Clock and ordering

Yang Wang

Review

- Happened-before relation
- Consistent global state
- Chandy Lamport protocol

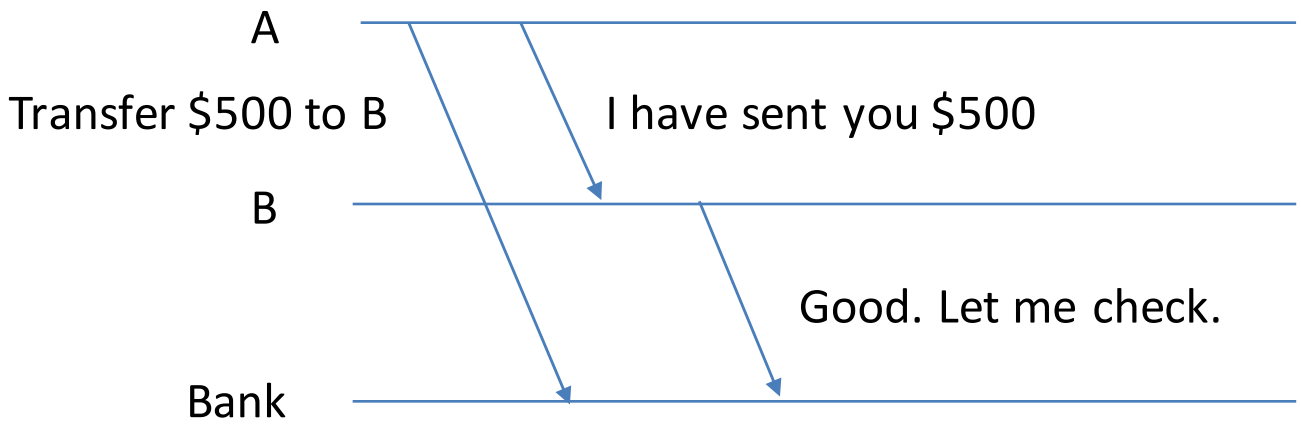
New problem

- Monitor node sometimes needs to observe other nodes' events continuously
- Distributed snapshot is a heavy protocol
  - Performing a snapshot for each event is unacceptable
- If there are only two nodes, TCP (FIFO channel) can solve the problem
  - But how about more nodes?

Message delivery

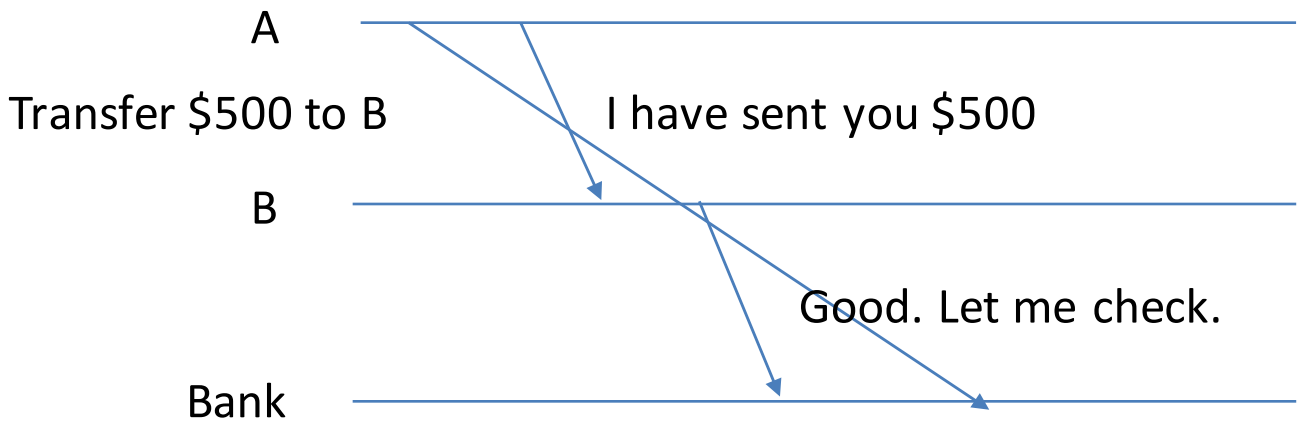
- Receive: a node gets a message from the network
- Delivery: the node actually processes the message
- Can a node deliver a message when receiving it, assuming each channel is FIFO?

Counter example



Normal case.

Counter example



Message delay can cause unexpected behavior. The bank can receive two events (transfer and check) in an inconsistent order.

Casual delivery

- $send_i(m) \rightarrow send_j(m') \Rightarrow deliver_k(m) \rightarrow deliver_k(m')$

- TCP can guarantee casual delivery when i=j, but cannot guarantee it when i≠j.

Start with stronger assumptions

- Assume all nodes have access to a global clock
- Assume synchronous network:
  - Network delays are bounded (Δ).
- Can you think about a solution?

Start with stronger assumptions

- Attach a timestamp ts to each message
- Deliver messages in the order of ts
- Is this complete?

Start with stronger assumptions

- Attach a timestamp ts to each message
- Deliver messages in the order of ts
- When a node's clock reaches t, it can deliver all messages with ts <= t – Δ.

An ideal clock

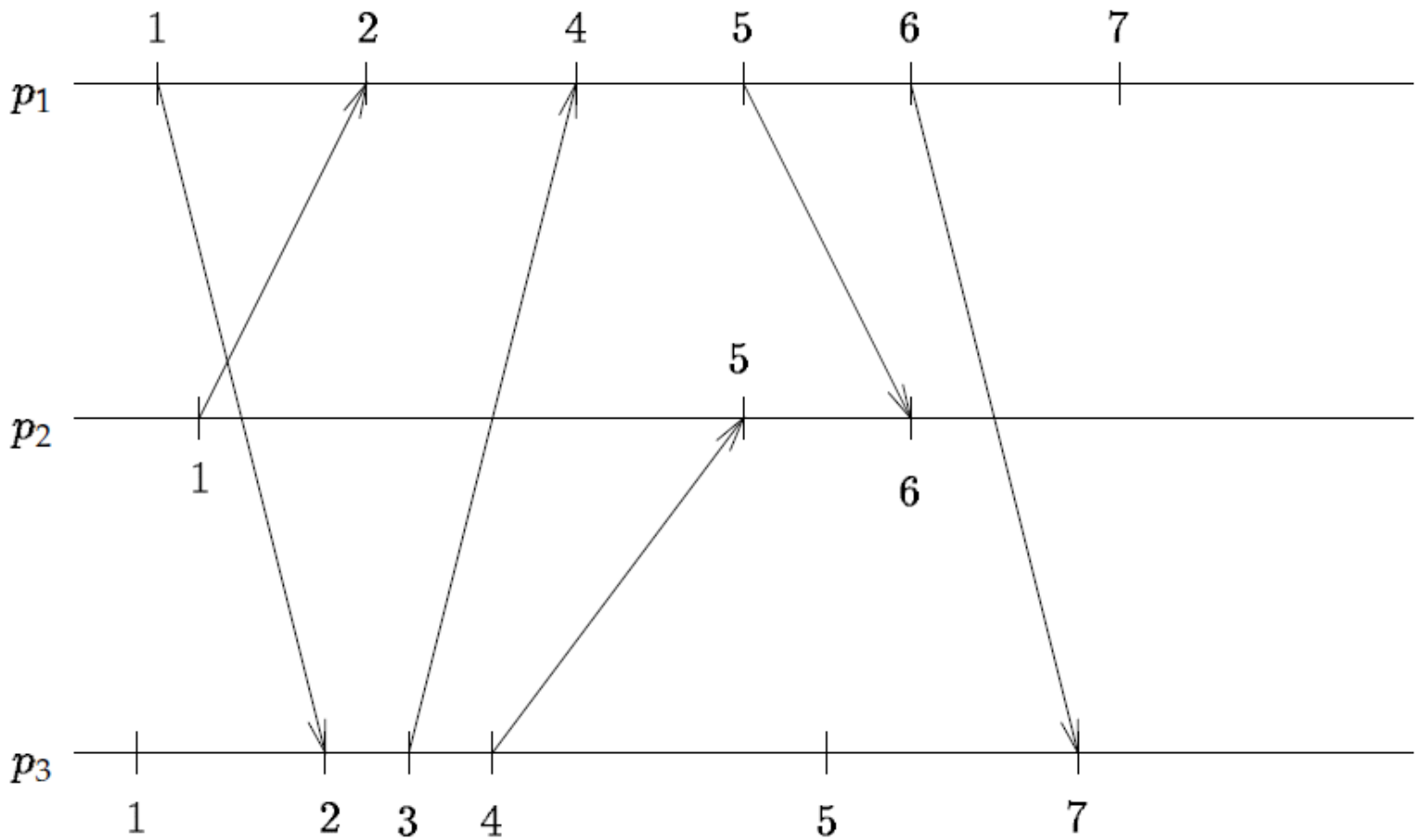
- Clock condition (for correctness): $e \rightarrow e' \Rightarrow clock(e) < clock(e')$
  - Is the opposite required?
- Gap detection (for liveness): given two events e and e' with $clock(e) < clock(e')$, determine whether e'' exists such that $clock(e) < clock(e'') < clock(e')$
  - Even real time clock cannot achieve this. Need additional help.

Logic clock

- Also called Lamport clock
- Each node maintains a local integer called LC

$$LC(e_i) := \begin{cases} LC + 1 & \text{if } e_i \text{ is an internal or send event} \\ \max\{LC, TS(m)\} + 1 & \text{if } e_i = receive(m) \end{cases}$$

Logic clock



Logic clock

- Also called Lamport clock
- Each node maintains a local integer called LC

$$LC(e_i) := \begin{cases} LC + 1 & \text{if } e_i \text{ is an internal or send event} \\ \max\{LC, TS(m)\} + 1 & \text{if } e_i = receive(m) \end{cases}$$

- Does it satisfy clock condition?
- Can it provide gap detection?

Logic clock

- Also called Lamport clock
- Each node maintains a local integer called LC

$$LC(e_i) := \begin{cases} LC + 1 & \text{if } e_i \text{ is an internal or send event} \\ \max\{LC, TS(m)\} + 1 & \text{if } e_i = receive(m) \end{cases}$$

- Does it satisfy clock condition? Yes.
- Can it provide gap detection? No.
  - Could you think about a solution?

Logic clock

- Also called Lamport clock
- Each node maintains a local integer called LC

$$LC(e_i) := \begin{cases} LC + 1 & \text{if } e_i \text{ is an internal or send event} \\ \max\{LC, TS(m)\} + 1 & \text{if } e_i = receive(m) \end{cases}$$

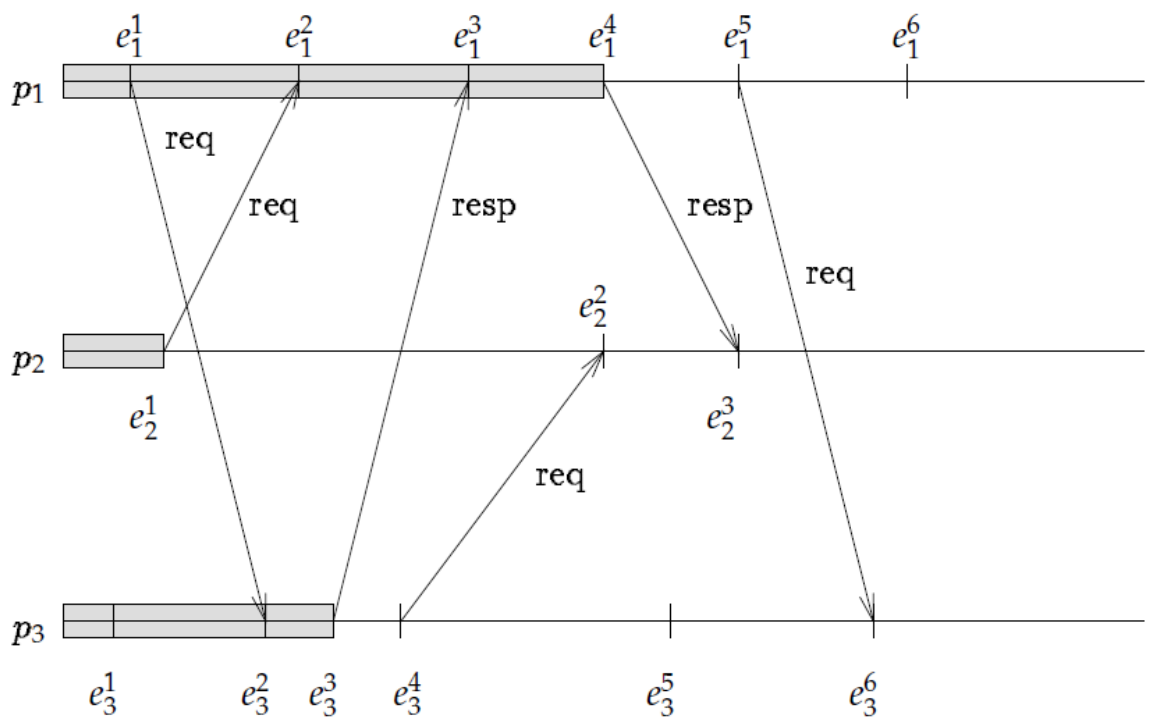
- Does it satisfy clock condition? Yes.
- Can it provide gap detection? No.
  - A node can deliver e with ts if it receives a message with a higher ts from each node.

Limitation of Logic clock

- Cannot differentiate concurrent events and related events
  - Clock condition: $e \rightarrow e' \Rightarrow clock(e) < clock(e')$. Implementation uses it in the opposite direction: $clock(e) < clock(e') \Rightarrow$ deliver e before e'
  - Can cause unnecessary delays in message delivery
- It's better to have a stronger property:
  - Strong clock condition: $e \rightarrow e' \equiv clock(e) < clock(e')$.

Vector clock

- The clock of an event e is the minimal consistent cut θ(e) that includes e



Vector clock

- The clock of an event e is the minimal consistent cut $\theta(e)$ that includes e
- Define clock comparison as set operation:
  - $e \rightarrow e' \equiv \theta(e) \subset \theta(e')$
  - e and e' are concurrent if $\theta(e) \not\subset \theta(e')$ and $\theta(e') \not\subset \theta(e)$

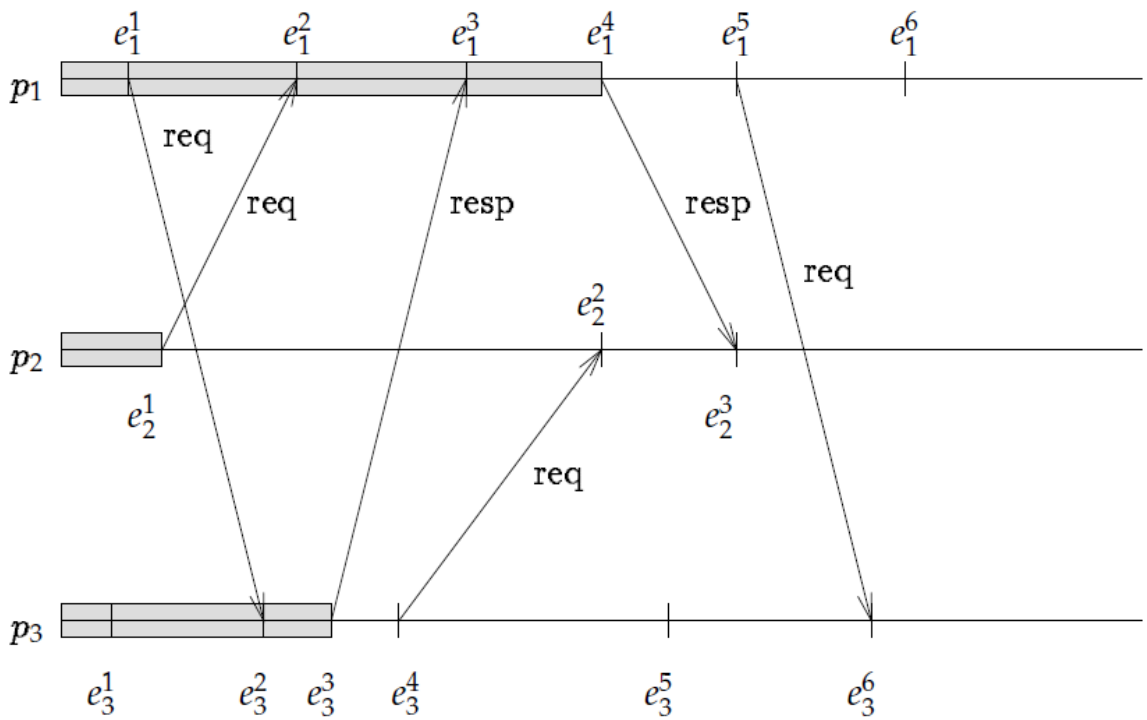
Vector clock

- A consistent cut can be uniquely identified by its frontier

- We can use a vector VC(e) to represent the frontier of the minimal consistent cut of e
  - VC(e)[i] = number of events on node i

Vector clock

- $VC(e^4_1) = [4, 1, 3]$

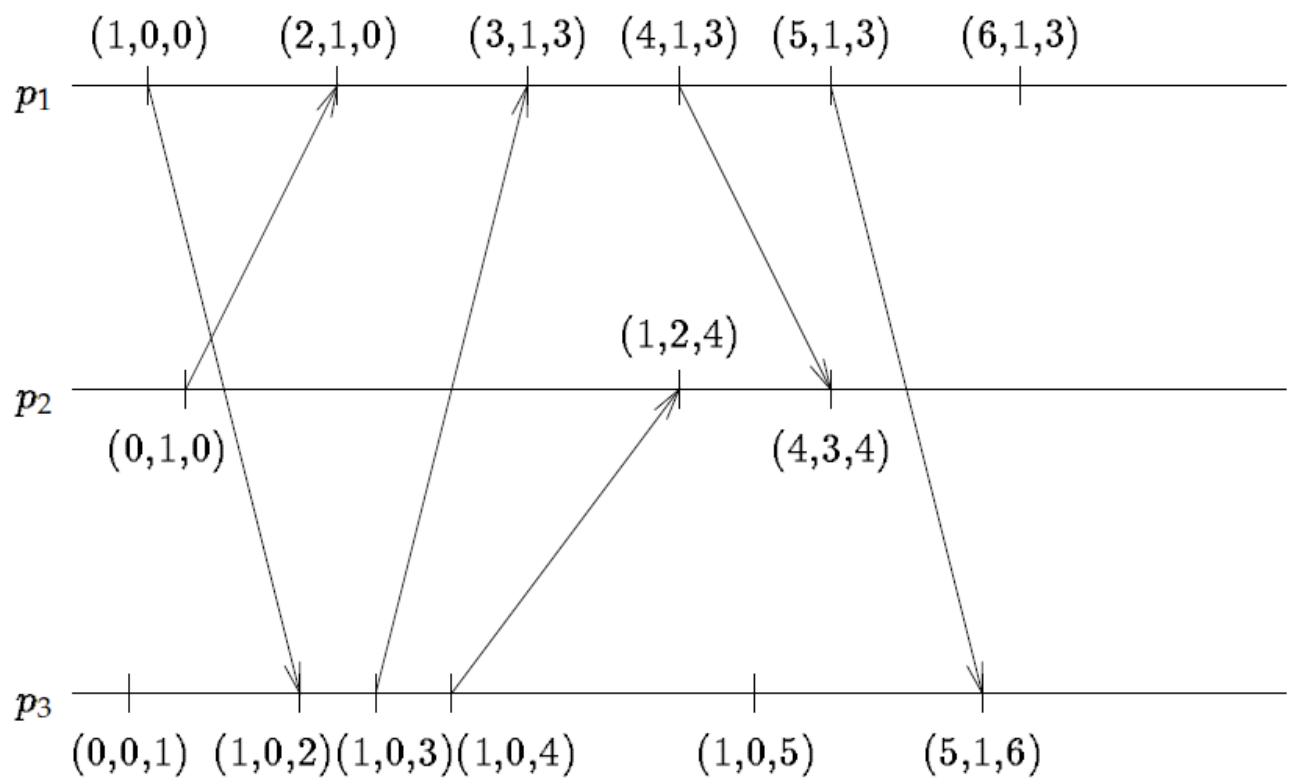


How to compare two vector clocks?

Vector clock

$$VC(e_i)[i] := VC[i] + 1 \quad \text{if } e_i \text{ is an internal or send event}$$

$$VC(e_i) := \max\{VC, TS(m)\} \quad \text{if } e_i = receive(m)$$
$$VC(e_i)[i] := VC[i] + 1$$



Properties of vector clock

- Property 1 (Strong Clock Condition)
  - $e \rightarrow e' \equiv VC(e) < VC(e')$
- Property 2 (Simple Strong Clock Condition)
  - $e_i \rightarrow e_j \equiv VC(e_i)[i] \leq VC(e_j)[i]$
  - Can you prove it?
- Property 3 (Concurrent)
  - $e_i || e_j \equiv (VC(e_i)[i] > VC(e_j)[i]) \wedge (VC(e_j)[j] > VC(e_i)[j])$

Properties of vector clock

- Properties 4 - 6: Read the paper. 4 and 5 are used to determine whether a cut is consistent.

- Property 7 (Weak Gap-Detection)
  - $VC(e_i)[k] < VC(e_j)[k] \Rightarrow \exists e_k$, s.t. $\neg(e_k \rightarrow e_i) \wedge (e_k \rightarrow e_j)$
  - When i=k, $\exists e_k$, s.t. $e_i \rightarrow e_k \rightarrow e_j$

Go back to the problem

- Assume processes send a notification message to the monitor for all of their events.
- When the monitor receives m from p_j, when can the monitor deliver m?

Go back to the problem

- Assume processes send a notification message to the monitor for all of their events.
- When the monitor receives m from p_j, when can the monitor deliver m?
  - When the monitor has received all messages happened before m.

Solution

- Any earlier message happened before m?
  - Any earlier message from p_j?
  - Any earlier message from p_k ($k \neq j$)?

Solution

- Any earlier message happened before m?
  - Any earlier message from p_j? – VC(m)[j]-1 from p_j has been delivered
  - Any earlier message from p_k ($k \neq j$)?

Solution

- Any earlier message happened before m?
  - Any earlier message from p_j? – VC(m)[j]-1 from p_j has been delivered
  - Any earlier message from p_k ($k \neq j$)? – Get help from the weak gap-detection property.

Solution

- Any earlier message happened before m?
  - Any earlier message from p_j? – VC(m)[j]-1 from p_j has been delivered
  - Any earlier message from p_k ($k \neq j$)? – Get help from the weak gap-detection property.
    - Maintain the last message delivered from each process
    - Suppose m' is the last message delivered from p_k, if VC(m')[k] < VC(m)[k], then there exists one.

Optimization

- Suppose m' is the last message delivered from p_k, if VC(m')[k] < VC(m)[k], then there exists one.
  - For a message m' from p_k, only VC(m')[k] is useful.
  - Just keep VC(m')[k] instead of VC(m')
- Monitor maintains an array D[1-n]
  - When it delivers m' from p_k, set D[k] = VC(m')[k]

Optimization

- Casual Delivery: Deliver message m from process p_j as soon as both of the following conditions are satisfied:
  - $D[j] = VC(m)[j]-1$
  - $D[k] \geq VC(m)[k], \forall k \neq j$

Example