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 physical clocks.

  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.

Applications of Logical Time: Clocks, Ordering.

Review.

Clock condition.
For any events a, b: if \( a \rightarrow b \) then \( c(a) \leq c(b) \).

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 Pi and a happens-before b at Pi then \( c_i(a) \leq c_i(b) \)

C2. If a is a send event at Pi and b is a receive event at Pj then \( c_i(a) \leq c_j(b) \).

Implementation Rules.

IR1. Each process Pi increments Ci between any two successive events.

IR2. (a) If a is a send event at Pi then the message m contains \( T_m = c_i(a) \).
(b) Upon receiving m at Pj, Pj sets Cj 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, use an arbitrary 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.
3.

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 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)

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 \( \langle \text{time}, P_i, \text{request resource} \rangle \) to every other process and puts that message on its request queue.

2. When \( P_j \) receives \( \langle \text{time}, P_i, \text{request resource} \rangle \), it places it on its request queue and sends a time-stamped ACK to \( P_i \).
3. Release: \( P_i \) removes \( <T_{m_i}, P_i, \text{request}> \) 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 \( <T_{m_i}, P_i, \text{request}> \) from its request queue.

5. \( P_i \) is granted the resource when:
   i) \( \exists <T_{m_i}, P_i, \text{request}> \) 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 than \( T_{m_i} \).

Analysis

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

$\Rightarrow$ 5(i) and 5(ii) hold at both $S_i$ and $S_j$.
$\Rightarrow$ at $t$, both $S_i$ and $S_j$ 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 $@ S_j$

$\Rightarrow$ 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

$$e_i \rightarrow e_j \Rightarrow c(e_i) < c(e_j)$$

However

$$c(e_i) < c(e_j) \not\Rightarrow e_i \rightarrow e_j$$

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

$$c(e_i) < c(e_j) \Rightarrow e_i \rightarrow e_j$$

and

$$e_i \rightarrow e_j \Rightarrow c(e_i) < c(e_j)$$
**Vector Time**:  

appeared in FiMi82.
formalized later.

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

\[ P_i : \mathbf{v}_i[1...n] \]

where

\[ \mathbf{v}_i[i] : \text{is the local logical clock of } P_i \]

\[ \mathbf{v}_i[j] : \text{represents } P_i \text{'s latest knowledge} \]

of process \( P_j \text{'s local time}. \)

Initially, all \( \mathbf{v}_i \)'s are set to \([0,0,...,0]\).

Use the following rule to update the clock:

**R2.** Before executing an event, \( P_i \) updates its  
logical time as

\[ \mathbf{v}_i[i] = \mathbf{v}_i[i] + d \quad (d > 0) \]
R2. Each message is piggybacked with the vector clock \( vt \) of the sender process at sending time. On receipt \((m, vt)\), process \(pi\) executes the following sequence of actions:

1. Update its global logical time as:

\[
vt_i[k] = \max(vt_i[k], vt[k])
\]

Explain what this means intuitively.

2. Execute R1.

3. Delete \( m \).
\[ v_h = v_k \implies \forall x : v_h[x] = v_k[x] \]

\[ v_h < v_k \iff \forall x : v_h[x] < v_k[x] \]

\[ v_h < v_k \implies v_h \leq v_k \land \exists x : v_h[x] < v_k[x] \]

\[ v_h \parallel v_k \iff \neg (v_h < v_k) \land \neg (v_k < v_h) \]