CSC2/458 Parallel and Distributed Systems
More Clocks and Mutual Exclusion

Sreepathi Pai

March 27, 2018

URCS

Outline

More on Total Order Multicast using Logical Clocks

Vector Clocks

Mutual Exclusion

Outline

Total-ordered multicast

- Each client multicasts a message to all replicas
  - Each message is timestamped according to local logical clock
  - Assume no loss of messages
  - Assume reliable ordering
- Each replica places received messages in a queue
- Each replica acknowledges receipt of messages using a multicast
- Each replica processes messages in order of their timestamps
  - Only when it has received acknowledgement for that message from all other replicas

This protocol ensures all processes see the same queue.

Invariants

- Process a message if:
  - it has been acknowledged by all other processes
- If multiple such messages exist:
  - process them in sender-timestamp order

Empty queue

Consider yourself to be a process.

- Your queue is empty
  - What do you do?

Queue with a message

- Your queue contains a single message
  - (or multiple messages)
- But no acknowledgements
  - What do you know?
  - What do you do?

Queue with acknowledgements

- Your queue contains only acknowledgements
- But no messages
  - What do you know?
  - What do you do?

Queue with message + all acknowledgements

- Your queue contains a message and all its acknowledgements
  - no other message (if any) has all its acknowledgements
  - What do you know?
  - What do you do?
- What if a message without its acknowledgements has a lower sender-timestamp?

Could it happen?

- Your queue contains a message A and all its acknowledgements
  - and no other message or acknowledgements
- Some other process contains a message B and all its acknowledgements
  - and no other message or acknowledgements

Checking all possible states of a finite state machine

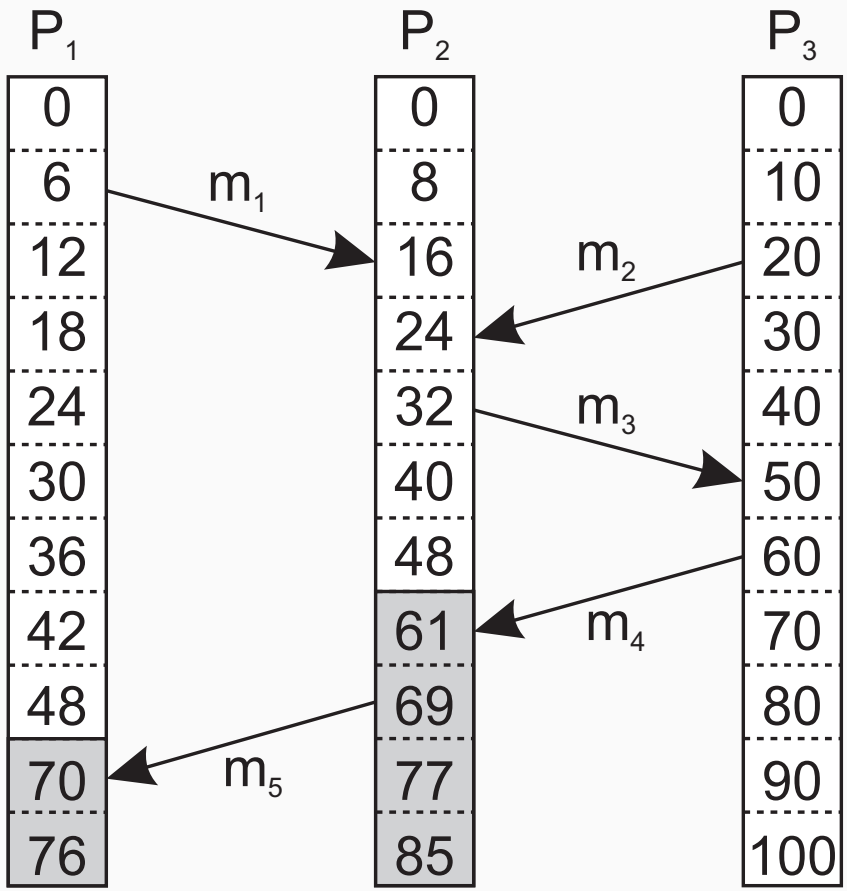
- A formal method called *model checking*
- Used by Amazon (among others)
    - How Amazon Web Services uses Formal Methods, CACM 48(4)
- Tools, TLA+ and TLC
    - The TLA home page

Outline

Causality

- Consider a messaging board where messages and replies are multicast (or broadcast)
- Messages must appear to everybody before their replies
- I.e. Replies are "caused by" messages
- In logical clocks:
  - $a \rightarrow b$ implies $C(a) < C(b)$
  - but $C(a) < C(b)$ does not imply $a \rightarrow b$

Example



- Are m_1 and m_2 causally related?
  - note: maybe better to read: did m_1 happen before m_2?

Easy way

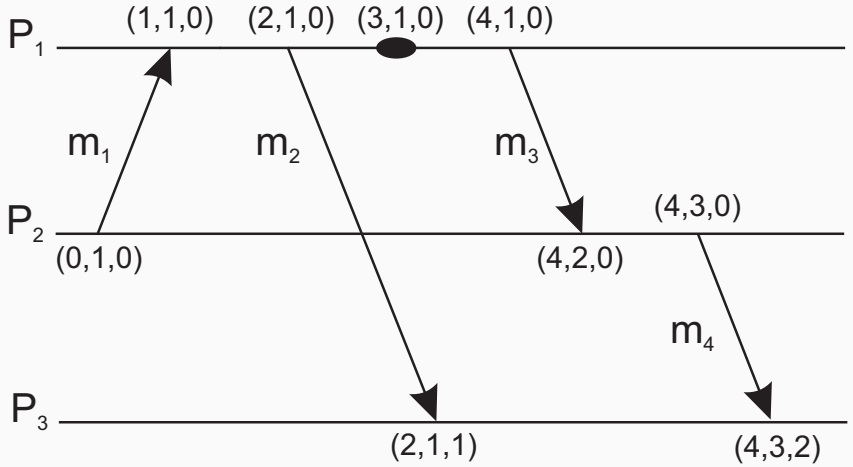
- Each message carries a list of all messages seen by sender
  - Causal history
- Easy to see when messages are not causally related
  - If *b* happened after *a*, it must have seen *a*
  - See textbook for formal definition

Vector Clocks

- Encode global knowledge into timestamps
- Each timestamp $ts(m)$ for message m is now a vector (i.e. an array)
  - Contains n items for n processes
  - $V_i[j]$ is vector clock at process i, containing last known timestamp at process j
  - $V_i[i]$ is incremented every time an event is generated (i.e. it is like i's local clock)
- Importantly, $V_i[j] = k$ means that process i knows k events have happened at process j
- Update:
  - $V_i[k] = \max\{V_i[k], ts(m)[k]\}$ for all k

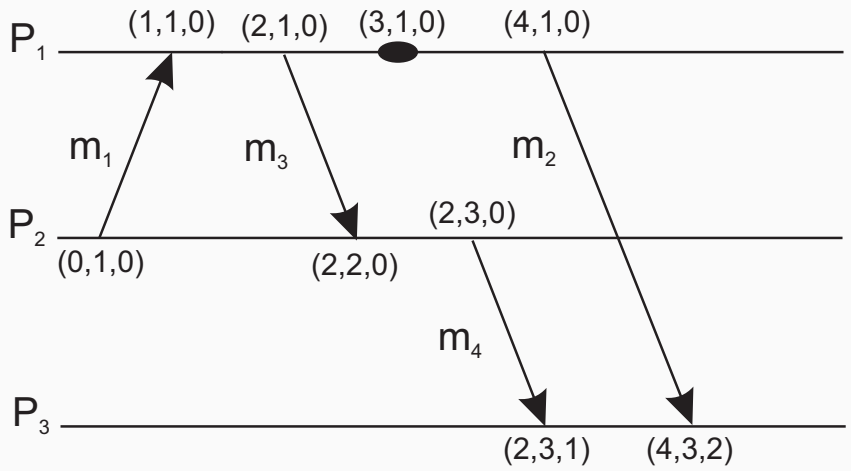
Example: Determining ordering

- Define $ts(a) < ts(b)$ for messages a and b if and only:
  - $ts(a)[k] \leq ts(b)[k]$ for all k
  - $ts(a)[k'] < ts(b)[k']$ for some k'
- Did m_2 happen before m_4?

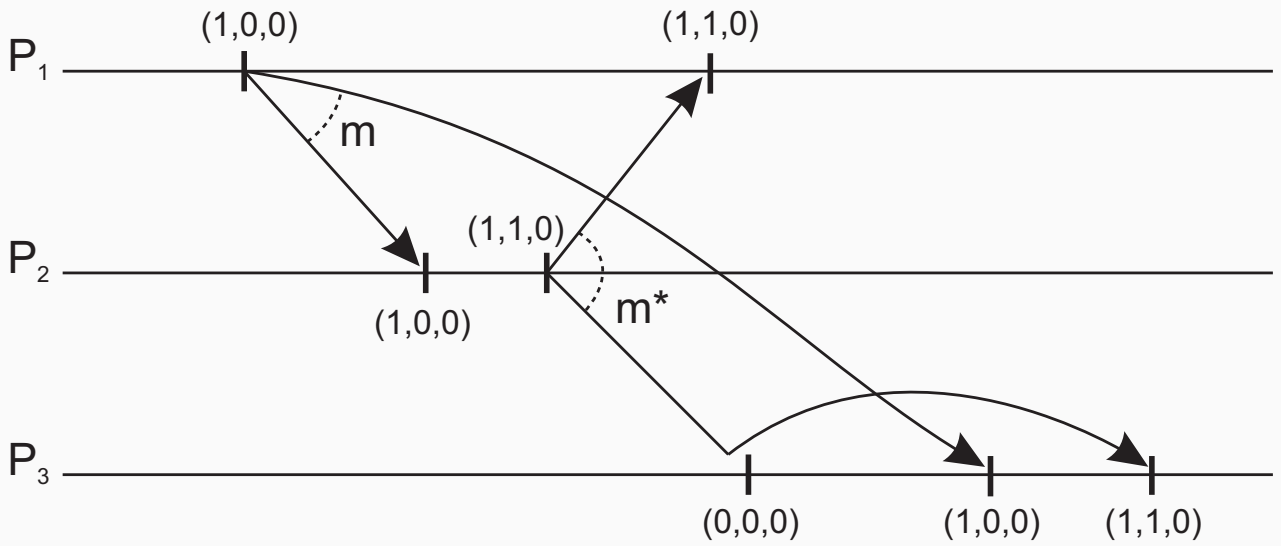


Example: Determining ordering – contd

- Did m_2 happen before m_4?



Causal-ordered Multicast Board



Outline

Centralized Mutual Exclusion

- One Coordinator
- All processes *Request* exclusive access from Coordinator
- Coordinator
  - *Grants* access if no other process is requesting the same resource
  - does not reply if another process is granted resource
  - places request in queue
- Process
  - blocks waiting for reply from Coordinator
  - accesses resource on grant from Coordinator
  - *Releases* resource by informing Coordinator
- Coordinator
  - on release, informs next process in queue that requested resource

Evaluating Centralized Mutual Exclusion

- Scalability
  - Single coordinator may become performance bottleneck
- Availability
  - Single coordinator may crash
  - What about process crashes?
- Number of messages
  - to enter critical section: 2

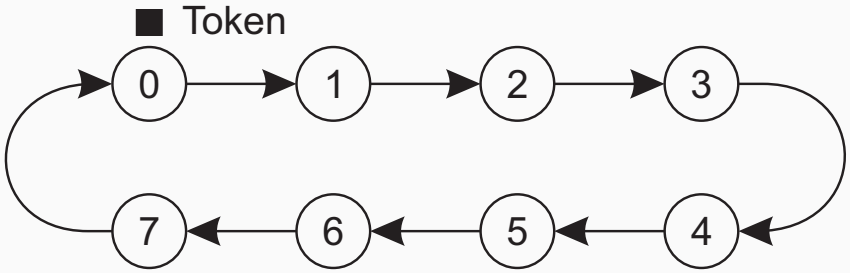
Mutual Exclusion using Totally ordered Multicasts

- Total ordered multicast produces a total order among all messages
- Can be used to implement mutual exclusion
- Messages:
  - ENTER: process multicasts that it wants to enter a critical section
  - ALLOW: process unicasts permission to ENTERing process
  - RELEASE: process multicasts that it has left a critical section

Evaluating Totally Ordered Multicasts

- Scalability
  - No single coordinator
  - But what about requiring permission from *everybody*?
- Availability
  - What if a process crashes?
- Number of Messages
  - to enter critical section?
- Multicasts and complexity
  - what if there is no multicast primitive?

Token Ring Mutual Exclusion



- Construct ring overlay (i.e. logical) network
  - Has no relation to physical network
  - how to construct this?
- Generate token
- On receiving token
  - Optionally, perform accesses to any shared resources
  - Pass token to neighbour

Evaluating Token Ring Mutual Exclusion

- Scalability
  - No centralized coordinator
- Availability
  - What if token is lost?
  - What if a process not holding a token crashes?
  - What if a process holding a token crashes?
- Number of messages
  - to enter critical section: $N - 1$ (max.)

Decentralized Mutual Exclusion using Voting

- Replicate resource N times
- Each replica controlled by different coordinator
- When a process requests access to a resource
  - It must get permission from more than $N/2$ coordinators (does it need to wait for all coordinators?)
  - Coordinators may refuse to give access if they've already given access
  - A process that is refused access sends releases to coordinators it got access from and will backoff and retry after some time
- Of interest, a coordinator may crash and "forget" it had given access
  - Incorrectly give access to a process
  - When will this cause a problem?

Evaluating Mutual Exclusion using Voting

- Scalability
  - Multiple centralized coordinators, only require majority
- Availability
  - Probabilistic arguments against all coordinators crashing
  - What about processes holding locks?
- Utilization
  - Does at least one process that competes for a resource get it?
- Number of messages
  - to enter a critical section: ?