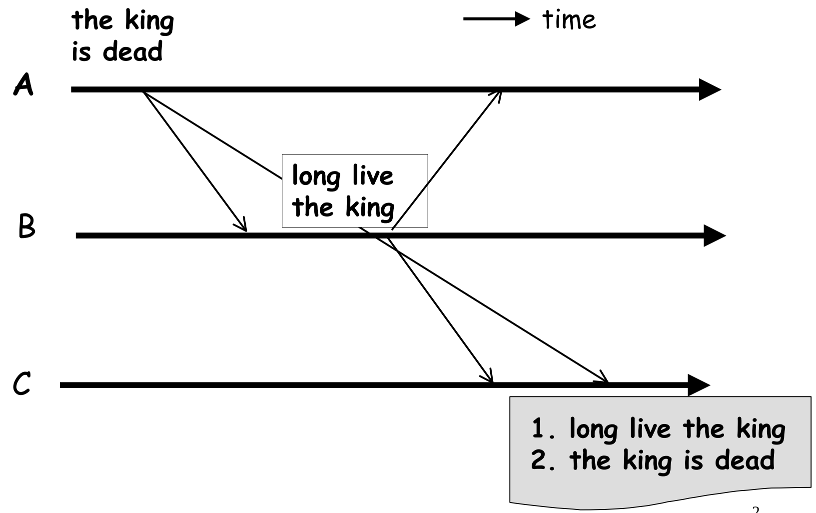
Logical Time in Asynchronous Systems

- 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

"Time, Clocks and Ordering of Events in a Distributed Systems", Leslie Lamport, Comm. ACM, Vol 21, No 7, July 1978, pp 558-565

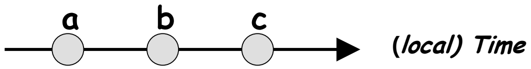
Email example



Assumptions:

(i) Processes communicate only via messages.

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

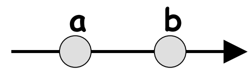


(iii) Sending or receiving a message is an event.

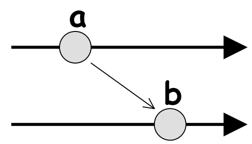
Happens Before relation $\rightarrow$

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*

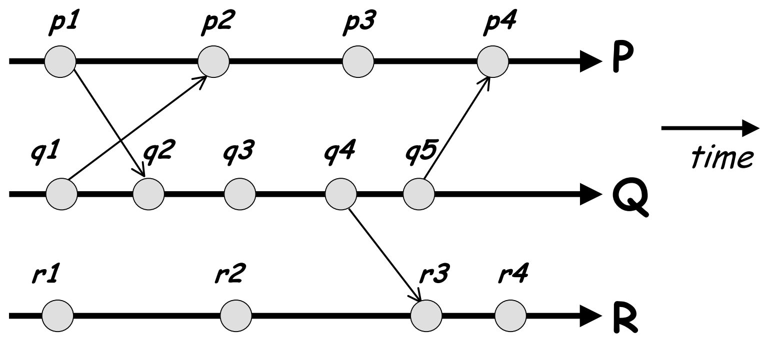
Note: $a \not\rightarrow a$ - *irreflexive*

Concurrent events

Two distinct events ***a*** and ***b*** are said to be **concurrent** (***a* || *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***

Space – Time Diagram



$a \rightarrow b$: path from ***a*** to ***b*** in the diagram moving forward in time along the process and message lines.

$p1 \rightarrow r4$, $q4 \rightarrow r3$, $p2 \rightarrow p4$, $q3 \,||\, p3$, $q3 \,||\, r2$

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.

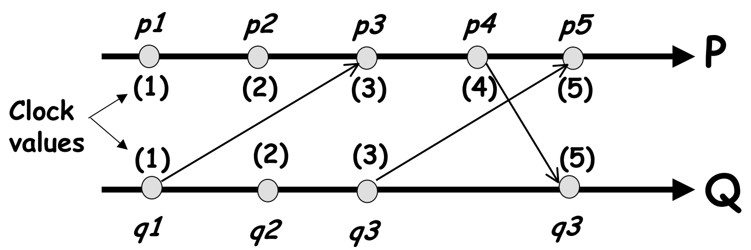
IR2: (i) if a is an event representing the sending of a message m by P_i to P_j, then m contains the timestamp **Tm** = $C_i(a)$

(ii) receiving m by process P_j :

C_j **:= max(** C_j**,Tm+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 ⇒ by:

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

$\boxed{\text{Note : if } a \rightarrow b \text{ then } a \Rightarrow b}$

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.

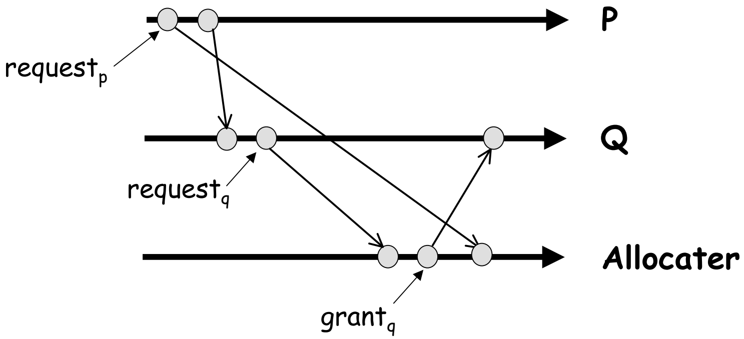
(II) Different requests must be granted in the order they are made.

Safety

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

Liveness

Centralized solution

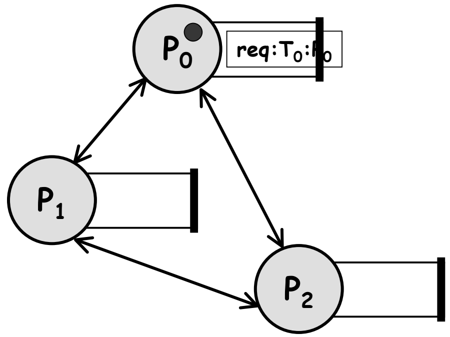


This allocation violates condition (II) since $request_p \rightarrow request_q$

Distributed Solution using ⇒

Assume point to point FIFO channels between processes P_0 .. P_n.

Each process maintains its own ***request queue***. Initially, the queues are empty except for P_0 which currently holds the resource and has the message **req**:T_0:P_0 where timestamp T_0 is less than the value of any clock.



Algorithm

1. Requesting the resource at process P_i:
   send **req**:T_m:P_i to every other process

2. Receipt of **req**:T_m:P_i at process P_j:
   place in request queue and send **ack**:$T_{m'}$:P_j to P_i

3. Releasing the resource at process P_i:
   remove any **req**:T_m:P_i from request queue
   send **rel**:$T_{m'}$:P_i to every other process

4. Receipt of **rel**:T_m:P_i at process P_j:
   remove any **req**:T_x:P_i from request queue

continued…

Algorithm (continued)

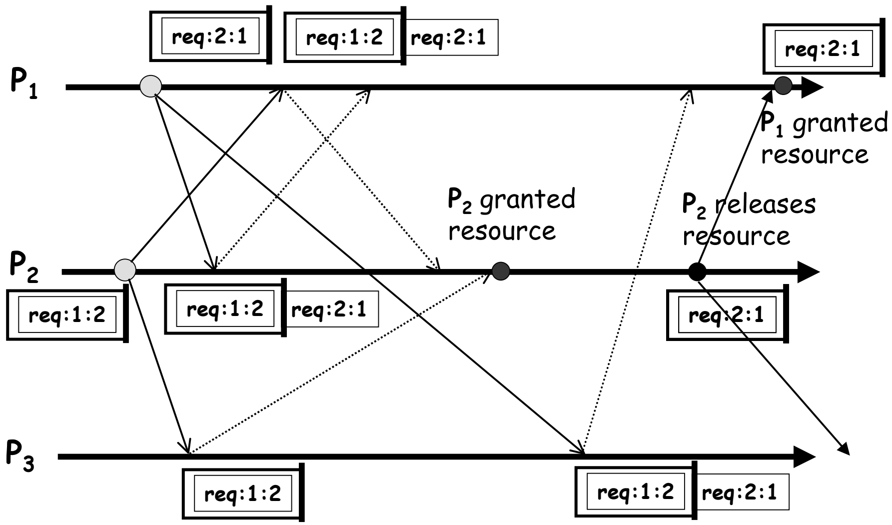
5. Grant the resource at process P_i:

if

(i) there is **req**:T_m:P_i 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 T_m

Example



Proof – Mutual Exclusion

By contradiction:

Assume **Pi** & **Pj** have been granted the resource concurrently. Therefore 5(i) & 5(ii) must hold at both sites.

Implies that at some instant *t*, both **Pi** & **Pj** have their requests at the top of their respective queues 5(i).

Assume **Pi**'s request has smaller timestamp than **Pj.**
By 5(ii) and the FIFO property of channels, at instant *t*, the request of **Pi** must be present in the queue of **Pj.**
Since it has a smaller timestamp it must be at the top of **Pj**'s request queue.

However, by 5(i), **Pj**'s request must be at the top of **Pj**'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

G. Ricart and A.K. Agrawala, "An Optimal Algorithm for Mutual Exclusion in Computer Networks", Comm ACM, Jan 1981.

Ricart – Agrawala Algorithm

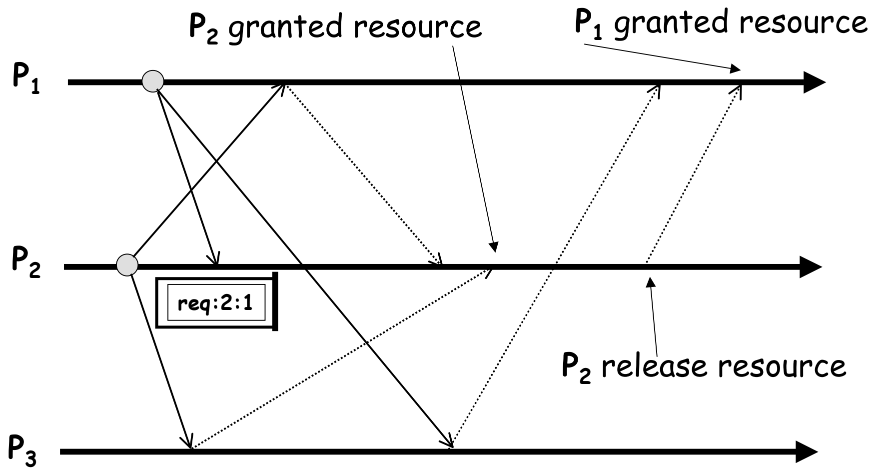
1. Requesting the resource at process P_i:
   send **req:T_m:P_i** to every other process

2. Receipt of **req:T_m:P_i** at process **P_j:**
   if **P_j** has resource, defer request
   if **P_j** requesting and **req_j** $\Rightarrow$ **req_i**, defer request
   else send **ack:$T_{m'}$:P_j** to P_i

3. Releasing the resource at process **P_i**:
   send **ack:T_m:P_i** to deferred requests

4. Grant the resource at process **P_i**:
   When got **ack** from all other processes.

Ricart – Agrawala Example



Ricart – Agrawala Proof

By contradiction:
Assume **Pi** & **Pj** have been granted the resource concurrently, and that **Pi**'s request has smaller timestamp.

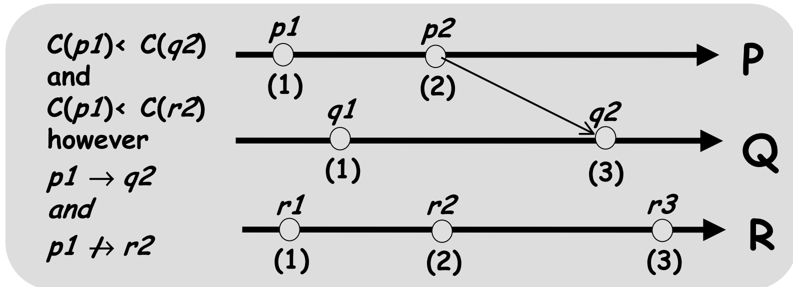
Therefore, **Pi** received **Pj**'s request after it made its request. **Pj** can concurrently be granted the resource with **Pi** only if **Pi** returns an **ack** to **Pj** before **Pi** releases the resource.

However, this is impossible since **Pj** 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 VC_i with an entry for each process.

Implementation Rules:

IR1: Process $\mathbf{P_i}$ increments $\mathbf{VC_i}[i]$ immediately after the occurrence of a local event.

IR2: (i) message ***m*** (send event ***a***) from $\mathbf{P_i}$ to $\mathbf{P_j}$, is timestamped with $\mathbf{VTm} = \mathbf{VC_i}(\boldsymbol{a})$

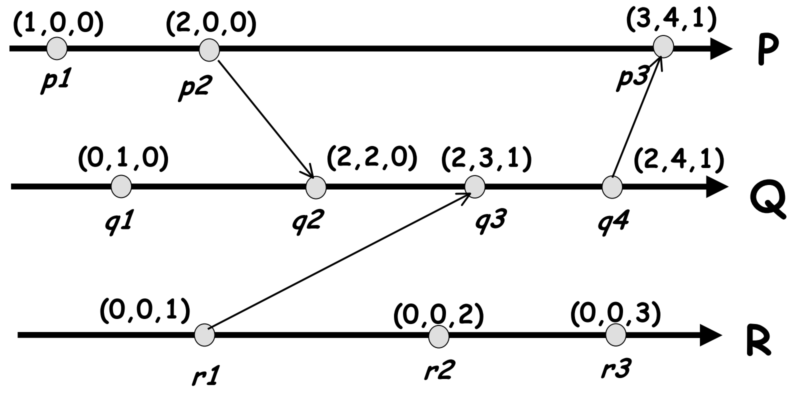
(ii) receiving ***m*** by process $\mathbf{P_j}$:

$\forall k$, $\mathbf{VC_j}[k] := \mathbf{max}(\ \mathbf{VC_j}[k], \mathbf{VTm}[k]\ )$

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

Mattern (Proc. of Int. Conf. on Parallel and Dist, Algorithms, 1988)
Fidge (Proc. of 11th Australian Computer Sc. Conf. 1988)

Example



Causally Related Events

For two vector timestamps **Ta** & **Tb:**

$\mathbf{Ta} \neq \mathbf{Tb}$ iff $\exists i$, $\mathbf{Ta}[i] \neq \mathbf{Tb}[i]$

$\mathbf{Ta} \leq \mathbf{Tb}$ iff $\forall i$, $\mathbf{Ta}[i] \leq \mathbf{Tb}[i]$

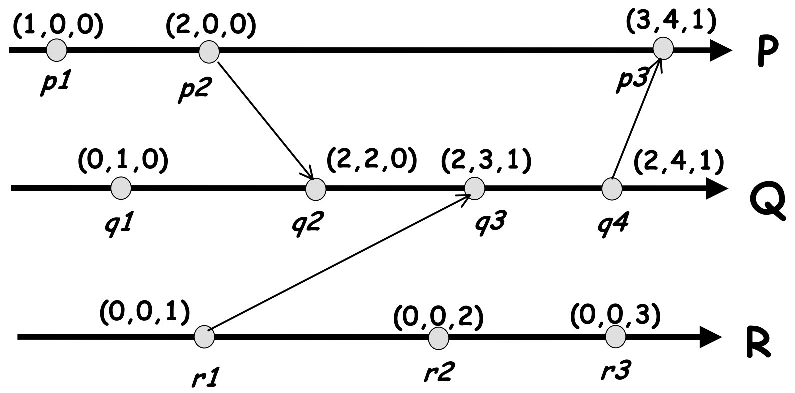
$\mathbf{Ta} < \mathbf{Tb}$ iff $(\ \mathbf{Ta} \leq \mathbf{Tb} \wedge \mathbf{Ta} \neq \mathbf{Tb}\ )$

Events **a** and **b** are causally related, if **Ta** < **Tb** or **Tb** < **Ta**.
Otherwise they are concurrent.

$\boldsymbol{a} \rightarrow \boldsymbol{b}$ iff $\mathbf{Ta} < \mathbf{Tb}$.

Vector timestamps represent causality precisely.

Example



We can observe that p1 || r3
since $\neg$ (1,0,0) < (0,0,3)
and $\neg$ (0,0,3) < (1,0,0)

Also that r1 $\rightarrow$ p3
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 $\rightarrow$ 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)