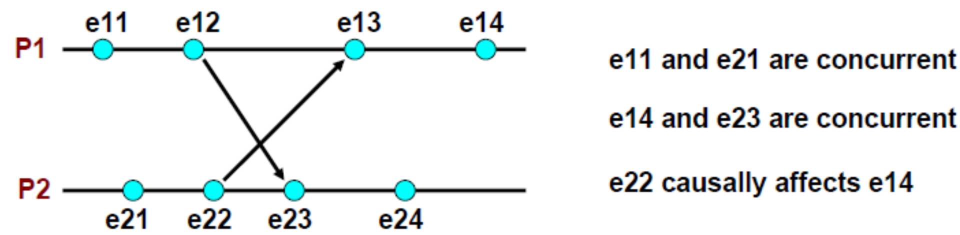
Lecture 7: Logical Time

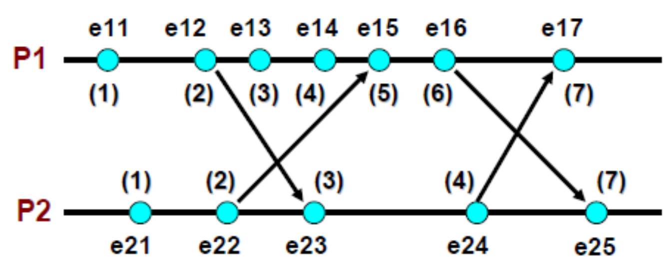
1. Question from reviews
    a. In protocol, is in-order delivery assumption reasonable?
        i. TCP provides it …
    b. In protocol, need all participants to be present
        i. Is this a reasonable assumption?
        ii. Need separate protocol to handle membership …
        iii. Will not scale
    c. How accurate can you get now?
        i. GPS, if machine can see satellites, provides nanosecond accuracy
        ii. NTP with GPS keeps other machines to 50 microseconds
        iii. NTP over lan ~ 1ms
        iv. NTP over WAN ~10ms
2. Key problem: how do you keep track of the order of events.
    a. Examples: did a file get deleted before or after I ran that program?
    b. Did this computers crash after I sent it a message?
    c. QUESTION: Why is this a problem?
        i. Clocks may be different on different machines
            1. E.g. processors in a multiprocessor system
            2. Machines in a cluster
        ii. QUESTION: How different do they have to be?
            1. More than the minimum time to send a message (1 ms), which is not much
        iii. Relativity: given different computers executing simultaneously and sending messages asynchronously, how can you tell?
    d. QUESTION: what do we really care about?
        i. If one thing happened at time X, and another at time X+delta, and they never communicate, does it matter?
        ii. Focus on "happens before" relationship
        iii. Don't need real clocks for many uses; since we are more interested in the **order of events** then in when the actually happened
    e. Examples:
        i. What kind of clock is good for security logs?
            1. Wall clock – want to correlate with human-scale events
            2. Absolute time – coordinate with outside world

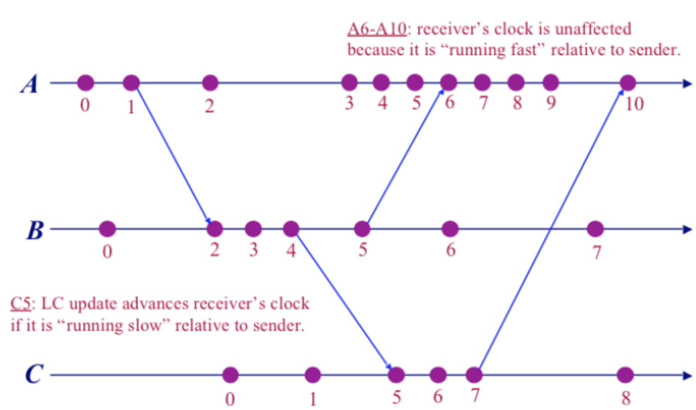
        ii. What kind of clock is good for figuring out which machines communicated and when?
            1. Logical clock: want to be able to order the communication from different machines (relative order)
    f. QUESTION: Is there an application to computer games?
        i. E.g. in a distributed environment, you can tell where another player is logically?
3. CONTEXT FOR SOLUTION
    a. General approach of theoretical papers: strip out all practical concerns not relevant to the problem, as they can be layered on afterwards if you get the basics right
    b. Example: ignore message loss, reordering on a link
        i. Easy to solve with TCP/IP
    c. Example: Ignore process/link failure
        i. Hard to solve, but need a separate protocol and this system works fine between times
    d. QUESTION: Why?
        i. Addressing all these concerns is orthogonal to the problem in many cases, clutters paper
        ii. Note: real clocks and message delay are relevant, so they are incldued
4. Happens before
    a. Intutive idea:
        i. Events in a single process are ordered (they are sequential)
        ii. A message send always precedes the receipt of that message (no speculation!)
    b. For two events a, b, a happens before b (a --→ b) if:
        i. A and b are events in the same process and a occurred before b, or
        ii. A is a send event of a message m and b is the corresponding receive event at the destination process, or
        iii. A -→ c and c-→b for some event c (transitive)
    c. Indicates causal relationship; a can affect b
5. Concurrent events:
    a. Not a-→b and not b-→ a

b. 

c. Space time diagrams: time moves left, space is vertical (rotated from paper)
d. Note: this is a partial order
    i. Not all events are ordered, some are before others (or after), but some are not.
    ii. QUESTION: in a distributed system, do you need a complete order or a partial order?

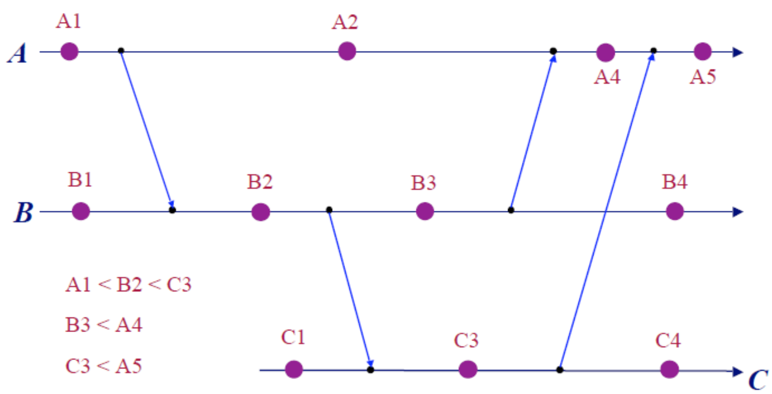
6. Logical clock: any counter that assigns times to events such that
    a. Clock condition: A -→B implies C(a) < C(b)
7. Lamport Logical Clocks
    a. Each process Pi maintains a register (counter) C
    b. Each event a in Pi is timestamped Ci(a), the value of C when a occurred
    c. IR1: Ci is incremented by 1 for each event in Pi
    d. IR2: If a is the send of a message m from process Pi to Pj, then on receive of m:
        i. Cj = max(Cj, Ci(a)+1)

e. 

f. 

DRAW TICK LINES

(connect zeroes, 1s, etc0



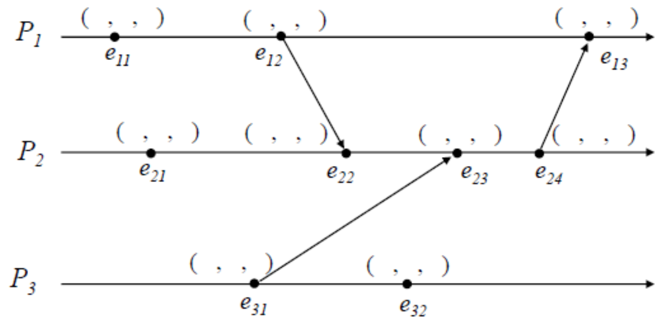
g.

8. Notes on logical clocks:
   a. It provides the guarantee that a -→ b implies C(a) < C(b)
   b. But, C(a) < C(b) does not imply a -→ b: see events e24 and e15 above
   c. C(a) == C(b) implies a and b are concurrent, but not vice versa (see e24, e14)
9. IN LOOKING AT TICK LINES:
   a. Must be line between two concurrent events
   b. Must be line between send and receipt of a message
10. QUESTION: What happens with failures? How does that affect ordering
    a. Tood Frederick comment …
11. Total order
    a. What if you need to agree on a total order for events?
    b. Use logical clocks and break ties deterministically: using process ID or node ID as a tie breaker
    c. QUESTION: is this really a total order?
       i. Real thing: an agreed upon order consistent with reality for happens-before
    d. QUESTION: What happens with failures?
12. QUESTION: Is this prone to errors? Sandeep …

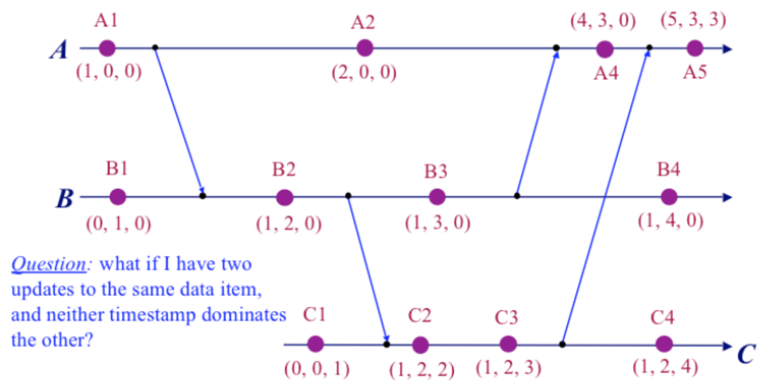
13. Use of logical clocks
    a. Suppose everybody broadcasts updates
    b. How do you impose a fixed order on updates?
    c. Do them in logical time order (assuming you wait forever…)
14. BIG QUESTION:
    a. How useful is this?
       i. When you care about order?
       ii. When you don't have synchronized time

            1. Sensors
            2. Loosely coupled machines
        iii. When you cannot afford a common time base
            1. Multiprocessors

15. Physical clock extensions
    a. Similar rule, but advance time according to clock received + minimum possible delay
    b. Need clock to be monotonic increasing
    c. Is the basis for NTP – send multiple messages to learn the minimum delay in each direction, use that to sync clocks to bounds tighter than delay
16. Vector clocks (also used as Version Vectors)
    a. Extension of logical clocks to capture more information
    b. Suppose A sends to B, D at time 2 (A changes object, sends it out)
        i. Time of B is 3
        ii. Time of D is 3
        iii. D then sends to B
            1. At B: has D seen A's message yet? Does the copy of the object from D include A's change?
            2. Cannot answer with logical clocks
                a. C(D send) > C(A send) does not imply D send logically occurs after A sends
    c. Solution: "vector clocks"
        i. Keep one logical clock **per process**, only incremented with local events
        ii. Maintain a local **vector clock** tracking received timestamps
        iii. Transmit all logical clock values you have seen
        iv. Set local vector clock to pairwise max(received vector, local vector)
        v. So:
            1. Ci[i] = Pi's own logical clock
            2. Ci[j] = Pi's best guess of logical time at Pj
                a. Or: latest thing that Pj did that Pi knows about directly or indirectly
        vi. Implementation rules:
            1. Events A and B in the same process: Ci[i] for a = Ci[i] for b + delta
            2. Send vector clock Tm on all messages M
            3. If A is sending and B is receiving of a message M from Pi to Pj:
                a. For all K, Cj[k] = max(Cj[k], Tm[k]
        vii. Example:



viii.



ix.

d. Rules for comparison:

i.
- Vector timestamps can be compared in the obvious way:
  - $t^a = t^b$ iff $\forall i,\ t^a[i] = t^b[i]$
  - $t^a \neq t^b$ iff $\exists i,\ t^a[i] \neq t^b[i]$
  - $t^a \leq t^b$ iff $\forall i,\ t^a[i] \leq t^b[i]$
  - $t^a < t^b$ iff $(\ t^a \leq t^b \ \wedge\ t^a \neq t^b\ )$
- Impoortant observation:
  - $\forall i, \forall j\ : C_i[i] \ \geq\ C_j[i]$

ii. So:
   1. Equal if all elements equal
   2. Not equal if at least one element not equal
   3. Ta <= Tb if all elements less or equal
   4. Ta < Tb if Ta <= Tb and Ta != Tb
      a. Means must be at least one element where Ta[k] < Tb[k]

iii. Causally related events with vector clocks:
   1. A-→B if and only if Ta < Tb

iv. Concurrent with vector clocks:

            1. Ta !< Tb and Tb !< Ta
            2. Consider past example: (A changes an object, sends it out)
                a. Suppose A sends to B, D at time 2
                    i. Time of B is 3
                    ii. Time of D is 3
                    iii. D then sends to B
                        1. At B: has D seen A's message yet?
                        2. Cannot answer with logical clocks
                            a. C(D send) > C(A send) does not imply D send logically occurs after A sends
                b. A (1,0,0) sends to B and D
                c. B receives at (0,3,0), sets clock to (1,3,0)
                d. D receives at (0,0,2), sets clock to (1,0,3)
                e. D sends to B at (1,0,4)
                f. B receives when clock is (1,4,0)
                    i. B knows that D has received A's message, because it has a 1 for A's clock
    e. Issues with vector clocks
        i. How big are vectors?
            1. Same size as the number of machines
        ii. What if the set of machines changes? Can you get rid of elements
            1. Only if you are sure it will never come back
        iii. When used?
            1. Good for replication (multiple copies of an object)
                a. Can modify at multiple points
                b. Can exchange updates pairwise
                c. Want to know if the other side saw an update you saw

17. Replicated state machine: Using logical clocks:
    a. Real problem: want a set of nodes to see same set of state transitions
        i. E.g. lock requests, acquires, releases.
    b. Problem:
        i. Want to have a group of nodes perform the same set of actions on a set of messages
        ii. General approach: each node implements a state machine
            1. Has local state
            2. Receives messages causing it to update state, send reply message
            3. In some cases, must receive messages in same order at every node

        4. Or, states must be commutative (can receive out of order without changing outcome)
    iii. For example: a distribute service storing your bank balance
        1. Send messages to deposit/withdraw to multiple copies, want outcomes to be the same
    iv. For example: decide who gets to modify a shared object (e.g. access shared storage)
        1. Send request to access to all nodes
        2. All nodes agree on an order of who gets to access next
        3. When it is your turn, do the access
        4. When done, send message to release access
c. How it works for mutual exclusion:
    i. Rules we want to implement:
        1. A process granted the resource must release it before anyone else can access it (safety)
        2. Grants of the resource are made in the order the requests are made
        3. If every grant is eventually release, then every request eventually granted (liveness)
    ii. What if we use a central scheduler? (assuming asynchronous messages)
        1. P0 has resource
        2. P1 sends a message to P1 requesting resource, then P2
        3. P2 receives P1's message, then sends a request to P0 asking for resource
        4. P0 receives P2's request before P1s (violation condition 2)
    iii. Assume:
        1. P0 starts with resource
        2. FIFO channels
        3. Eventual delivery (no failures)
    iv. Solution:
        1. Each process maintains a local **request queue** initialized to T0P0 (because P0 requests resource at time T0)
        2. To request the resource, process Pi sends a **RequestResource** message Tm:Pi to all other processes and places it in its own request queue
        3. When process Pj receives a request resource message, it places it in its request queue and sends a (timestamped) ack message back to Pi

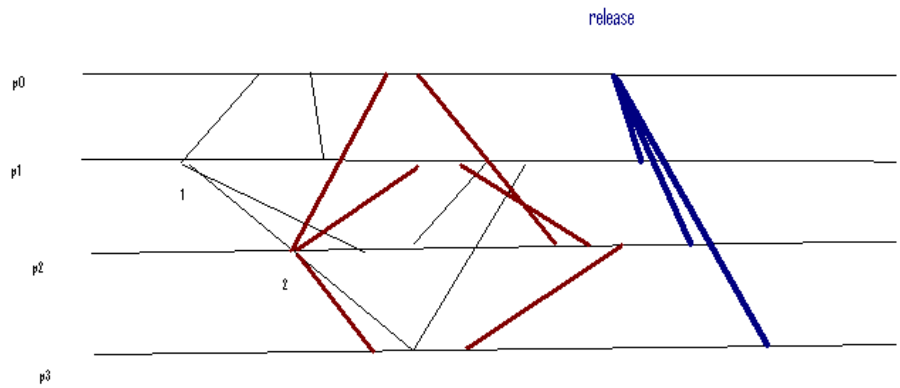
4. To release a resource, Pi remove the **RequestResource** message for Pi from its own queue and sends a **Tm:Pi Release Resource** message to all other processes (old Tm:Pi)
5. When process Pj receives a release message, it removes Tm:Pi, it removes any Tm:Pi request resource message from its queue
    a. Note: this must be after the request and after the ack
6. Process Pi is granted the resource when:
    a. There is a Tm:Pi **RequestResource** message in its queue when Tm < any other Tm (assuming a total order for messages)
    b. Pi has received a message from every other process with a time > Tm

v. Why works?
1. Condition b in part 6 above (Pi has received messages) ensures that Pi would have heard about any other request from any other process with a timestamp < Tm
2. Messages not deleted until granter sends a release message, so it will be in everyone's queue
3. Overall, don't take resource until everyone else ACKs and you know you are the least. On release resource, as soon as you get a release, you can go next, because you know everybody else agrees you will go next

vi. QUESTION: What happens if there is a failure (message lost, time out etc)?
1. Need to retry on a link-to-link basis

vii. NOTE: relies on common knowledge
1. When you get the acks from everyone else, a process has common knowledge that everyone knows of its request, and they know that Pi knows of their requests when they see the ack

viii. Example:
1. For processes: P0, P1, P2, P3
2. P1, P2 send "request messages", P1 at local time 1, P2 at local time 2
3. P0-P3 put P1:1 and P2:2 in their queue and ack
4. P0 sends release message
5. P1 takes over. When done, sends release
6. P2 takes over



7.

18. Benefits of state machine approach

   a. **Everybody decides on right thing to do locally, knows everybody else will make the same decision (common knowledge)**
   b. If everybody has the same initial state (e.g. lock release at low time) and sees the same sequence of messages in the same order, they will compute the same result in a distributed fashion
      i. Basis for lots of mechanisms – replication

19. Note: Given protocol pretty unrealistic – it really is an example of how it could work

   a. But basics of protocol are used – e.g. chubby lock servers use similar replicated state machines

20.