Process groups and message ordering

If processes belong to groups, certain algorithms can be used that depend on group properties

- membership
  create (name), kill (name)
  join (name, process), leave (name, process)

- internal structure?
  NO (peer structure) – failure tolerant, complex protocols
  YES (a single coordinator and point of failure) – simpler protocols
  e.g. all join requests must go to the coordinator – concurrent joins avoided

- closed or open?
  OPEN – a non-member can send a message to the group
  CLOSED – only members can send to the group

- failures?
  a failed process leaves the group without executing leave

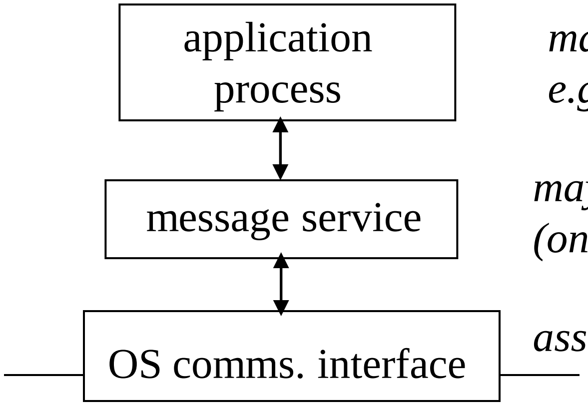
- robustness
  leave, join and failures happen during normal operation – algorithms must be robust

Message delivery for a process group - assumptions

ASSUMPTIONS

- messages are multicast to named process groups
- reliable channels: a given message is delivered reliably to all members of the group
- FIFO from a given source to a given destination
- processes don’t crash (failure and restart not considered)
- processes behave as specified e.g. send the same values to all processes
  - we are not considering so-called Byzantine behaviour
    (when malicious or erroneous processes do not behave according to their specifications
    see Lamport’s Byzantine Generals problem).

Ordering message delivery


application process

message service

OS comms. interface

may specify delivery order to message service
e.g. arrival (no) order, total order, causal order

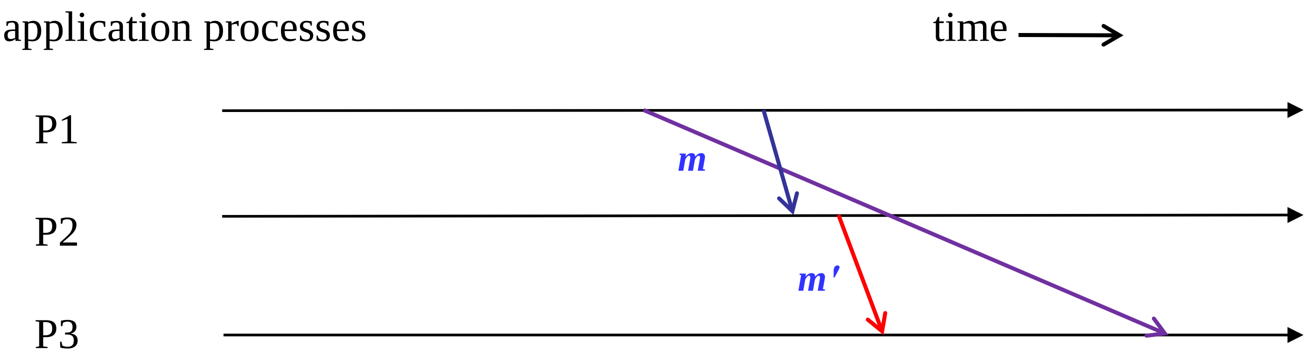
may reorder delivery to application
(on request for some order) by buffering messages

assume FIFO from each source at this level
(done by lower levels)

total order = every process receives all messages in the same order (including its own).
We first consider *causal order*

Message delivery – causal order

First, define causal order in terms of one-to-one messages; later, multicast to a process group



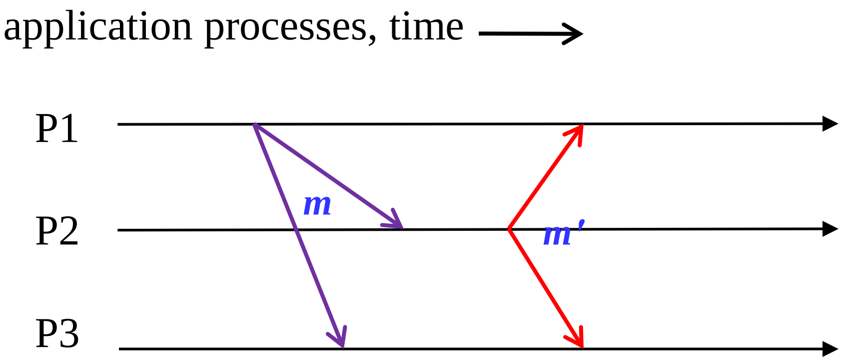
P1 sent message ***m*** *provably before* P2 sent message ***m'***
The above diagram shows a violation of causal delivery order
Causal delivery order requires that, at P3, ***m*** is delivered before ***m'***
The definition relates to POTENTIAL causality, not application semantics

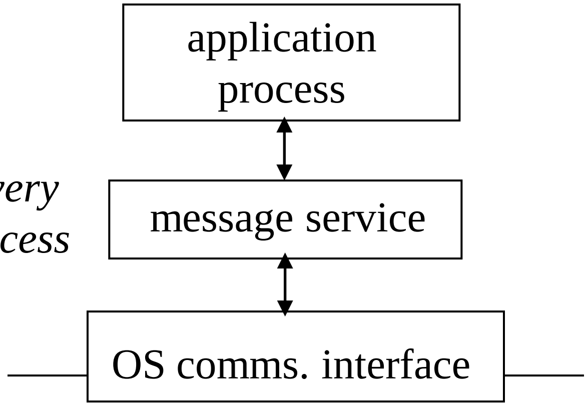
DEFINITION of causal delivery order (where < means “happened before”)

$$send_i(m) < send_j(m') \Rightarrow deliver_k(m) < deliver_k(m')$$

Message delivery – causal order for a process group

If we know that all processes in a group receive all messages, the message delivery service can implement causal delivery order (for total order, see later)




application process

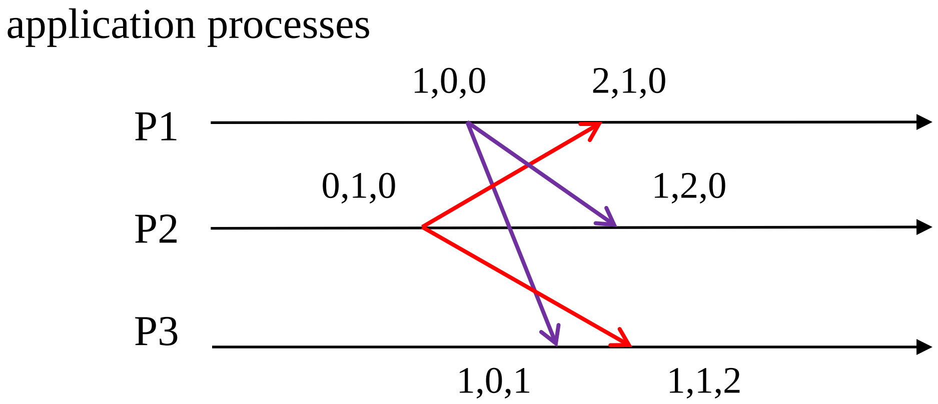
message service

OS comms. interface

the message service can postpone the delivery of messages to the application process

Message delivery – causal order using Vector Clocks

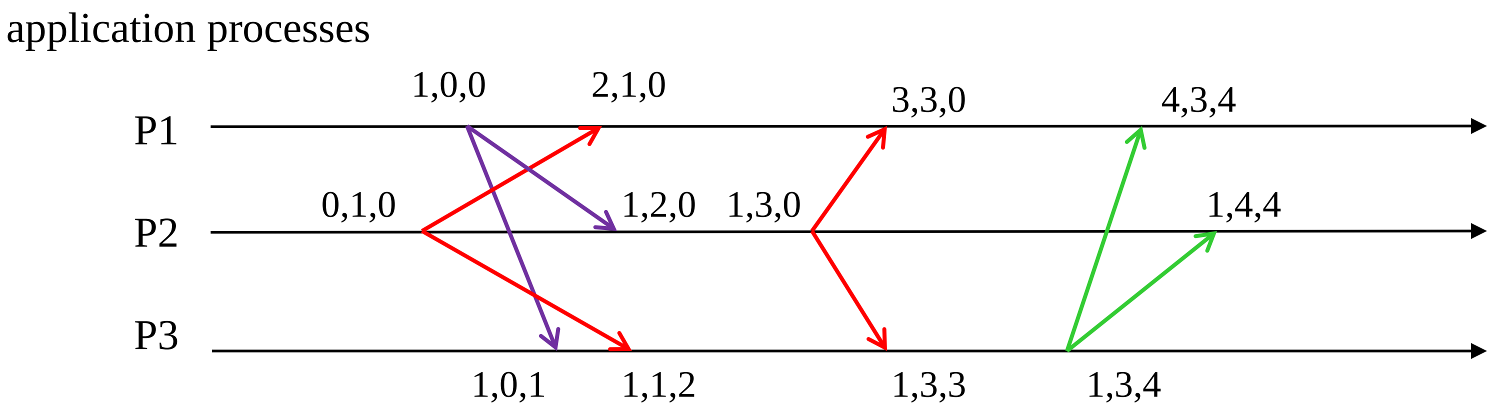
A vector clock is maintained by the message service at each node for each process:



vector notation:

- fixed number of processes N
- each process’s message service keeps a vector of dimension N
- for each process, each entry records the most up-to-date value of the state counter delivered to the application process, for the process at that position

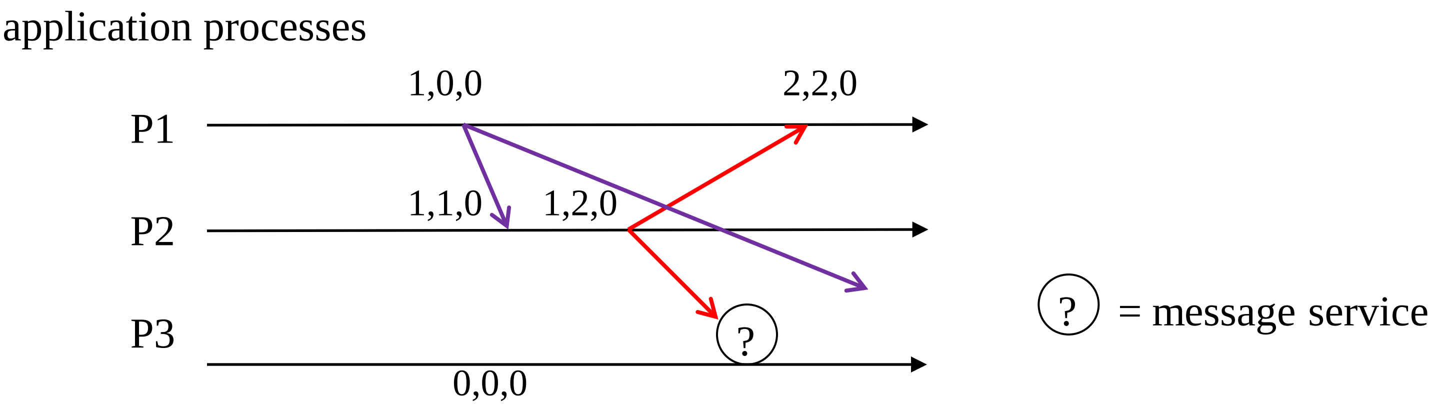
Vector Clocks – message service operation



Message service operation:

- before ***send*** increment local process state value in local vector
- on ***send***, timestamp message with sending process's local vector
- on ***receive*** *by message service* – see below
- on ***deliver*** *to receiving application process*, increment receiving process's state value in its local vector and update the other fields of the vector by comparing its values with the incoming vector (timestamp) and recording the higher value in each field, thus updating this process's knowledge of system state

Implementing causal order using Vector Clocks



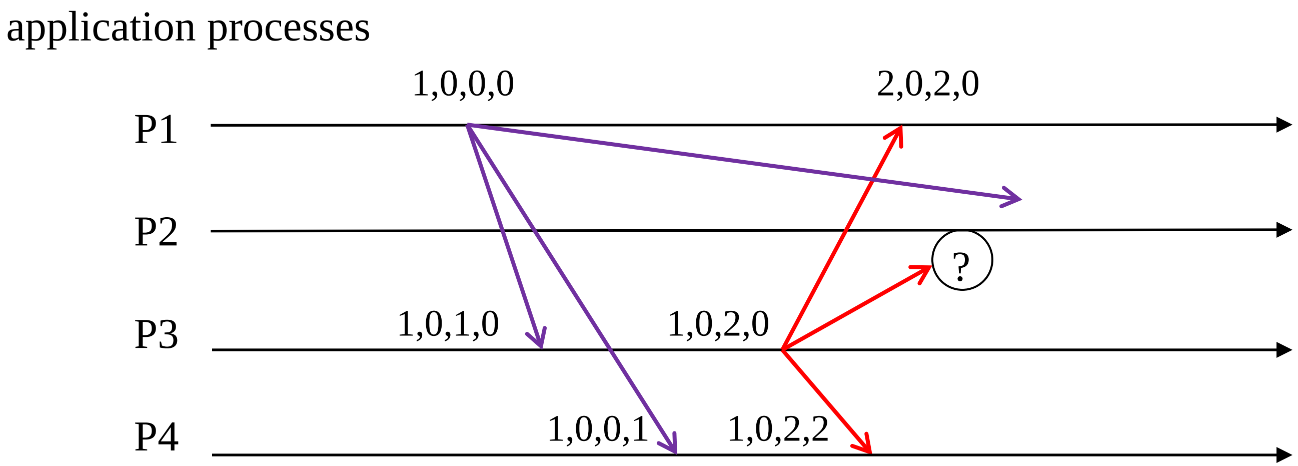
P3's vector is at (0,0,0) and a message with timestamp (1,2,0) arrives from P2
i.e. P2 has received a message from P1 that P3 hasn't seen.

More detail of P3's message service:

receiver vector	sender	sender vector	decision	new receiver vector
0,0,0	P2	1,2,0	buffer	0,0,0
	P3 *is missing a message from* P1 *that sender* P2 *has already received*			
0,0,0	P1	1,0,0	deliver	1,0,1
1,0,1	P2	1,2,0	deliver	1,2,2

In each case: do the *sender and receiver agree on the state of all other processes*?
If the sender has a higher state value for any of these others, the receiver is missing a message, so buffer the message

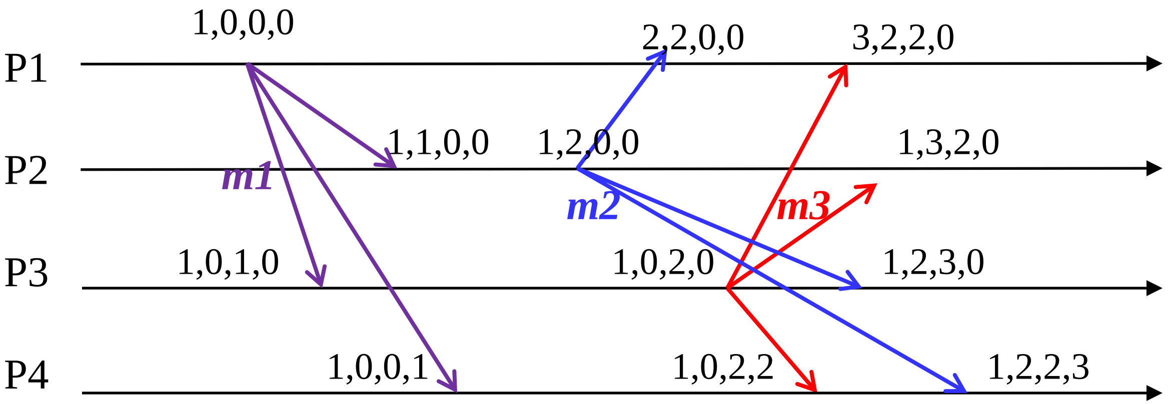
Vector Clocks - example



sender	sender vector	receiver	receiver vector	decision	new receiver vector
P3	1,0,2,0	P1	1,0,0,0	deliver	P1 -> 2,0,2,0
	same states for P2 *and* P4				
P3	1,0,2,0	P4	1,0,0,1	deliver	P4 -> 1,0,2,2
	same states for P1 *and* P2				
P3	1,0,2,0	P2	0,0,0,0	buffer	P2 -> 0,0,0,0
	same state for P4, *different for* P1- *missing message*				
P1	1,0,0,0	P2	0,0,0,0	deliver	P2 -> 1,1,0,0
	reconsider buffered message:				
P3	1,0,2,0	P2	1,1,0,0	deliver	P2 -> 1,2,2,0
	same states for P1 *and* P4				

Total order is not enforced by the vector clocks algorithm

application processes



m2 and ***m3*** are not causally related

P1 receives ***m1***, ***m2***, ***m3***
P2 receives ***m1***, ***m2***, ***m3***
P3 receives ***m1***, ***m3***, ***m2***
P4 receives ***m1***, ***m3***, ***m2***

If the application requires total order this could be enforced by modifying the vector clock algorithm to include ACKs and delivery to self.

Totally ordered multicast

The vectors can be a large overhead on message transmission and a simpler algorithm can be used if only total order is required.

Recall the ASSUMPTIONS

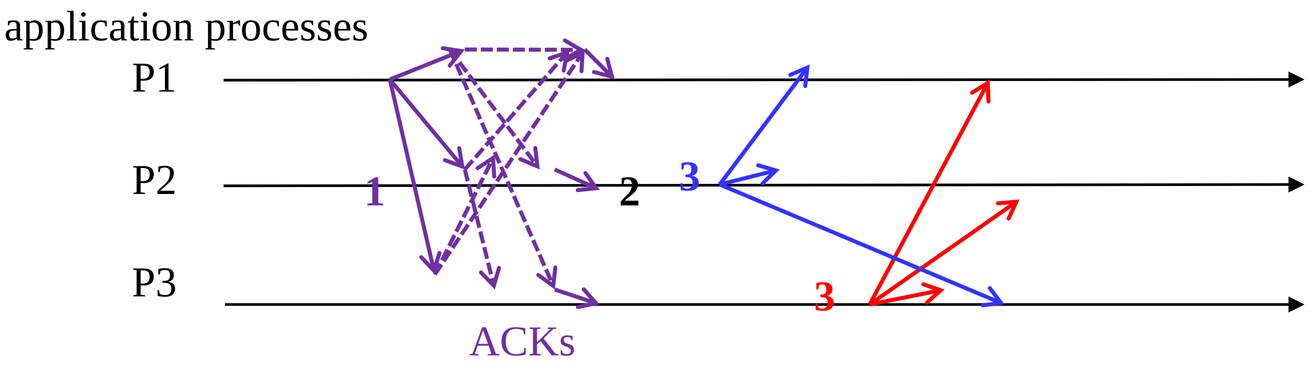
- messages are multicast to named process groups
- reliable channels: a given message is delivered reliably to all members of the group
- FIFO from a given source to a given destination
- processes don't crash (failure and restart not considered)
- no Byzantine behaviour

total order algorithm

- sender multicasts to all including itself
- all acknowledge receipt as a multicast message
- message is delivered in timestamp order after all ACKs have been received

If the delivery system must support both, so that applications can choose, vector clocks can achieve both causal and total ordering.

Total ordered multicast – outline of approach



P1 increments its clock to 1 and multicasts a message with timestamp 1
All delivery systems collect the message, multicast ACK and collect all ACKs
 - no contention – deliver message to application processes
and increment local clocks to 2.

P2 and P3 both multicast messages with timestamp 3
All delivery systems collect messages, multicast ACKs and collect ACKs.
 - contention – so use a tie-breaker (e.g. lowest process ID wins)
and deliver P2s message before P3s

This is just a sketch of an approach. In practice, timeouts would have to be used to take account of long delays due to congestion and/or failure of components and/or communication links