Lecture #3 Logical Time in Distributed Model of Computation

- Concept of causality (happened-before or happened-after) is fundamental to the design and analysis of distributed and parallel computations

- In a centralized environment such as parallel multiprocessors, causality of events can be tracked on the basis of a physical clock.

- Unfortunately, there is no single physical clock and hence not possible to have a notion of global time.

- This leads us to explore an alternative notion of time which is virtual or logical and is based on the notion of assigning logical time to events so that causal order of event is ensured.

Lecture #3. Week #2

Applications of Logical Time, Clocks, & Ordering.

<u>Review</u>

Clock condition.

For any events a, b: if $a \rightarrow b$ then $C(a) < C(b)$.

Cannot expect the converse condition since that would imply that two concurrent events must occur at the same time.

Clock condition is satisfied if the following two properties hold:

C1. If a and b are events in P_i and a happens-before b at P_i then $C_i(a) < C_i(b)$

C2. If a is send event at P_i and b is a receive event at P_j then $C_i(a) < C_j(b)$.

Implementation Rules.

IR1. Each process P_i increments C_i between any two successive events

IR2. (a) if a is send event at P_i then the message m contains $T_m = C_i(a)$.

(b) upon receiving m at P_j, P_j sets C_j greater than or equal to its present value and greater than T_m.

PARTIAL ORDER of Events.

Total Ordering of Events.

- Use the logical clocks to impose a total order.
- Simply order the events by the logical times at which they occur.
- To break ties, we use an arbitrary total ordering $<$ of processes.

More precisely, define $\Rightarrow$ as follows:

if a is in P_i and b in P_j then $a \Rightarrow b$ if and only if either:

i) $C_i(a) < C_j(b)$, or

ii) $C_i(a) = C_j(b)$ and $P_i < P_j$.

Note that: if $a \rightarrow b$ then $a \Rightarrow b$.

- That is, $\Rightarrow$ relation is a way of completing the "happened-before" partial order to a total order

- Note that $\Rightarrow$ is not unique.

Application of Logical Clocks, $\rightarrow$, and $\Rightarrow$ relations.

Distributed Mutual Exclusion.

1. Set of processes $\{P_0, P_1, \ldots, P_N\}$.

2. One single shared Resource.

Constraint: only one process can use the resource at a time, so the processes must synchronize themselves to avoid conflict.

Find an algorithm such that:

1. A process which has been granted the resource must release it before it can be granted to another process. (Safety)

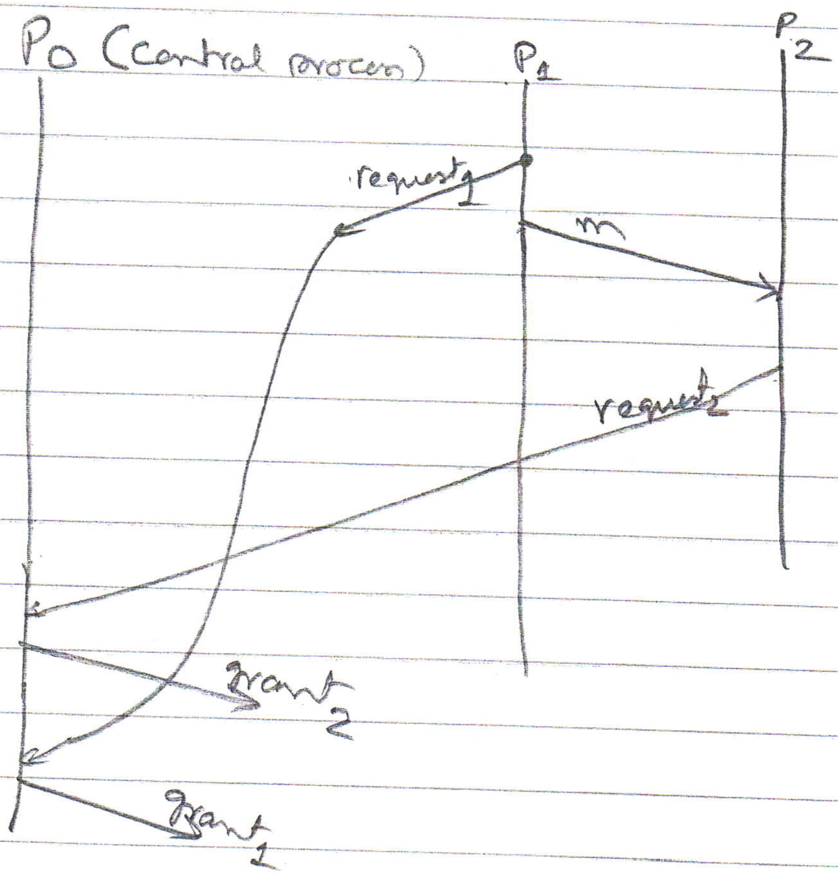
2. Different requests for the resource must be granted in the order in which they are made. (ordering). (fairness)

3. If every process which is granted the resource eventually releases it, then every request is eventually granted (liveness).

Discussion

- A non-trivial problem.

- For example, a central scheduling process which grants requests in the order they are received will not work.



Lamport's Algorithm.

Assumptions:

1. Pair-wise message order preservation: for any two processes P_i and P_j, the messages sent from P_i to P_j are received in the same order as they are sent.

2. No lost messages: Every message is eventually received.

Algorithm

DS: Each process maintains its own request queue

1. Request: P_i sends <T_m: P_i, request resource> to every other process and puts that message on its request queue.

2. When P_j receives <T_m: P_i, request resource>, it places it on its request queue and sends a timestamped ACK to P_i.

3. Release: P_i removes $\langle T_m : P_i, \text{request} \rangle$ from its request queue and sends a timestamped "P_i releases request" message to every other process.

4. When P_j receives a "P_i releases request" message, it removes $\langle T_m : P_i, \text{request} \rangle$ from its request queue.

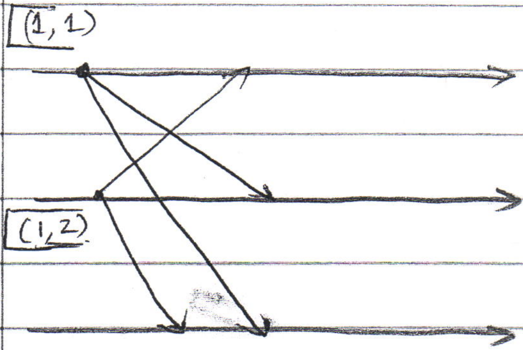
5. P_i is granted the resource when:
   i) $\exists \langle T_m : P_i, \text{request} \rangle$ in its request queue and is ordered before any other request in its queue by $\Rightarrow$.

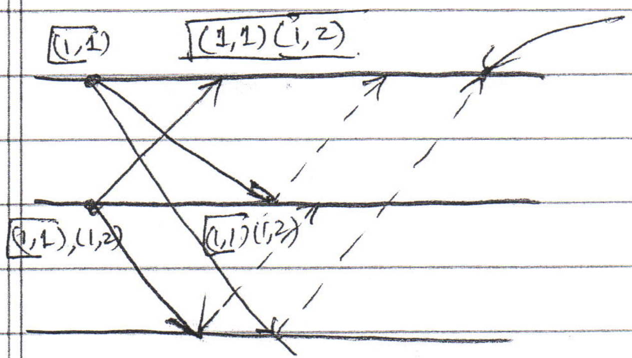
   ii) P_i has received a message from every other process timestamped later that T_m.

<u>Analysis</u>

(ii) of rule 5 + ordered message delivery guarantees that P_i has learned about all requests which preceded its current request.

$\boxed{\text{More discussion}}$





Performance

(N-1) Request

(N-1) ACK.

(N-1) Release.

3(N-1) messages

Theorem 1. Lamport's algorithm is correct.

Proof by contradiction.

Suppose S_i and S_j are in <u>CS</u>.

$\Rightarrow$ 5(i) and 5(ii) hold at both S_i + S_j,

$\Rightarrow$ at t, both S_i + S_j have their own req. at the top.

WLOG, let $\langle T; S_i, req\rangle < \langle T'; S_j, req\rangle$

From 5(ii) + FIFO

at t; S_i's req must be present in request_queue$_j$ @ S_j

$\Rightarrow$ $\boxed{\text{contradiction}}$

Theorem 2. Lamport's algorithm is fair.

→ follows from the ordering of requests in TO order.

Observation.

The system of scalar clocks is not strongly consistent; i.e.,

for any two events e_i and e_j ~~we know~~

$$\text{if } e_i \rightarrow e_j \Rightarrow C(e_i) < C(e_j)$$

however

$$\text{if } C(e_i) < C(e_j) \not\Rightarrow e_i \rightarrow e_j .$$

Can we design a system of clocks that will guarantee the following:

$$\text{if } C(e_i) < C(e_j) \text{ then } e_i \rightarrow e_j$$

and

$$\text{if } e_i \rightarrow e_j \Rightarrow C(e_i) < C(e_j)$$

Vector Time :

appeared in FiMa82.

formalized later.

The time domain is represented by a set of n-dimensional vectors (n processes)

P_i : $vt_i[1..n]$

where

$vt_i[i]$: is the local logical clock of P_i

$vt_i[j]$: represents P_i's latest knowledge of process P_j's local time.

Initially, all vt's are set to $[0, 0, \ldots 0]$.

Use the following rules to update the clock:

R1. Before executing an event, P_i updates its logical time as

$$vt_i[i] = vt_i[i] + d \quad (d > 0)$$

R2. Each message is piggybacked with the vector clock vt of the sender process at sending time. On receipt (m, vt), process p_i executes the following sequence of actions:

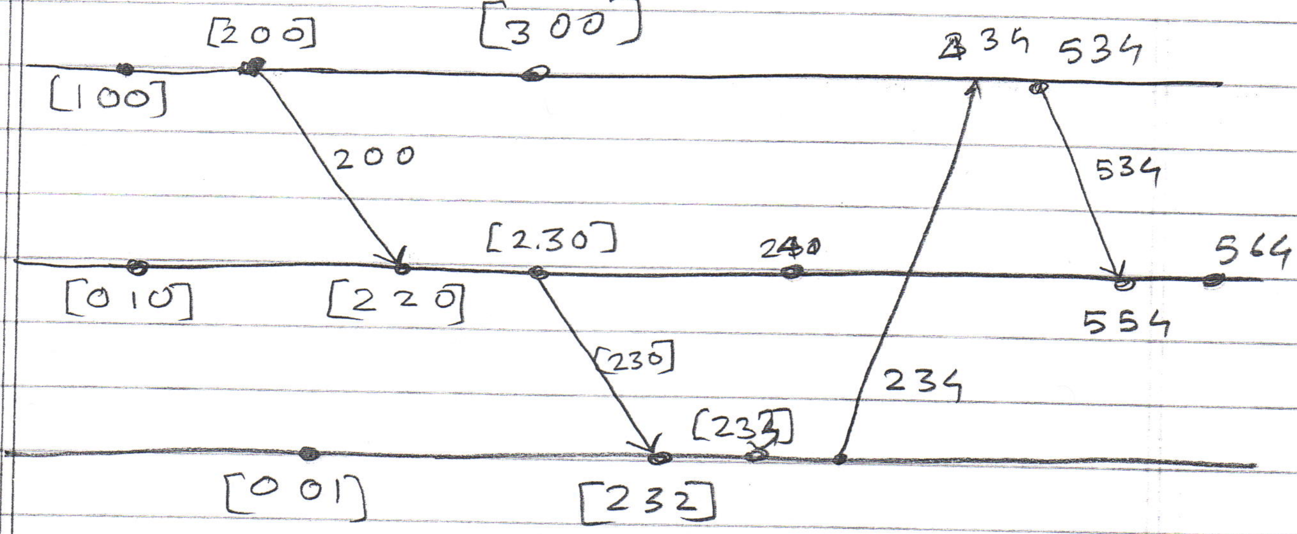
1. update its global logical time as,

$$\forall k : vt_i[k] = \max(vt_i[k], vt[k])$$

explain what this means intuitively.

2. Execute R1.

3. deliver m.



$$vh = vk \iff \forall x : vh[x] = vk[x]$$

$$vh \leq vk \iff \forall x : vh[x] \leq vk[x]$$

$$vh < vk \iff vh \leq vk \wedge \exists x ; vh[x] < vk[x]$$

$$vh \| vk \iff \neg(vh < vk) \wedge \neg(vk < vh)$$