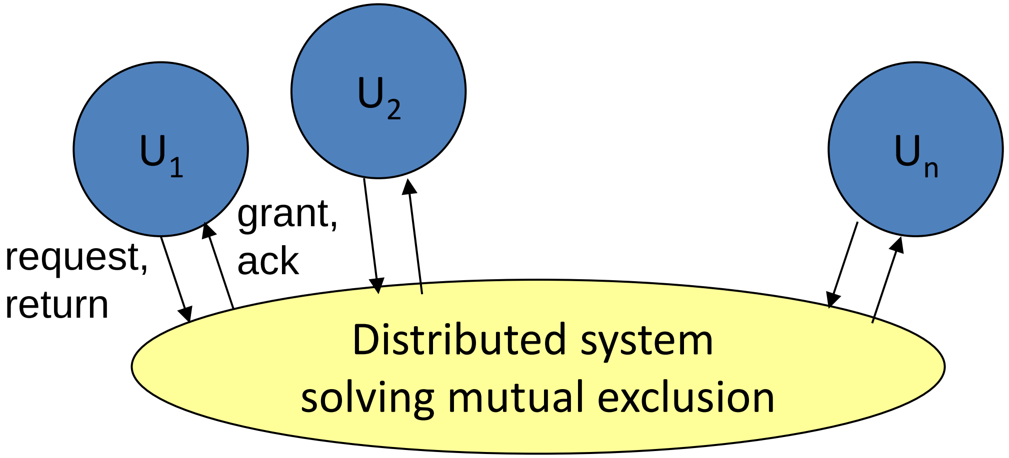
- Don't bcast---just perform them locally.
- Now, doesn’t satisfy atomicity.
- Satisfies weaker property, serializability.

No serialization points…



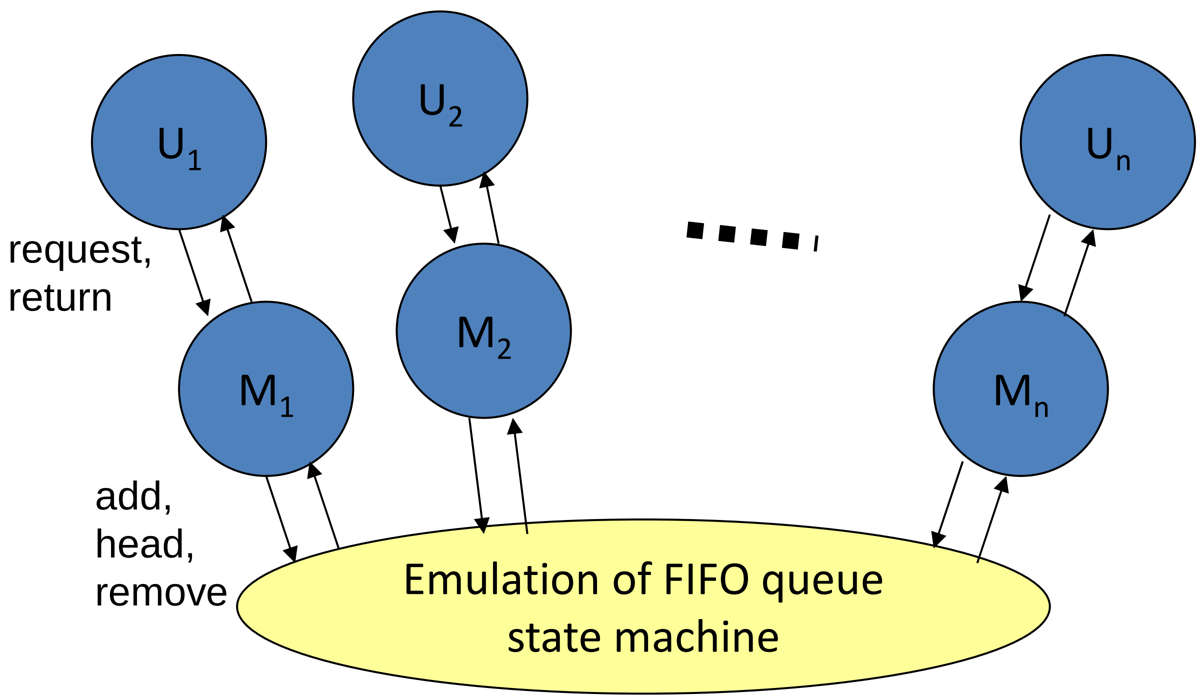
Application of RSM: Distributed mutual exclusion

- Distributed mutual exclusion problem:
  - Users at different locations submit requests for a resource from time to time.
  - System grants requests, so that:
    - No two users get the resource at the same time, and
    - Every request is eventually granted.
  - Users must return the resource.
- Can solve distributed mutual exclusion using a distributed simulation of a centralized state machine.
- See book, p. 609-610.



Distributed mutual exclusion

- Use one emulated FIFO queue state machine:
  - State contains a FIFO $queue$ of process indices.
  - Operations:
    - $add(i)$, i a process index: Adds i to end of $queue$.
    - $head$: Returns head of $queue$, or “empty”.
    - $remove(i)$: Removes all occurrences of i from $queue$.



Distributed mutual exclusion

- Given (emulated) shared queue, mutex processes cooperate to implement mutual exclusion.
- Process i operates as follows:
  - To request the resource:
    - Invoke $add(i)$, adding i to the end of the queue.
    - Repeatedly invoke $head$, until the response yields index i.
    - Then grant the resource to the local user.
  - To return the resource:
    - Invoke $remove(i)$.
    - Return ack to user.
- Complete distributed mutual exclusion algorithm:
  - Use Lamport's logical time algorithm to give logical times.
  - Use RSM algorithm, based on logical time, to emulate the shared queue.
  - Use mutex algorithm above, based on shared queue.

Weak Logical Time and Vector Timestamps

Weak Logical Time

- Logical time imposes a total ordering on events, assigning them values from a totally-ordered set T.
- Sometimes we don't need to order all events---it may be enough to order just the ones that are causally dependent.
- Mattern (also Fidge) developed an alternative notion of logical time based on a partial ordering of events, assigning them values from a partially-ordered set P.
- Function ltime from events in α to partially-ordered set P is a weak logical time assignment if:
  1. $ltimes$ are distinct: $ltime(e_1) \neq ltime(e_2)$ if $e_1 \neq e_2$.
  2. $ltimes$ of events at each process are monotonically increasing.
  3. $ltime$(send) < $ltime$(receive) for the same message.
  4. For any t, the number of events e with $ltime(e) < t$ is finite.
- Same as for logical time, but using partial order.

Weak Logical Time

- In fact, Mattern's partially-ordered set P represents causality exactly.
- Timestamps of two events are ordered in P if and only if the two events are causally related (related by the causality ordering).
- Might be useful in distributed debugging: A log of local executions with weak logical times could be observed after the fact, used to infer causality relationships among events.

Algorithm for weak logical time

- Based on vector timestamps: vectors of nonnegative integers indexed by processes.
- Algorithm:
  - Each process maintains a local vector clock, called $vclock$.
  - When an event occurs at process i, it increments its own component of its $vclock$, which is $vclock(i)$, and assigns the new $vclock$ to be the vector timestamp of the event.
  - Whenever process i sends a message, it attaches the vector timestamp of the send event.
  - When i receives a message, it first increases its $vclock$ to the component-wise maximum of its current $vclock$ and the incoming vector timestamp. Then it increments its $vclock(i)$ as usual, and assigns the new $vclock$ to the receive event.
- A process' $vclock$ represents the latest known "tick values" for all processes.
- Partially ordered set P:
  - The vector timestamps, ordered based on $\leq$ in all components.
  - $V \leq V'$ if and only if $V(i) \leq V'(i)$ for all i.

Key theorems about vector clocks

- Theorem 1: The vector clock assignment is a weak logical time assignment.
- Lemma 1: If event π causally precedes event π', then the logical times are ordered, in the same order.
- Proof:
  - True for direct causality.
  - Use induction on number of direct causality relationships.
- Claim this assignment exactly captures causality:
- Lemma 2: If the vector timestamp V of event π is (component-wise) $\leq$ the vector timestamp V' of event $\pi' \neq \pi$, then π causally precedes π'.
- Proof: Prove the contrapositive: Assume π does not causally precede π' and show that V is not $\leq V'$.

Proof of Lemma 2

- Lemma 2: If the vector timestamp V of event π is (component-wise) $\leq$ the vector timestamp V' of event $\pi' \neq \pi$, then π causally precedes π'.
- Proof:
  - Prove the contrapositive: Assume π does not causally precede π' and show that V is not $\leq V'$.
  - Case 1: π and π' are events of the same process i.
    - Then since π does not causally precede π', it must be that π' precedes π in time.
    - Then $V'(i) < V(i)$.
    - So V is not $\leq V'$.
  - Case 2: π is an event of process i and π' an event of another process $j \neq i$.

Proof of Lemma 2

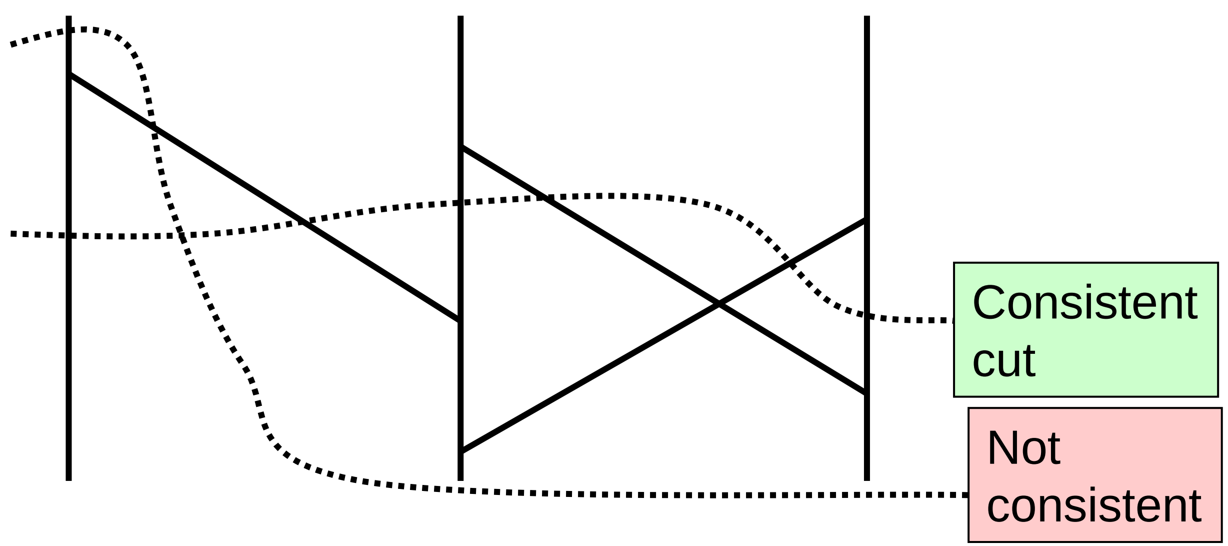
- Lemma 2: If the vector timestamp V of event π is (component-wise) $\leq$ the vector timestamp V' of event $\pi' \neq \pi$, then π causally precedes π'.
- Proof:
  - Prove the contrapositive: Assume π does not causally precede π' and show that V is not $\leq V'$.
  - Case 2: π is an event of process i and π' an event of process $j \neq i$.
    - i increases its $vclock(i)$ for event π, say to value t.
    - Without causality, there is no way for this value t for i to propagate to j before π' occurs.
    - So, when π' occurs at process j, j's $vclock(i) < t$.
    - So V is not $\leq V'$.

Back to Theorem 1

- Theorem 1: The vector clock assignment is a weak logical time assignment.
- Lemma 1: If event π causally precedes event π', then the logical times are ordered, in the same order.
- Lemma 2: If the vector timestamp V of event π is (component-wise) $\leq$ the vector timestamp V' of event $\pi' \neq \pi$, then π causally precedes π'.
- Proof of Theorem 1:
  - The ordering is a partial order.
  - Lemma 1 yields Properties 2 and 3.
  - Lemma 2 yields Property 1 (uniqueness).
  - Property 4 (non-Zeno) LTTR.

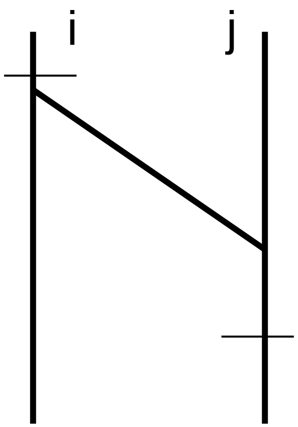
Another important theorem about vector timestamps [Mattern]

- Relates timestamps to consistent cuts of the causality graph.
- Cut: A point between events at each process.
  - Specify a cut by a vector giving the number of preceding steps at each location.
- Consistent cut: "Closed under causality": If event π causally precedes event π' and π' is before the cut, then so is π.
- Example:



The theorem

- Consider any particular cut.
- Let V_i be the vector clock of process i exactly at i's cut-point.
- Then $V = \max(V_1, V_2, \dots, V_n)$ gives the maximum information obtainable by combining everyone knowledge at the cut-points.
  - Component-wise max.
- Theorem 2: The cut is consistent iff, for every i, $V(i) = V_i(i)$.
- That is, the maximum information about i that anyone knows at its cut point is the same as what i knows about itself at its cut point.
- "No one else knows more about i than i itself knows."
- Rules out j receiving a message before its cut point that i sent after its cut point; in that case, j would have more information about i than i had about itself.



The theorem

- Let V_i be the vector clock of process i exactly at i's cut-point.
- $V = \max(V_1, V_2, \ldots, V_n)$.
- Theorem 2: The cut is consistent iff, for every i, $V(i) = V_i(i)$.
- Stated slightly differently:
- Theorem 2: The cut is consistent iff, for every i and j, $V_j(i) \leq V_i(i)$.
- Proof: LTTR (see Mattern's paper).

- Q: What is this good for?

Application: Debugging

- Theorem 2: The cut is consistent iff, for every i and j, $V_j(i) \leq V_i(i)$.
- Example: Debugging
  - Each node keeps a log of its local execution, with vector timestamps for all events.
  - Collect information, find a cut for which $V_j(i) \leq V_i(i)$ for every i and j. (Mattern gives an algorithm to do this.)
  - By Theorem 2, this is a consistent cut.
  - Such a cut yields:
    - States for all processes at the cut, and
    - Information about messages sent before the cut and not received until after the cut, i.e., messages "in transit" at the cut.
  - Put this together, get a "consistent" global state (we will study this next time).
  - Use this to check correctness properties for the execution, e.g., invariants.

Next time

- Consistent global snapshots
- Stable property detection
- Reading: Chapter 19