1. CSV888 - Distributed Systems

Three Models

1. Time Order
2. Distributed Algorithms
3. Nature of Distributed Systems

Index - Models to study [2]

- • 1. LAN based systems (in Distributed Systems)
- • 2. Web based systems
- • 1. –! uses synchronous communications
- • 2. –! uses Asynchronous communications
- - Time-Stamp order ?
- • Other Algorithms
- - Transaction Recovery ?
- - System Recovery ?

[I] Index 2 - Models to study [3]

- 1. Global time and event order (in Distributed Systems)

- 2. Distributed Mutual Excusion

- Concept of Transactions

- Concurrency Control

- Locking
- Validation
- Time-Stamp order

- Atomicity

- Transaction Recovery
- System Recovery

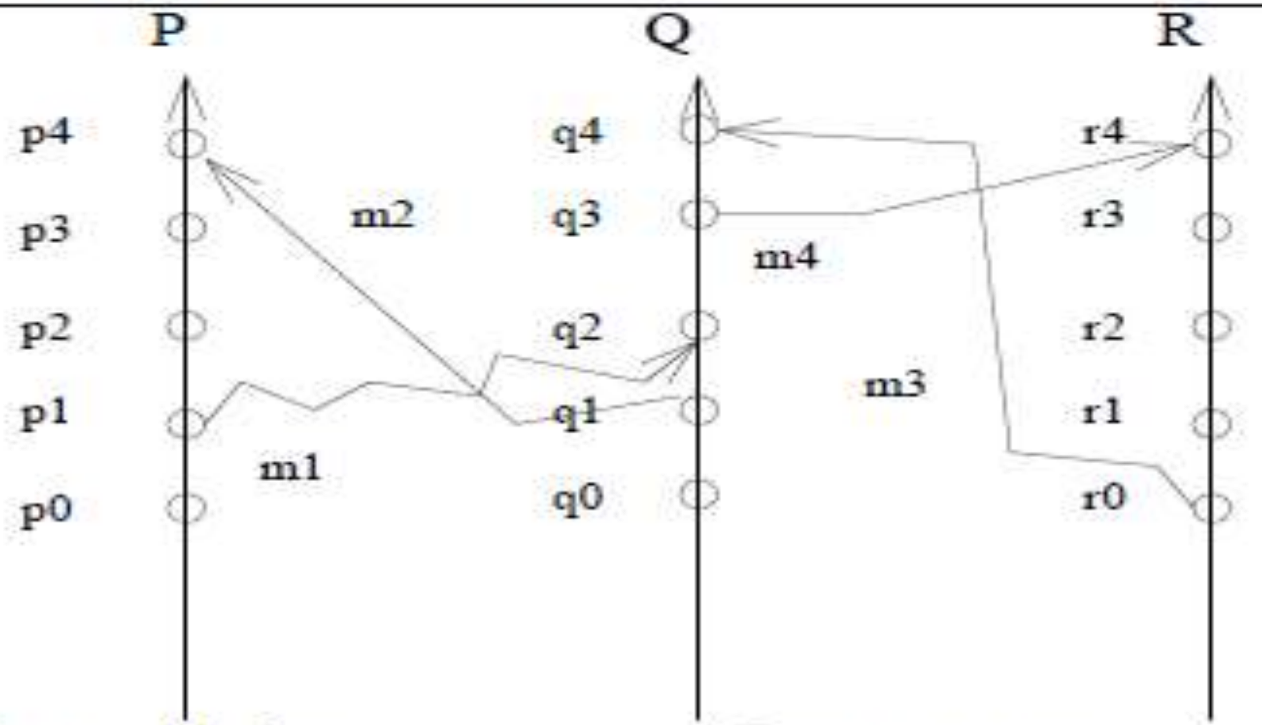
[I] Models → Implementation [4]

Models	• [A] Fail-stop model : Atomicity • [B] Fail-stop channel • [C] transaction model - 4 properties: isolation, consistency, atomicity, durability • [D] Time-event ordering • [E] Distributed Mutual Excusion
Lower level Implementation	• [A] 2 pahse commit • [B] Parity check • [C] transaction model implementation - - Data Locking, 2 phase commit, ... • [D] Events and Numbering protocols • [E] centralized/distributed: different protocols

[I] Event Order [5]

Time	• which event is before and which is after
order	• No global Clock
event	• [A] Happened-before relation ($\rightarrow$) • [B] If A and B are events in the same process, and A was executed before B, then A $\rightarrow$ B • [C] If A is the event of sending a message by one process and B is the event of receiving that message by another process, then A $\rightarrow$ B • [D] If A $\rightarrow$ B and B $\rightarrow$ C then A $\rightarrow$ C

[I] Relative Time for 3 processes



Happened−before:

p1 ⟶ q2

r0 ⟶ q3

q3 ⟶ r4

p1 ⟶ q4 (transitive order p1 −>q2 and q2−> q4)

Concurrent events:

q0 and p2, r0 and q3

r0 and p3, q3 and p3

[I] Implementation of $\rightarrow$ [7]

- Associate a timestamp with each system event.

- Require that for every pair of events A and B,
- if, A $\rightarrow$ B,
- then time-stamp of A is less than time-stamp of B

- Within each process P_i a logical clock LC_i is associated

- simple counter for each event inside a process

- A process advances its logical clock when it receives a message whose timestamp is greater than the current value of its logical clock.

- If the timestamps of two events A and B are the same,

- then the events are concurrent.
- use the process identity numbers to break ties and
- to create a total ordering.

Ordering events in a distributed system

- Lamport → Ordering events in a Distributed System (Lamport Clock)

- Vector Clock

Agenda

- Physical Clock
- Logical Clock

- Logical Clock algorithm
  Lamport’s logical clock

  Vector clock

Physical Clocks

Need for physical clocks

Processors share a common bus → The entire system shares the same understanding of time: It is consistent.

Physical clock - Multiple systems

In distributed systems, →

each system has its own timer that drives its clock.

Each timer might change with time, temperature, etc.

This implies each systems time will drift away from the true time (at a different rate).

Logical Clocks

- Messages sent between machines may arrive zero or more times at any point after they are sent.

- If two machines do not interact, no need to synchronize them

Can we order the events on different machines using local time?

Causality →

The purpose of a logical clock **is not** necessarily to maintain the same notion of time as a reliable watch !

Aim, is to keep track of information about the order of events

Lamport's logical clock

Key Ideas

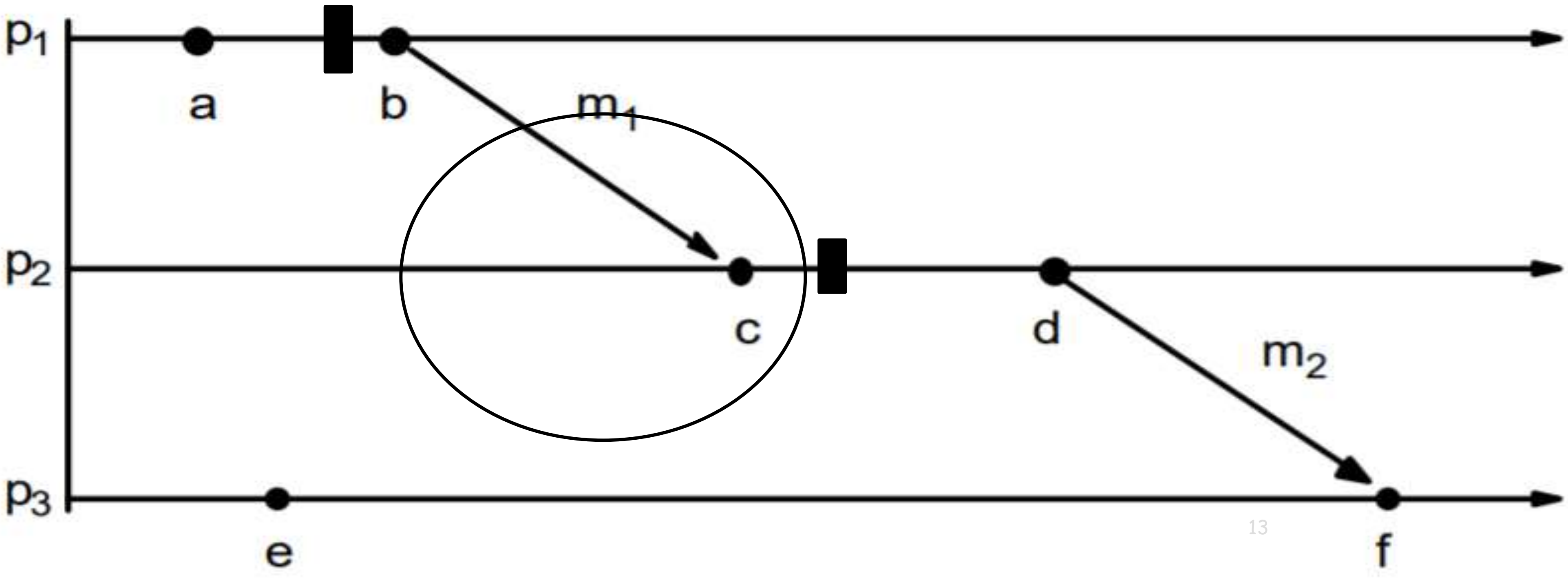
- Processes exchange messages
- Message must be sent before received
- Send/receive are used to order events and to synchronize clocks

- Happened before relation
- Causally ordered events
- Concurrent events
- Implementation
- Limitation of Lamport's clock

Lamport's logical clock

Happened before relation

- a -> b : Event a occurred before event b. Events in the same process p1.
- b -> c : If b is the event of sending a message m1 in a process p1 and c is the event of receipt of the same message m1 by another process p2.
- a -> b, b -> c, then a -> c; "->" is transitive.



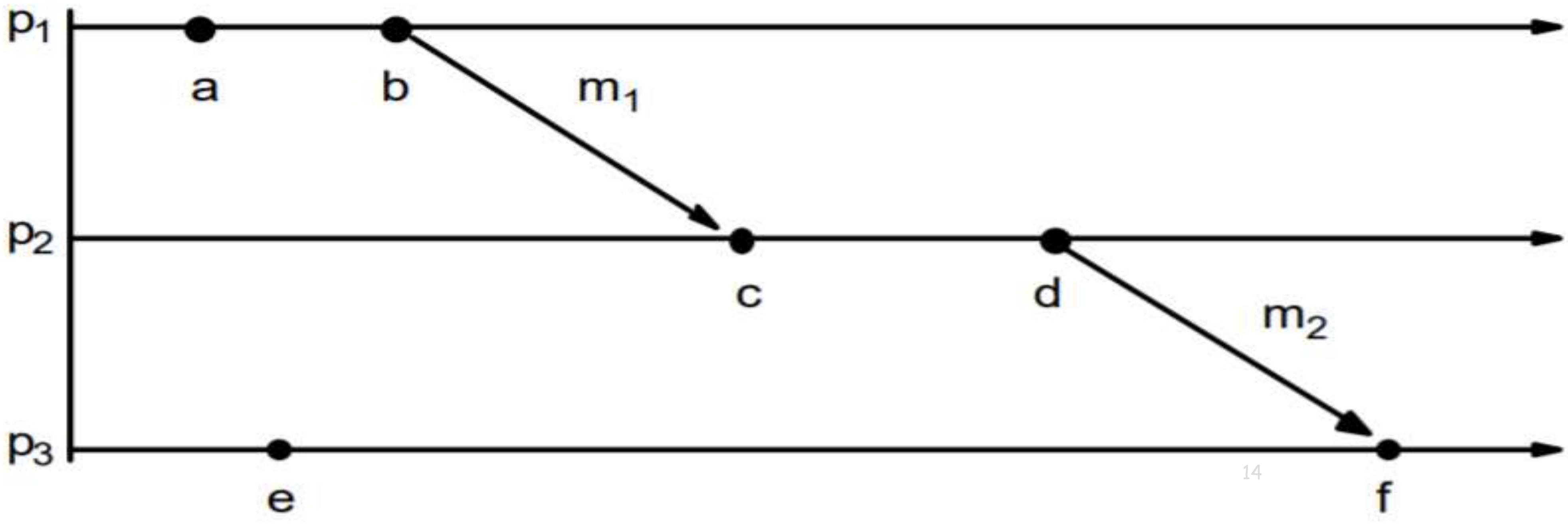
Lamport's logical clock

Causally Ordered Events

a -> b : Event a "causally" affects event b

Concurrent Events

a || e: if a !-> e and e !-> a



Lamport's logical clock

Algorithm

Sending end

```
time = time+1;
time_stamp = time;
send(message, time_stamp);
```

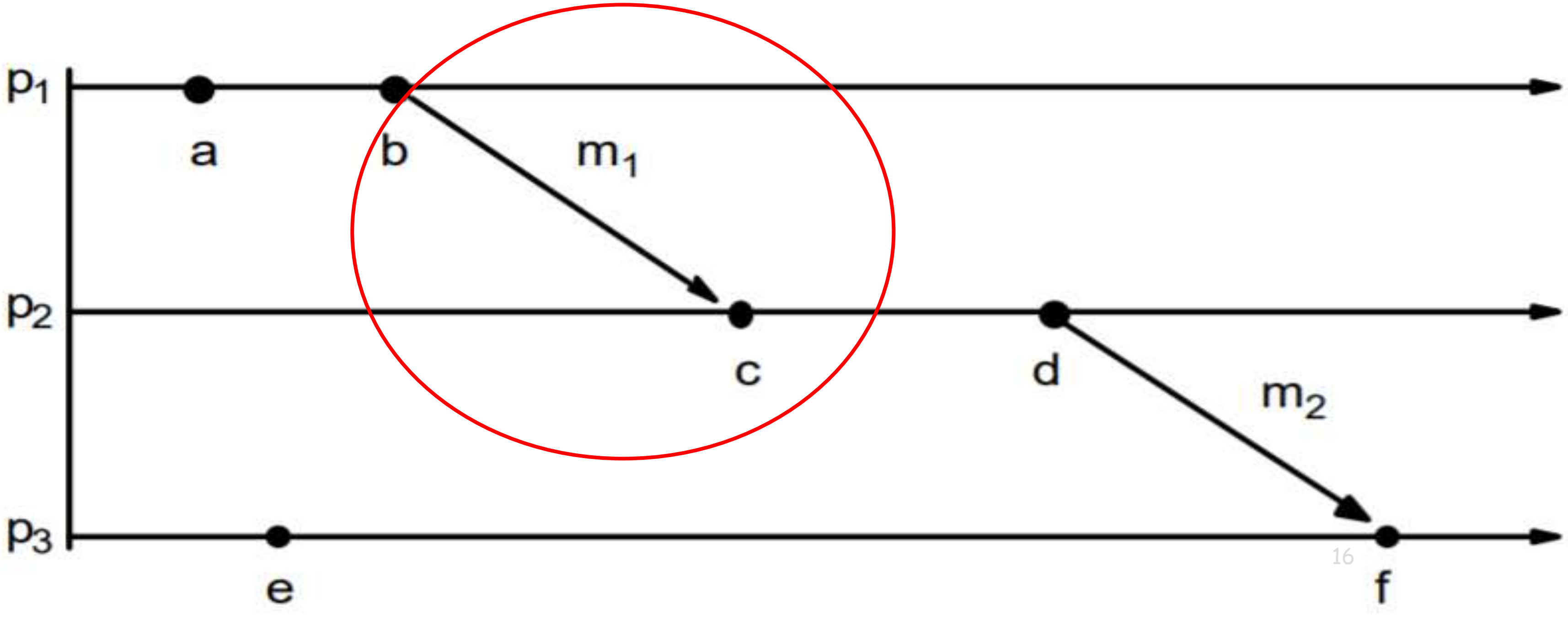
Receiving end

```
(message, time_stamp) = receive();
time = max(time_stamp, time)+1;
```

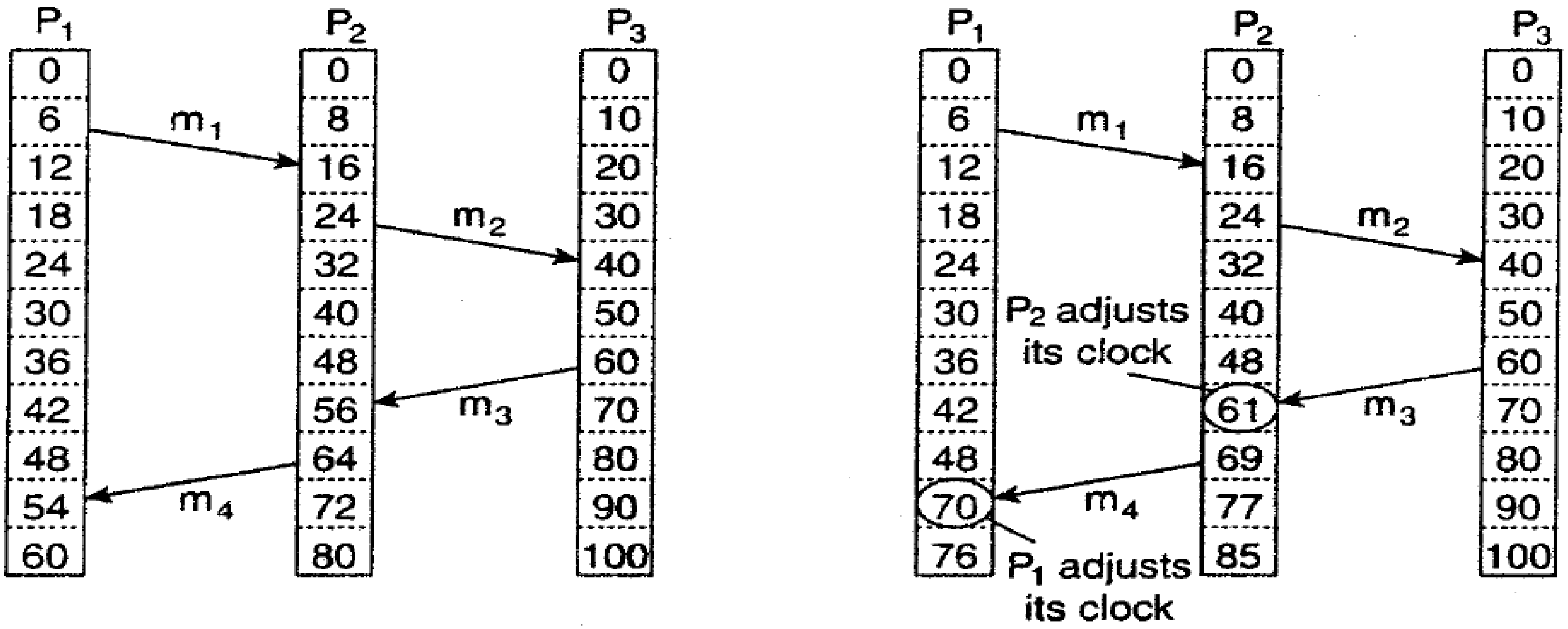
Lamport’s logical clock

a -> b C(a) < C(b)

b -> c C (b) and C(c) must be assigned in such a way that C(b) < C(c) and the clock time, C, must always go forward (increasing), never backward (decreasing). Corrections to time can be made by adding a positive value, never by subtracting one.

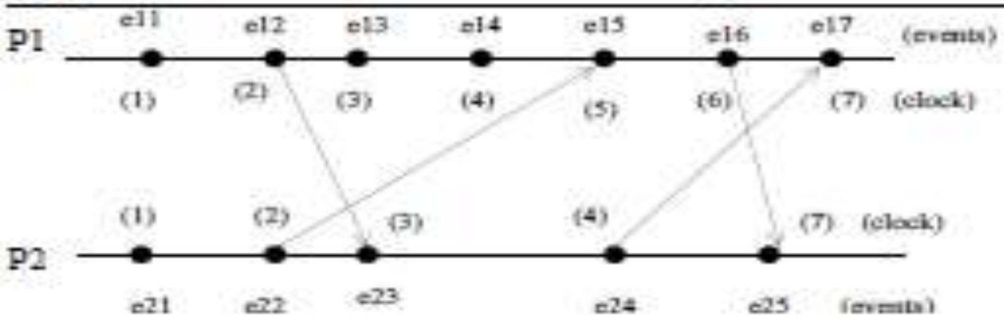


Lamport’s logical clock



An illustration: Three processes, each with its own clock. The clocks run at different rates and Lamport's algorithm corrects the clocks.

[I] Example of logical clocks [8]



clocks under Lamport's scheme

- Figure shows an example of how logical clocks are updated under Lamport's scheme.
- Clocks at P1 and P2 are assumed to be at zero initially. Clocks are incremented by 1 each time by a fixed increment value (increment value, d = 1).

Rules for updating the clocks are:

Rule 1 - If a and b are two successive events at process P, and **event a** "happens before" **event b** ($a \rightarrow b$), then clock value $C(b) = C(a) + d$.

Rule 2 - If the receiving process receives a message with time of message $tm = C(a)$, then new local clock value is found by -

$C(b) = Max(C(localclock), tm) + d$

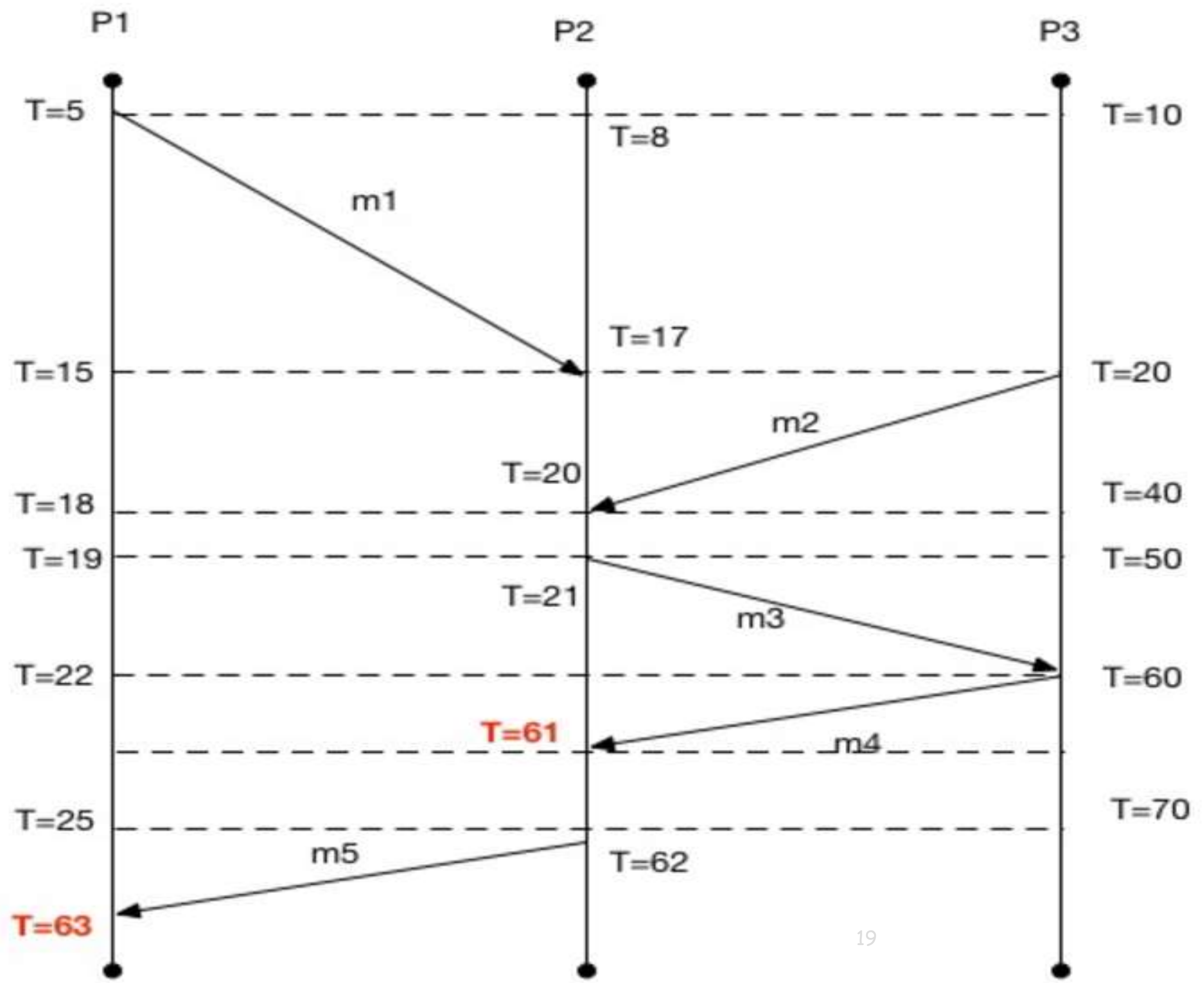
Lamport's logical clock

Limitations

- m1−>m3
  C(m1)<C(m3)

- m2−>m3
  C(m2)<C(m3)

m1 or m2 caused m3 to be sent?



Lamport’s logical clock

- Lamport’s logical clocks → **all events in a distributed system are totally ordered.**
  That is, if a -> b, then we can say C(a)<C(b).

- Lamport’s clocks → **nothing can be said about the actual time of a and b.**
  logical clock says a -> b, that does not mean in terms of real time.

- Lamport clocks → do not capture causality.

- **If a -> c and b -> c we do not know which action initiated c.**

- → **Problems** : when trying to replay events in a distributed system
  (such as when trying to recover after a crash).

- The theory goes that if one node goes down, if we know the causal relationships between messages, then we can replay those messages and respect the causal relationship to get that node back up to the state it needs to be in.
  → **Piece-wise Deterministic (PWD) ?**

Vector clocks

Vector clocks allow causality to be captured

- Rules of Vector Clocks

- Properties of a process
- Implementation

Vector clocks

Rules and properties

- A vector clock VC(i) is assigned to an event i.

- If VC(i)<VC(j) for events i and j, then event i is known to causally precede j.

- Each process i maintains a vector V such that
  - Vi [i] : number of events that have occurred at i
  - Vi [j] : number of events I knows have occurred at process j

Vector clocks

Implementation

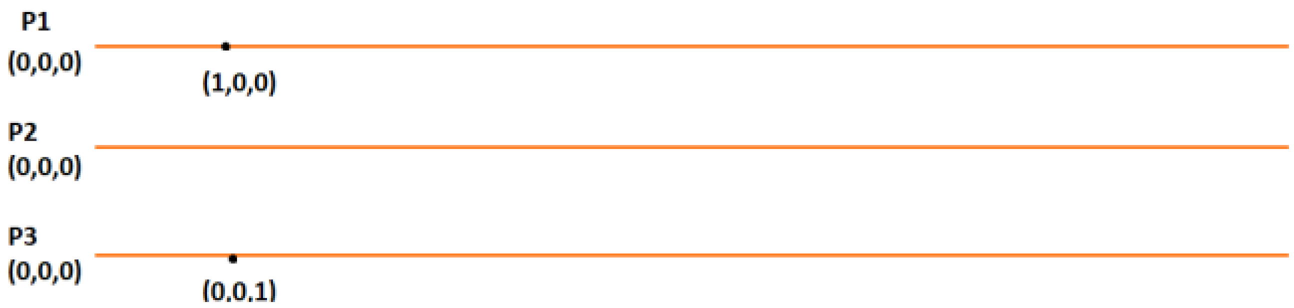
Before executing an event (i.e., sending a message over the network, delivering a message to an application, or some other internal event),

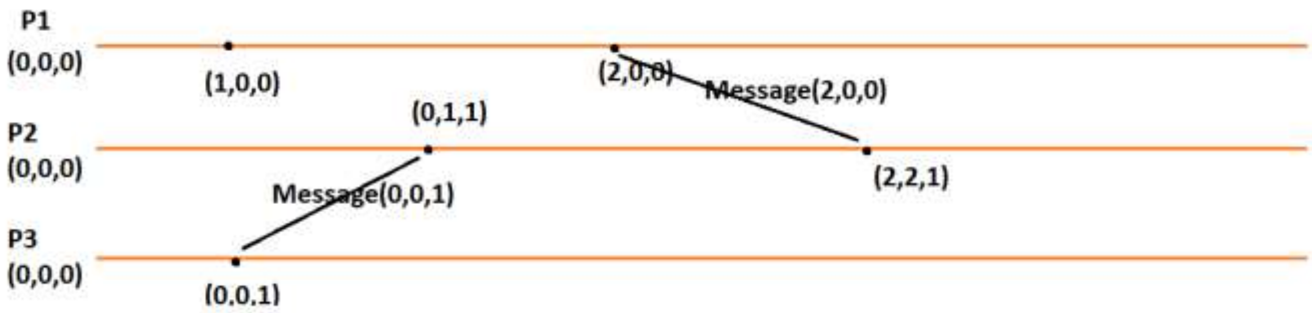
1. Pi executes VCj [i] ~ VCj [i] + 1.

2. When process Pi sends a message m to Pj, it sets m's (vector) timestamp ts (m) equal to VCj after having executed the previous step.

3. Upon the receipt of a message m, process lj adjusts its own vector by setting VCj [k] ~ max{VCj [k], ts (m)[k]} for each k, after which it executes the first step and delivers the message to the application.

Vector clocks





Sum Up: for Checkpoints and Recovery → To Prevent Orphan process

Lamport’s timestamps

- Integer clocks assigned to events
- Obeys causality
- Cannot distinguish concurrent events

Vector timestamps

- Obeys causality
- By using more space, can also identify concurrent events

Model 2 – Fully Distributed Algorithm

- Web Services: (Web Services – Business Applications)
- No central coordination (except UDDI)
- Fully Distributed (LAN based systems ?): ex: 2-phase commit ?
- Pair-wise interaction
- Peer – to - peer

[I] Distributed Mutual Exclusion (DME) [10]

problem	• Share common resources - example, PRINTER
system	• Co-ordination among processes at different sites
outline	• [A] Assumptions - - The system consists of n processes; each process Pi resides at a different processor. - Each process has a critical section that requires mutual exclusion. • [B] Requirement - - If Pi is executing in its critical section, then no other process Pj is executing in its critical section. • [C] Two approaches - • Centralized approach • Distributed approach

[I.E] DME: Centralized Approach [11]

One of the processes in the system is chosen to **coordinate** the entry to the critical section.

A process that wants to enter its critical section sends a request (message → coordinator.

The coordinator decides which process can enter the critical section next, and it sends (reply message → that process)

When the process receives a reply message from the coordinator, it **enters its critical section.**

[I.E] DME: Centralized Approach (contd.) [12]

After exiting its critical section, the process sends (release message $\rightarrow$ coordinator) and proceeds with its execution.

This scheme requires three messages per critical-section entry:
request,
reply,
release

[I] DME: Fully Distributed Approach [13]

- When process Pi wants to enter its critical section, it generates a new timestamp, TS , and sends the message request (Pi, TS) $\rightarrow$ all other processes.
- When process Pj receives a request message, it may reply immediately or it may defer sending a reply back.
- When process Pi receives a reply message from all other processes in the system, it can enter its critical section.
- After exiting its critical section, the process sends reply messages to all its deferred requests.

[I] Fully Distributed Approach (Cont.) [14]

• The decision whether process Pj replies immediately to a request (Pi, TS) message or defers its reply is based on three factors:

If Pj is in its critical section, then it defers its reply to Pi.

If Pj does not want to enter its critical section, then it sends a reply immediately to Pi.

If Pj wants to enter its critical section but has not yet entered it, then it compares its own request timestamp with the timestamp TS.
- If its own request timestamp is greater than TS, then it sends a reply immediately to Pi (Piasked first).
- Otherwise, the reply is deferred.

[I] Good Points - Fully Distributed Approach [15]

- Freedom from Deadlock is ensured.
- Freedom from starvation is ensured, since entry to the critical section is scheduled according to the timestamp ordering. The timestamp ordering ensures that processes are served in a first-come, first served order.
- The number of messages per critical-section entry is

  $2 \times (n - 1)$.

This is the minimum number of required messages per critical-section entry when processes act independently and concurrently.

[I] Three Undesirable Problems [16]

- The processes need to know the identity of all other processes in the system, which makes the dynamic addition and removal of processes more complex.

- If one of the processes fails, then the entire scheme collapses. This can be dealt with by continuously monitoring the state of all the processes in the system.

- Processes that have not entered their critical section must pause frequently to assure other processes that they intend to enter the critical section. This protocol is there fore suited for small, stable sets of cooperating processes.

Three Undesirable Problems

- Problem 1: Identity of the participating processes

  a) 2-phase commit ?

  b) Distributed deadlocks ?

Problem 2: Network Status ? Delay in the process OR Failure ?

Block-chain

Problem 3: Slow processes ! Scalability …

Model 3: Nature of Distributed Systems - Complexity in Distributed Systems

- Multiple Nodes
- Messages
- Modes of Communication: Sync / Async

- Consider: ebay (cart) dealing with, a customer, booking a HP notebook, Sony Camera, Cannon color printer/scanner, UPS, ..

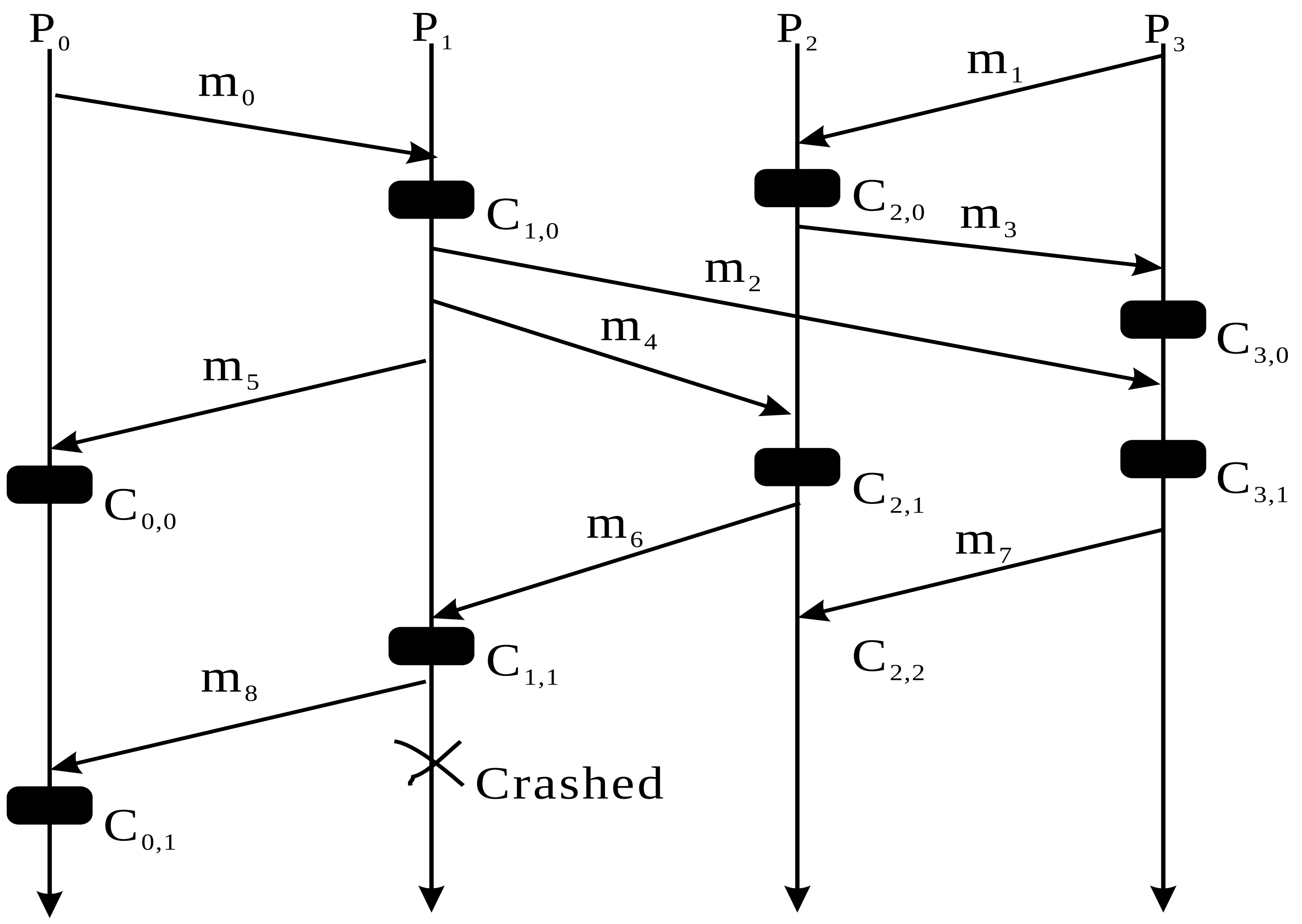
  Requires a 2-phase commit

Problems: Long Running Process

- Blue Gene – (1999) parallel computer, for the study of bio-molecular phenomena such as protein folding

- Checkpoint (1, …., n) : **STABLE STORE** ← Data, threads, register values
- Run-time **overhead**; **Failure** → most recent **checkpoint**
- **64 x 64 grid of parallel computers → middleware for checkpoints**

Cooperating Processes →Distributed System



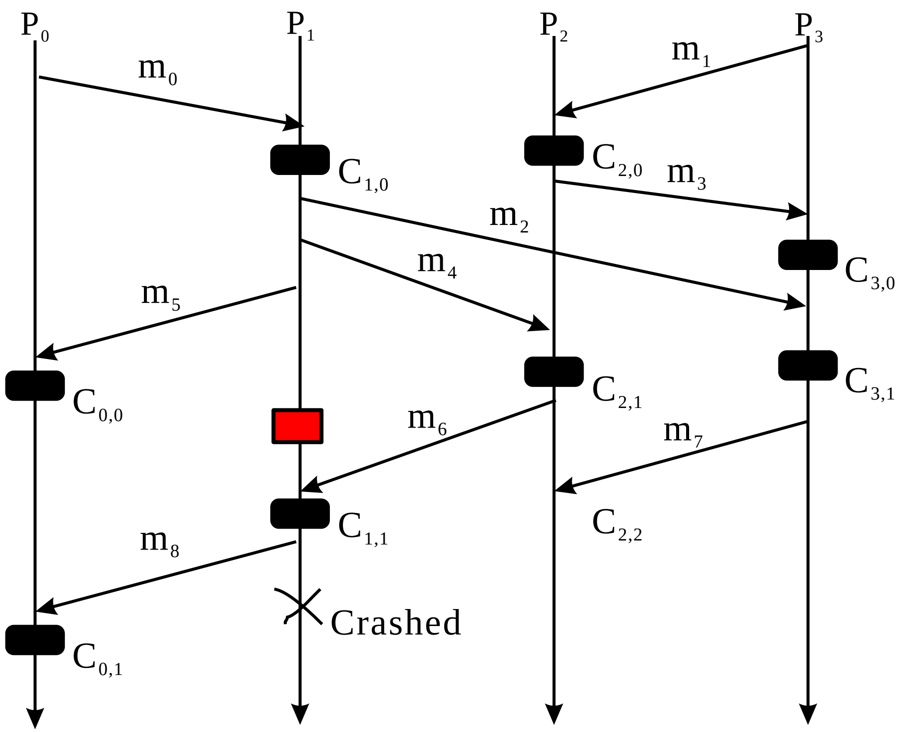
Middleware ←→ Distributed System

- Distributed system → a **collection of processes** that communicate through **messages** in a network
- Fault tolerance → periodically using stable storage to **save the processes**' states during the failure-free execution.
- After a failure → a failed process restarts from one of its saved states,

  → reducing the amount of lost computation.
- Each of the saved states is called a **checkpoint**

Checkpont → Cascading Rollback Problem

- Last checkpoint: $C_{1,1}$ by P1, before P1 crashed
- Cannot use $C_{0,1}$ at P0 because it is inconsistent with $C_{1,1}$
  => P0 rollbacks to $C_{0,0}$
- Cannot use $C_{2,1}$ at P2 because it fails to reflect the sending of m6
  => P2 rollbacks to $C_{2,0}$



- Cannot use $C_{3,1}$ and $C_{3,0}$ as a result => P3 rollbacks to initial state

Checkpoint based Recovery: Overview

- Uncoordinated checkpointing: →

  Processes take checkpoints independently

- Coordinated checkpointing: Process coordinate their checkpoints → to save a system-wide consistent state.

  → Such checkpoints can be used to bound the rollback

- Communication-induced checkpointing: It forces each process to take checkpoints based on information piggybacked on the application messages it receives from other processes.

Outline: Checkpoints

- Checkpointing and logging
  - Checkpoint-based protocols
    - Uncoordinted checkpointing
    - Coordinated checkpointing
  - Logging-based protocols
    - Pessimistic logging
    - Optimistic logging
    - Causal logging

Uncoordinated Checkpointing

- Uncoordinated checkpoints:

  → full autonomy, and simple.

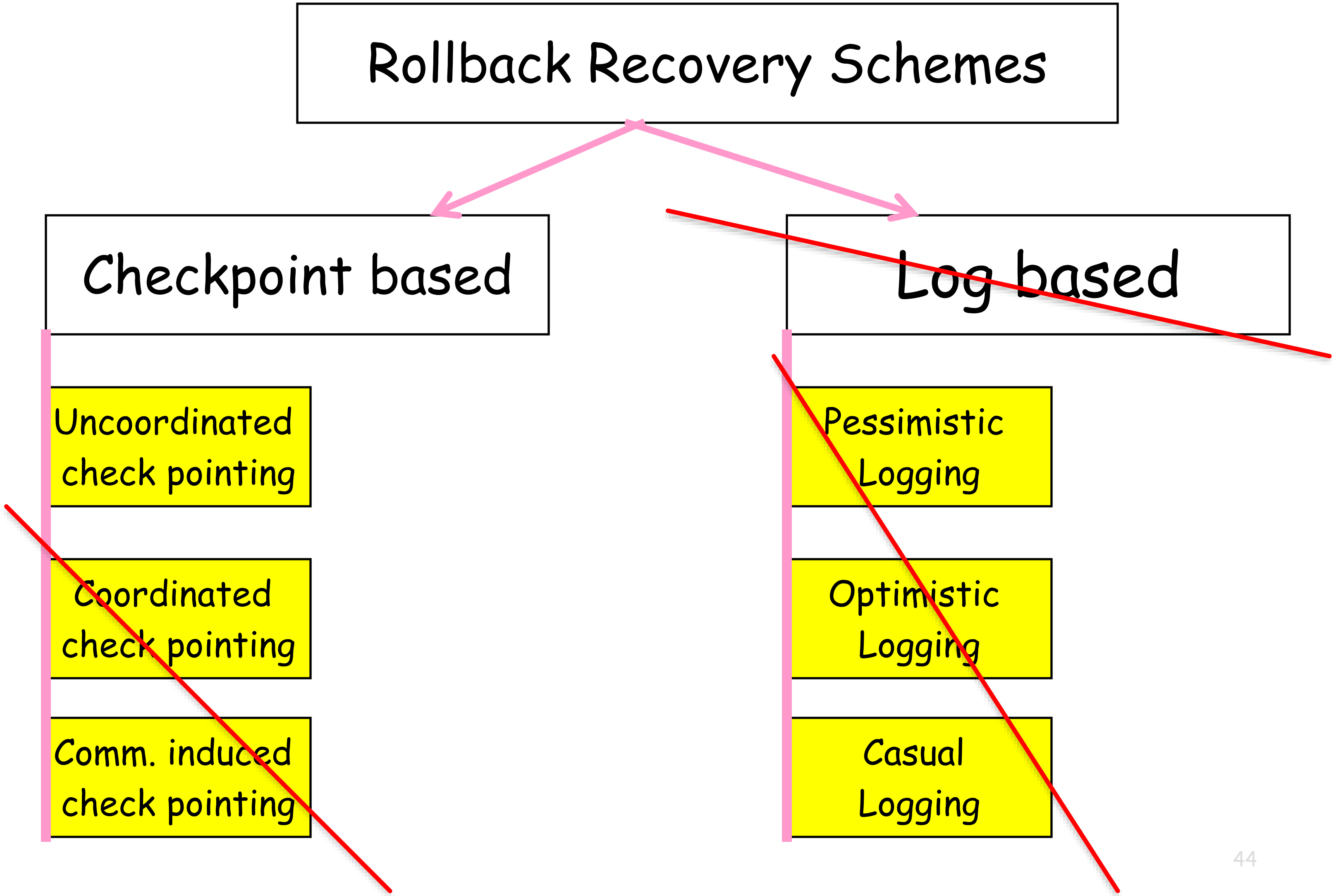
- Problems
  - Most Checkpoints are not be useful
    - **Cascading rollback** to the initial state (**domino effect**)
  - To select a set of consistent checkpoints during a recovery, the dependency of checkpoints has to be determined and recorded together with each checkpoint
    - Extra overhead and complexity => not simple after all

Disadvantages of Uncoordinated Checkpointing

- Susceptible to the domino effect

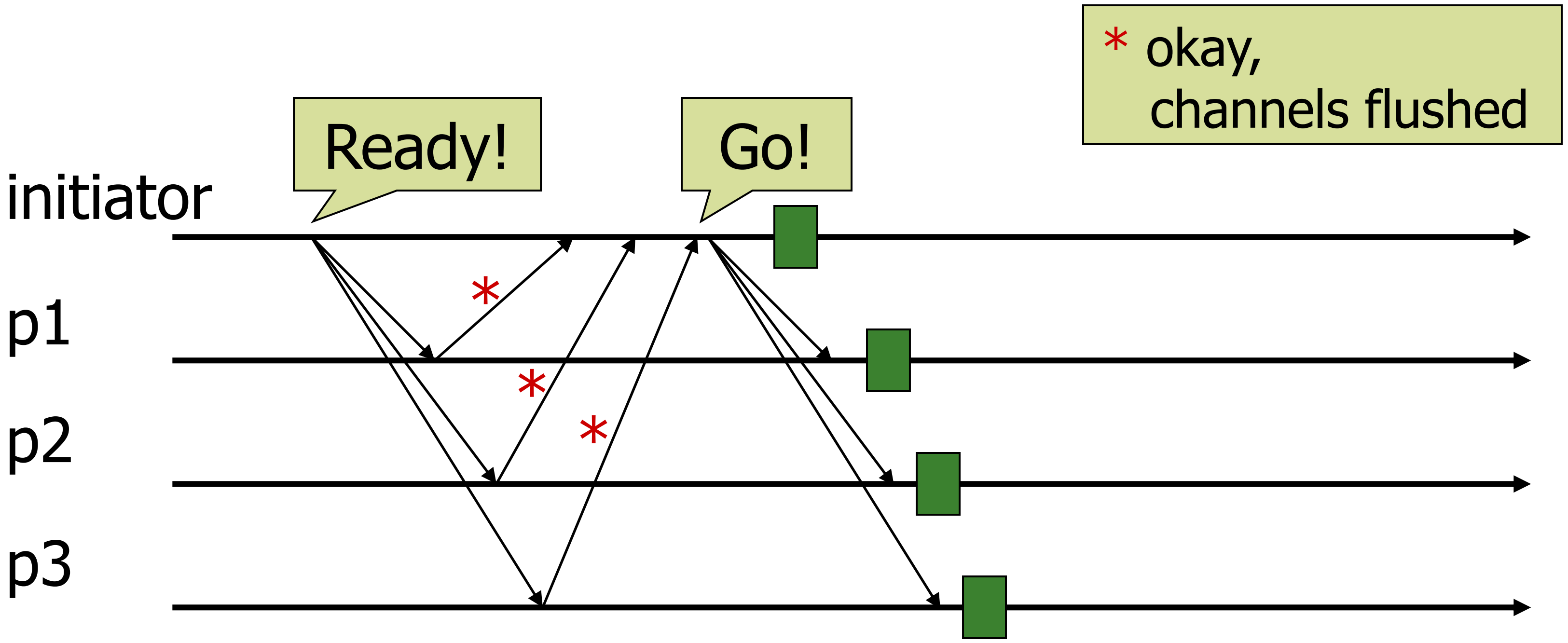
- Checkpoints that will never be part of a global consistent state are recorded
  - Stable Storage overhead
  - do not advance the recovery line
- A process needs to **maintain multiple checkpoints** and to use **garbage collector** to reclaim checkpoints
- Not suitable for output commit, because **output commit requires global coordination** to compute the recovery line

Different Rollback Recovery Schemes



Coordinated Blocking

- Processes are coordinated to form a consistent global state, and …



Next: Coordinated Blocking Chkpnt (cont')

Coordinated Blocking (cont')

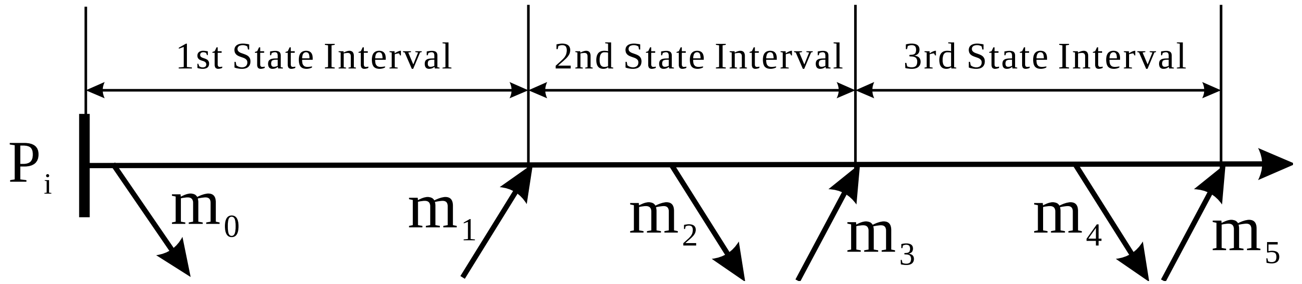
- Advantage
  - Always consistent
  - No Domino Effect
  - Less storage overhead

- Disadvantage
  - Large latency to chkpnt!

Next: Coordinated Non-blocking Chkpnt

Log Based Protocols

- Work might be lost upon recovery using checkpoint-based protocols
- By logging messages, we may be able to recover the system to where it was prior to the failure
- System mode: the execution of a process is modeled as a set of consecutive state intervals
  - Each interval is initiated by a nondeterministic state or initial state
  - We assume the only type of nondeterministic event is receiving of a message



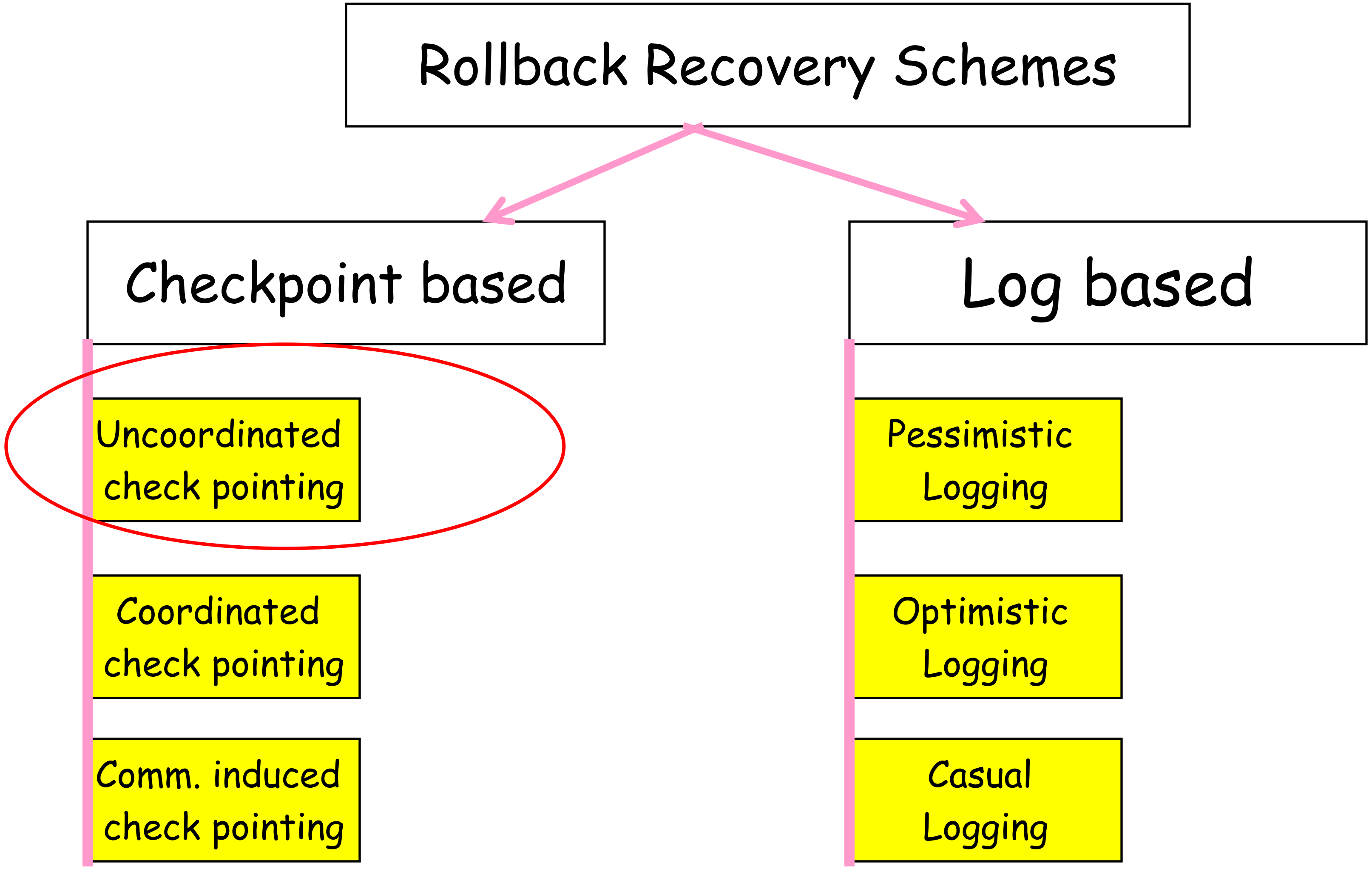
Log Based Protocols

- In practice, logging is always used together with checkpointing
  - Limits the recovery time: start with the latest checkpoint instead of from the initial state
  - Limits the size of the log: after taking a checkpoint, previously logged events can be purged
- Logging protocol types:
  - Pessimistic logging: msgs are logged prior to execution
  - Optimistic logging: msgs are logged asynchronously
  - Causal logging: nondeterministic events that not yet logged (to stable storage) are piggybacked with each msg sent
- For optimistic and causal logging, dependency of processes has to be tracked => more complexity, longer recovery time

Pessimistic Logging

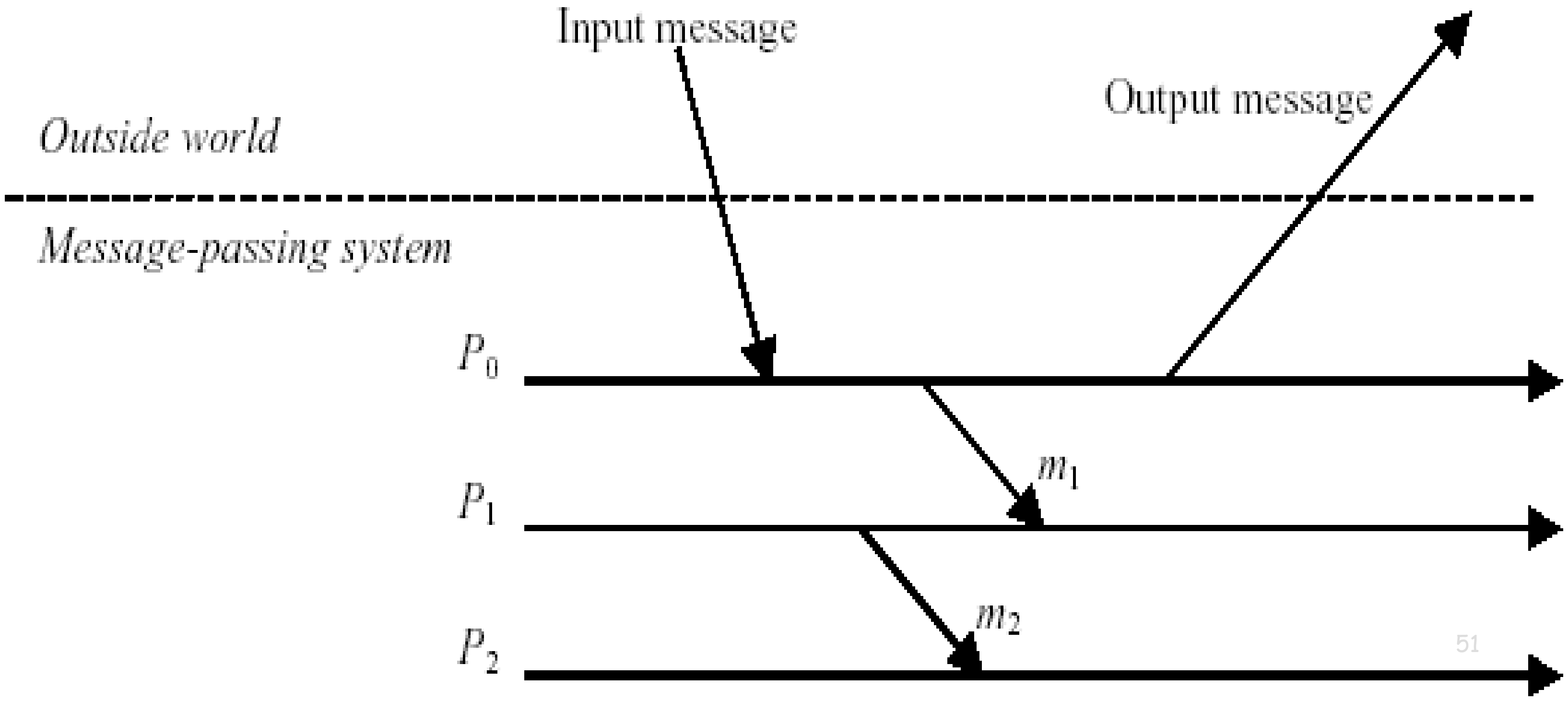
- Synchronously log every incoming message to stable storage prior to execution
- Each process periodically checkpoints its state: no need for coordination
- Recovery: a process restores its state using the last checkpoint and replay all logged incoming msgss

Different Rollback Recovery Schemes



Dependency Tracking: System Model

- A constant number of processes (*N*)
  - Communicate through **messages to Cooperate**
  - **Interact with outside world through messages**



Adoption of Blockchains

- Asynchronous Communication
- Unpredicable Network Delays
- Complexity in Distributed systems- Backup and recovery
- No recent study

  - Studies consider reliable LAN based networks

[I] Class Exercise: Lamport's Clock [9]

- Find out the logical clock values for events in the Figure.
- Find out all "happens before" relationships in Figure.
- Is there any difference between clock and event order at events e32 and e33.

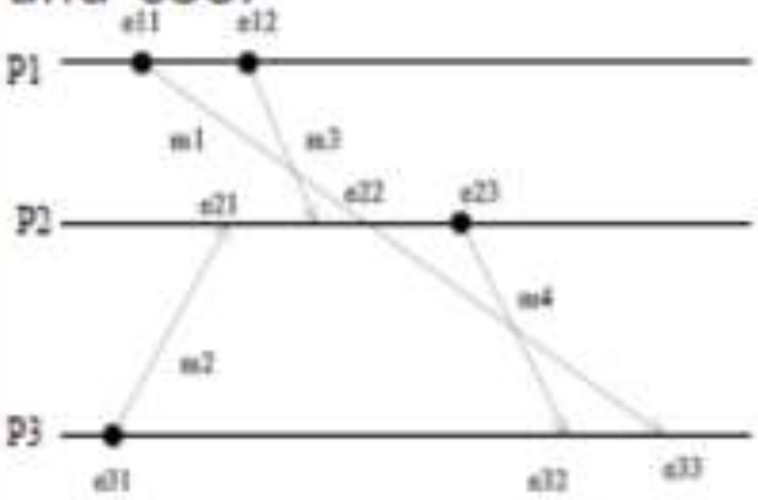


Figure - Events in a Distributed System