



# Distributed Systems Synchronization

**Gianpaolo Cugola**

Dipartimento di Elettronica e Informazione  
Politecnico di Milano, Italy

`cugola@elet.polimi.it`

`http://home.dei.polimi.it/cugola`



# Contents

- **Synchronization in distributed systems: An Introduction**
- Synchronizing physical clocks
- Logical time
  - Scalar clocks
  - Vector clocks
- Mutual exclusion
- Leader election
- Collecting global state
  - Termination detection
- Distributed transactions
  - Detecting distributed deadlocks



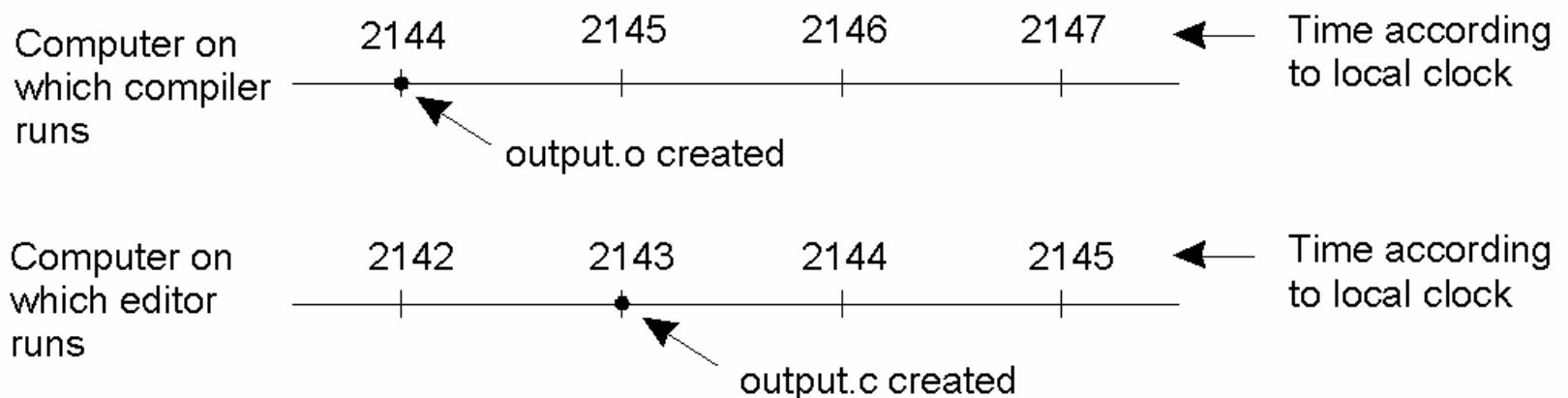
# Synchronization in distributed systems

- The problem of synchronizing concurrent activities arises also in non-distributed systems
- However, distribution complicates matters:
  - Absence of a global physical clock
  - Absence of globally shared memory
  - Partial failures
- In these lectures, we study distributed algorithms for:
  - Synchronizing physical clocks
  - Simulating time using logical clocks & preserving event ordering
  - Mutual exclusion
  - Leader election
  - Collecting global state & termination detection
  - Distributed transactions
  - Detecting distributed deadlocks



# Time and distributed systems

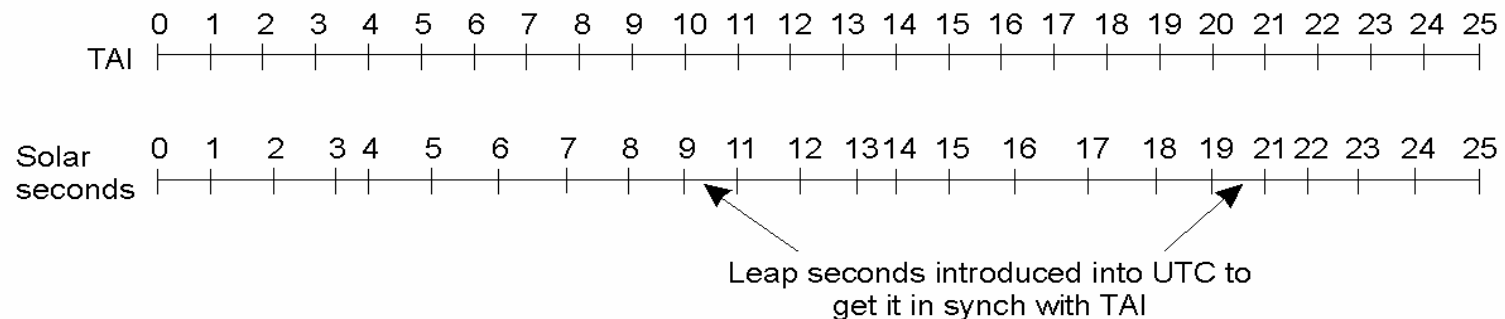
- Time plays a fundamental role in many applications:
  - Execute a given action at a given time
  - Time stamping objects/data/messages enables reconstruction of event ordering
    - File versioning
    - Distributed debugging
    - Security algorithms
- Problem: ensure all machines “see” the same global time
- Example: The make case





# Time

- Time is a tricky issue per se:
  - Up to 1940, time is measured astronomically
    - 1 second = 1/86400th of a mean solar day (the mean time interval between two consecutive transits of the sun)
    - Earth is slowing down, making measures “inaccurate”
  - Since 1948, time is measured physically (International Atomic Time)
    - 1 second = 9,192,631,770 transitions of an atom of Cesium 133
    - Collected and averaged in Paris from 50 labs around the world
  - Skew between TAI and solar days accommodated by UTC (Coordinated Universal Time) when greater than 800ms
    - Greenwich Mean Time is only astronomical
    - About 30 leap seconds from 1958 to now
    - UTC disseminate via radio stations (DCF77 in Europe, WWV in US), GPS and GEOS satellite systems





# Contents

- Synchronization in distributed systems: An Introduction
- **Synchronizing physical clocks**
- Logical time
  - Scalar clocks
  - Vector clocks
- Mutual exclusion
- Leader election
- Collecting global state
  - Termination detection
- Distributed transactions
  - Detecting distributed deadlocks



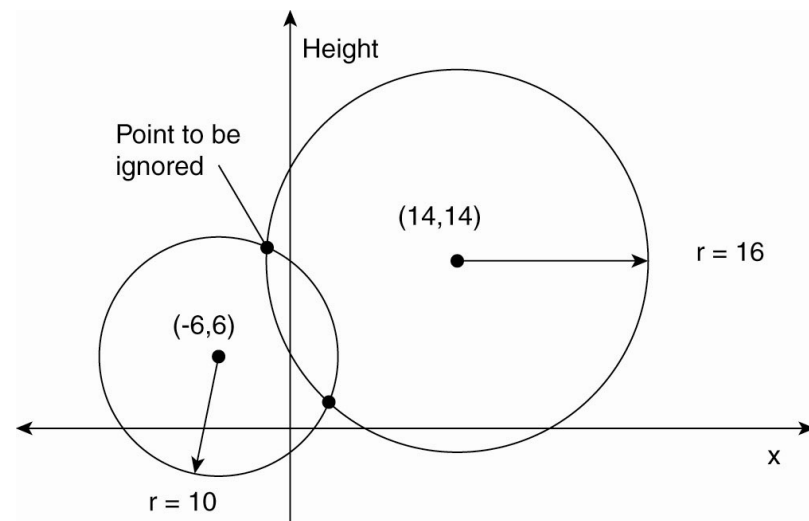
# Synchronizing physical clocks

- First of all: Computer clocks are not clocks, they are timers
- To guarantee synchronization:
  - Maximum *clock drift rate*  $\rho$  is a constant of the timer
    - For ordinary quartz crystals,  $\rho=10^{-6}$  s/s, i.e., 1s every 11.6 days
  - Maximum allowed *clock skew*  $\delta$  is an engineering parameter
  - If two clocks are drifting in opposite directions, during a time interval  $\Delta t$  they accumulate a skew of  $2\rho\Delta t$ 
    - ➔ resynch needed at least every  $\delta/2\rho$  seconds
- The problem is either:
  - Synchronize all clocks against a single one, usually the one with external, accurate time information (accuracy)
  - Synchronize all clocks among themselves (agreement)
- At least time monotonicity must be preserved
- Several protocols have been devised



# Positioning and time: GPS

- Basic idea: get an accurate account of time as a side effect of GPS
- How GPS works:
  - Position is determined by triangulation from a set of satellites whose position is known
  - Distance can be measured by the delay of signal
  - But satellite and receiver clock must be in sync
  - Since they are not we must take clock skew into account







# Positioning and time: GPS

- Let:
  - $\Delta_r$  be the unknown deviation of the receiver's clock w.r.t. the atomic clocks installed on board of satellites
  - $x_r, y_r, z_r$  be the unknown coordinates of the receiver
  - $T_i$  be the timestamp of message sent by a satellite  $i$
- Suppose that the messages sent by the  $i$ -th satellite is received at time  $T_r$  according to the receiver time, which corresponds to  $T_{\text{now}}$  in the real, actual time. Then:
  - $T_{\text{now}} = T_r - \Delta_r$
  - $\Delta_i = T_r - T_i$  is the measured delay of message
  - $c \times \Delta_i$  is the measured distance of satellite  $i$
- So  $c \times \Delta_i = c \times (T_{\text{now}} - T_i + \Delta_r) = c \times (T_{\text{now}} - T_i) + c \times \Delta_r$   
where the first addendum must be equal to the real distance:
$$\sqrt{(x_i - x_r)^2 + (y_i - y_r)^2 + (z_i - z_r)^2}$$
- With four satellites we have four equations in four unknowns (including  $\Delta_r$ )
  - We can solve them and determine both the node position and its clock skew



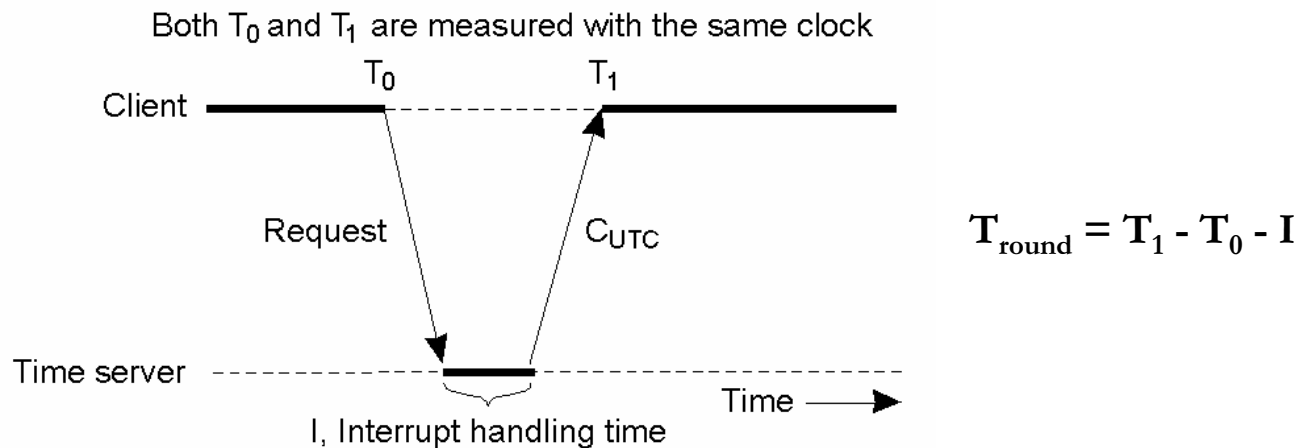
# Positioning and time: GPS

- Notice that things are more complex than previous description could suggest
  - Earth is not spherical
  - Atomic clocks in the satellites are not perfectly in sync
  - The position of satellites is not known precisely
  - The receiver's clock has a finite accuracy
  - The signal propagation speed is not constant
  - ...
- In any case, even cheap GPS receivers can be precise within range of few meters and few tens of nanoseconds



# Simple algorithms: Cristian's

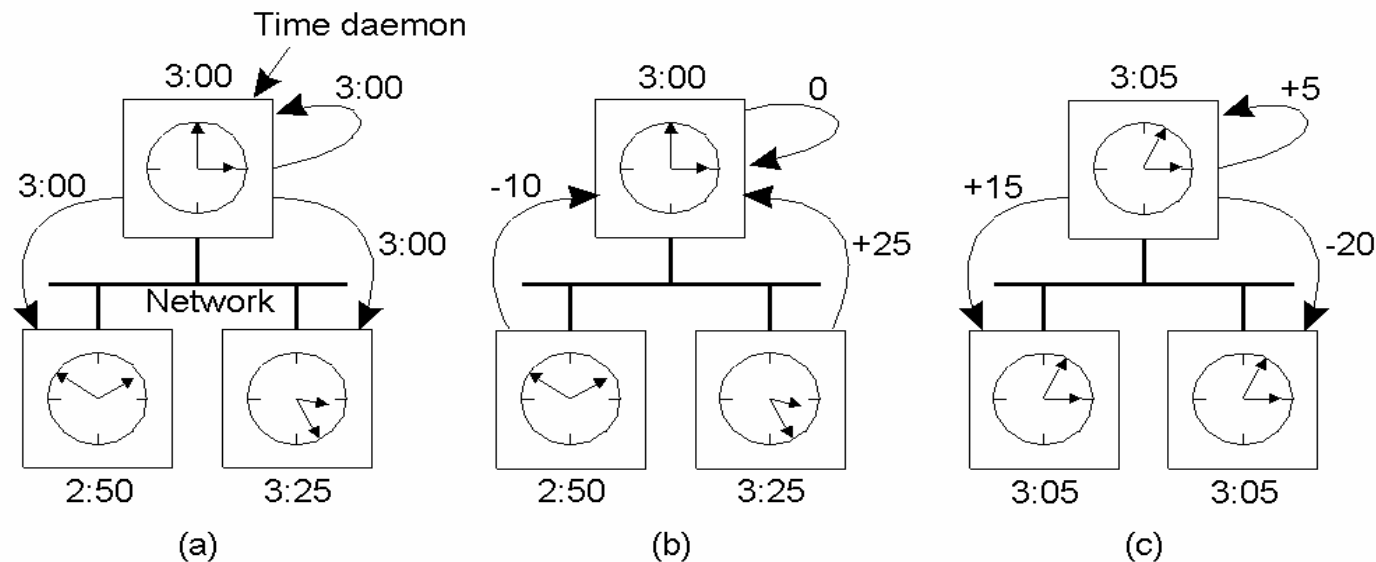
- Cristian (1989)
  - Periodically, each client sends a request to the time server
  - Messages are assumed to travel fast w.r.t. required time accuracy
  - Problems:
    - Major: time might run backwards on client machine. Therefore, introduce change gradually (e.g., advance clock 9ms instead of 10ms on each clock tick)
    - Minor: it takes a non-zero amount of time to get the message to the server and back
      - Measure round-trip time and adjust, e.g.,  $T_1 = C_{UTC} + T_{round}/2$
      - Average over several measurements





# Simple algorithms: Berkeley

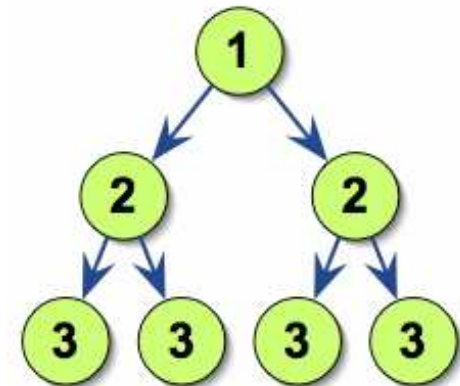
- Introduced by Berkeley UNIX (1989)
- The time server is active: It collects the time from all clients, averages it, and then retransmits the required adjustment





# Network Time Protocol (NTP)

- Designed for UTC synch over large-scale networks
  - Used in practice over the Internet, on top of UDP
  - Estimate of 10-20 million NTP clients and servers
  - Widely available (even under Windows)
  - Synchronization accuracy:  $\sim 1\text{ms}$  over LANs, 1-50ms over the Internet
  - More info at [www.ntp.org](http://www.ntp.org)
- Hierarchical synchronization subnet organized in strata
  - Servers in stratum 1 are directly connected to a UTC source
  - Lower strata (higher levels) provide more accurate information
  - Leaf servers execute in users' workstations
  - Connections and strata membership change over time
- Synchronization mechanisms
  - Multicast (over LAN)
    - Servers periodically multicast their time to other computers on the same LAN
  - Procedure-call mode
    - Similar to Cristian's
  - Symmetric mode
    - For higher levels that need the highest accuracies





# NTP: Procedure-call and symmetric mode

- Servers exchange pairs of messages, each bearing timestamps of recent message events
  - The local time when the previous message between the pairs was sent and received, and the local time when the current message was transmitted
- If  $t$  and  $t'$  are the messages' transmission times, and  $o$  is the time offset of the clock at B relative to that at A, then:
  - $T_{i-2} = T_{i-3} + t + o$  and  $T_i = T_{i-1} + t' - o$

This leads to calculate the total transmission time  $d_i$  as:

$$d_i = t + t' = T_{i-2} - T_{i-3} + T_i - T_{i-1}$$

If we define  $o_i$  as:

$$o_i = (T_{i-2} - T_{i-3} + T_{i-1} - T_i)/2$$

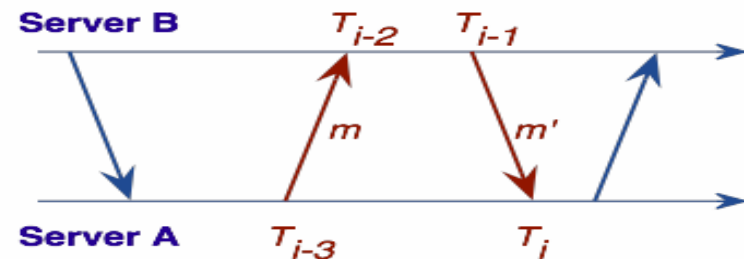
from the first two formulas we have:

$$o = o_i + (t' - t)/2$$

and since  $t, t' \geq 0$

$$o_i - d_i/2 = o_i + (t' - t)/2 - t' \leq o \leq o_i + (t' - t)/2 + t = o_i + d_i/2$$

**Thus  $o_i$  is an estimate of the offset and  $d_i$  is a measure of the accuracy of this estimate**





# Contents

- Synchronization in distributed systems: An Introduction
- Synchronizing physical clocks
- **Logical time**
  - **Scalar clocks**
  - **Vector clocks**
- Mutual exclusion
- Leader election
- Collecting global state
- Termination detection
- Distributed transactions
- Detecting distributed deadlocks



# Some observations

- In many applications it is sufficient to agree on a time, even if it is not accurate w.r.t. the absolute time
- What matters is often the ordering and causality relationships of events, rather than the timestamp itself
- If two processes do not interact, it is not necessary that their clocks be synchronized





# Logical time - Scalar clocks

L. Lamport. "Time, clocks, and the ordering of events in a distributed system".  
Communications of the ACM, 21(7):558-565, July 1978.

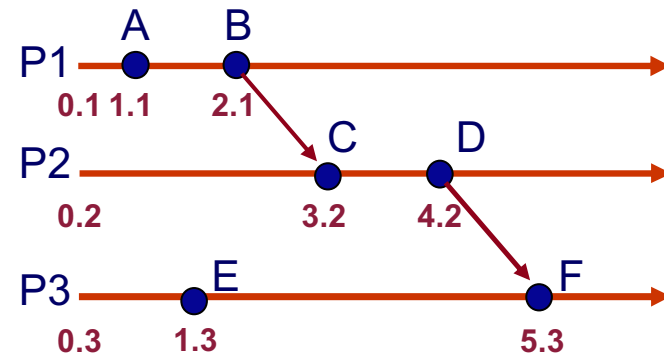
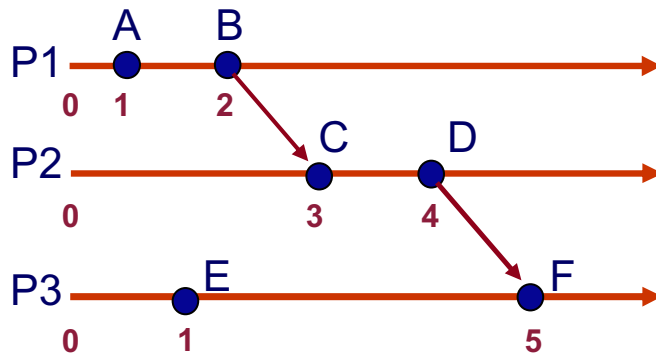
- Let define the *happens-before* relationship  $e \rightarrow e'$ , as follows:
  - If events  $e$  and  $e'$  occur in the same process and  $e$  occurs before  $e'$ , then  $e \rightarrow e'$
  - If  $e = \text{send}(msg)$  and  $e' = \text{recv}(msg)$ , then  $e \rightarrow e'$
  - $\rightarrow$  is transitive
- If neither  $e \rightarrow e'$  nor  $e' \rightarrow e$ , they are concurrent ( $e \parallel e'$ )
- The happens-before relationship captures *potential causal ordering* among events
  - Two events can be related by the happens-before relationship even if there is no real (causal) connection among them
  - Also, since information can flow in ways other than message passing, two events may be causally related even neither of them happens-before the other
- Lamport invented a simple mechanism by which the happened before ordering can be captured numerically
  - Using integers to represent the clock value
  - No relationship with a physical clock whatsoever



# Logical time - Scalar clocks

L. Lamport. "Time, clocks, and the ordering of events in a distributed system".  
Communications of the ACM, 21(7):558-565, July 1978.

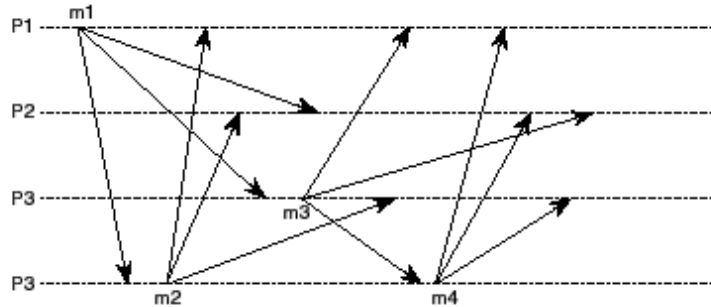
- Each process  $p_i$  keeps a logical scalar clock  $L_i$ 
  - $L_i$  starts at zero
  - $L_i$  is incremented before  $p_i$  sends a message
  - Each message sent by  $p_i$  is timestamped with  $L_i$
  - Upon receipt of a message,  $p_i$  sets  $L_i$  to:  
 $\text{MAX}(\text{msg timestamp}, L_i) + 1$
- It can easily be shown, by induction on the length of any sequence of events relating two events  $e$  and  $e'$ , that:  
 $e \rightarrow e' \Rightarrow L(e) < L(e')$
- Note that only partial ordering is achieved. Total ordering can be obtained trivially by attaching process IDs to clocks





# Exercise

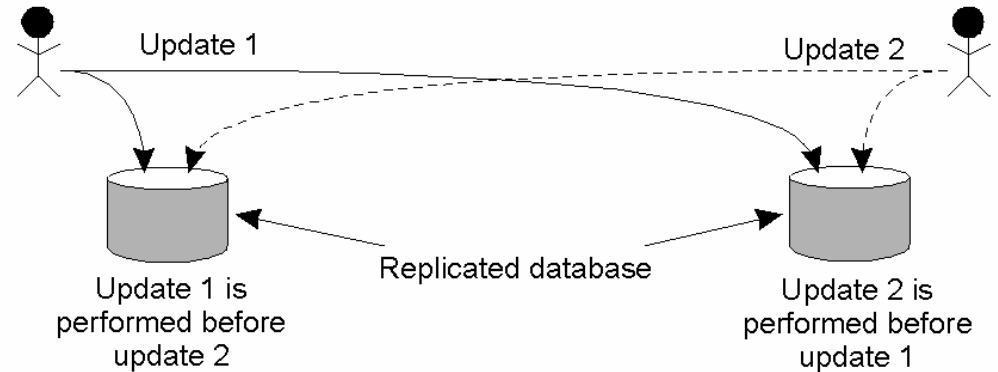
- Consider 4 processes exchanging messages as in figure:



Which is the value of Lamport's clocks at the end of the reported period?



# Example: Totally ordered multicast



- Updates in sequence:
  - Customer deposits \$100
  - Bank adds 1% interest
- Updates are propagated to all locations:
  - If updates in the same order at each copy, consistent result (e.g., \$1111)
  - If updates arrive in opposite orders, inconsistent result (e.g., \$1110)
- Totally ordered multicast delivers messages in the same global order
- Using logical clocks (assuming reliable and FIFO links):
  - Messages are sent and acknowledged using multicast
  - All messages (including acks) carry a timestamp with the sender's scalar clock
  - Scalar clocks ensures that the timestamps reflect a consistent global ordering of events
  - Receivers (including the sender) store all messages in a queue, ordered according to its timestamp
  - Eventually, all processes have the same messages in the queue
  - A message is delivered to the application only when it is at the highest in the queue and all its acks have been received
    - Since each process has the same copy of the queue, all messages are delivered in the same order everywhere



# Vector Clocks

- Problem:
  - In scalar clocks,  $e \rightarrow e' \Rightarrow L(e) < L(e')$
  - But the reverse does not necessarily hold, e.g., if  $e \mid e'$
- Solution: Vector clocks
- In vector clocks each process  $P_i$  maintains a vector  $V_i$  of  $N$  values ( $N = \# \text{processes}$ ) such that:
  - $V_i[i]$  is the number of events that have occurred at  $P_i$
  - If  $V_i[j] = k$  then  $P_i$  knows that  $k$  events have occurred at  $P_j$
- Rules for updating the vectors:
  - Initially,  $V_i[j] = 0$  for all  $i, j$
  - Local event at  $P_i$  causes an increment of  $V_i[i]$
  - $P_i$  attaches a timestamp  $t = V_i$  in all messages it sends (incrementing  $V_i[i]$  just before sending the message, according to previous rule)
  - When  $P_i$  receives a message containing  $t$ , it sets  $V_i[j] = \max(V_i[j], t[j])$  for all  $j \neq i$  and then increments  $V_i[i]$

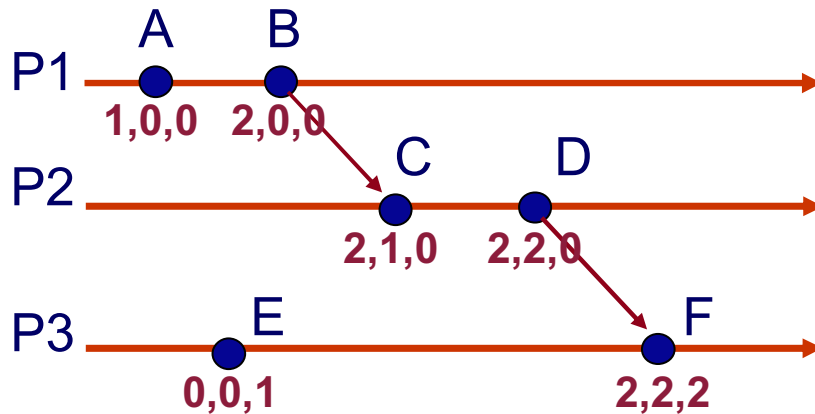


# Vector Clocks (cont'd)

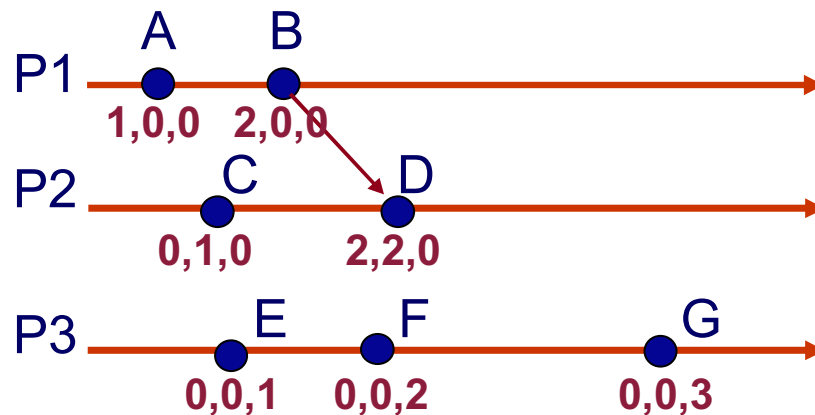
- Definitions (partial ordering)
  - $V = V'$     *iff*     $V[j] = V'[j]$ , for all  $j$
  - $V \leq V'$     *iff*     $V[j] \leq V'[j]$ , for all  $j$
  - $V < V'$     *iff*     $V \leq V' \wedge V \neq V'$
  - $V \parallel V'$     *iff*     $\neg(V < V') \wedge \neg(V' < V)$
- An isomorphism between the set of partially ordered events and their timestamps (i.e., vector clocks)
- Determining causality:
  - $e \rightarrow e' \Leftrightarrow V(e) < V(e')$
  - $e \parallel e' \Leftrightarrow V(e) \parallel V(e')$



# Examples



- By looking *only* at the timestamps we are able to determine whether two events are causally related or concurrent



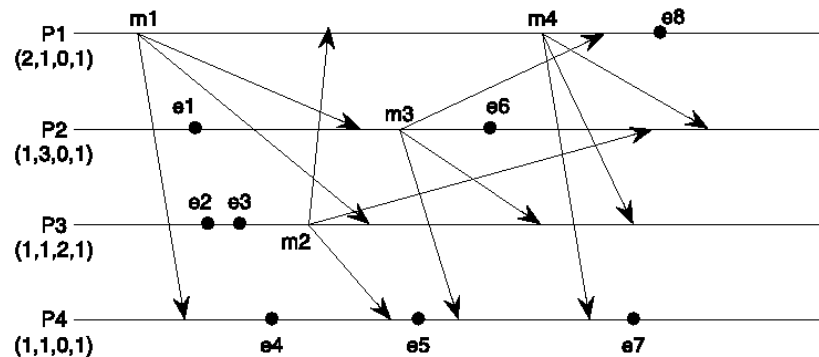


# Exercise

- Three processes are involved in a distributed algorithm. At the end their vector clocks are:
  - P1 : (4, 5, 6)      P2 : (5, 6, 6)      P3 : (3, 2, 7)

Is this possible? Why?

- Consider 4 processes exchanging messages as in figure:

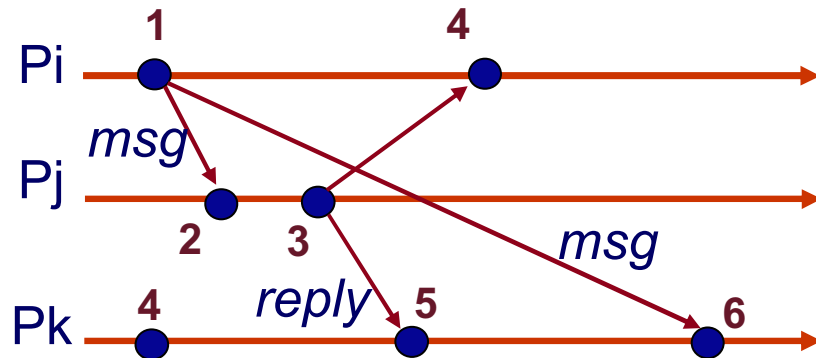


Which is the value of each process' vector clock at the end of the reported period?

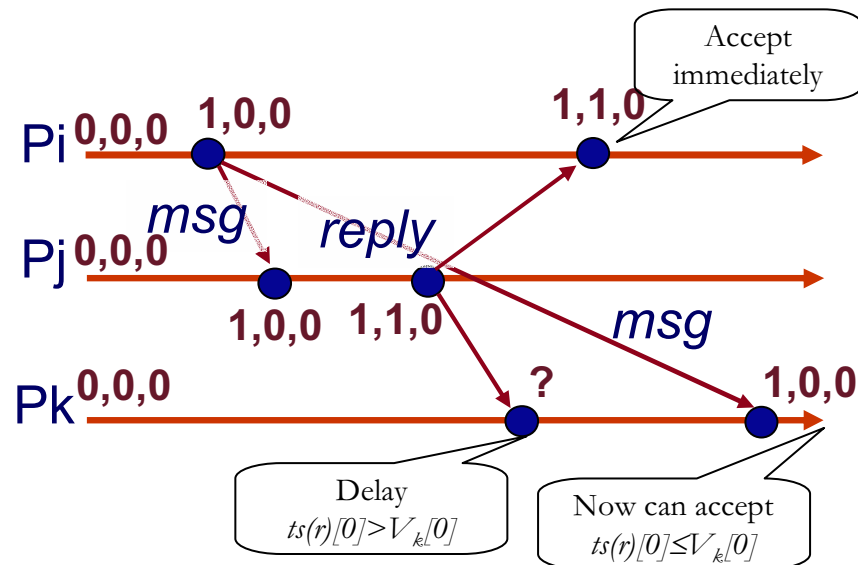




# Vector clocks for causal delivery



- A slight variation of vector clocks can be used to implement causal delivery of messages in a totally distributed way
- Example: bulletin boards
  - Messages and replies sent (using reliable, FIFO ordered, channels) to all the boards in parallel
  - Need to preserve the ordering *only* between messages and replies
  - Totally ordered multicast is too strong
    - If M1 arrives before M2, it does not necessarily mean that the two are related
  - Using vector clocks:
    - Variation: Increment clock only when sending a message. On receive, just merge, not increment
    - Hold a reply until the previous messages are received:
      - $ts(r)[j] = V_k[j] + 1$
      - $ts(r)[i] \leq V_k[i]$  for all  $i \neq j$





# Contents

- Synchronization in distributed systems: An Introduction
- Synchronizing physical clocks
- Logical time
  - Scalar clocks
  - Vector clocks
- **Mutual exclusion**
- Leader election
- Collecting global state
  - Termination detection
- Distributed transactions
  - Detecting distributed deadlocks



# Mutual exclusion

- Required to prevent interference and ensure consistency of resource access
- Critical section problem, typical of OS
  - But here, no shared memory
- Requirements:
  - Safety property: At most one process may execute in the critical section at a time
  - Liveness property: All requests to enter/exit the critical section eventually succeed (no deadlock, no starvation)
  - Optional: If one request happened-before another, then entry is granted in that order
- Assumptions:
  - **Reliable channels and processes**
- Simplest solution: A server coordinating access
  - Emulates a centralized solution
  - Server manages the lock using a “token”
  - Resource access request and release obtained with respective messages to the coordinator
  - Easy to guarantee mutual exclusion and fairness
  - Drawbacks: Performance bottleneck and single point of failure

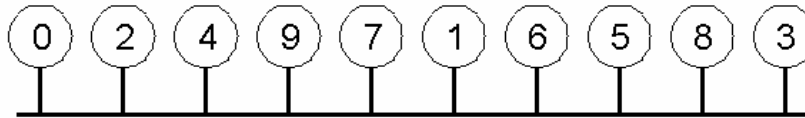


# Mutual exclusion with scalar clocks

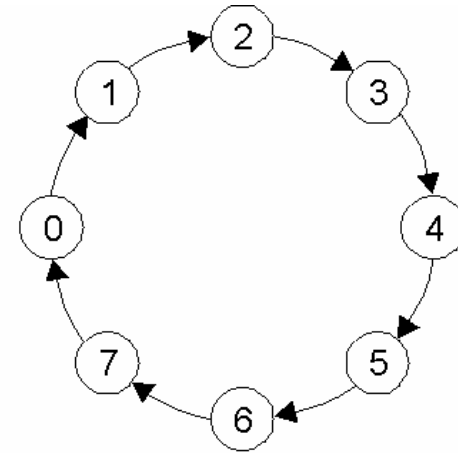
- To request access to a resource:
  - A process  $P_i$  multicasts a resource request message  $m$ , with timestamp  $T_m$ , to all processes (including itself)
  - Upon receipt of  $m$ , a process  $P_j$ :
    - If it does not hold the resource and it is not interested in holding the resource,  $P_j$  sends an acknowledgment to  $P_i$
    - If it holds the resource,  $P_j$  puts the requests into a local queue ordered according to  $T_m$  (process ids are used to break ties)
    - If it is also interested in holding the resource and has already sent out a requests,  $P_j$  compares the timestamp  $T_m$  with the timestamp of its own requests
      - If the  $T_m$  is the lowest one,  $P_j$  sends an acknowledgement to  $P_i$ , otherwise it put the request into the local queue above
- On releasing the resource, a process  $P_i$  acknowledges all the requests queued while using the resource
- A resource is granted to  $P_i$  when its request has been acknowledged by all the other processes



# A token ring solution



(a)



(b)

- Processes are logically arranged in a ring, regardless of their physical connectivity
  - At least for the purpose of mutual exclusion
- Access is granted by a token that is forwarded along a given direction on the ring
  - A process not interested in accessing the resource forwards the token
  - Resource access is achieved by retaining the token
  - Resource release is achieved by forwarding the token



# Comparison

| Algorithm             | Messages per entry | Delay before entry (in message times) | Problems                  |
|-----------------------|--------------------|---------------------------------------|---------------------------|
| Centralized           | 2                  | 2                                     | Coordinator crash         |
| Distributed (Lamport) | $2(n - 1)$         | $2(n - 1)$                            | Crash of any process      |
| Token ring            | 1 to $\infty$      | 0 to $n - 1$                          | Lost token, process crash |

Request, grant

If nobody wants to enter the critical section, the token circulates indefinitely

**Comments?**



# Contents

- Synchronization in distributed systems: An Introduction
- Synchronizing physical clocks
- Logical time
  - Scalar clocks
  - Vector clocks
- Mutual exclusion
- **Leader election**
- Collecting global state
  - Termination detection
- Distributed transactions
  - Detecting distributed deadlocks



# Leader election

- Many distributed algorithms require a process to act as a coordinator (or some other special role)
  - E.g., for server-based mutual exclusion
- Problem: Make everybody agree on a new leader
  - When the old is no longer available, e.g., because of failure or applicative reasons
- Minimal assumption: **Nodes are distinguishable**
  - Otherwise, no way to perform selection
  - Typically use the identifier (the process with the highest ID becomes the leader) or some other measure (e.g., 1/load)
- Also, closed system: **Processes know each other and their IDs**
  - But do not know who is up and who has failed
- The non-crashed process with the highest ID at the end of the election must be the winner
  - And every other non-crashed process must agree on this
- Algorithms differ on the selection process



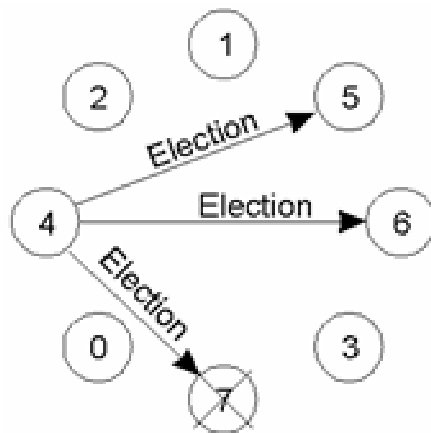


# The bully election algorithm

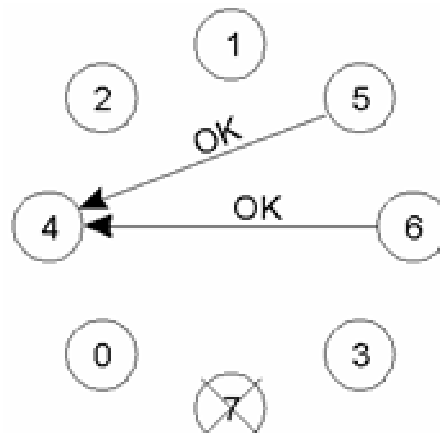
- Additional assumptions
  - Reliable links
  - It is possible to decide who has crashed (synchronous system)
- Algorithm
  - When any process  $P$  notices that the actual coordinator is no longer responding requests it initiates an election
  - $P$  sends an *ELECT* message, including its ID, to all other processes with higher IDs
  - If no-one responds  $P$  wins and sends a *COORD* message to the processes with lower IDs
  - If a process  $P'$  receives an *ELECT* message it responds (stopping the former candidate) and starts a new election (if it has not started one already)
  - If a process that was previously down comes back up, it holds an election
    - If it happens to be the highest-numbered process currently running it wins the election and takes over the coordinator's job (hence the name of the algorithm)



# The bully election algorithm: Example

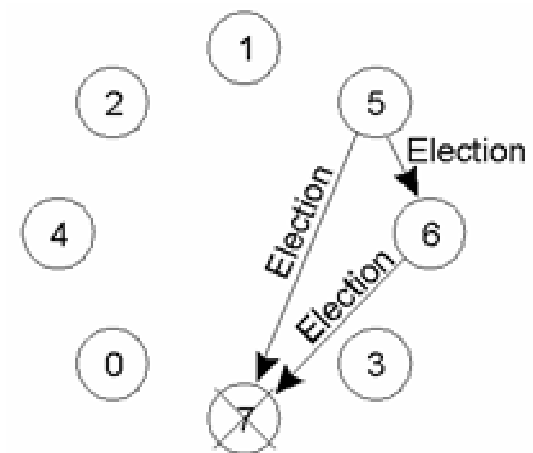


(a)

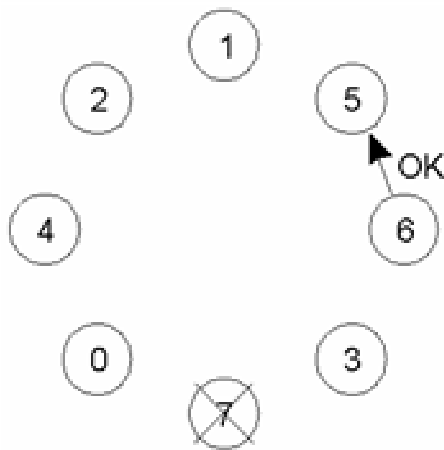


Previous coordinator  
has crashed

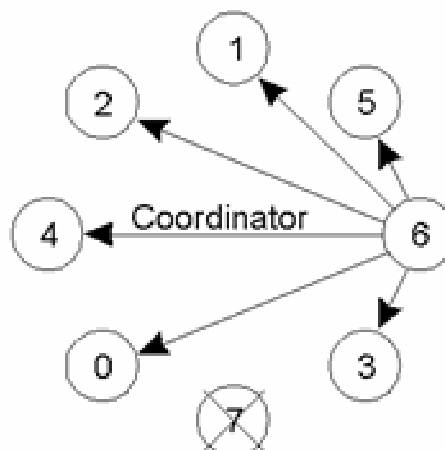
(b)



(c)



(d)



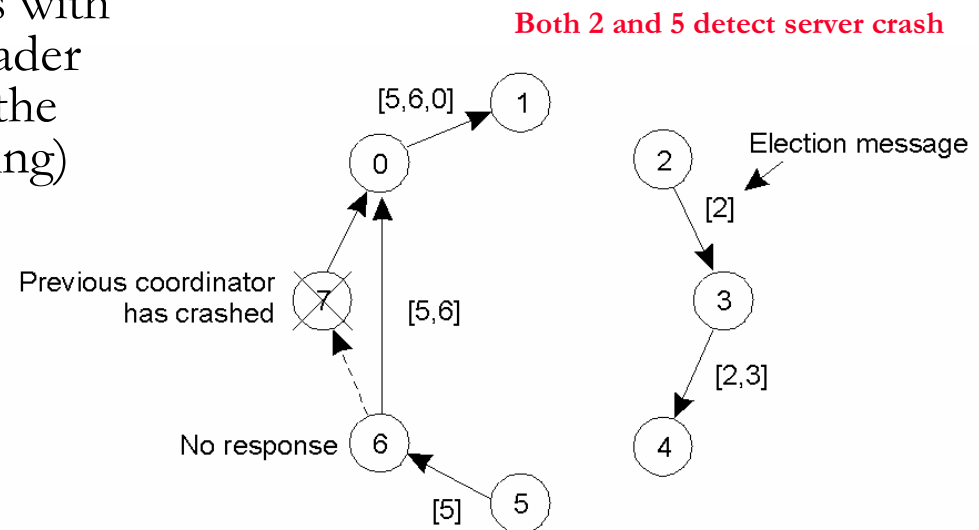
(e)

- The leader, 7, fails
  - Process 4 (noticing the failure of 7) holds an election
  - Process 5 and 6 respond, telling 4 to stop
  - Now 5 and 6 each hold an election
  - 6 tells 5 to stop
  - 6 wins and tells everyone
- When 7 comes back it holds an election and wins



# A ring-based algorithm

- Assume a (physical or logical) ring topology among nodes
- When a process detects a leader failure, it sends an *ELECT* message containing its ID to the closest alive neighbor
- Upon receipt of the election message a process *P*:
  - If *P* is not in the message, add *P* and propagate to next alive neighbor
  - If *P* is in the list, change message type to *COORD*, and re-circulate
- On receiving a *COORD* message, a node considers the process with the highest ID as the new leader (and is also informed about the remaining members of the ring)
- Multiple messages may circulate at the same time
  - Eventually converge to the same content





# Exercise

- Compare the two election algorithms in terms of:
  - Number of messages required to end the election
  - Assumptions
  - Ease of guaranteeing such assumptions



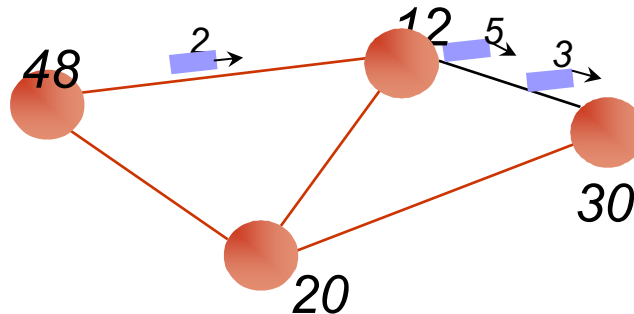
# Contents

- Synchronization in distributed systems: An Introduction
- Synchronizing physical clocks
- Logical time
  - Scalar clocks
  - Vector clocks
- Mutual exclusion
- Leader election
- **Collecting global state**
  - **Termination detection**
- Distributed transactions
  - Detecting distributed deadlocks



# Capturing global state

- The global state of a distributed system consists of the local state of each process (depends on the application), together with the message in transit over the links
- Useful to know for distributed debugging, termination detection, deadlock detection, ...
- Banking example:
  - Constant amount of money in the bank system (e.g., 120)
  - Money is transferred among banks in messages
  - Problem: find out if any money accidentally lost

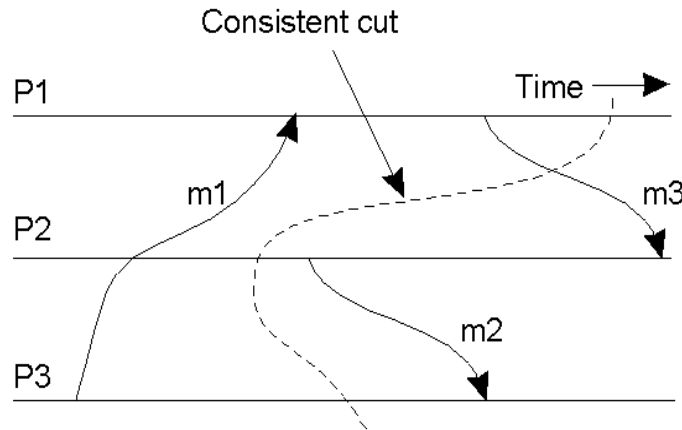


- Capturing the global state of a distributed system would be easy if we could access a global clock... but we do not have one
- We must accept recording the state of each process at, potentially, different times

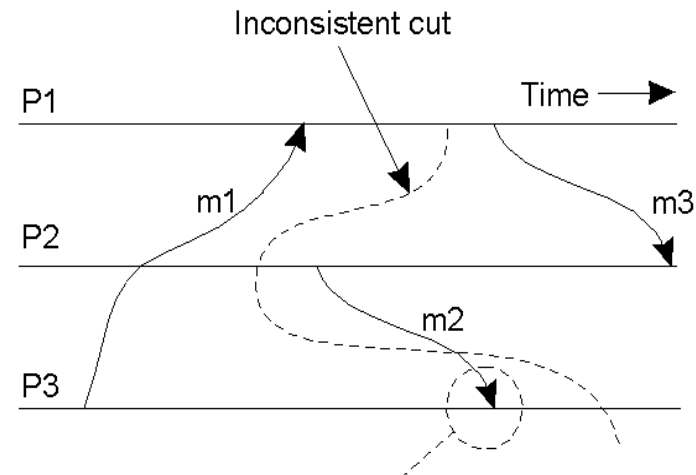


# Cuts & distributed snapshots

- A *distributed snapshot* reflects a (consistent, global) state in which the distributed system might have been
- Particular care must be taken when reconstructing the global state to preserve consistency
  - If a message receipt is recorded, the message sending must as well, but the contrary is not required
- Conceptual tool: *Cut*



(a)



(b)



# More on consistent cuts

- Formally a *cut* of a system  $S$  composed of  $N$  processes  $p_1, \dots, p_n$  can be defined as the union of the histories of all its processes up to a certain event

$$C = h_1^{k_1} \cup h_2^{k_2} \cup \dots \cup h_n^{k_n}$$

where

$$h_i^{k_i} = \langle e_i^0, e_i^1 \dots e_i^{k_i} \rangle$$

- A cut  $C$  is consistent if for any event  $e$  it includes, it also includes all the events that happened before  $e$ .

Formally:

$$\forall e, f: e \in C \wedge f \rightarrow e \Rightarrow f \in C$$





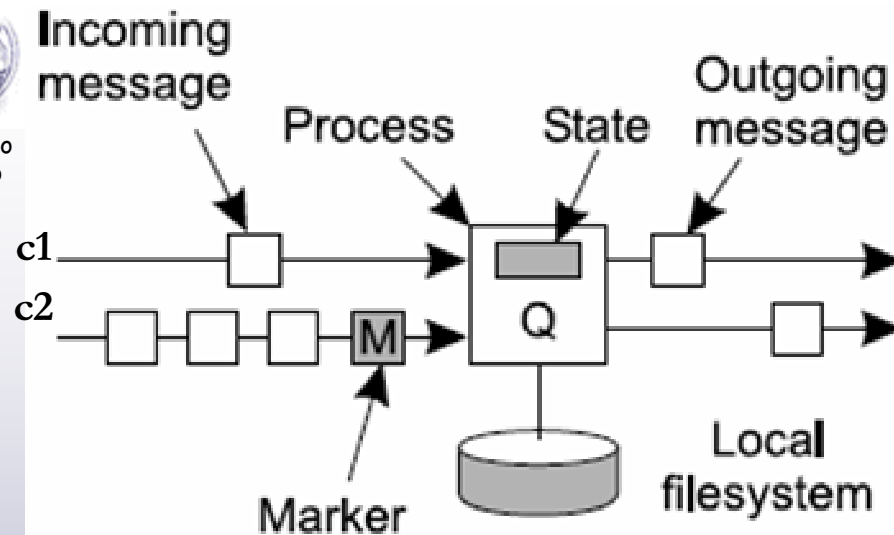
# Distributed snapshot

## Chandy-Lamport, 1985

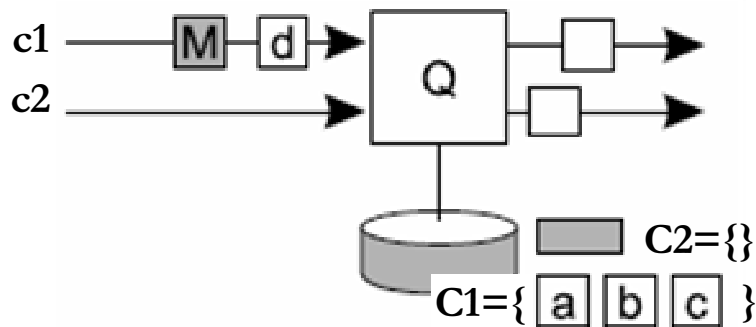
- Assume FIFO, reliable links/nodes + strongly connected graph
- Any process  $p$  may initiate a snapshot by
  - Recording its internal state
  - Sending a token on all outgoing channels
    - This signals a snapshot is being run
  - Start recording a local snapshot
    - I.e., record messages arriving on every incoming channel
- Upon receiving a token, a process  $q$ 
  - If not already recording local snapshot
    - Records its internal state
    - Sends a token on all outgoing channels
    - Start recording a local snapshot (see above)
  - In any case stop recording incoming message on the channel the token arrived along
- Recording messages
  - If a message arrives on a channel which is recording messages, record the arrival of the message, then process the message as normal
  - Otherwise, just process the message as normal
- Each process considers the snapshot ended when tokens have arrived on all its incoming channels
  - Afterwards, the collected data can be sent to a single collector of the global state



Politecnico  
di Milano

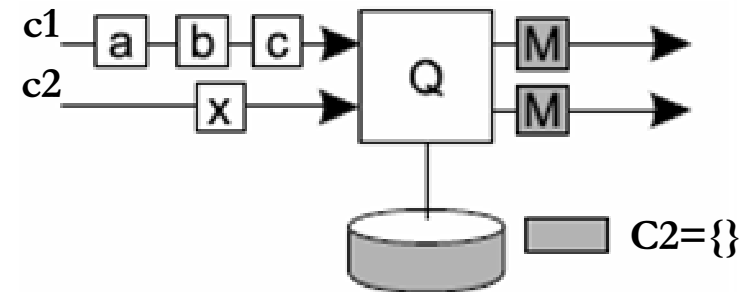


Normal processing, first marker  
about to be received

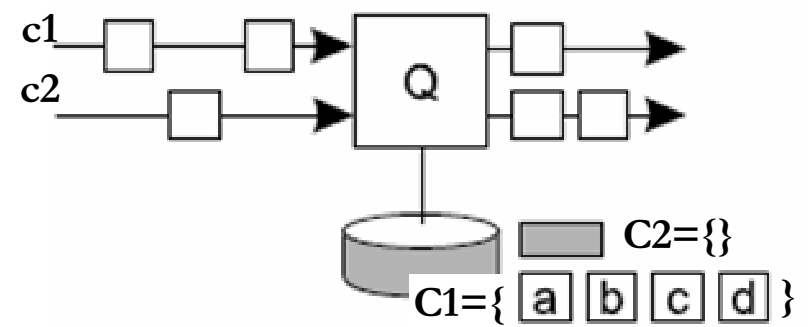


Recording of messages from the incoming  
links from which a token has been  
received

## Processing



Token just received for the first time,  
state saved, token forwarded to outgoing  
links, begin recording messages



Token received,  
state recording is stopped

Recorded  
state



# Characterizing the observed state

## Theorem

The distributed snapshot algorithm selects a consistent cut

## Proof

Let  $e_i$  and  $e_j$  be two events occurring at  $p_i$  and  $p_j$ , respectively, such that  $e_i \rightarrow e_j$ .  
Suppose  $e_j$  is part of the cut and  $e_i$  is not. This means  $e_j$  occurred before  $p_j$  saved its state, while  $e_i$  occurred after  $p_i$  saved its state

If  $p_i = p_j$  this is trivially impossible, so suppose  $p_i \neq p_j$

Let  $m_1, \dots, m_b$  be the sequence of messages that give rise to the relation  $e_i \rightarrow e_j$

If  $e_i$  occurred after  $p_i$  saved its state then  $p_i$  sent a marker ahead of  $m_1, \dots, m_b$

By FIFO ordering over channels and by the marker propagating rules it results that  $p_j$  received a marker ahead of  $m_1, \dots, m_b$

By the marker processing rule it results that  $p_j$  saved its state before receiving  $m_1, \dots, m_b$ , i.e., before  $e_j$ , which contradicts our initial assumption



# Some observations

- Important: the distributed snapshot algorithm does not require blocking of the computation
  - Collecting the snapshot is interleaved with processing
- What happens if the snapshot is started at more than one location at the same time?
  - Easily dealt with by associating an identifier to each snapshot, set by the initiator
- Several variations have been devised
  - E.g., incremental snapshots take an initial snapshot, each node remembers where it has sent/received messages, and when a new snapshot is requested, only these links are included in the result
- How is the snapshot result collected?
  - Again, several variations (this step is not part of the algorithm)



# Termination detection

- Want to know when a computation has completed or deadlocked (no more useful work can be done)
  - All processes should be idle
  - There should be no messages in the system
    - Messages need to be processed, i.e., some process must become non-idle
- Can a distributed snapshot be used?
  - Yes, but channels must be empty when finished
  - Simple solution (from Tanenbaum):
    - Let call *predecessor* of a process  $p$ , the process  $q$  from which it got the first marker. The successors of  $p$  are all those processes  $p$  sent the marker
    - When a process  $p$  finishes its part of the snapshot, it sends a DONE message back to its predecessor only if two conditions are met:
      - All of  $p$ 's successors have returned a DONE message
      - $P$  has not received any message between the point it recorded its state and the point it had received the marker along each of its incoming channels
    - In any other case  $p$  sends a CONTINUE
    - If the initiator receives all DONE the computation is over; otherwise, another snapshot is necessary

Does it work?



# Diffusing computations

- In a diffusing computation, initially all process are idle except the init process
- A process is activated only by a message being sent to it
- Termination condition (same as before): when processing is complete at each node, and there are no more messages in the system



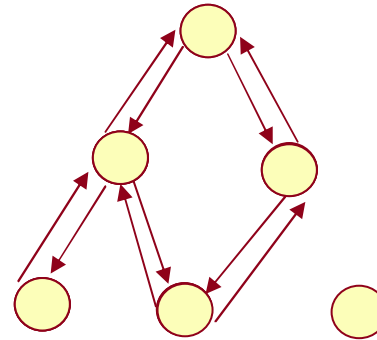
# Dijkstra-Scholten termination detection

- Works for diffusing computations
- Key concepts
  - Create a tree out of the active processes
  - When a node finishes processing and it is a leaf, it can be pruned from the tree
  - When only the root remains, and it has completed processing, the system has terminated
- Challenges
  - How to create a tree, and keep it acyclic
  - Must detect when a node is a leaf



# Dijkstra-Scholten termination detection

- Each node keeps track of nodes it sends a message to (those it may have woken up), its children
- If a node was already awake when the message arrived, then it is already part of the tree, and should not be added as a child of the sender
- When a node has no more children and it is idle, it tells its parent to remove it as a child



 Processing

 Idle





# Comparison of termination detection approaches

- Use distributed snapshot
  - Overhead is one message per link
  - Plus cost to collect result
  - If system not terminated, need to run again!
- Use Dijkstra-Scholten
  - Overhead depends on the number of messages in the system
    - Acknowledgments sent when already part of network and when become idle
    - Can be added to network more than once, so this value is not fixed
  - Does not involve never-activated processes
  - Termination detected when last ack received



# Contents

- Synchronization in distributed systems: An Introduction
- Synchronizing physical clocks
- Logical time
  - Scalar clocks
  - Vector clocks
- Mutual exclusion
- Leader election
- Collecting global state
  - Termination detection
- **Distributed transactions**
  - **Detecting distributed deadlocks**



# Distributed transactions

- Protect a shared resource against simultaneous access by several concurrent processes
- Transactions are sequences of operations, defined with appropriate programming primitives, e.g.:

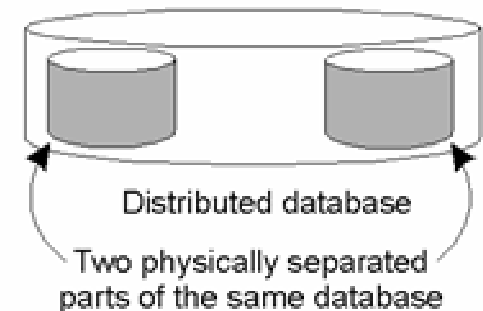
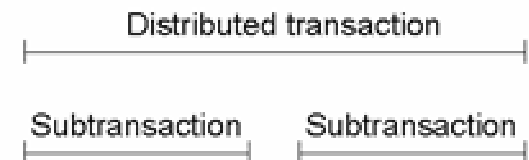
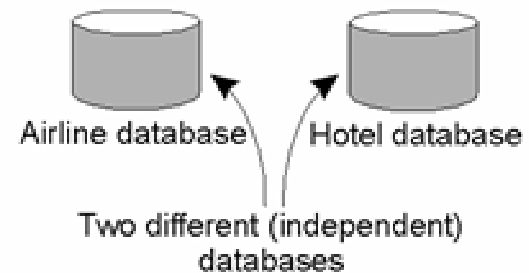
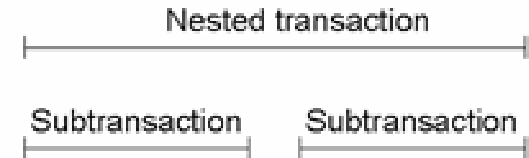
| Primitive         | Description                                     |
|-------------------|---|
| BEGIN_TRANSACTION | Make the start of a transaction                 |
| END_TRANSACTION   | Terminate the transaction and try to commit     |
| ABORT_TRANSACTION | Kill the transaction and restore the old values |
| READ              | Read data from a file, a table, or otherwise    |
| WRITE             | Write data to a file, a table, or otherwise     |

- All-or-nothing (ACID properties):
  - *Atomic*: to the outside world, the transaction happens indivisibly
  - *Consistent*: the transaction does not violate system invariants
  - *Isolated* (or serializable): concurrent transactions do not interfere with each other
  - *Durable*: once a transaction commits, the changes are permanent



# Transaction types

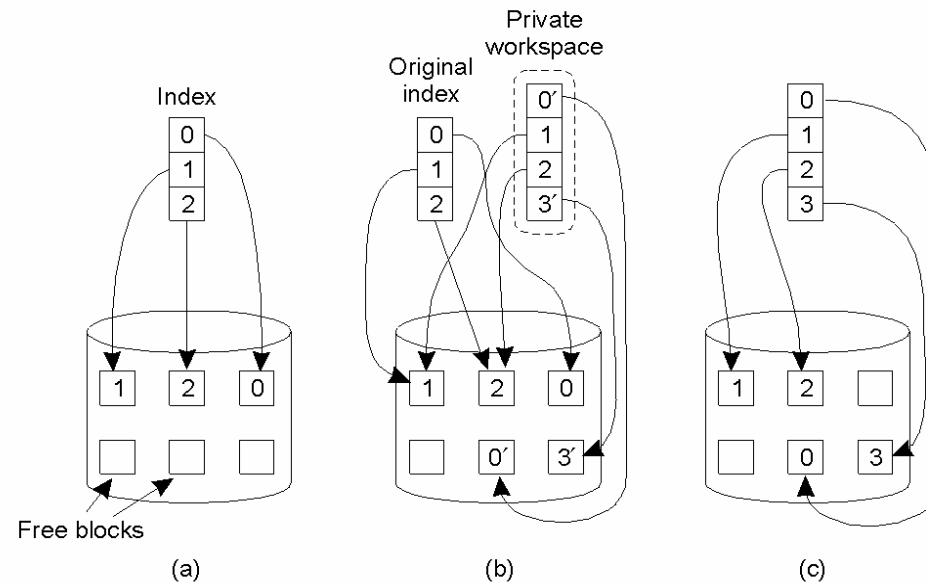
- *Flat*
  - ACID transactions as defined earlier
- *Nested*
  - Constructed from sub-transactions
  - Sub-transactions can be undone once committed (if their parent transaction aborts): durability applies only to top-level transactions
  - Sub-transactions conceptually operate on a private copy of the data:
    - If it aborts the private copy disappears
    - If it commits, the modified private copy is available to the next sub-transaction
  - Typically, each sub-transaction runs on a different host, providing a given service
- *Distributed*
  - Accounts for data distribution
  - Essentially flat transactions on distributed data
    - Instead, nested transactions are hierarchical
  - Need distributed locking





# Achieving atomicity

- Approach 1. Private workspace
  - Copy what the transaction modifies into a separate memory space, creating *shadow blocks* of the original file
  - If the transaction is aborted, this private workspace is deleted, otherwise they are copied into the parent's workspace
  - Optimize by replicating the index, and not the whole file
  - Works fine also for the local part of distributed transactions





# Achieving atomicity

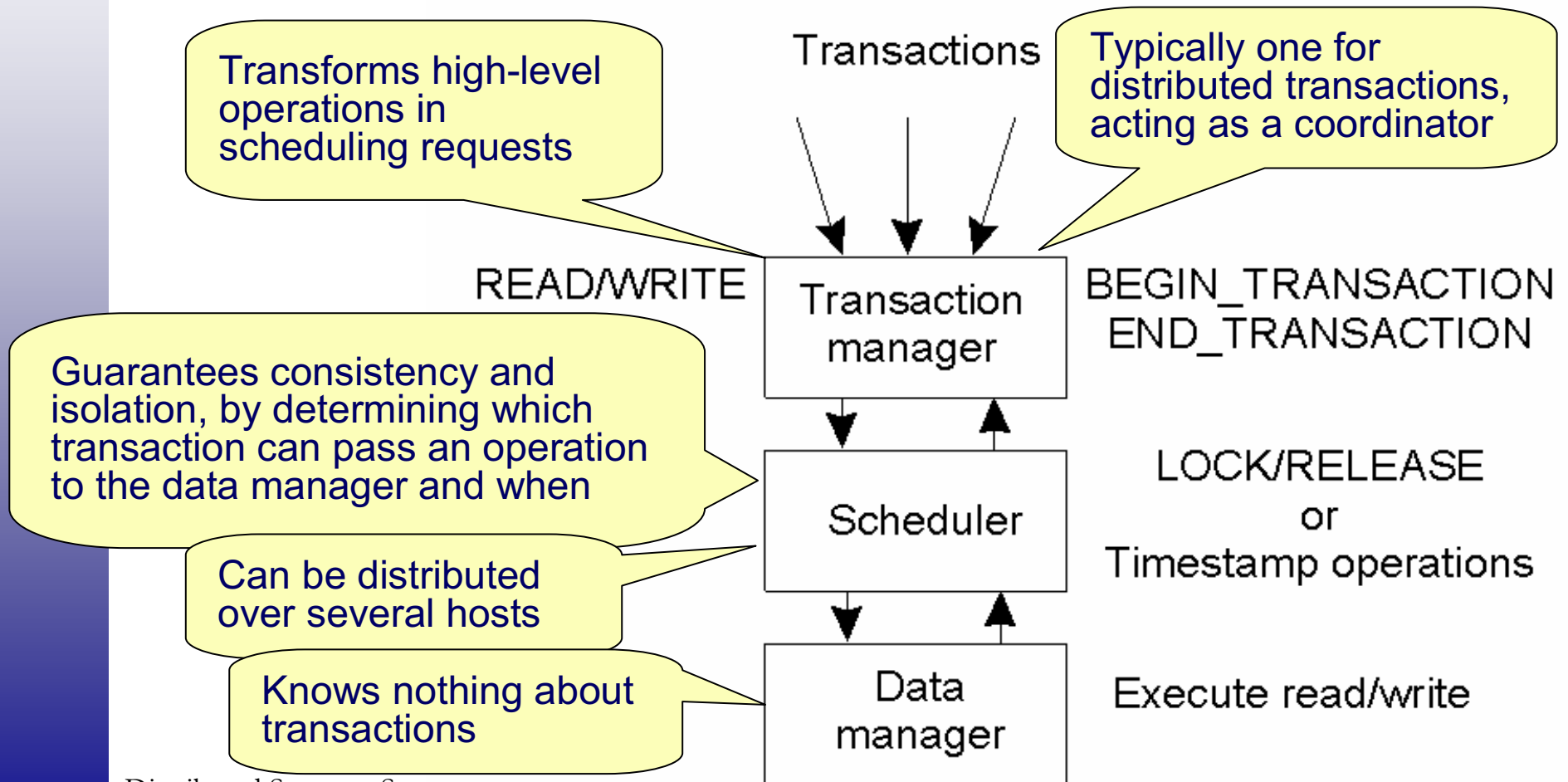
- Approach 2. Writeahead log
  - Files are modified in place (commit is fast), but a log is kept with
    - Transaction that made the change
    - Which file/block
    - Old and new values
  - After the log is written successfully, the file is actually modified
  - If transaction succeeds, commit written to log; if it aborts, original state restored based on logs, starting at the end (rollback)

|                    |           |           |           |
|--------------------|-----------|-----------|-----------|
| x = 0;             | Log       | Log       | Log       |
| y = 0;             |           |           |           |
| BEGIN_TRANSACTION; |           |           |           |
| x = x + 1;         | [x = 0/1] | [x = 0/1] | [x = 0/1] |
| y = y + 2          |           | [y = 0/2] | [y = 0/2] |
| x = y * y;         |           |           | [x = 1/4] |
| END_TRANSACTION;   |           |           |           |
| (a)                | (b)       | (c)       | (d)       |



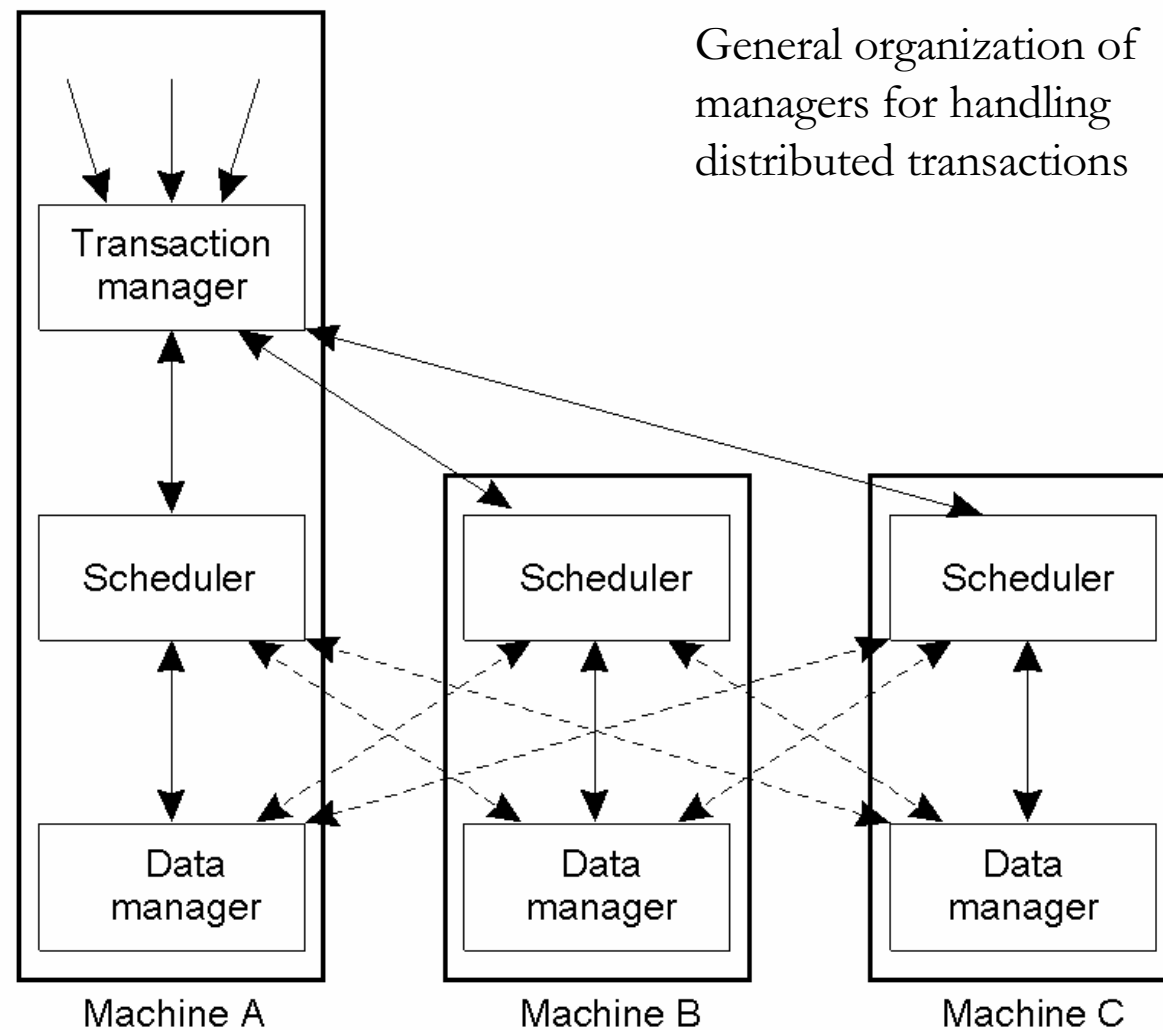
# Controlling concurrency

- Allow several transactions to be executed simultaneously, but safely (i.e., consistently)





# Controlling concurrency: Distributed scheduler







# Serializability

BEGIN\_TRANSACTION

$x = 0;$

$x = x + 1;$

END\_TRANSACTION

BEGIN\_TRANSACTION

$x = 0;$

$x = x + 2;$

END\_TRANSACTION

BEGIN\_TRANSACTION

$x = 0;$

$x = x + 3;$

END\_TRANSACTION

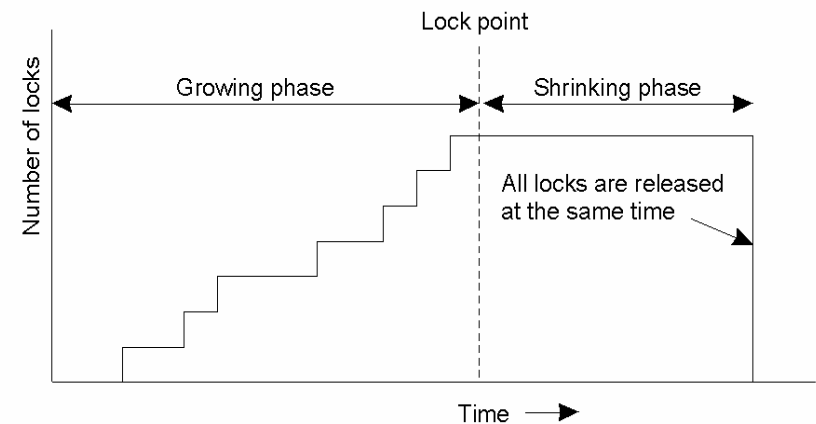
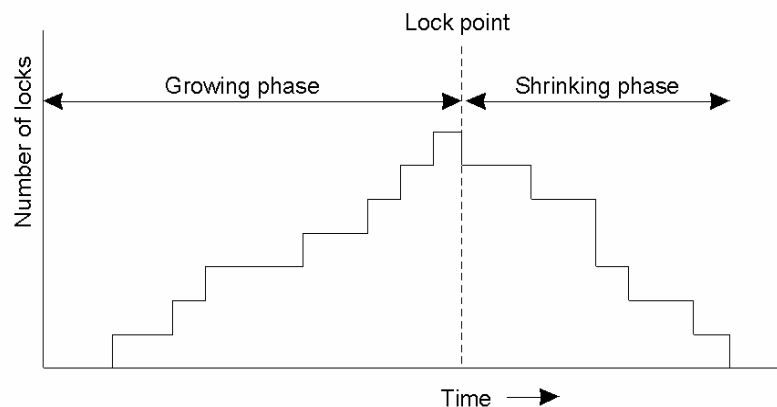
|            |   |         |
|------------|---|---------|
| Schedule 1 | $x = 0;$ $x = x + 1;$ $x = 0;$ $x = x + 2;$ $x = 0;$ $x = x + 3;$ | Legal   |
| Schedule 2 | $x = 0;$ $x = 0;$ $x = x + 1;$ $x = x + 2;$ $x = 0;$ $x = x + 3;$ | Legal   |
| Schedule 3 | $x = 0;$ $x = 0;$ $x = x + 1;$ $x = 0;$ $x = x + 2;$ $x = x + 3;$ | Illegal |

- Various interleavings are possible
  - Only the ones corresponding to some linearization of the involved transactions are legal
- Transaction systems must ensure operations are interleaved correctly, but also free the programmer from the burden of programming mutual exclusion
- Need to properly schedule conflicting operations
  - Read-write and write-write...but not read-read
- Mutual exclusion (locks) vs. explicit operation ordering
- Pessimistic vs. optimistic concurrency control



# Two-Phase Locking (2PL)

- When a process needs to access data it requests the scheduler to grant a lock
- Two-phase locking
  - The scheduler tests whether the requested operation conflicts with another that has already received the lock: if so, the operation is delayed
  - Once a lock for a transaction  $T$  has been released,  $T$  can no longer acquire it
  - *Strict 2PL*, i.e., releasing the locks all at the same time, prevents cascaded aborts by requiring the shrink phase to take place only after transaction termination
  - Proven that 2PL leads to serializability... but it may deadlock
  - Widely used





# Implementing 2PL

- Centralized 2PL
  - The transaction manager contacts a centralized lock manager, receives lock grant, interacts directly with the data manager, then returns the lock to the lock manager
- Primary 2PL
  - Multiple lock managers exist
  - Each data item has a primary copy on a host: The lock manager on that host is responsible for granting locks
  - The transaction manager is responsible for interacting with the data managers
- Distributed 2PL
  - Assume data may be replicated on multiple hosts
  - The lock manager on a host is responsible for granting locks on the local replica and for contacting the (local) data manager

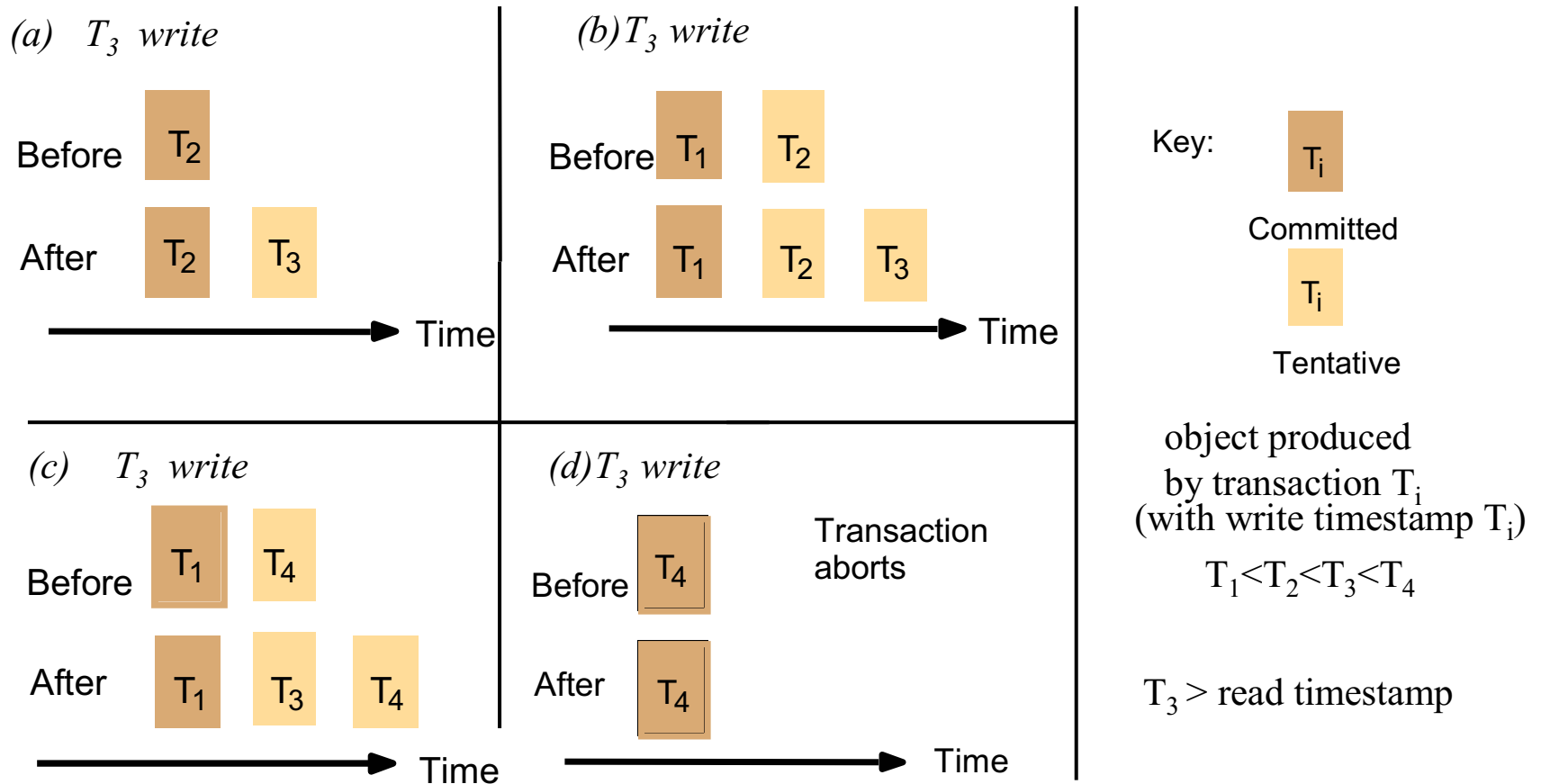


# Pessimistic timestamp ordering

- Assign a timestamp to each transaction (e.g., using logical clocks)
- Write operations on a data item  $x$  are recorded in tentative versions, each with its own write timestamp  $ts_{wr}(x_i)$ , until commit is performed
  - We refer to the write timestamp of the committed version of  $x$  as  $ts_{wr}(x)$
- Each data item  $x$  has also a read timestamp  $ts_{rd}(x)$ : That of the last transaction which read  $x$
- The scheduler operates as follow:
  - When receives  $write(T, x)$  at  $time=ts$ 
    - If  $ts > ts_{rd}(x)$  and  $ts > ts_{wr}(x)$  perform tentative write  $x_i$  with timestamp  $ts_{wr}(x_i)$
    - else abort  $T$  since the write request arrived too late
  - Scheduler receives  $read(T, x)$  at  $time=ts$ 
    - If  $ts > ts_{wr}(x)$ 
      - Let  $x_{sel}$  be the latest version of  $x$  with the write timestamp lower than  $ts$
      - If  $x_{sel}$  is committed perform read on  $x_{sel}$  and set  $ts_{rd}(x) = \max(ts, ts_{rd}(x))$
      - else wait until the transaction that wrote version  $x_{sel}$  commits or abort then reapply the rule
    - else abort  $T$  since the read request arrived too late
- Aborted transactions will reapply for a new timestamp and simply retry
- Deadlock-free

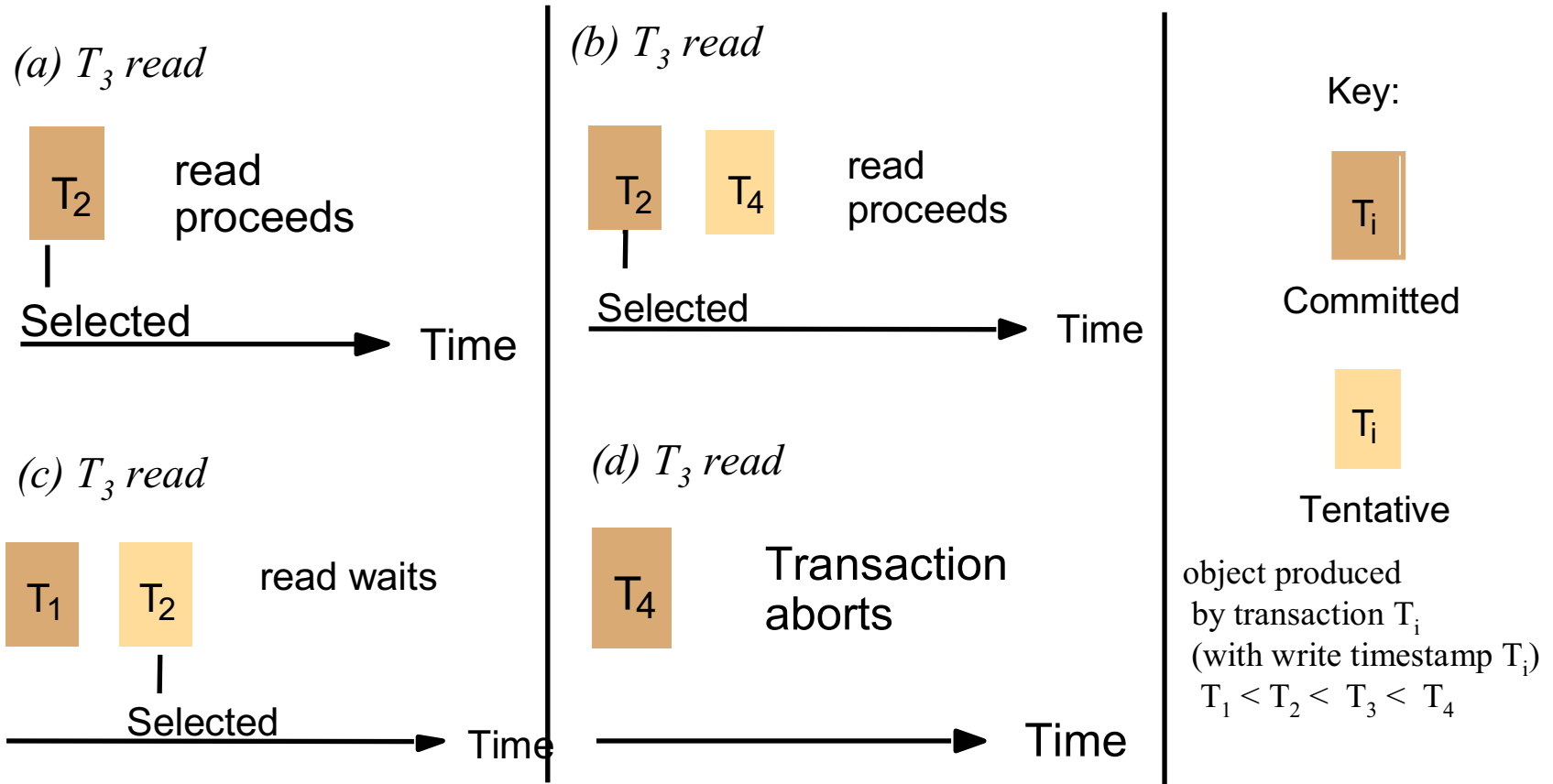


# Write operations and timestamps





# Read operations and timestamps





# Optimistic timestamp ordering

- Based on the assumption that conflicts are rare
- Therefore: do what you want without caring about others, fix conflicts later
  - Stamp data items with start time of transaction
  - At commit, if any items have been changed since start, transaction is aborted, otherwise committed
  - Best implemented with private workspaces
- Deadlock-free, allows maximum parallelism
- Under heavy load, there may be too many rollbacks
- Not widely used, especially in distributed systems



# Distributed deadlocks

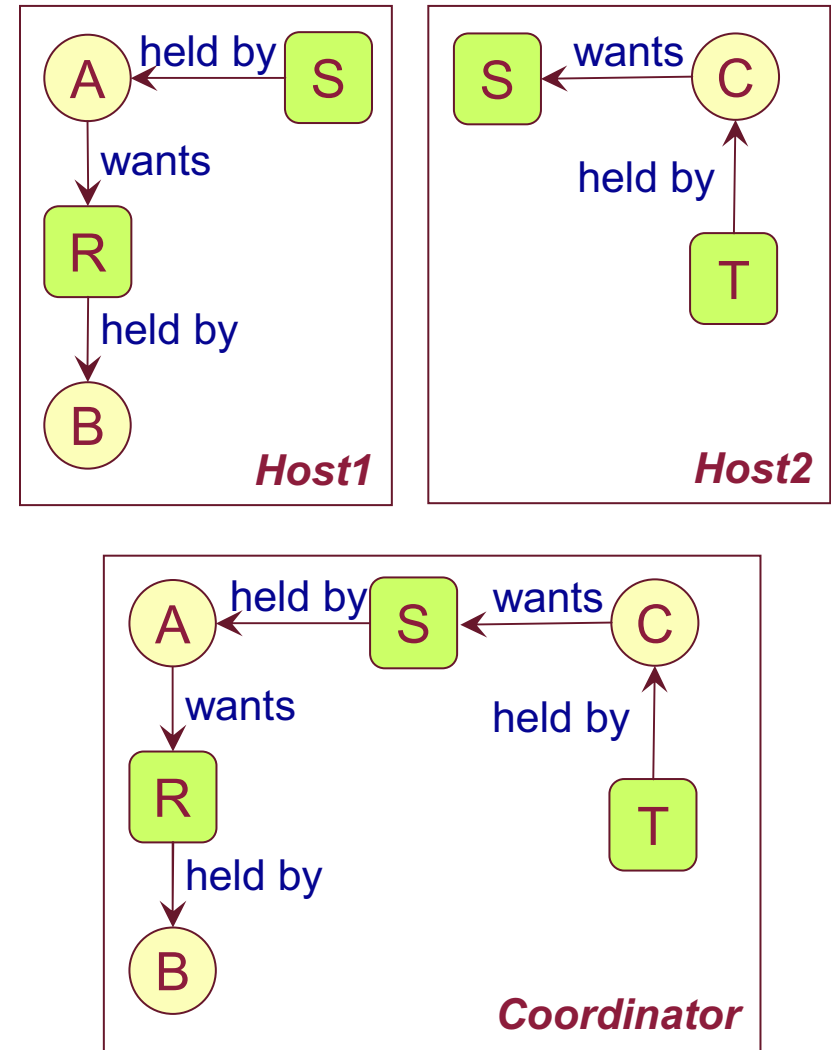
- Same concept as in conventional systems
  - But worse to deal with, since in a distributed system resources are spread out
- Approaches
  - Ignore the problem
    - Most often employed, actually meaningful in many settings
  - Detection
    - And recovery: typically by killing one of the processes
  - Prevention
  - Avoidance
    - Never used in (distributed) systems, as it implies a priori knowledge about resource usage
- Distributed transactions are helpful
  - To abort a transaction (and perform a rollback) is less disruptive than killing a process





# Centralized deadlock detection

- Each machine maintains a resource graph for its own resources and processes, and reports it to a coordinator
- Options for collecting this information:
  - Whenever an arc is added or deleted, a message is sent to the coordinator with the update
  - Periodically, every process sends a list of arcs added or deleted since the last update
  - Coordinator can request information on-demand
- None works well, because of false deadlocks
  - For instance, if B releases R and acquires T and the coordinator receives data from host 1 before receiving data from host 2

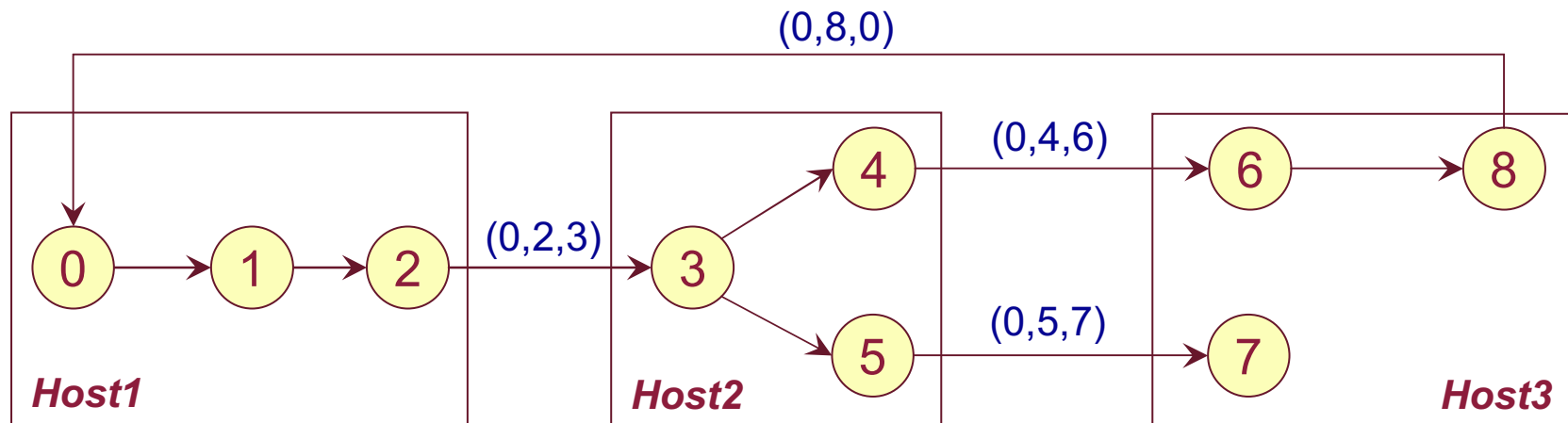




# Distributed deadlock detection

## Chandy-Misra-Haas (1983)

- There is no coordinator in charge of building the global wait-for graph
- Processes are allowed to request multiple resources simultaneously
- When a process gets blocked, it sends a probe message to the processes holding resources it wants to acquire
  - Probe message: (initiator, sender, receiver)
  - A cycle is detected if the probe makes it back to the initiator





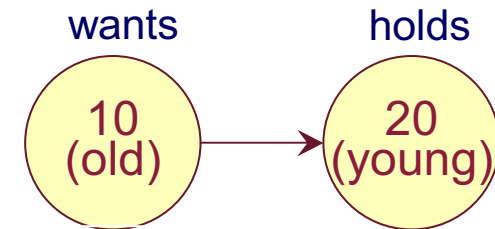
# Distributed detection in practice

- How to recover when a deadlock is detected?
  - Initiator commits suicide
    - Many processes may be unnecessarily aborted if more than one initiator detects the loop
  - Alternative
    - The initiator picks the process with the higher identifier and kills it
    - Requires each process to add its identifier to the probe
- In practice: 90% of all deadlock cycles involve just 2 processes [Gray, 1981]
  - At least in databases

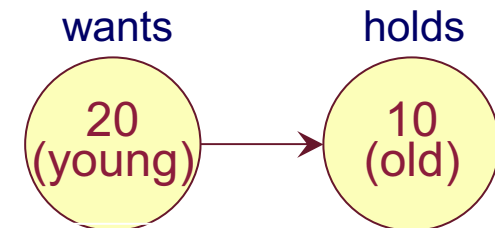


# Distributed prevention

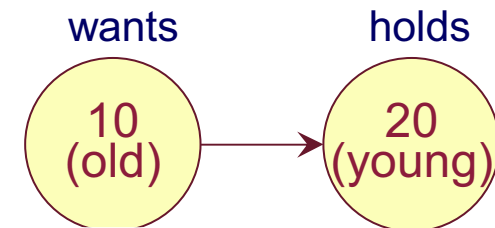
- Make deadlocks impossible by design
- For instance, using global timestamps (wait-die algorithm):
  - When a process A is about to block for a resource that another process B is using, allow A to wait only if A has a lower timestamp (it is older) than B; otherwise kill the process A
  - Following a chain of waiting processes, the timestamps will always increase (no cycles)
  - In principle, could have the younger wait
- If a process can be preempted, i.e., its resource taken away, an alternative can be devised (wound-wait algorithm):
  - Preempting the young process aborts its transaction, and it may immediately try to reacquire the resource, but will wait
  - In wait-die, young process may die many times before the old one releases the resource: here, this is not the case



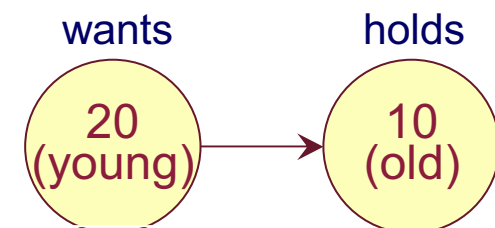
**waits**



**dies**



**preempts**



**waits**