6.852 Lecture 11

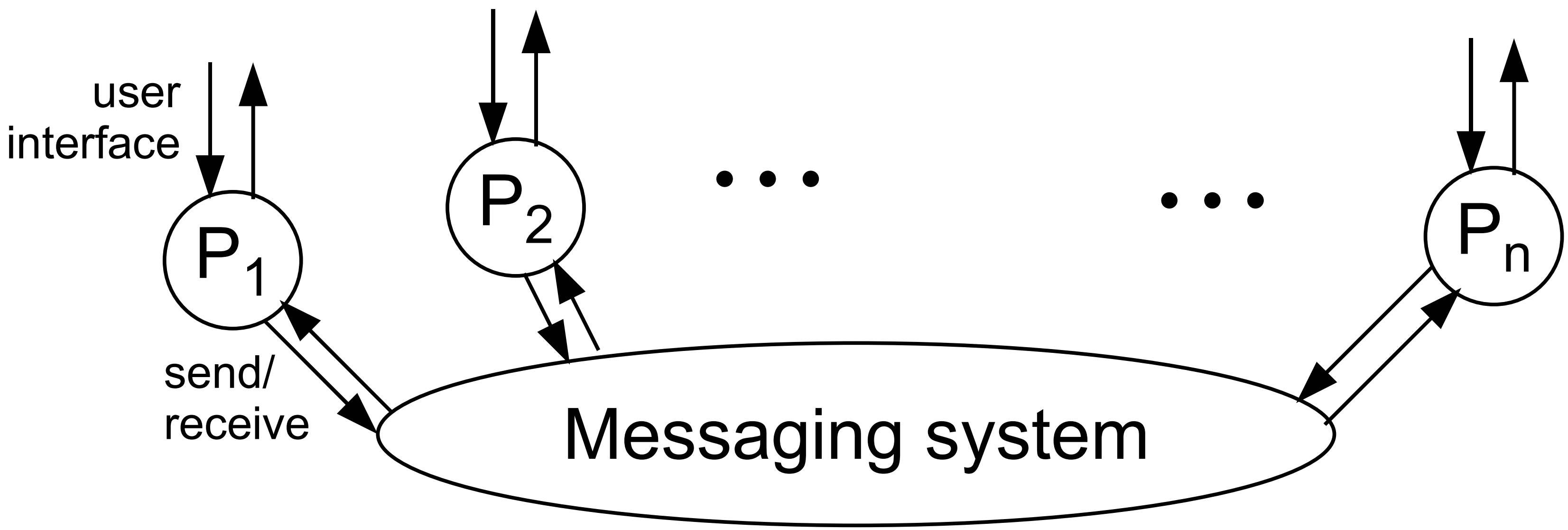
- Logical time
- Replicated state machines
- Reading: Chapter 18, Lamport paper (1978)

> “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

- Simplify asynchronous setting by making it appear sequential (cf. synchronizers)
- Problem: assign “logical time” to all events in an execution, should “look like real time”
  - each process should know the logical time of its events
- Ordering events at different locations
  - the problem of simultaneity (cf. relativity, interleaving semantics)
  - causality and the “happens before” relation
- Applications
  - global snapshot
  - replicated state machines

Logical time

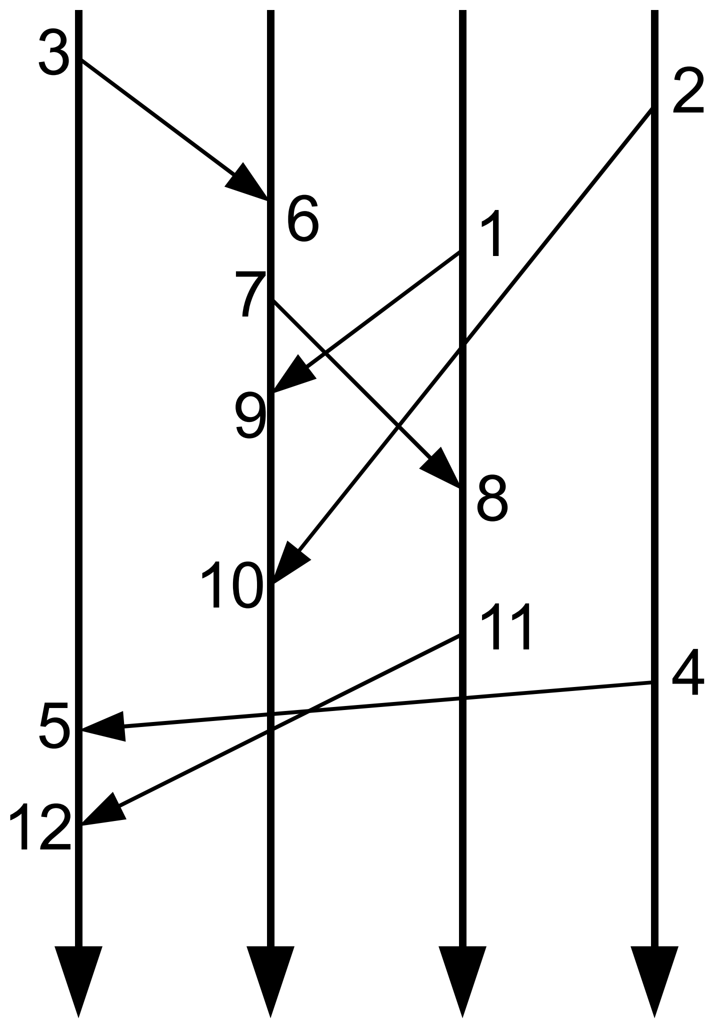


- Consider a send/receive system with FIFO channels
  - user interface events
  - send/receive events
  - internal events of process automata
- What conditions must logical times satisfy?

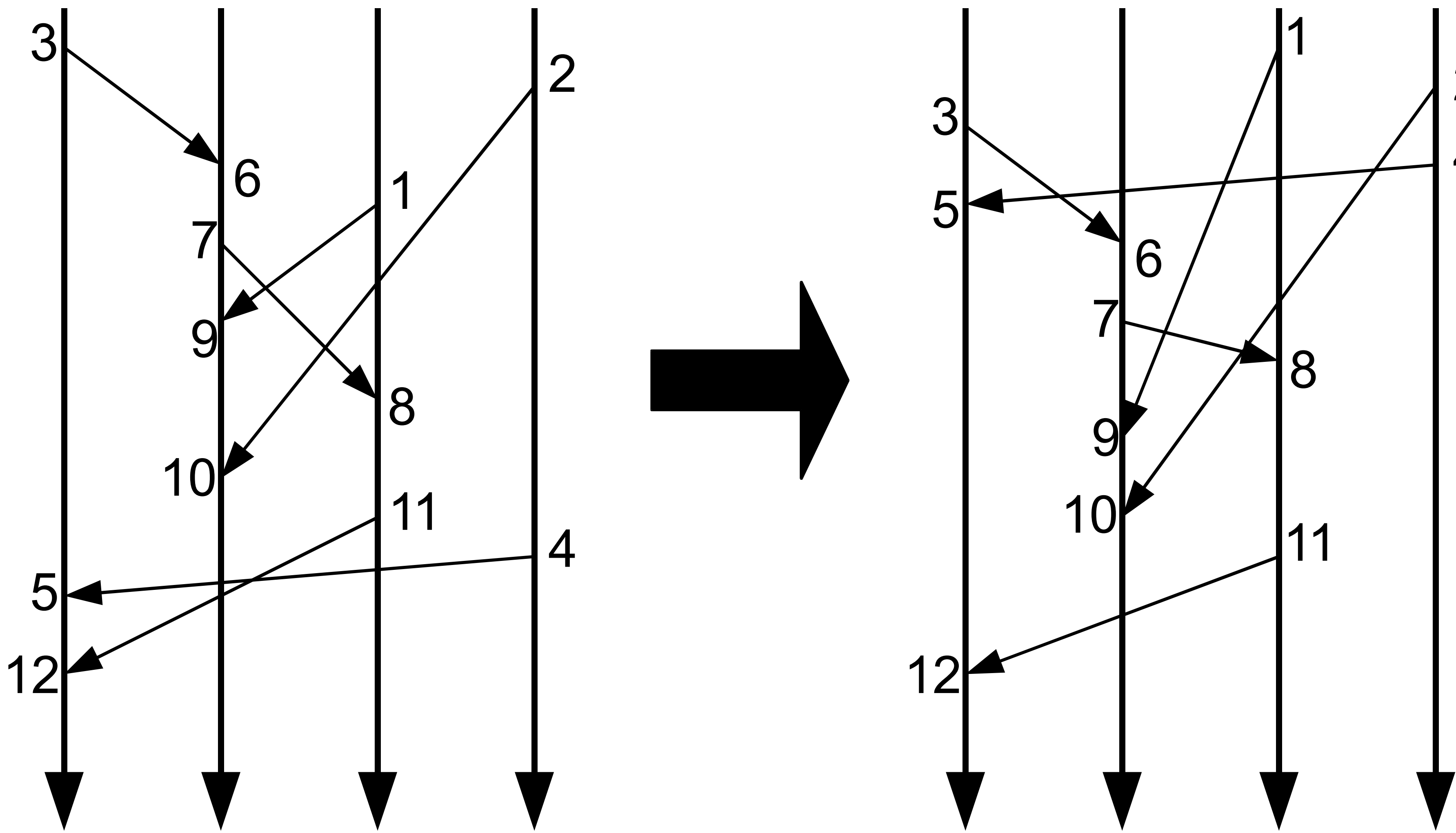
Logical time

- For execution α, function **ltime** from events in α to reals is a **logical time assignment** if:
  1. **ltime**s are distinct: **ltime**$(e_1) \neq$ **ltime**(e_2) if $e_1 \neq e_2$
  2. **ltime**s of events at each process are monotonically increasing
  3. **ltime**(send) < **ltime**(receive) for same message
  4. for any t, number of events e with **ltime**(e) < t is finite
- Theorem: For all fair execs α, there is an fair exec α' with events in **ltime** order such that $\alpha'|P_i = \alpha|P_i$ for all i.
  - **ltime** "looks like real time" (indistinguishable to each process)
  - use properties of **ltime** to prove
    - unique α' by properties 1 and 4
    - indistinguishable to each process by **causality** (prop 2 and 3)

Logical time



Logical time



Logical time

- Initial algorithm by Lamport
  - based on timestamping algorithm by Johnson and Thomas
  - each process maintains local “clock” (a natural number)
    - every event of process increases clock by at least 1
  - every event increments clock
  - on every msg sent, piggyback clock value (after incrementing)
  - when msg received,
    - take max of current clock and value in msg, then increment

Logical time

- Initial algorithm by Lamport
  - each process maintains local “clock” (a natural number)
  - every event increments clock
  - on every msg sent, piggyback clock value (after incrementing)
  - when msg received,
    - take max of current clock and value in msg, then increment
  - logical time of an event is (c,i)
    - c = clock value immediately after event
    - i = process index, to break ties
    - number of processes must be finite

Logical time

- What if we already have clocks?
  - monotone, nondecreasing, unbounded
  - can't change the clock (maintained by external service)
- Alternative algorithm by Welch
  - Idea: delay “early” messages
    - msgs sent carry clock value
    - buffer msgs received until local clock value ≥ msg clock value
  - logical time of event is (c,i,k)
    - c = local clock value when event “occurs” (well-defined?)
      - receive events “occur” when **removed** from buffer
    - i = process index
    - k = sequence number (second-order tiebreaker)

Logical time

- Analogous definition for broadcast systems:
- For execution α, function **ltime** from events in α to reals is a logical time assignment if:
  1. **ltime**s are distinct: **ltime**(e_1) ≠ **ltime**(e_2) if $e_1 \neq e_2$
  2. **ltime**s of events at each process are monotonically increasing
  3. **ltime**(bcast) < **ltime**(receive) for same message
  4. for any t, number of events e with **ltime**(e) < t is finite
- Theorem: For all fair execs α, there is an fair exec α' with events in **ltime** order such that $\alpha'|P_i = \alpha|P_i$ for all i.

Applications of logical time

- Banking system with transfers (no deposit/withdrawal)
  - asynchronous send/receive system
  - each process has an “account” with **money** ≥ 0
  - processes can send money at any time to anyone
    - send message with value, subtract value from **money**
    - add value received in messages to **money**
  - add “dummy” transfers (heartbeat msgs)
  - each process should output local balance
    - triggered by input action some process(es)
      - processes can awaken other processes that didn't receive input
    - sum of outputs should be equal to total money in system
      - well-defined because there are no deposits/withdrawals

Applications of logical time

- Assume logical-time algorithm
  - each process knows logical time for each of its events
- Use algorithm that assumes agreed-upon logical time t
  - each process determines value of **money** at logical time t
    - after all events with **ltime** ≤ t and before all events with **ltime** > t
  - for each incoming channel, determine amount in transit at time t
    - in messages sent at **ltime** ≤ t and received at **ltime** > t
    - count from when local clock > t, stop when msg timestamp > t
- What if local clock > t when node wakes up?
  - keep logs
  - try with different values of t

Applications of logical time

- Global snapshot
  - generalization of banking system example
  - given arbitrary algorithm on asynchronous send/receive system
  - want instantaneous global snapshot of system state
    - some “time” after a “triggering” action (typically an input)
  - must not stop entire system
  - useful for debugging, system backups, detecting termination

Applications of logical time

- Replicated state machines
  - important use of logical time: focal point of Lamport's paper
  - make distributed system simulate any centralized state machine
    - no fault-tolerance

Replicated state machines

- Centralized state machine
  - V: set of possible states
  - v_0: initial state
  - invs: set of possible invocations
  - resps: set of possible responses
  - trans: transition function: invs × V → resps × V
- Users of distributed system submit invocations, get responses in well-formed manner (blocking invocations)
  - want system to look like "atomic" variable (Chapter 13)
    - could weaken requirement to "sequential consistency"
  - no fault-tolerance

Replicated state machines

- Assume broadcast network
- Each process maintains
  - X: copy of simulated variable
  - inv-buffer: invocations it has heard about and their timestamps
    - timestamp based on logical time of bcast event
  - known-time: vector of “latest” logical times for each process
    - for itself: logical time of last event
    - for others: logical time of latest bcast event received
- Perform invocation π in inv-buffer when
  - π has smallest timestamp of any invocation in its inv-buffer, and
  - known-time(j) $\geq$ timestamp(π) for all j
  - respond if π was invoked locally

Replicated state machines

- Correctness
  - Liveness (termination)
    - requires unbounded logical time at each process...
    - and for other processes to know about it
  - Safety (looks like centralized system)
    - each process applies operations in the same (logical time) order
    - “serialize” when all processes have reached logical time of bcast
      - this is called the “serialization point” (or “linearization point”)
      - why is this in the operation's “interval”
    - requires FIFO channels to make sure that no invocations are “late”

Replicated state machines

- Special handling for “reads”
  - don't bcast: just perform locally
  - atomicity?
  - sequential consistency

Vector timestamps

- Logical time imposes a **total** order
  - this orders events that don't need to be ordered
- Weak logical time
  - same properties 1-4 as before, but
  - logical times are only *partially ordered*
- Vector timestamps
  - weak logical time
  - logical times ordered if and only if events are causally ordered
  - each process maintains “known time” of every process
    - send entire vector with each msg

Next lecture

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