Course 2G1126 Distributed Algorithms

Luc Onana Alima
KTH/IMIT/LECS/DCS

Spring 2003

Objectives

- Understand some of the fundamental aspects of distributed systems
- Focus is on **algorithmic** aspects
- Learn how to read research papers

Achieving the objectives

- 14 lectures mainly based on (text book)
  - *Distributed Operating Systems & Algorithms*
    - Randy Chow and Theodore Johnson, Addison Wesley, 1997

- Personal work
  - You will have to study one or two research papers

General information

- Staff
  - Lecturer
    - Luc Onana Alima: onana@imit.kth.se
  - Teaching assistant
    - Zacharias El Banna: zeb@sics.se
  - Course leader
    - Seif Haridi: seif@imit.kth.se

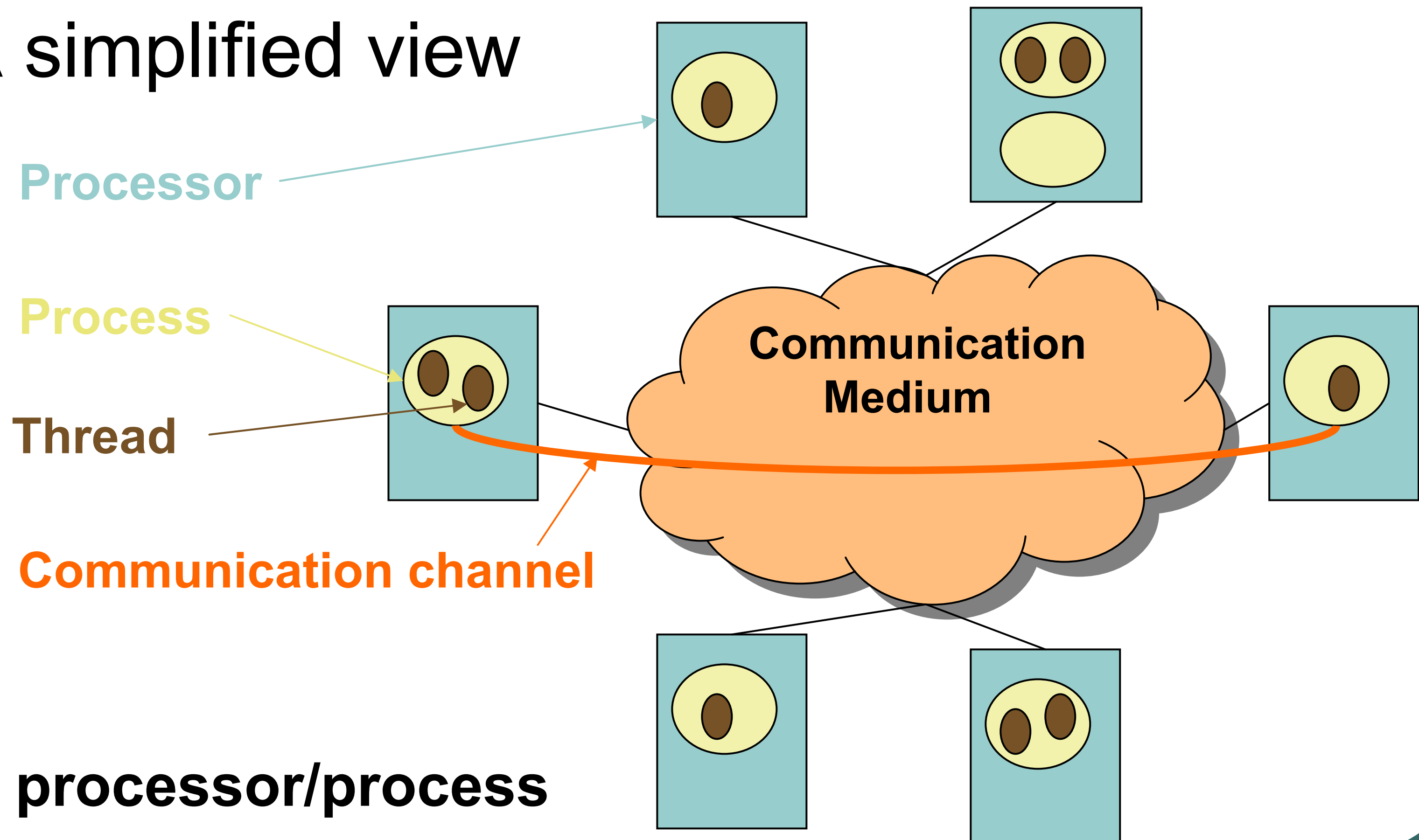
General information

- Reading of papers
  - In groups of two or three
  - Each group will read one or two research papers
- For each paper studied
  - Identify the problem
  - Explain the solution(s) presented in the paper
  - Identify positive and negative aspects of the paper
  - Propose your own solution if any
  - Provide a report
  - Give a presentation to the class

What is a distributed system

Distributed system

- A simplified view



Node: processor/process

Distributed system

- Set of computing nodes that cooperate in order to achieve a well defined goal
- Nodes cooperate through communication
- Communication is by message passing

Why distributed systems?

- Information exchange (collaborative work)
- Resources sharing (e.g. printer, backup storage, disk units, etc.)
- Cost reduction
- Increase of availability (partial-failure)
- Increase of performance through parallelism,...

Main characteristics

- No shared memory between nodes
  - Each node has its memory
  - Communication by message passing
- No global clock
  - Each node has its own clock
- Impossible for a node to obtain an instantaneous global state of the system

Focus of the course

- How to design distributed algorithms
  - Study of some fundamental problems
  - Analysis of distributed algorithms
- How to achieve fault-tolerance in a distributed system
  - Fault-tolerance: ability for a system to provide useful service despite the failure of some of its components
  - Very important for high availability

Why studying distributed algorithms?

- Distributed algorithms are backbone of distributed computing systems
- They are essential for the implementation of distributed systems
  - Distributed operating systems
  - Distributed databases, communication systems,
  - Real-time process-control systems,
  - Transportation, etc.

Classes of distributed algorithms

- **Fully decentralized**
  - Fault-tolerant
  - More difficult in general

- **With a centralized coordinator**
  - Conceptually simpler
  - Single point of failure, bottleneck
  - Require efficient mechanisms for selecting a new coordinator if the current one fails

References

- Text book:
  - *Distributed Operating Systems & Algorithms*
    - Randy Chow and Theodore Johnson, Addison Wesley, 1997
- Others
  - Distributed Algorithms
    - Nancy A. Lynch, 1996
  - Research papers

Course outline

- ***Introductory chapter (chapter 9)***
  - Causality
    - Ordering of events, Logical Clocks (timestamps)
    - Causal communication
  - Distributed snapshots
    - Detecting stable properties, Diffusing computation
  - Modeling a distributed computation
    - Expressing correctness properties of a dist. algo.
  - Failures in a distributed system

Course outline

- ***Chapter 10***
  - Synchronization
    - Distributed mutual exclusion: needed to regulate accesses to a common resource that can be used only by one process at a time
  - Election
    - Used for instance, to designate a new coordinator when the current coordinator fails

Course outline

- ***Chapter 11***
  - Distributed agreement
    - How to get a set of nodes to agree on a value
  - Distributed agreement is used for instance,
    - To determine which nodes are alive in the system
    - To confine malicious behavior of some components
    - (Fault-tolerance again!)

Course outline

- ***Chapter 12:***
  - Replicated data management
    - A key for high availability is to replicate components (data/files, servers, etc.)
  - We shall be concerned with
    - Techniques for maintaining replicated data in a distributed system, (database techniques)
    - Atomic broadcast/multicast
    - Membership

Course outline

- ***Chapter 13***
  - Check-pointing and recovery
    - Error recovery is essential for fault-tolerance
    - When a processor fails and then is repaired, it will need to recover its state of the computation
    - To enable recovery, check-pointing (recording of the state into a stable storage) is needed
    - We will be concerned with techniques used for this, in the context of distributed systems

Models of distributed Computation

Luc Onana Alima
KTH/IMIT/LECS/DCS

Outline

- Definitions
- Notations and assumptions
- Causality: timestamps, causal communication
- Distributed Snapshots
- Modeling a Distributed Computation
- Execution DAG Predicates
- Failures in Distributed System

Definitions

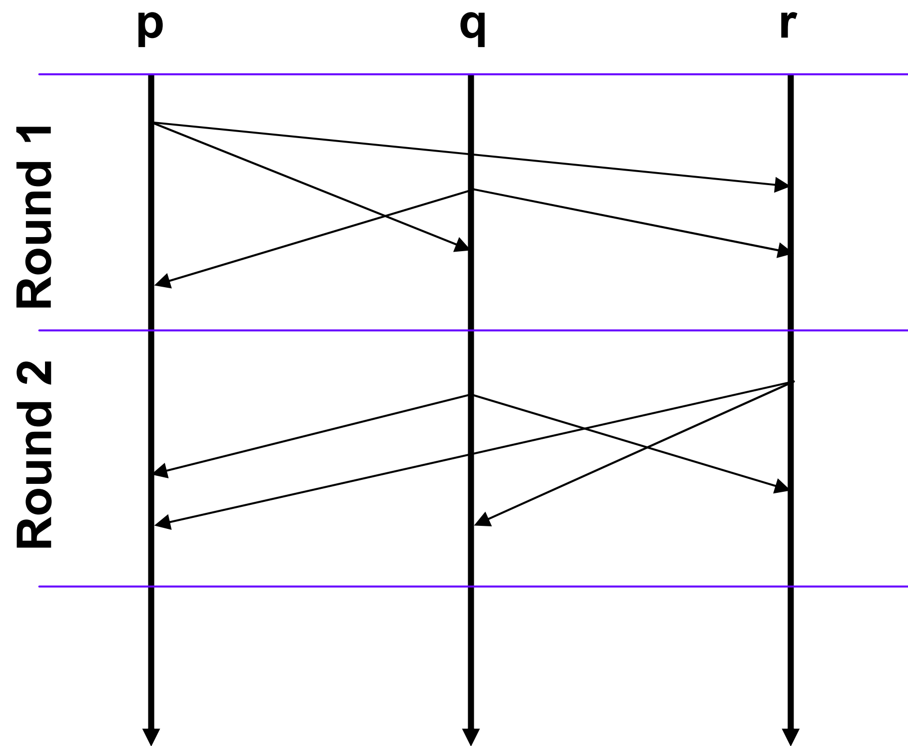
- Distributed system
  - A set of autonomous computing nodes that cooperate to achieve a well defined goal
  - Appears to its users as a single computing system
  - Each node
    - Performs local computation steps
    - Cooperates with others through message passing

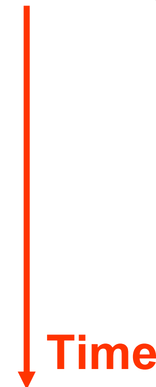
Definitions

- Synchronous Distributed System
  - We know an upper bound on the time needed to perform a local step at each node
  - We know an upper bound on the time required for a message to move from the sender to a receiver
  - One can assume that the system evolves in lock-steps (**rounds of computation**)
  - Timeouts can be used for failure detection

Definitions

- Synchronous Distributed System





Round:
1-send messages out
2-receive messages sent to you in this round
3-change your state

Definitions

- Asynchronous distributed System
  - We **don't** know an upper bound on the time needed to perform a local step at each node
  - We **don't** know an upper bound on the time required for a message to move from the sender to a receiver. **But** this time is finite
- There are also partially synchronous distributed systems (see N. Lynch 96)

Notations and assumptions

- A_1: No shared variables among processors
- A_2: One or more executing threads at a node
- A_3: Communication is by message passing
  - send(dest,action,param)
  - Sending a message is non-blocking

Notations and assumptions

- A4: Event-driven algorithms
  - Reaction upon receipt of a declared event
    (nodes execute events)
  - Events
    - Sending/receiving a message
    - An event is buffered until it is handled
    - Dedicated thread to handle some events at any time

Notations and assumptions

- Declaring events

```
wait for A1,A2,…,An
  on Ai (source;param) do
      code to handle Ai , 1<= i <=n
  end
```

Notations and assumptions

- Waiting for an event from *p* up to T seconds

```
wait until  p  sends (event;param), timeout=T
   on timeout do
        timeout action
   end
```

Notations and assumptions

- Waiting for an event from *p* up to T seconds, with no action on time out

```
wait until  p  sends (event;param), timeout=T
   on timeout do
        ;
   end
  if no timeout do
        Successful response actions
```

Notations and assumptions

- Definition
  - Let P be a protocol. If instance of P at processor q consists of threads T_1, T_2, T_3, …, T_n , we say that T_1, T_2, …, T_n are in the *same family*
- Threads in the same family share variables
  - Need for concurrency control within a node

Notations and assumptions

- Assumption used (A_5):
  - Once a thread gains control of the processor, it does not release control to a thread of the same family until it is blocked.

Causality

- No global clock in distributed systems
- Consequence
  - No processor within a distributed system can have a view of the current global system state
- Causality serves as a supporting property

Causality

- Provided traveling backward in time is excluded, distributed systems are causal
  - The cause precedes the effect.
- The sending of a message *precedes* the receipt of that message

Causality

- System composition
  - We assume a distributed system composed of the set processors/processes $P = \{p_1, \ldots, p_M\}$.
- Each processor reacts upon receipt of an event

Causality

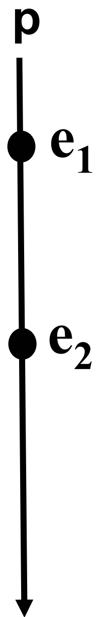
- Events include
  - Communication events:
    - sending a message; receiving a message
  - Internal events:
    - local input/output, raising of a signal
    - decision on a commit point (database); etc.
- **Convention**
  - E : the set of all possible events in the system
  - E_p : the set of all events in E that occur at processor p

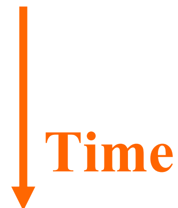
Causality

- Establishing order between events in a distributed system is important
  - Fair allocation of a non-divisible shared resource (distributed mutual exclusion)
  - Debugging/analysis of distributed computation
    - Was event $\mathbf{e_1}$ at processor **p** responsible for causing event $\mathbf{e_2}$ at processor **q**?

Causality

- Events are totally ordered on the same processor

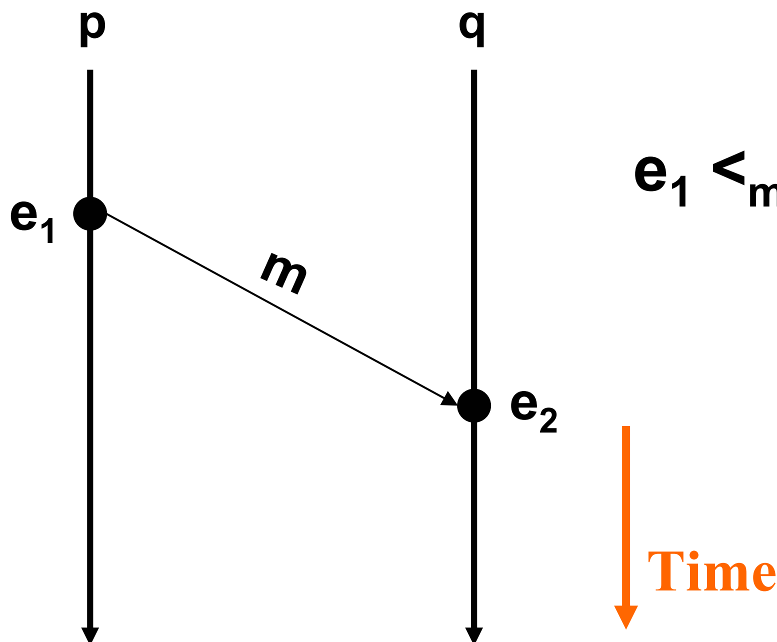




$e_1 <_p e_2 \equiv$ ``e_1 occurs before e_2 in p´´

Causality

- Event of sending a message **m** occurs before the event of receiving **m**



$e_1 <_m e_2 \equiv$ ``e_1 occurs before e_2´´

Causality

- Happens before relation
  - A relation, we denote $<_H$, defines on E such that
    - If $e_1 <_p e_2$ then $e_1 <_H e_2$
    - If $e_1 <_m e_2$ then $e_1 <_H e_2$
    - If $e_1 <_H e_2$ and $e_2 <_H e_3$ then $e_1 <_H e_3$
- Note
  - $e_1 <_H e_2$ means
    - "e_1 ***causally affects*** e_2" or
    - "e_1 ***causally precedes*** e_2" or
    - "e_2 ***causally follows*** e_1"

Causality

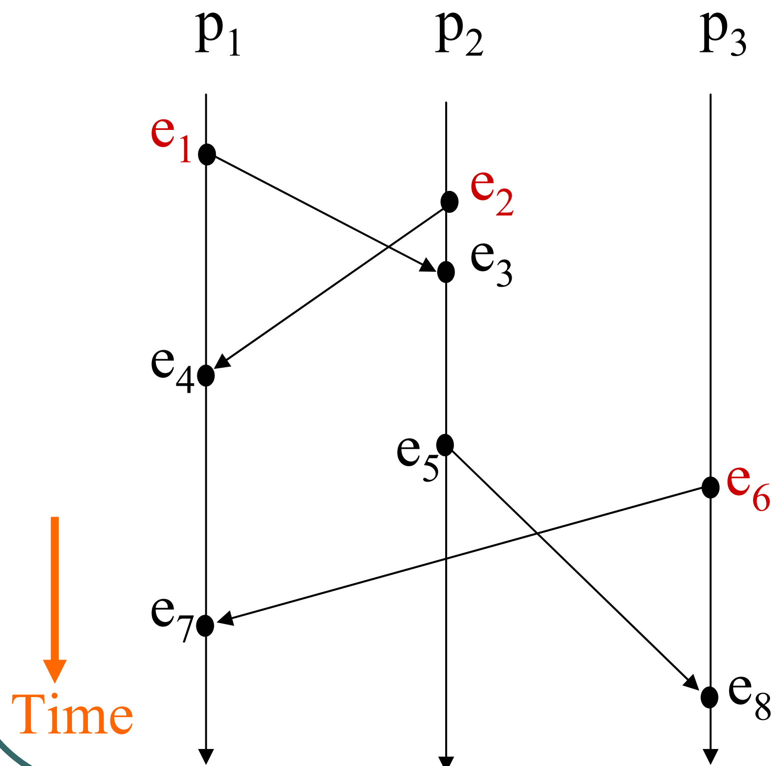
- **Causal path from event e to event e'**
  - Sequence of events $\mathbf{e_1, e_2, \ldots, e_n}$ such that
  - $\mathbf{e = e_1}$ and $\mathbf{e' = e_n}$
  - For each *i* in {1,..,n-1}, $\mathbf{e_i <_H e_{i+1}}$
- Note
  - $\mathbf{e <_H e'}$ iff there is a causal path from e to e'

Causality

- Concurrent events
  - Two distinct events $\mathbf{e_1}$ and $\mathbf{e_2}$ are said to be concurrent if neither $\mathbf{e_1 <_H e_2}$ nor $\mathbf{e_2 <_H e_1}$
- **Relation $<_H$ defines** a partial order on E
  - Because there could be concurrent events in **E**

Causality

- Happens before and concurrent events illustrated



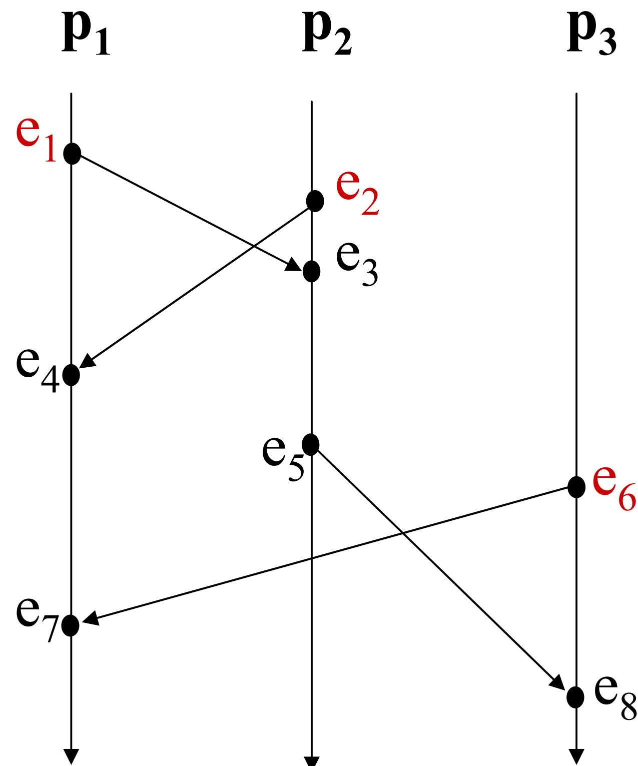
No causal path neither from e_1 to e_2 nor from e_2 to e_1
e_1 and e_2 are concurrent

No causal path neither from e_1 to e_6 nor from e_6 to e_1
e_1 and e_6 are concurrent

No causal path neither from e_2 to e_6 nor from e_6 to e_2
e_2 and e_6 are concurrent

Causality

- Exercise



Using the Happens before relation, compare:

- **e_1 and e_7**
- **e_1 and e_8**
- **e_5 and e_2**
- **e_4 and e_6**

Time

Logical clocks

- Physical clocks, not suitable/needed for many distributed applications
- Logical clocks an alternative
  - Assign numbers to events
  - Impose total ordering of the observed events
- The logical clock for processor p
  - A function $\mathbf{C_p}$ that assigns a number $\mathbf{C_p}(\mathbf{e})$ to each event **e** that occurs at **p**

Logical clocks

- The entire system of logical clocks
  - A function **C** that assigns a number **C(e)** to each event **e** of **E**, with $\mathbf{C(e)=C_p(e)}$ if **e** occurs at **p**

- Clock condition (consistency with $\mathbf{<_H}$)
  - For any pair of events $\mathbf{e_1}$ and $\mathbf{e_2}$
    if $\mathbf{e_1 <_H e_2}$ then $\mathbf{C(e_1) < C(e_2)}$

Satisfying the clock condition

- To satisfy the clock condition, it is sufficient to ensure the following two conditions
  - If $\mathbf{e_1}$ and $\mathbf{e_2}$ are events in **p** and $\mathbf{e_1 <_p e_2}$ then
    $\mathbf{C_p(e_1) < C_p(e_2)}$
  - If $\mathbf{e_1}$ is the sending of message **m** by **p** and $\mathbf{e_2}$ is the receipt of message **m** by **q** then
    $\mathbf{C_p(e_1) < C_q(e_2)}$

Implementing the logical clocks

- Lamport's logical clocks (timestamps)
  - Gives a global logical clock consistent with $<_H$
- Principle
  - Each processor has a local logical clock: **my_TS**
  - Each event **e** has a timestamp **e.TS**
  - Each message **m** carries the timestamp **m.TS** of the sending event

Implementing the logical clocks: Lamport's algorithm (pseudo-code)

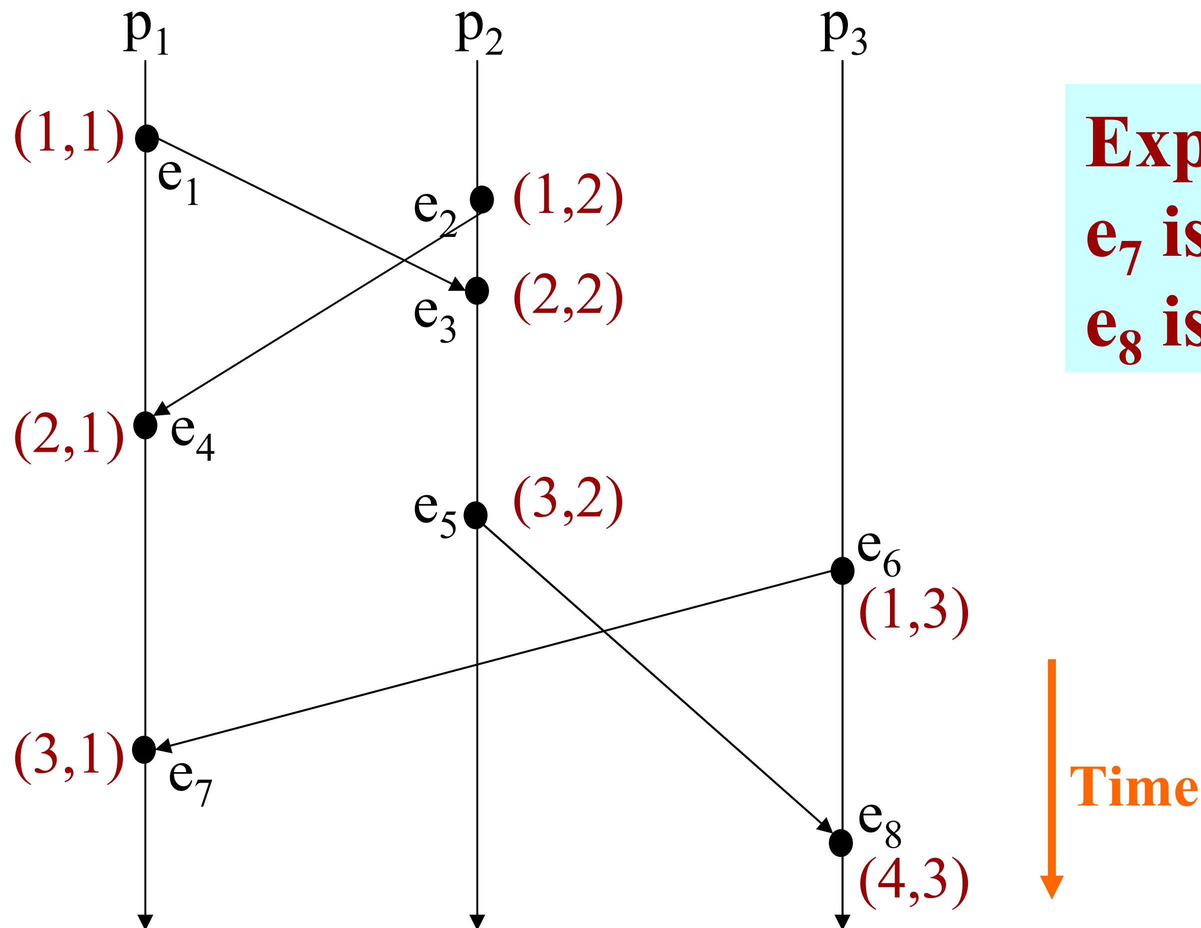
```
Initially,
      my_TS = 0

on event e do
      if e is the receipt of message m then
          my_TS := max(m.TS,my_TS)+1;
          e.TS := my_TS
      elseif e is an internal event then
          my_TS := my_TS+1 ; e.TS := my_TS
      elseif e is the sending of message m then
          my_TS := my_TS+1 ; e.TS := my_TS;
          m.TS = my_TS
 end
```

Total ordering of events

- The algorithm in the previous slide places a partial order over the events
- To place a total ordering **ε**, we use a total ordering **<** over processor identifiers to break ties
- Hence if $\mathbf{e_1}$ occurs at $\mathbf{p_1}$ and $\mathbf{e_2}$ at $\mathbf{p_2}$ then
  - $\mathbf{e_1 \; \varepsilon \; e_2}$ iff $\mathbf{e_1.TS < e_2.TS}$ or
    $\mathbf{e_1.TS = e_2.TS}$ and $\mathbf{p_1 < p_2}$

Lamport's timestamps illustrated



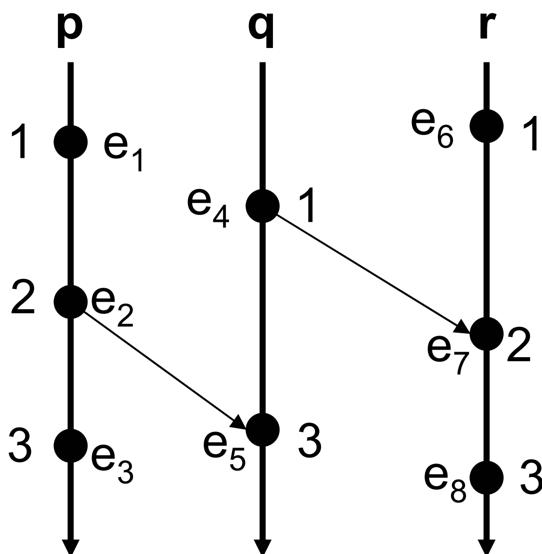
Explain why
e_7 is labeled (3,1)?
e_8 is labeled (4,3)?

Lamport's algorithm: properties

- Fully decentralized
- Simple
- Fault-tolerant
- Minimum overhead
- Many applications

Limitation of Lamport's algorithm

- If $\mathbf{e_1} <_H \mathbf{e_2}$ then $\mathbf{e_1.TS} < \mathbf{e_2.TS}$
  But it is not necessarily the case that
  if $\mathbf{e_1.TS} < \mathbf{e2.TS}$ then $\mathbf{e_1} <_H \mathbf{e_2}$



$e_1.TS < e_8.TS$ but $e_1 \not<_H e_8$

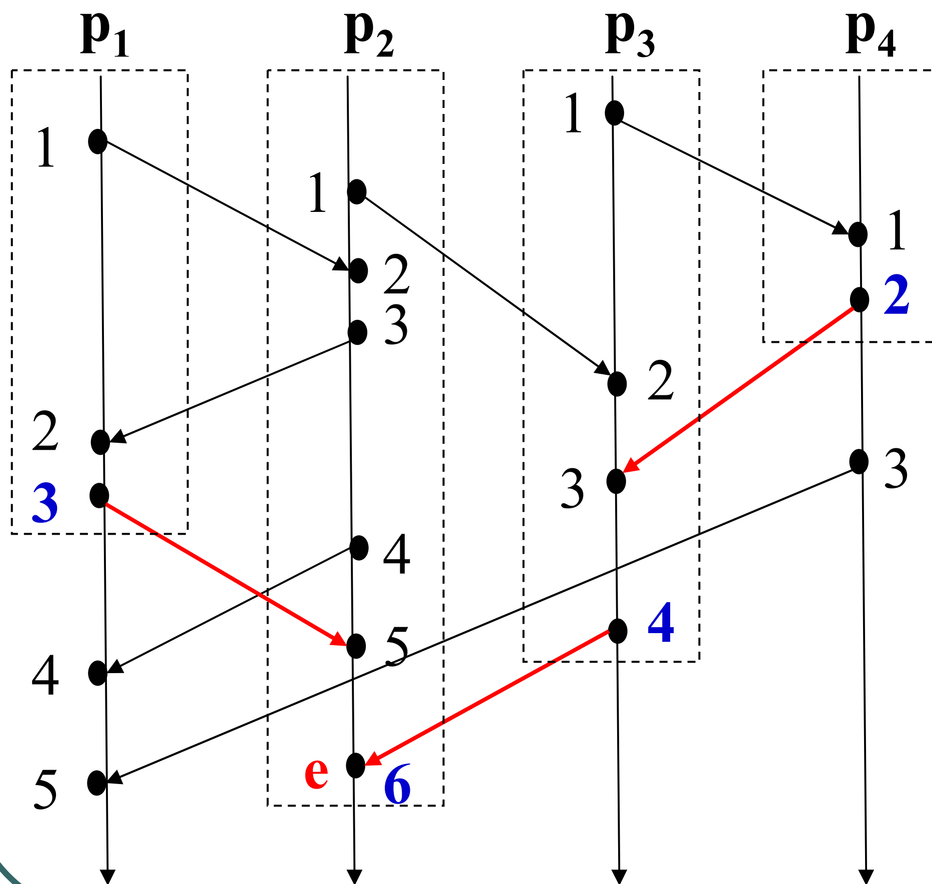
Given two events, we cannot say whether they are causally related from their timestamps

Vector timestamps

- Permit to capture causality precisely

- Principle
  - Each processor has a vector logical clock: **my_VT[1..M]**
  - Each event **e** has a vector timestamp: **e.VT**
    - If **e.VT[p]=k**, then **e** causally follows the first **k** events that occurred at processor **p**
  - Each message **m** carries the vector timestamp **m.VT** of the sending event

Meaning of e.VT[p] illustrated



The vector timestamp attached to event **e** indicates the logical time of the last event at:

- $\mathbf{p_1}$ that causally precedes **e**
- $\mathbf{p_2}$ that causally precedes **e**
- $\mathbf{p_3}$ that causally precedes **e**
- $\mathbf{p_4}$ that causally precedes **e**

Hence

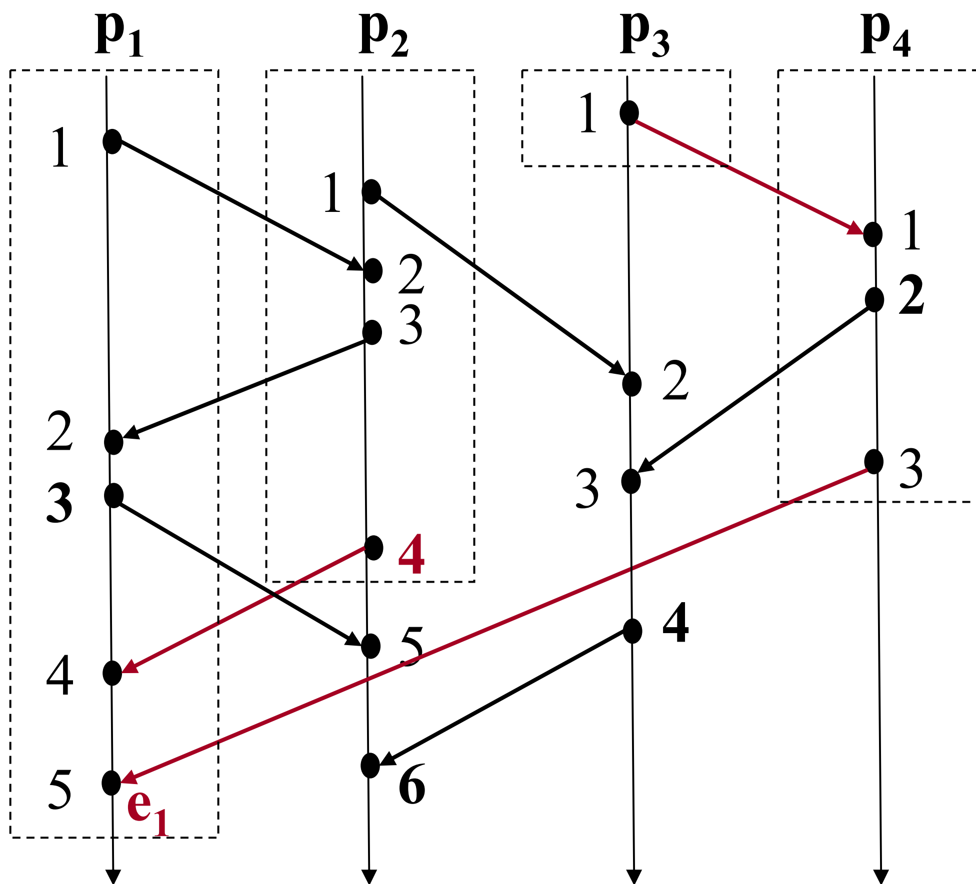
e.VT[p_1]=3
e.VT[p_2]=6
e.VT[p_3]=4
e.VT[p_4]=2

Note:
e causally precedes e

Implementing vector timestamps (pseudo-code)

```
Initially,
   my_VT = [0,…,0]
on e do
    if e is the receipt of message m then
      for i := 1 to M do
        my_VT[i] := max(m.VT[i],my_VT[i]);
        my_VT[self] := my_VT[self] +1
        e.VT := my_VT
     end
    elseif e is an internal event then
       my_VT[self] := my_VT[self]+1 ;
       e.VT := my_VT
    elseif e is the sending of message m then
       my_VT[self] := my_VT[self]+1 ;
       e.VT := my_VT
       m.VT = my_VT
end
```

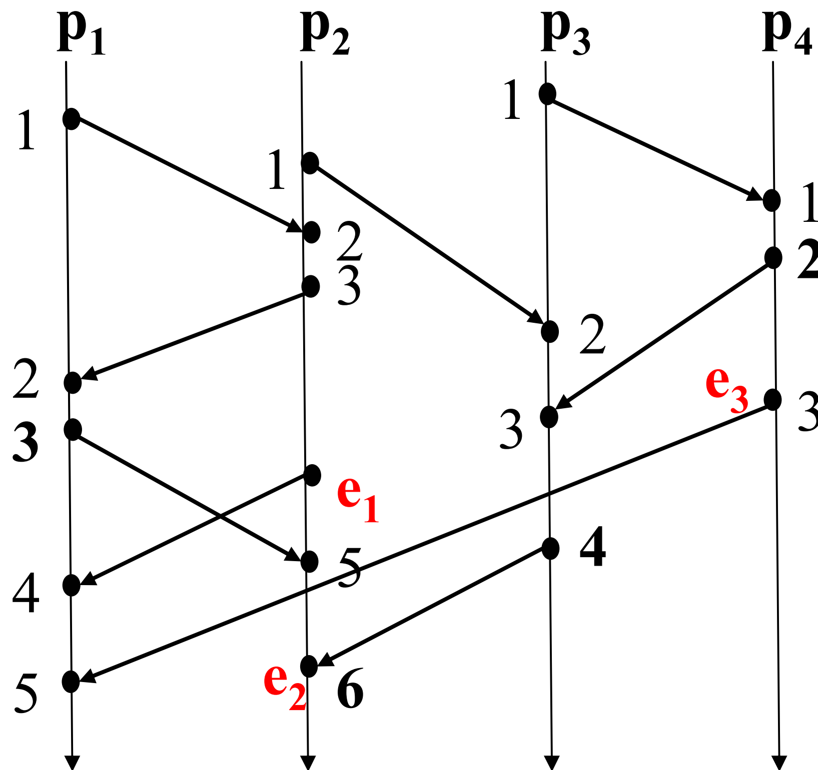
Vector timestamps illustrated



e_1.VT=(5,4,1,3)

Vector timestamps illustrated

Exercise



Give the vector timestamps for the events e_1, e_2 and e_3

Comparing vector timestamps

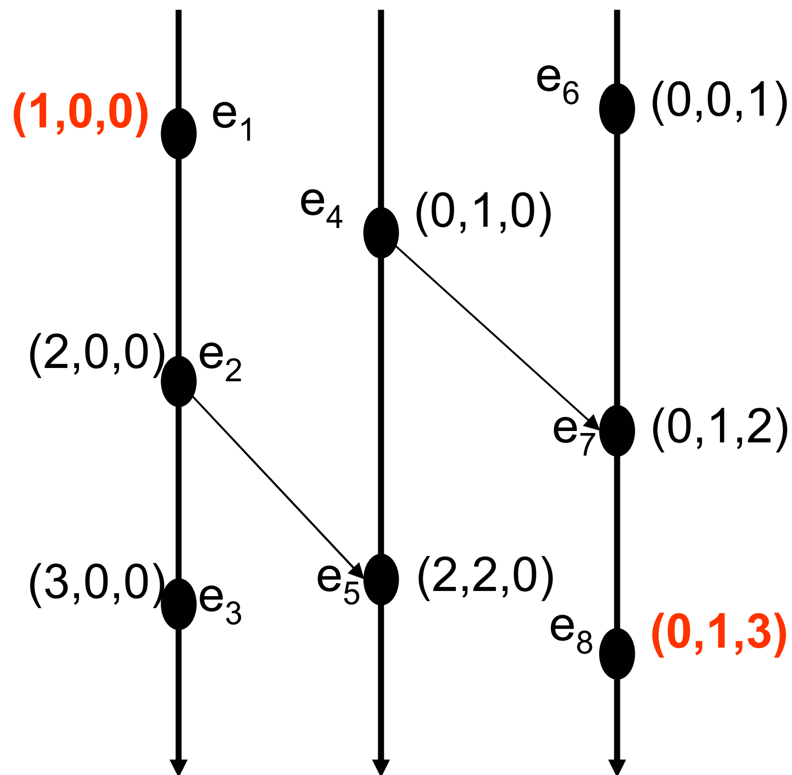
- Let **VT1** and **VT2** be two vector timestamps
- $\mathbf{VT_1 \neq VT_2}$ iff there is a **p** such that $\mathbf{VT_1[p] \neq VT_2[p]}$
- $\mathbf{VT_1 \leq VT_2}$ iff for all **p** $\mathbf{VT_1[p] \leq VT_2[p]}$
- $\mathbf{VT_1 <_V VT_2}$ iff $\mathbf{VT_1 \leq VT_2}$ and $\mathbf{VT_1 \neq VT_2}$

Causally related events

- Let $\mathbf{e_1}$ and $\mathbf{e_2}$ be two arbitrary events with vector timestamps $\mathbf{e_1.VT}$ and $\mathbf{e_2.VT}$

- $\mathbf{e_1}$ and $\mathbf{e_2}$ are causally related if $\mathbf{e_1.VT <_V e_2.VT}$ or $\mathbf{e_2.VT <_V e_1.VT}$. Otherwise, $\mathbf{e_1}$ and $\mathbf{e_2}$ are concurrent

An application of vector timestamps



Now, given the vector timestamps of e_1 and e_8, one can determine that they are concurrent, since

$(1,0,0) \nless_V (0,1,3)$

and

$(0,1,3) \nless_V (1,0,0)$