# Time, Clocks, and the Ordering of Events in a Distributed System

Leslie Lamport

Presentation: Yunyun Zhu

Read Group Seminar Apr 13rd, 2012

# Introduction

- Distributed system definition:
  - A collection of distinct *processes* which are spatially separated and which communicate with one another by exchanging messages.

- Distributed system examples:
  - A banking system
  - A tsunami warning system

# Introduction

- *Event*: the execution of a subprogram on a computer,  or the execution of a machine instruction
- Each process consists of a *sequence* of events
- No global clock → hard to judge which event happens earlier in a distributed system

# "Happened before" relation

- A partial order relation (defined as $\rightarrow$)
  - If event a and event b are in the same process and a comes before b, then $a \rightarrow b$
  - If a is the sending of a message by one process and b is the receipt of that message by another process, then $a \rightarrow b$
  - If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$

Note: a and b are *concurrent* if

$$a \nrightarrow b \text{ and } b \nrightarrow a$$

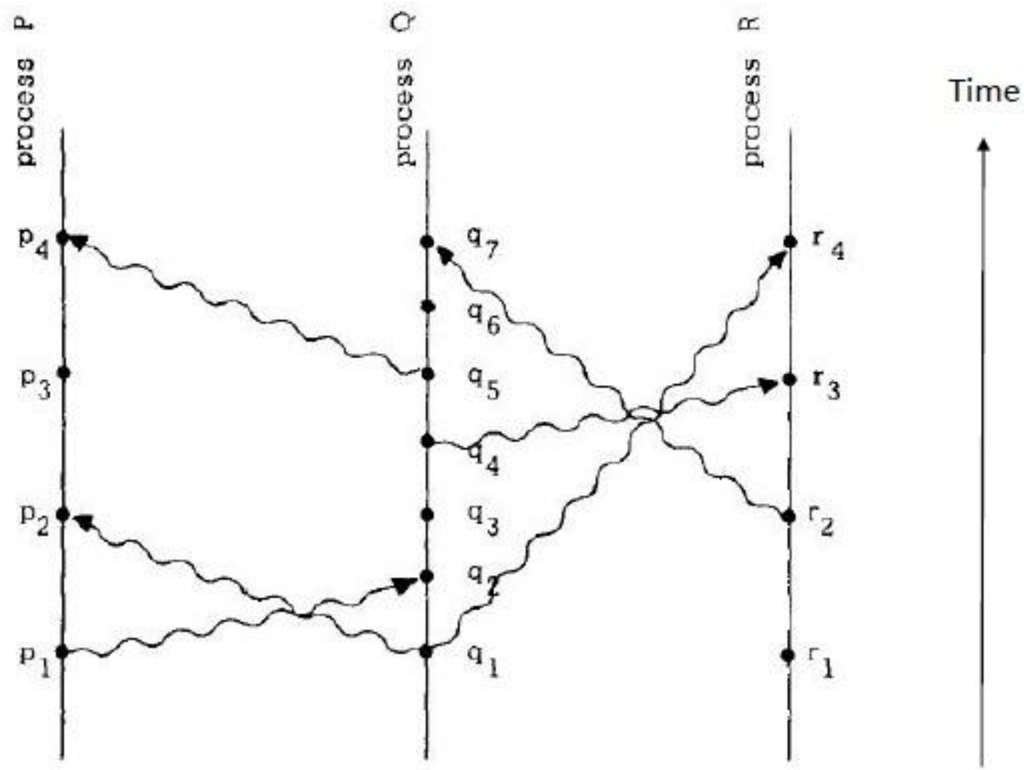
# Examples

p1 → q2

r2 → r3

p1 → r4
(via q2, q4 and r3)

p3 and q3 are concurrent



# Logical clocks

- Clock: assigning a number to an event
- Each process $P_i$ has a logical clock $C_i$
- $C_i(a)$: number assigned to a in $P_i$
- No relation to physical clocks

# Logical clock condition

- Clock Condition (which means the system of clocks are correct):
  - For any events *a*, *b*: if a → b then C(a) < C(b)
    (*If event a occurs before event b then a should happen at an earlier time than b*)

- Two conditions should hold to satisfy the Clock Condition:
  - **C1.** If *a* and *b* are events in process $P_i$ and *a* comes before *b*, then $C_i(a) < C_i(b)$
  - **C2.** If *a* is the sending of a message by process $P_i$ and *b* is the receipt of that message by process $P_j$ then $C_i(a) < C_j(b)$

# Implementation Rules

- IR1 (for C1). Clock $C_i$ must be increased between any two successive events in process $P_i$: $C_i := C_i + 1$

- IR2 (for C2). (a) If event *a* is the sending of a message *m* by process $P_i$, then the message *m* contains a timestamp $T_m = C_i(a)$

- IR2 (for C2). (b) When the same message *m* is received by a different process $P_j$, $C_j$ is set to a value greater than the current value of the counter and the timestamp carried by the message:

  $C_j := \max(C_j, T_m) + 1$
- Example on blackboard

# Ordering the Events Totally

- Break ties by a total ordering of the processes
- Total ordering of events ($a \Rightarrow b$)
- If a is an event in process $P_i$ and b is an event in process $P_j$, then $a \Rightarrow b$ if either
  - $C_i(a) < C_j(b)$, or
  - $C_i(a) = C_j(b)$ and $P_i \prec P_j$, where $\prec$ is an arbitrary relation that totally orders the processes to break ties.
- Example on blackboard

# A mutual exclusion problem

- A distributed system obtaining the total ordering
- Specification:
  - A collection of processes sharing a single resource
  - Only one process uses the resource at a time
- Requirements
  - The resource must be released by the current process first before it is granted to another one
  - Messages are delivered in FIFO order

# Lamport's algorithm

- Requesting resource
  - $P_i$ sends REQUEST($ts_i$, i) to every other process and puts the request on $request\_queue_i$, where $ts_i$ denotes the timestamp of the request
  - When $P_j$ receives REQUEST($ts_i$, i) from $P_i$ it returns a timestamped REPLY to $S_i$ and places $S_i$'s request on $request\_queue_j$
- $P_i$ is granted the Resource when
  - L1: $P_i$ has received a message from every other process timestamped later than $P_i$'s request($ts_i$, i)
  - L2: $P_i$'s request ($ts_i$, i) is at the top of $request\_queue_i$ by the relation $\Rightarrow$

# Lamport's algorithm

- Releasing resource
  - $P_i$ removes request from top of request_queue$_i$ and sends timestamped RELEASE message to every other process
  - When $P_j$ receives a RELEASE messages from $S_i$ it removes $S_i$'s request from request_queue$_j$

- Example on blackboard

# Proof of Correctness

- Mutual exclusion achieved
- Proof is by contradiction. Suppose $P_i$ and $P_j$ are occupying the resource concurrently, which implies conditions L1 and L2 hold at both of the processes concurrently.
- This means that at some instant in time, say t, both $P_i$ and $P_j$ have their own requests at the top of their request queues and condition L1 holds at them. Assume that $P_i$ 's request is ordered before than the request of $P_j$ by the relation $\Rightarrow$.
- From condition L1 and that messages are delivered FIFO, it is clear that at instant t the request of $P_i$ must be present in request $queue_j$ when $P_j$ was occupying the resource. This implies that $P_j$ 's own request is at the top of its own request queue when an earlier request, $P_i$ 's request, is present in the request $queue_j$ – a contradiction!

# Performance

- For each procedure of occupying a resource, Lamport's algorithm requires (N − 1) REQUEST messages, (N − 1) REPLY messages, and (N − 1) RELEASE messages.
- Thus, Lamport's algorithm requires 3(N − 1) messages per procedure of occupying a resource.
- Synchronization delay in the algorithm is T.

# An optimization

- REPLY messages can be omitted sometimes. For example, if $P_j$ receives a REQUEST message from $P_i$ after it has sent its own REQUEST message with timestamp higher than the timestamp of $P_i$'s request, then $P_j$ need not send a REPLY message to $P_i$.
- This is because when $P_i$ receives $P_j$'s request with timestamp higher than its own, it can conclude that $P_j$ does not have any smaller timestamp request which is still pending.
- With this optimization, Lamport's algorithm requires between $3(N - 1)$ and $2(N - 1)$ messages for a procedure of occupying the resource.