# Module 4
# Synchronization

# Synchronization Problem

- How processes cooperate and synchronize with one another in a distributed system
  - ➡ In single CPU systems, critical regions, mutual exclusion, and other synchronization problems are solved using methods such as semaphores.
  - ➡ These methods will not work in distributed systems because they implicitly rely on the existence of shared memory.
  - ➡ Examples:
    - ✦ Two processes interacting using a semaphore must both be able to access the semaphore. In a centralized system, the semaphore is stored in the kernel and accessed by the processes using system calls
    - ✦ If two events occur in a distributed system, it is difficult to determine which event occurred first.
- How to decide on relative ordering of events
  - ➡ Does one event precede another event?
  - ➡ Difficult to determine if events occur on different machines.

# Issues

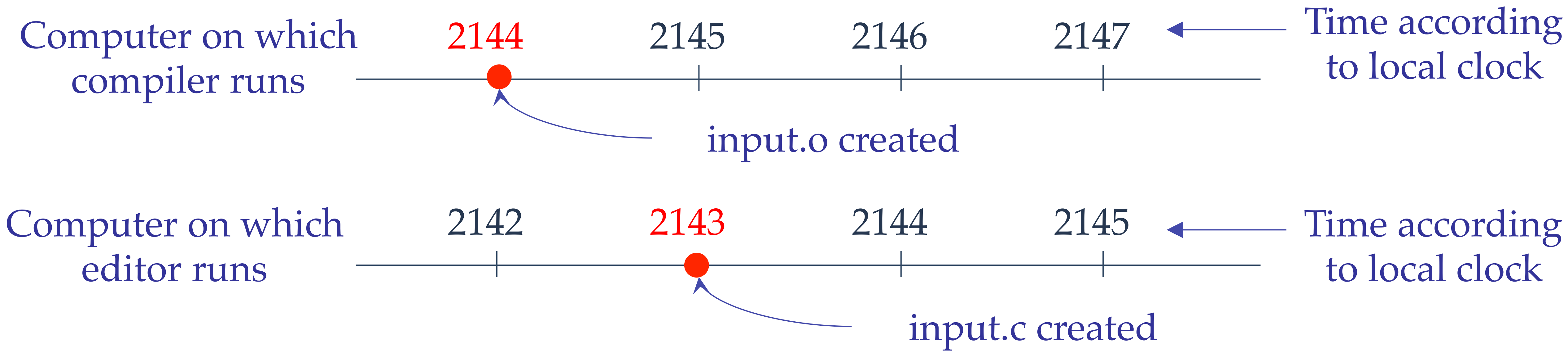
- Part 1 - Clocks
  - How to synchronize events based on actual time?
    - Clock synchronization
  - How to determine relative ordering?
    - Logical clocks
- Part 2 - Global State and Election
  - What is the “global state” of a distributed system?
  - How do we determine the “coordinator” of a distributed system?
- Part 3 - How do we synchronize for sharing
  - Mutual exclusion
  - Distributed transactions

# Clock synchronization

- In a centralized system:
  - ➡ Time is unambiguous: A process gets the time by issuing a system call to the kernel. If process *A* gets the time and latter process *B* gets the time. The value *B* gets is higher than (or possibly equal to) the value *A* got
  - ➡ Example: UNIX `make` examines the times at which all the source and object files were last modified:
    - ✦ If time (`input.c`) > time(`input.o`) then recompile input.c
    - ✦ If time (`input.c`) < time(`input.o`) then no compilation is needed

# Clock synchronization (2)

- In a distributed system:
  - ➡ Achieving agreement on time is not trivial!


Computer on which compiler runs
2144
2145
2146
2147
Time according to local clock
input.o created
Computer on which editor runs
2142
2143
2144
2145
Time according to local clock
input.c created

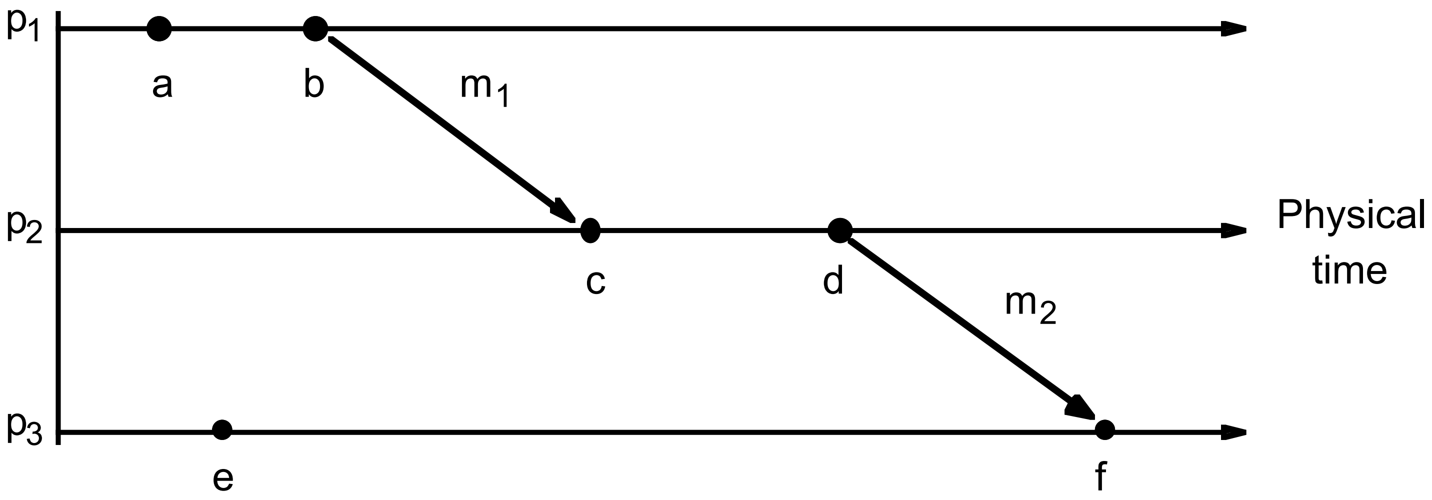
■ Is it possible to synchronize all the clocks in a distributed system?

# Logical vs Physical Clocks

- Clock synchronization need not be absolute! (due to Lamport, 1978):
  - ➡ If two processes do not interact, their clocks need not be synchronized.
  - ➡ What matters is not that all processes agree on exactly what time is it, but rather, that they agree on the order in which events occur.
  - ➡ We will discuss this later under "logical clock synchronization".
- For algorithms where only internal consistency of clocks matters (not whether clocks are close to real time), we speak of logical clocks.
- For algorithms where clocks must not only be the same, but also must not deviate from real-time, we speak of physical clocks.

# Logical Clock Synchronization

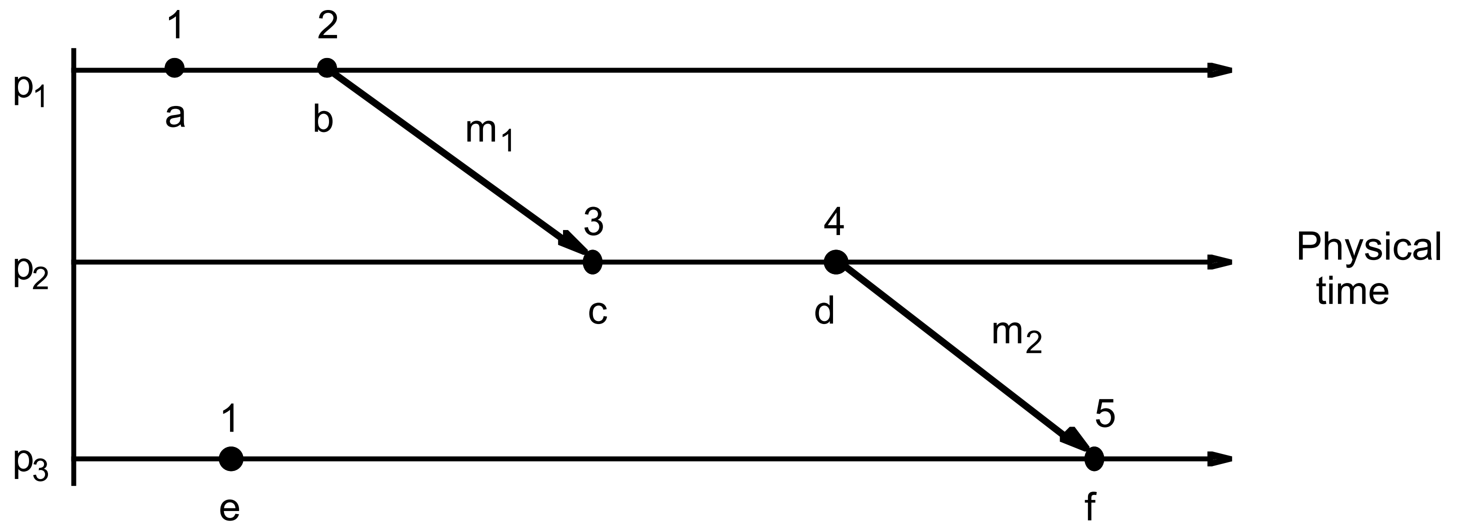
- Happens-before relation
  - ➡ If $a$ and $b$ are events in the same process, and $a$ happens-before $b$, then $a \rightarrow b$ is true
  - ➡ If $a$ is the event of a message being sent by one process, and $b$ is the event of the same message being received by another process, then $a \rightarrow b$ is also true
  - ➡ If two events, $a$ and $b$, happen in different processes that do not exchange messages , then $a \rightarrow b$ is *not* true, but neither is $b \rightarrow a$. These events are said to be concurrent ($a \mid\mid b$).
  - ➡ happens-before is transitive: $a \rightarrow b$ and $b \rightarrow c \Rightarrow a \rightarrow c$



From Coulouris, Dollimore and Kindberg, Distributed Systems: Concepts and Design, 3rd ed.
© Addison-Wesley Publishers 2000

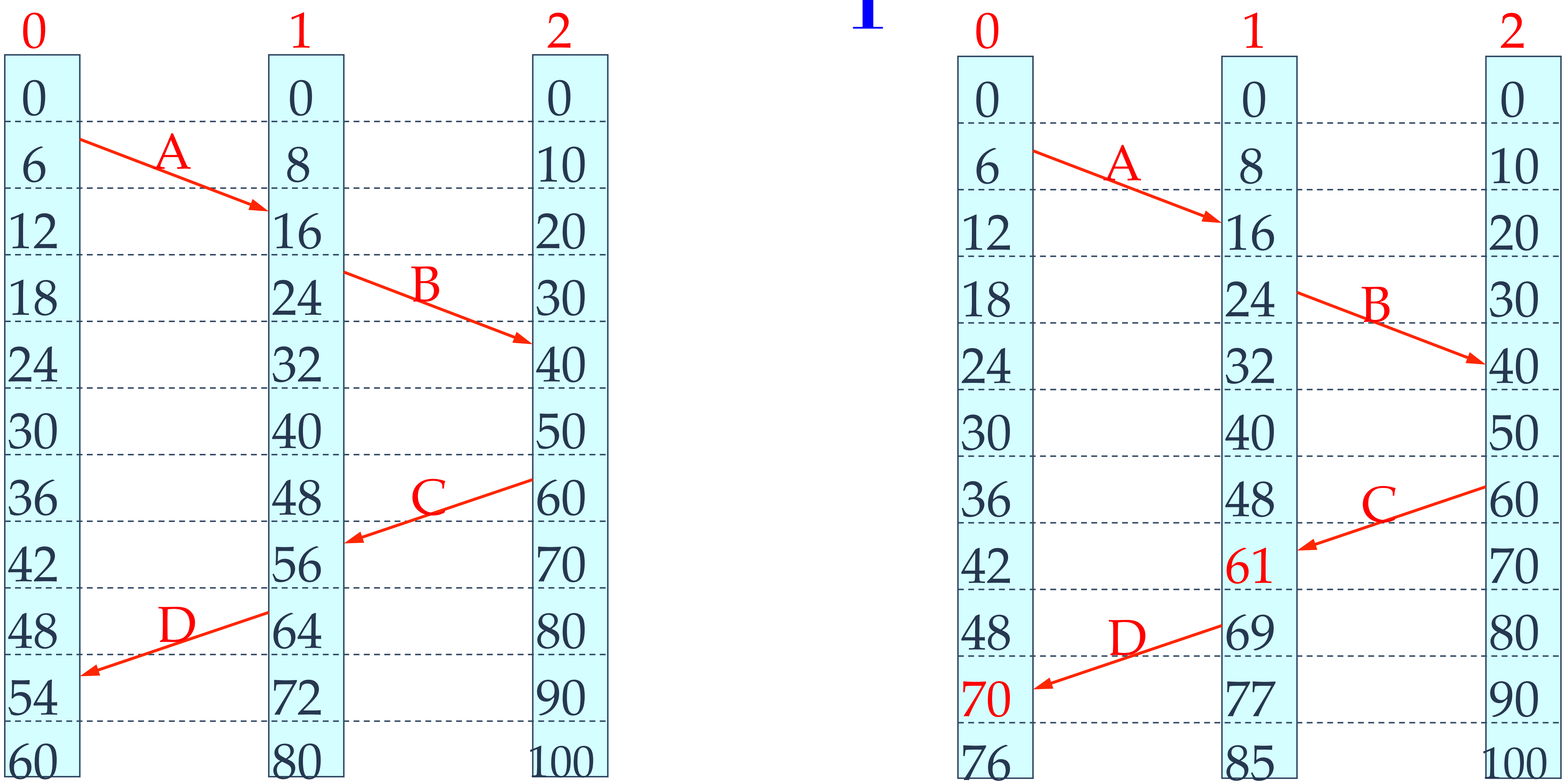
# Lamport's Algorithm

- Capturing happens-before relation
- Each process $p_i$ has a local monotonically increasing counter, called its logical clock $L_i$
- Each event $e$ that occurs at process $p_i$ is assigned a Lamport timestamp $L_i(e)$
- Rules:
  - ➡ $L_i$ is incremented before event $e$ is issued at $p_i$: $L_i := L_i + 1$
  - ➡ When $p_i$ sends message $m$, it adds $t = L_i$: $(m, t)$ [this is event $send(m)$]
  - ➡ On receiving $(m, t)$, $p_j$ computes $L_j := \max(L_j, t)$; $L_j := L_j + 1$; timestamp event $receive(m)$



From Coulouris, Dollimore and Kindberg, Distributed Systems: Concepts and Design, 3rd ed.

# Example



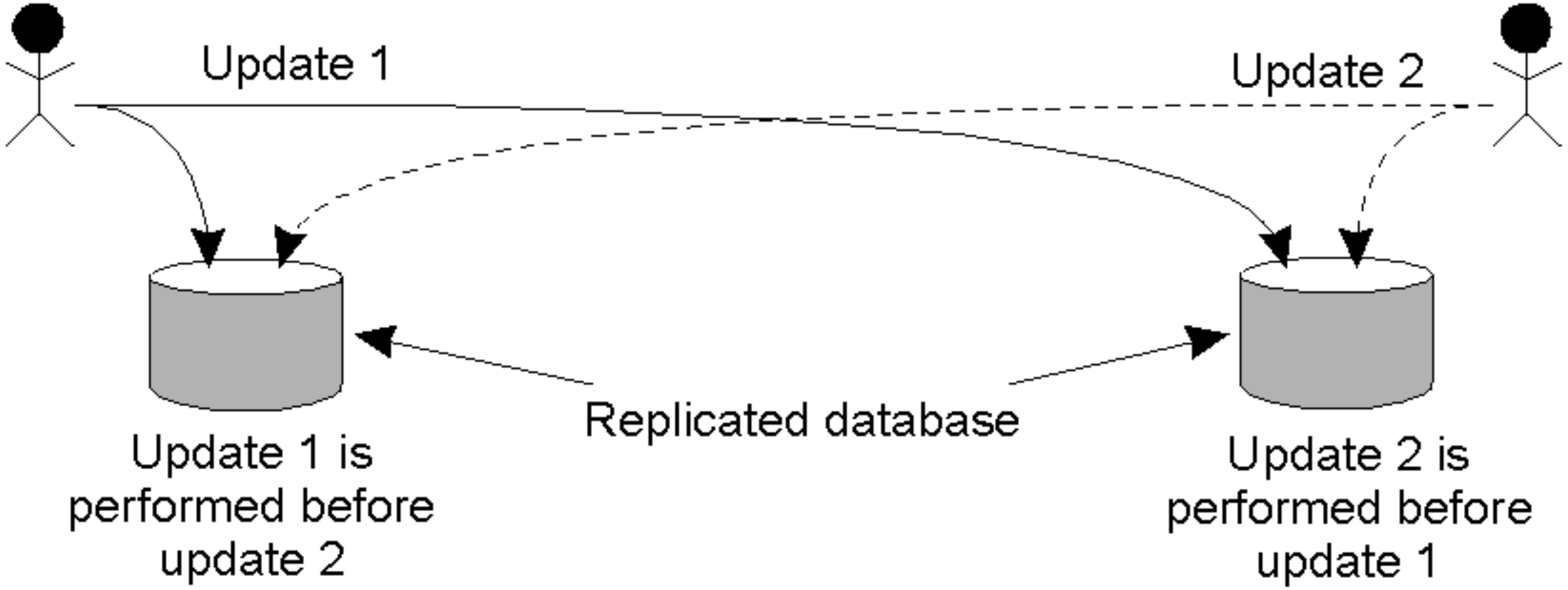
a) Three processes, each with its own clock. The clocks run at different rates.

b) Lamport's algorithm corrects the clocks.

- Lamport solution:
  - ➡ Between every two events, the clock must tick at least once
  - ➡ No two events occur at exactly the same time. If two events happen in processes 1 and 2, both with time 40, the former becomes 40.1 and the latter becomes 40.2

From Tanenbaum and van Steen, Distributed Systems: Principles and Paradigms, 2nd edition

# Lamport Timestamps


Update 1
Update 2
Replicated database
Update 1 is performed before update 2
Update 2 is performed before update 1

a) The updates have to be ordered the same way across the replicas.

b) This can be achieved by a totally ordered multicast algorithm.

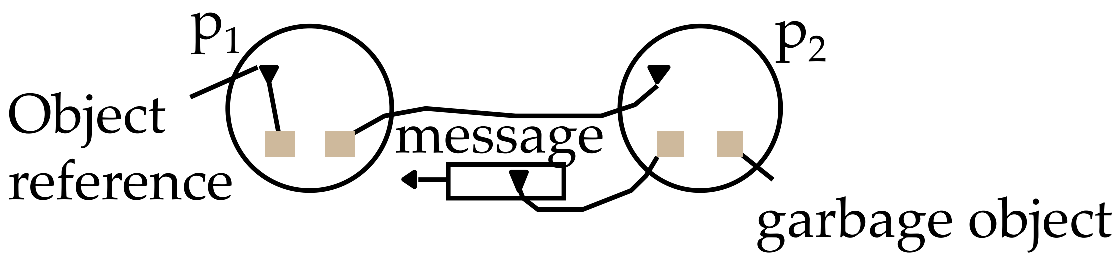
c) Totally ordered multicasting can be achieved by means of Lamport clocks.

From Tanenbaum and van Steen, Distributed Systems: Principles and Paradigms, 2nd edition
© Prentice-Hall, Inc. 2007

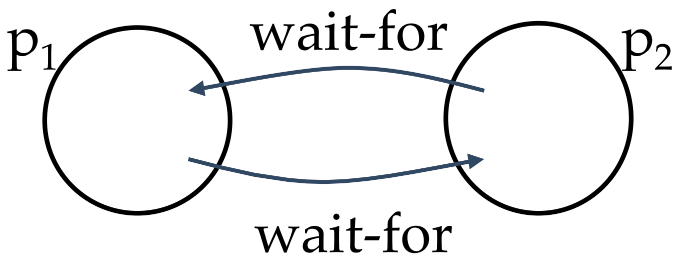
# Part 2
# Global State & Election Algorithms

# Global State

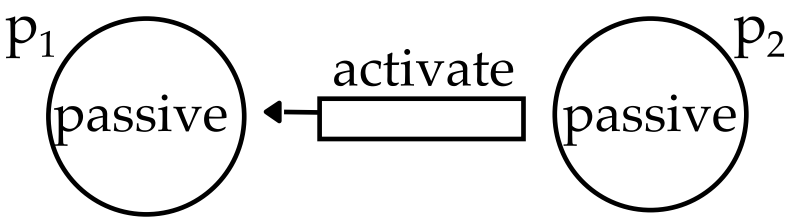
- Sometimes it is useful to know the global state of a distributed system
  - Garbage collection



  - Deadlocks



  - Termination

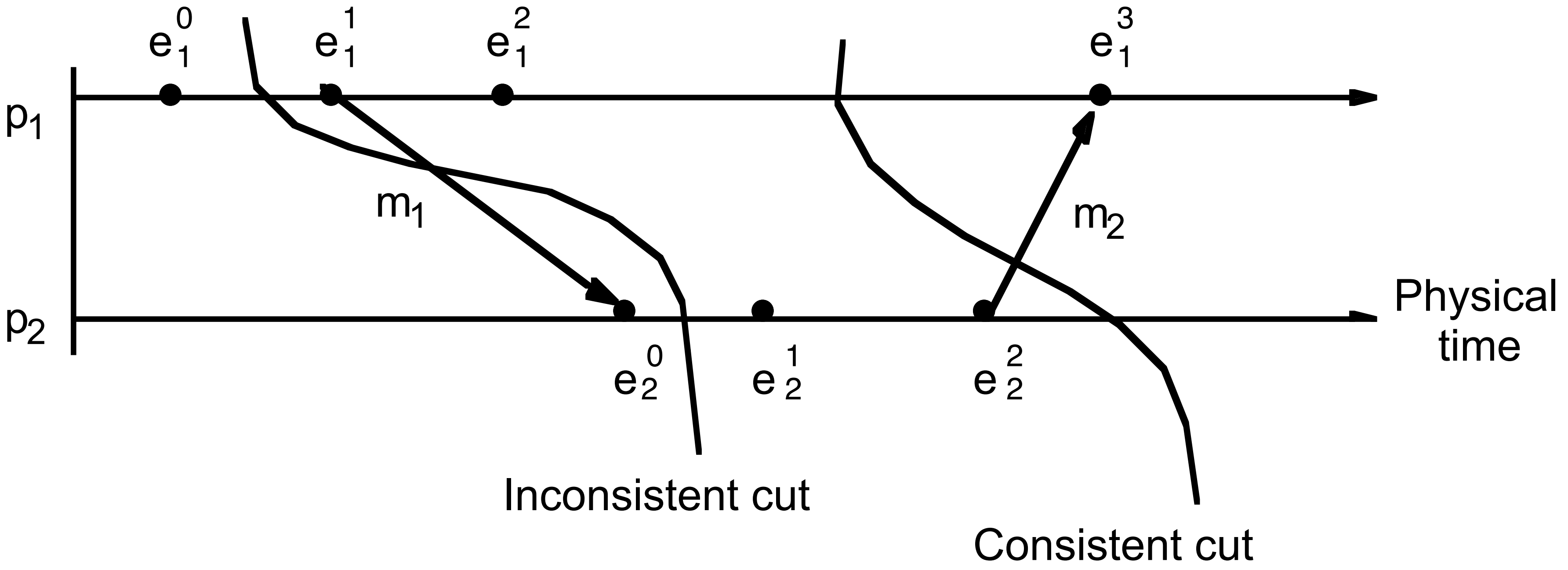


- Global state = local state of each process + msg's in transit
  - Local state depends on the process

From Coulouris, Dollimore and Kindberg, Distributed Systems: Concepts and Design, 3rd ed.

# Distributed Snapshot

- Represents a state in which the distributed system might have been.
  - ➡ Consistent global state: If *A* sends a message to *B* and and the system records *B* receiving the message, it should also record *A* sending it.



- Cut of the system's execution: subset of its global history identifying set of events *e*.
  - ➡ Cut *C* is consistent iff $\forall e \in C, f \rightarrow e \Rightarrow f \in C$

From Coulouris, Dollimore and Kindberg, Distributed Systems: Concepts and Design, 3rd ed.
© Addison-Wesley Publishers 2000

# Election Problem

- Many algorithms require one process to act as a coordinator
    - Example: Coordinator in the centralized mutual exclusion algorithm
- In general, it does not matter which process takes on this special responsibility, but one has to do it.
- How to elect a coordinator?

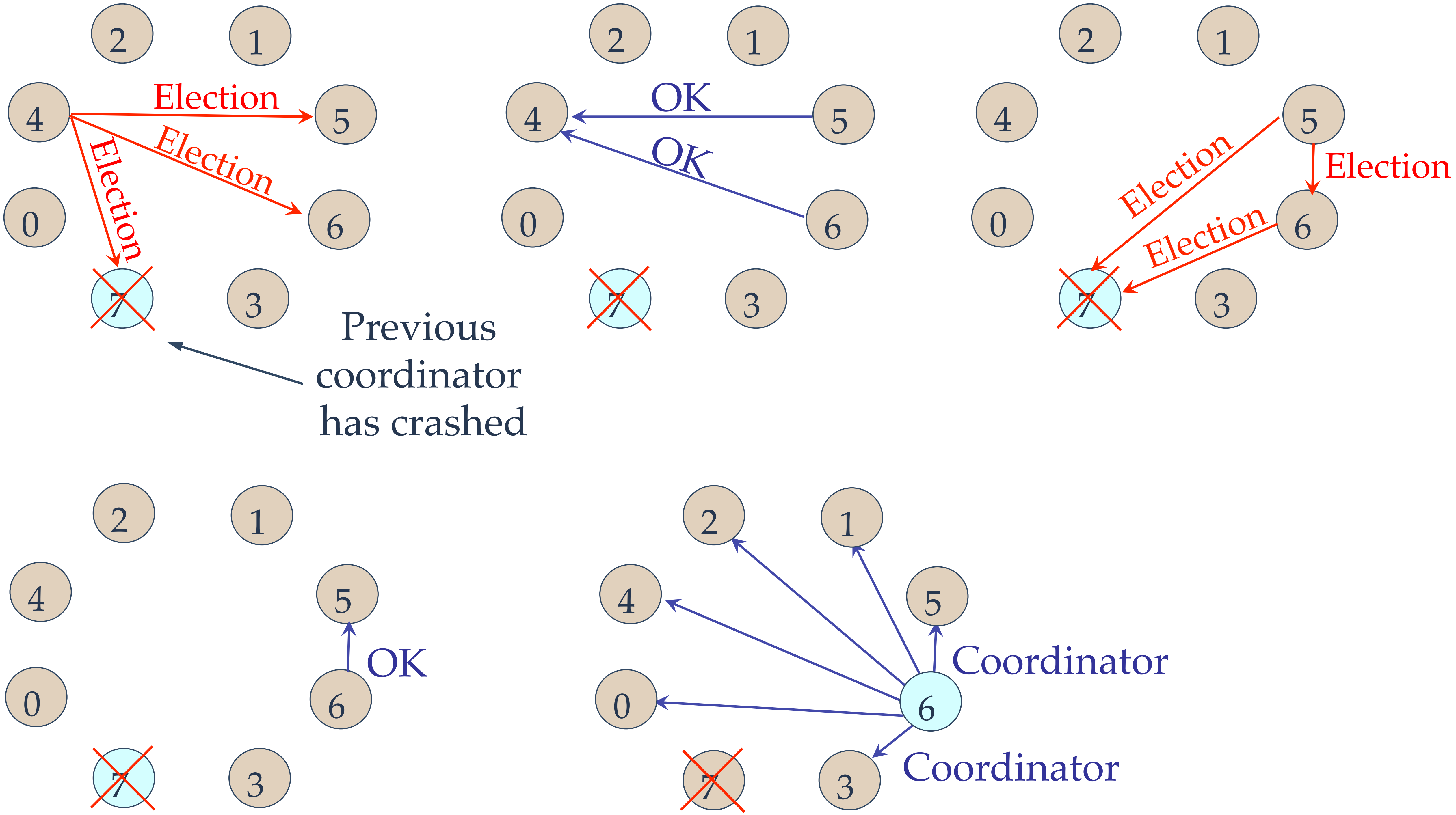
# General Approach

- Assumptions:
  - ➡ Each process has a unique number (e.g., its network address) to distinguish them and hence to select one
  - ➡ One process per machine: For simplicity
  - ➡ Every process knows the process number of every other process
  - ➡ Processes do not know which processes are currently up and which ones are currently down
- Approach:
  - ➡ Locate the process with the highest process number and designate it as coordinator.
  - ➡ Election algorithms differ in the way they do the location

# The Bully Algorithm

- When a process, P, notices that the coordinator is no longer responding to requests, it initiates an election:
  - ➡ P sends an ELECTION message to all processes with higher numbers
  - ➡ If no one responds, P wins the election and becomes coordinator
  - ➡ If one of the higher-ups answers, it takes over. P's job is done
- When a process gets an ELECTION message from one of its lower-numbered colleagues:
  - ➡ Receiver sends an OK message back to the sender to indicate that he is alive and will take over
  - ➡ Receiver holds an election, unless it is already holding one
  - ➡ Eventually, all processes give up but one, and that one is the new coordinator.
  - ➡ New coordinator announces its victory by sending all processes a message telling them that starting immediately it is the new coordinator
- If a process that was previously down comes back:
  - ➡ It holds an election. If it happens to be the highest process currently running, it will win the election and take over the coordinator's job
- The biggest guy in town always wins!

# Example



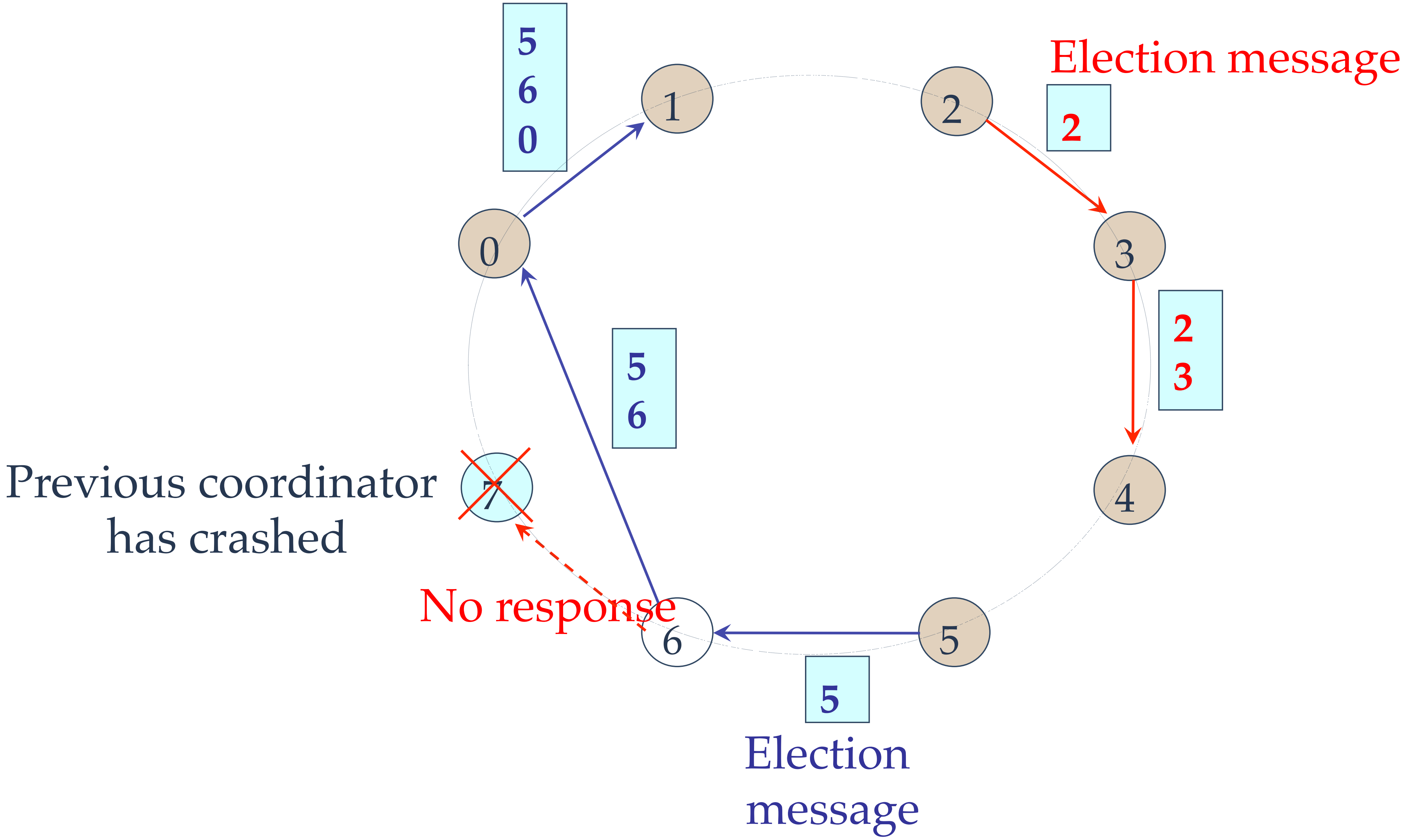
Previous coordinator has crashed

From Tanenbaum and van Steen, Distributed Systems: Principles and Paradigms, 2nd edition
© Prentice-Hall, Inc. 2002

# A Ring Algorithm

- Use a ring (processes are physically or logically ordered, so that each process knows who its successor is). No token is used.
- Algorithm:
  - When any process notices that coordinator is not functioning:
    - Builds an ELECTION message (containing its own process number)
    - Sends the message to its successor (if successor is down, sender skips over it and goes to next member along the ring, or the one after that, until a running process is located)
    - At each step, sender adds its own process number to the list in the message
  - When the message gets back to the process that started it all:
    - Process recognizes the message containing its own process number
    - Changes message type to COORDINATOR
    - Circulates message once again to inform everyone else: Who the new coordinator is (list member with highest number); Who the members of the new ring are
    - When message has circulated once, it is removed

# Example



From Tanenbaum and van Steen, Distributed Systems: Principles and Paradigms, 2nd edition

# Part 3
# Sharing in Distributed Systems

# Mutual Exclusion Problem

- Systems involving multiple processes are often programmed using critical regions.
- When a process has to read or update certain shared data structure, it first enters a critical region to achieve mutual exclusion, i.e., ensure that no other process will use the shared data structure at the same time
- In single-processor systems, critical regions are protected using semaphores, and monitors or similar constructs. How to accomplish the same in distributed systems?
  - ➡ We assume, for now, that transactions are not supported by the system

# Protocols & Requirements

- Application-level protocol

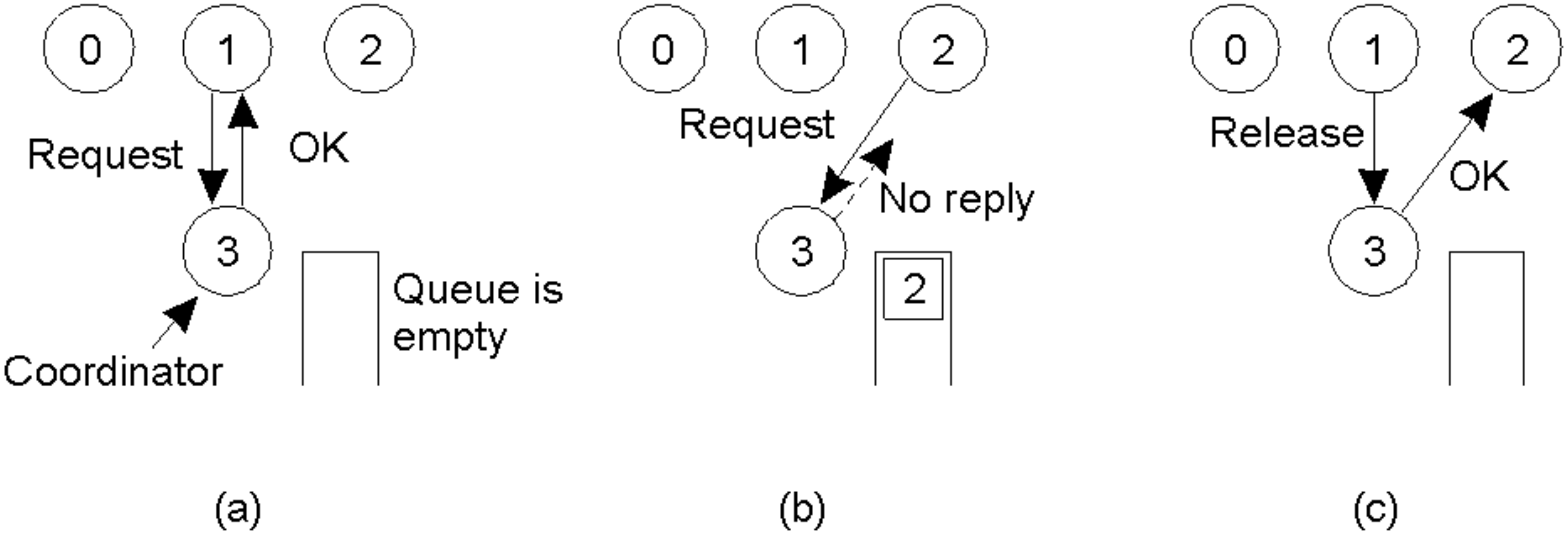
  *enter*() //enter critical section - block if necessary

  *resourceAccesses*() // access shared resources in critical section

  *exit*() // leave critical section - other processes may enter

- Requirements
  1. At most one process may execute in the critical section (safety)
  2. Requests to enter and exit the critical section eventually succeed (liveness)
  3. If one request to enter the critical section happened-before another, then entry to the critical section is granted in that order (→ order)

# A Centralized Algorithm (Central Server Algorithm)



One process is elected as the coordinator:

a) Process 1 asks the coordinator for permission to enter a critical region. Permission is granted

b) Process 2 then asks permission to enter the same critical region. The coordinator does not reply.

c) When process 1 exits the critical region, it tells the coordinator, when then replies to 2

From Tanenbaum and van Steen, Distributed Systems: Principles and Paradigms, 2nd edition
© Prentice-Hall, Inc. 2002

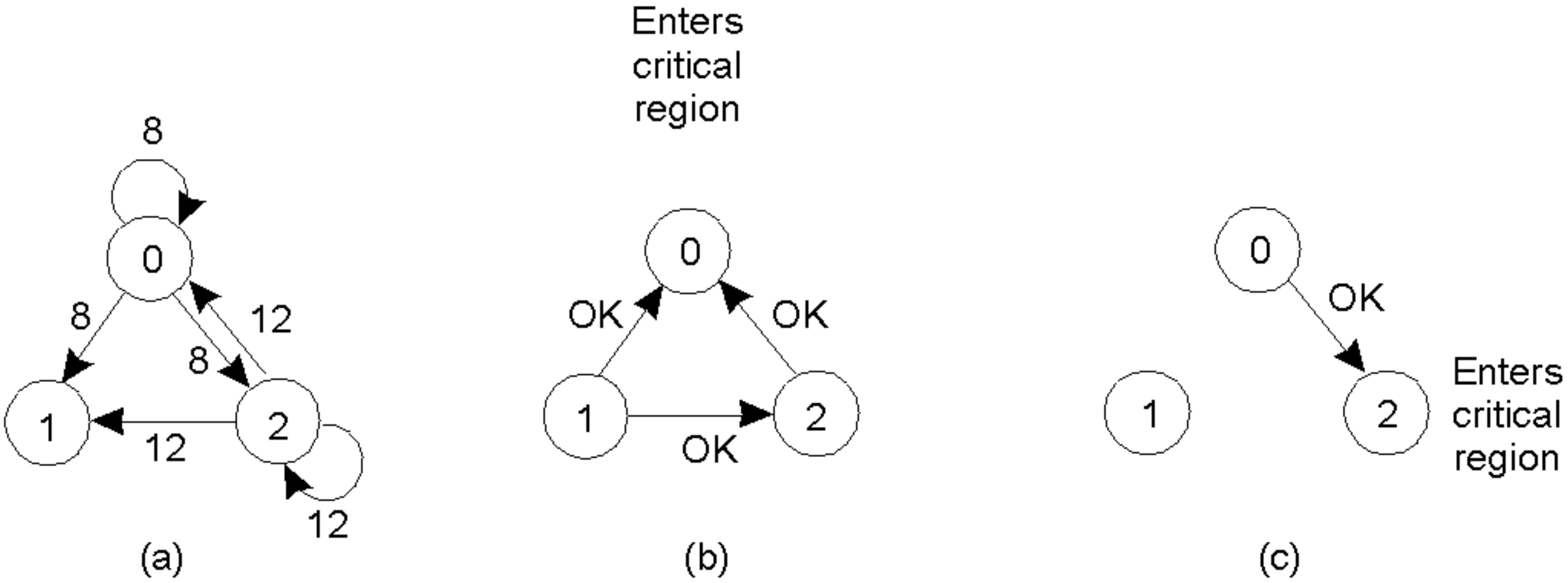
# A Centralized Algorithm (2)

- Advantages
  - ➡ Obviously it guarantees mutual exclusion
  - ➡ It is fair (requests are granted in the order in which they are received)
  - ➡ No starvation (no process ever waits forever)
  - ➡ Easy to implement (only 3 messages: request, grant and release)
- Shortcomings
  - ➡ Coordinator: A single point of failure; A performance bottleneck
  - ➡ If processes normally block after making a request, they cannot distinguish a dead coordinator from “permission denied”

# A Distributed Algorithm

- Based on a total ordering of events in a system (happens-before relation)
- Algorithm:
  - When a process wants to enter a critical region:
    - Builds a message: {name of critical region; process number; current time}
    - Sends the message to all other processes (assuming reliable transfer)
  - When a process receives a request message from another process:
    - If the receiver is not in the critical region and does not want to enter it, it sends back an OK message to the sender
    - If the receiver is already in the critical region, it does not reply. Instead it queues the request
    - If the receiver wants to enter the critical region but has not yet done so, it compares the timestamp with the one contained in the message it has sent everyone: If the incoming message is lower, the receiver sends back an OK message; otherwise the receiver queues the request and sends nothing

# Example



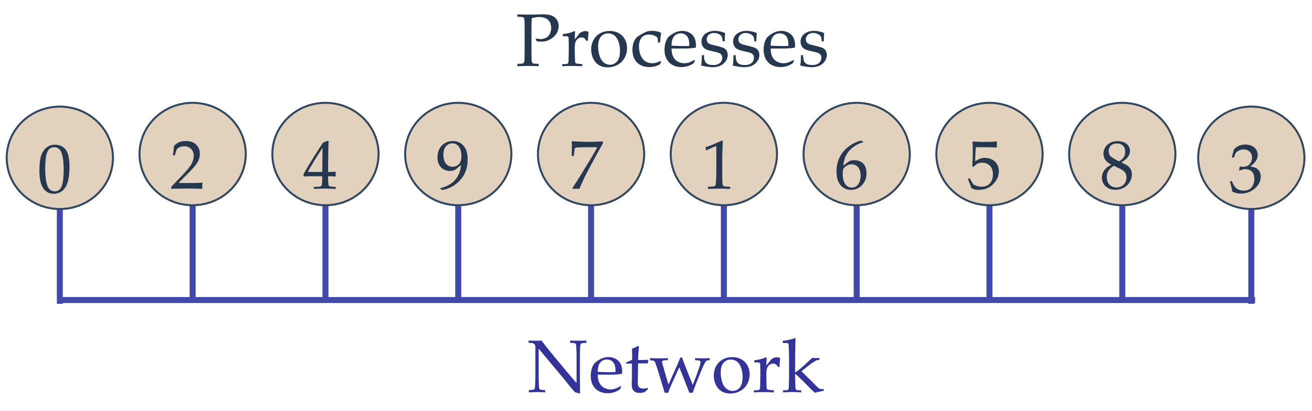
a) Two processes want to enter the same critical region at the same moment.

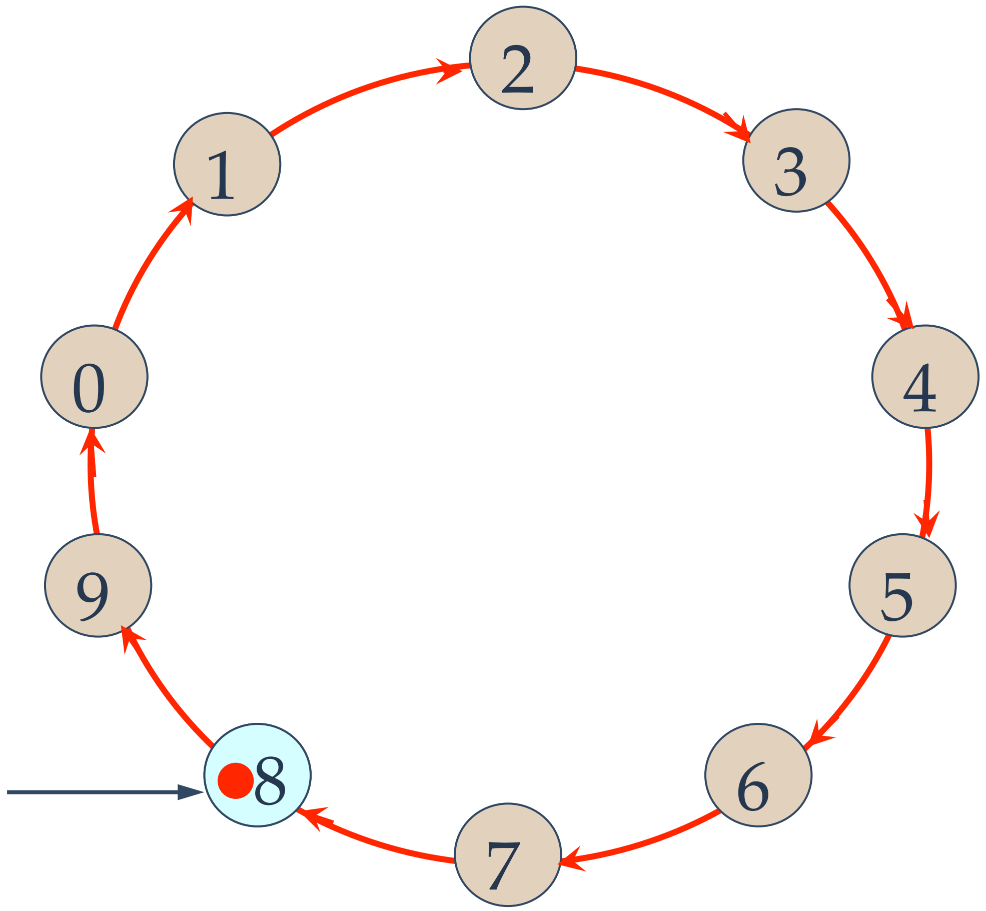
b) Process 0 has the lowest timestamp, so it wins.

c) When process 0 is done, it sends an OK also, so 2 can now enter the critical region.

From Tanenbaum and van Steen, Distributed Systems: Principles and Paradigms, 2nd edition

# A Token Ring Algorithm





Token holder may enter critical region or pass the token

From Tanenbaum and van Steen, Distributed Systems: Principles and Paradigms, 2nd edition
© Prentice-Hall, Inc. 2007

# A Token Ring Algorithm (2)

- Pros:
  - ➡ Guarantees mutual exclusion (only 1 process has the token at any instant)
  - ➡ No starvation (token circulates among processes in a well-defined order)
- Cons:
  - ➡ Regeneration of the token if it is ever lost: Detecting that the token is lost is difficult, since the amount of time between successive appearances of the token on the network is unbounded
  - ➡ Recovery if a process crashes: It is easier than in other cases though. If we require a process receiving the token to acknowledge receipt. A dead process is detected when its neighbor tries to give the token and fails. Dead process is then removed and the token handed to the next process down the line.

# Comparison of the 3 Algorithms

| **Algorithm** | **Messages per entry/exit** | **Delay before entry (in message times)** | **Problems** |
|---|---|---|---|
| Centralized | 3 | 2 | Coordinator crash |
| Distributed | 2 ( n – 1 ) | 2 ( n – 1 ) | Crash of any process |
| Token ring | 1 to ∞ | 0 to n – 1 | Lost token, process crash |

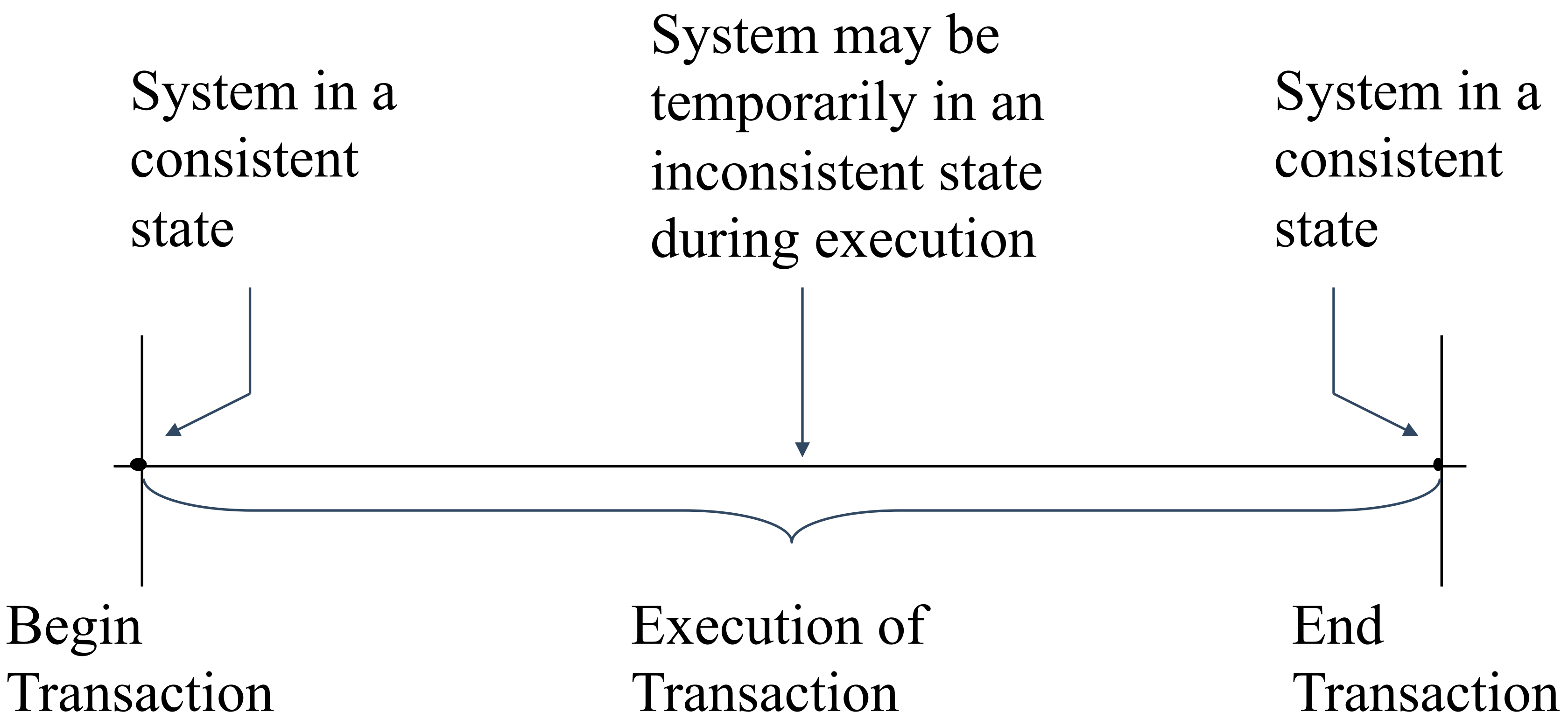
- Centralized Algorithm: Simplest and most efficient. A request and a grant to enter, and a release to exit. Only 2 message times to enter.
- Decentralized Algorithm: Requires *n*-1 request messages, one to each of the other processes, and an additional *n*-1 grant messages. Entry takes 2(n-1) message times assuming that messages are passed sequentially over a LAN
- Token Ring Algorithm: If every process constantly wants to enter a critical region (each token pass will result in one entry). The token may sometimes circulate for hours without anyone being interested in it (number of messages per entry is unbounded). Entry delay varies from 0 (token just arrived) to *n*-1 (token just departed)

From Tanenbaum and van Steen, Distributed Systems: Principles and Paradigms, 2nd edition

# Transaction

A transaction is a collection of actions that make consistent transformations of system states while preserving system consistency.

- ➡ concurrency transparency
- ➡ failure transparency



System in a consistent state

System may be temporarily in an inconsistent state during execution

System in a consistent state

Begin Transaction

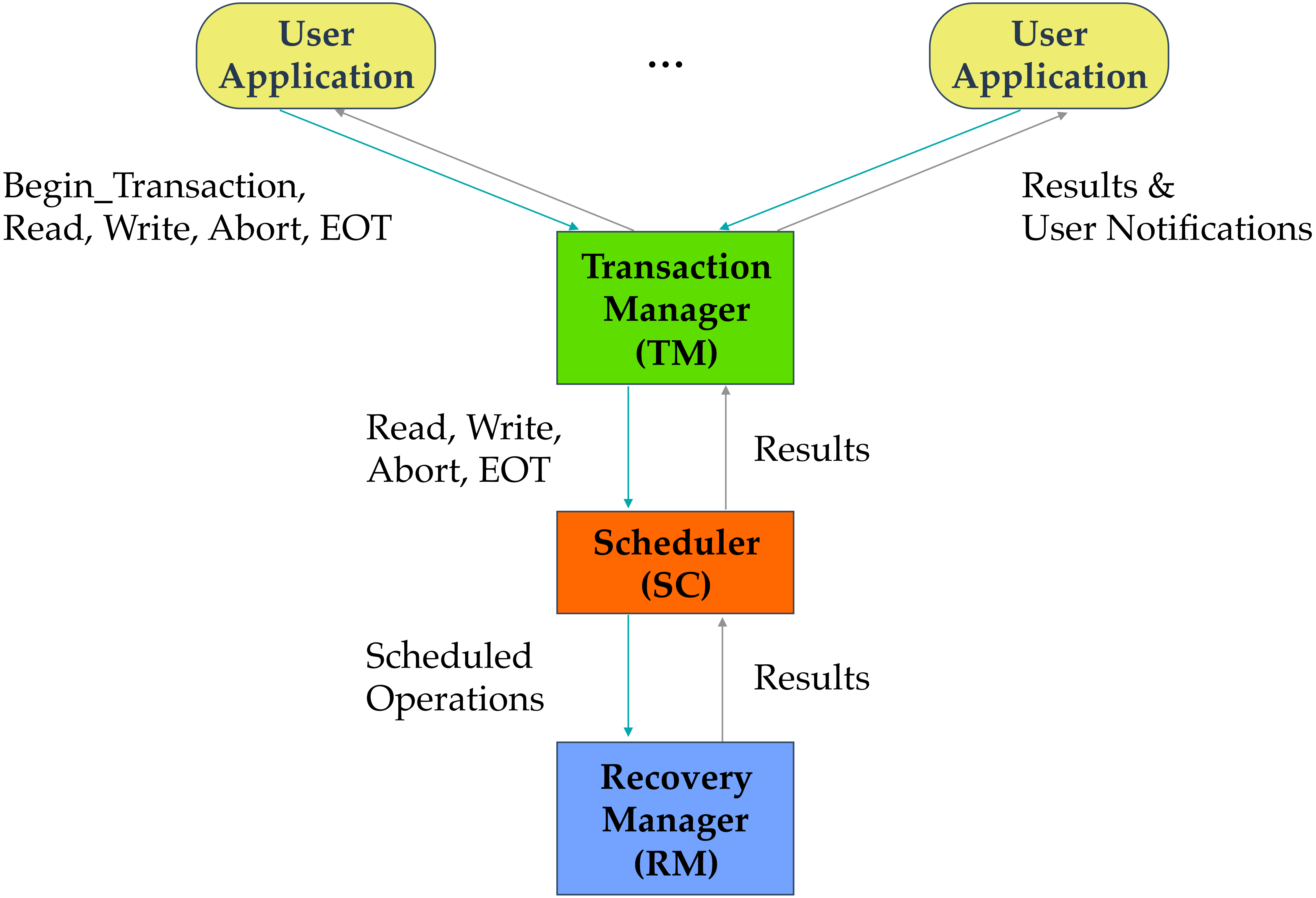
Execution of Transaction

End Transaction

# Transaction Primitives

- Special primitives required for programming using transactions
  - ➡ Supplied by the operating system or by the language runtime system
- Examples of transaction primitives:
  - ➡ BEGIN_TRANSACTION: Mark the start of a transaction
  - ➡ END_TRANSACTION (EOT): Terminate the transaction and try to commit (there may or may not be a separate COMMIT command)
  - ➡ ABORT_TRANSACTION: Kill the transaction and restore the old values
  - ➡ READ: Read data from a file (or other object)
  - ➡ WRITE: Write data to a file (or other object)

# Centralized Transaction Execution


User Application

…

User Application

Begin_Transaction,
Read, Write, Abort, EOT

Results &
User Notifications

Transaction Manager (TM)

Read, Write,
Abort, EOT

Results

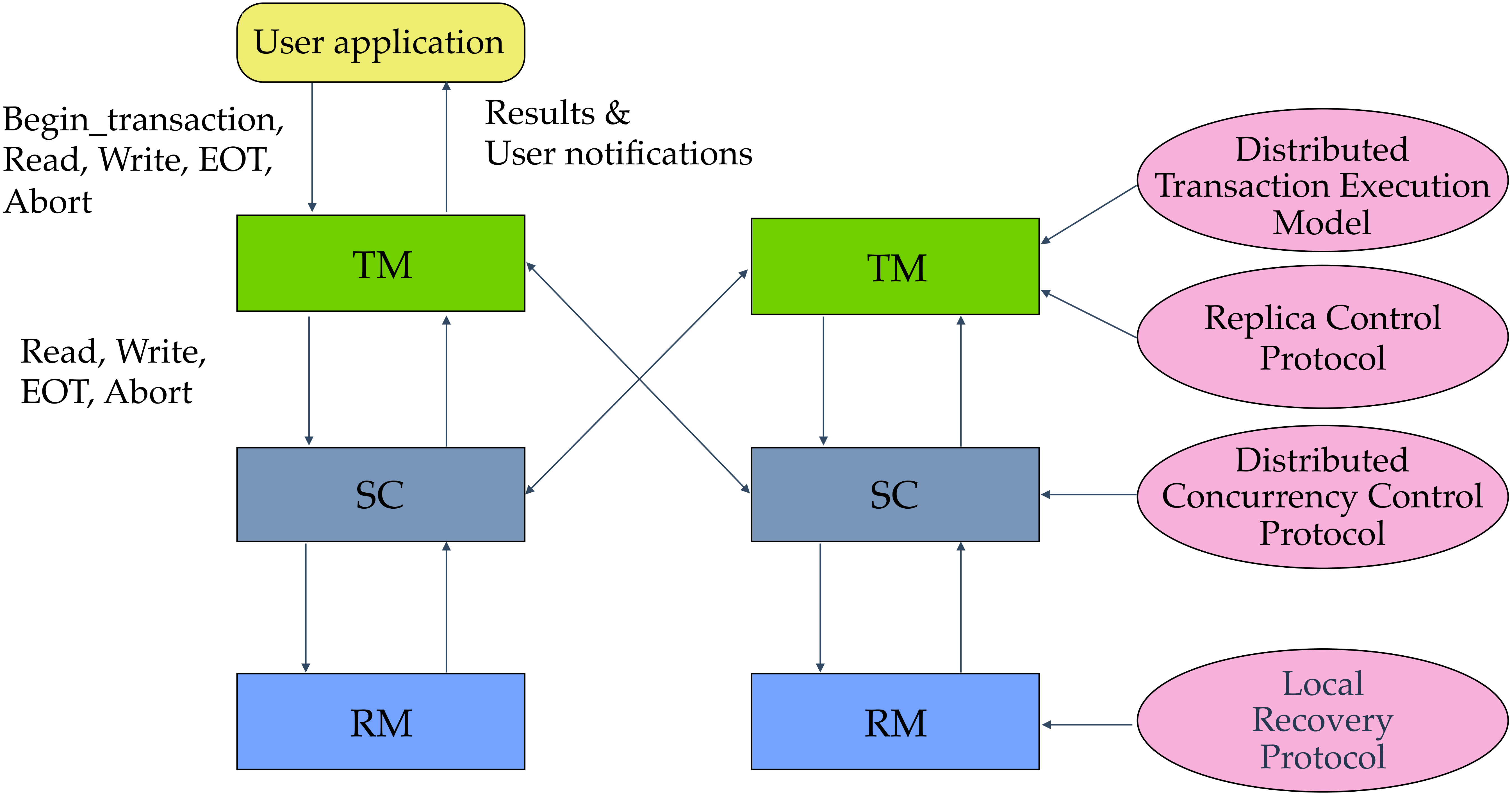
Scheduler (SC)

Scheduled
Operations

Results

Recovery Manager (RM)

# Distributed Transaction Execution


User application
Begin_transaction, Read, Write, EOT, Abort
Results & User notifications
TM
TM
Distributed Transaction Execution Model
Read, Write, EOT, Abort
Replica Control Protocol
SC
SC
Distributed Concurrency Control Protocol
RM
RM
Local Recovery Protocol

# Properties of Transactions

## ATOMICITY

- All or nothing
- Multiple operations combined as an atomic transaction

## CONSISTENCY

- No violation of integrity constraints
- Transactions are correct programs

## ISOLATION ← Our focus in this module

- Concurrent changes invisible → serializable

## DURABILITY

- Committed updates persist
- Database recovery

# Transactions Provide…

- Atomic and reliable execution in the presence of failures
- Correct execution in the presence of multiple user accesses
- Correct management of replicas (if they support it)

# Lost Update Problem

Initial values: A=100, B=200, C=300

| **Transaction *T*:** | | **Transaction *U*:** | |
|---|---|---|---|
| *balance* = *b*.*getBalance*();<br>*b*.*setBalance*(*balance**1.1);<br>*a*.*withdraw*(*balance*/10); | | *balance* = *b*.*getBalance*();<br>*b*.*setBalance*(*balance**1.1);<br>*c*.*withdraw*(*balance*/10); | |
| *balance* = *b*.*getBalance*(); | $200 | | |
| | | *balance* = *b*.*getBalance*(); | $200 |
| | | *b*.*setBalance*(*balance**1.1); | $220 |
| *b*.*setBalance*(*balance**1.1); | $220 | | |
| *a*.*withdraw*(*balance*/10); | $80 | | |
| | | *c*.*withdraw*(*balance*/10); | $280 |

From Coulouris, Dollimore and Kindberg, Distributed Systems: Concepts and Design, 3rd ed.

# Inconsistent Retrievals Problem

Initial values: A=200, B=200

| **Transaction *V*:**<br>*a.withdraw*(100);<br>*b.deposit*(100); | | **Transaction *W*:**<br>*aBranch.branchTotal*(); | |
|---|---|---|---|
| *a.withdraw*(100); | $100 | | |
| | | *total* = *a.getBalance*(); | $100 |
| | | *total* = *total*+*b.getBalance*(); | $300 |
| | | *total* = *total*+*c.getBalance*(); | |
| *b.deposit*(100); | $300 | : | |

From Coulouris, Dollimore and Kindberg, Distributed Systems: Concepts and Design, 3rd ed.

# Conflict Rules

| *Operations of different transactions* | | *Conflict* | *Reason* |
|---|---|---|---|
| *read* | *read* | No | Because the effect of a pair of *read* operations does not depend on the order in which they are executed |
| *read* | *write* | Yes | Because the effect of a *read* and a *write* operation depends on the order of their execution |
| *write* | *write* | Yes | Because the effect of a pair of *write* operations depends on the order of their execution |

From Coulouris, Dollimore and Kindberg, Distributed Systems: Concepts and Design, 3rd ed.

# Resolving Lost Update Example

| **Transaction *T*:** | | **Transaction *U*:** | |
|---|---|---|---|
| balance = b.getBalance();<br>b.setBalance(balance*1.1);<br>a.withdraw(balance/10); | | balance = b.getBalance();<br>b.setBalance(balance*1.1);<br>c.withdraw(balance/10); | |
| balance = b.getBalance(); | $200 | | |
| b.setBalance(balance*1.1); | $220 | | |
| | | balance = b.getBalance(); | $220 |
| | | b.setBalance(balance*1.1); | $242 |
| a.withdraw(balance/10); | $80 | | |
| | | c.withdraw(balance/10); | $278 |

From Coulouris, Dollimore and Kindberg, Distributed Systems: Concepts and Design, 3rd ed.

# Resolving Inconsistent Retrieval Problem

| **Transaction *V*:** | | **Transaction *W*:** | |
|---|---|---|---|
| *a.withdraw*(100);<br>*b.deposit*(100); | | *aBranch.branchTotal*(); | |
| *a.withdraw*(100); | $100 | | |
| *b.deposit*(100); | $300 | | |
| | | *total = a.getBalance*(); | $100 |
| | | *total = total+b.getBalance*(); | $400 |
| | | *total = total+c.getBalance*(); | |
| | | ... | |

From Coulouris, Dollimore and Kindberg, Distributed Systems: Concepts and Design, 3rd ed.

# Distributed Transaction Serializability

- Somewhat more involved. Two execution histories have to be considered:
  - ➡ local histories
  - ➡ global history
- For global transactions (i.e., global history) to be correct, two conditions are necessary:
  - ➡ Each local history should be correct (serializable).
  - ➡ Two conflicting operations should be in the same relative order in all of the local histories where they appear together.

# Locking-Based Algorithms

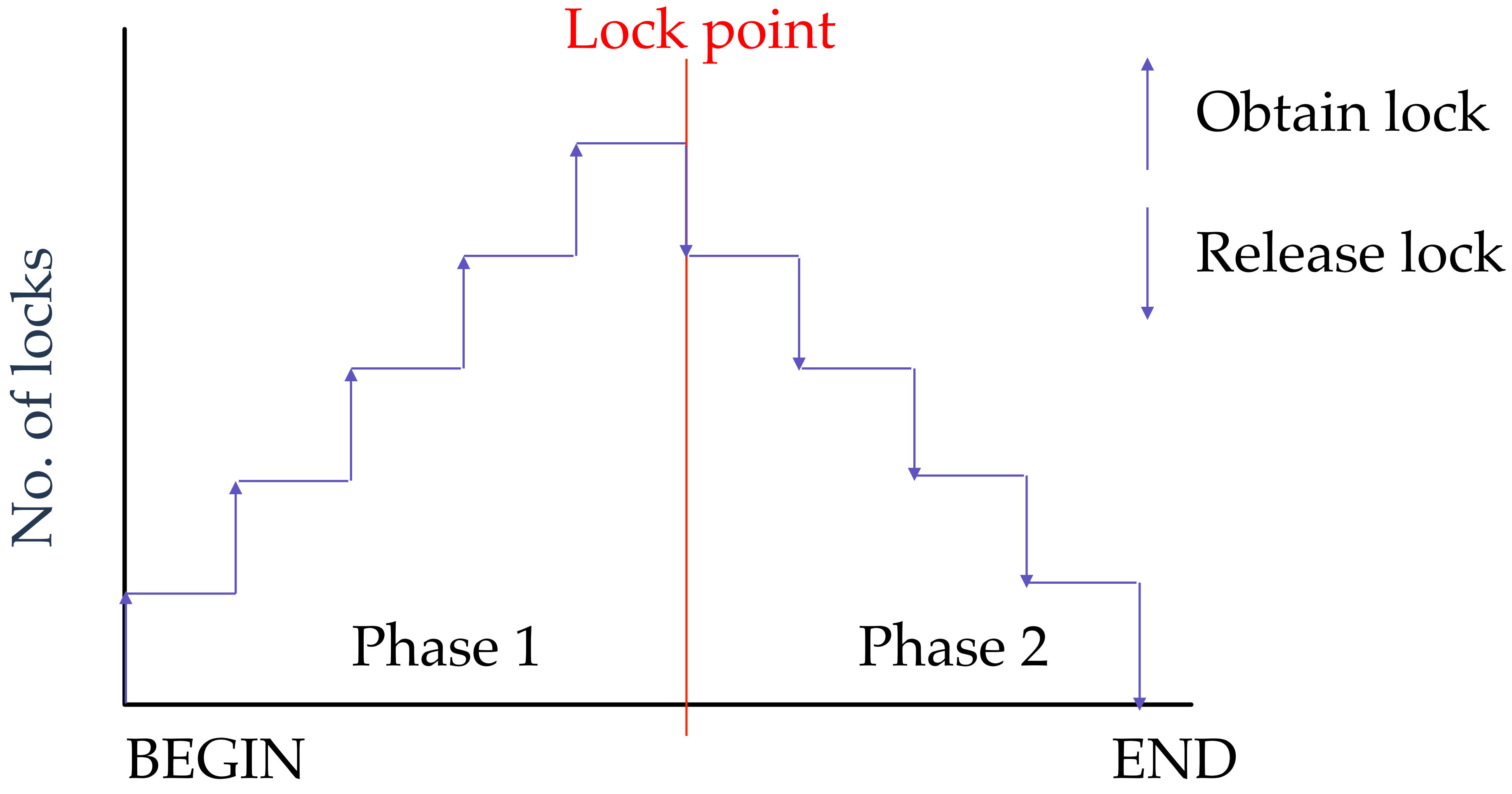
- Transactions indicate their intentions by requesting locks from the scheduler (called lock manager).
- Locks are either read lock (*rl*) [also called shared lock] or write lock (*wl*) [also called exclusive lock]
- Read locks and write locks conflict (because Read and Write operations are incompatible)

| | *rl* | *wl* |
|---|---|---|
| *rl* | yes | no |
| *wl* | no | no |

- Locking works nicely to allow concurrent processing of transactions.

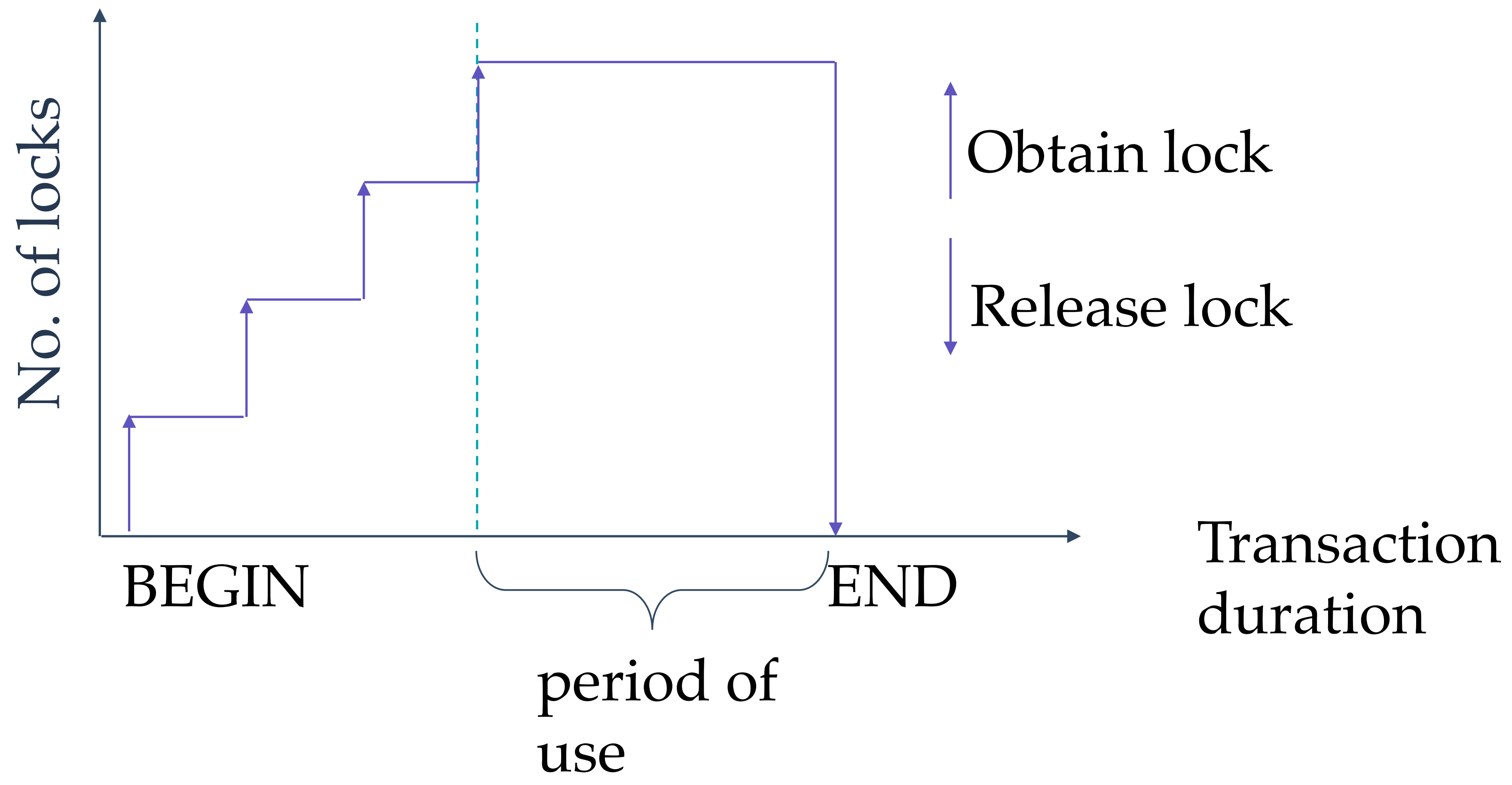
# Two-Phase Locking (2PL)

1. A transaction locks an object before using it.
2. When an object is locked by another transaction, the requesting transaction must wait.
3. When a transaction releases a lock, it may not request another lock.



# Strict 2PL

Hold locks until the end.



# Locking Example

| **Transaction *T*:** | | **Transaction *U*:** | |
|---|---|---|---|
| *balance* = *b.getBalance*(); | | *balance* = *b.getBalance*(); | |
| *b.setBalance*(*bal**1.1); | | *b.setBalance*(*bal**1.1); | |
| *a.withdraw*(*bal*/10); | | *c.withdraw*(*bal*/10); | |
| Operations | Locks | Operations | Locks |
| *openTransaction* | | | |
| *bal* = *b.getBalance*(); | write lock *B* | | |
| *b.setBalance*(*bal**1.1); | | *openTransaction* | |
| *a.withdraw*(*bal*/10); | write lock *A* | *bal* = *b.getBalance*(); | waits for *T*'s lock on *B* |
| *closeTransaction* | unlock *A*, *B* | ● ● ● | |
| | | | write lock *B* |
| | | *b.setBalance*(*bal**1.1); | |
| | | *c.withdraw*(*bal*/10); | write lock *C* |
| | | *closeTransaction* | unlock *B*, *C* |

From Coulouris, Dollimore and Kindberg, Distributed Systems: Concepts and Design, 3rd ed.
© Addison-Wesley Publishers 2000