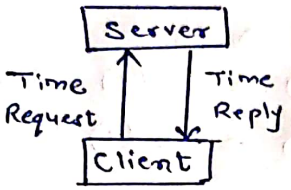
Distributed Computing
Module 3
Chapter 3 - Synchronization

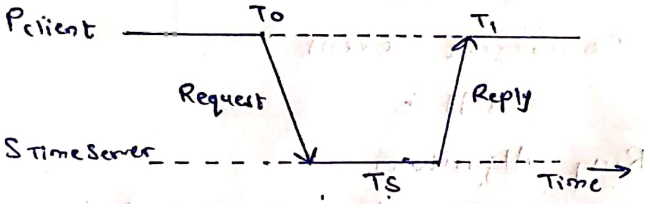
In synchronization, there are 2 types of clocks
① Logical clock
② Physical clock

Physical clock -
① Cristian Algorithm
② Berkeley Algorithm
③ Network Time Protocol

Cristian's Algorithm



$$T_{new} = T_{server} + \frac{T_1 - T_0}{2}$$



Algorithm:
- Let S be the time server and Ts be its Time.
- Process P requests the time from S.
- After receiving the request from P, S prepares response and append the time Ts from its own clock and then send it back to P.

Example:

Send Request → 8:08:18:100 (T_0)

Receive Response → 8:08:18:900 (T_1)

Response → 8:09:28:300 (T_{server})

$$\therefore T_{New} = 8:09:28:300 + \frac{900-100}{2}$$

$$= 8:09:28:300 + \frac{800}{2}$$

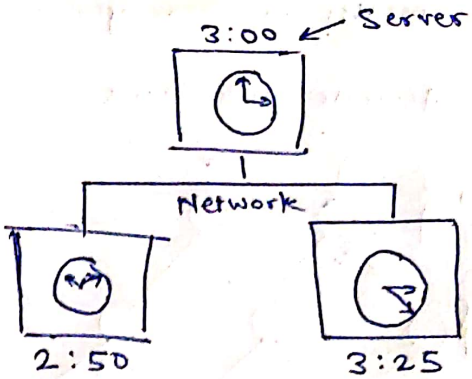
$$= 8:09:28:300 + 400$$

$$= 8:09:28:700$$

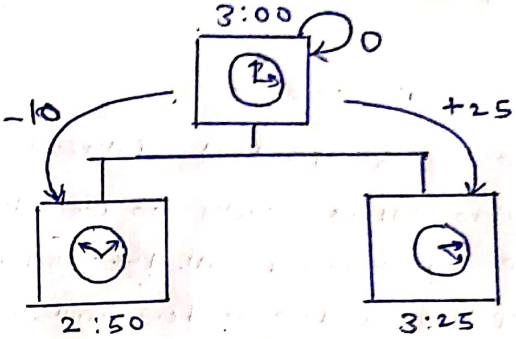
- In Cristian algorithm, client approaches server.
- In Berkeley Algorithm, server approaches client.

Note: Both algorithms are centralized.

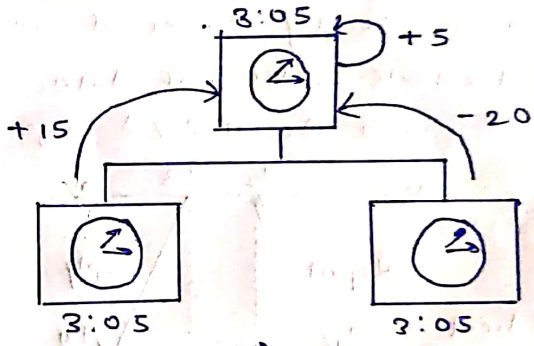
Berkeley Algorithm



(a)



(b)

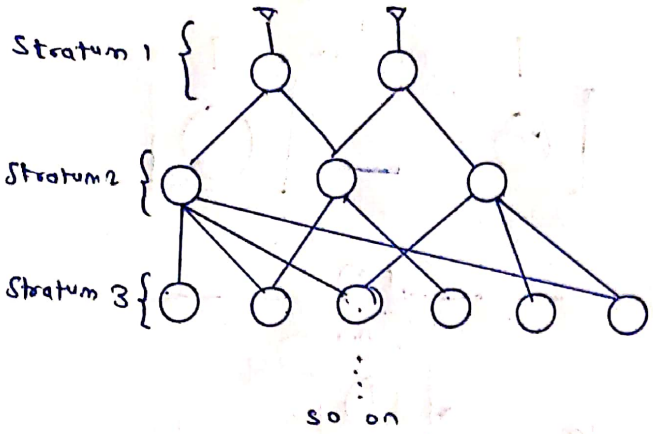


(c)

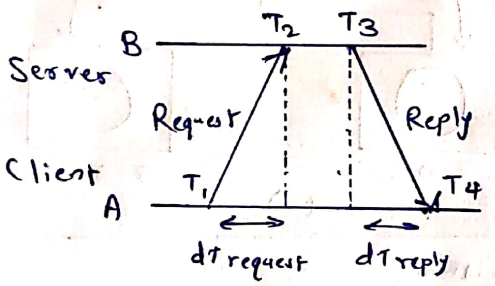
Algorithm:
- Let Tm be the time estimate of master clock.
- Let T[i] contains the time at each i slave at master, where i = 1, ... N.
- If master
  → Send its Tm along with query for t[i]
  → Adjust = sum (t[i] / N)
  i.e., (25 - 10 + 0) / 3 = 5
  → Send offset [i] = Adjust - t[i] to each slave,
  i.e, 5 - (-10) = +15
  5 - 25 = -20
  5 - 0 = +5
  → Slave sends query response at t[i] = T[i] - Tm
  → Time set to, t[i] = t[i] + offset [i].

Network Time Protocol

- NTP operates in hierarchical model.



- NTP is a standard followed ~~by~~ to synchronization clocks on the internet.
- NTP synchronizes all the participating computers within a few milliseconds of coordinated with UTC.
- Survive lengthy losses of connectivity
- Authenticate source of data.
- NTP is a decentralized algorithm.

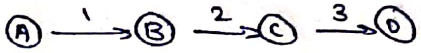


$$RTT = (T_4 - T_1) - (T_3 - T_2)$$

$$offset = \frac{(T_2 - T_1) + (T_3 - T_4)}{2}$$

Logical Clocks

- Logical clocks refer to implementing a protocol on all machines within your distributed systems, so that the machines are able to maintain consistent ordering of events within some virtual time span.
- Example!



Processes must be sequential.

Method 1 : Direct manipulation (Impossible in real world)

Method 2 : Timestamps must be assigned to the events.

↳ Causality : It is an influence by which one event contributes to the production of another event.

↳ Happen - Before Relationship.

$Ts(A) < Ts(B)$

Transitive Relation

If $Ts(A) \rightarrow Ts(B)$ and $Ts(B) \rightarrow Ts(C)$ then $Ts(A) \rightarrow Ts(C)$

Causality Ordered Relation

$a \rightarrow b$

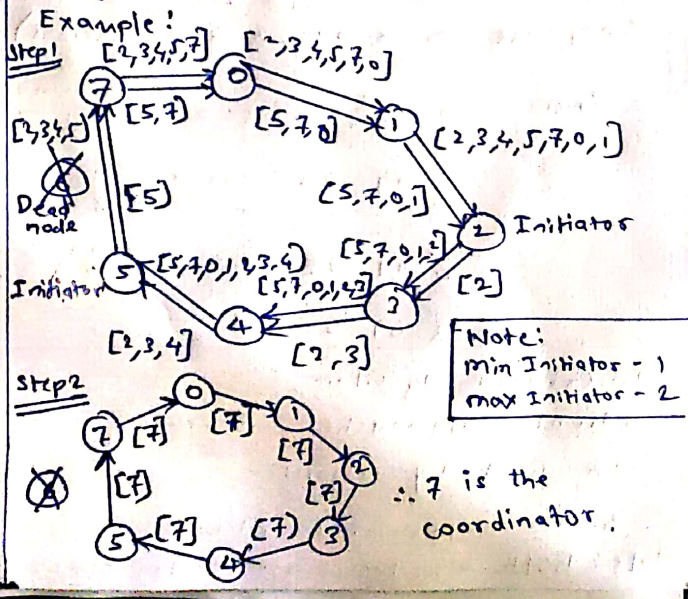
Concurrent event

$a \parallel b$

Ring Algorithm

- When the node notices that coordinator is dead.
- Builds and sends election message to nodes.
- At every step nodes keep on adding its own id at the end of the list.
- The process stops when initiators receives the message it sent.
- After this the node with highest id is declared to be a coordinator.
- Initiator annonces the coordinator by sending message to nodes

Example!

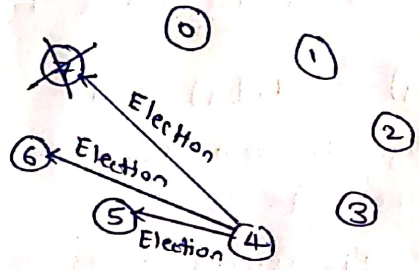


Bully Algorithm

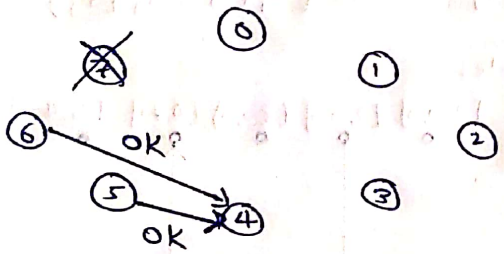
- Many distributed algorithms require a process to act as a coordinator.
- Each process can become coordinator which will be organizing the actions of another processes.
- What if coordinator fails?
- How the new coordinator will be elected?

- Assumptions:
  - Each process has unique id.
  - Each process knows the unique id of another processes.

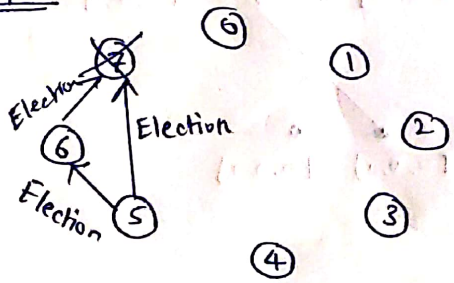
Step 1



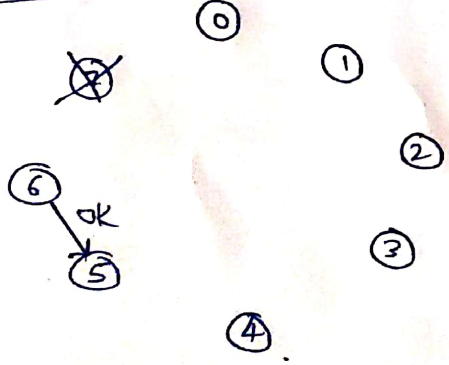
Step 2



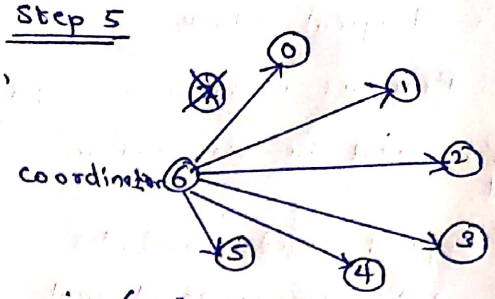
Step 3



Step 4



Step 5



∴ 6 is the coordinator

- A process P notices that coordinator is no longer responding it will initiate an election.
- It will send election to all other process with higher process id.
- If no one responds, process p wins and becomes coordinator.
- If one of the higher one answers, P's job is done and it will take over.
- When process p gets a message from one of its lowered id, receiver sends an ok message to sender that it will take over and its alive and P's job is done.
- Eventually, all processes will give up apart from one, that one is the coordinator.
- The coordinator finally wins and announces its victory by sending coordinator message to all the processes.

Lamport's Logical Clock

Key idea: Processes exchange messages.

- Messages must be sent before received.
- Sent / Request used to order events and synchronize logical clocks.
- Assign sequence numbers to messages
- All coordinating processes can agree on order of events.
- v/s Physical clocks - Time of delay.
- Assume that there is no central time source.

→ Each system has its own local clock.

→ No Total ordering of events.

→ No concept of happened when.

- Lamport's Happened - Before relation if $a \rightarrow b$ [event a happen before event b]

$$a \rightarrow b \rightarrow c$$

$$T(x) < T(y)$$

- Transitive:
  if $a \rightarrow b$ and $b \rightarrow c$

  $\therefore a \rightarrow c$
- Assume clock value to each event. if $a \rightarrow b$ then

  $\therefore$ clock (A) < clock (B)

  $\because$ Time cannot go backwards.
- If a and b occurs on different processes that don't interact with other.

  $\therefore$ Neither $a \rightarrow b$ nor $b \rightarrow a$ are true.

  Such events are concurrent.
  $\therefore$ a || b

$\therefore$ Instead of clock synchronization Lamport's ordering of events can be done.

Vector clocks

- In lamport's clock, if $x \rightarrow y$ then $T(x) < T(y)$.
- But it does not tell about relationship between event x and y.
- Lamport clocks do not capture causality.
- The causal relationship between messages is captured through vector clocks.

Algorithm:

- Vector initialized to zero for each process.

  $V_i[j] = 0$ for $i, j = 1, 2, 3, \ldots N$.
- Increment vector before timestamp event.

  $V_i[j] = V_i[j] + 1$
- Message is sent from P_i with V_i attached to it.
- When P_j receives message

  $V_j[i] = \max [V_j[i], V_j[i]]$

Example: