Synchronization Algorithms

Reference for study:

Van Steen, Tanenbaum, "Distributed Systems", chapter 6

Synchronization

- Processes in a DS are asynchronous
- Sometimes, we need to synchronize them
  - a process must wait until another process completes another operation
  - some ordering of events must be enforced
  - some timing requirements must be satisfied
- Two possible approaches for decentralized systems:
  - synchronize physical clocks
  - logical clocks

Physical Clock Synchronization

- Different possibilities exist
  - UTC Receivers
    - synchronization can be very accurate (up to 50ns precision) but expensive
  - NTP
    - cheaper solution but not so accurate

	NTP accuracy
LAN	<1ms
public internet	10-50ms
public internet under congestion	>100ms

Logical Clocks

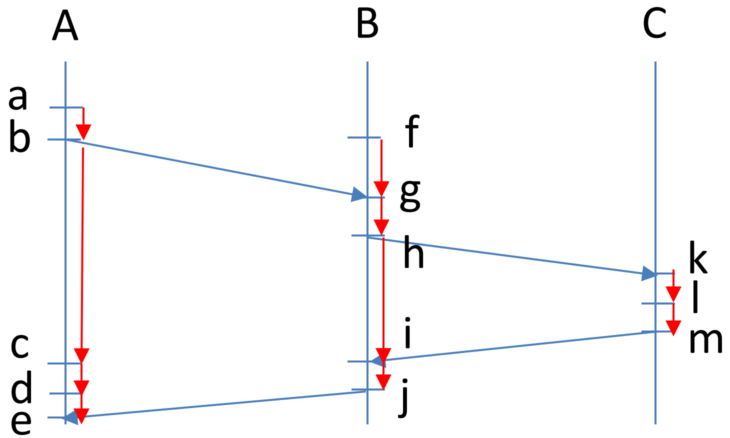
- For synchronization purposes, agreement on **ordering of events** is often sufficient
- Logical clocks are distributed algorithms that can be used to achieve this sort of agreement

The Happens-before Relation in a DS

- Definition:

  x → **y** event **x** happens before event **y**

- Baseline: in a distributed system based on message passing, **x** → **y** can be observed in these cases:
  - **x** happens before **y** in the same process

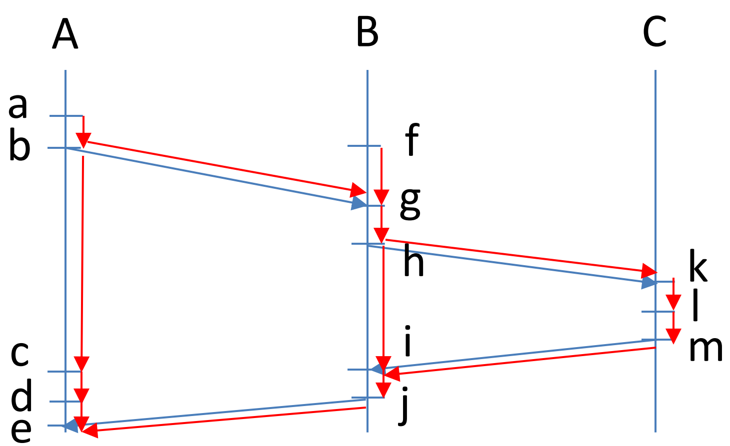


The Happens-before Relation in a DS

- Definition: the Happens-before relation

  x → **y**   event **x** happens before event **y**

- Baseline: in a distributed system based on message passing, **x** → **y** can be observed in these cases:
  - **x** happens before **y** in the same process
  - **x** and **y** are the events of sending and receiving the same message (by different processes)
  - transitive property holds



Happens-before and Causality

- The happens-before relation also captures the possibility of **causal relation** (causality) between two events
  - if $\mathbf{x} \rightarrow \mathbf{y}$ then **x** and **y** **may be** causally related

    in cause-effect relation (**x** cause, **y** effect) OR **y** depends on/is influenced by **x**

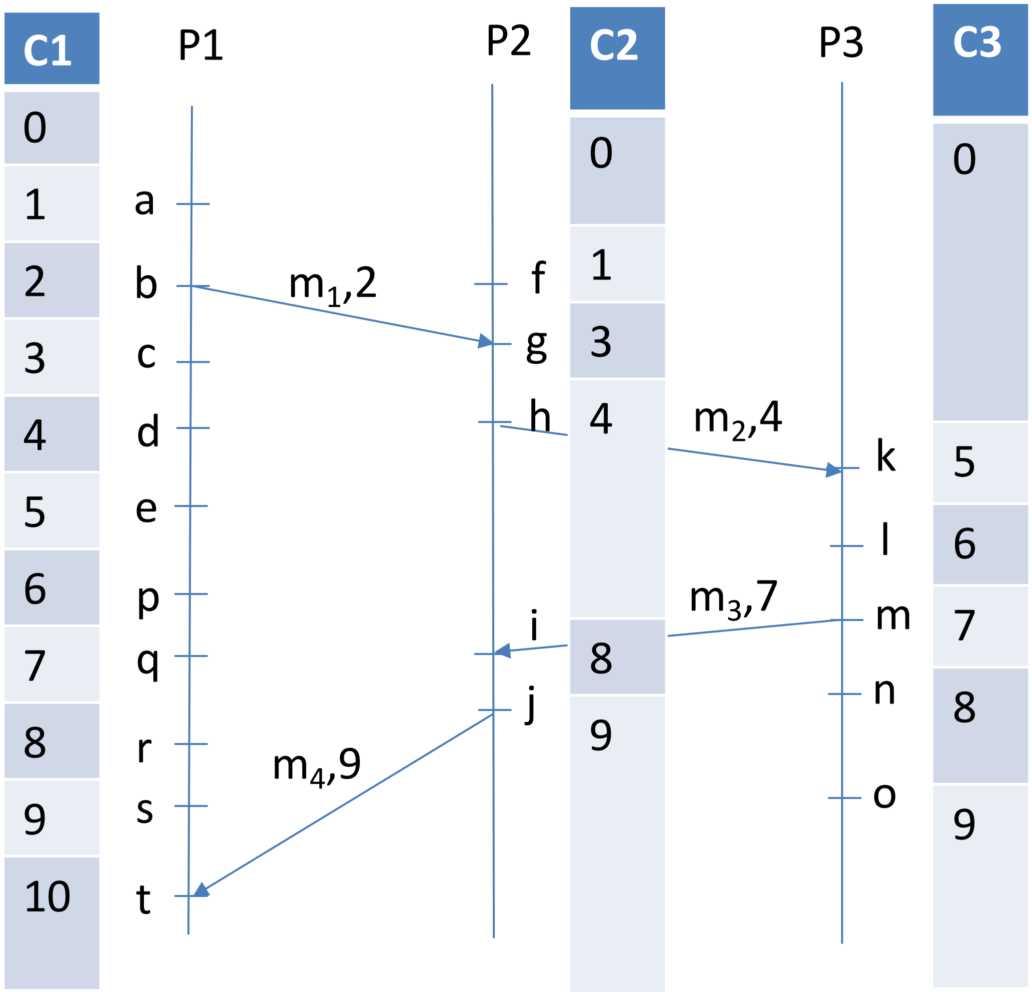
  - if $\mathbf{x} \not\rightarrow \mathbf{y}$ and $\mathbf{y} \not\rightarrow \mathbf{x}$ then **x** and **y** **are not** causally related (we say they are **concurrent**)

Lamport Clocks

- Goal: assign timestamps (i.e., clock values) to events in such a way that if $\mathbf{x} \rightarrow \mathbf{y}$ then $C(\mathbf{x})<C(\mathbf{y})$
- Algorithm:
  - each process Pi keeps a local time counter Ci
  - Pi timestamps each local event **x** with the current Ci value Ci(**x**), after having incremented Ci
  - Pi timestamps each message it sends with the timestamp Ci(**s**) assigned to the send event **s**
  - when Pi receives a message with timestamp Cr,
    - if Cr>Ci, Pi sets Ci=Cr (clock adjustment) and then it timestamps the receive event as usual

equivalent to Ci=max(Ci,Cr)

Example

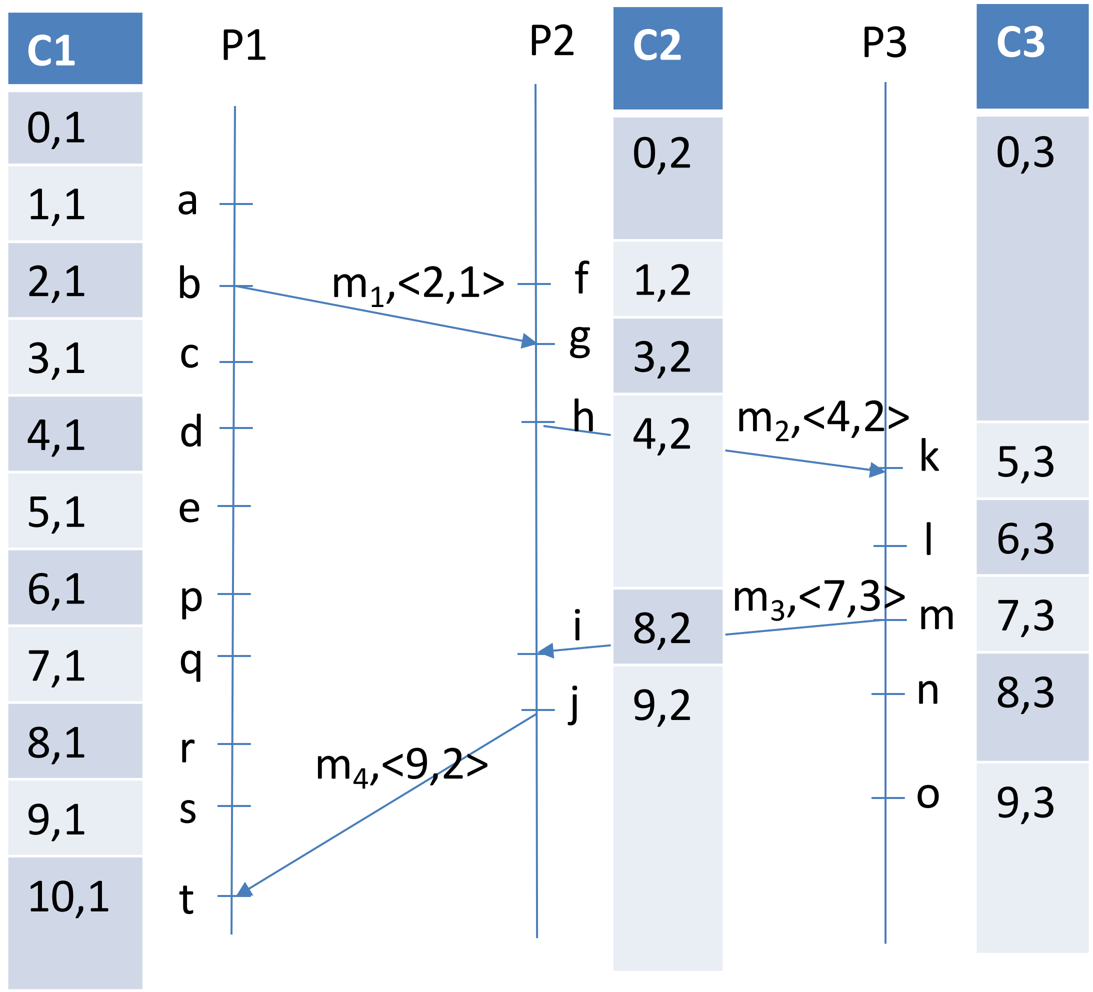


C1
P1
P2
C2
P3
C3
0
1
2
3
4
5
6
7
8
9
10
a
b
c
d
e
p
q
r
s
t
f
g
h
i
j
k
l
m
n
o
m_1,2
m_2,4
m_3,7
m_4,9

Discussion

- Some events (in different processes) are assigned the same timestamp
  - If we want to avoid it, we can include the process id into the timestamp: $C(\mathbf{x})=\langle C_i(\mathbf{x}),i\rangle$ if event **x** occurs in Pi
    - assumption: process ids are unique
    - $\langle C_i(\mathbf{x}),i\rangle < \langle C_j(\mathbf{y}),j\rangle$ iff $C_i(\mathbf{x})<C_j(\mathbf{y})$ or $C_i(\mathbf{x})=C_j(\mathbf{y})$ and $i<j$
- $\mathbf{x} \rightarrow \mathbf{y} \Rightarrow C(\mathbf{x})<C(\mathbf{y})$ but **not** vice versa
- The algorithm can be implemented by
  - a timestamping function that increments and returns the counter
  - hooks in the send/receive functions that timestamp send/receive events and messages and adjust the counter

Example with Total Ordering



Application: Total-ordered Multicast

- Problem statement:
  - a set of processes that send multicast messages
  - all multicast messages must be delivered in the same order to each receiver (total-ordered multicast)
- Example
  - 2 or more identical replicated servers
    - replication goals: fault-tolerance and latency reduction
  - service requests can be sent to any server in the group
  - the receiving server multicasts the request to all servers
  - operations must be executed in each replica in the same order

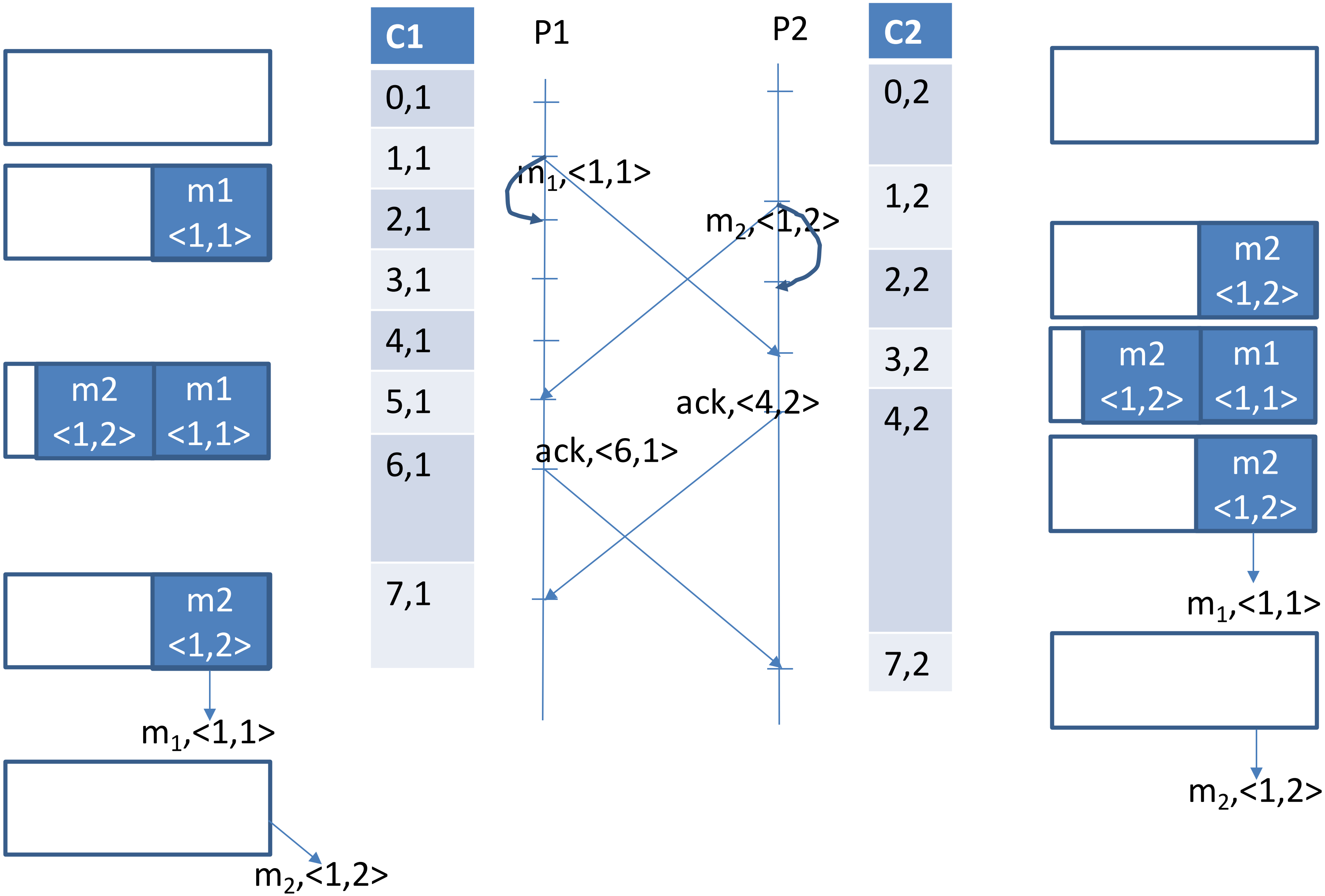
Total-ordered Multicast Algorithm

- Lamport clocks are maintained in each process.

  => Each multicast message is timestamped by the sender with the send event timestamp

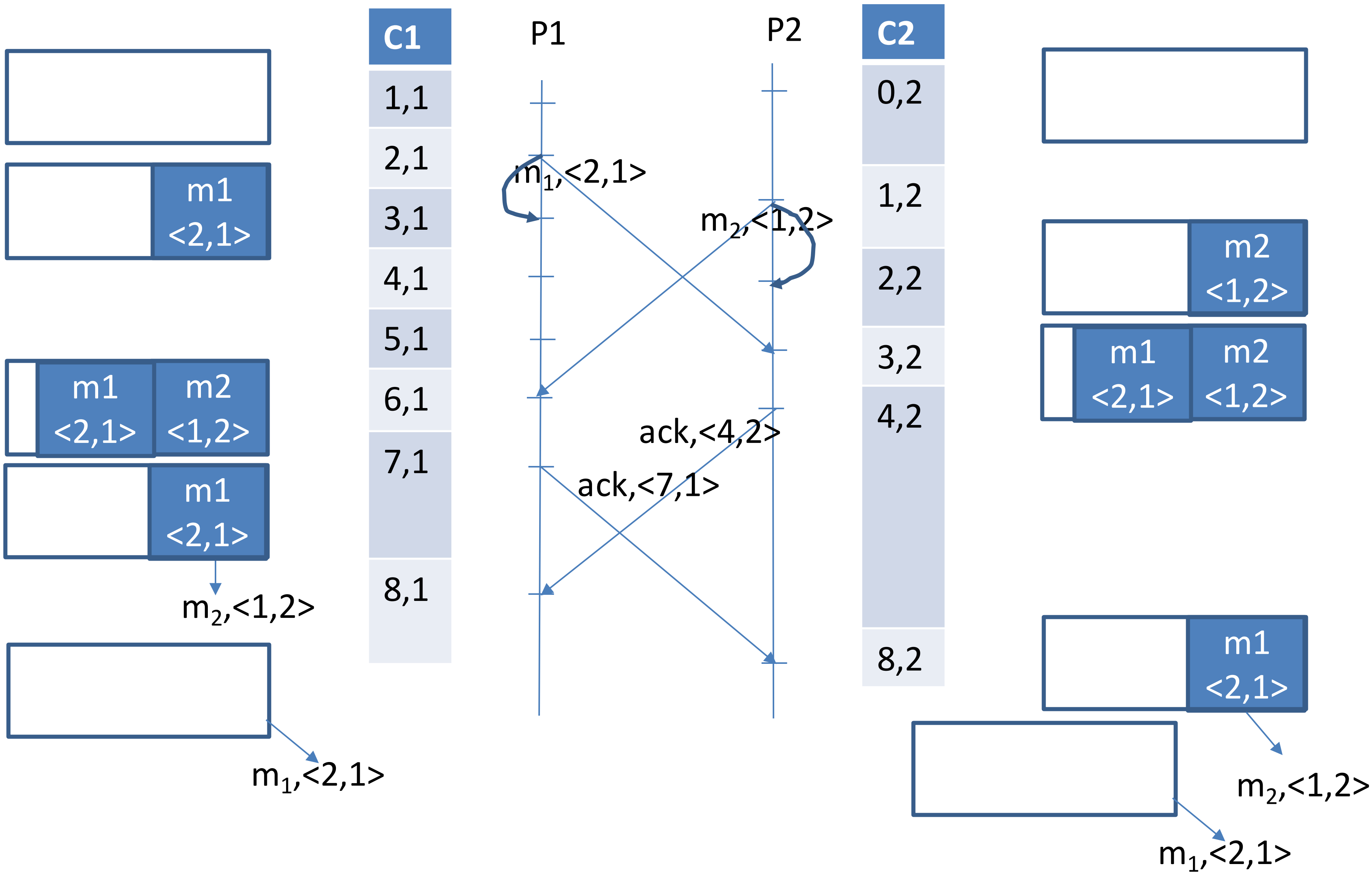
- Received messages are queued, ordered by timestamp
- Each receiver acknowledges message reception to the other receivers
- When acknowledgments for the message at the head of the queue have been received from all other receivers, that message is dequeued and handed to the application

Example



C1
0,1
1,1
2,1
3,1
4,1
5,1
6,1
7,1
P1
P2
C2
0,2
1,2
2,2
3,2
4,2
7,2
m_1,<1,1>
m_2,<1,2>
ack,<4,2>
ack,<6,1>
m1 <1,1>
m2 <1,2>
m2 <1,2>
m1 <1,1>
m2 <1,2>
m_1,<1,1>
m_2,<1,2>
m2 <1,2>
m2 <1,2>
m1 <1,1>
m2 <1,2>
m_1,<1,1>
m_2,<1,2>

Example: Other Scenario



Total-ordered Multicast: Discussion

- The algorithm has been shown to deliver messages to all destination processes in the same order under the following assumptions:
  - no messages are lost
  - messages from the same sender are received in the same order they were sent
- In case the messages have the meaning of operations to be executed on replicated data in the same order, the algorithm implements state machine replication

Vector Clocks

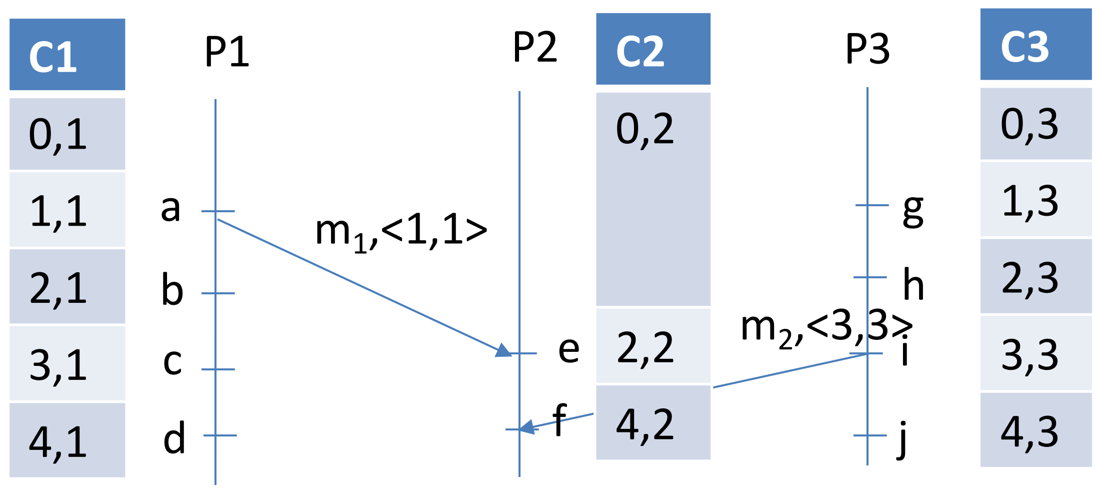
- Lamport clocks provide a **total ordering** of events, but this ordering does not necessarily imply *causality*

  C(**x**) < C(**y**) ? **x** → **y**

- Example:

  C(**e**) < C(**i**)

  but not **e** → **i**



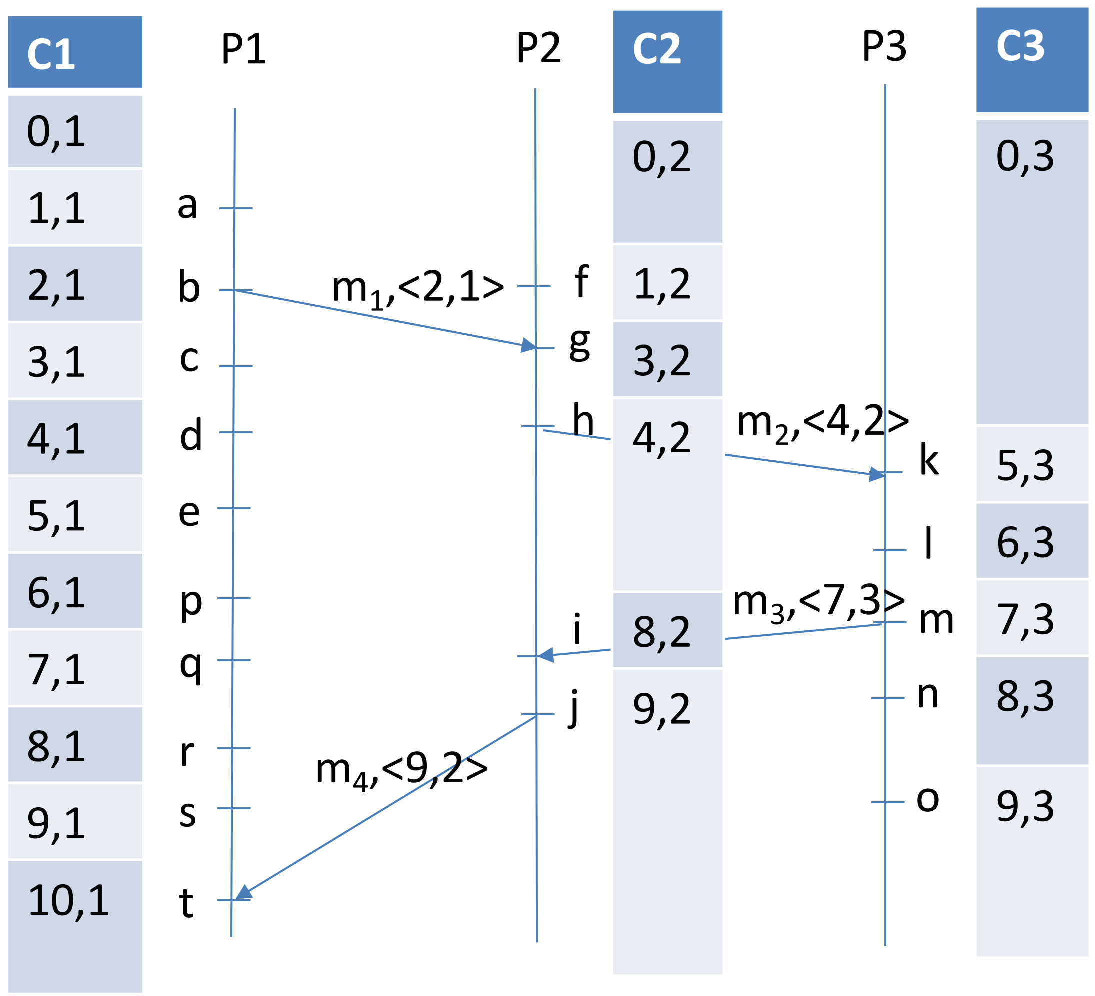
- Vector clocks are a way to provide a **partial ordering** of events that captures causality, i.e.

  C(**x**) < C(**y**) <=> **x** → **y**

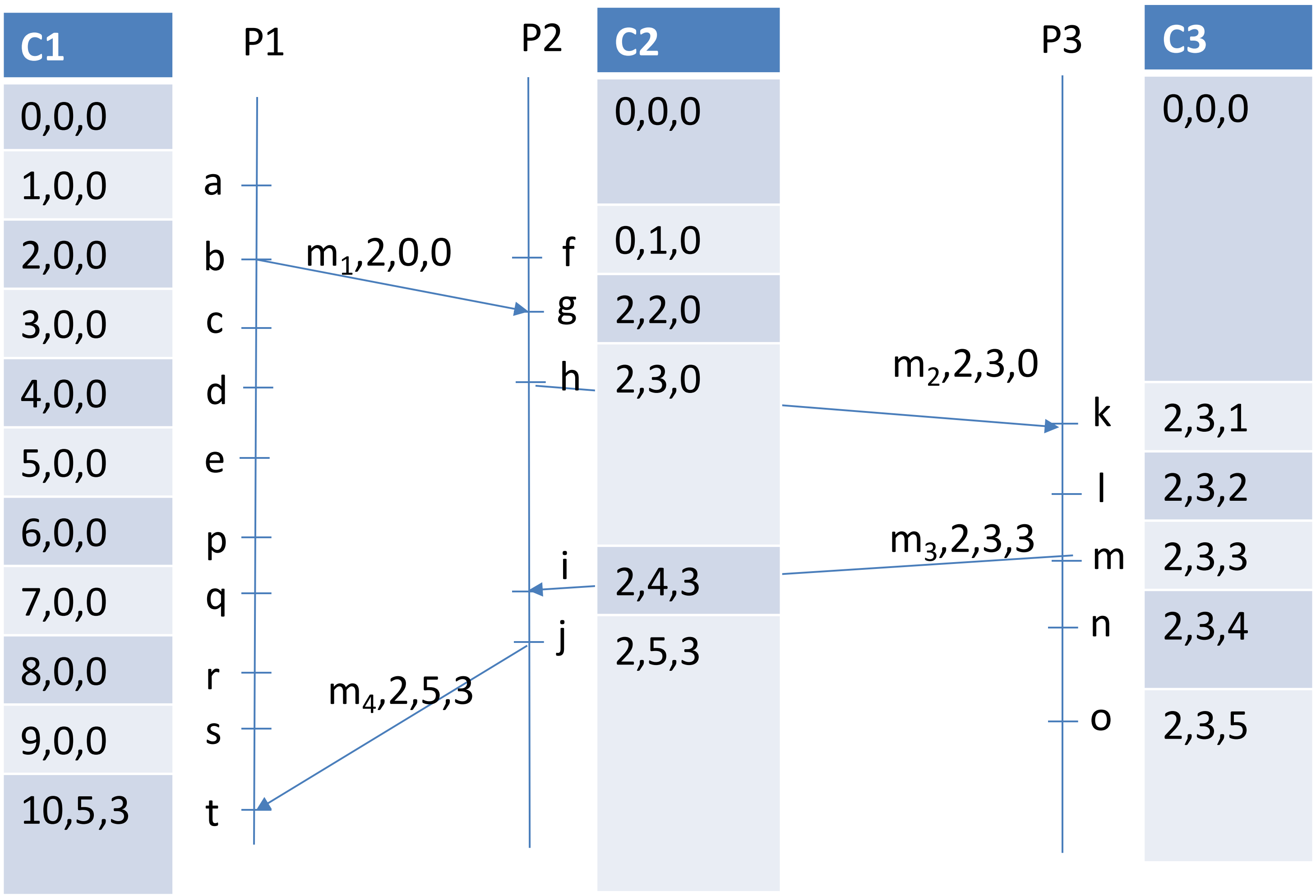
Vector Clocks Algorithm

- Each process Pi keeps a local **vector** of time counters VCi[]
  - VCi[i] is the local event counter (i.e. local time at Pi, as with Lamport)
  - VCi[j] is Pi's knowledge of the local time at Pj
- Pi timestamps each local event **x** with the current VCi value ts(**x**), after having incremented VCi[i]
- Pi timestamps each message it sends with the timestamp ts(**s**) assigned to the send event **s**
- when Pi receives a message with timestamp tsr,
  - Pi sets VCi[k]=max(VCi[k],tsr[k]) for each k (vector clock adjustment) and then it timestamps the receive event as usual

Example: How does this scenario change with Vector Clocks?



Example with Vector Clocks



Vector Clock Application Example

- Causal Ordered Multicast
  - each multicast message must be delivered after the delivery of all other causally related messages that were sent before
  - non causally related messages that were sent before may be received after
- Use Example
  - in a replicated chat, we may accept that the ordering of posts is different in each replica provided each reply to a post p does never come before p