Coordination-**Logical Clocks**

Distributed Systems

FACULTY OF INFORMATION TECHNOLOGY

Physical clocks

Problem

Sometimes we simply need the exact time, not just an ordering.

Solution: Universal Coordinated Time (UTC)

Note

UTC is broadcast through short-wave radio and satellite. Satellites can give an accuracy of about ± 0.5 ms.

The Happened-before relationship

Issue

What usually matters is not that all processes agree on exactly what time it is, but that they agree on the order in which events occur. Requires a notion of ordering.

The happened-before relation

- If a and b are two events in the same process, and a comes before b, then $a \to b$.
- If a is the sending of a message, and b is the receipt of that message, then $a \to b$
- If $a \to b$ and $b \to c$, then $a \to c$

Logical clocks

Problem

How do we maintain a global view on the system's behavior that is consistent with the happened-before relation?

Attach a timestamp $C(e)$ to each event e, satisfying the following properties:

- P1 If a and b are two events in the same process, and $a \rightarrow b$, then we demand that $C(a) < C(b)$.
- P2 If a corresponds to sending a message m, and b to the receipt of that message, then also $C(a) < C(b)$.

Problem

How to attach a timestamp to an event when there's no global clock $\Rightarrow$ maintain a consistent set of logical clocks, one per process.

Logical clocks: solution

Each process P_i maintains a local counter C_i and adjusts this counter

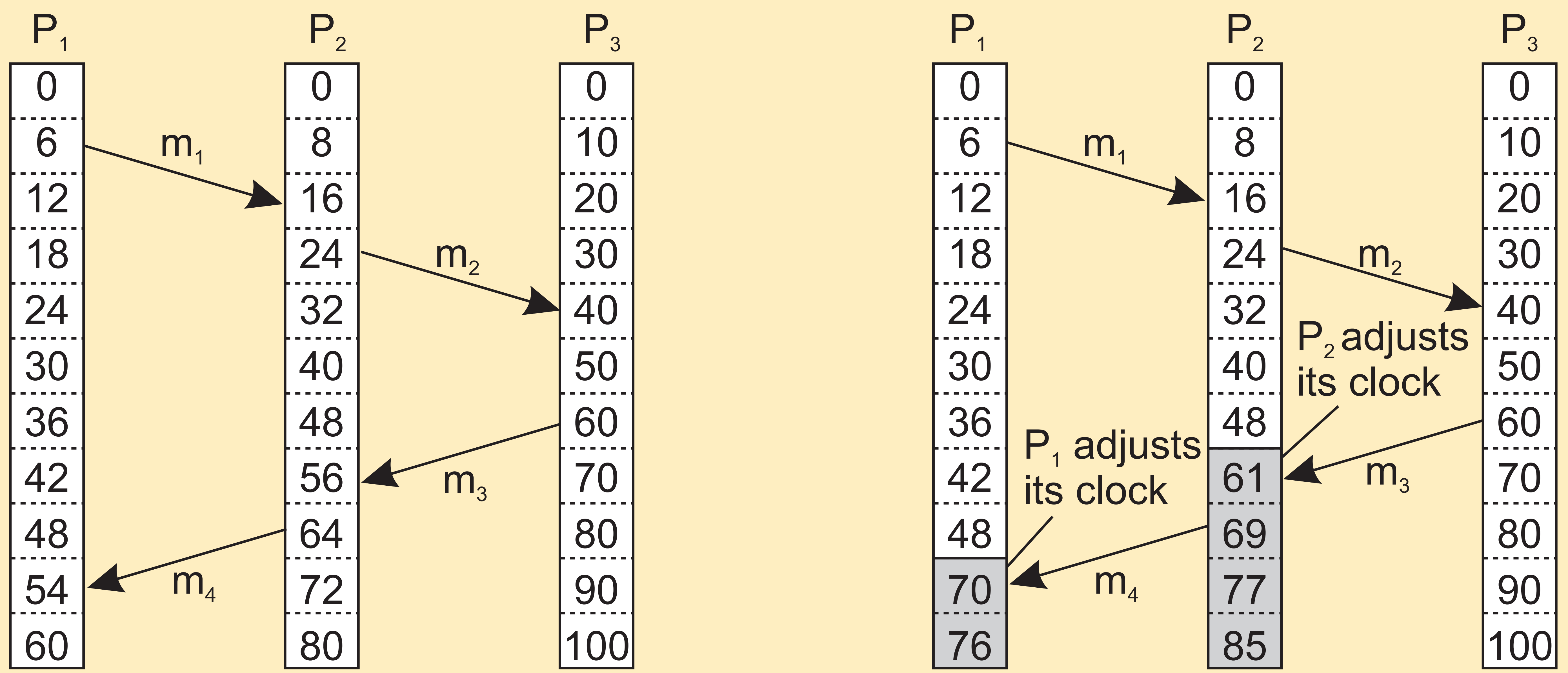
1. For each new event that takes place within P_i, C_i is incremented by 1.
2. Each time a message m is sent by process P_i, the message receives a timestamp $ts(m) = C_i$.
3. Whenever a message m is received by a process P_j, P_j adjusts its local counter C_j to $\max\{C_j, ts(m)\}$; then executes step 1 before passing m to the application.

Notes

- Property P1 is satisfied by (1); Property P2 by (2) and (3).
- It can still occur that two events happen at the same time. Avoid this by breaking ties through process IDs.

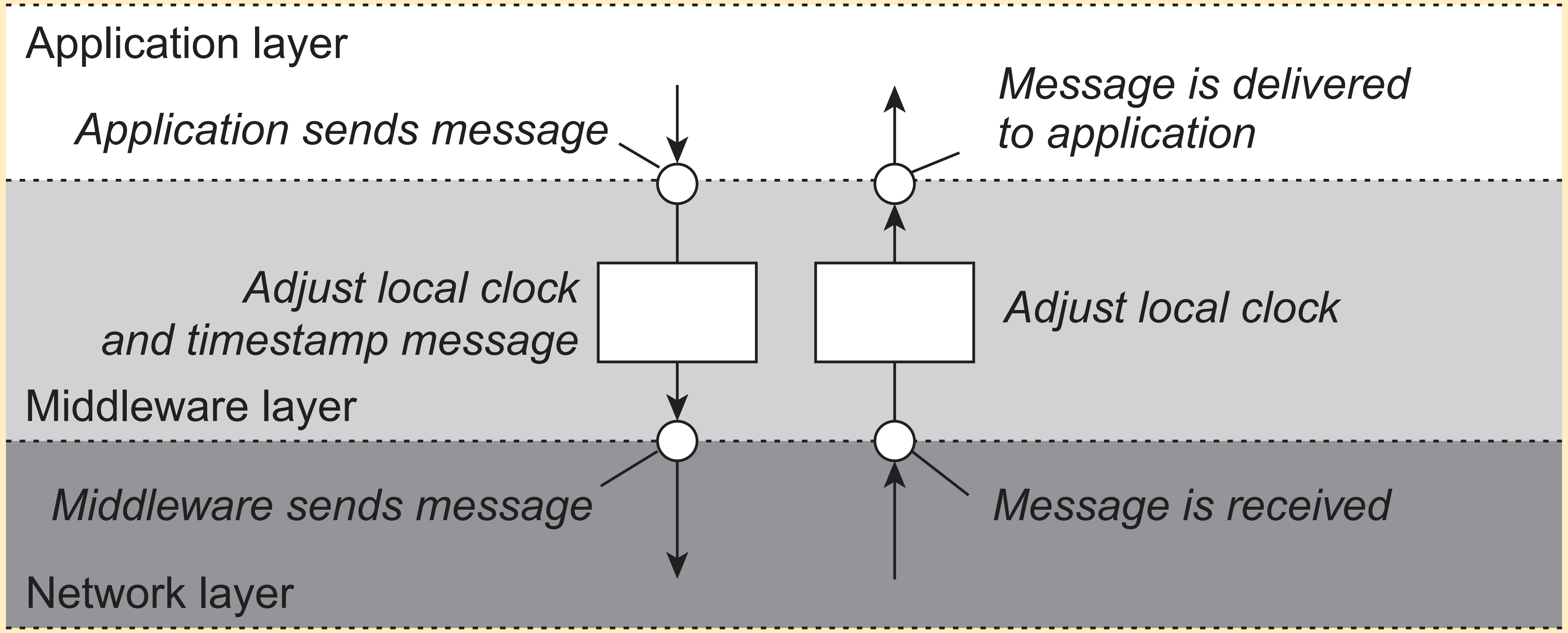
Logical clocks: example

Consider three processes with event counters operating at different rates



Logical clocks: where implemented

Adjustments implemented in middleware



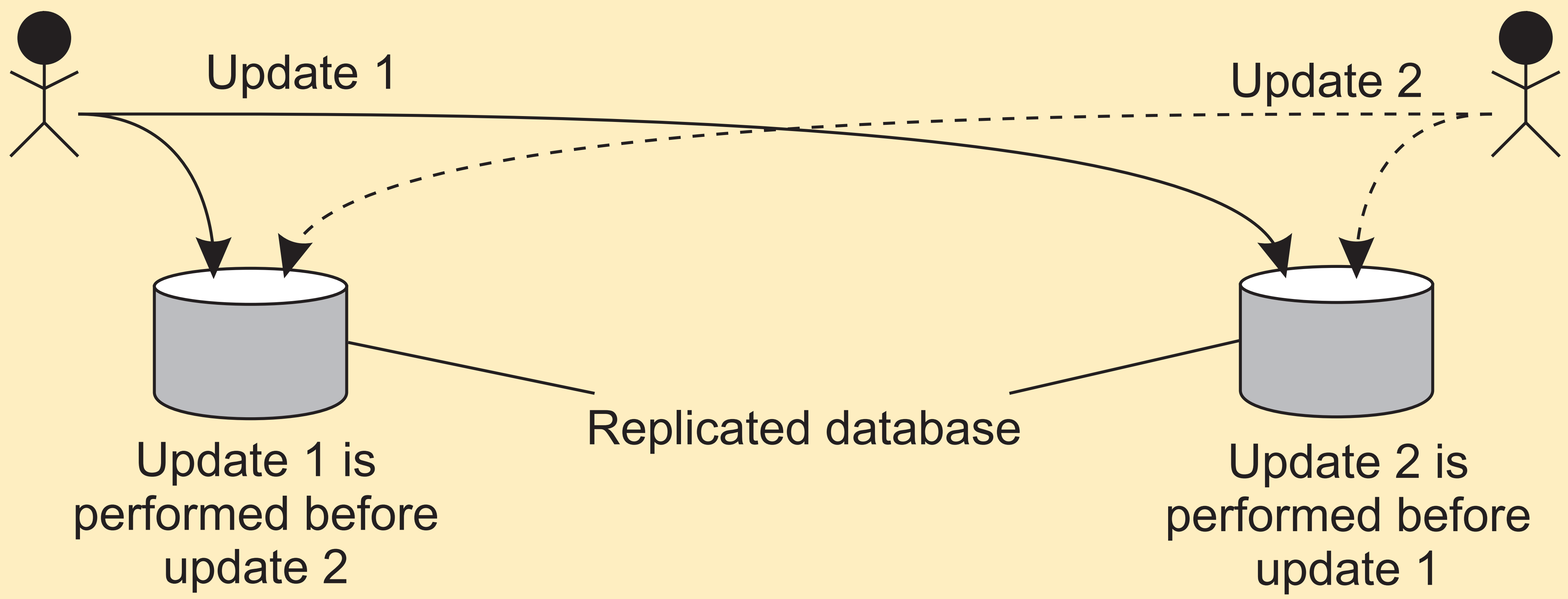
To implement Lamport's logical clocks, each process P_i maintains a *local* counter C_i. These counters are updated according to the following steps

1. Before executing an event (i.e., sending a message over the network, delivering a message to an application, or some other internal event), P_i increments C_i: $C_i \leftarrow C_i + 1$.
2. When process P_i sends a message m to process P_j, it sets m's timestamp $ts(m)$ equal to C_i after having executed the previous step.
3. Upon the receipt of a message m, process P_j adjusts its own local counter as $C_j \leftarrow \max\{C_j, ts(m)\}$ after which it then executes the first step and delivers the message to the application.

Example: Total-ordered multicast

Concurrent updates on a replicated database are seen in the same order everywhere

- P_1 adds \$100 to an account (initial value: \$1000)
- P_2 increments account by 1%
- There are two replicas



Result

In absence of proper synchronization:
replica #1 $\leftarrow$ \$1111, while replica #2 $\leftarrow$ \$1110.

Example: Total-ordered multicast

Solution

- Process P_i sends timestamped message m_i to all others. The message itself is put in a local queue $queue_i$.
- Any incoming message at P_j is queued in $queue_j$, according to its timestamp, and acknowledged to every other process.

P_j passes a message m_i to its application if:

(1) m_i is at the head of $queue_j$

(2) for each process P_k, there is a message m_k in $queue_j$ with a larger timestamp.

Note

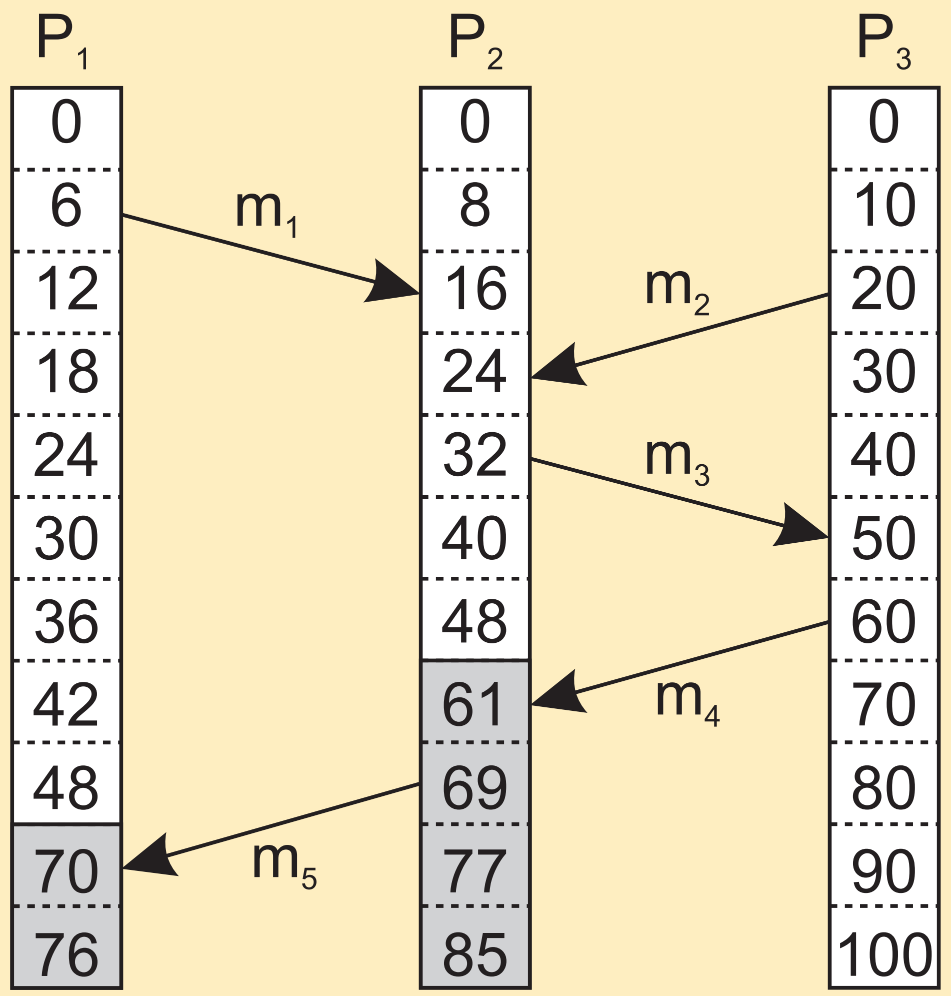
We are assuming that communication is reliable and FIFO ordered.

Vector clocks

Observation

Lamport's clocks do not guarantee that if $C(a) < C(b)$ that a causally preceded b.

Concurrent message transmission using logical clocks



Observation

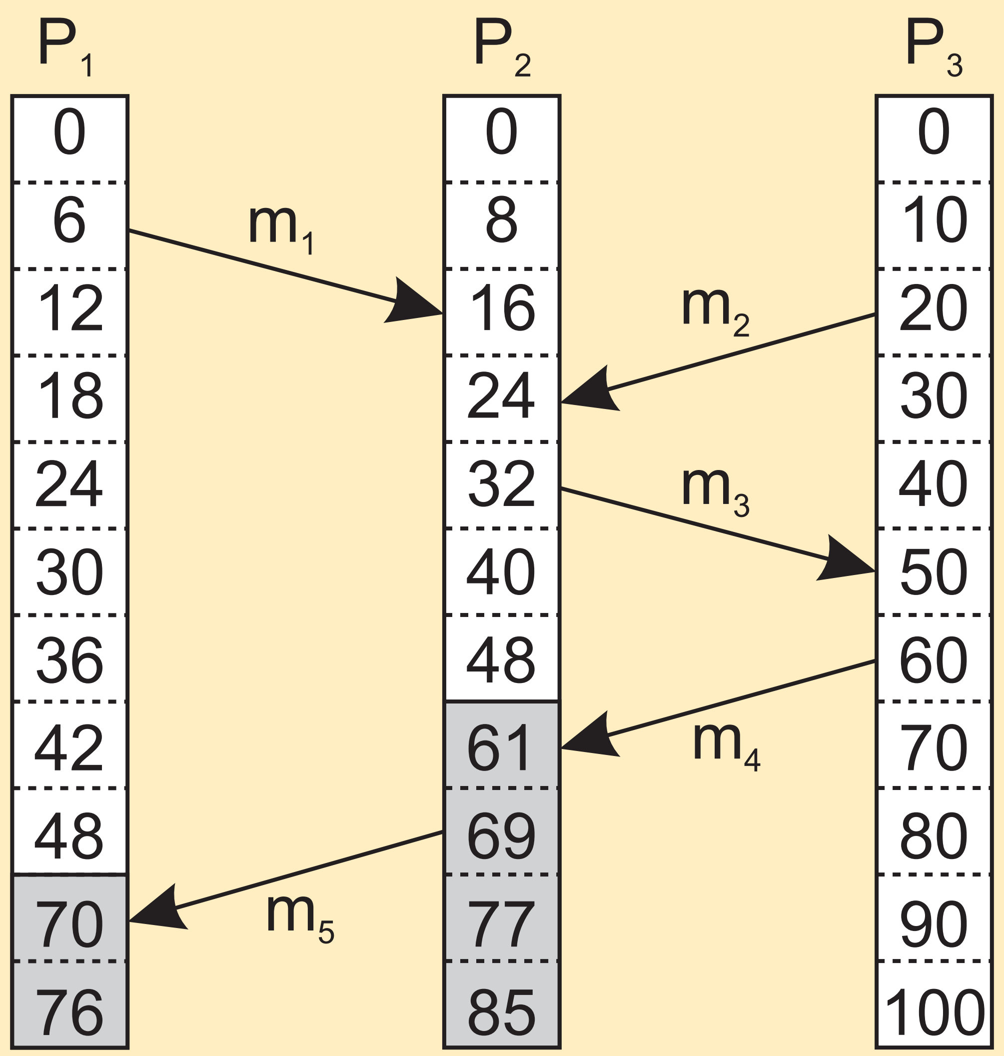
Event a: m_1 is received at $T = 16$;
Event b: m_2 is sent at $T = 20$.

Vector clocks

Observation

Lamport's clocks do not guarantee that if $C(a) < C(b)$ that a causally preceded b.

Concurrent message transmission using logical clocks



Observation

Event a: m_1 is received at $T = 16$;
Event b: m_2 is sent at $T = 20$.

Note

We cannot conclude that a causally precedes b.

Capturing causality

Solution: each P_i maintains a vector VC_i

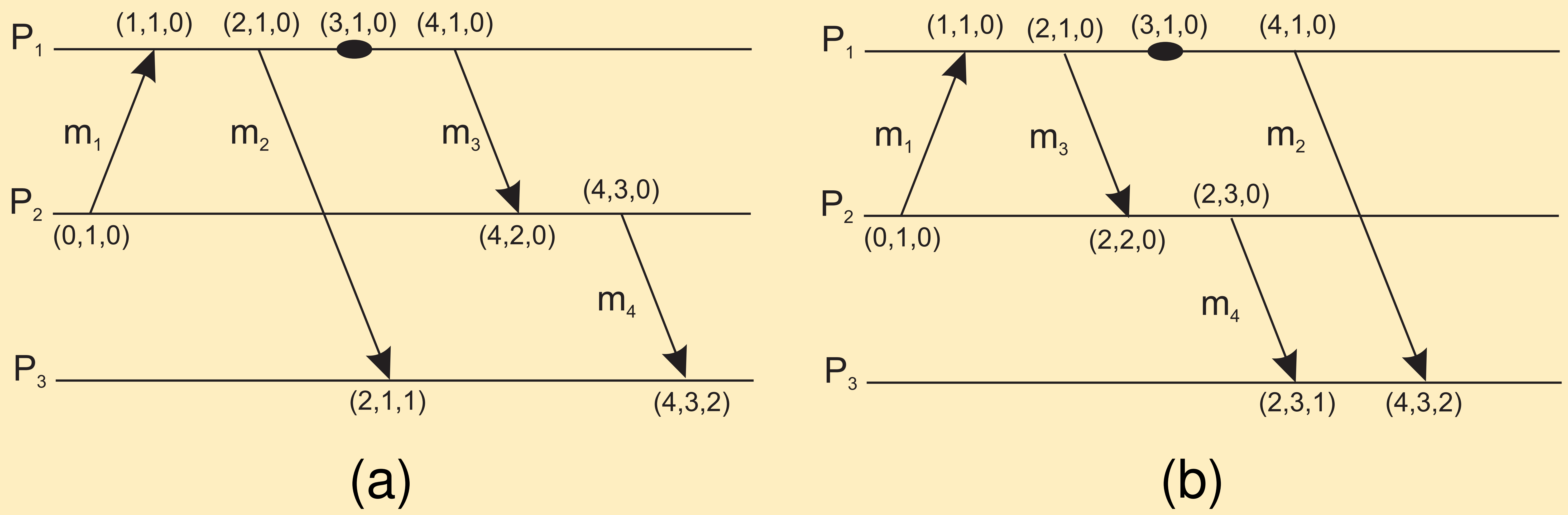
- $VC_i[i]$ is the local logical clock at process P_i.
- If $VC_i[j] = k$ then P_i knows that k events have occurred at P_j.

Maintaining vector clocks

1. Before executing an event P_i executes $VC_i[i] \leftarrow VC_i[i] + 1$.
2. When process P_i sends a message m to P_j, it sets m's (vector) timestamp $ts(m)$ equal to VC_i after having executed step 1.
3. Upon the receipt of a message m, process P_j sets $VC_j[k] \leftarrow \max\{VC_j[k], ts(m)[k]\}$ for each k, after which it executes step 1 and then delivers the message to the application.

Vector clocks: Example

Capturing potential causality when exchanging messages



In Figure (a), P_2 sends a message m_1 at logical time $VC_2 = (0,1,0)$ to process P_1. Message m_1 thus receives timestamp $ts(m_1) = (0,1,0)$. Upon its receipt, P_1 adjusts its logical time to $VC_1 \leftarrow (1,1,0)$ and delivers it. Message m_2 is sent by P_1 to P_3 with timestamp $ts(m_2) = (2,1,0)$. Before P_1 sends another message, m_3, an event happens at P_1, eventually leading to timestamping m_3 with value $(4,1,0)$. After receiving m_3, process P_2 sends message m_4 to P_3, with timestamp $ts(m_4) = (4,3,0)$.

Now consider the situation shown in Figure (b). Here, we have delayed sending message m_2 until after message m_3 has been sent, and after the event had taken place. It is not difficult to see that $ts(m_2) = (4,1,0)$, while $ts(m_4) = (2,3,0)$.

Causally ordered multicasting

Observation

We can now ensure that a message is delivered only if all causally preceding messages have already been delivered.

Adjustment

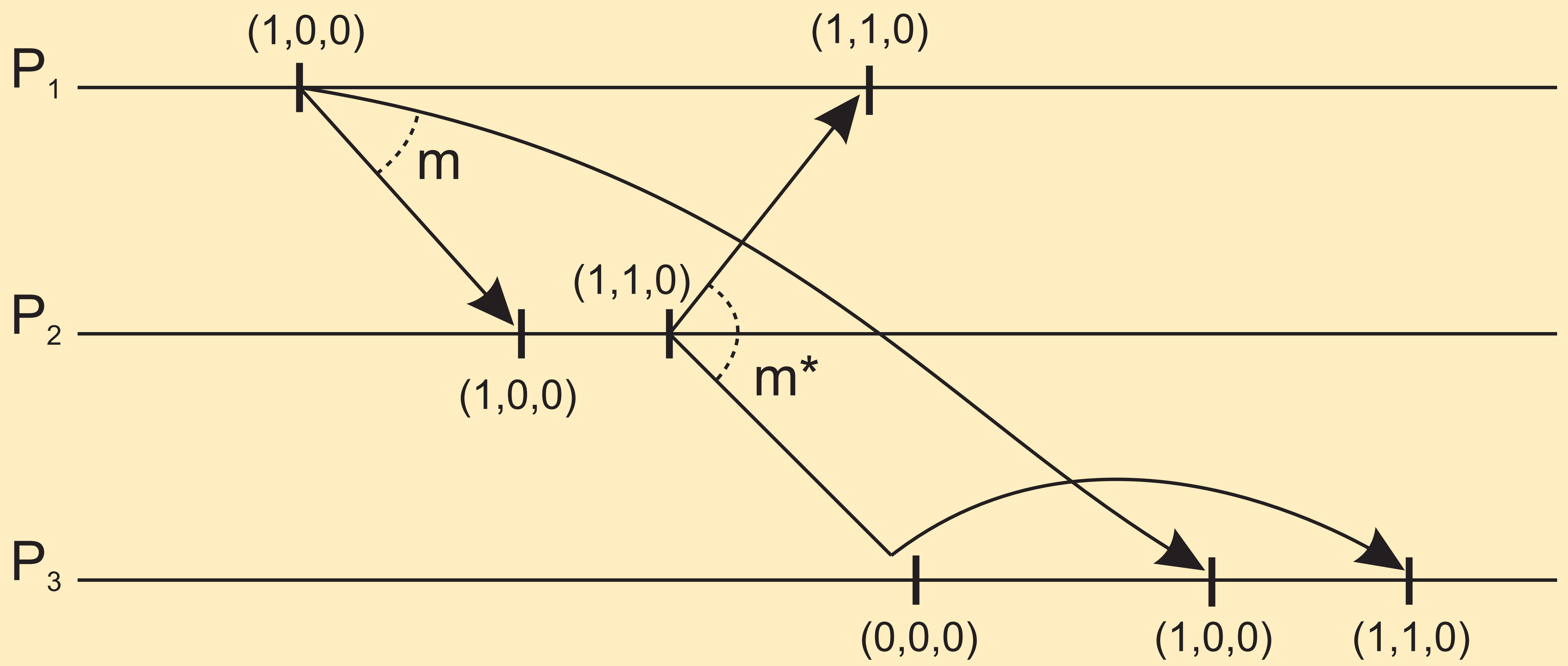
P_i increments $VC_i[i]$ only when sending a message, and P_j “adjusts” VC_j when receiving a message (i.e., effectively does not change $VC_j[j]$).

P_j postpones delivery of m until:

1. $ts(m)[i] = VC_j[i] + 1$
2. $ts(m)[k] \leq VC_j[k]$ for all $k \neq i$

Causally ordered multicasting

Enforcing causal communication



As an example, consider three processes P_1, P_2, and P_3 as shown in Figure At local time $(1, 0, 0)$, P_1 sends message m to the other two processes. Note that $ts(m) = (1, 0, 0)$. Its receipt and subsequent delivery by P_2, will bring the logical clock at P_2 to $(1, 0, 0)$, effectively indicating that it has received one message from P_1, has itself sent no message so far, and has not yet received a message from P_3. P_2 then decides to send m^*, at updated time $(1, 1, 0)$, which arrives at P_3 *sooner* than m.

When comparing the timestamp of m with its current time, which is $(0, 0, 0)$, P_3 concludes that it is still missing a message from P_1 which P_2 apparently had delivered before sending m^*. P_3 therefore decides to postpone the delivery of m^* (and will also not adjust its local, logical clock). Later, after m has been received and delivered by P_3, which brings its local clock to $(1, 0, 0)$, P_3 can deliver message m^* and also update its clock.