CS x55: Distributed Systems [Logical Clocks]

Shrideep Pallickara
Computer Science
Colorado State University

Frequently asked questions from the previous class survey

- Is it ok to think of Gntuella (with superpeers) and BitTorrent as semi-structured networks?
- The trade-off in structured/unstructured systems seems to apply to nodes and the resiliency of the networks to be “available”, but what about resiliency from a “file” or “data” perspective in these systems
  - Replication!

Topics covered in this lecture

- Logical clocks
- Vector clocks
- Matrix clocks

Logical Clocks

It will not stir for Doctors -
This Pendulum of snow -
The Shopman importunes it -
While cool - concernless No

Nods from the Gilded pointers -
Nods from Seconds slim -
Decades of Arrogance between
The Dial life -
And Him.

Emily Dickinson

Physical time in a distributed system is problematic

- This is not because of the effects of special relativity, which are negligible or non-existent for normal computers
  - Unless you count computers travelling in spaceships
- It is because of the *inability to accurately timestamp* events at different nodes
  - We need this to **order** any pairs of events

If two processes do not interact with each other?

- Their clocks need not be synchronized
- Lack of synchronization is not observable
  - Does not cause problems

Logical clocks

- Within a single process, events are ordered uniquely by times shown on local clock
- But we cannot synchronize clocks perfectly across a distributed system [Lamport 1978]
  - We cannot use physical time to find out the order of an arbitrary pair of events in a distributed system

We can use a scheme that is similar to physical causality to order events

① If two events occurred at the same process p_i ($i=1, 2, \ldots, N$) ?

- Then they occurred in the order in which p_i observes them
  - This is the order $\rightarrow_i$

② When a message is sent between processes?

- The event of sending the message occurred *before* the event of receiving the message

The → relation

- Lamport called the **partial ordering** obtained by generalizing the previous 2 relationships
  - The *happened-before* or *happens-before* relation
- Sometimes also known as the relation of *causal ordering* or *potential causal ordering*

Lamport's logical clocks

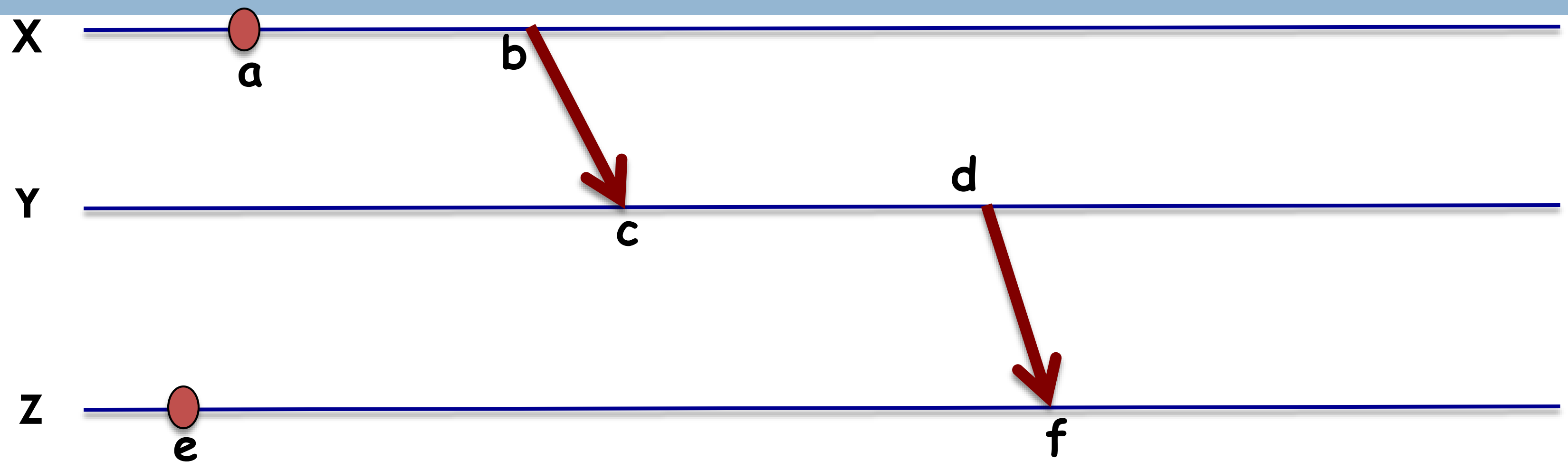
- The **happens-before** relation ➔
- a and b are events in the process; and a occurs before b
  - Then a ➔ b is true
- a is event of message sent by one process;
  b is event of message being received in another process
  - Then a ➔ b is true

Some more things about the happens-before relation

- If $a \rightarrow b$ and $b \rightarrow c$; then $a \rightarrow c$
  - **Transitive**

- If events x and y occur in processes that do not exchange messages, then …
  - $x \rightarrow y$ is not true
  - But, neither is $y \rightarrow x$
  - These events are said to be **concurrent**

Events occurring at three processes

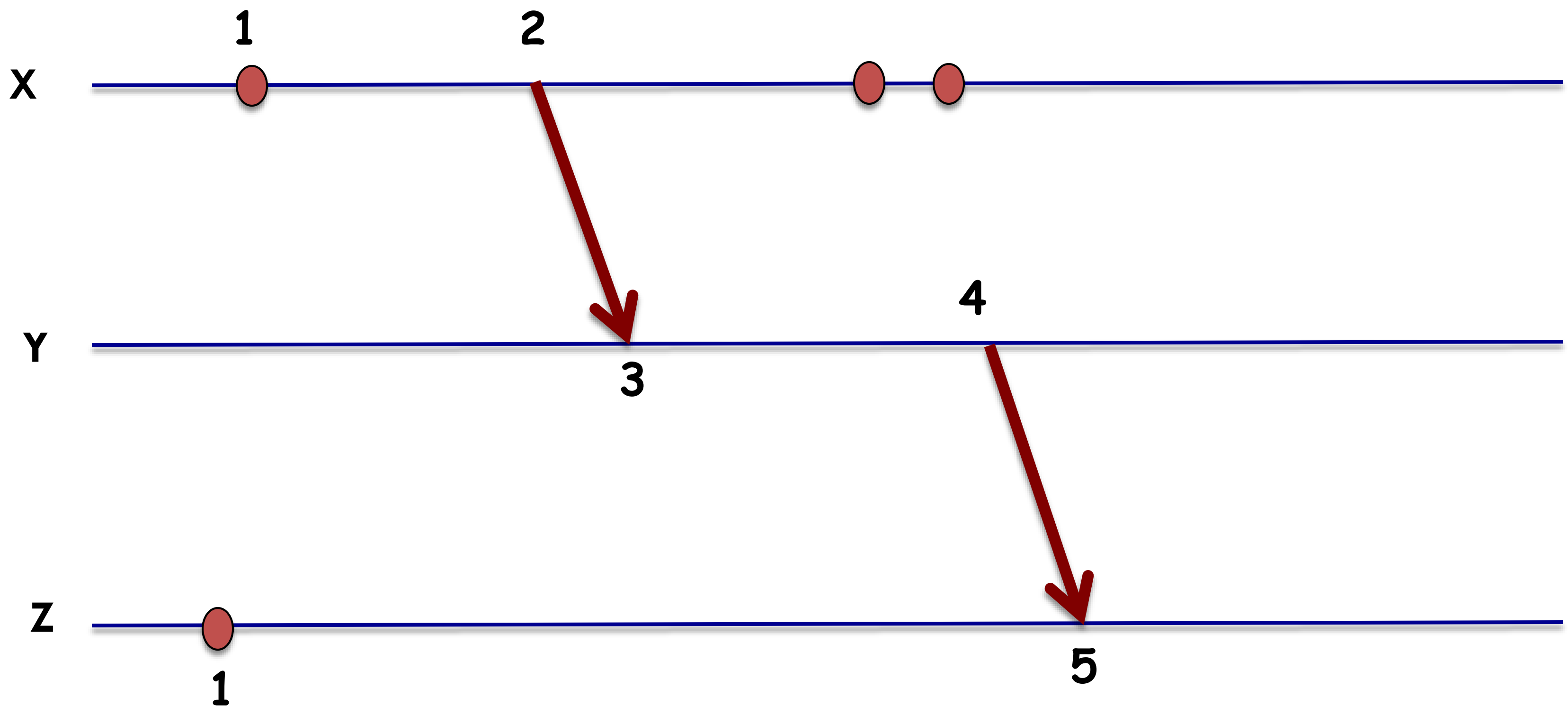


- $a \rightarrow b$ and $c \rightarrow d$
  - These occur within the same process
- $b \rightarrow c$ and $d \rightarrow f$
  - Events that correspond to sending and receiving messages
- We can use transitivity to say $a \rightarrow f$
- No relationship between a and e; these are **concurrent** $a \parallel e$

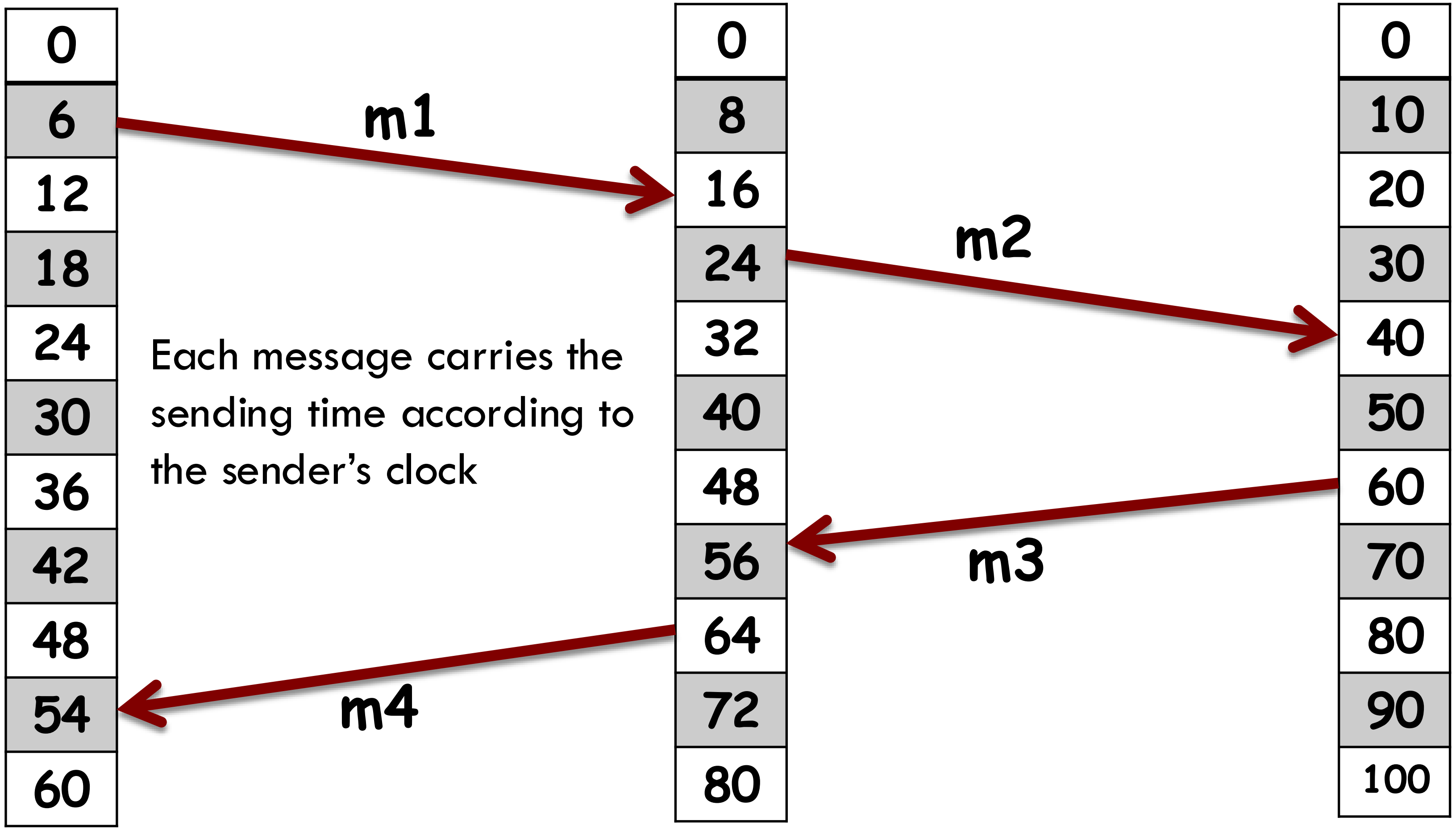
If the → relation holds between two processes

- The first event might or might-not have caused the second
  - The → relation only captures **potential causality**
    - *i.e.* two events can be related by → without a real connection between them
- EXAMPLE 1: If the server receives a request and sends a response?
  - Then reply is caused by the request
- EXAMPLE 2: A process might receive a request and subsequently issue another message
  - But this could be one that it issues every 5 minutes anyway

A simple example of Lamport timestamps

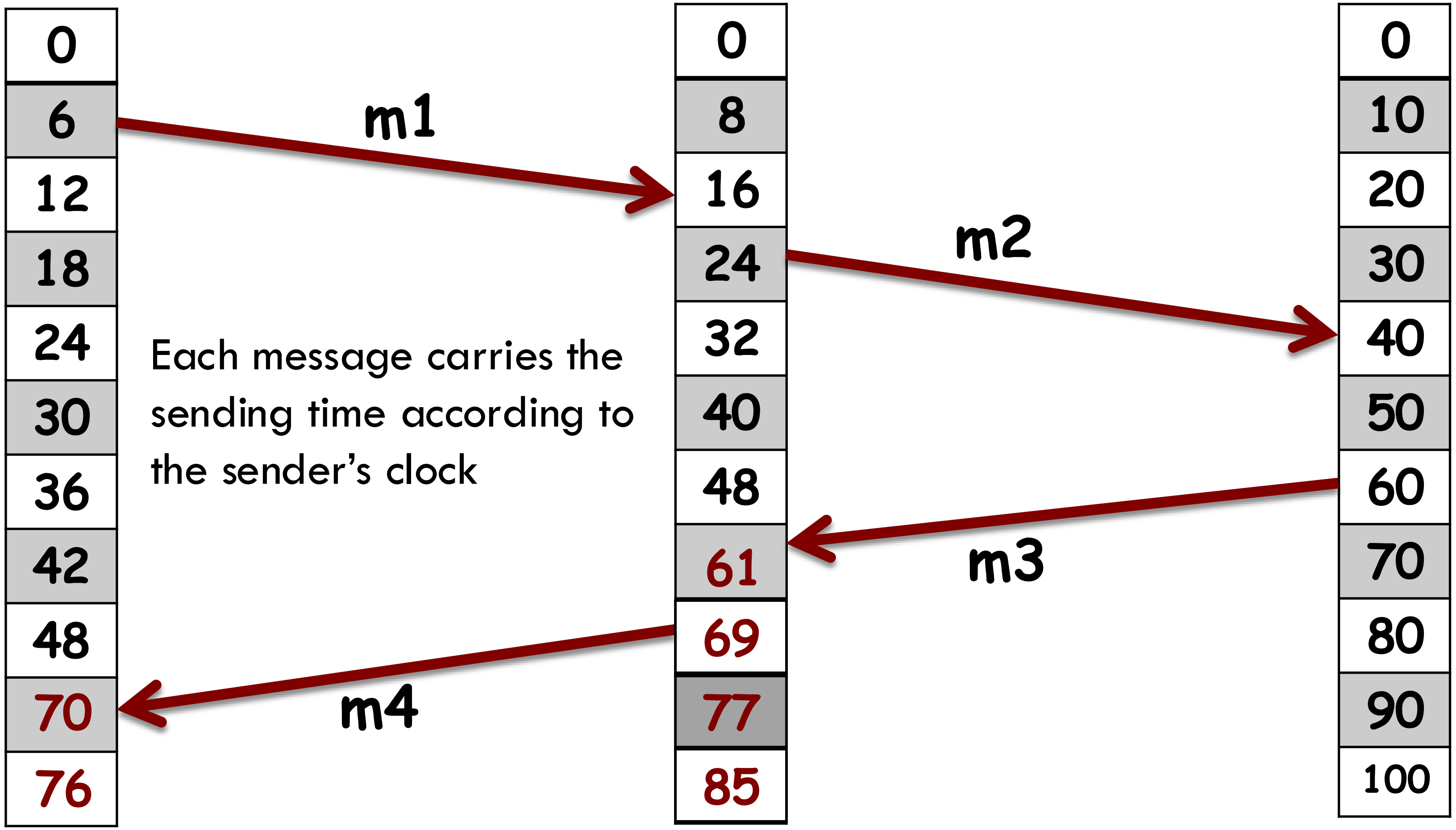


An example of Lamport's algorithm:



Each clock runs at a constant (but different rate)

An example of Lamport's algorithm:



Each clock runs at a constant (but different rate)

Implementing Lamport's clocks

① Before executing an event; P_i executes

$$C_i = C_i + 1$$

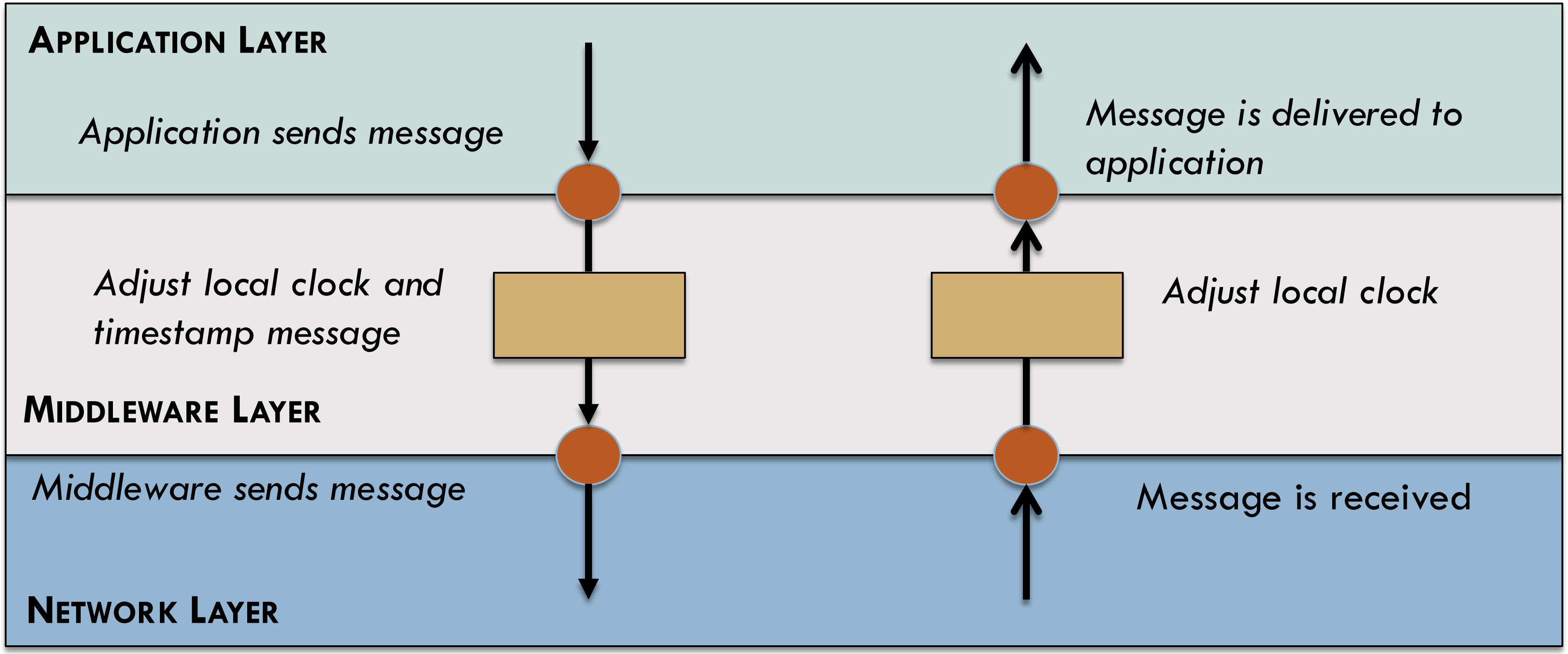
② When P_i sends a message m to P_j ; it sets m's timestamp $ts(m)$ to C_i in previous step

③ Upon receipt of message m, P_j adjusts its own local counter

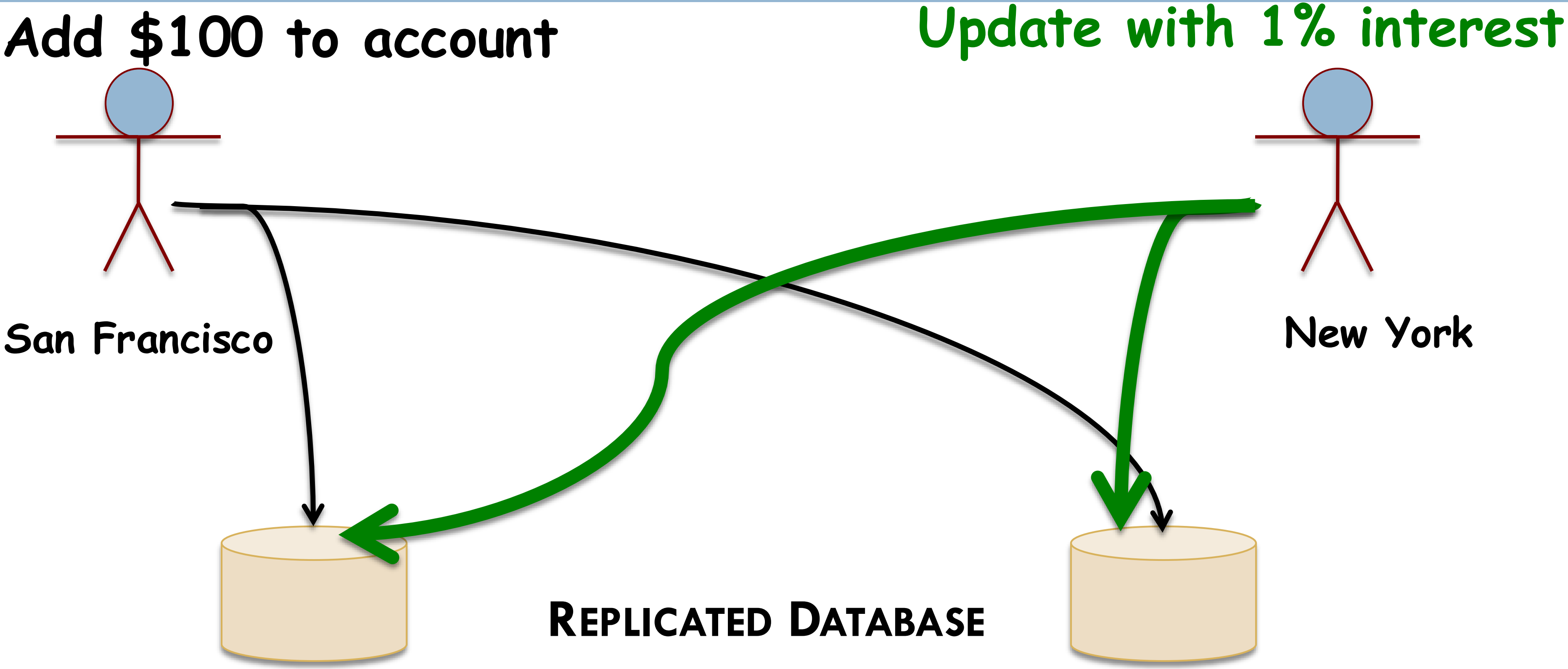
$$C_j = max\ \{C_j, ts(m)\}$$

do step (1) and deliver message

The positioning of Lamport's clocks in distributed systems



An application of Lamport's clock: User has $1000 in bank account initially



Add $100 .... Total:$1100
Give 1% interest on total= $11
Balance: $1111

Give 1% interest … Total= $1010
Add $100
Balance: $1110

There is a difference when the orders are reversed

- Our objective for now is consistency
- Both copies must be exactly the same

Use Lamport's clock to order messages

- Process puts received messages into local queue
  - Ordered according to the message's timestamp
- Message can be delivered only if it is **acknowledged** by all the other processes
- If a message is at the head of the queue, and acknowledged by all processes
  - It is delivered and processed

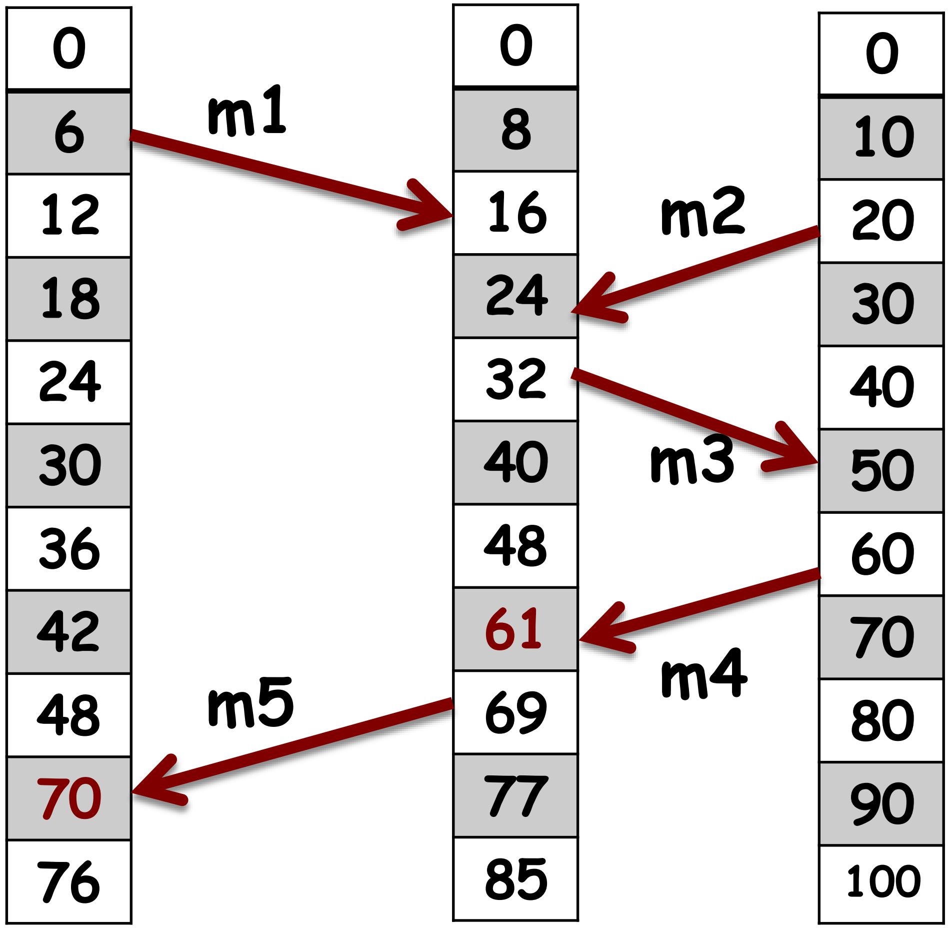
Lamport's Clocks order events based on the happened-before relationship

- If a happened before b, then $C(a) < C(b)$
- But nothing can be said about two events a and b by merely comparing their values
- $C(a) < C(b)$?
  - Does not mean a happened before b

Let's look a little closer

- $T_{snd}(m_i)$: Time m_i was sent
- $T_{rcv}(m_i)$: Time m_i was received
- $T_{snd}(m_i) < T_{rcv}(m_i)$
- BUT
  - $T_{snd}(m_i) < T_{rcv}(m_j)$?
    - NO

Concurrent message transmissions



Sending m3 MAY HAVE depended on m1

$T_{rcv}(m1) < T_{snd}(m2)$

But sending of m2 has nothing to do with receipt of m1

Lamport clocks do not capture causality

Devouring time, blunt thou the lion’s paws
And make the earth devour her own sweet brood,
Pluck the keen teeth from the fierce tiger’s jaws
And burn the long-liv’d phoenix in her blood,
Make glad and sorry seasons as thou fleet’st,
And do what e’er thou wilt, swift-footed time

Sonnet 19: Devouring time; William Shakespeare

Vector Clocks

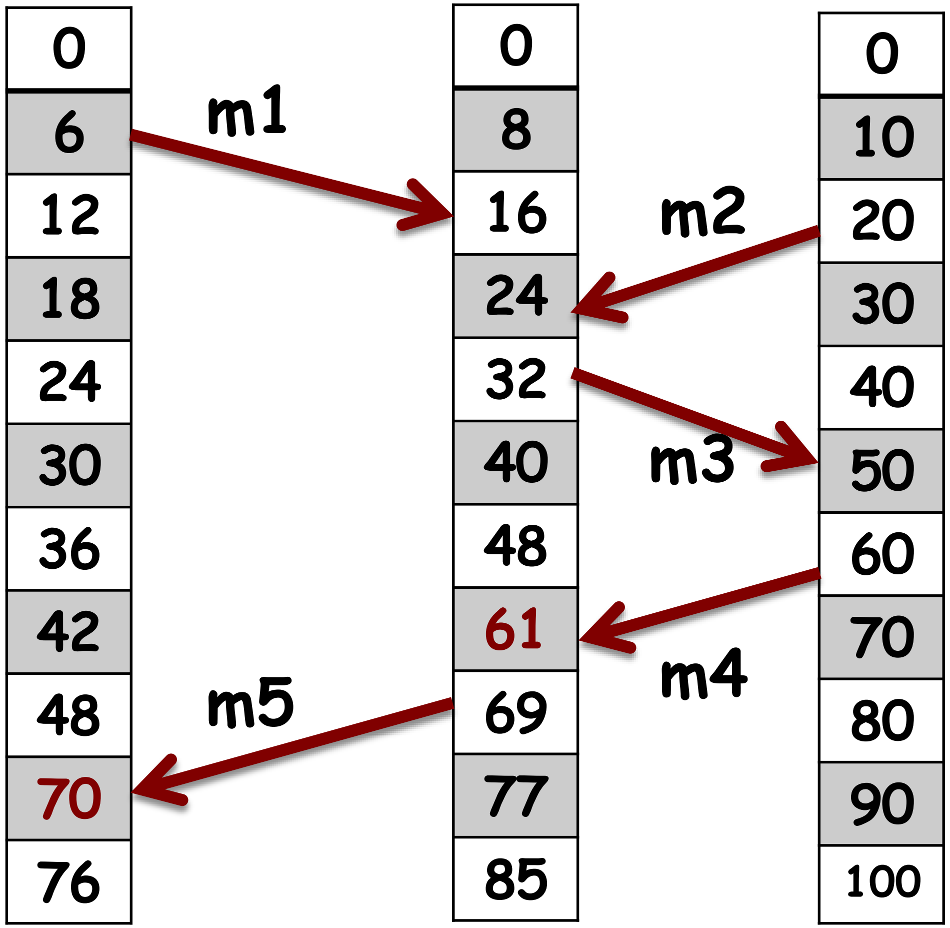
Lamport's Clocks order events based on the happened-before relationship

- If a happened before b, then $C(a) < C(b)$
- But nothing can be said about two events a and b by merely comparing their values
- $C(a) < C(b)$?
  - Does not mean a happened before b

Let's look a little closer

- $T_{snd}(m_i)$: Time m_i was sent
- $T_{rcv}(m_i)$: Time m_i was received
- $T_{snd}(m_i) < T_{rcv}(m_i)$
- BUT
  - $T_{snd}(m_i) < T_{rcv}(m_j)$?
    - NO

Concurrent message transmissions



Sending m3 MAY HAVE depended on m1

But sending of m2 has nothing to do with receipt of m1

$T_{rcv}(m1) < T_{snd}(m2)$

Lamport clocks do not capture causality

Vector clocks

- Developed by Mattern [1989] and Fidge [1991] to ***overcome shortcomings*** of Lamport's clocks
  - i.e. if $C(a) < C(b)$ *then we cannot conclude* $a \rightarrow b$
- A vector clock for a system of N processes is an **array** of N integers
- Each process keeps its own vector clock VC_i
  - Process uses it vector clock to timestamp messages

Causal precedence can be captured by Vector clocks

- Event *a* is known to **causally precede** event *b* iff $VC(a) < VC(b)$
  - $VC(a)<VC(b)$ iff $VC(a)[k] \leq VC(b)[k]$ for all *k* and *at least one* of those relationships is **strictly smaller**

- Each process P_i maintains a vector VC_i
- $VC_i[i]$ is number of events so far at P_i
- If $VC_i[j] = k$
  - P_i knows k events occurred at P_j
  - P_i's knowledge of local time at P_j

Vectors are piggybacked along with any messages that are sent

① Before executing an event (sending, delivering, or internal) P_i executes

- $VC_i[i] = VC_i[i] + 1$

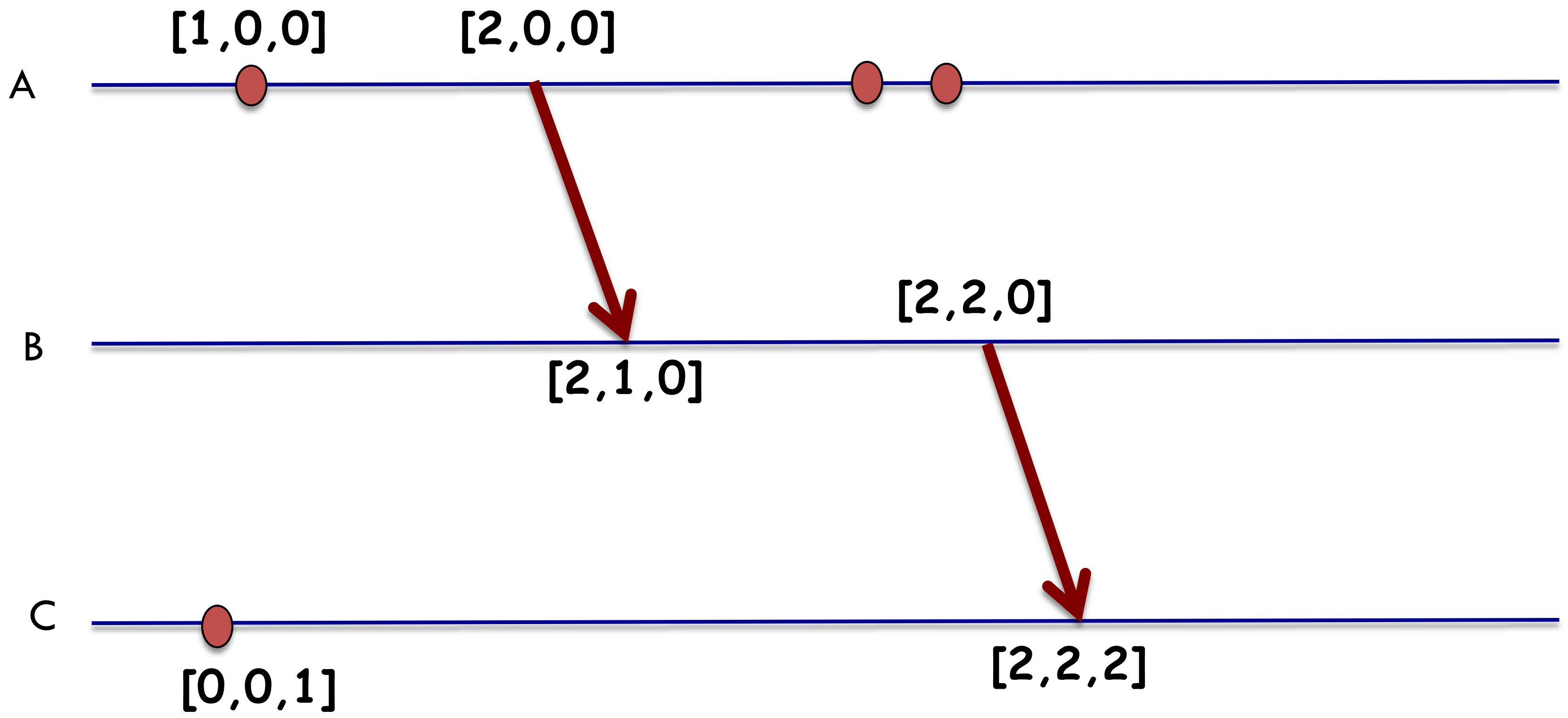
② When P_i sends a message m to P_j

- Set m's timestamp $ts(m)$ to VC_i *after* doing (1)

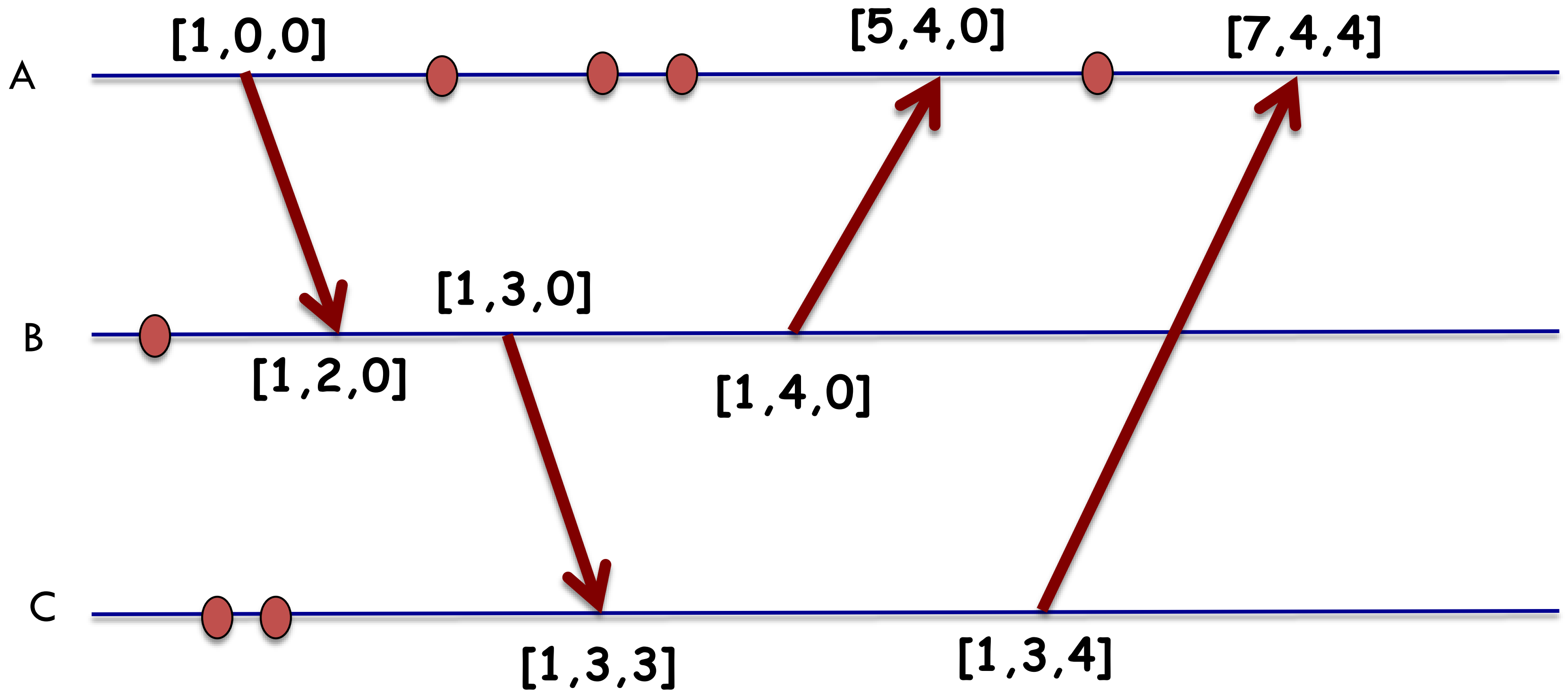
③ After receiving m, process P_j adjusts its vector

- $VC_j[k] = \max\{VC_j[k],\ ts(m)[k]\}$ for each k
- Execute step (1) and deliver

Vector clocks example 1



Vector clocks example 2



Vector timestamps allow us to determine causality and concurrency

- Event a happened before event b iff
  - $ts(a) \leq ts(b)$ for each process i
    - And one of those relationships is *strictly smaller*

- If this is not true
  - Events a and b are concurrent

Vector Clocks: Other aspects

- If event a has timestamp, $ts(a)$:
  - $ts(a)[i]-1$
    - Denotes number of events at P_i that precede a
- When P_j receives message m from P_i with timestamp $ts(m)=VC_i$
  - P_j knows about the number of events at P_i that causally preceded m
  - Also, P_j knows about how many events at *other* processes have preceded the sending of m, and on which m may causally depend

Vector clocks: Disadvantages

- Storage and message payload is proportional to N, the number of processes
- It's been shown ([Charron-Bost 1991]) that if we are to tell if two events are concurrent by inspecting timestamps?
  - The dimension of N is unavoidable

Using Vector Clocks for Causally ordered Multicasting

Contrasting totally-ordered and causally-ordered **multicasting**

- Causally-ordered multicasting is **weaker than** totally-ordered multicasting
- If two messages are *not in any way related* to each other?
  - We do not care about the order in which they are delivered to applications
  - Could be delivered in *different order* at *different applications*

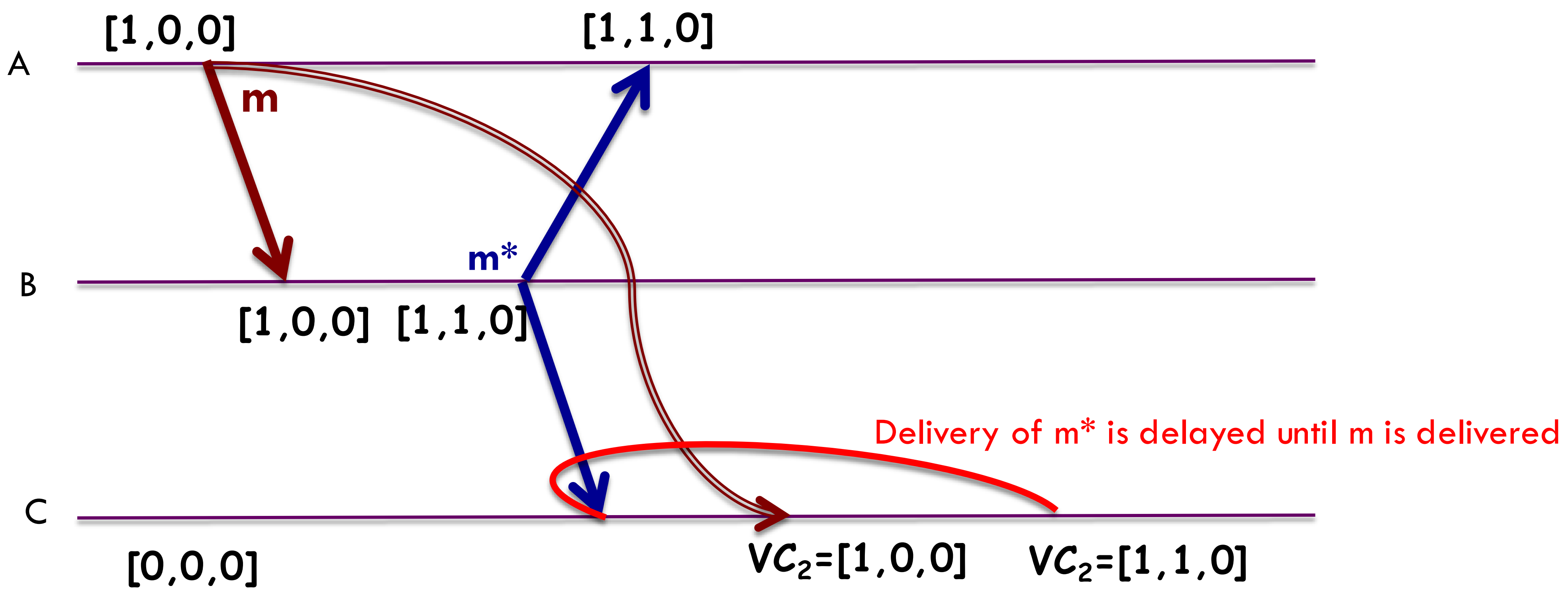
Using Vector Clocks for causally-ordered **multicasting**

- Clocks are ONLY **adjusted when sending and receiving** messages
- Upon sending a message, process P_i will only increment $VC_i[i]$ by 1
- When P_i delivers a message m with timestamp $ts(m)$ it adjusts $VC_i[k]$
  - To $max(VC_i[k], ts(m)[k])$ for each k

When process P_j receives a message m from P_i

- Delivery of the message m to the application layer is delayed until 2 conditions are met:
  1. $ts(m)[i] = VC_j[i] + 1$
     - This means m is the next message that P_j was expecting from P_i
  2. $ts(m)[k] \leq VC_j[k]$ for all $k \neq i$
     - This means that P_j has seen all messages that have been seen by P_i when it receives m

An example showing enforcement of causal communications



[Errata fixed on this slide.]

Matrix clocks

- Generalizes the notion of vector clocks
- Processes keep estimates of other processes' vector time [Raynal & Singhal, 1996]
- Essentially, a vector of vector clocks for each of the communicating processes

The contents of this slide-set are based on the following references

- *Distributed Systems: Principles and Paradigms. Andrew S. Tanenbaum and Maarten Van der Steen. 2nd Edition. Prentice Hall. ISBN: 0132392275/978-0132392273.* [Chapter 6]
- *Distributed Systems: Concepts and Design. George Coulouris, Jean Dollimore, Tim Kindberg, Gordon Blair. 5th Edition. Addison Wesley. ISBN: 978-0132143011.* [Chapter 14]
- http://en.wikipedia.org/wiki/Matrix_clocks