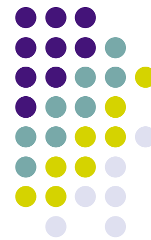
LINF 2345
Clocks and Distributed Computations

Seif Haridi
Peter Van Roy



Clocks and Distributed Computations

- Physical clocks
  - Clock synchronization
- Models of distributed computation
  - Causality
- Logical clocks
  - Lamport clock
  - Vector clock

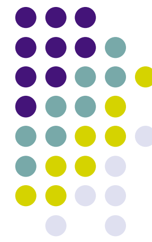
Centralized Time and Distributed Time

- In a centralized system, time is unambiguous
  - If process A asks for the time, and a little later process B asks for the time, the second value will be greater than or equal to the first
  - Just think about what would happen to the "make" utility if this were not the case
    - For example, output.c is modified but has an "earlier" time than output.o, so it will not be recompiled! Chaos!
- In a distributed system, achieving agreement on time is not easy
  - Let us look at some of the approaches

Physical Time and Logical Time

- Physical time: Sometimes we need the exact time, not just an ordering of events
  - Universal Coordinated Time (UTC) is the worldwide exact time
  - Clock synchronization can be used to get systems to agree on a time
- Logical time: Sometimes it is enough to order events, without knowing the exact time
  - Ordering is achieved by logical clocks in distributed systems
  - We will see two kinds of logical clock, the Lamport clock and the vector clock

Physical Clocks



Universal Coordinated Time UTC

- Based on the number of transitions per second of the cesium 133 atom (pretty accurate)
  - UTC is derived from International Atomic Time (TAI) with adjustments
- At present, the real time is taken as the average of some 50 cesium clocks around the world
- Introduces a leap second from time to time to compensate that days are getting longer (rotation of the Earth is slowing down due to tidal action)

Universal Coordinated Time UTC

- UTC is **broadcast** through short wave radio and satellite
- Satellites can give an accuracy of about ±0.5 ms
- **Question:** Does this solve all our problems? Don't we now have some global timing mechanism?

Physical Clocks: UTC

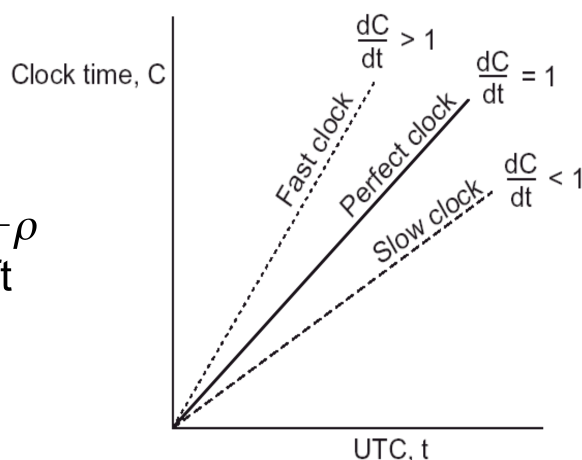
- **Problem:** Suppose we have a distributed system with a UTC receiver somewhere in it ⇒ we still have to distribute its time to each machine
- Machines do not always have radio receivers or the receivers might not receive any signal
- And anyway, the signal is only accurate to 0.5 ms, which is still a long time for a computer

Time Distribution and Synchronization

- Every machine has a timer that generates an interrupt H times per second
- There is a clock in machine p that **ticks** on each timer interrupt
- Denote the value of that clock by $C_p(t)$, where t is UTC time
- Ideally, we have that for each machine p, $C_p(t) = t$, or, in other words, $dC_p/dt = 1$

Time Distribution and Synchronization

In practice
$1\text{-}\rho \le dC/dt \le 1+\rho$
ρ is a clock drift



Time Distribution and Synchronization

- **Goal:** Never let two clocks in any system differ by more than δ time units
- **Solution**: Synchronize at least every $\delta/(2\rho)$ seconds

S. Haridi and P. Van Roy, LINF2345 11

Example

- Assume ρ = 0.01
  - Each second (1000ms) we have a drift of 10ms
- Let's calculate $\delta/2\rho$
  - We want δ=1ms
  - 2ρ=0.02
  - Therefore, we have to synchronize each 50 ms

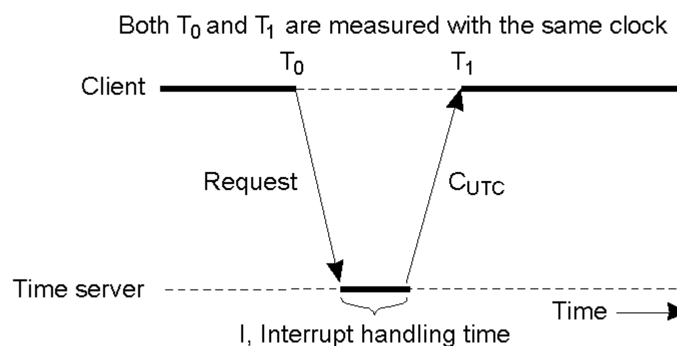
S. Haridi and P. Van Roy, LINF2345 12

Clock Synchronization Methods
Method I

- Every machine asks a **time server** for the accurate time at least once every $\delta/(2\rho)$ seconds
- Okay, but you need an accurate measure of round trip delay, including interrupt handling and processing incoming messages

Cristian's Algorithm

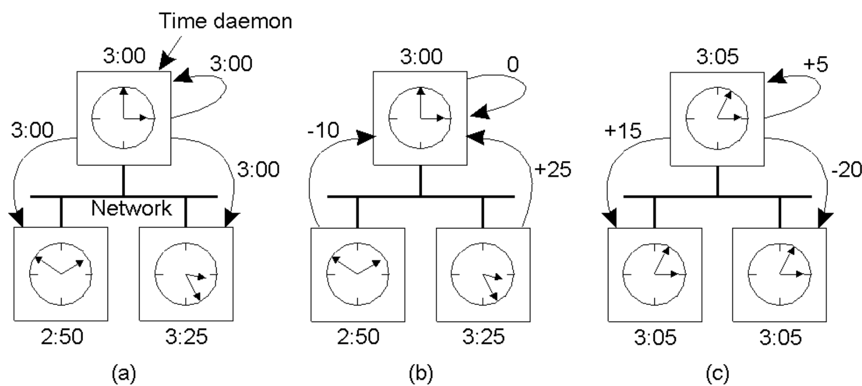
- Getting the current time from a time server.



Clock Synchronization Methods
Method II

- Let the time server scan all machines periodically, calculate an average, and inform each machine how it should adjust its time relative to its present time
- In this method, you don't even need to propagate UTC time (why not?)

The Berkeley Algorithm



a) The time daemon asks all the other machines for their clock values
b) The machines answer
c) The time daemon tells everyone how to adjust their clock

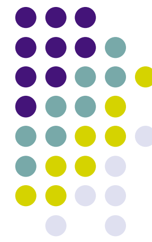
Clock Synchronization Methods
Methods I & II

- For both methods, there is a fundamental problem
  - The time is not allowed to run backward
  - How do we do it? We never adjust the time itself, but we adjust the rate at which the time is updated (speeding up or slowing down the clock)

Network Time Protocol (NTP)

- Protocol for synchronizing clocks over packet-switched variable-latency networks
  - Variable latency (jitter)!
  - One of the oldest Internet protocols still in use (≤1985)
- Uses Marzullo's algorithm for producing an optimal estimate from a set of estimates with confidence intervals
- NTPv4 can maintain time to within 10 ms over Internet, 200μs over LANs under ideal conditions

Models of Distributed Computation



Why?

- If we want to understand distributed systems, we have to understand in a more fundamental way how they execute
- This will be the basis of a number of distributed algorithms that we will present

Outline

- Definitions
- Models of distributed computation
- Notations and assumptions
- Causality: timestamps, causal communication
- Distributed snapshots

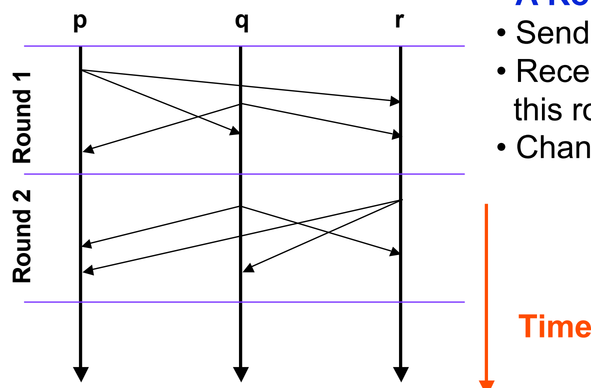
Definitions
Distributed System

- 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

Synchronous Distributed System

- The first kind of distributed system does synchronous execution
- Computations proceed in rounds (lock steps)
- 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
- Timeouts can be used for failure detection
  - This is the main strength of the synchronous model: definite information is available after the timeout
  - Unfortunately, we usually won't be able to use this model

Synchronous Distributed System



- **A Round**
- Send messages out
- Receive messages sent in this round
- Change your state

Time

Asynchronous Distributed System

- A more common kind of distributed system does asynchronous execution
- 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 we **do** know that this time is finite.
- There are also partially synchronous distributed systems (see the book of Nancy Lynch 1996)

Notations and Assumptions

- A_1: No shared variables among processes
- A_2: One or more executing threads in a process
- A_3: Communication is by message passing
  - Send Action(Destination; Params)
  - Sending a message is non-blocking

Notations and Assumptions

- A_4: Algorithms are event-driven
  - Reaction upon receipt of an event (nodes execute events)
  - Events
    - Sending/receiving a message
    - Internal events
    - An event is buffered until it is handled
    - Dedicated thread to handle some events at any time

Notations and Assumptions

- Each process waits for possible events $A_1, \ldots, A_n$
- Similar to Dijkstra guarded commands

input
  A_1(*source;param*)$_1$ **then**
  Code to handle A_1
[] …
[] A_n(*source;param*)$_n$ **then**
  Code to handle A_n
end

Notations and Assumptions

- Each process waits for possible events $A_1, \ldots, A_n$

input
A_1(*source;param*)$_1$ && ⟨*Condition*⟩ **then**
Code to handle A_1
[] …
[] A_n(*source;param*)$_n$ && ⟨*Condition*⟩ **then**
Code to handle A_n
end

Notations and Assumptions

- Waiting for an event A_i from *P* up to T seconds

input
A_1(*P;param*) **then**
Code for A_1
[] …
after T **then**
Timeout action
end

Notations and Assumptions

- Waiting for events A_i from *P* up to T seconds, with no action on time out

input
  A_1*(P;param)* **then**
  Code to handle A_1
[] …
after T **then skip**
end

Notations and Assumptions

- **input**
  A_i(*source;param*)$_i$
- **condition**
  ⟨*Condition*⟩
- **action**
  Code to handle A_i

Causality Assumptions

- No physical clock
  - No external source of “true time”; we’ve seen how hard it is to get such a source (and we haven’t even seen yet how failures and security issues make it even harder)
- Consequence
  - No processor within a distributed system can see the current global system state
  - We can, however, reason about the global state (God’s view), even though we can never see it from within the system
- Causality serves as a supporting property
  - Lacking any other external support, we can still rely on causality

Causality

- Provided traveling backward in time is excluded, distributed systems are causal
  - The cause always precedes the effect
  - In Aristotle’s classification, this is the *efficient cause*
  - (Other causes: material cause, formal cause, final cause)
- The sending of a message *precedes* the receipt of that message
- A process’s future depends on its past

Causality
System Composition

- A distributed system is composed of a set of processes:
  $P = \{p_1, \ldots, p_M\}$
- Each process reacts upon receipt of an event

Causality
Events

- 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 events in the system
  - E_p: the set of all events in E that occur at process p

Causality
Why

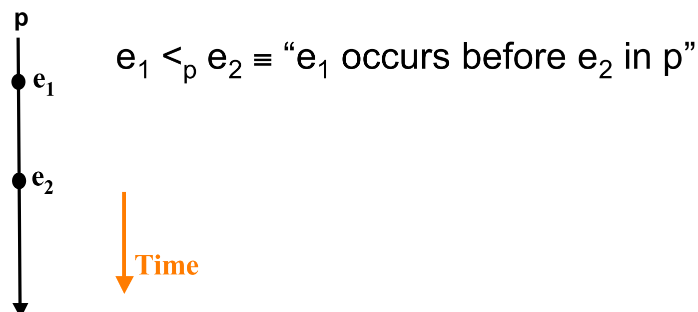
- Establishing order between events in a distributed system is important

- 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

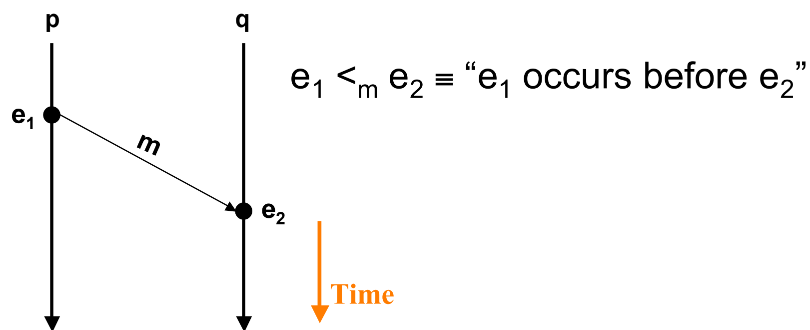
- Events are totally ordered on the same process



$e_1 <_p e_2 \equiv$ "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, denoted by $<_H$, defined on E:
  - If $\mathbf{e_1 <_p e_2}$ then $\mathbf{e_1 <_H e_2}$
  - If $\mathbf{e_1 <_m e_2}$ then $\mathbf{e_1 <_H e_2}$
  - If $\mathbf{e_1 <_H e_2}$ and $\mathbf{e_2 <_H e_3}$ then $\mathbf{e_1 <_H e_3}$

- Note $\mathbf{e_1 <_H e_2}$ means
  - "$\mathbf{e_1}$ ***causally affects*** $\mathbf{e_2}$" or
  - "$\mathbf{e_1}$ ***causally precedes*** $\mathbf{e_2}$" or
  - "$\mathbf{e_2}$ ***causally follows*** $\mathbf{e_1}$"

Causal Path

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

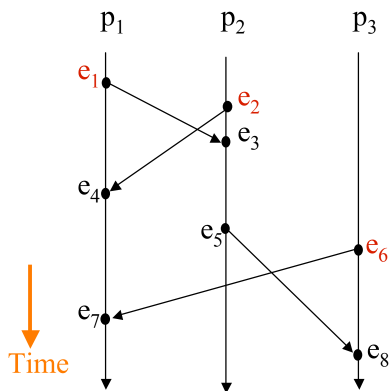
S. Haridi and P. Van Roy, LINF2345 41

Causality
Concurrent Events

- Two distinct events $\mathbf{e_1}$ and $\mathbf{e_2}$ are said to be concurrent if neither $\mathbf{e_1} <_H \mathbf{e_2}$ nor $\mathbf{e_2} <_H \mathbf{e_1}$
- The relation $<_H$ defines a partial order on E
  - A partial order is a transitive and anti-symmetric relation
  - It is not a total order; there can be concurrent events in **E**

S. Haridi and P. Van Roy, LINF2345 42

Causality Examples

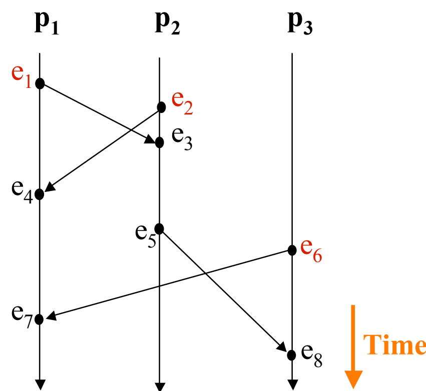


No causal path neither from e_1 to e_2 nor from e_2 to e_1
e_1 and e_2 are concurrent

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

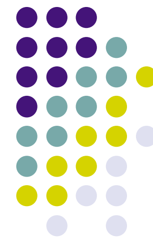
Causality Exercise



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

Logical Clocks



Logical Clocks

- Physical clocks are either not suitable or not needed for many distributed applications
- Logical clocks are an alternative
  - Assign numbers to events
  - Impose (possibly total) ordering of the observed events
- The logical clock for process p
  - A function $\mathbf{C_p}$ that assigns a number $\mathbf{C_p(e)}$ to each event 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 $C(e)=C_p(e)$ if **e** occurs at **p**
- Clock condition (consistency with $<_H$)
  - For any pair of events e_1 and e_2 if $e_1 <_H e_2$ then $C(e_1) < C(e_2)$

Satisfying the Clock Condition

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

Implementing Logical Clocks

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

Lamport's Algorithm

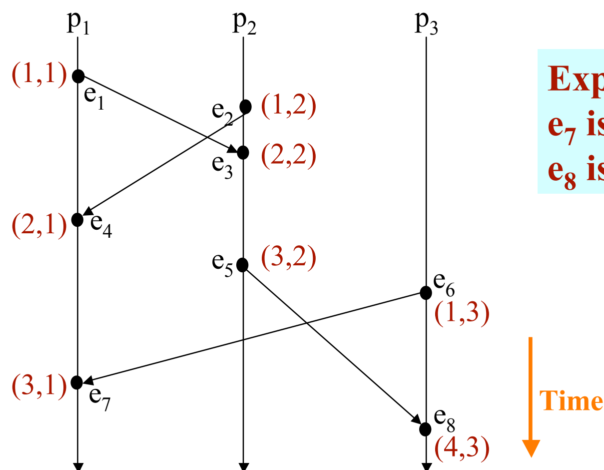
```
Initially,
   C := 0

input event e then
   if e is the receipt of message m then
        C := max(m.TS, C) + 1
   elseif e is an internal event then
        C := C+1
   else e is the sending of message m then
        C := C+1
        m.TS := C
   end
   e.TS := C
end
```

Total Ordering of Events

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

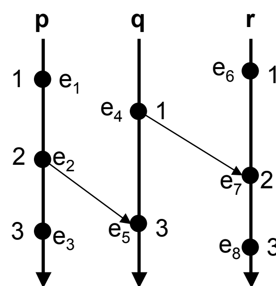
Lamport's Timestamps Example



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

Limitation of Lamport's Algorithm

- If $e_1 <_H e_2$ then $e_1.TS < e_2.TS$
  But it is not necessarily the case that
  if $e_1.TS < e_2.TS$ then $e_1 <_H e_2$



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

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

Lamport's Algorithm Properties

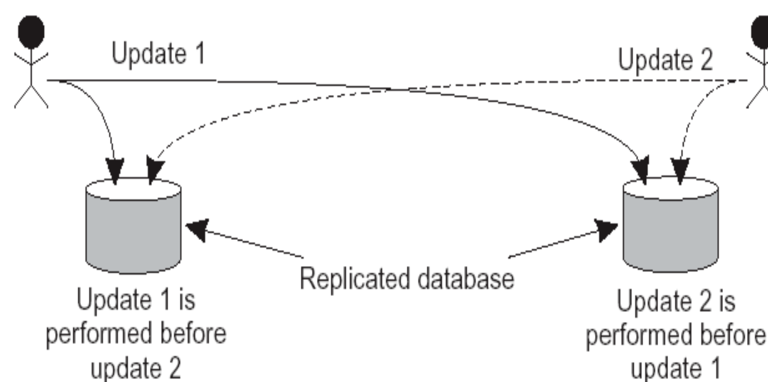
- Fully decentralized
- Simple
- Fault-tolerant (process failure, message loss)
- Minimum overhead
- Many applications (see next)

Example
Totally-Ordered Multicast

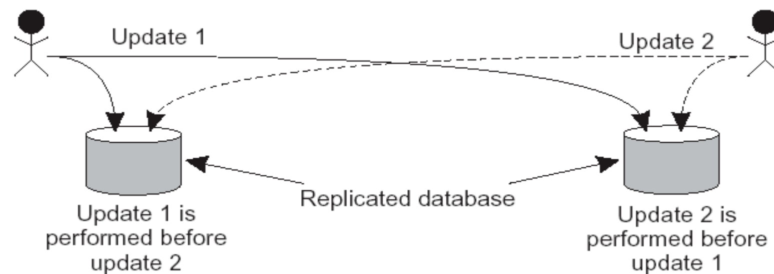
- We sometimes need to guarantee that concurrent updates on a replicated database are seen in the same order everywhere

- Process *P*1 adds $100 to an account (initial value: $1000)
- Process *P*2 increments account by 1%
- There are two replicas

Example
Totally-Ordered Multicast



Example
Totally-Ordered Multicast



Outcome: in absence of proper synchronization, replica #1 will end up with $1111, while replica #2 ends up with $1110

Totally-Ordered Multicast

- Assumptions
  - Communication is reliable and FIFO ordered
  - The number of processes are known
  - For each update message received an *ack* message is sent to all processes

Totally-Ordered Multicast

- Process P_i sends timestamped message msg_i to all others. The message itself is put in a local queue Q_i
- Any incoming message at P_j is queued in Q_j according to its timestamp
- P_j passes a message msg_i to its application if:
  - msg_i is at the head of Q_i
  - For each process P_k, there is a message ack_k in Q_j with a larger timestamp
  - All ack messages are removed from local queue

Extension to Multicasting Vector Timestamps

- Lamport timestamps do not guarantee that if $a.TS < b.TS$ that *a* indeed happened before *b*.
- Given two events, we cannot say whether they are causally related from their timestamps
- We need Vector Timestamps for this

Vector Timestamps

- Assume the number n of processes is known
- Each process P_i has an array $V_i[1..n]$, where $V_i[j]$ denotes the number of events that process P_i knows have taken place at process P_j.
- When P_i sends a message m, it adds 1 to $V_i[i]$, and sends V_i along with m as **vector timestamp** $m.vt$
- Result: upon arrival, each other process knows P_i's timestamp

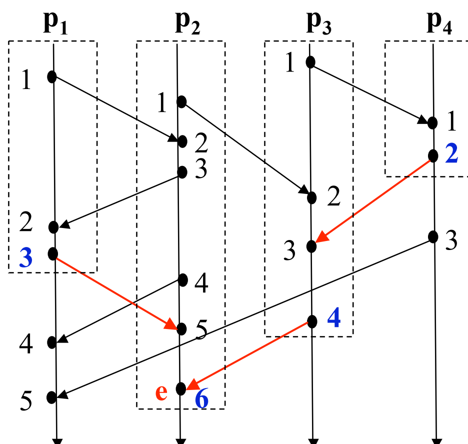
Vector Timestamps

- When a process P_j receives a message m from P_i with vector timestamp $m.vt$:
  - it updates each $V_j[k]$ to $\max\{V_j[k], m.vt[k]\}$,
  - and increments $V_j[j]$ by 1
- **Question:** What does $V_i[j] = k$ mean in terms of messages sent and received?
- **NOTE:** Book is wrong!

Vector Timestamps

- Permit to capture causality precisely
- Each event *e* carries (sometimes implicitly) a vector timestamp: *e.vt*
- If *e.vt*[*i*]=*k*, then *e* causally follows the first *k* events that occurred at processor P_i

Meaning of *e.vt*[*p*] Illustrated



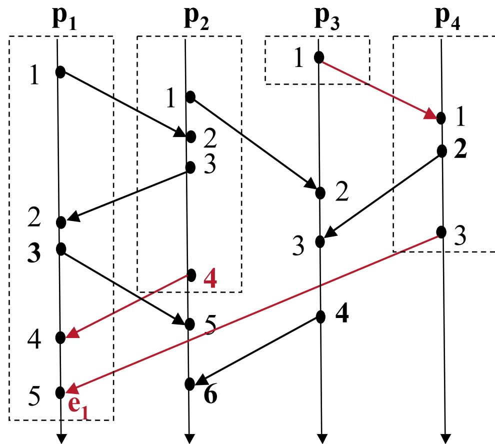
The vector timestamp V_2 after event **e** indicates the logical time of the last event at:

- P_1 that causally precedes **e**
- P_2 that causally precedes **e**
- P_3 that causally precedes **e**
- P_4 that causally precedes **e**

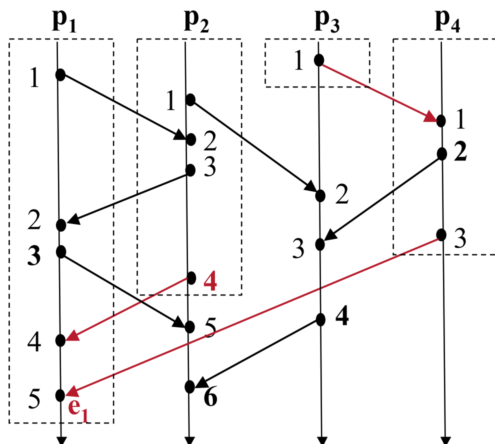
Hence

e.vt[1]=3
e.vt[2]=6
e.vt[3]=4
e.vt[4]=2

Vector Timestamps Illustrated



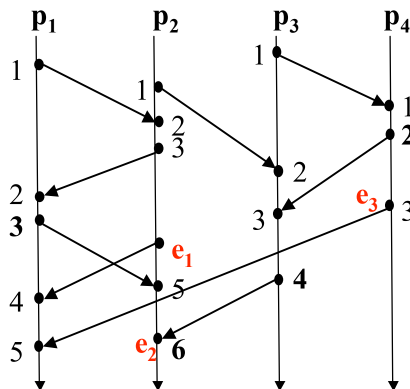
Vector Timestamps Illustrated



$e_1.vt=(5,4,1,3)$

Vector Timestamps Illustrated

Exercise



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

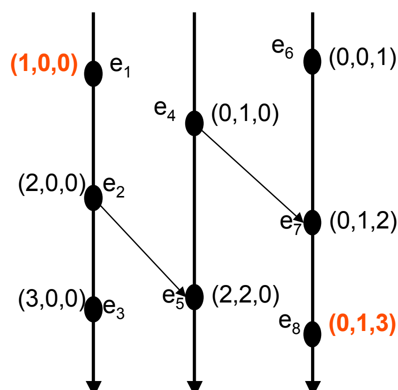
Comparing Vector Timestamps

- Let **vt1** and **vt2** be two vector timestamps
- $\mathbf{vt_1 \neq vt_2}$ iff there is an **i** such that $\mathbf{vt_1[i] \neq vt_2[i]}$
- $\mathbf{vt_1 \leq vt_2}$ iff for all **i**: $\mathbf{vt_1[i] \leq vt_2[i]}$
- **vt1** is strictly less than **vt2:**
  $\mathbf{vt_1 <_V vt_2}$ 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 iff $\mathbf{e_1.vt} <_V \mathbf{e_2.vt}$ or $\mathbf{e_2.vt} <_V \mathbf{e_1.vt}$
- Otherwise, $\mathbf{e_1}$ and $\mathbf{e_2}$ are concurrent

Causally Related Events



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

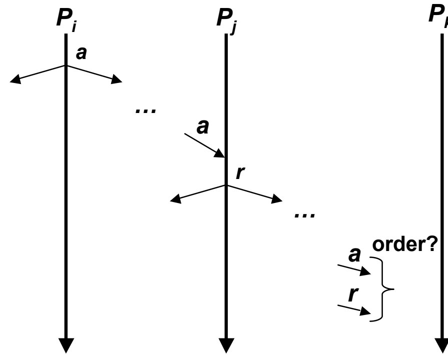
Vector Timestamps Multicast

- Consider a newsgroup, implemented by each user multicasting to all the others
  - The multicast should support causality, i.e., if two articles are independent their order should not matter and if one article is a reaction to another, it should arrive after
  - Totally-ordered multicasting is too strong since it forces all articles over the whole world to arrive in the same order, even if they have nothing to do with each other
- We cannot use Lamport timestamps since causality cannot be inferred from them
  - All are integers, which are ordered, even if there is no causal connection. So we would get a total order, which is too much.
- We can use vector timestamps!

Vector Timestamps Multicast

- Remember, for vector timestamps each process P_i maintains a vector V_i:
  - $V_i[i]$ is the number of events that have occurred so far at P_i
  - If $V_i[j]=k$ then P_i knows that k events have occurred at P_j
- With a slight adjustment, this can be used to guarantee causal message delivery
  - Assume that $V_i[i]$ is incremented only when P_i sends a message
- When a message r arrives at P_k, we postpone delivery of the message to the application until all causally preceding messages have been received

Vector Timestamps Multicast

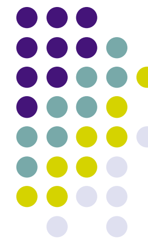


- P_i sends message *a*; P_j receives it and sends message *r* as reply
- P_k should receive *r* after *a*; how can we ensure this?

Vector Timestamps Ensuring Causal Arrival Order

- P_j sends a reply *r* to the message *a* sent by P_i
- How can we ensure that P_k receives *r* after *a*?
- By the following two conditions at P_k:
  - $r.vt[j] = V_k[j]+1$
    - *r* is the next message that P_k was expecting from P_j
  - $r.vt[i] \leq V_k[i]$ for all $i \neq j$
    - It's not allowed that: P_k sees messages not seen by P_j when it sent *r*
    - In other words, P_k's knowledge of P_j goes up to *r* but no further
- Things become subtle!
  - Another way to see it:
    - Wait to deliver *r* until *a* has surely been delivered, that is,
    - All other processes have seen all the messages seen by P_j at the moment that *r* was sent

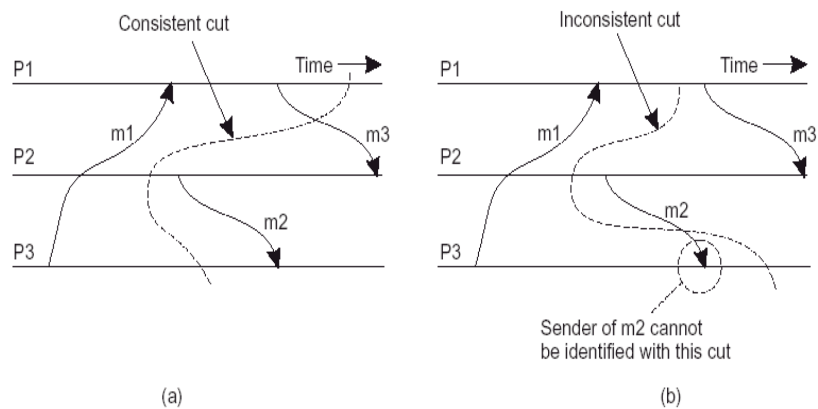
Distributed Snapshots



Global State

- Sometimes you want to collect the current state of a distributed computation, called a **distributed snapshot**
- It consists of all local states and messages in transit
- **Important:** A distributed snapshot should reflect a possible **consistent** state
  - The state need not have really existed, it just has to be possible

Distributed Snapshots



Distributed Snapshots

- Global states are consistent with causality
- Such global states can be used for detecting stable properties, e.g.
  - Distributed deadlock detection
  - Distributed garbage collection
  - Distributed termination detection
- Main difficulty
  - System state changes during observation

Distributed Snapshots
Assumptions

- The system is **connected**, that is there is a path between every pair of processes
- The number of processes is not necessarily known
- $C_{i,j}$ channel from p_i to p_j
- Channels are reliable and FIFO
  - Messages sent are eventually received in order

Distributed Snapshots
Process and Channel State

- Process state
  - State of a process (at any instant) is the assignment of values to the local variables of that process
  - Up to the application to define relevant state
- Channel state
  - State of $C_{i,j}$ is the ordered list of messages sent by p_i but not yet received at p_j

Distributed Snapshots
Global State

- Global state of the system is a pair $\langle S,L \rangle$ where
  - $S = \{s_1,.., s_M\}$ denotes the process states
  - L = channel states
- Note
  - A global state cannot be taken instantaneously
  - It must be computed in a distributed manner

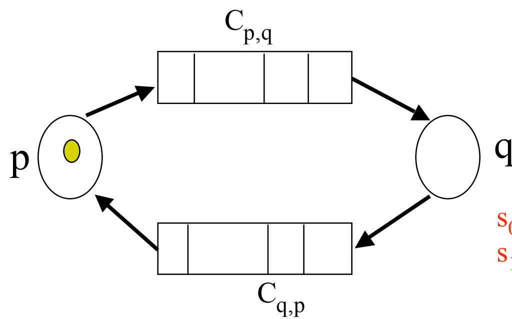
S. Haridi and P. Van Roy, LINF2345 81

Distributed snapshots

- The problem
  - Devise a distributed algorithm that computes a *consistent global state*
- What do we mean by consistent global state?

S. Haridi and P. Van Roy, LINF2345 82

Distributed Snapshot
Meaning of Consistent Global state

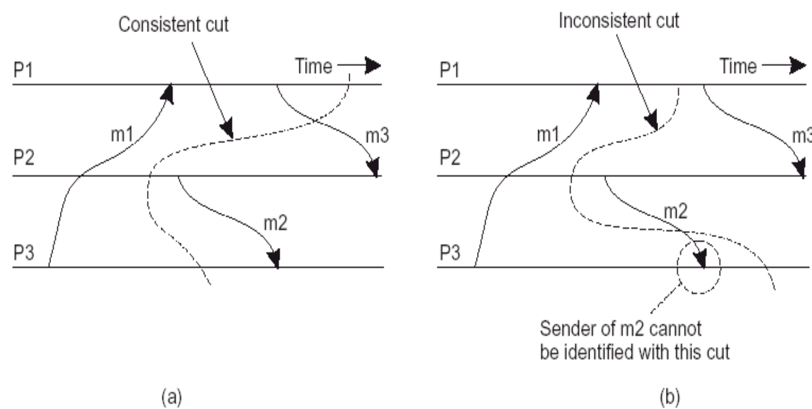


Two possible states for each processor: s_0, s_1

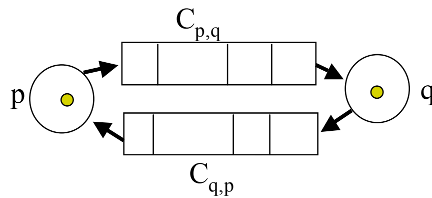
s_0: the processor hasn't the token
s_1: the processor has the token

The system contains exactly one token which moves back and forth between p and q. Initially, p has the token
Events: sending/receiving the token

Distributed Snapshots



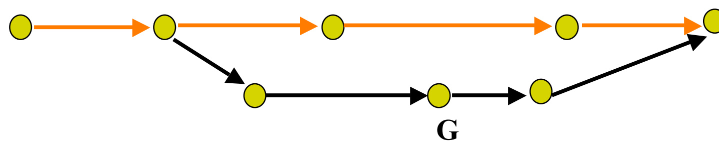
Meaning of consistent global state (inconsistent state)



Meaning of consistent global state

A global state G is consistent if it is one that **could have occurred**

Consider a system with two possible runs (non-determinism!)



Actual transitions

The output of the snapshot algorithm can be G !

Distributed Snapshots

- $S=\{s_1,..,s_M\}$ (The state of processes)
- o_i: event of observing s_i at p_i
- $O(S)=\{o_1,..,o_M\}$

- S is a consistent cut iff $O(S) = \{o_1,..,o_M\}$ is consistent with causality

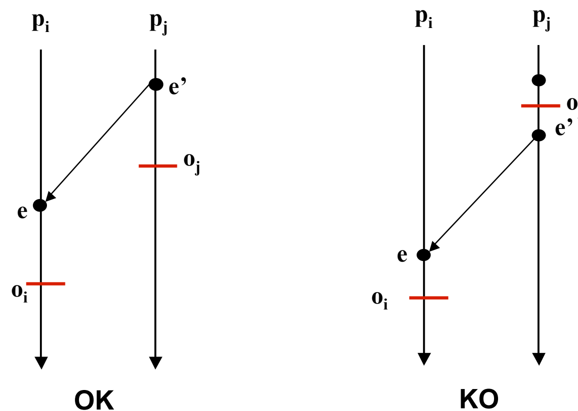
Distributed snapshots
Consistency with causality

- o_i is-consistent-with o_j iff
  for any $e \in E_i$, $e' \in E_j$,
  if $e <_H o_i$ and $e' <_H e$ then $e' <_H o_j$
- $\{o_1,..,o_M\}$ is consistent with causality if each pair o_i and o_j are consistent
- Informally: if an event e is observed in O(S) then all causally preceding events should be observed in O(S)

Distributed snapshots
Consistency with causality

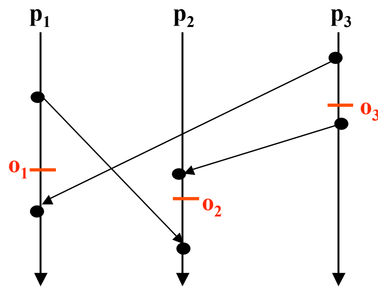
- o_i is-consistent-with o_j iff for any $e \in E_i$, $e' \in E_j$,
  if $e <_H o_i$ and $e' <_H e$ then $e' <_H o_j$
- $\{o_1,..,o_M\}$ is consistent with causality if each pair o_i and o_j are consistent
- An observation that indicates the receipt of a message must include the sending of the message

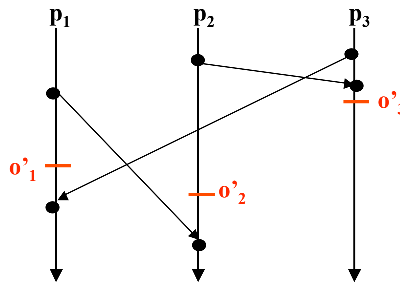
Distributed snapshots: intuition



Distributed snapshots

Exercise

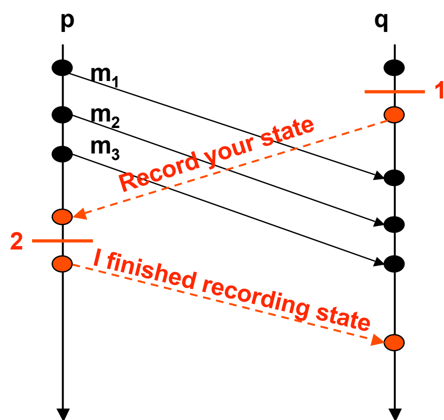




Is $O=\{o_1,o_2,o_3\}$ consistent with causality?

Is $O'=\{o'_1,o'_2,o'_3\}$ consistent with causality?

Messages sent but not yet received



Processor **q** observes its state at 1
Then, **q** sends **special message** to **p** asking **p** to record its state

Processor **p** records its state at 2
then **p** sends a **special message** to **q** saying that it recorded its state

When **q** receives the **special message** from **p**, processor **q** determines that messages **m_1**, **m_2**, **m_3** were sent but were not yet received

In the resulting global state, the state of **C_{pq}** must be **[m1, m2, m3]**

The algorithm
Overview

- Assumptions again!
  - Reliable FIFO communication
  - Asynchronous communication model
  - No process failure
  - Single initiator of snapshot (can be relaxed)
  - The graph of processes is connected
  - No need to know the number of processes

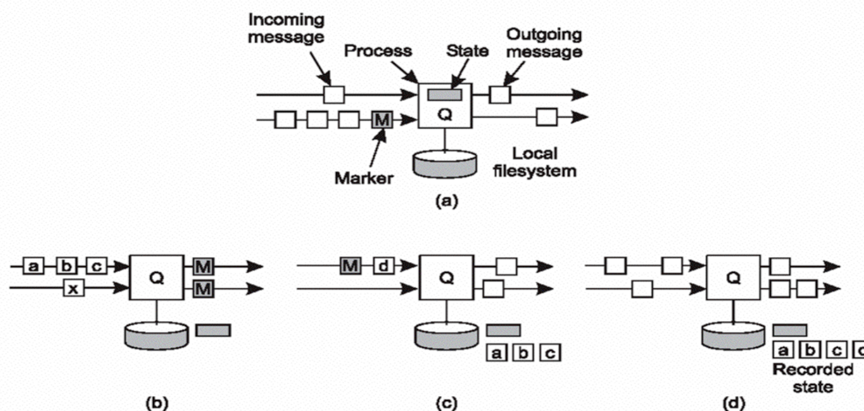
The algorithm
Overview

- Any process P can initiate taking a distributed snapshot
- P starts by recording its own local state s_P and subsequently it sends a marker along each of its outgoing channels

The algorithm
Overview

- When *Q* receives a marker through channel *C*, its action depends on whether it had already recorded its local state
- Not yet recorded
  - It records its local state, and sends the marker along each of its outgoing channels
- Already recorded
  - The marker on *C* indicates that the *channel's* state should be recorded: all messages received before this marker since the time *Q* recorded its own state
- *Q* is finished when it has received a marker along each of its incoming channels

The algorithm
Overview



(a) Q about to receive marker, (b) Q receives marker for first time and records state, (c) Q records all incoming messages, (d) Q has received markers on all incoming channels and finishes

Distributed snapshots
Termination

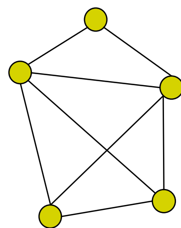
- Local termination of a snapshot
  - A process learns that its participation to a snapshot computation is terminated when it has recorded its state and received a *Marker* on each of its incoming channels

- When does an initiating process learn that the global snapshot computation is terminated?
  - There's a difference between the system has terminated and *knowing* at a process that the system has terminated
  - Knowing that the system has terminated requires extra work!

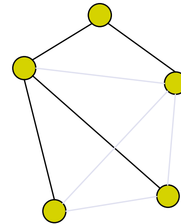
Distributed snapshots
Termination

- The algorithm for the snapshot computation can be easily adapted to construct a spanning tree of the system
- A spanning tree of the system whose connectivity is given by the undirected graph $G = (V,E)$ where V is the set of nodes and E is the set of edges, is
  - A *connected* graph (there is a path from any node to any other node)
  - An *acyclic* graph (no path has a cycle)

Distributed snapshots
Spanning trees



Graph G



One possible spanning tree of Graph G

Distributed snapshots
Spanning Tree Construction

- Each process maintains a variable *Parent* initially *undefined*
- Each process is initially *unmarked*
- A process that starts a snapshot computation is the *root* of the associated spanning tree. It sends *mark* messages to all neighbors, and sets its state to *marked*.
- When a process *Q* receives a *mark* message from *P*
  - If *Q* is unmarked then it sets *Parent* to *P*, it sends mark messages to all neighbors, and it sets its state to marked

Distributed snapshots: adding termination detection

- Using the spanning tree associated to a snapshot computation, we can let the process that started that snapshot computation detect when the computation is terminated

Distributed snapshots: adding termination detection

- When Q receives a *marker* from *P* for the first time
  - it sends a *marker* to all outgoing channels and sets *Parent* to *P*
  - It sends a *parent* message to P, and a *nonParent* message to all other outgoing channels
- When Q receives a *parent* message from P
  - It sets P as a child
- When Q receives a *nonParent* message from P
  - It sets P as a nonchild

Distributed snapshots: adding termination detection

- When *Q* has received a *marker* on all its incoming channels
  - It has finished: it changes its state to *finished*
  - It waits for a *finished* message from all its children
  - When all *finished* messages are received it sends *finished* to its parent
  - If Q is the root it learns that the computation has terminated

Distributed snapshots

- Exercise
- Explain why the proposed algorithm computes consistent global states

Distributed snapshots
the algorithm's properties

- Shows an important technique for designing distributed algorithms: Diffusing computation
  - The computation starts at one processor, then the computation diffuses through the whole set of processors
- Shows how to flush communication channels
  - The use of Marker messages
- Can record a global state that never occurred

The algorithm
rule 1 initiator, computing

- **Input**
  snapshotReq
- **Condition**
  S==computing
- **Action**
  DB.saveState(LocalState)
  for Q **in** OutChannels **do** Send(self;mark) **to** Q **end**
  S := recording

The algorithm
rule 2 mark, computing

- **Input**
  (P,mark)
- **Condition**
  S==computing
- **Action**
  DB.saveState(LocalState)
  DB.saveMessage(P,mark)
  for Q **in** OutChannels **do** Send(self;mark) **to** Q **end**
  S := recording

The algorithm
rule 3 non-mark message

- **Input**
  (P,M)
- **Condition**
  M is a computation message **and** S==recording
- **Action**
  DB.saveMessage(P,M)

The algorithm
rule 4 mark, recording

- **Input**
  (P;mark)
- **Condition**
  S==recording
- **Action**
  DB.saveMessage(P,mark)
  if DB.querySet(Q:Message(Q,mark)) == InChannels **then**
  S := finished
  else S := recording **end**

Distributed snapshots

- Use of non-consistent global state can lead to wrong conclusions
- E.g. a false deadlock can be claimed!

False deadlock illustrated

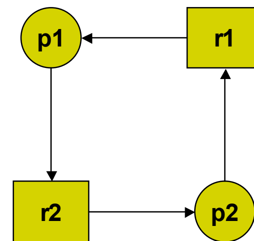
- Assume
  - $\mathbf{r_1}$ and $\mathbf{r_2}$ are two resources
  - Each resource is either *available* or *un-available*
  - $\mathbf{p_1}$ and $\mathbf{p_2}$ two processes that access resources
- To access a resource **r**, a process **p** sends Request (Req) to **r** and **p** starts to wait until it receives Ok from **r**

False deadlock illustrated

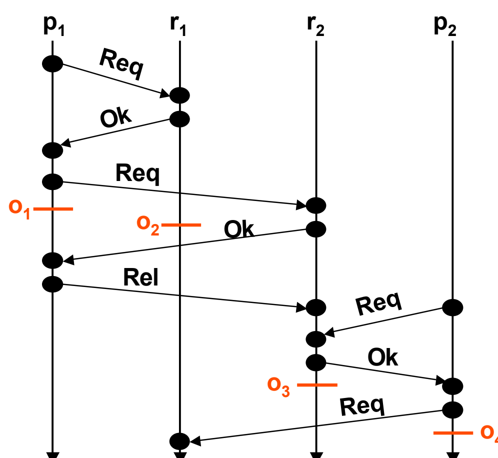
- When a resource **r** in state *available* receives Req from **p**,
  - **r** sends Ok to **p**
  - **r** becomes *un-available*
  - **r** starts to wait until it receives Release (Rel) from **p**
- A deadlock situation occurs when there is a cycle in the wait-for-graph amongst processes and resources

Wait-For Graph

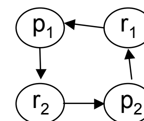
- Nodes are alternating processes p_i and resources r_j
- p→r means p requests r
- r→p means p has acquired r



False deadlock illustrated



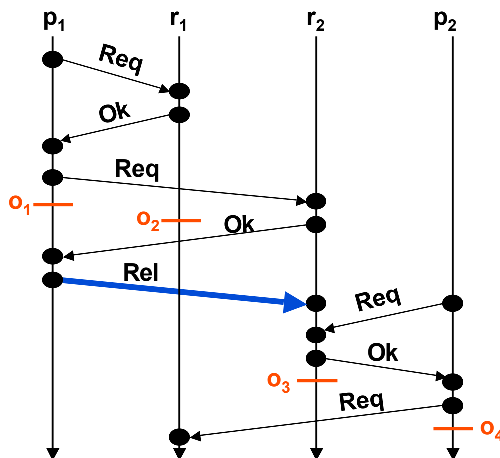
Inference from
o_1: p_1 waits for r_2
o_2: r_1 used by p_1
o_3: r_2 used by p_2
o_4: p_2 waits for r_1



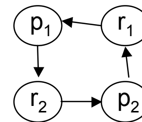
From $O=\{o_1,o_2,o_3,o_4\}$ the deadlock detector concludes there is a deadlock!

What's wrong with O?

False deadlock illustrated



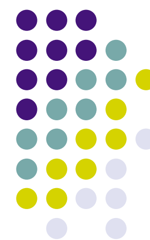
Inference from
o_1: p_1 waits for r_2
o_2: r_1 used by p_1
o_3: r_2 used by p_2
o_4: p_2 waits for r_1



From $O=\{o_1,o_2,o_3,o_4\}$ the deadlock detector concludes there is a deadlock!

What's wrong with O?

Distributed Snapshot with Termination



Variables/Constants

- Root:Boolean initially **false**
- DB initially **empty**
- S **in** {computing, recording, finished} initially computing
- Children, NonChildren initially **empty**
- Parent initially **undefined**
- InChannels, OutChannels

The algorithm
rule 1 initiator, computing

- **Input**
  snapshotReq
- **Condition**
  S==computing
- **Action**
  Root := **true**
  DB.saveState(LocalState)
  for Q **in** OutChannels **do** Send(self;mark) **to** Q **end**
  S := recording

The algorithm
rule 2 mark, computing

- **Input**
  (P,mark)
- **Condition**
  S==computing
- **Action**
  DB.saveState(LocalState)
  DB.saveMessage(P,mark)
  Send(self;parent) **to** P
  for Q **in** OutChannels/P **do**
      Send(self;nonparent) **to** Q
  end
  for Q **in** OutChannels **do** Send(self;mark) **to** Q **end**
  S := recording
  Parent := P

The algorithm
rule 3 computation message

- **Input**
  (P,M)
- **Condition**
  M is a computation message **and** S==recording
- **Action**
  DB.saveMessage(P,M)

The algorithm
rule 4 mark, recording

- **Input**
  (P;mark)
- **Condition**
  S==recording
- **Action**
  DB.saveMessage(P,mark)
  if DB.querySet(Q:Message(Q,mark)) == InChannels **and**
  all children are recorded
  then
  send (self;finished) **to** Parent
  S := finished
  else S := recording **end**

The algorithm
rule 4 parent, recording

- **Input**
  (P;parent)
- **Condition**
  S==recording
- **Action**
  Children.add(P)
  if DB.querySet(Q:Message(Q,mark)) == InChannels **and**
  all children are recorded
  then
  send (self;finished) **to** Parent
  S := finished
  else S := recording **end**

The algorithm
rule 4 parent, recording

- **Input**
  (P;nonparent)
- **Condition**
  S==recording
- **Action**
  NonChildren.add(P)
  if DB.querySet(Q:Message(Q,mark)) == InChannels **and**
  all children are recorded
  then
  send (self;finished) **to** Parent
  S := finished
  else S := recording **end**

all children are recorded

- Union(Children,NonChildren)==InChannels