COS 316 Precept #8

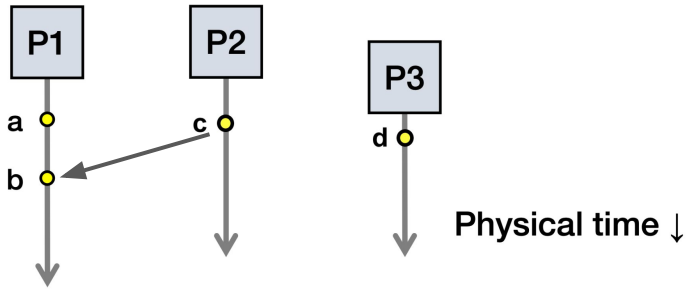
Concurrency and Clocks

Staying Synchronized

- On a single machine, its wall clock is sufficient
- Matches intuition
- In a distributed setting, each machine's clock may differ from the other machines
  - There is no longer an objective time
- Need some other mechanism to order events

- A single machine has a single clock
- When an event occurs, that machine can check its clock and apply a timestamp to that event
- And the clock will function as an objective source of time
- If you want to know the order two events happened in, check their timestamps!
- But when we introduce concurrency and distributed systems, that intuition falls apart
- In a distributed setting, there's a lack of inherent synchrony
- The clock on one machine may be different from the clock on another machine
- All kinds of strange things can happen, so the clocks of different machines can't be relied upon

Causality and Happens-Before



- Let's talk about causal relationships
- For two events, we assume a causal relationship between them if one event could've affected the other
- We say that the first event "happens before" the second event
- For a single process, we assume that events happen before all events that occur afterward in that process
- But what if two events occur on *different* processes?
- If there's no communication between those processes, then the events on one process could not have affected the other process
- We call these events *concurrent*
- Looking at the figure on the slide: There can be no causal relationship between C and D because those processes never communicate
- There IS a causal relationship between C and B because B is the receipt of a message and C is the sending of that message.
- And there is another causal relationship between A and B because A has a causal relationship with all events that happen after it in the same process
- For pairs of events where there is a causal relationship between them, we want to be able to communicate that, because in the total order of events, they need to appear in the correct order

Lamport Clocks

- Provide a way to totally order events (when used with some tie-breaking mechanism)
- LC(A) < LC(B) => B -/-> A

Q: a → b => LC(a) < LC(b)

Q: LC(a) < LC(b) => b -/-> a (a → b or a || b)

Q: a || b => LC(a) < LC(b) OR LC(a) > LC(b)

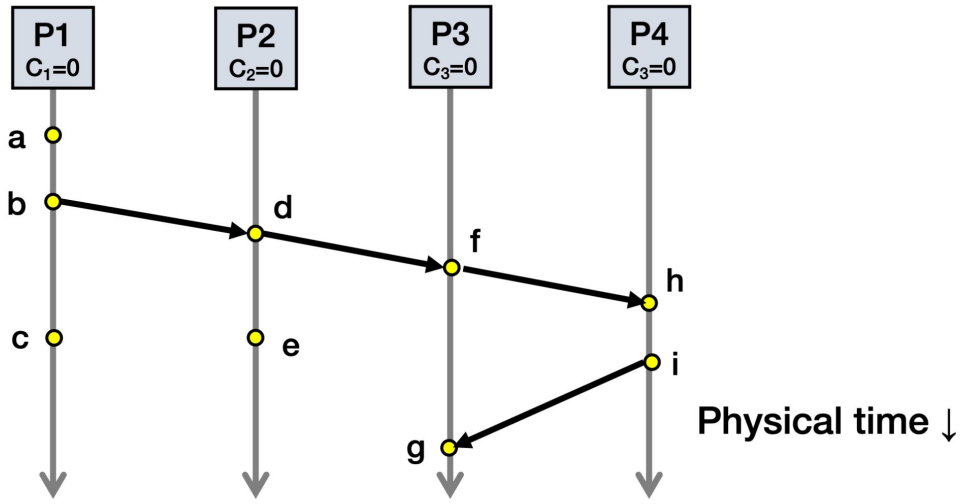
- In lecture, you were introduced to lamport clocks
- Lamport clocks provide a way to totally order events in a distributed system, when used with some tie-breaking criterion
  - The actual criterion that's used isn't important. It just needs to be deterministic. The one that you learned in class is to use process ID. That's both arbitrary and completely fine.
- Each event that occurs will be assigned a timestamp that can be used to sort them
- After the tie-breaking criterion has been applied, there should be no ties: every event should be sortable by their lamport timestamp
- For any two events, A, and B, if A happens before B, A's lamport timestamp is guaranteed to be lower than B's lamport timestamp

Lamport Clock Algorithm

1. Before executing an event b, Ci = Ci + 1:
2. Set event time C(b) ← Ci
3. Send the local clock in the message m
4. On process Pj receiving a message m:
5. Set Cj and receive event time C(c) ← 1 + max{ Cj, C(m) }

- The algorithm for computing lamport clocks is simple
- Each process keeps a local clock
- Note that this is not an actual clock. Each process doesn't care about the actual time that things happen, it cares about the order of events
- There are two types of events that would trigger an update of a process's lamport clock
- The first is an event that the process is responsible for. This could be sending a message to another process or some entirely local event that doesn't involve communication.
- In this scenario, before the event is labeled with a timestamp, the process increments its lamport clock by 1 and applies this new clock value as the timestamp. If the event is the sending of a message to another process, the timestamp is included with the message
- The other type of event is the receipt of a message. This time, a comparison needs to be made. The receiving process will look at its own lamport clock and the timestamp that was included in the message it received
- It will take the max of these values and set its own clock to that value. It will then increment its new clock value before applying the timestamp to the "receive message" event.

Lamport Clock Example



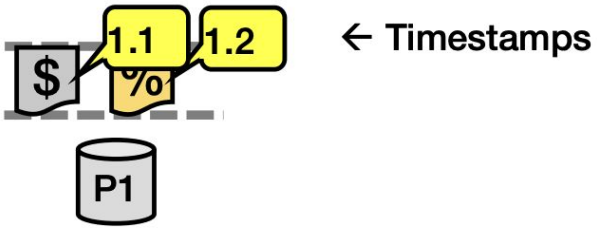
- Let's solidify our understanding of the algorithm with an example
- We want to assign a lamport timestamp to each of these events

- A = 1.1
- B = 2.1
- C = 3.1
- D = 3.2
- E = 4.2
- F = 4.3
- H = 5.4
- I = 6.4
- G = 7.3

Totally-Ordered Multicast

- Goal: All sites apply updates in (same) Lamport clock order
- Client sends update to one replica site j
  - Replica assigns it Lamport timestamp Cj . j
- Key idea: Place events into a sorted local queue
  - Sorted by increasing Lamport timestamps

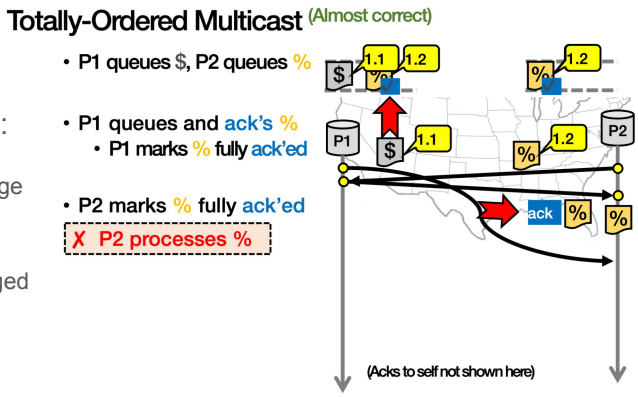
Example: P1's local queue:



- Totally-ordered multicast is an application of lamport clocks
- Its goal is to have a set of distributed processes apply updates in the same global order
- The protocol uses lamport clocks to define that global order
- Each process keeps a sorted queue of events that have yet to be executed
- The queue is always kept sorted by lamport timestamp
- And the events in every queue should execute in the order of their lamport timestamps

Totally-Ordered Multicast (Incorrect)

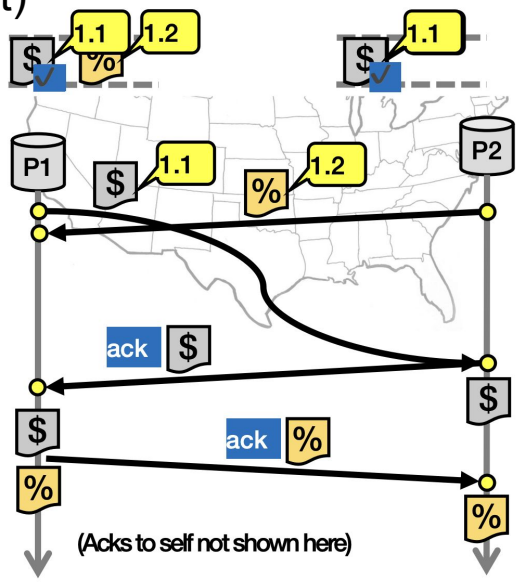
1. On receiving an update from client, broadcast to others (including yourself)
2. On receiving an update from replica:
   a. Add it to your local queue
   b. Broadcast an acknowledgement message to every replica (including yourself)
3. On receiving an acknowledgement:
   a. Mark corresponding update acknowledged in your queue
4. Remove and process updates everyone has ack'ed from head of queue



- This is the incorrect version of totally-ordered multicast that was shown in class
- It's because updates can be applied out of order!
- In the example above, P2 processes event % before it processes $, even though $ has a lower lamport clock

Totally-Ordered Multicast (Correct)

1. On receiving an update from client, broadcast to others (including yourself)
2. On receiving an update from replica:
   a. Add it to your local queue
   b. Broadcast an acknowledgement message to every replica (including yourself) only from the head of the queue
3. On receiving an acknowledgement:
   a. Mark corresponding update acknowledged in your queue
4. Remove and process updates everyone has ack'ed from head of queue



- This is the correct version of totally-ordered multicast that was shown in class
- Notice that step 2 has changed so that acknowledgements are only sent for the event at the head of the queue, which should have the smallest lamport timestamp
- The difference is that P1 doesn't send an ack immediately after receiving the % operation anymore since not everything in front of the % operation in P1's queue is ready to be executed (the $ operation)

Lamport Clock Weaknesses

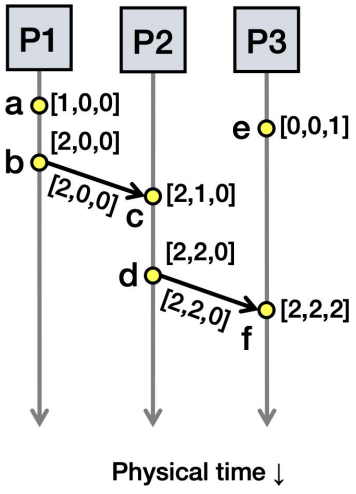
Q: a → b => LC(a) < LC(b)

Q: LC(a) < LC(b) => b -/-> a (a → b or a || b)

Q: a || b => LC(a) < LC(b) OR LC(a) > LC(b)

- Let's take a closer look at the guarantees provided by Lamport clocks
- If there is a causal relationship between two events, meaning A happens before B, then the lamport clock for A is guaranteed to be lower than that of B
- But what if only have the lamport clocks and not visibility into the ground truth of a situation?
- Given the lamport clocks for A and B, if LC(A) is lower than LC(B), we can conclude that B did NOT happen before A
- Which is equivalent to saying that A happened before B OR A and B are concurrent, which is ambiguous
- What if we need to know for sure whether two things are causally related?
- Then lamport clocks fall short!

Vector Clocks



- Vector clocks are an improvement over lamport clocks in that they allow us to differentiate between a causal relationship and concurrency
- With this mechanism, each process keeps a vector clock of length N, where each element of the vector corresponds to a process in the system
- Now, when an event occurs that a process is solely responsible for (some calculation or sending a message), it increments the element of the vector that corresponds to it
- Similar to before, when a message is sent, the ENTIRE vector clock is included in the message
- When a process receives a message, it iterates through the elements of the received clock and its own clock pairwise, taking the max of each element
- It then increments the element of its vector that corresponds to it

Comparing Vector Timestamps

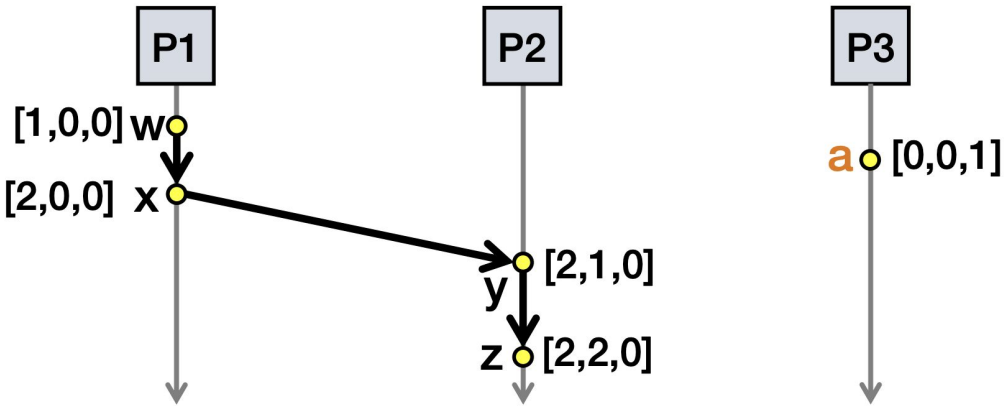
- **Rule for comparing vector timestamps:**
  - **$V(a) = V(b)$ when $a_k = b_k$ for all k**
  - **$V(a) < V(b)$ when $a_k \leq b_k$ for all k and $V(a) \neq V(b)$**

- **Concurrency:**
  - **$V(a) \parallel V(b)$ if $a_i < b_i$ and $a_j > b_j$, some i, j**

With vector clocks, there are now explicit cases for when there is a happens-before relationship between two events and when those two events are concurrent!

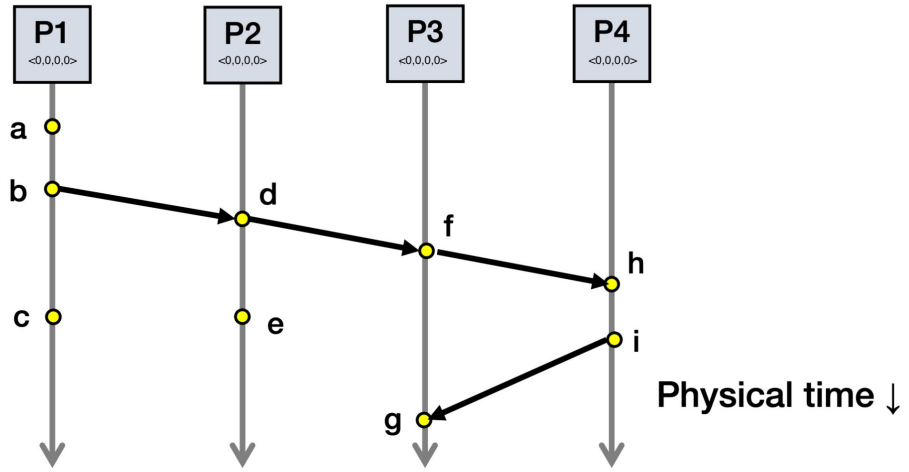
- This removes ambiguity!

Vector Clock Weaknesses



- A vector clock's weaknesses comes down to its size. Vector clocks are larger than lamport clocks and their size scales with the number of processes in the system
- For a large number of processes, vector clocks can get pretty long

Vector Clock Example



- A: <1, 0, 0, 0>
- B: <2, 0, 0, 0>
- C: <3, 0, 0, 0>
- D: <2, 1, 0, 0>
- E: <2, 2, 0, 0>
- F: <2, 1, 1, 0>
- G: <2, 1, 2, 2>
- H: <2, 1, 1, 1>
- I: <2, 1, 1, 2>