Coordination

Slide set 3
Distributed Systems

Graduate Level

K. N. Toosi Institute of Technology
Dr. H. Khanmirza
h.khanmirza@kntu.ac.ir

Coordination

- Distributed processes need for coordination
  - Locking file, exclusive access to a resource, who must get the lock
  - Ordering events: Which process should send or receive messages in what order?

- Coordination
  - Goal is to manage the interactions and dependencies between activities

Coordination

- Synchronization
  - Process synchronization: one process waits for another to complete its operation
  - Data synchronization: ensure that two sets of data are the same

- From this perspective, one could state that coordination encapsulates synchronization.

Time & Clock Synchronization

Global Timing

- Consider a distributed build system with Make
  - Make checks times of *.c files and their corresponding *.o (object) files
  - If timestamp of *.o file is earlier than *.c make do recompile
  - Otherwise do nothing
    - time(input.c) > time(input.o) → recompile
    - time(input.c) <= time(input.o) → do nothing
  - If clock of developer systems are different it leads to mixture of object files from old and new sources!!

- Other scenarios
  - Financial Applications
  - Security Auditing
  - Collaborative Sensing

Physical Clocks

- Found in every computer
- "Timer" is better word

- It is a precisely machined quartz crystal oscillates in an electric field

- Associated with each crystal are two registers:
  - Counter and a holding register.
  - Each oscillation of crystal decrements the counter by one.
  - When counter → 0, an interrupt is generated and counter reloaded from holding register.
  - Each interrupt is called one **clock tick**.
  - It is possible to program a timer to generate an interrupt 60 times a second, or at any other desired frequency

Physical Clocks

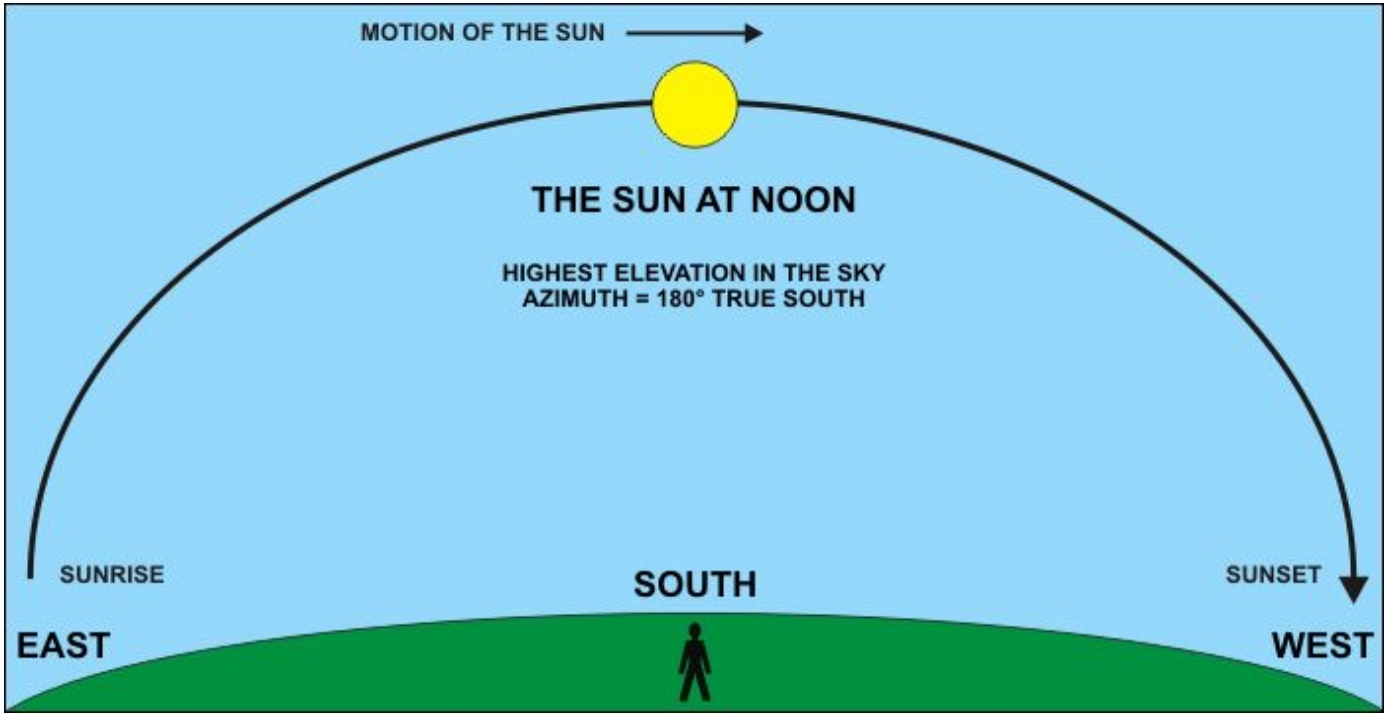
- In a multi-CPU system, each CPU has its own clock
  - Crystals are chosen very accurately, but there is always a very small difference in their running frequency
- Clock Skew
  - A system with n computers, all n crystals run at slightly different rates
  - This causes OS clocks gradually to get out of sync
- Facts
  - Having multiple clocks is desirable for efficiency and redundancy
  - In some real-time application the actual clock time is important, which needs to external clock sources

Measuring Time

- Mean solar seconds
- TAI
- UTC

Mean Solar Seconds

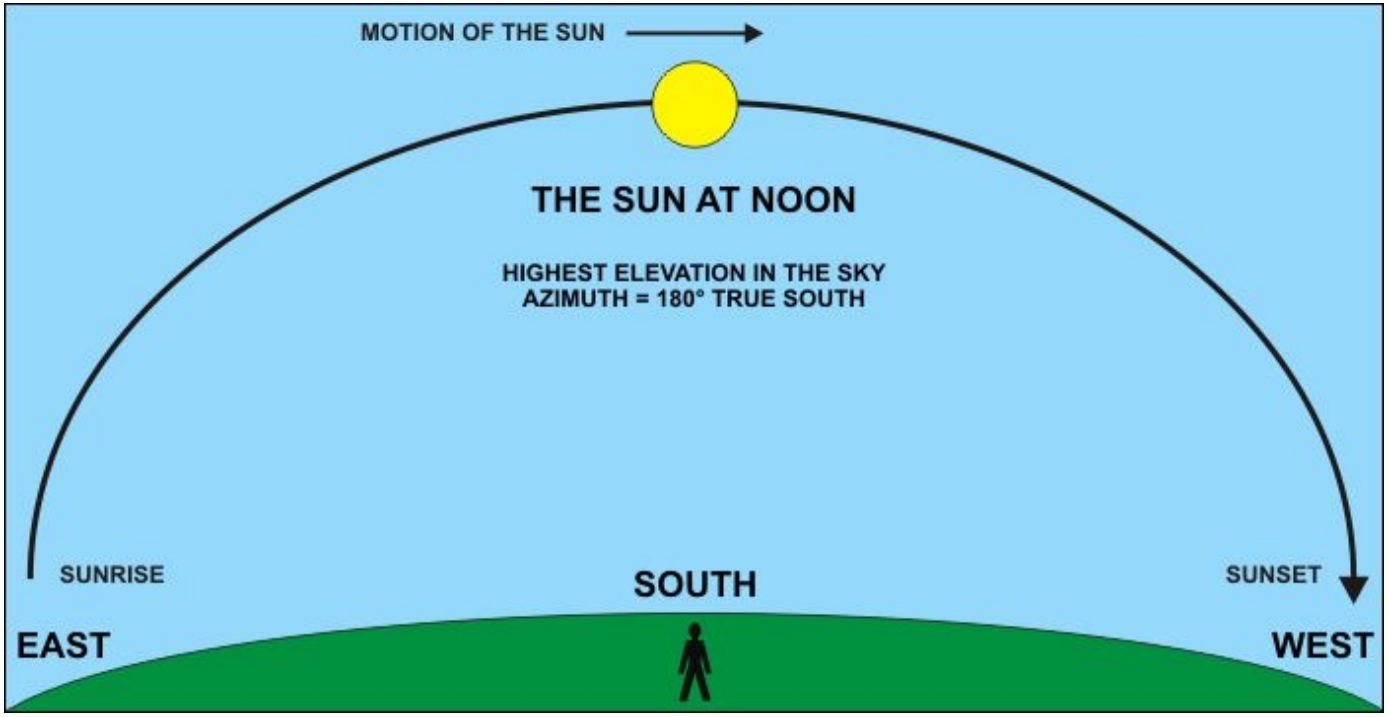
- Sun at its highest apparent point is called the transit of the sun.
- The interval between two consecutive transits is the solar day.
- Since there are 24 hours in a day, each containing 3600 seconds, the solar second is defined as exactly 1/86400th of a solar day.



Mean Solar Seconds

- In 1940s, it was established the period of the earth's rotation is not constant, Days may become longer or shorter!!

- Average of several days divided by 86,400, is called the mean solar second.



TAI

- International Atomic Time
- The second is the time for the cesium 133 atom to make exactly 9,192,631,770 transitions.
- Several Labs have such a clock
- Periodically they report how many times their clock has ticked
- Average no. of ticks is TAI which is the mean number of ticks since midnight on Jan. 1, 1958

UTC

- TAI is highly stable, but 86,400 TAI seconds is now about 3 msec less than a mean solar day
  - Over the years, noon time will be earlier and earlier
- Whenever the discrepancy between TAI and solar time grows to 800 msec, a leap second is added to TAI, This time system is known as UTC (Universal Coordinated Time)
- UTC is the basis of all modern civil timekeeping
  - It is distributed with ground stations pulsing to each other at the start of each second (acc. $\pm 40ms$)
  - With (commercial) satellites provide time with accuracy of 0.5ms

Clock Synchronization Algorithms

- Synchronization goals
  - Having a UTC receiver, keep all the machines synchronized to it
  - Having no UTC receivers, each machine keeps track of its own time, and the goal is to keep all the machines together as well as possible

Clock Synchronization Algorithms

- Suppose
  - $C_p(t)$: The value of software clock on machine p
  - t: UTC time
- Clock Synchronization algorithm with Precision π means
  - $\forall t, \forall p, q \quad \left|C_p(t) - C_q(t)\right| < \pi$
- Having an external reference point, like UTC, accuracy is bounded
  - $\forall t, \forall p, q \quad \left|C_p(t) - t\right| < \alpha$
- Set of clocks that are accurate within bound α, will be precise within bound $\pi = 2\alpha$

Clock Synchronization Algorithms

- Clock Drift
  - Hardware clocks are subject to various conditions like temperature
  - Frequency over the time is changed and they start showing different values for time

- Clock Drift Rate:
  - The difference per unit of time from a perfect reference clock
  - Ordinary quartz clocks drift by about 1 sec in 11-12 days. (10^{-6} secs/sec).
  - High precision quartz clocks drift rate is about 10^{-7} or 10^{-8} secs/sec

Clock Synchronization Algorithms

- Suppose
  - ρ: max clock drift rate
  - F: ideal constant oscillator frequency
  - $F(t)$: oscillator frequency at time t

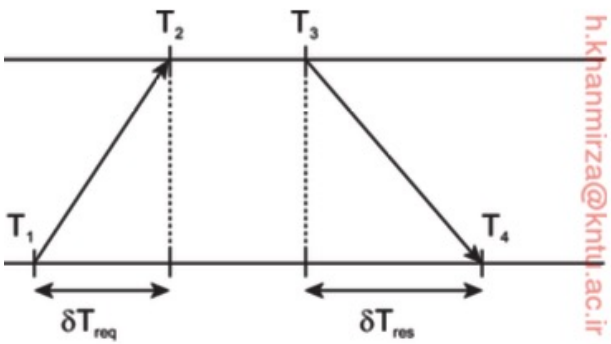
- Then: $(1-\rho) < \frac{F(t)}{F} < (1+\rho)$
  - We'd like ideal clock to be the same current clock: $F(t) = F$

Clock Synchronization Algorithms

- Software clock is the mean of $F(t)$
  - $C_p(t) = \frac{1}{F}\int_0^t F(t) \Rightarrow \frac{dC_p(t)}{dt} = \frac{F(t)}{F} \Rightarrow (1-\rho) < \frac{dC_p(t)}{dt} < (1+\rho)$
  - Two clocks may differ at time Δt as much as $\Delta t.2\rho$
    - $\left|\frac{dC_{p1}(t)}{dt} - \frac{dC_{p2}(t)}{dt}\right| < 2\rho$
  - To have precision of $\pi \rightarrow \Delta t = \frac{\pi}{2\rho}$
    - To have clocks with no more than π seconds difference, they must be synchronized at least every $\frac{\pi}{2\rho}$ seconds

Network Time Protocol (NTP)

- A common approach in many protocols is to let clients contact a time server.
- Time server reports
  - T_2:time of receiving reference message)
  - T_3: time of send the reply
- Client calculates
  - $\delta = \frac{(T_2-T_1)+(T_4-T_3)}{2}$
  - T_3 plus the average propagation delays should be equal to T_4 but it is not, then their subtract gives the offset
  - $\theta = T_3 + \delta - T_4 = \frac{(T_2-T_1)+(T_3-T_4)}{2}$

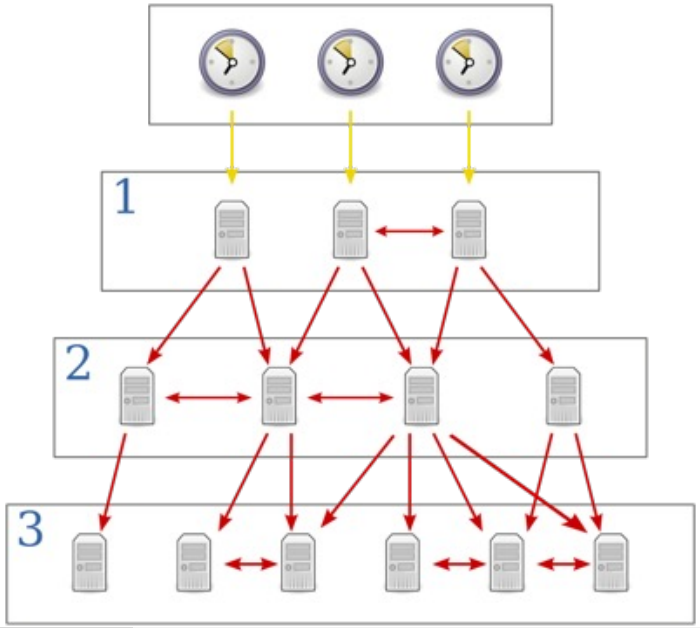


Network Time Protocol

- What means if $\theta < 0$
  - Means A's clock is faster!
  - Should A set its clock backward?
    - It is considered harmful
- In general changes to time is done gradually
  - If every hardware interrupt adds 10 ms to the time
  - For setting backward, system can add 9ms
  - For setting forward system can add 11 ms
- NTP
  - A and B probe each other
  - Minimum δ is used for calculating θ

Network Time Protocol

- What if A is more accurate than B. Pairwise synchronizing is foolish!
- NTP divides machines into strata levels
  - Machine with reference clock is in the level (stratum-1)
  - Lower stratum levels have better accuracy
- When A contacts B, A adjusts time if is in the higher stratum
  - $stratum(A) > stratum(B) \Rightarrow sychnronize$
- After synchronization, $stratum(A) > stratum(B) + 1$

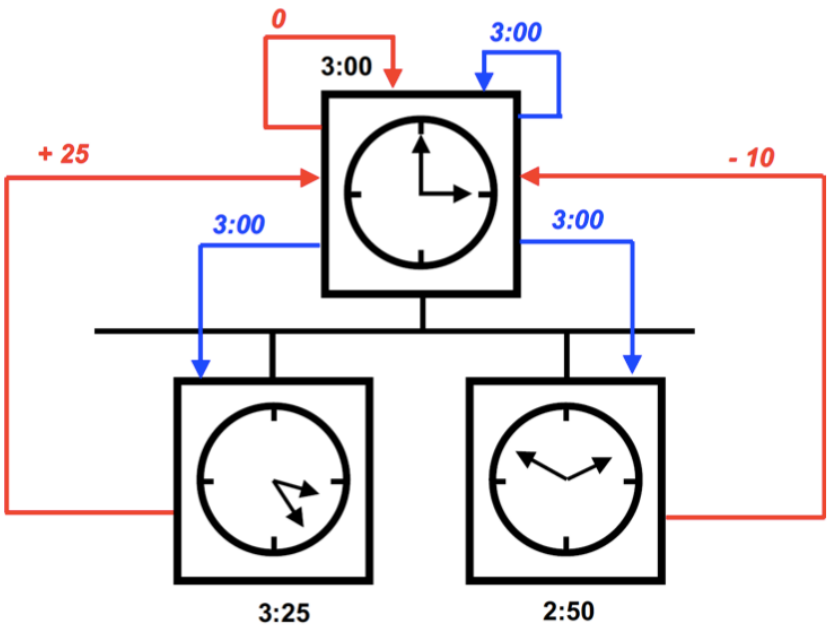


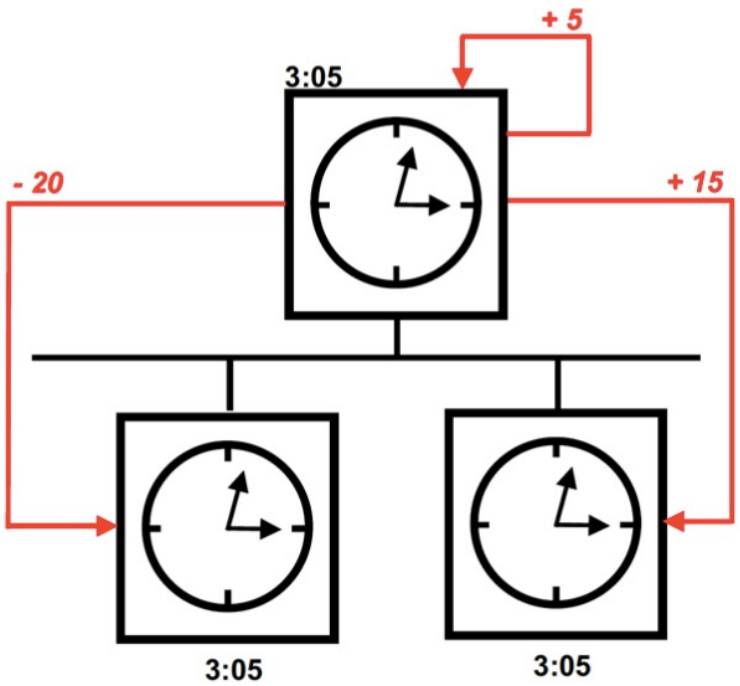
The Berkeley Algorithm

- Time Server (leader) is selected
- Server polls time of followers periodically
  - Time server is active
- Server observes RTT and estimates each followers' time
- Server computes an average time
  - Ignoring outliers
- Server sends followers to how much adjust their clock
  - Avoiding RTT negative effect

- This method is suitable for a system in which no machine has a UTC receiver. Time is manually set by an operator on time server
  - System doesn't need an exact time, only needs agreement

The Berkeley Algorithm





Google TrueTime Service

- Several Time Master machines
- Use different source of measuring time (atomic, UTC receiver, ⋯)
- Several time master per data center
- Time slave queries multiple time masters and get a collection of time sources with a high degree of mutual independence
  - With some algorithms outliers are removed
- Result is guaranteed time accuracy of 6ms!

Clock Synchronization Algorithms

Time Synchronization in WSNs

- Nodes are resource constrained
  - Energy efficiency of protocols is important
  - Limited processing power
  - Multi-hop routing is expensive in WSN
- Sensors often placed in harsh environments
  - Fluctuations in temperature, pressure, humidity!
- Wireless medium
  - Higher error rates, radio interference, …
- Mobility
  - Topology change, density changes
- Node failures

Reference Broadcast Synchronization (RBS)

- Sender-Receiver Synchronization

- Design principles
  - In WSNs propagation time to other nodes is roughly constant

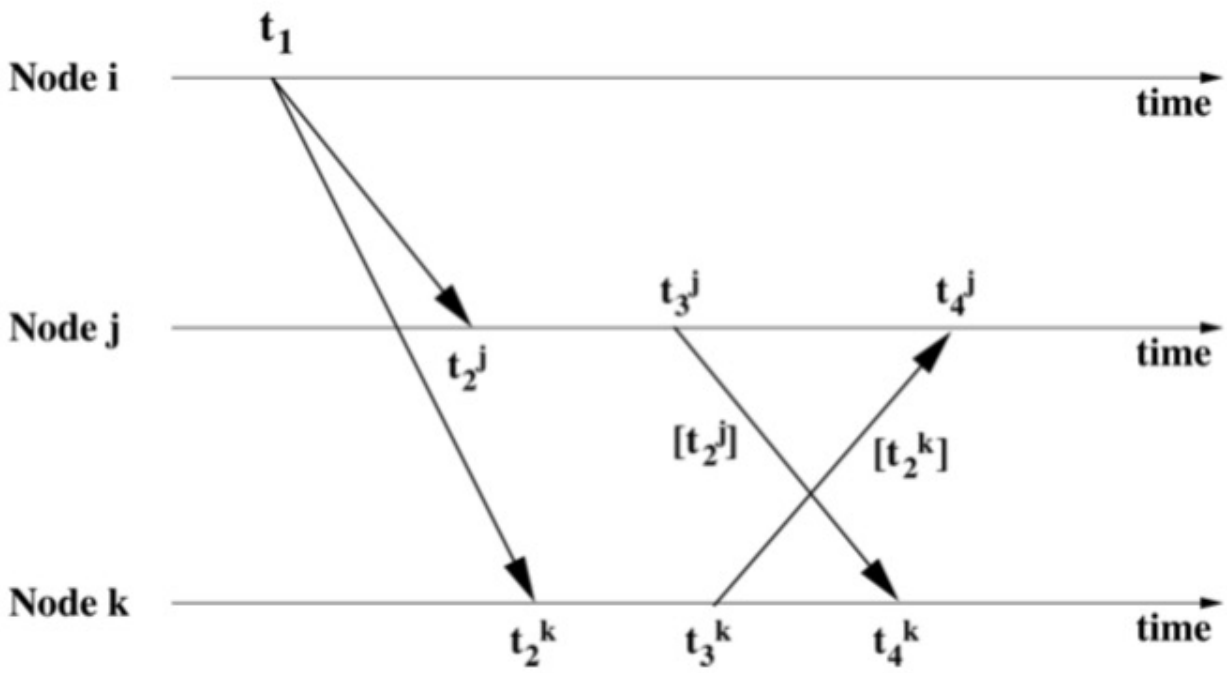
  - Time is measured from the moment a message leaves NIC, variable times are omitted:
    - Time spent to build a message
    - Time spent to access the network (at MAC layer)

Reference Broadcast Synchronization (RBS)

- A sender p broadcasts a reference message (m) with its timestamp
- Neighbor nodes record their local receive time $T_{p,m}$ of m, $p \in N(p)$
- Two nodes p and q exchange M messages and estimate their offset
- $Offset[p,q] = \frac{\sum_{k=1}^{M}(T_{p,k} - T_{q,k})}{M}$
- Average offset cannot solve clock drift problem
  - Use linear regression $Offset[p,q](t) = \alpha t + \beta$
  - Coefficients are computed from the pairs ($T_{p,k}$, $T_{q,k}$)

Receiver-Receiver Time Synchronization

- Uses constant delivery time principle



Time-synch Protocol (Tree-based protocols)

- Two phases
  - Level Discovery phase
    - One node starts a LEVEL_DISCOVERY message and announces itself as level 0
    - Receiver knows its level as message level + 1 and re-sends it to its neighbors
    - Receivers discard other messages
    - When root dies, leader election process starts
    - A hierarchical tree is built
  - Synchronization phase
    - Each node in level (i) synchronizes with level (i-1) (uses the approach introduced for NTP)
    - Nodes in level (i + 1) overhear level (i) synchronization, then after a random time they start synchronization with level (i)
- Error is propagated throughout the tree hierarchy

Logical Clocks

Logical Time and Logical Clocks

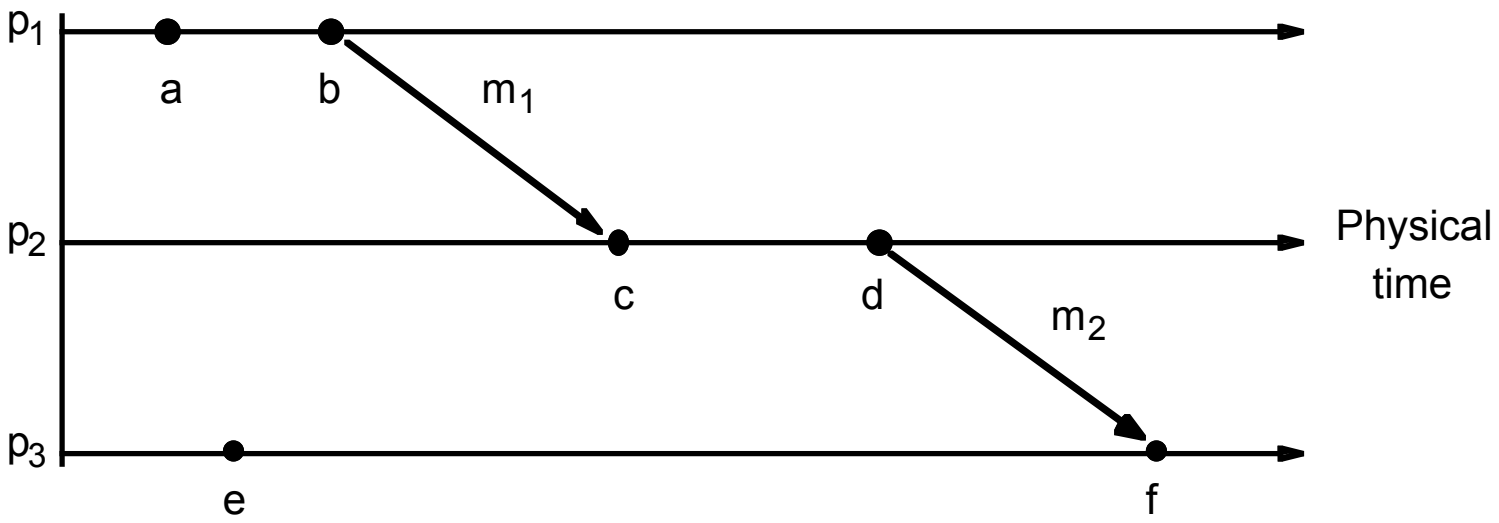
- Lamport stated that
  - In a distributed system, it is enough, processes agree on the order of events occur
    - It is not necessary all processes agree on exactly what time it is

- Lamport defines happened before relation as "$a \rightarrow b$"

- Rules
  1. If two events occurred at the same process p_i, then they occurred in the order observed by p_i, that is the definition of: "$\rightarrow$ i" ("happened before" i)
  2. A message, *m* is sent between two processes, *send(m)* happens before *receive(m)*
  3. The "happened before" relation is transitive $a \rightarrow b, b \rightarrow c \Rightarrow a \rightarrow c$

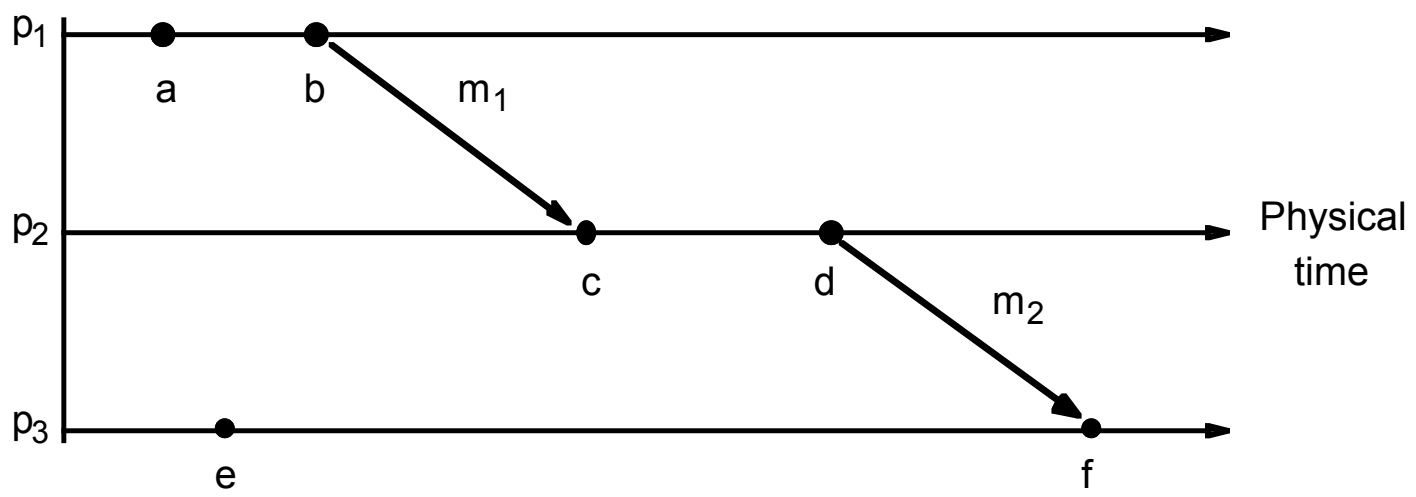
Logical Time and Logical Clocks

- $a \rightarrow b$ (at p_1) $c \rightarrow d$ (at p_2)
- $b \rightarrow c$ because of m_1
- $d \rightarrow f$ because of m_2



Logical Time and Logical Clocks

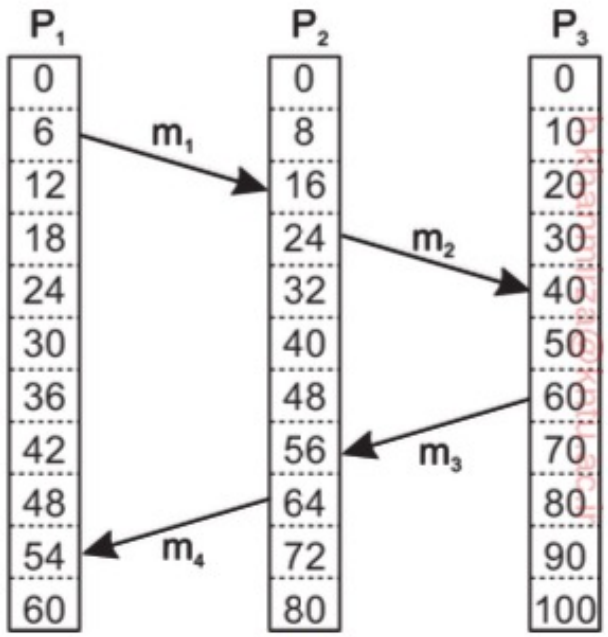
- Not all events are related by → relation
  - Consider *a* and *e* (no chain of messages to relate them)
    - They are not related by →
  - They are said to be concurrent written as $a \parallel e$
  - This means nothing can be said about when the events happened or which event happened first.



Logical Time and Logical Clocks

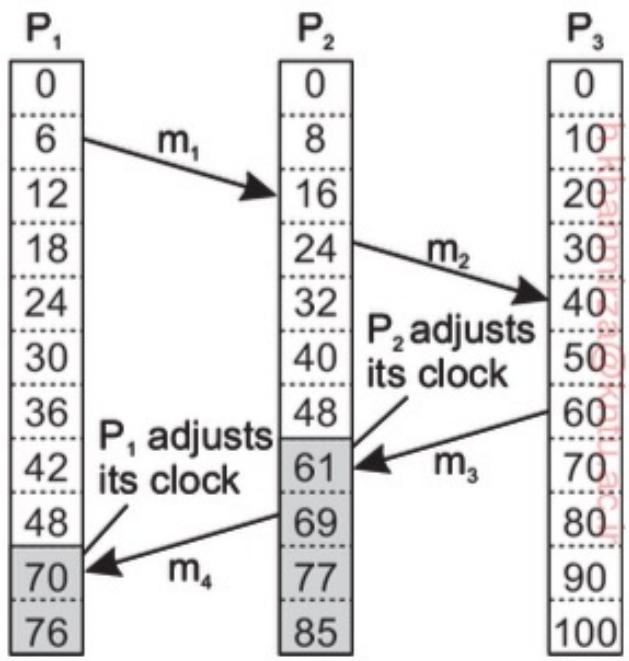
- A logical clock is a local monotonically increasing software counter
  - No need for any relation with physical clock

- Example
  - Assume three processes P1, P2, P3
  - Their clock is incremented by 6, 8, 10 units (due to drift or anything)
  - M1 is sent at 6 and received at 16 "→" is held
  - M2 is sent at 24 and received at 40 "→" is held
  - M3 is sent at 60 and received at 56 "→" is not held
  - M4 is sent at 64 and received at 54 "→" is not held



Lamport Logical Clock

- Example (cont.)
  - M3 is sent at 60 and received at 56 "→" is not held
  - P2 adjusts its clock to 60 + 1 → 61
  - Also for M4



- To break ties in distributed environment, process id is also sent in messages in addition to the logical clock
  - If a process receives $\langle 60|i\rangle, \langle 60|j\rangle$
  - $if\ i < j \rightarrow \langle 60|i\rangle < \langle 60|j\rangle$

- It can be proved that
  - $if\ a \rightarrow b \Rightarrow C(a) < C(b)$, $C_i(a)$:clock of event at process i
  - The reverse is not true!!

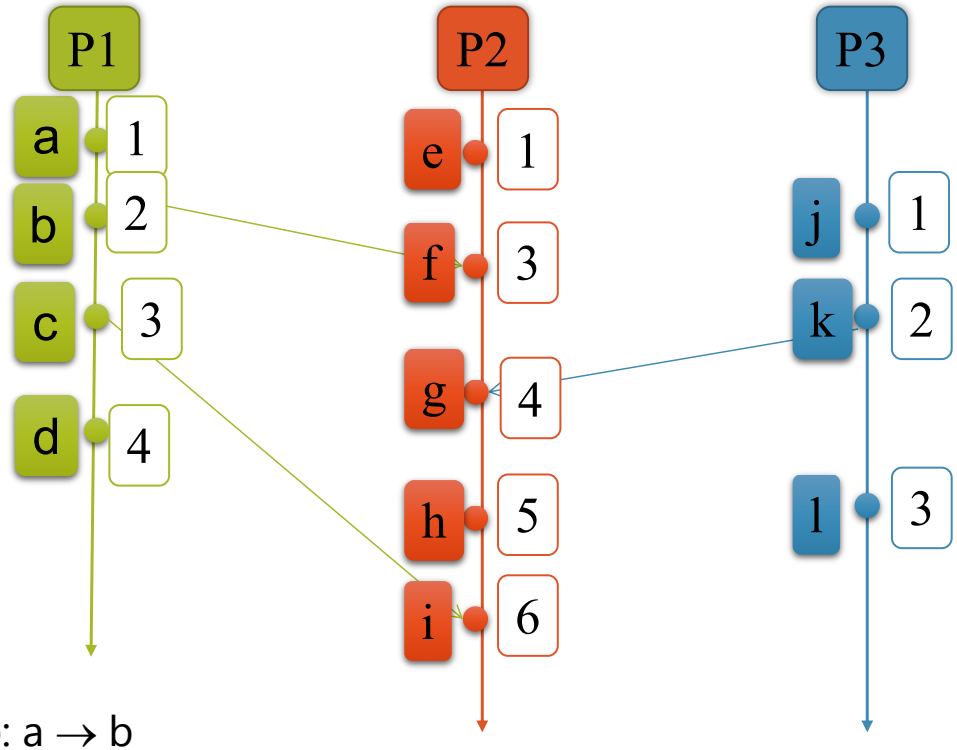
Lamport Logical Clock

- Lamport Logical Clock Algorithm

1. Before each event at process p: $C_p = C_p + 1$
2. Send message m, with local time $m = (m \quad , P_{id}, C_p)$
3. On receiving $(\mathrm{m}, \mathrm{P_{id}}, C_m)$, process q computes
   $C_q = \max(C_q, C_m) + 1$

Lamport Logical Clock

Assume that each process's logical clock is set to 0x



- Between a and b: a → b
- Between b and f: b → f
- Between e and k: concurrent
- Between c and h: concurrent
- Between k and h: k → h

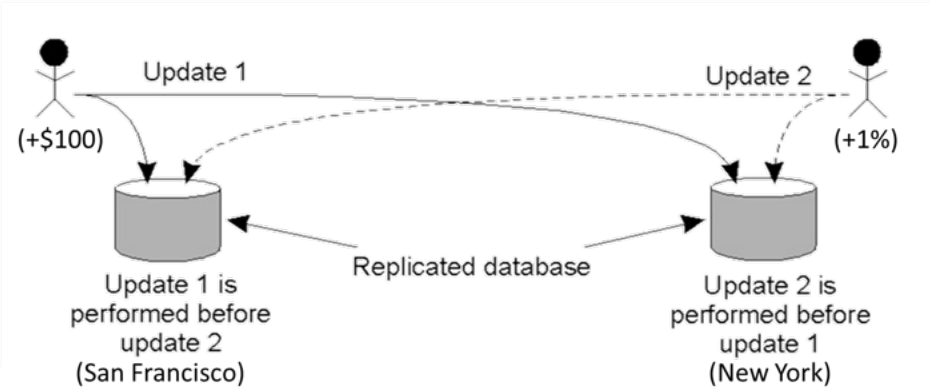
Totally Ordered Systems

- Lamport logical clock system results in Totally Ordered systems

- All events in a distributed system are totally ordered if event *a* happened before event *b*, then *a* will also be positioned in that ordering before *b*: $C(a) < C(b)$.

Totally Ordered Multicast

- Sometimes events must occur in order.
  - Scenario:
    - Data is saved in multiple data bases.
    - Upon update of one copy, refresh data in all other copies.
  - Example Application:
    - An account has 1000$
    - Actions are: X=Add 100$ Y=compute 1% interest
    - If in one copy X is done before Y → balance = 1111$
    - If in other copy X is done after Y → balance = 1110$
- Result:
  - Update messages MUST be considered in order



Totally Ordered Multicast

- Deliver multicast operations to receiver apps, in the same order
- Sender
  - Put local logical clock in each message (timestamp)
  - Send message to all group members (also yourself)
    - Assumptions: No loss, no out-of-order
    - Assumption: Send is done through multicast mechanisms, thus sending a message to all group members increments logical clock by 1

Totally Ordered Multicast

- Receiver
  - Puts messages in a local queue ordered by their timestamp
    - Each receive increments logical clock by 1
  - Multicasts an ACK to all other processes
  - Deliver message to application if
    - If it is at the head of the queue
    - It is ack'ed by all members

Totally Ordered Multicast

Totally Ordered Multicast

Totally Ordered Multicast

P1
- m1 → ACK2,ACK3
- m2 → ~~ACK2~~,ACK3

P2
- m1 → ~~ACK1~~,ACK3
- m2 → ACK1,ACK3

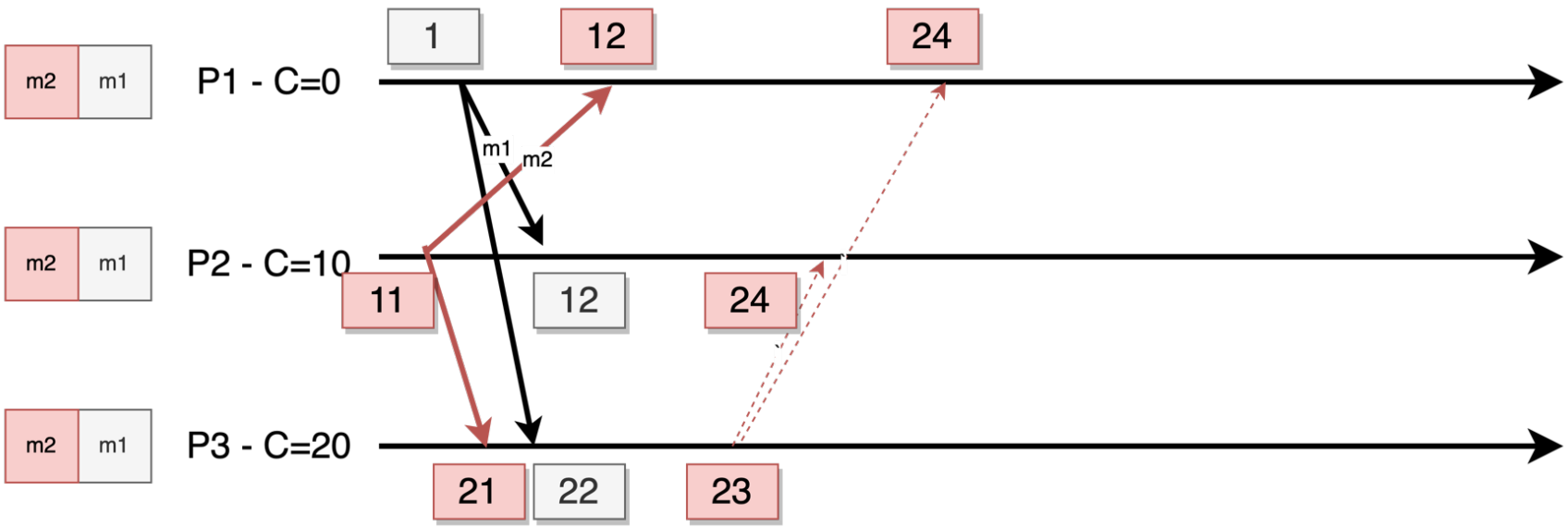
P3
- m1 → ~~ACK1~~,ACK2
- m2 → ACK1,~~ACK2~~

P1
m1 → ACK2,ACK3

means

Process P1 needs (→) ACK from P2 and P3

Totally Ordered Multicast



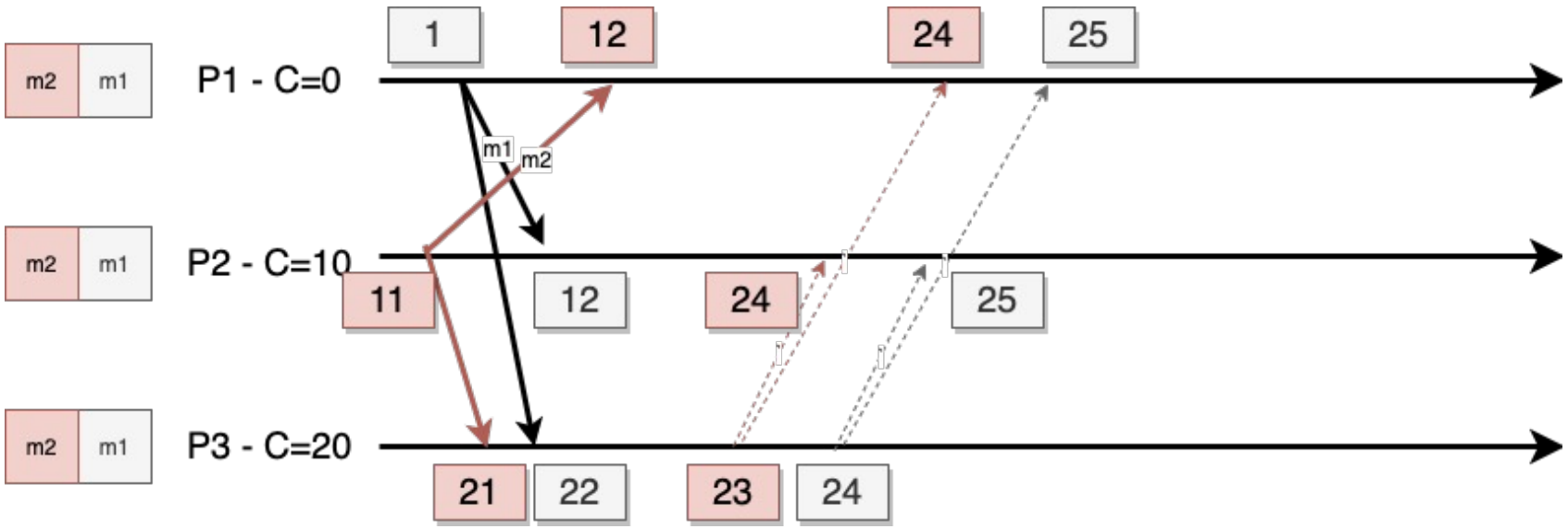
P1
- m1 → ACK2,ACK3
- m2 → ~~ACK2~~,~~ACK3~~

P2
- m1 → ~~ACK1~~,ACK3
- m2 → ACK1,~~ACK3~~

P3
- m1 → ~~ACK1~~,ACK2
- m2 → ACK1,~~ACK2~~

P1
has all ACKs for m2,
but cannot consume

Totally Ordered Multicast



P1
- m1 → ACK2,~~ACK3~~
- m2 → ~~ACK2~~,~~ACK3~~

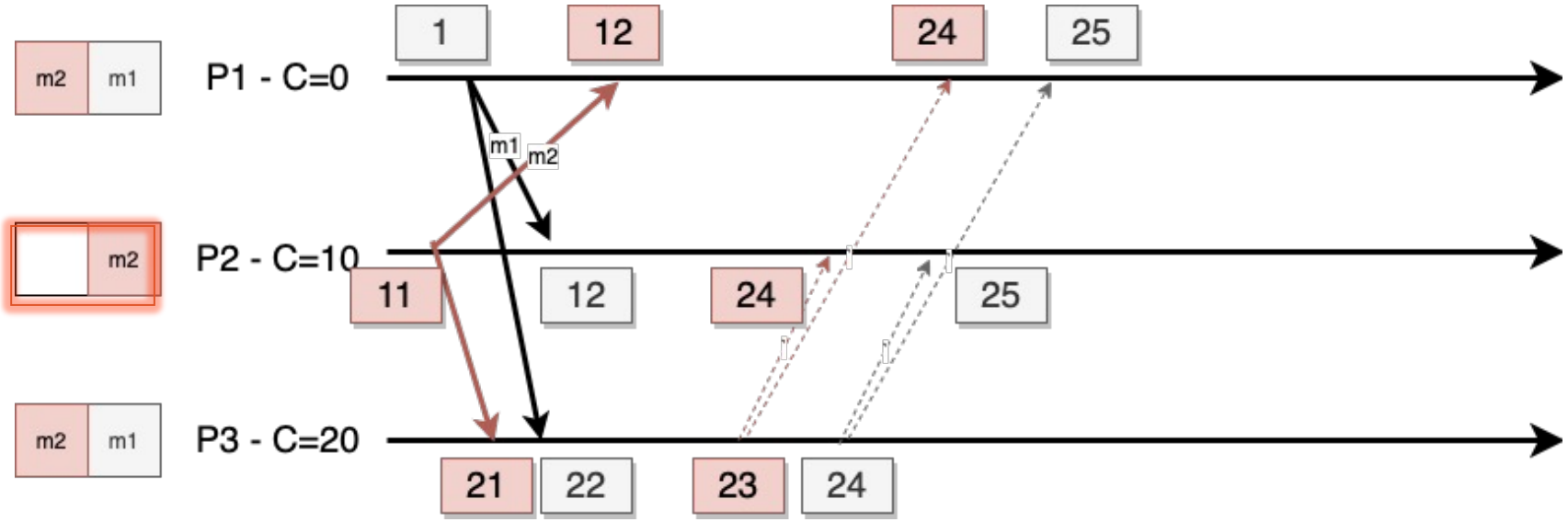
P2
- m1 → ~~ACK1~~,~~ACK3~~
- m2 → ACK1,~~ACK3~~

P3
- m1 → ~~ACK1~~,ACK2
- m2 → ACK1,~~ACK2~~

P1
has all ACKs for m2,
but cannot consume

P2 can consume m1

Totally Ordered Multicast



P1
- m1 → ACK2,~~ACK3~~
- m2 → ~~ACK2,ACK3~~

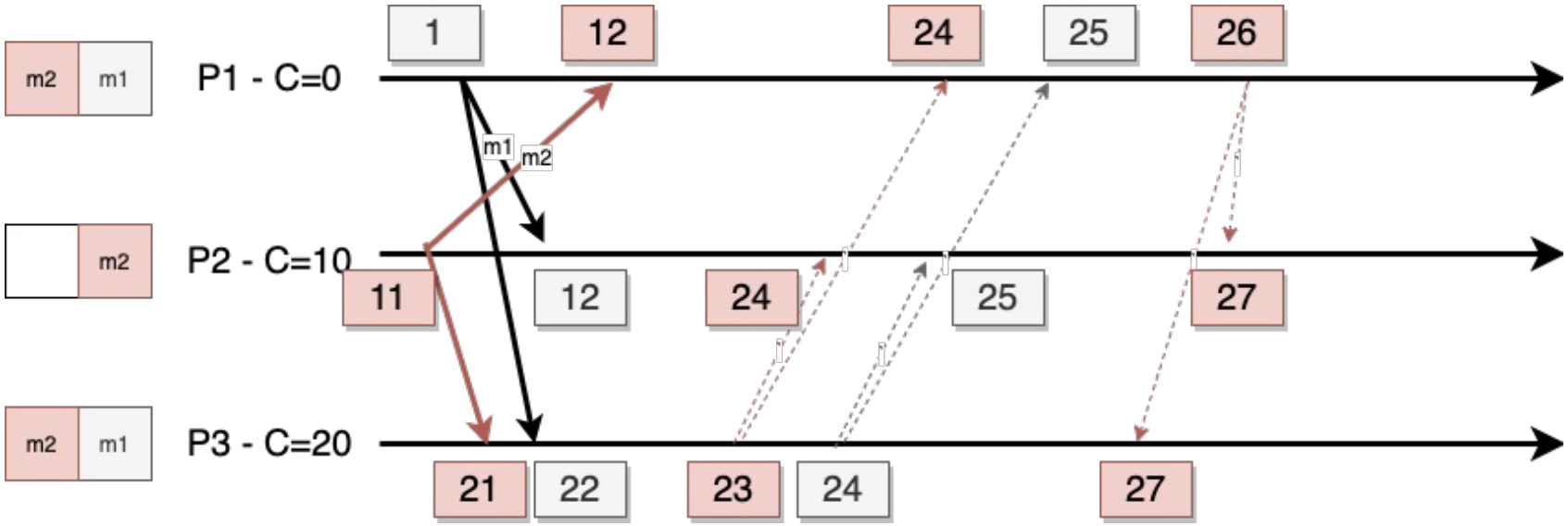
P2
- ~~m1 → ACK1,ACK3~~
- m2 → ACK1,~~ACK3~~

P3
m1 → ~~ACK1~~,ACK2
m2 → ACK1,~~ACK2~~

P1
has all ACKs for m2,
but cannot consume

P2 consumed m1

Totally Ordered Multicast



P1
- m1 → ACK2,~~ACK3~~
- m2 → ~~ACK2,ACK3~~

P2
- ~~m1 → ACK1,ACK3~~
- m2 → ~~ACK1,ACK3~~

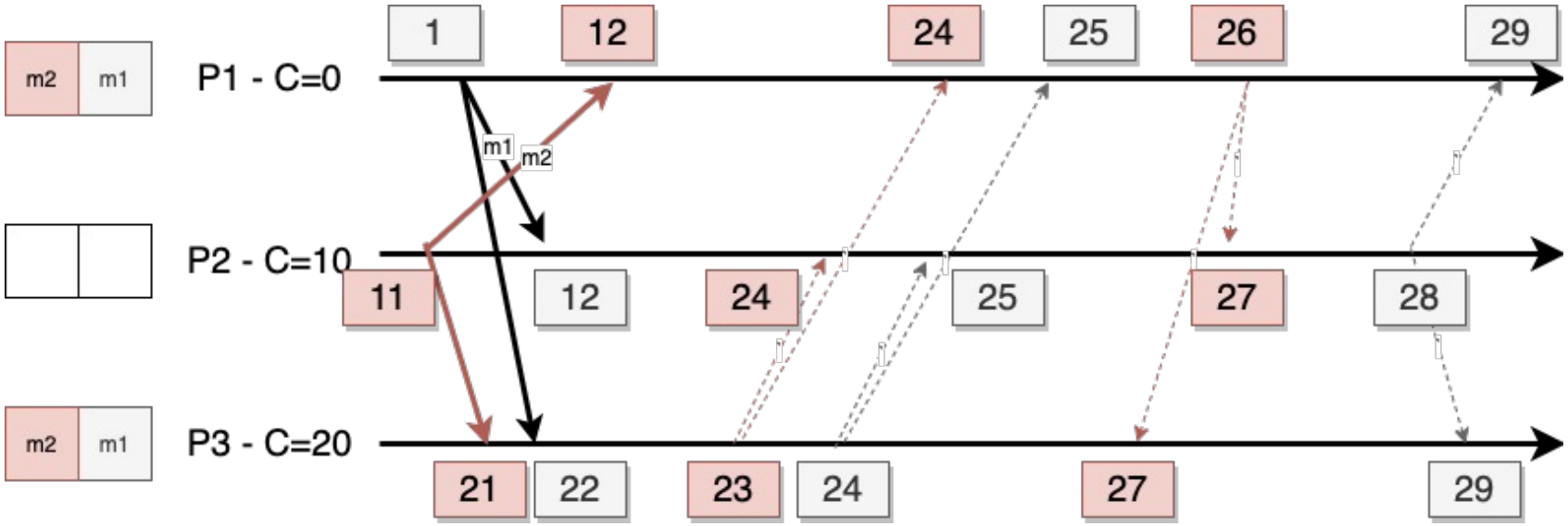
P3
m1 → ~~ACK1~~,ACK2
m2 → ~~ACK1,ACK2~~

P1
has all ACKs for m2,
but cannot consume

P2 consumed m1
P2 can consume m2

P3
has all ACKs for m2,
but cannot consume

Totally Ordered Multicast



P1
- ~~m1 → ACK2,ACK3~~
- m2 → ~~ACK2,ACK3~~

P2
- ~~m1 → ACK1,ACK3~~
- ~~m2 → ACK1,ACK3~~

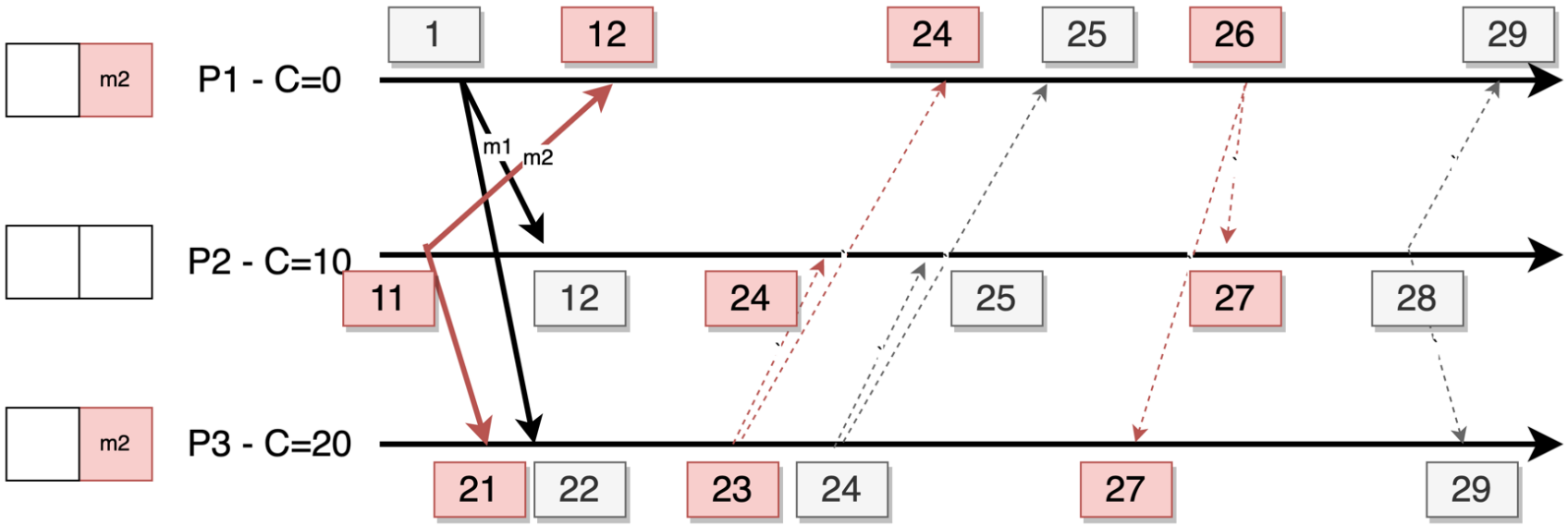
P3
~~m1 → ACK1,ACK2~~
m2 → ~~ACK1,ACK2~~

P1 consumed m1
P1 can consume m2

P2 consumed m1
P2 consumed m2

P3 consumed m1
P3 can consume m2

Totally Ordered Multicast



P1
- ~~m1 → ACK2,ACK3~~
- m2 → ~~ACK2,ACK3~~

P2
- ~~m1 → ACK1,ACK3~~
- ~~m2 → ACK1,ACK3~~

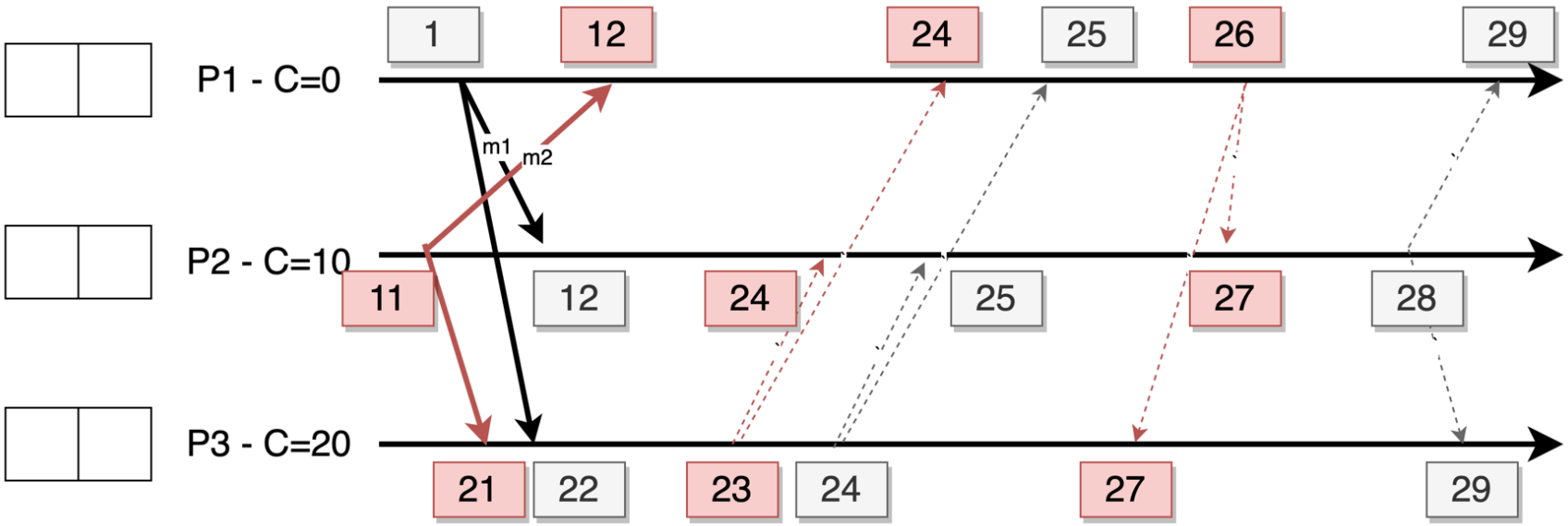
P3
~~m1 → ACK1,ACK2~~
m2 → ~~ACK1,ACK2~~

P1 consumed m1
P1 can consume m2

P2 consumed m1
P2 consumed m2

P3 consumed m1
P3 can consume m2

Totally Ordered Multicast



P1
- ~~m1 → ACK2,ACK3~~
- ~~m2 → ACK2,ACK3~~

P2
- ~~m1 → ACK1,ACK3~~
- ~~m2 → ACK1,ACK3~~

P3
~~m1 → ACK1,ACK2~~
~~m2 → ACK1,ACK2~~

P1 consumed m1
P1 consumed m2

P2 consumed m1
P2 consumed m2

P3 consumed m1
P3 consumed m2

Vector Clock

- With Lamport's logical clock
  - $a \rightarrow b \implies C(a) < C(b)$
  - $C(a) < C(b) \not\Rightarrow a \rightarrow b$
    - If we have *C(a) < C(b),* we cannot conclude *a→ b*
  - Lamport model do not capture Causality.

Vector Clock

- We can track causalities by keeping history of causes
  - Assume node P had two events then history $H(p_2) = \{p_1, p_2\}$
  - Node Q had only one event $H(q_1) = \{q_1\}$
  - P sends a msg $p_3$to Q with the most recent history: $H(p_3) = \{p_1, p_2, p_3\}$
  - Q records p_3 with event q_2 then $H(q_2) = \{q_1, p_1, p_2, p_3, q_2\}$
  - Checking whether an event p_3 causally precedes an event q can be checked by $H(p) \subset H(q)$
  - Good but very inefficient

Vector Clock

- No need to keep all the history, just keep the number of events happened in a node
- Implementation

- P_i keeps a $VC_i[n_1, \dots, n_n]$ where
  - n_j number of events happened in P_j
  - n_i number of events happened in P_i(= logical clock of node i)

- $VC_i[j] = k$ means P_i knows about P_j
  - Logical clock
  - At least *k* events have occurred in that node

Vector Clock

- Algorithm

1. With each event, nodes increment their clock $VC_i[i] \mathrel{+}= 1$
2. With each message, P_i piggybacks VC_i (its whole clock vector)
3. Upon receiving a message $VC_j[k] = \max\big(VC_j[k], ts(m)[k]\big)\ \forall k \in [n_1, n_n]$

Vector Clock

- If an event a has timestamp $ts(a)$, then $ts(a)[i] - 1$ denotes the number of events processed at P_i that causally precede a.

- When P_j receives a m from P_i with timestamp $ts(m)$, it knows about the number of events that have occurred at P_i that causally preceded the sending of m.

- P_j is also told how many events at other processes have taken place, known to P_i, before sending m.

Vector Clock

- P2 sends a message m1 with VC2 = (0, 1, 0) to P1 ts(m1) = (0, 1, 0)

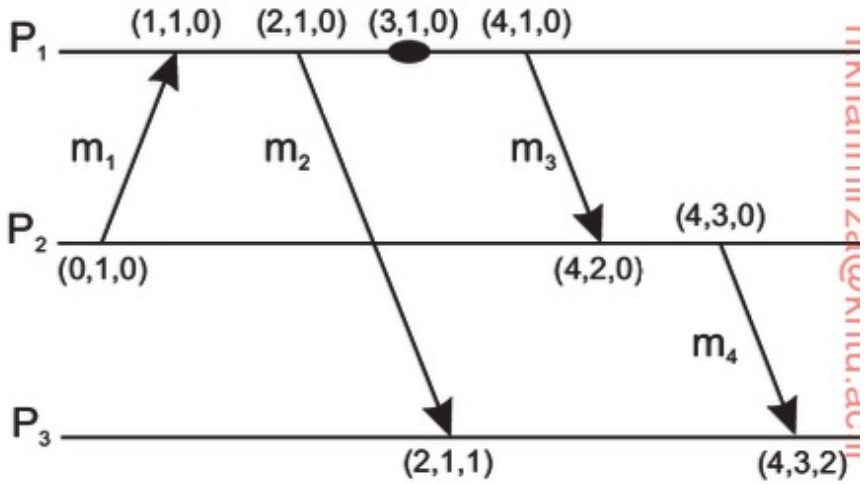
- P1 receives m1
  - Adjusts logical time to VC1 (1, 1, 0) and delivers it.
  - Increments its own clock as it is received a message

- Message m2 is sent by P1 to P3 with ts(m2) = (2,1,0)

- Before P1 sends another message, m3, an event happens at P1,

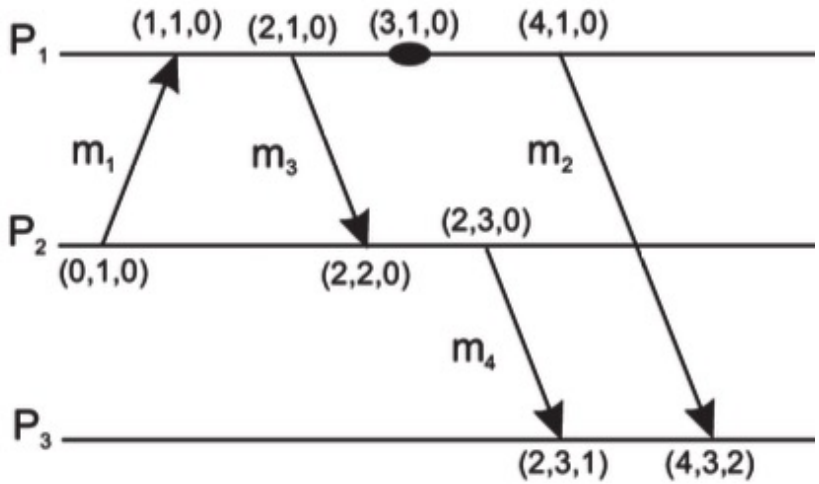
- P1 sends m3 with ts(m3)=(4, 1, 0).
- After receiving m3, process P2 sends message m4 to P3, with ts(m4) = (4, 3, 0)

- ts(m2)=(2,1,0) ≤ ts(m4)=(4,3,0)
  - m2 causally precedes m4



Vector Clock

- M3 is sent before m2
  - ts(m2)=(4,1,0)
  - ts(m4)=(2,3,0)
  - ts(m2) $\not\prec$ ts(m4)!
    - m2 and m4 may have conflict, we cannot say anything about their causalities



Causal-Ordered Multicast

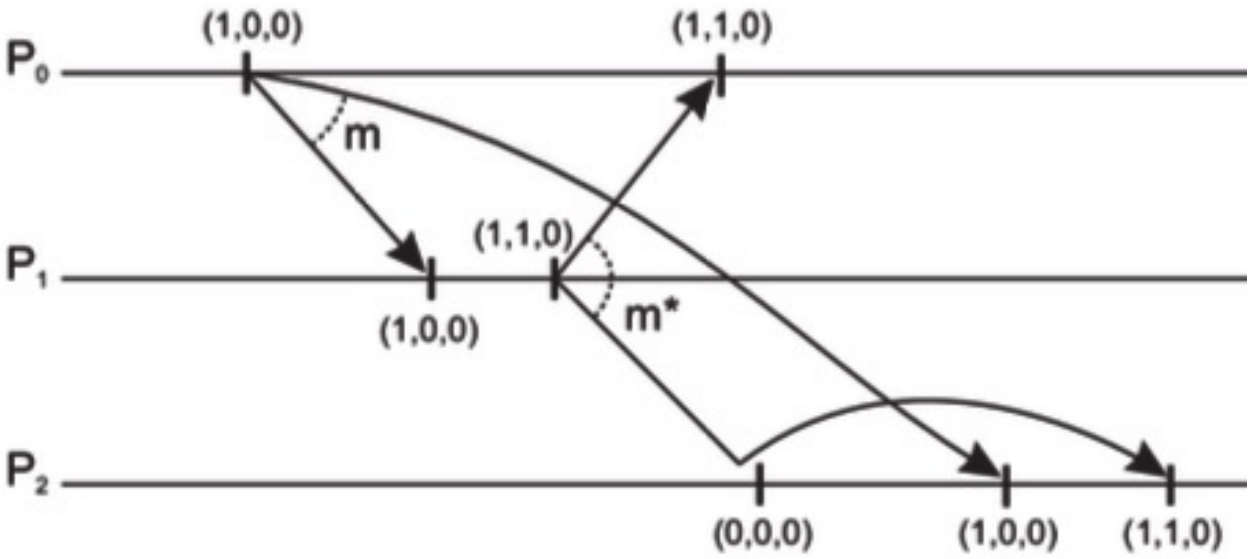
- A message is delivered only if all messages that may have causally precede it have been received as well

- If two messages are not related to each other, we do not care in which order they are delivered to applications

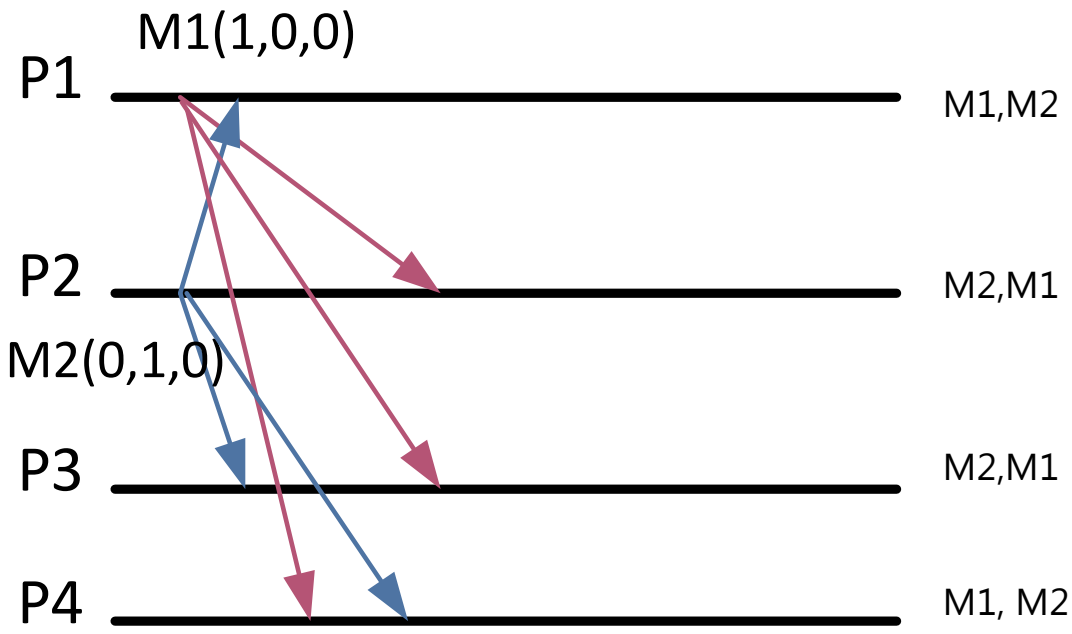
Causal-Ordered Multicast

- Principles
  - Clock is incremented when sending a message
  - No increment for receiving
  - Clock is adjusted to $\max(ts(m)[k], VC_i[k])$ when delivering message to the application (not upon receiving)
- Message from P_i received by P_j is delivered if
  - $ts(m)[i] = VC_j[i] + 1$
    - Message m is the next message that P_j was expecting from process P_i
  - $ts(m)[k] \leq VC_j[k] \quad \forall k \neq i$
    - P_j has delivered all the messages that have been delivered by P_i when it sent message m.

Causal-Ordered Multicast



Causal-Ordered Multicast



Distributed Mutual Exclusion

coordination

Mutual Exclusion

```
Perform local operations
Acquire(lock)
Execute critical section
Release(lock)
```

- Must ensure that only one instance of code is in critical section
- Multithreaded systems use shared memory. In a distributed system processes can only coordinate via message passing

Mutual Exclusion

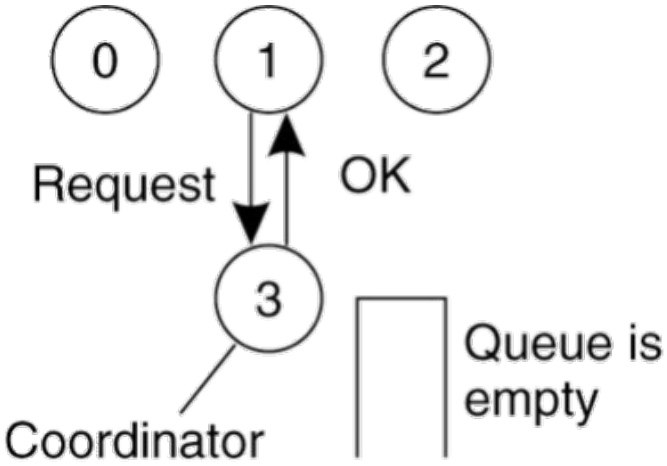
- Desired Properties of Mutual Exclusion
  - Mutual Exclusion (Correctness): single process in CS at a time
  - Progress (Efficiency): Processes don't wait for available resources
  - Bounded Waiting (Fairness): No process waits forever for a resource, i.e. a notion of fairness

Mutual Exclusion

Distributed Mutual Exclusion Other Requirements

1. Low message overhead
2. No bottlenecks
3. Tolerate out-of-order messages
4. Allow processes to join protocol or to drop out
5. Tolerate failed processes
6. Tolerate dropped messages

Centralized Algorithm



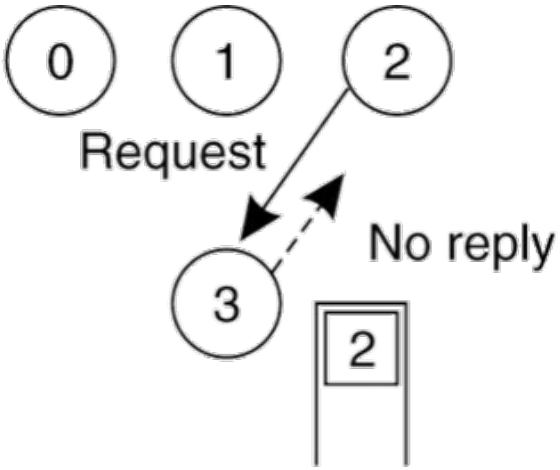
@ Server:

```
while true:
      m = Receive()
      If m == (Request, i):
            If Available():
                  Send (Grant) to i
```

@ Client → Acquire:

```
Send (Request, i) to coordinator
Wait for reply
```

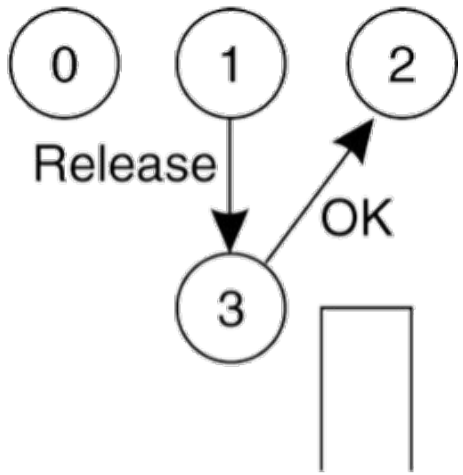
Centralized Algorithm



@ Server:

```
while true:
    m = Receive()
    If m == (Request, i):
        If Available():
            Send (Grant) to i
        else:
            Add i to Q
```

Centralized Algorithm



@ Client → Release:

```
Send (Release) to
coordinator
```

@ Server:

```
while true:
    m = Receive()
    If m == (Request, i):
        If Available():
            Send (Grant) to i
        else:
            Add i to Q
    If m == (Release)&&!empty(Q):
        Remove ID j from Q
        Send (Grant) to j
```

Centralized Algorithm

- Correctness:
  - Clearly safe
  - Process gets the resource if available
  - No starvation, of course strongly depends on queuing policy.
    - E.g., if always gave priority to lowest process ID, then processes 1 & 2 lock out 3
- Performance
  - One lock needs 3 messages per cycle (1 request, 1 grant, 1 release)
    - Cycle is a complete round of the protocol with one process i entering its critical section and then exiting
  - Server may become a bottleneck
- Issues
  - What happens when coordinator crashes? (Single-point-of-failure)
  - What happens when it reboots? (Requests are lost? Clients are waiting!)

Lamport's Distributed ME

- Uses Lamport's logical clock solution
- Algorithm:

- Each process maintains a request queue and a logical clock

- To enter a critical section
  - Call to `requestToEnter()`
    - Inserting an ENTER message with timestamp (clock,procID) into the local queue
    - Sending that message to all other processes
  - The operation `cleanup()` essentially sorts the queue.

Lamport's Distributed ME

```
def requestToEnter(self):
  self.clock = self.clock + 1                          # Increment clock value
  self.queue.append((self.clock, self.procID, ENTER)) # Append request to q
  self.cleanupQ()                                      # Sort the queue
  self.chan.sendTo(self.otherProcs, (self.clock,self.procID,ENTER)) # Send request
```

Lamport's Distributed ME

```
def receive(self):
  msg = self.chan.recvFrom(self.otherProcs)[1]        # Pick up any message
  self.clock = max(self.clock, msg[0])                # Adjust clock value...
  self.clock = self.clock + 1                         # ...and increment
  if msg[2] == ENTER:
    self.queue.append(msg)                            # Append an ENTER request
    self.allowToEnter(msg[1])                         # and unconditionally allow
  elif msg[2] == ALLOW:
    self.queue.append(msg)                            # Append an ALLOW
  elif msg[2] == RELEASE:
    del(self.queue[0])                                # Just remove first message
  self.cleanupQ()                                     # And sort and cleanup
```

Lamport's Distributed ME

```
def allowToEnter(self, requester):
  self.clock = self.clock + 1                          # Increment clock value
  self.chan.sendTo([requester], (self.clock,self.procID,ALLOW)) # Permit other

def allowedToEnter(self):
  commProcs = set([req[1] for req in self.queue[1:]]) # See who has sent a message
  return (self.queue[0][1]==self.procID and len(self.otherProcs)==len(commProcs))
```

Conditions to enter critical section
Executed with each msg arrival

Lamport's Distributed ME

```
def release(self):
  tmp = [r for r in self.queue[1:] if r[2] == ENTER]  # Remove all ALLOWs
  self.queue = tmp                                    # and copy to new queue
  self.clock = self.clock + 1                         # Increment clock value
  self.chan.sendTo(self.otherProcs, (self.clock,self.procID,RELEASE)) # Release
```

- Note that this release method is executed locally
- Use Lamport's algorithm, but break ties using the process ID
  - $C(e) = M \times C_i(e) + i$
    - M = maximum number of processes
    - i = process ID

Lamport's Distributed ME

P	Clk=3	Queue	Q	Clk=10	Queue
→(4,P,ENT)	3+1=4	(4,P,ENT)	→(11,Q,ENT)	10+1=11	(11,Q,ENT)

Lamport's Distributed ME

P	Clk=3	Queue	Q	Clk=10	Queue
→(4,P,ENT)	3+1=4	(4,P,ENT)	→(11,Q,ENT)	10+1=11	(11,Q,ENT)
←(11,Q,ENT)	Max(4,11)+1=12	(11,Q,ENT)			
Q→(13,P,ALW)	12+1=13				

Lamport's Distributed ME

P	Clk=3	Queue	Q	Clk=10	Queue
→(4,P,ENT)	3+1=4	(4,P,ENT)	→(11,Q,ENT)	10+1=11	(4,P,ENT)
←(11,Q,ENT)	Max(4,11)+1=12	(11,Q,ENT)	←(4,P,ENT)	Max(4,11)+1=12	(11,Q,ENT)
Q→(13,P,ALW)	12+1=13		P→(13,Q,ALW)	12+1=13	

Lamport's Distributed ME

P	Clk=3	Queue	Q	Clk=10	Queue
→(4,P,ENT)	3+1=4	(4,P,ENT)	→(11,Q,ENT)	10+1=11	(4,P,ENT)
←(11,Q,ENT)	Max(4,11)+1=12	(11,Q,ENT)	←(4,P,ENT)	Max(4,11)+1=12	(11,Q,ENT)
Q→(13,P,ALW)	12+1=13		P→(13,Q,ALW)	12+1=13	
←(13,Q,ALW)	13+1=14	(13,Q,ALW)	←(13,P,ALW)	13+1=14	(13,P,ALW)

- AllowedToEnter is executed in P
  - its own request is at head
  - it has received all ALLOW messages, then enters CS

Lamport's Distributed ME

P	Clk=3	Queue	Q	Clk=10	Queue
→(4,P,ENT)	3+1=4	(11,Q,ENT)	→(11,Q,ENT)	10+1=11	(11,Q,ENT)
←(11,Q,ENT)	Max(4,11)+1=12		←(4,P,ENT)	Max(4,11)+1=12	(14,P,ALWD)
→(13,Q,ALW)	12+1=13		→(13,P,ALW)	12+1=13	
←(13,Q,ALWD)	13+1=14		←(13,P,ALWD)	13+1=14	
→(15,Q,RLS)	14+1=15		←(15,P,RLS)	15+1=16	

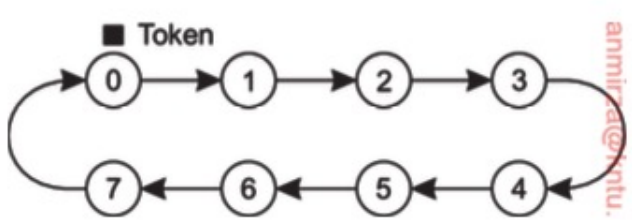
- After doing critical operation, removes its own request from the top
- Cleanup() procedure will remove all ALLOW messages from queue

Lamport's Distributed Mutual Exclusion

- Issues
  - Lamport's approach needs $3(N-1)$ message for each lock
    - P_i sends $n-1$ request messages
    - P_i receives $n-1$ reply messages
    - P_i sends $n-1$ release messages
  - Lamport's approach has N point of failure!
    - What if one of nodes fail and doesn't send response?

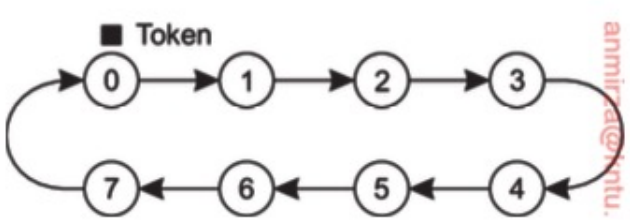
Token Ring Method

- Form a logical ring-shaped network among processes
- Each process knows only its next process
- A token is given to the first process
- Token circulates around the ring
- Assuming there are N processes, the token is passed from process P_k to process $P_{(k+1) \bmod N}$



Token Ring Method

- Upon receiving the token by a process
- If needs the resource, keep the token
- When done, pass token to the next process over ring
- It is not permitted to immediately enter the resource again using the same token.
- If the process is not interested in the resource, it just passes the token along.



Token Ring Method

- Correctness
  - Only one process has the token at a time
  - Token is circulated among processes, no process is starved
  - If a process wants the resource, at worst it will have to wait for the token
- Issues
  - High Latency
  - Token can be lost due to network or process failure
    - Detecting loss is not easy, since it maybe used in some process
  - A process may crash
- Solutions
  - When a process holds the token, sends ACK message to all others, Neighbor can detect process crash but may need the whole network re-configuration to repair the ring

Decentralized Algorithm

- Assume that there are n coordinators
- Access requires a majority vote from $m > \frac{n}{2}$ coordinators.
- A coordinator always responds immediately to a request with GRANT or DENY
- A coordinator denies if it has granted access to another process
- In case of unsuccessful attempt, back-off and retry later

Decentralized Algorithm

- Issues

- If large number of nodes request for access, the utilization rapidly drops
  - May none of nodes get enough votes, leads to wasting resources and large delays

- Coordinators may forget vote on reboot
  - If number of coordinators had reset recently exceeds $\frac{n}{2}$ then a violation occurs
  - Probability of violating correctness is very low and can be ignored

Comparison

Algorithm	Messages per entry/exit	Delay before entry (in message times)
Centralized	3	2
Distributed	$3 \cdot (N-1)$	$2 \cdot (N-1)$
Token ring	$1, \ldots, \infty$	$0, \ldots, N-1$
Decentralized	$2 \cdot m \cdot k + m, k = 1, 2, \ldots$	$2 \cdot m \cdot k$

m: number of coordinators
k: number of retries for acquiring lock

Distributed Leader Election

coordination

Leader Election

- Many distributed algorithms require one process to act as coordinator, initiator, or perform some special role
  - Master-slave scenarios, root selection in Spanning Tree Protocol

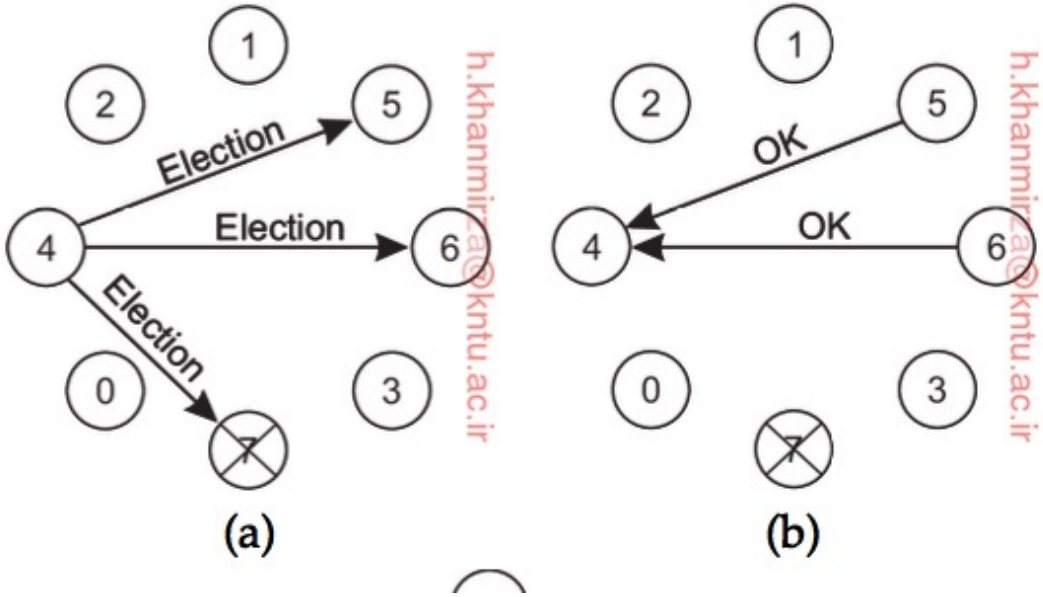
- Election algorithm ensures election process concludes with all processes agreeing on who is the new coordinator
  - We assume nodes at least have unique identifiers
  - Goal is finding the lowest/highest ID owner efficiently
    - Id can be anything like battery capacity, load, ⋯
    - All processes have complete knowledge of others IDs

Bully Algorithm

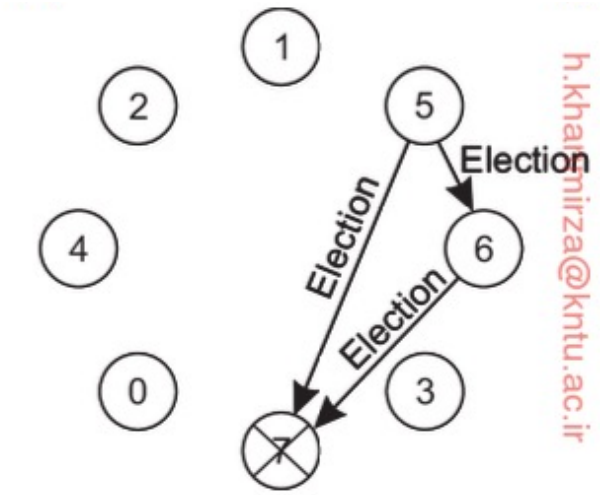
- Election Algorithm
  - Consider N processes $\{P_0, P_1, \dots, P_k\}$ and $id(P_k) = k$
  - If a process notices that no coordinator exists sends ELECTION message to all higher identifiers
    - If no one replies, the process becomes coordinator
    - If a process reply, it just become silent and waits!!
    - Always the process with higher identifier wins!
      - If the previous coordinator with highest id restart and become live, it sends ELECTION message and take over the coordinator job
  - Winner process sends COORDINATOR message to all others

- Needs $O(N^2)$ messages
  - If the process with lowest identifier detects the failure at first

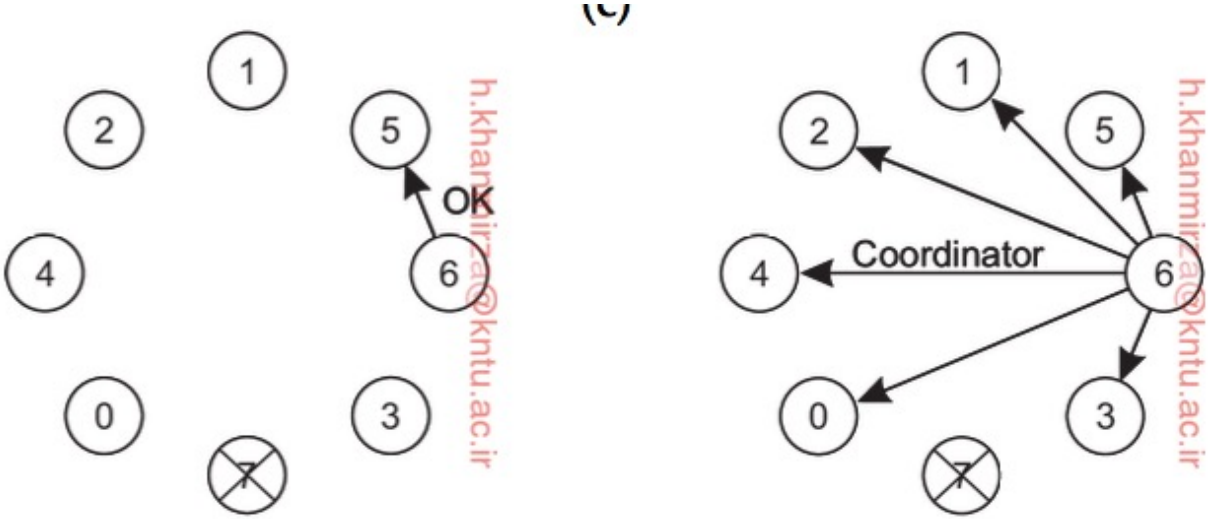
Bully Algorithm



Bully Algorithm



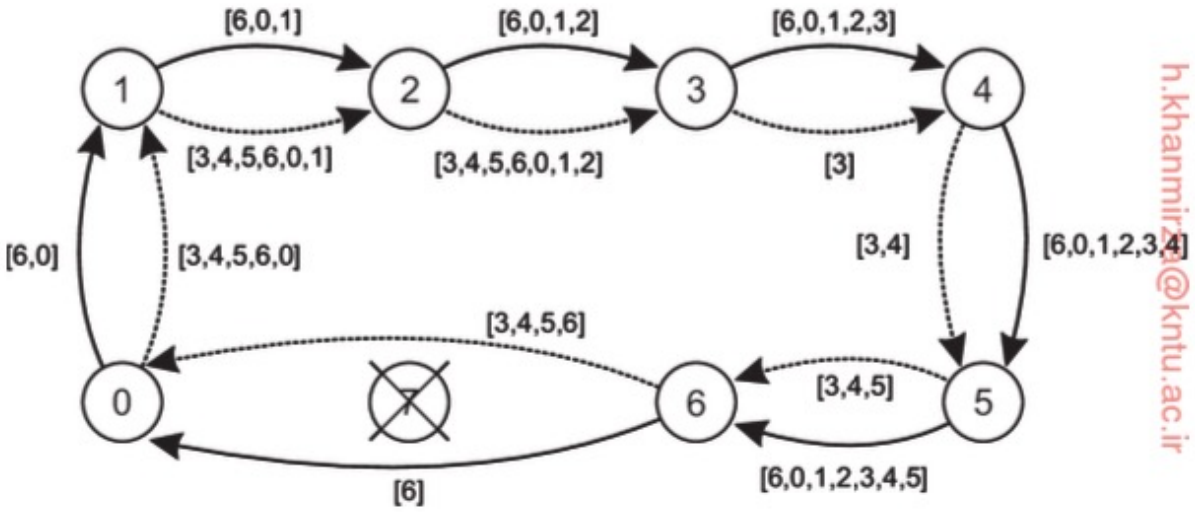
Bully Algorithm



Ring Algorithm

- Each process only knows its successor
  - In case of failure, it may know the next process, too!
- When a process notices the absence of the coordinator, sends an ELECTION message
  - This message contains the sender identifier
- Each process receives this message, appends its own id to the end of id list
- When the message is return back to the sender, it circulates a COORDINATOR message with the highest id in the ELECTION message
- Duplicate ELECTION and COORDINATOR messages is possible but not harmful
- Needs $2(N-1)$ messages

Ring Algorithm



Election in Wireless Networks

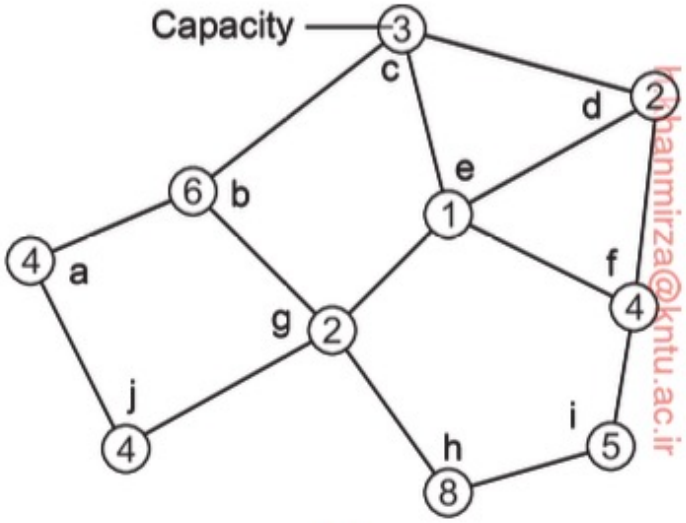
- In wireless environment
  - Message delivery is not reliable
  - Network topology may change

- Vasudevan Method proposed in 2004
  - Elects the best candidate
  - Assumes no topology change
  - Basically, the method builds a broadcast tree

Vasudevan Method

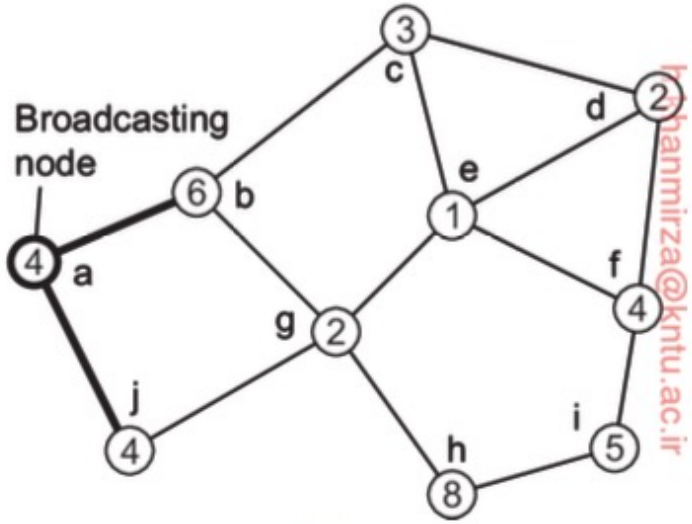
- A node sends an ELECTION message to its immediate neighbors
  - All nodes in transmission range
- Neighbors receive ELECTION message for the first time, select sender as parent.
  - If a node has parent and receive another ELECTION message it just acknowledges
  - In ACK message, nodes send their battery information, too
- Parent node compares the information of its down-stream nodes and elects the best and reports back the result to its own parent
- When the source nodes receives the results, elects the best and broadcasts the result to all nodes

Vasudevan Method



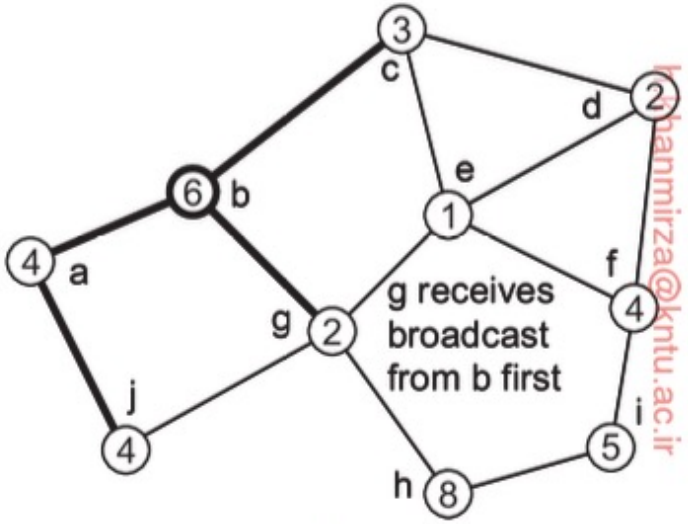
(a)

Vasudevan Method



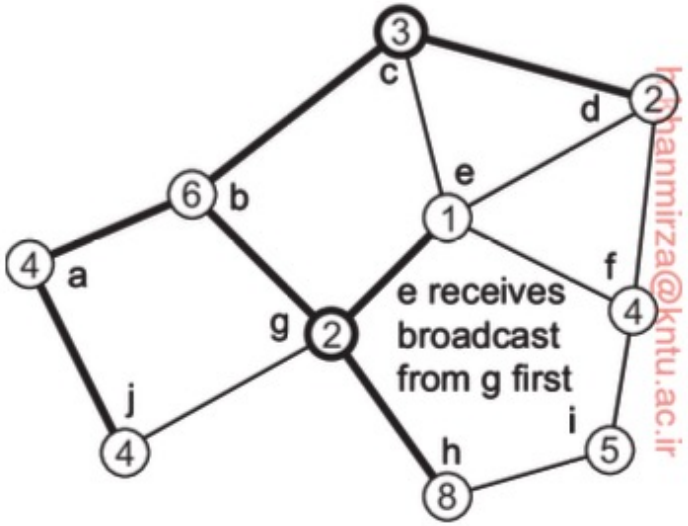
(b)

Vasudevan Method



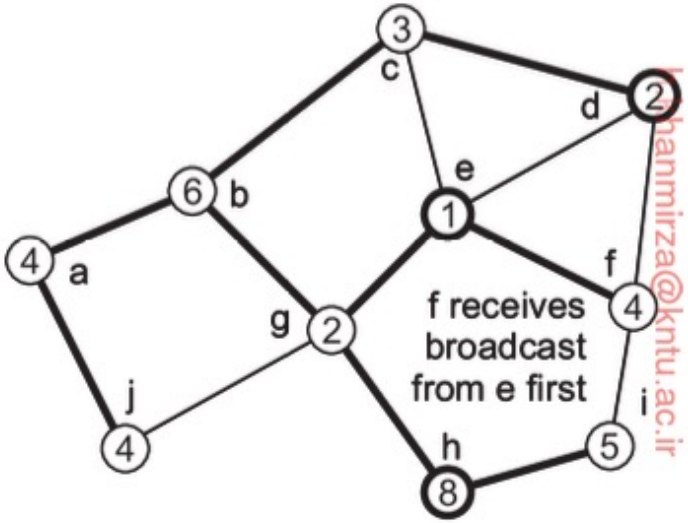
(c)

Vasudevan Method



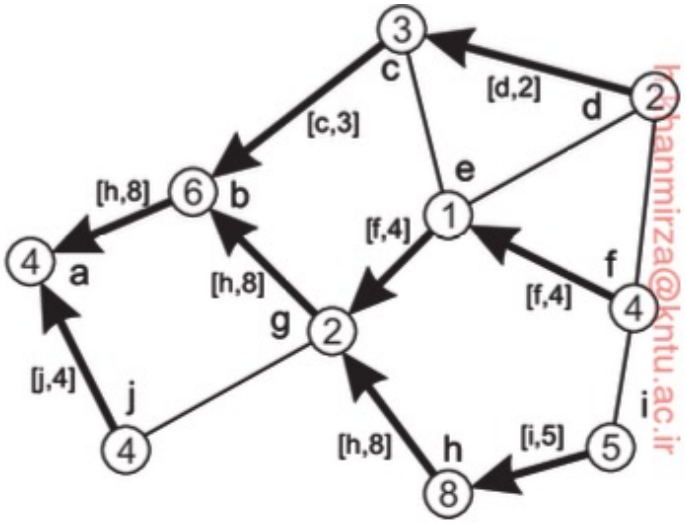
(d)

Vasudevan Method



(e)

Vasudevan Method



(f)

Vasudevan Method

- When multiple elections are initiated,
  - Each node will decide to join only one election
  - Each source tags its ELECTION message with a unique identifier
  - Nodes will participate only in the election with the highest identifier, stopping any running participation in other elections

Global States and Snapshots

Snapshot

- A set of local states

- Local state is the outcome of a recording event that follows a send, or a receive, or an internal action.

Snapshot

- Why Snapshots?

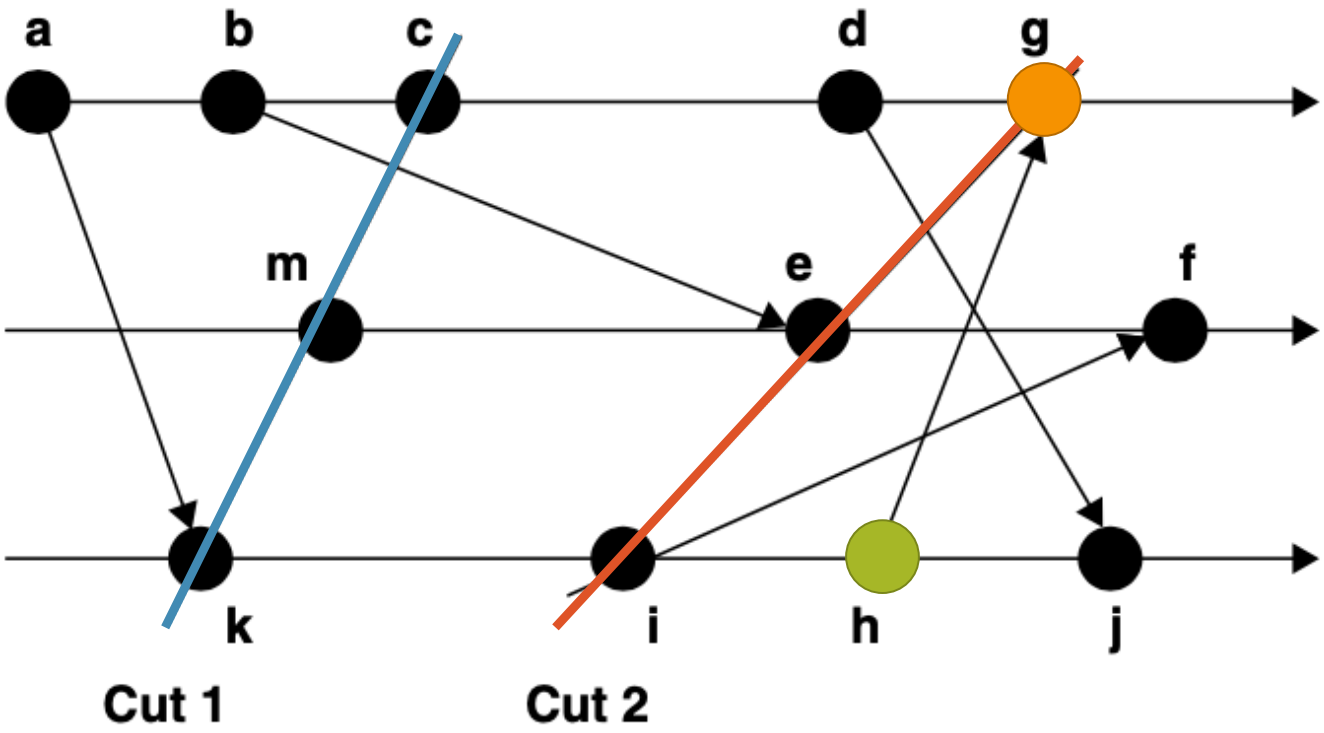
- Checkpointing
- Garbage Collection
- Deadlock Detection
- Debugging

Cut

- A set of events
- Contains at least one event per process.

- Consistent Cut
  - For each event that it contains, it also includes all events causally ordered before it.
  - Let *a*, *b* be two events in a distributed system. Then
  - $a \in C \wedge a \prec b \Rightarrow b \in C$
  - If recv(m) is in C, then send(m) must belong to C
  - A consistent cut induces a consistent snapshot

Cut



{a,b,c,m,k} {a,b,c,d,g,m,e,k,i}

Cut

- C_1, C_2 consistent cuts
- S_1, S_2 consistent snapshots induced by C_1, C_2

- If $C_1 \subset C_2$, C_2 is more recent than C_1
- Also, S_2 is more recent than S_1

- The set of local states following the most recent events of a cut defines a snapshot

Run

- A computation or a run is specified as a total order among the events of a distributed system

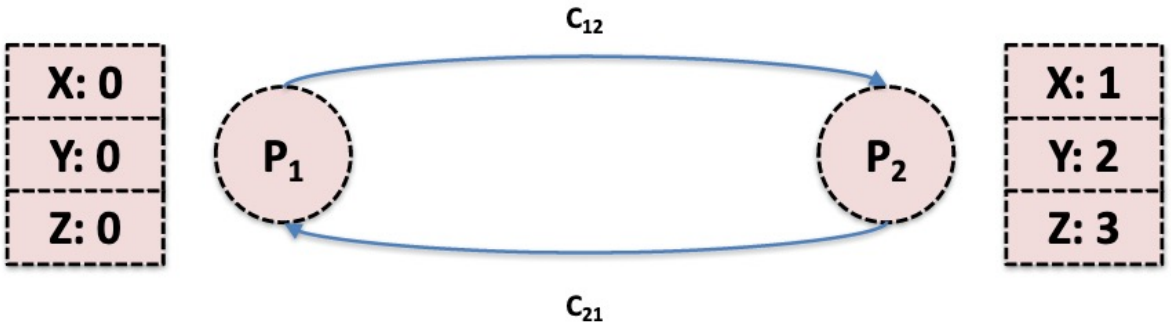
- A run is consistent when it satisfies the condition
- $\forall a, b: a \prec b \Rightarrow a\ precedes\ b$

- A consistent run reflects one of the feasible schedules of a central scheduler.

- $c||d$ two concurrent events
- {c,d} induces a consistent run U
- {d,c} induces a consistent run V

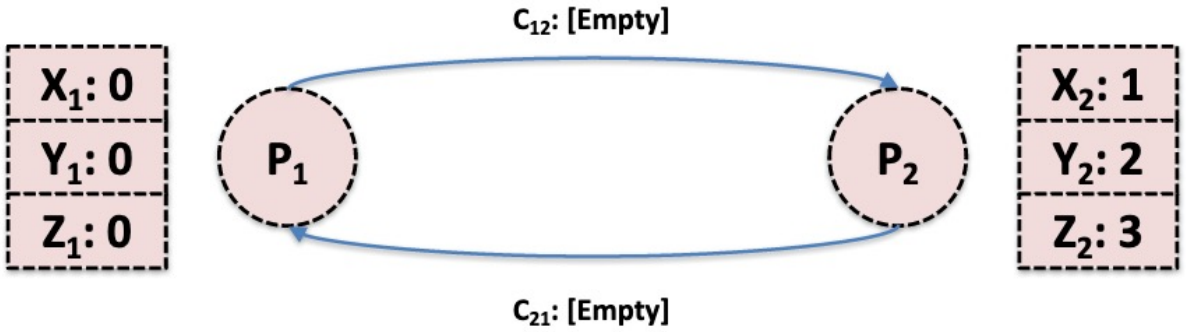
Chandy-Lamport Consistent Snapshot

- Suppose a distributed system as a strongly connected graph
- Processes are nodes
- Connected by FIFO channels
- All messages sent on channels arrive intact, unduplicated, in order



Chandy-Lamport Consistent Snapshot

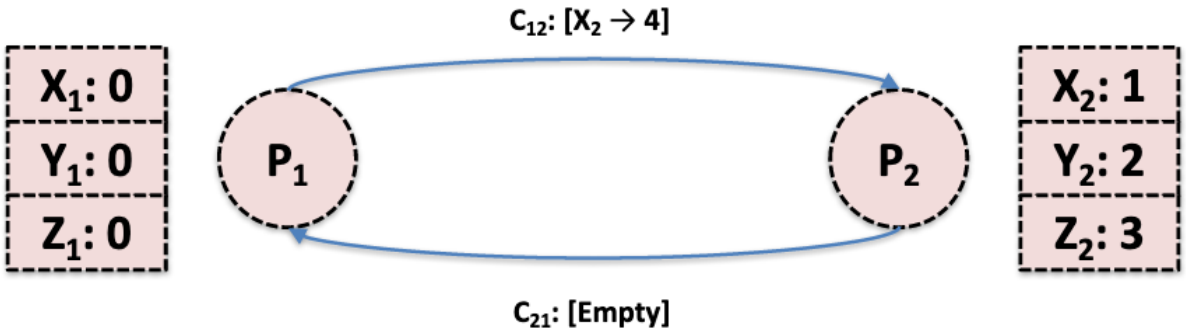
- Global state consists of
  - Local states of the processes
  - State of the channels



Chandy-Lamport Consistent Snapshot

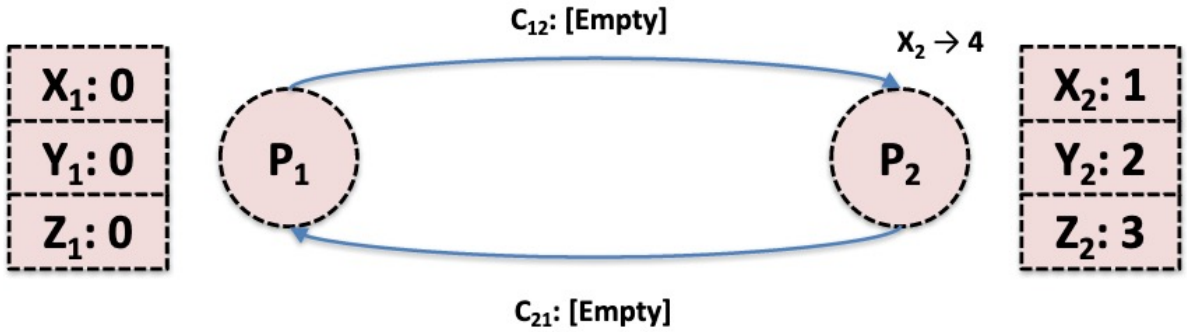
- P1 tells P2 to change a variable value
- This is another global state

- Why channel state should be recorded?
  - Missing states!



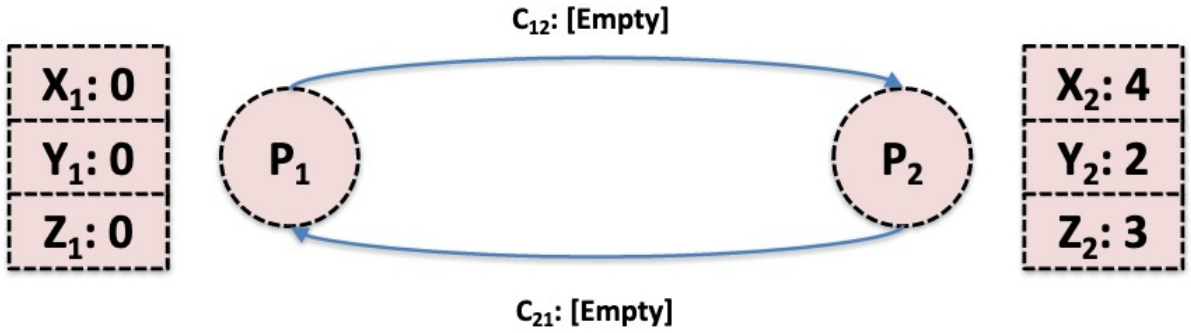
Chandy-Lamport Consistent Snapshot

- P2 receives the message
- This is another global state



Chandy-Lamport Consistent Snapshot

- P2 change the variable value
- This is another global state!



Chandy-Lamport Consistent Snapshot

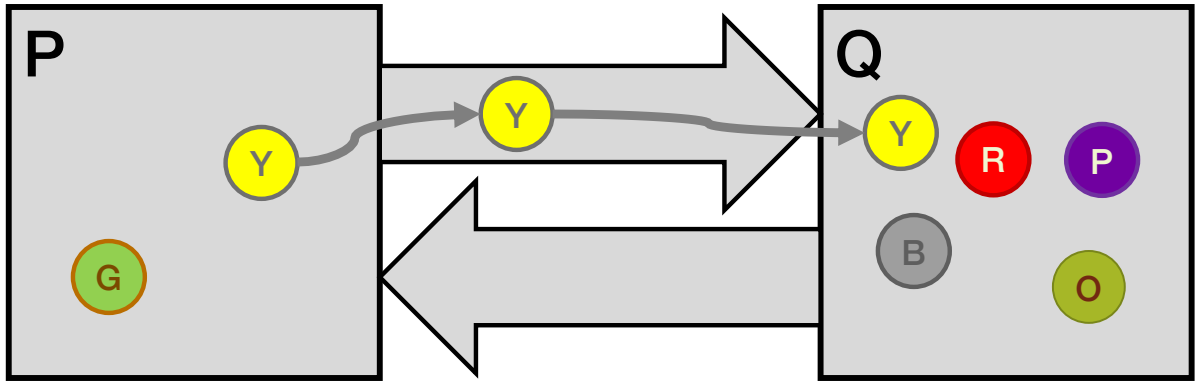
- The global state changes whenever an event happens
- Process sends message
- Process receives message
- Process takes a step

- Moving from state to state obeys causality

Chandy-Lamport Consistent Snapshot

- When snapshot should be taken?

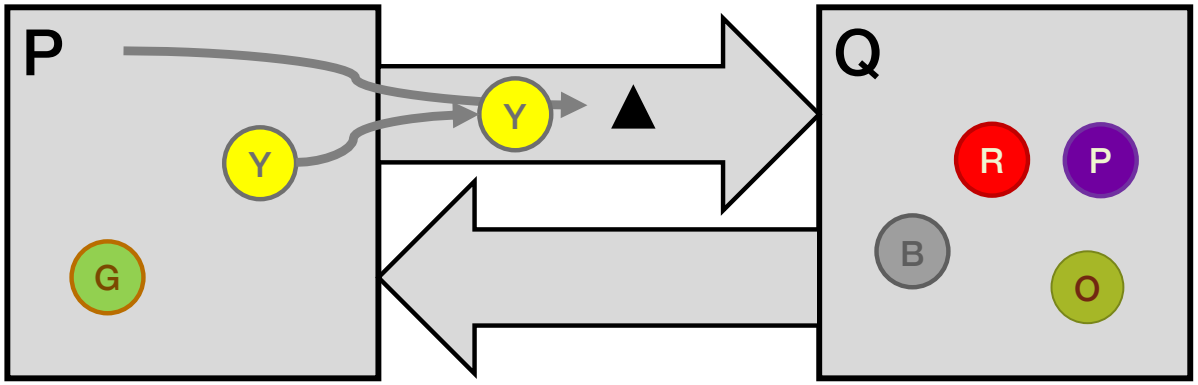
- Suppose P do snapshot before send and Q after receive
- Duplicate state!



P = { G, Y } Q = { Y, R, P, B, O }

Chandy-Lamport Consistent Snapshot

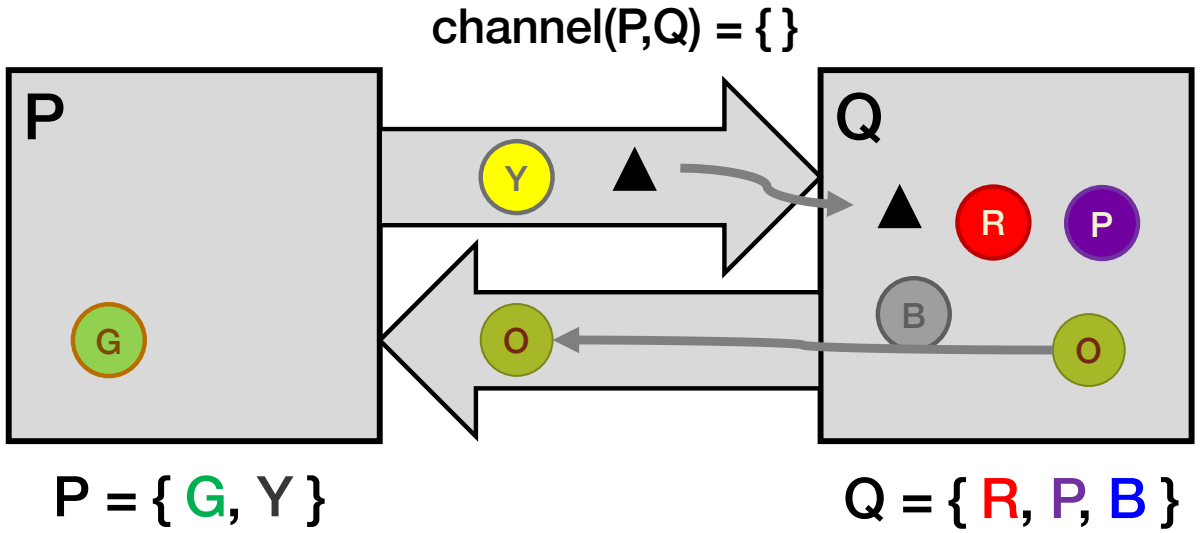
- INITIATOR node
  - Records its own state
  - Sends a special marker (▲) message in all channels
  - Any node can be initiator



snap: P = { G, Y }

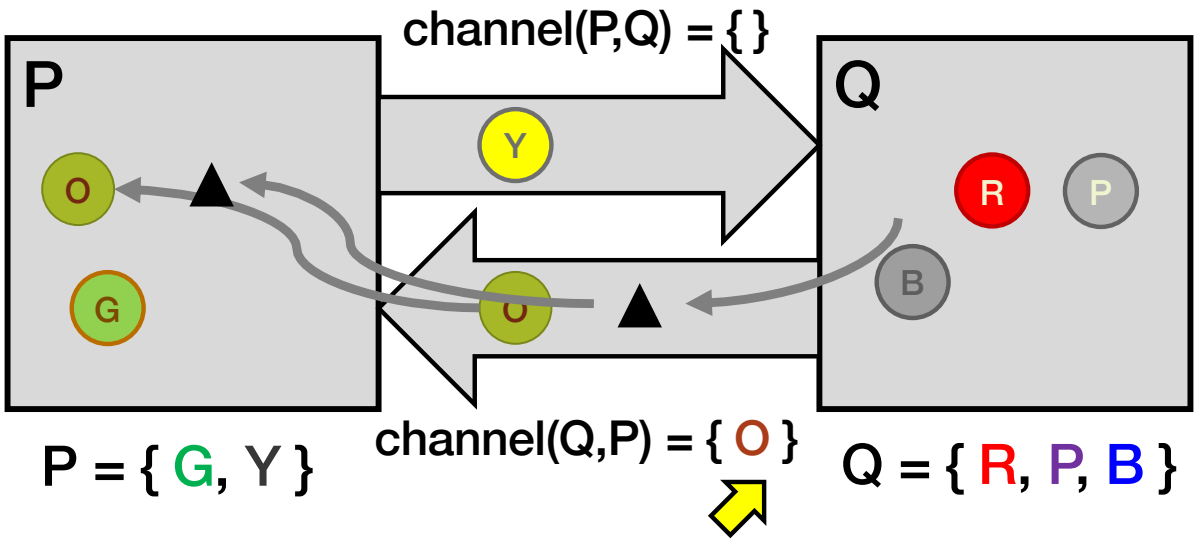
Chandy-Lamport Consistent Snapshot

- Receiver when receives a marker and not-snapshotted
  - Record the local state
  - Set receiving channel as Empty
  - Send marker in all outgoing channels



Chandy-Lamport Consistent Snapshot

- Receiver when receives a marker and snapshotted
  - Record the channel state



Chandy-Lamport Consistent Snapshot

- Termination:
- All processes have received a marker on all the N-1 incoming channels (and recorded their states)
- A central server can gather the partial states to build a global snapshot
- Note: recording of local state MUST be done atomically

Chandy-Lamport Consistent Snapshot

- Every cut corresponds to a global state and each global state can be represented by a cut

- A consistent global state corresponds to a cut in which every message received in the PAST of the cut has been sent in the PAST of that cut.

- All the messages that cross the cut from the PAST to the FUTURE are captured in the corresponding channel state.

Distributed Debugging

Distributed Debugging

- Our aim is to find if a condition has held during a run of a distributed system

- Possibly(ϕ)
  - There is a consistent run, $\phi(S) = True$ in a global state S

- Definitely(ϕ)
  - For every consistent run, there exists a global state of it in which predicate $\phi(S) = True$

Distributed Debugging

- Stable Predicates
  - Termination
  - Deadlock

- Unstable Predicates
  - Some properties may hold intermittently
  - Like: if $|x_1 - x_2| < 100$

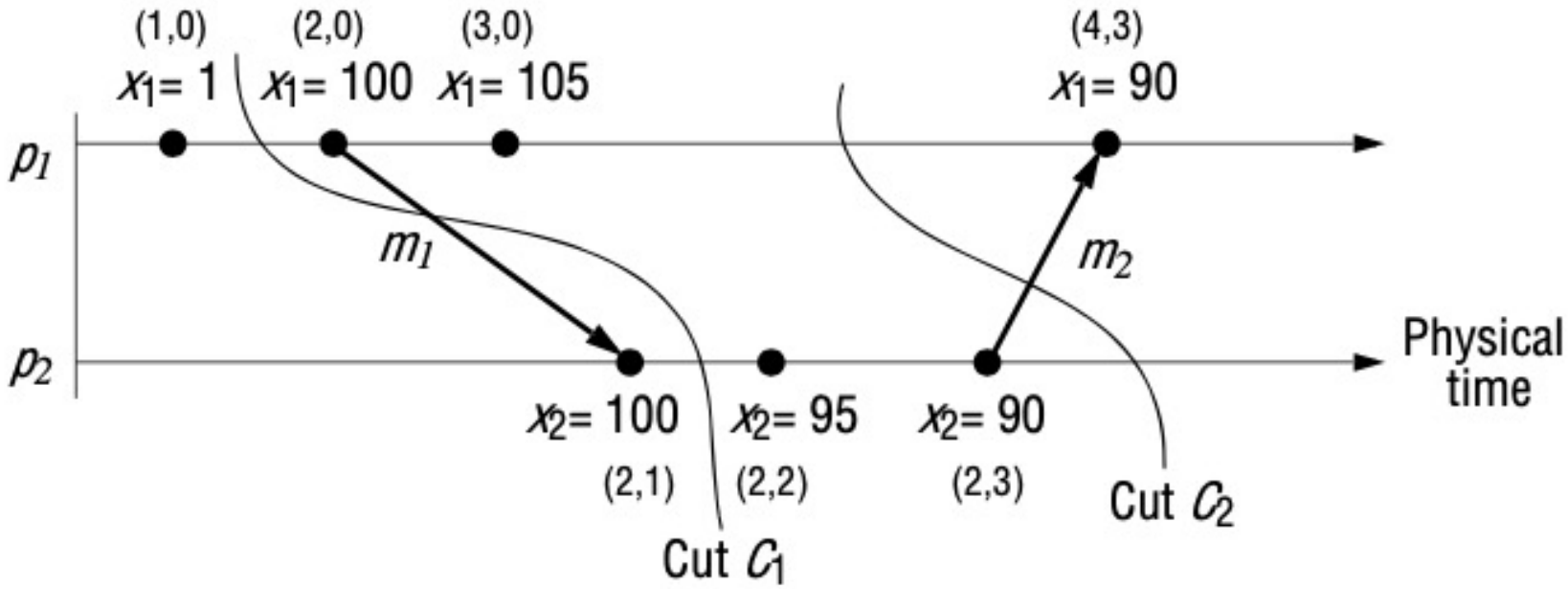
  - We should search in all snapshots to see if this predicate holds!
  - Several runs of a distributed system may pass through different global paths

Distributed Debugging

- A central Monitor receives state of processes P_i {i=1,..,N} through separate queue

- Processes send the state of the relevant variables
- Processes send if the state of the relevant variables is changed

Distributed Debugging



- Measurements must be done on consistent global states
- Cut C_1 is not consistent and x_2-x_1=99>50
- Cut C_2 is consistent and x_2-x_1=105-90<50

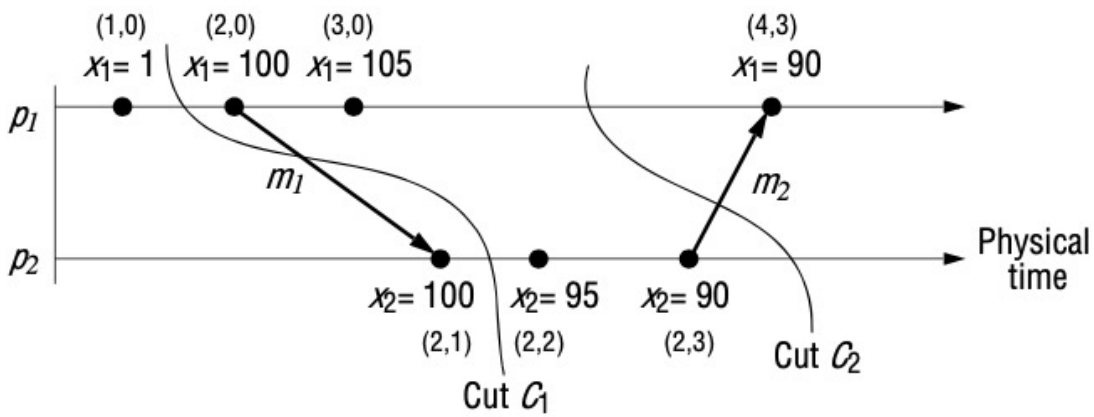
Distributed Debugging

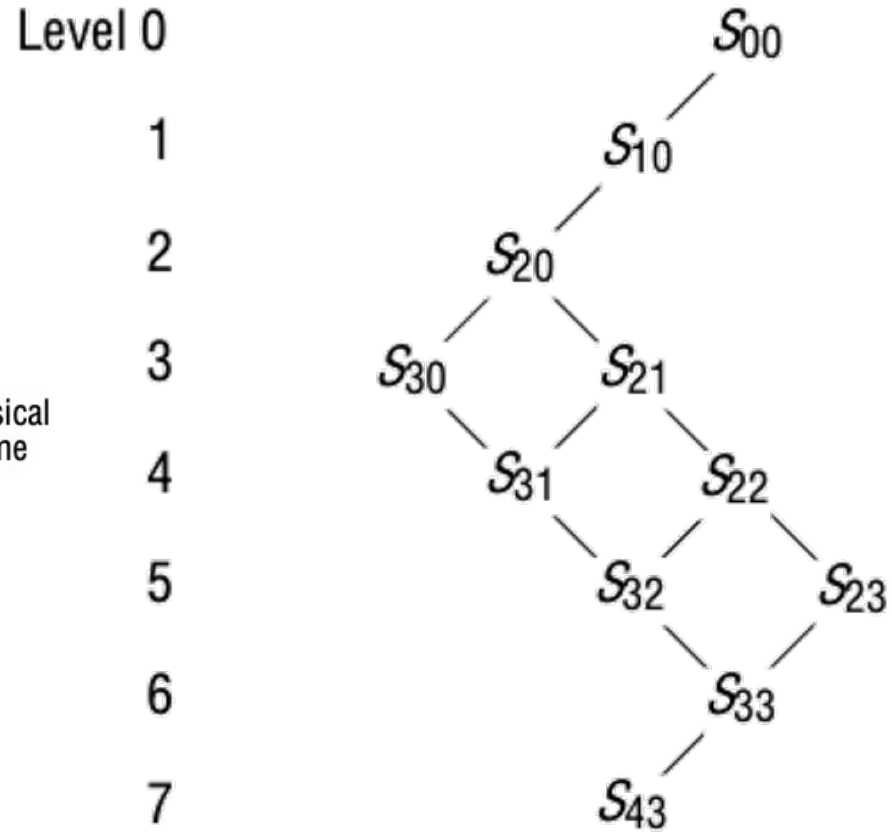
- Monitor should distinguish consistency of global states
- Processes send their Vector Clock with their local state
- $S = (s_1, s_2, \dots, s_N)$, global state gathered from state messages
- $V(s_i)$: vector clock of process p_i
- S is consistent global state if $V(s_i)[i] \geq V(s_j)[i], i, j = 1,2, \dots, N$
  - The sender of the state message has recorded more events
  - Others have not seen more events than the process itself! when sending their own local state

Distributed Debugging

- All consistent runs can be shown on a lattice
- The lattice shows us all the linearizations corresponding to a history

- S_{01} is not consistent, it does not appear in lattice

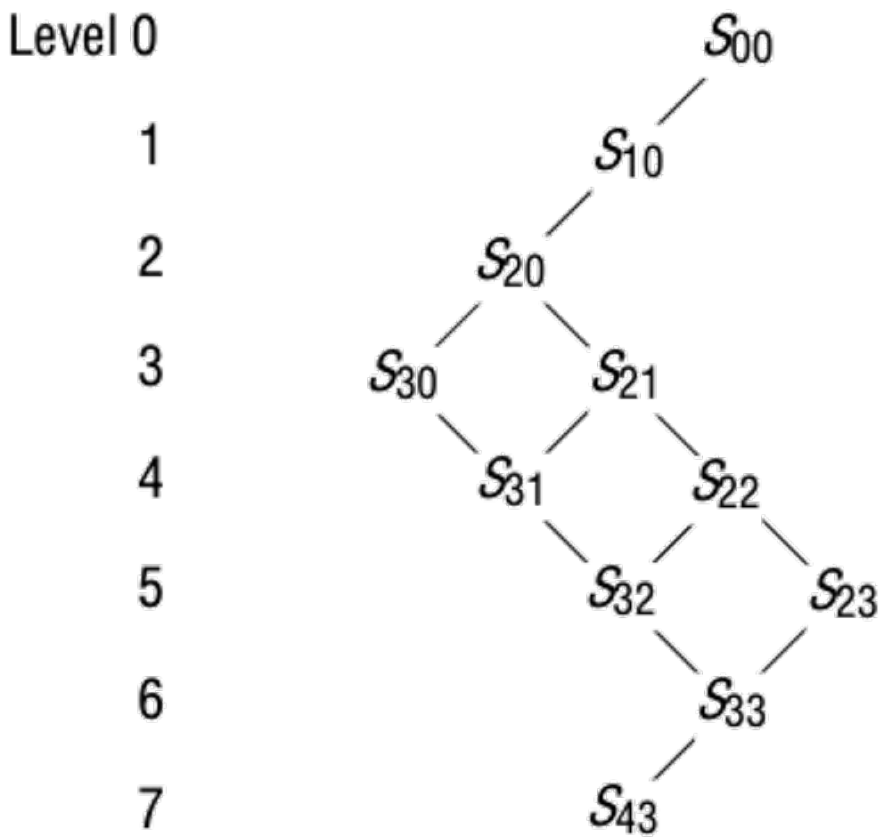




Distributed Debugging

- In each step of a run level is increased by 1
- S_{ij} is in the level of (i+j)

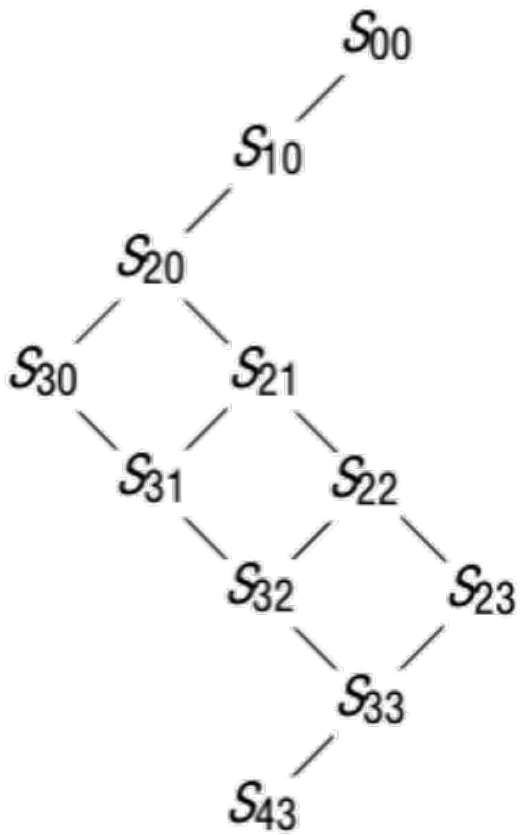
- S_{22} is reachable from S_{20}
- S_{22} is not reachable from S_{30}



Distributed Debugging

- Possibly(ϕ)
- Monitor starts at the initial state and steps through all consistent states and stops when $\phi(S) = True$

- Definitely(ϕ)
- Find states that all linearizations must pass
- At those states $\phi(S) = True$
- like $\phi(S_{20}) = True$



References

- Princeton Distributed Systems course, https://www.cs.princeton.edu/courses/archive/spring22/cos418/schedule.html

- Distributed Computing: Principles, Algorithms, and Systems, Ajay D. Kshemkalyani, 2011, Cambridge University Press

- Distributed Systems An Algorithmic Approach, Sahni, 2007, Taylor & Francis Group

The End!