Logical Time in Asynchronous Systems

- In a distributed system, it is often necessary to establish relationships between events occurring at different processes:
  - was event $a$ at $P_1$ responsible for causing $b$ at $P_2$?
  - is event $a$ at $P_1$ unrelated to $b$ at $P_2$?

- We discuss the partial ordering relation “happened before” defined over the set of events.


Assumptions:

(i) Processes communicate only via messages.

(ii) Events of each individual process form a totally ordered sequence:

```
 a  b  c  (local) Time
```

(iii) Sending or receiving a message is an event.

Happens Before relation $\rightarrow$

The relation $\rightarrow$ on the set of events of a system satisfies the following three conditions:

(i) if $a$ and $b$ are events in the same process, and $a$ comes before $b$ then $a \rightarrow b$

(ii) if $a$ is sending of a message by one process and $b$ is the receipt of the same message by another process, then $a \rightarrow b$

(iii) if $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$ - transitive

Note: $a \not\rightarrow a$ - irreflexive
Concurrent events

Two distinct events $a$ and $b$ are said to be concurrent ($a \parallel b$) if $a \not\rightarrow b$ and $b \not\rightarrow a$.

$\rightarrow$ defines a partial order over the set of events.

Partial since there could be concurrent events in the set, that by definition are not related by $\rightarrow$.

$a \rightarrow b$ means that it is possible for $a$ to causally affect $b$.

Logical Clocks - assigning numbers to events

A clock $C_i$ for each process $P_i$ is a function which assigns a number $C_i(a)$ to event $a$ in $P_i$.

(a timestamp).

The entire system of clocks is represented by the function $C$ which assigns to any event $b$ the number $C(b)$, where $C(b) = C_j(b)$ if $b$ is an event in $P_j$.

Clock Condition:

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

Satisfying the clock condition

The clock condition can be satisfied if the following two conditions hold:

$CL1$: if $a$ and $b$ are events in $P_i$ and $a \rightarrow b$, then $C_i(a) < C_i(b)$.

$CL2$: if $a$ is the sending of a message by $P_i$ and $b$ is the receipt of that message by $P_j$, then $C_i(a) < C_j(b)$.

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

also? : For any events $a, b$: if $C(a) < C(b)$ then $a \rightarrow b$?
Implementing Logical Clocks

Each process $P_i$ has a counter $C_i$ and $C_i(a)$ is the value contained in $C_i$ when event $a$ occurs.

Implementation Rules:

IR1: each process $P_i$ increments $C_i$ immediately after the occurrence of a local event.

IR2: (i) if $a$ is an event representing the sending of a message $m$ by $P_i$ to $P_j$, then $m$ contains the timestamp $T_m = C_i(a)$
(ii) receiving $m$ by process $P_j$:
   $C_j := \max(C_j, T_m + 1)$
   execute receive($m$) - event $b$ occurs.

Virtual Time

Virtual time, as implemented by logical clocks, advances with the occurrence of events and is therefore discrete. If no events occur, virtual time stops. Waiting for virtual time to pass is therefore risky!

Total Order relation ⇒

Lamport’s Clocks place a partial ordering on events that is consistent with causality.

In order to place a total ordering, we simply use a total order $<$ over process identities to break ties.

If $a$ is an event in $P_i$ and $b$ is an event in $P_j$, define total order relation $\Rightarrow$ by:

$a \Rightarrow b$ iff either (i) $C_i(a) < C_j(b)$
   or (ii) $C_i(a) = C_j(b)$ and $P_i < P_j$

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

Distributed Mutual Exclusion Problem

A fixed set of processes share a single resource. Only one process at a time may use the resource.

Conditions:

(I) A process which has been granted the resource must release it before it is granted to another process.

(II) Different requests must be granted in the order they are made.

(III) If every process that is granted the resource eventually releases it, then every request is eventually granted.
Centralized solution

This allocation violates condition (II) since requestₚ → requestₚ

Distributed Solution using ⇒

Assume point to point FIFO channels between processes P₀ ⋯ Pₙ.

Each process maintains its own request queue. Initially, the queues are empty except for P₀ which currently holds the resource and has the message req:T₀:P₀ where timestamp T₀ is less than the value of any clock.

Algorithm

1. Requesting the resource at process Pᵢ:
   send req:Tₗ:Pᵢ to every other process

2. Receipt of req:Tₗ:Pᵢ at process Pⱼ:
   place in request queue and send ack:Tₗ:Pⱼ to Pᵢ

3. Releasing the resource at process Pᵢ:
   remove any req:Tₗ:Pᵢ from request queue
   send rel:Tₗ:Pᵢ to every other process

4. Receipt of rel:Tₗ:Pᵢ at process Pⱼ:
   remove any req:Tₓ:Pᵢ from request queue

Algorithm (continued)

5. Grant the resource at process Pᵢ:
   if
   (i) there is req:Tₗ:Pᵢ in the request queue which is ordered before any other request by total order ⇒
   (ii) has received a message from every other process time stamped later than Tₗ

continued...
Proof – Mutual Exclusion

By contradiction:
Assume $P_i$ & $P_j$ have been granted the resource concurrently. Therefore 5(i) & 5(ii) must hold at both sites. Implies that at some instant $t$, both $P_i$ & $P_j$ have their requests at the top of their respective queues 5(i). Assume $P_i$'s request has smaller timestamp than $P_j$. By 5(ii) and the FIFO property of channels, at instant $t$, the request of $P_i$ must be present in the queue of $P_j$. Since it has a smaller timestamp it must be at the top of $P_j$'s request queue. However, by 5(i), $P_j$'s request must be at the top of $P_j$'s request queue - a contradiction!

Therefore Lamport's algorithm achieves mutual exclusion.

Communication Complexity

Cycle of acquiring and releasing the shared resource i.e. entering and leaving critical section:

3(n-1) messages = (n-1) request messages + (n-1) acknowledgements + (n-1) release messages

Improved Performance.........

Ricart - Agrawala Mutual Exclusion Algorithm

Optimization of Lamport's algorithm achieved by dispensing with release messages by merging them with acknowledgements.

Communication Complexity: 2(n-1) messages

Ricart - Agrawala Algorithm

1. Requesting the resource at process $P_i$:
   send $\text{req}:T_m:P_i$ to every other process

2. Receipt of $\text{req}:T_m:P_i$ at process $P_j$:
   if $P_j$ has resource, defer request
   if $P_j$ requesting and $\text{req}_j \Rightarrow \text{req}_i$, defer request
   else send $\text{ack}:T_m:P_j$ to $P_i$

3. Releasing the resource at process $P_i$:
   send $\text{ack}:T_m:P_i$ to deferred requests

4. Grant the resource at process $P_i$:
   When got $\text{ack}$ from all other processes.

Ricart - Agrawala Example

P1

P2

P3

P2 granted resource

P1 granted resource

Ricart - Agrawala Proof

By contradiction:
Assume $P_i$ & $P_j$ have been granted the resource concurrently, and that $P_i$'s request has smaller timestamp.

Therefore, $P_i$ received $P_j$'s request after it made its request. $P_j$ can concurrently be granted the resource with $P_i$ only if $P_i$ returns an $\text{ack}$ to $P_j$ before $P_i$ releases the resource.

However, this is impossible since $P_j$ has a larger timestamp.

Therefore, the Ricart-Agrawala implements mutual exclusion.

Limitation of Lamport's Clocks

If $a \rightarrow b$ then $C(a) < C(b)$; however, if $a$ and $b$ are in different processes, then it is not necessarily the case that if $C(a) < C(b)$ then $a \rightarrow b$.

If $C(a) < C(b)$ then $b \not\rightarrow a$; the future cannot influence the past. In general, we cannot say if two events in different processes are causally related or not from their timestamps.
Vector Time

Each process $P_i$ has a vector $V_C_i$ with an entry for each process.

**Implementation Rules:**

IR1: Process $P_i$ increments $V_C_i[i]$ immediately after the occurrence of a local event.

IR2: (i) message $m$ (send event $a$) from $P_i$ to $P_j$, is timestamped with $V_T m = V_C_i(a)$

(ii) receiving $m$ by process $P_j$:

$$\forall k, V_C_j[k] := \max( V_C_j[k], V_T m[k] )$$

execute receive($m$) - event $b$ occurs.

---

Causally Related Events

For two vector timestamps $T_a$ & $T_b$:

- $T_a \neq T_b$ iff $\exists i, T_a[i] \neq T_b[i]$
- $T_a \leq T_b$ iff $\forall i, T_a[i] \leq T_b[i]$
- $T_a < T_b$ iff $( T_a \leq T_b \land T_a \neq T_b )$

Events $a$ and $b$ are causally related, if $T_a < T_b$ or $T_b < T_a$, otherwise they are concurrent.

$a \rightarrow b$ iff $T_a < T_b$.

Vector timestamps represent causality precisely.

---

Example

We can observe that $p_1 || r_3$ since $\neg (1,0,0) < (0,0,3)$ and $\neg (0,0,3) < (1,0,0)$

Also that $r_1 \rightarrow p_3$ since $(0,0,1) < (3,4,1)$
Applications of Causal Ordering

- **Consistent Distributed Snapshots**
  Find a set of local snapshots such that:
  If b is in the union of all local snapshots, and a → b
  then a must be included in the global snapshot too.
  (ie. Consistent snapshots should be left-closed with respect to causality)
  Chandy and Lamport (ACM TOCS, 1985)

- **Causal Ordering of Messages**
  Preserves causal ordering in the delivery of messages in a distributed system.
  Delay delivery (buffer) unless message immediately preceding it has been delivered.
  (eg. For replicated data bases, updates are applied in same order to maintain consistency).
  Birman, Schiper and Stephenson (ACM TOCS, 1991)