Synchronization

Chapter 5

Clock Synchronization

- In a centralized system time is unambiguous.

  (*each computer has its own clock*)
- In a distributed system achieving agreement on time is not trivial.

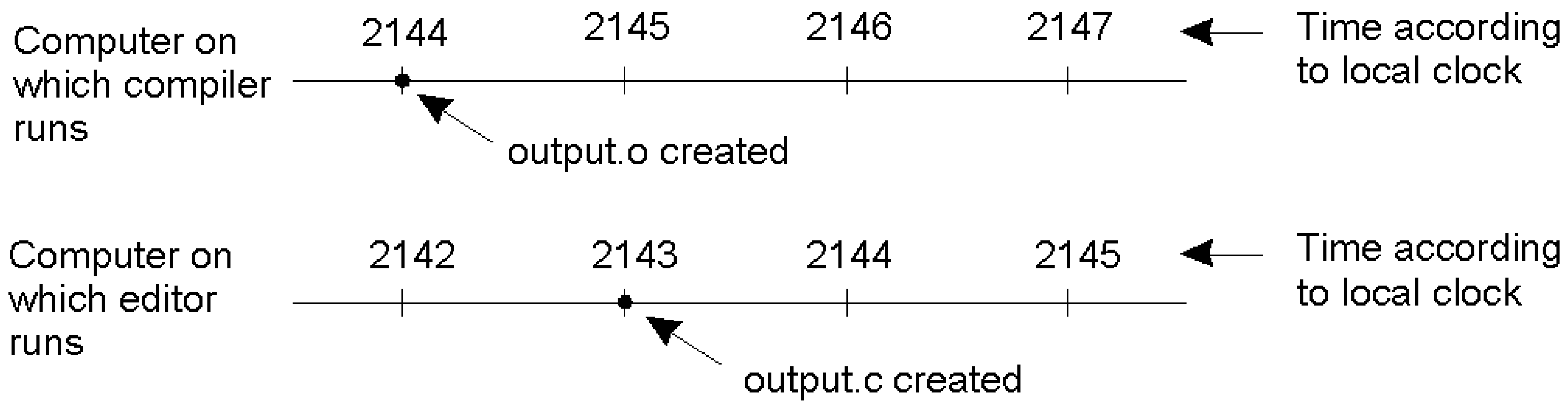
  (*it is impossible to guarantee that clocks run at exactly the same frequency*)

- Clock synchronization

- Logical clocks

Clock Synchronization

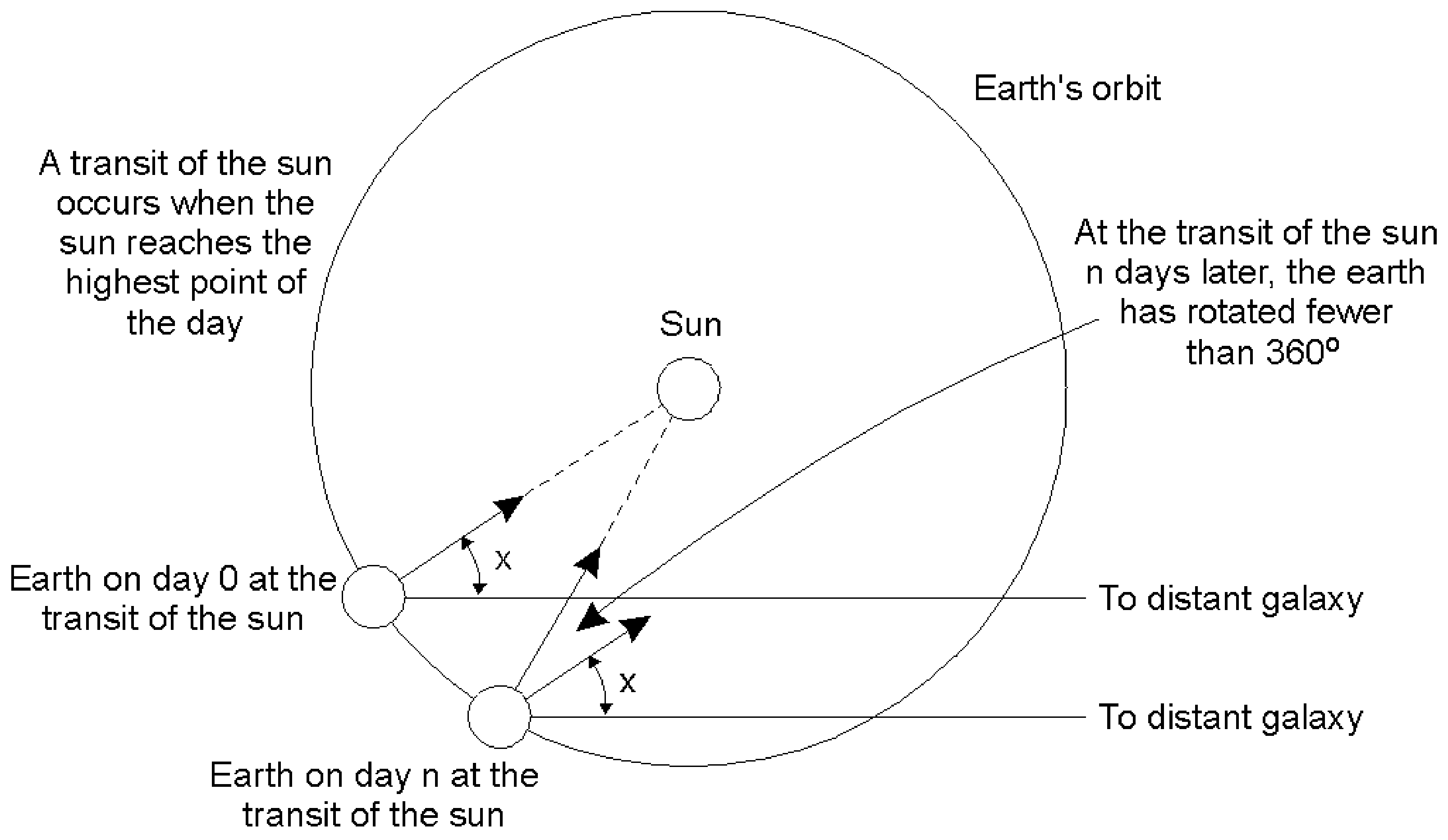
Example: the *make* program.



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

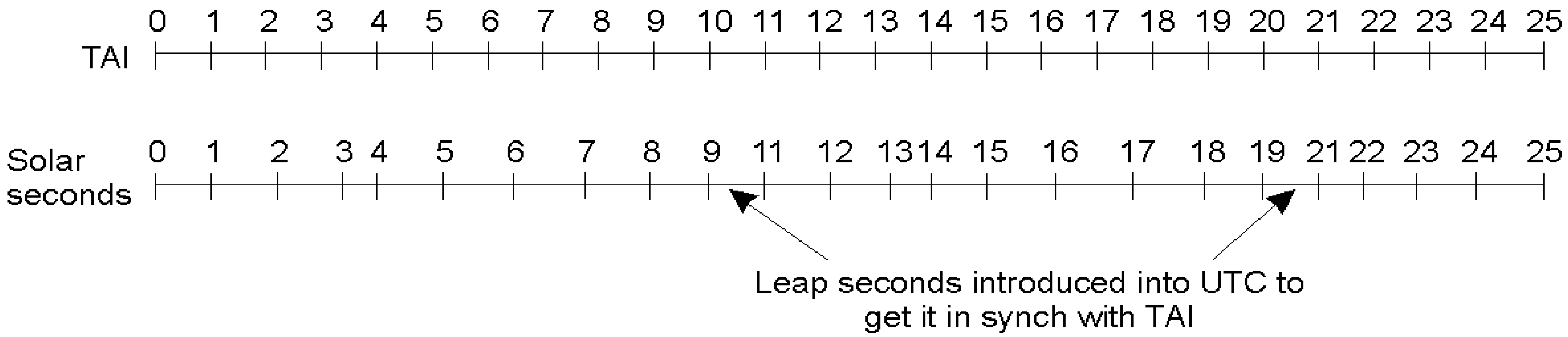
- When each machine has its own clock, an event that occurred after another event may nevertheless be assigned an earlier time.

Physical Clocks (1)


Earth's orbit
A transit of the sun occurs when the sun reaches the highest point of the day
At the transit of the sun n days later, the earth has rotated fewer than 360°
Sun
x
Earth on day 0 at the transit of the sun
To distant galaxy
x
To distant galaxy
Earth on day n at the transit of the sun

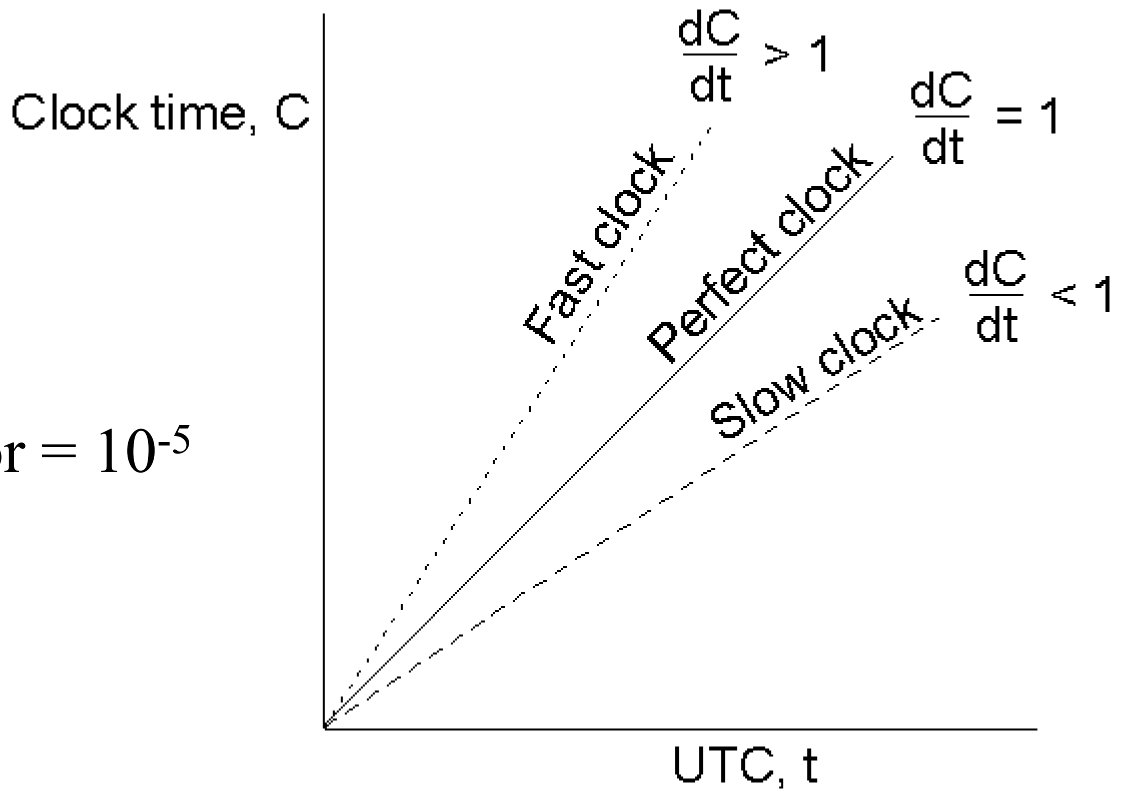
Computation of the mean solar day.

Physical Clocks (2)



TAI seconds are of constant length, unlike solar seconds. Leap seconds are introduced when necessary to keep in phase with the sun.

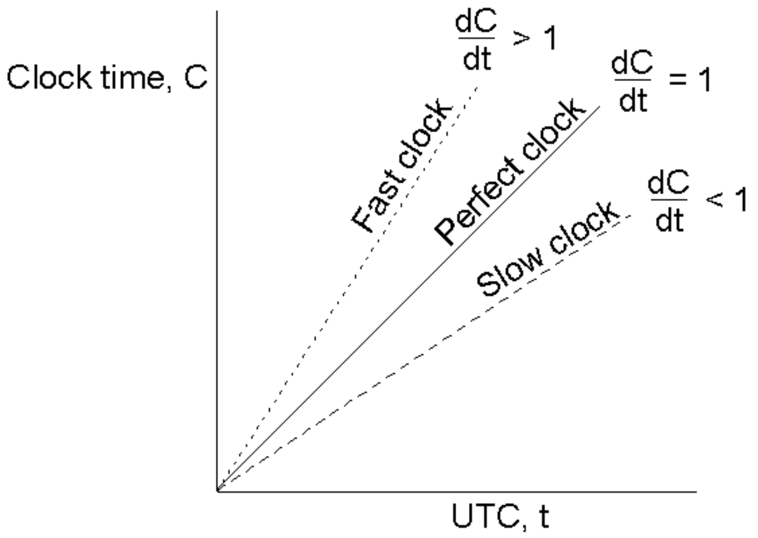
Clock Synchronization Algorithms



Relative error = 10^{-5}

Clock time and UTC when clocks tick at different rates.

Clock Synchronization Algorithms

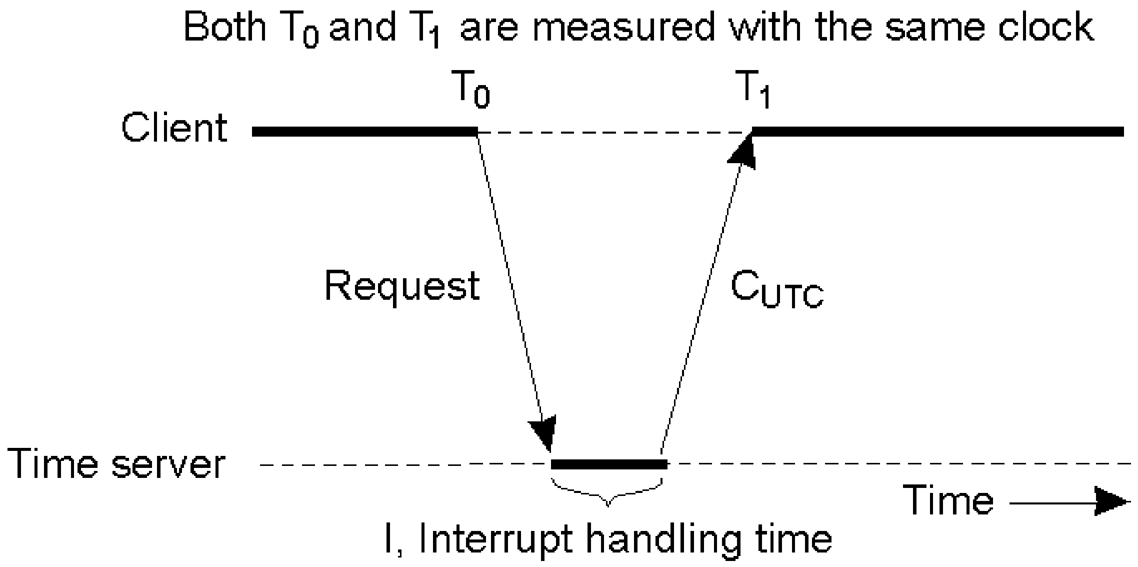


If there exists a constant ρ such that

$1 - \rho <= dC/dt <= 1 + \rho$ (maximum drift rate)

after Δt the difference can be ($2\rho\ \Delta t$).

Cristian's Algorithm

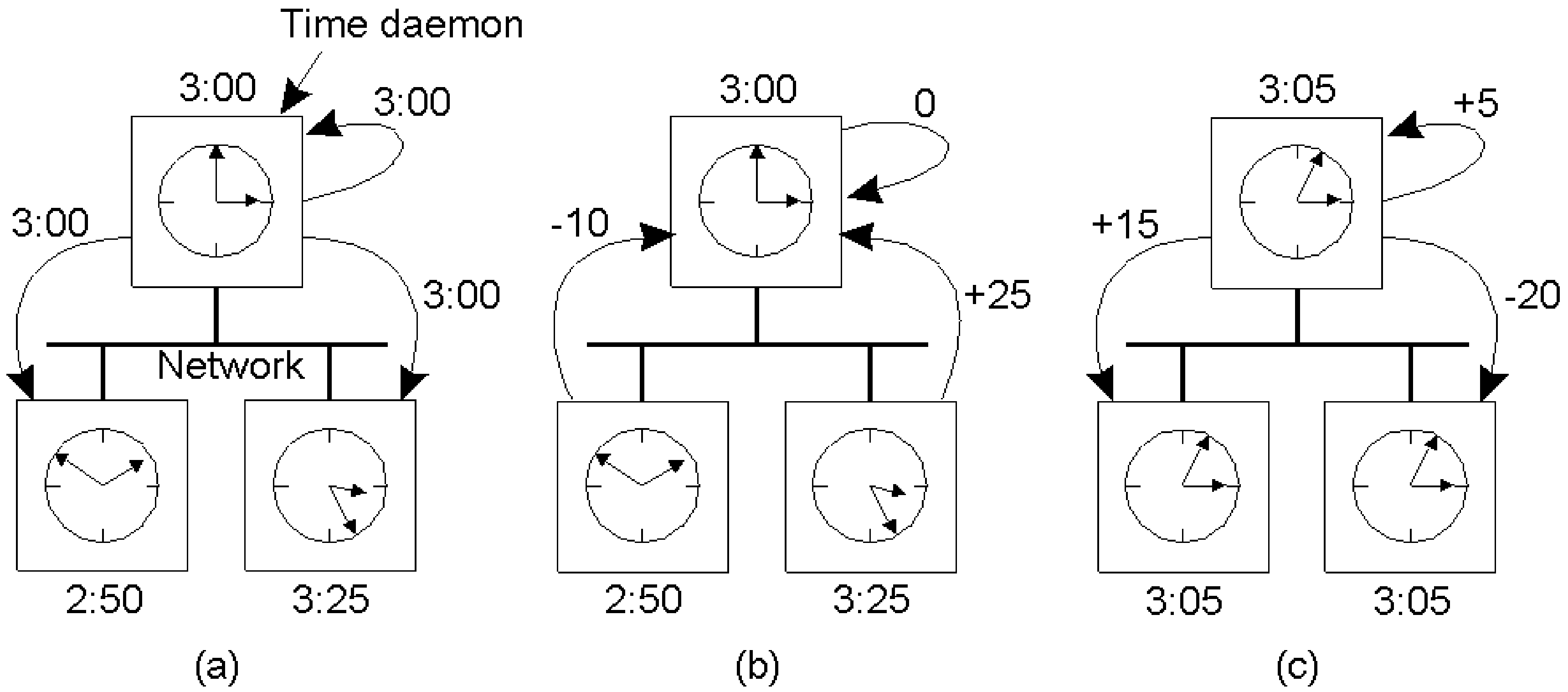

Both T_0 and T_1 are measured with the same clock
T_0
T_1
Client
Request
C_{UTC}
Time server
Time
I, Interrupt handling time

Getting the current time from a time server.

Two problems:

- time must never run backward (slow increase of clock)
- server reply with C_{UTC} requires an amount of time: (T1-T0-I)/2.

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

Logical Clocks

a) Logical Clocks are used when the internal consistency of clocks matters, not whether they are exactly equal to the real time.

b) Lamport:
   a) *if two processes do not interact it is not necessary to synchronize their clocks.*
   b) *What is important with interacting processes is the order in which events occur.*

Lamport Timestamps

- Relation: **happens-before** $\rightarrow$
- $a \rightarrow b$ means "a happens before b"

- If a and b are two events in the same process and a occurs before b, then $a \rightarrow b$ is true

- In two processes, if a is the event of sending the message m and b is the event of receiving the message m, then $a \rightarrow b$ is true

- If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$

Lamport Timestamps

- Considering two events x and y in two non-interacting processes, then $x \rightarrow y$ is not true, but neither is $x \rightarrow y$.

- x and y are said to be **concurrent**.

- For each event a what is needed is a global measure of time to assign a time value $C(a)$ on which **all processes agree**

- If $a \rightarrow b$ then $C(a) < C(b)$.

Lamport Timestamps

Total ordering can be achieved if :

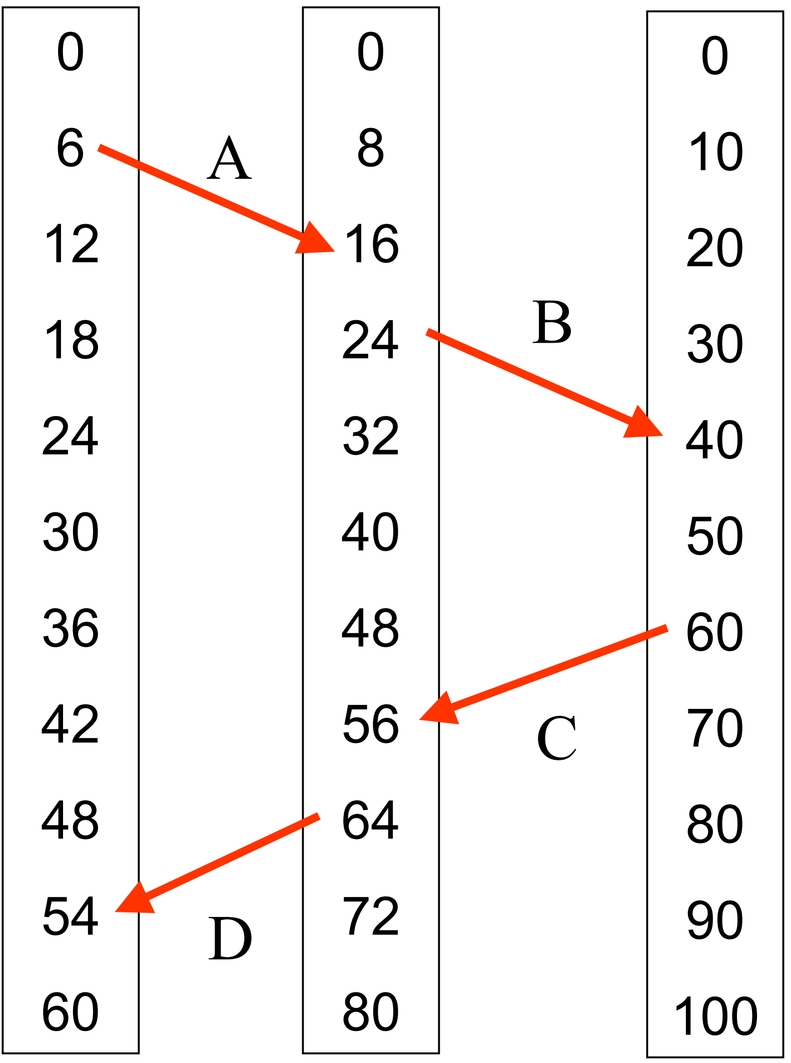
- Each message carries the sending time according to the sender's clock

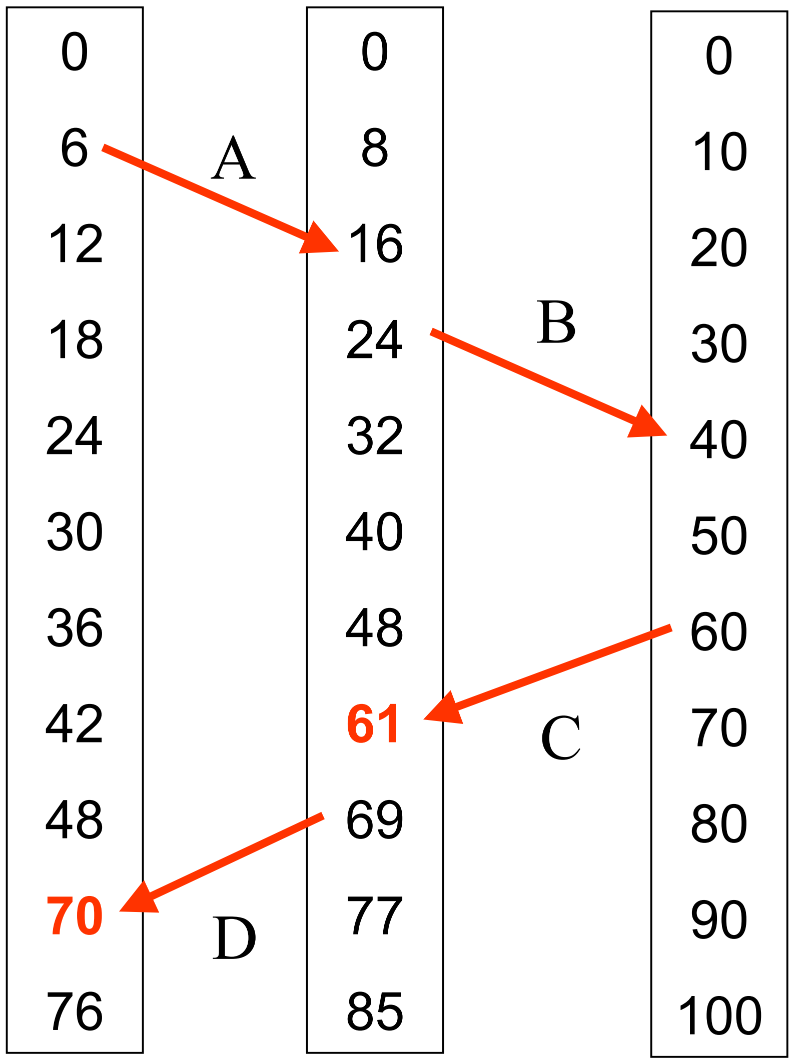
- When the message arrives the receiver clock must be at least one more than the sending time.

- Between two events the clock must tick al least once.

- No two events ever occur at exactly the same time.

Lamport Timestamps



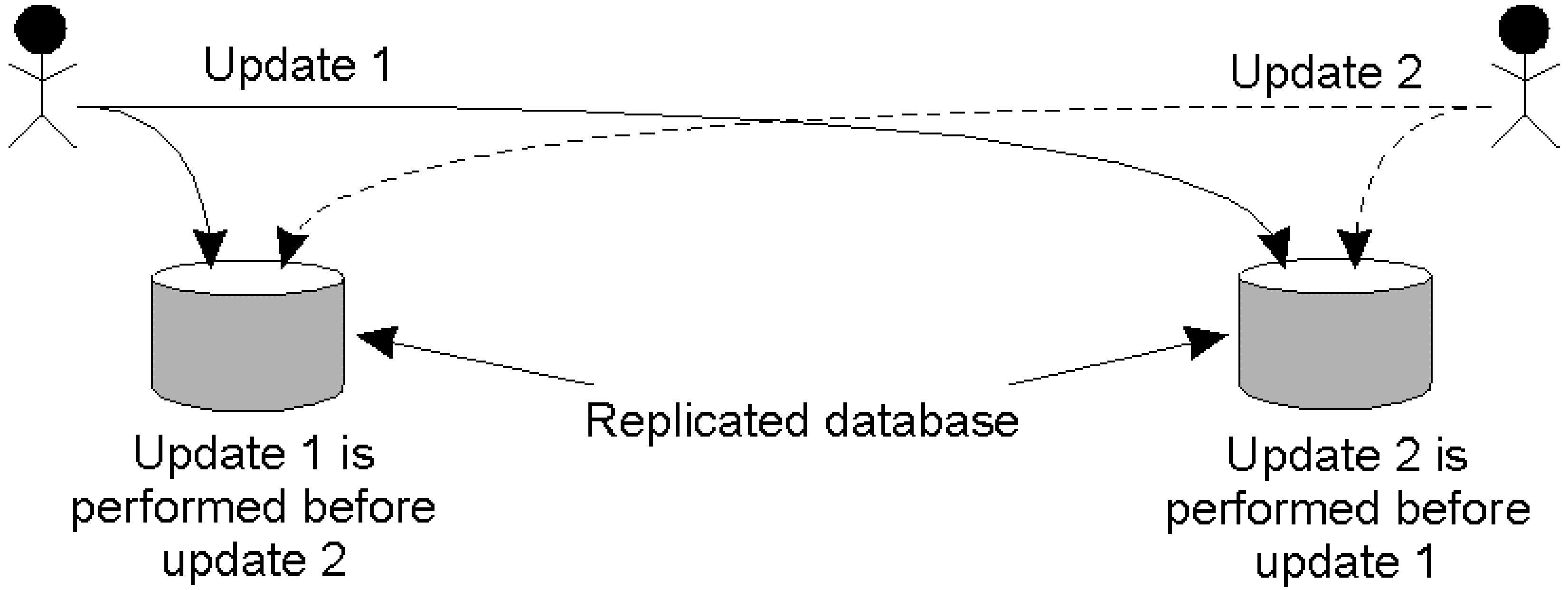


Three processes, each with its own clock. The clocks run at different rates.

Lamport's algorithm corrects the clocks.

Example: Totally-Ordered Multicasting

Replicated database in two sites


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

Updating a replicated database and leaving it in an inconsistent state.

A **totally-ordered multicast** (all the messages all delivered in th same order) is required.

Totally-Ordered Multicasting

- Each message is multicasted to each process and timestamped with the logical time of the sender and put on the queue in the timestamp order
- Messages are delivered in the order they are sent
- Each message is acknowledged to the other processes
- No two messages have the same timestamps
- Each process has the same copy of the queue.

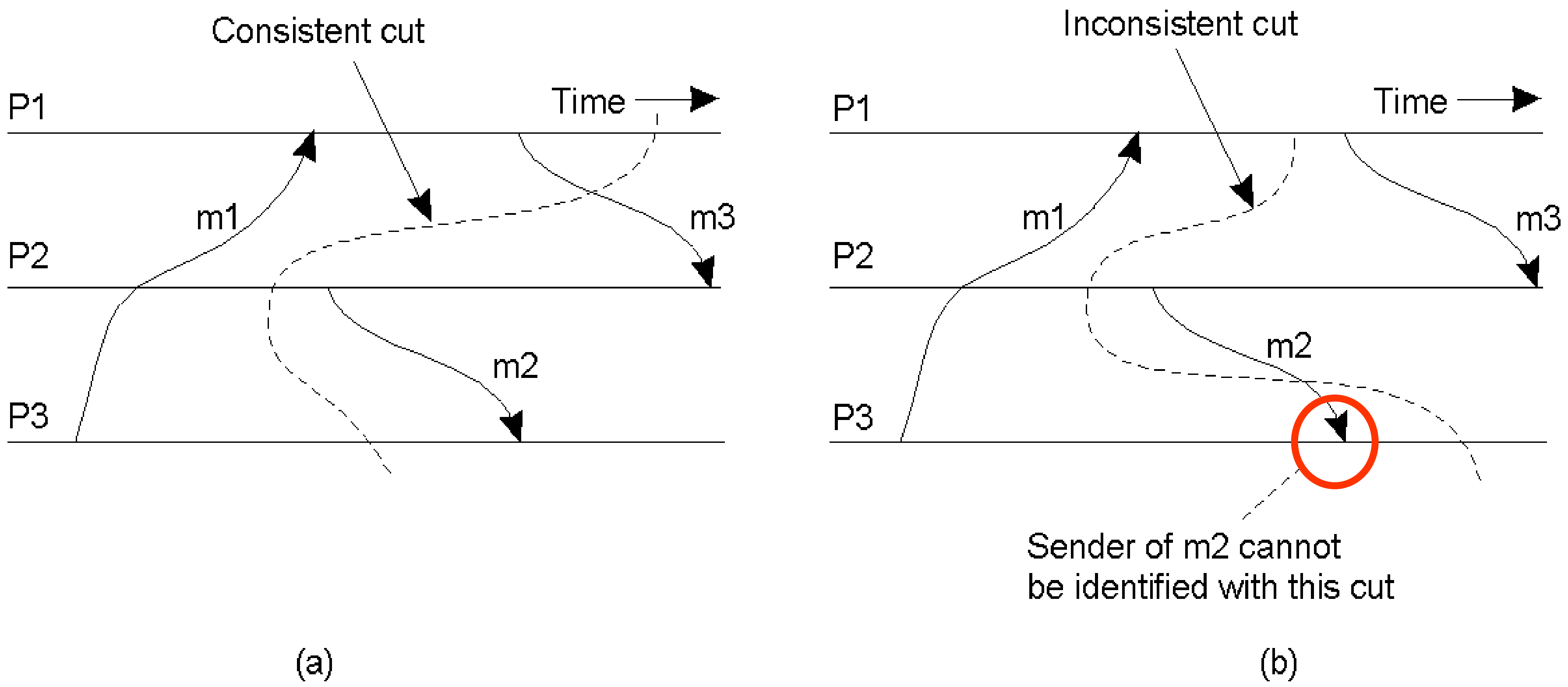
Global State (1)

a) The **global state** of a distributed system is given by the ***collection of local state*** of each process plus the ***messages in transit***.

b) Global state awareness is useful in several cases.

c) A **distributed snapshot** is a state in which the distributed system might have been (a consistent global state)

Global State (2)



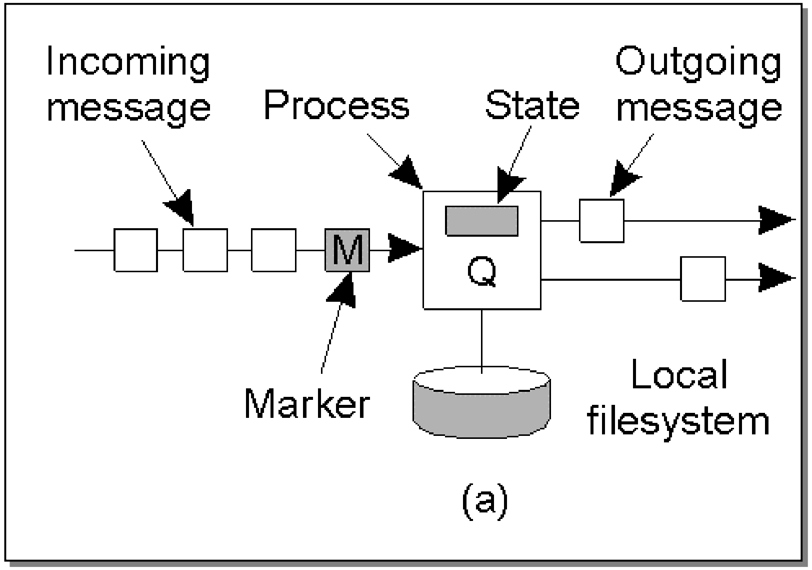
(a) A consistent cut

(b) An inconsistent cut

Global State (3)

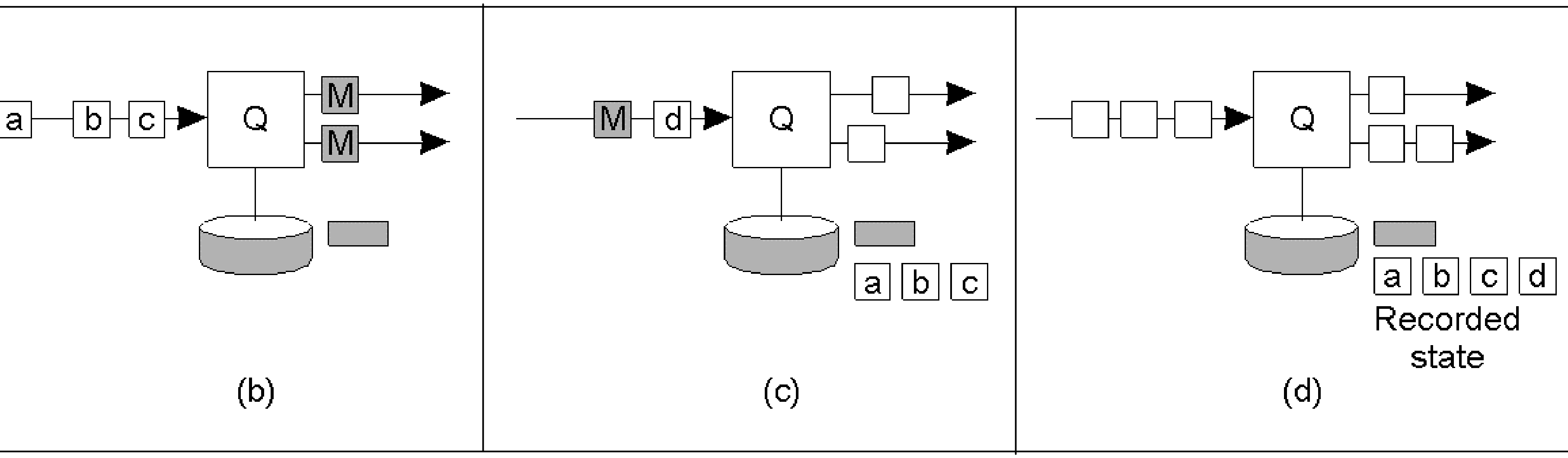
Using distributed snapshots is possible to record a global state.

1. A process *P* starts the algorithm recording its own state and sending a marker along its outgoing channels indicating the receiver should participate in recording the global state.



a. Organization of a process Q and channels for a distributed snapshot

Global State (4)



b. When process *Q* receives a marker for the first time records its local state and send the marker along its out channels.

c. *Q* records all incoming messages

d. *Q* receives a marker for its incoming channel and finishes recording the state of the incoming channel

Global State (5)

- When a process received and processed all the markers along all its incoming channels finishes its role in the algorithm and send the state to be collected.
- Any process can start the algorithm, thus the markers is tagged with the identifier of the starting process.

Distributed Termination (1)

- Detecting termination of a distributed computation is not trivial.
- A distributed snapshot may not show a termination state because messages can be still in transit.
- For termination detection with distributed snapshot is needed that all channels are empty.

Distributed Termination (2)

- When a process Q finishes its part of the snapshot can send a *DONE* message to its predecessors if two conditions are met
  - all Q's successors returned a DONE message
  - Q has not received messages between the time of recording its state and the receiving the marker along each of its channels
- In all other cases Q sends a *CONTINUE* message to its predecessor.
- When only *DONE* messages are received by the initiator process the computation is terminated.

Election Algorithms

Algorithms for **electing a coordinator** (with a special role) among the processes that compose a distributed computation.

- Each process is identified by a unique id number
- Every process knows the id num. of every other process
- But it does not know which one are up or down
- Election terminates when all processes agree on a coordinator.

The Bully Algorithm (1)

A process P holds an election as follows:

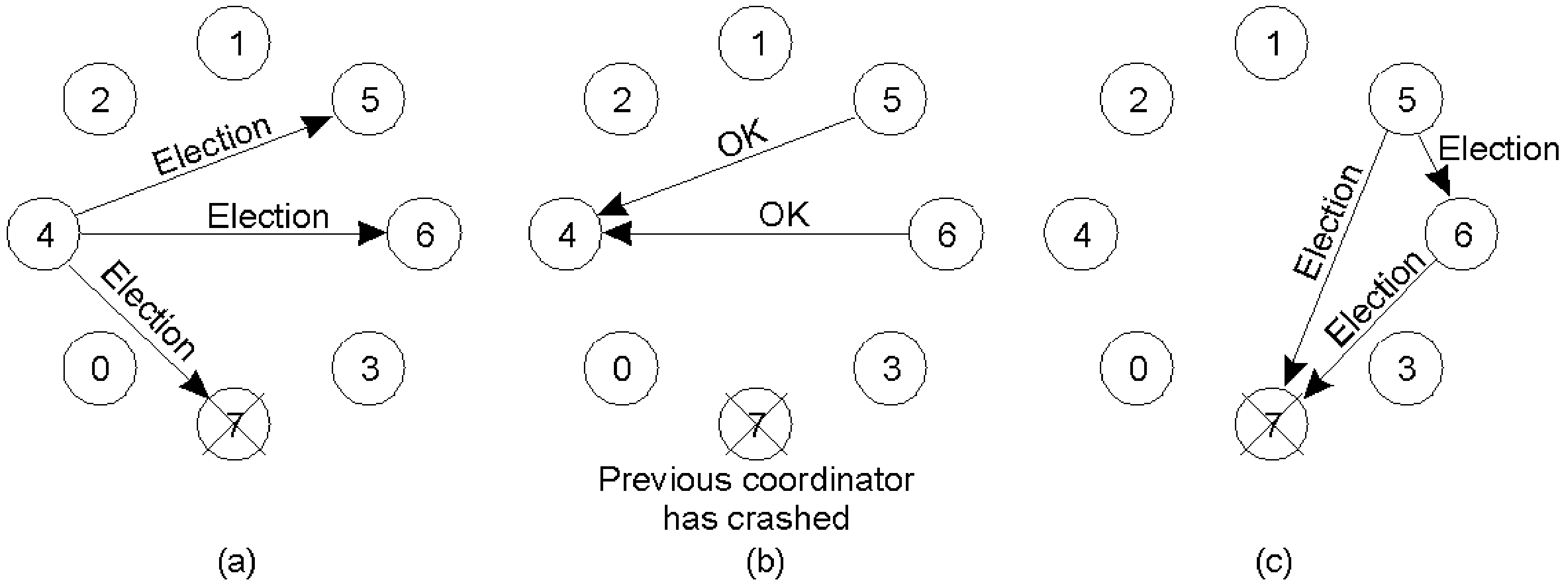
1. P send an *ELECTION* message to all processes with higher numbers

2. If no one responds, P becomes the new coordinator

3. If one with higher id num. Responds it takes over and continue the election algorithm.

4. The new coordinator notifies all the processes.

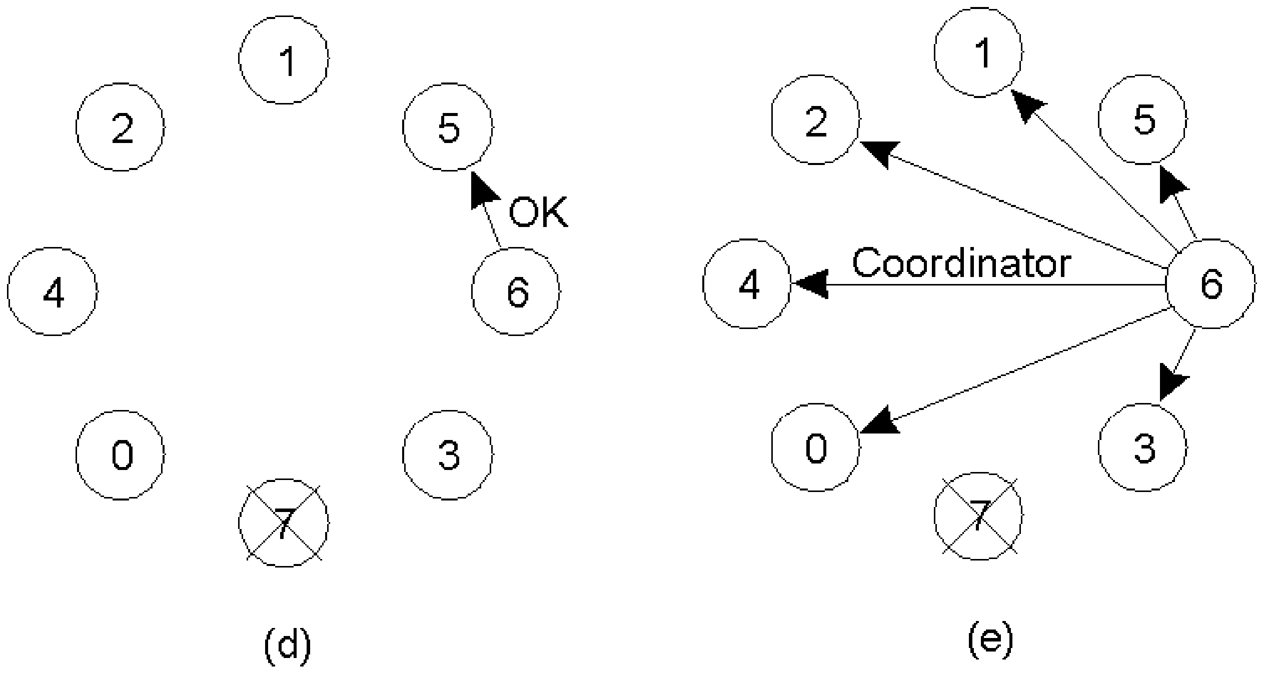
The Bully Algorithm (2)



The bully election algorithm

- Process 4 holds an election
- Process 5 and 6 respond, telling 4 to stop
- Now 5 and 6 each hold an election

The Bully Algorithm (3)



d) Process 6 tells 5 to stop

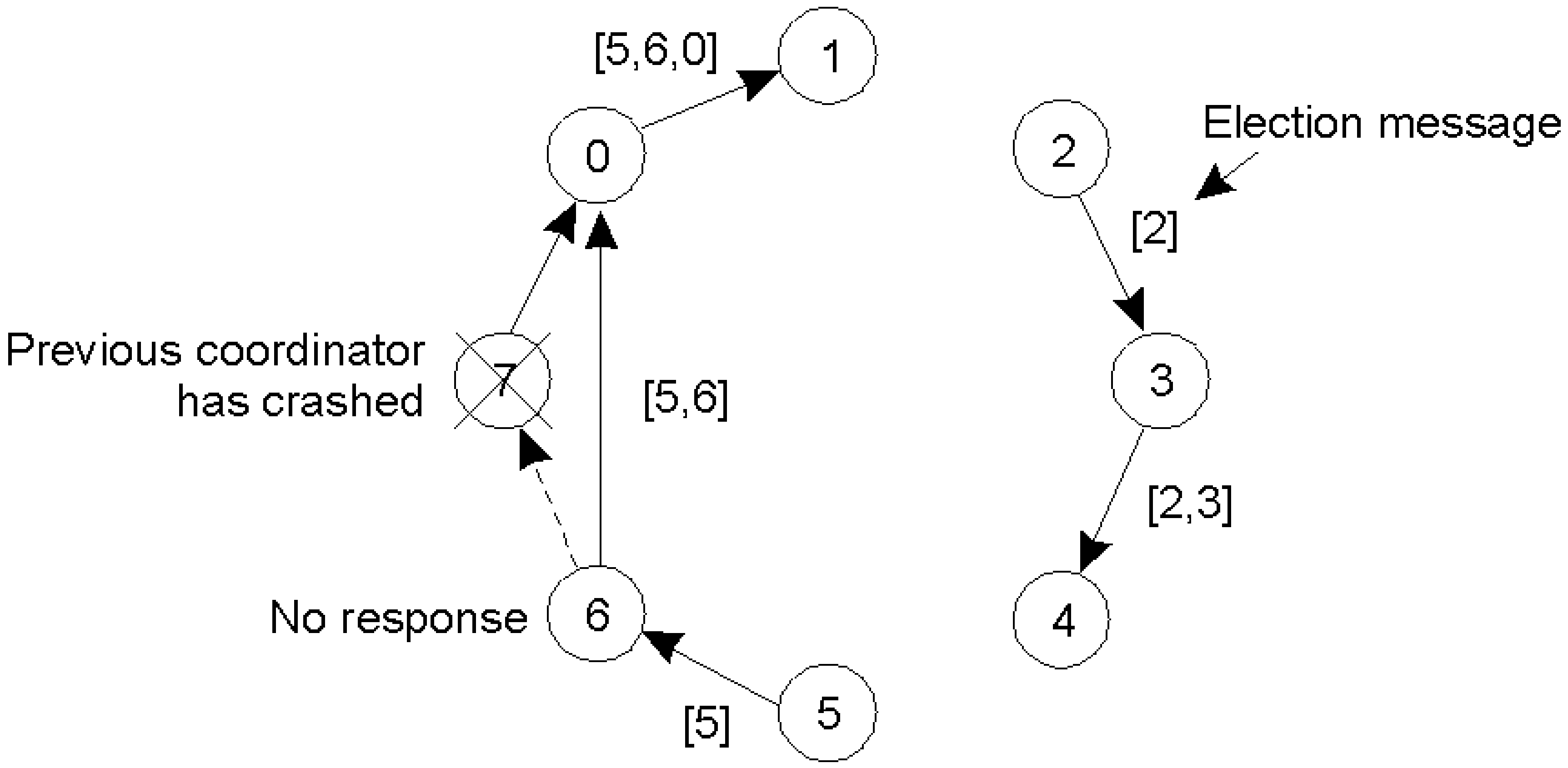
e) Process 6 wins and tells everyone

A Ring Algorithm (1)

Election algorithm using a ring:

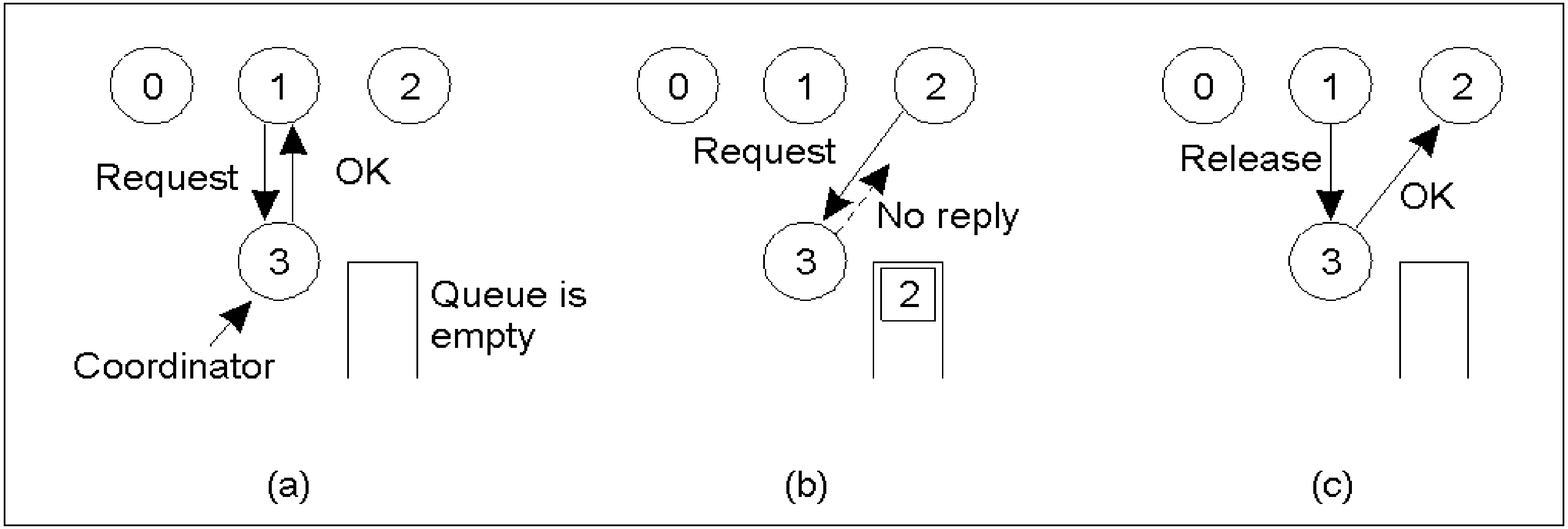
- Each process knows who its successor is
- The election process is initiated by a process that sends an *ELECTION* message with its number to its successor
- Each sender add its number to the message.
- When the message returns to the initiator, it looks for the highest number and send a *COORDINATOR* message in the ring with the number of the new coordinator.

A Ring Algorithm (2)



Election algorithm using a ring

Mutual Exclusion: A Centralized Algorithm



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

A Distributed Algorithm (1)

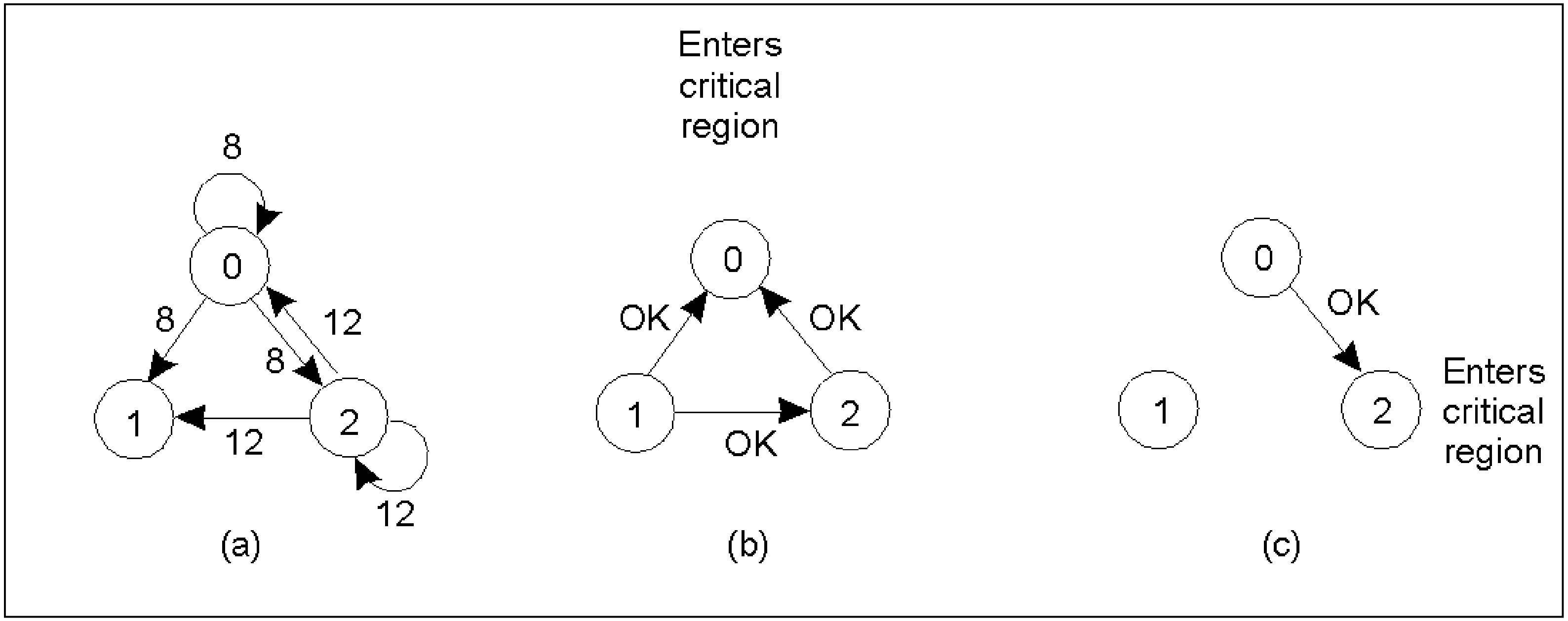
Message sending is reliable and total time ordering is assured.

a) When a process wants to enter a critical region sends to all processes <cr_name, proc_id, time>

b) When a process receives a message
   1. If it is not in a critical region and not want to enter, send back OK
   2. If it is in a critical region does not reply and queues the request
   3. If it wants to enter a critical region, compares the timestamp if its request with the timestamp of the received message, lower win
   4. When a process exits a critical region sends OK to all the processes on its queue

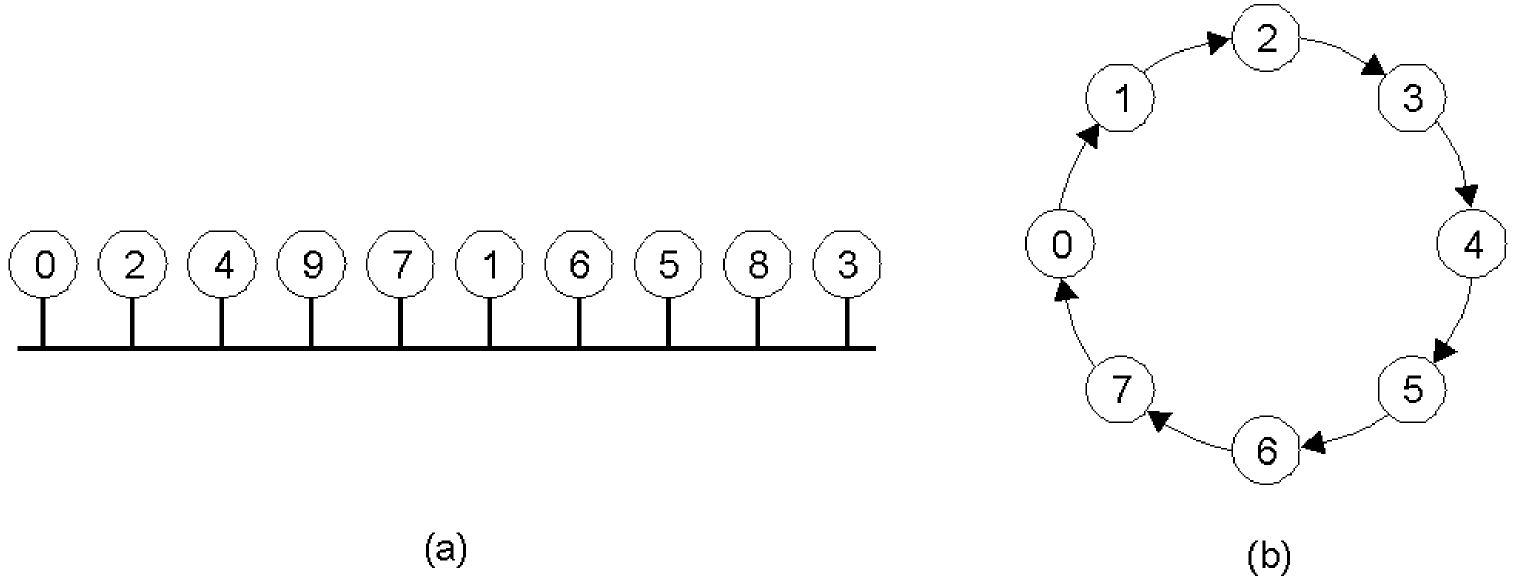
It works but it is not efficient!

A Distributed Algorithm (2)



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.

A Token Ring Algorithm



(a) An unordered group of processes on a network. (b) A logical ring constructed in software.

Process 0 is given a token and it circulate on the ring.

A process *N* that has the token may enter the critical region or pass it to *N+1*.

Comparison

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

A comparison of three mutual exclusion algorithms.

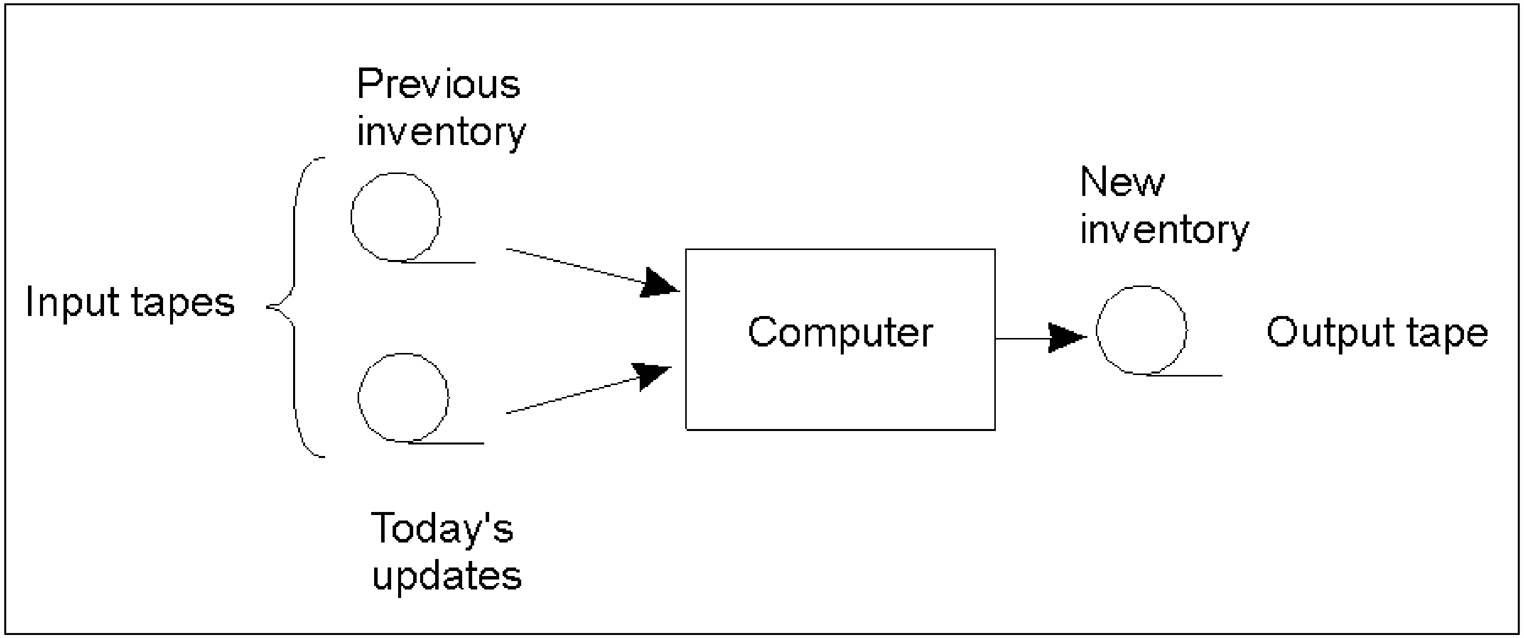
The Transaction Model (1)

- Transactions are composed of a set of operations that respect the all-or-nothing property.

- Example of transaction with 2 operations:
  - op1. Withdraw 1000 □ from account 1
  - op2. Deposit 1000 □ to account 2.

If a failure occurs between op1 and op2, transaction must be aborted.

The Transaction Model (2)



Updating a master tape is fault tolerant.

The Transaction Model (3)

Special primitives are defined for transactions.

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

Examples of primitives for transactions.

The Transaction Model (4)

```
BEGIN_TRANSACTION
  reserve WP -> JFK;
  reserve JFK -> Nairobi;
  reserve Nairobi -> Malindi;
END_TRANSACTION
```

(a)

```
BEGIN_TRANSACTION
  reserve WP -> JFK;
  reserve JFK -> Nairobi;
  reserve Nairobi -> Malindi full =>
ABORT_TRANSACTION
```

(b)

(a) Transaction to reserve three flights commits

(b) Transaction aborts when third flight is unavailable

The Transaction Model (5)

ACID PROPERTIES

- **ATOMIC**: the transaction happens as indivisible
- **CONSISTENT**: the transaction does not violate system invariants
- **ISOLATED**: concurrent transactions do not interfere with each other (SERIALIZABLE)
- **DURABLE**: after commit, changes are permanent.

Nested and Distributed Transactions

- Other than “flat transactions” other types of transactions are used.

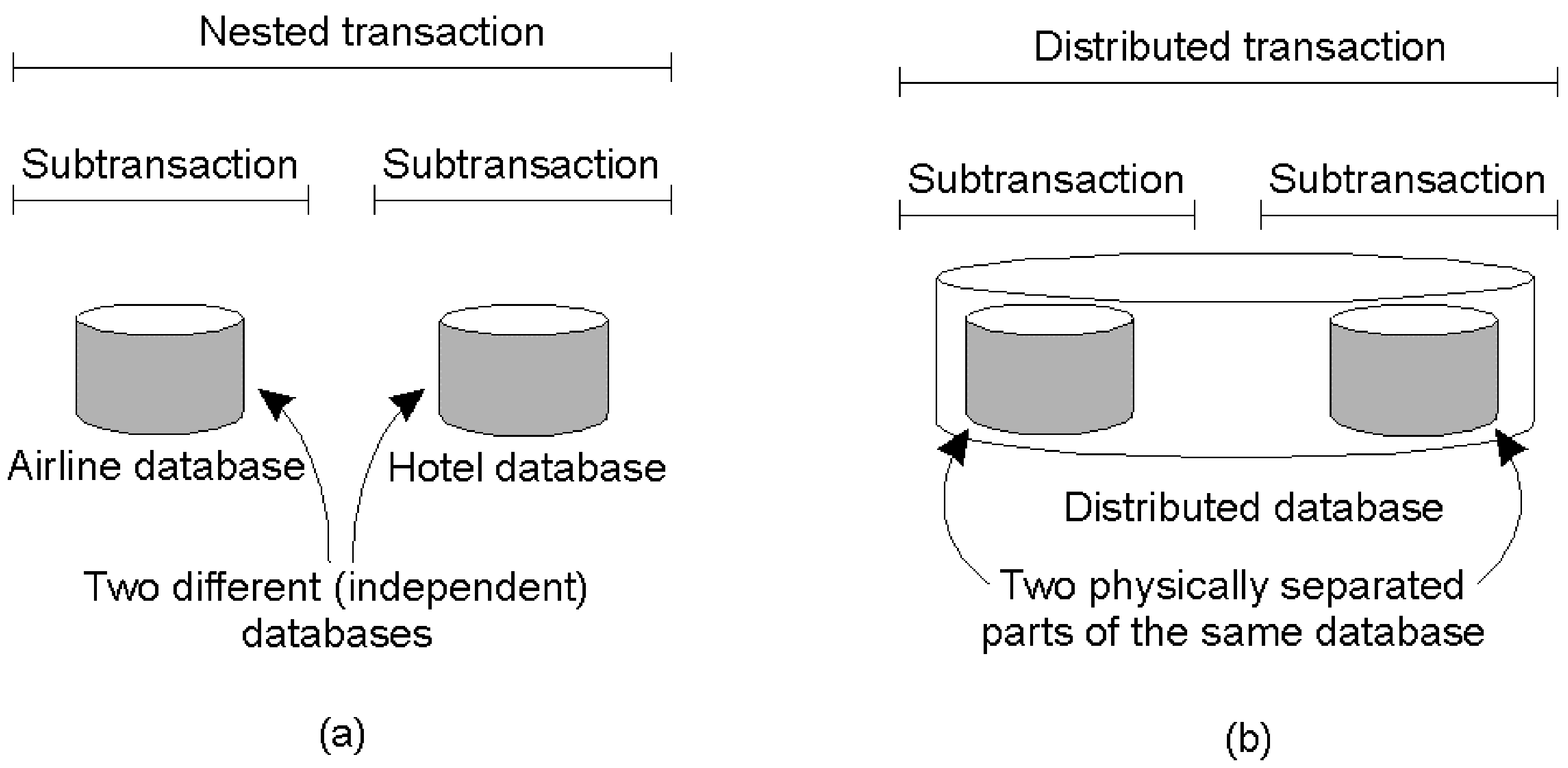
  A **nested transaction** is a transaction that is logically decomposed into a hierarchy of sub-transactions.

  A *hierarchical abort* mechanism is to be provided.

  A **distributed transaction** is a flat transaction that operated on distributed data.

  A *distributed locking* mechanism is needed.

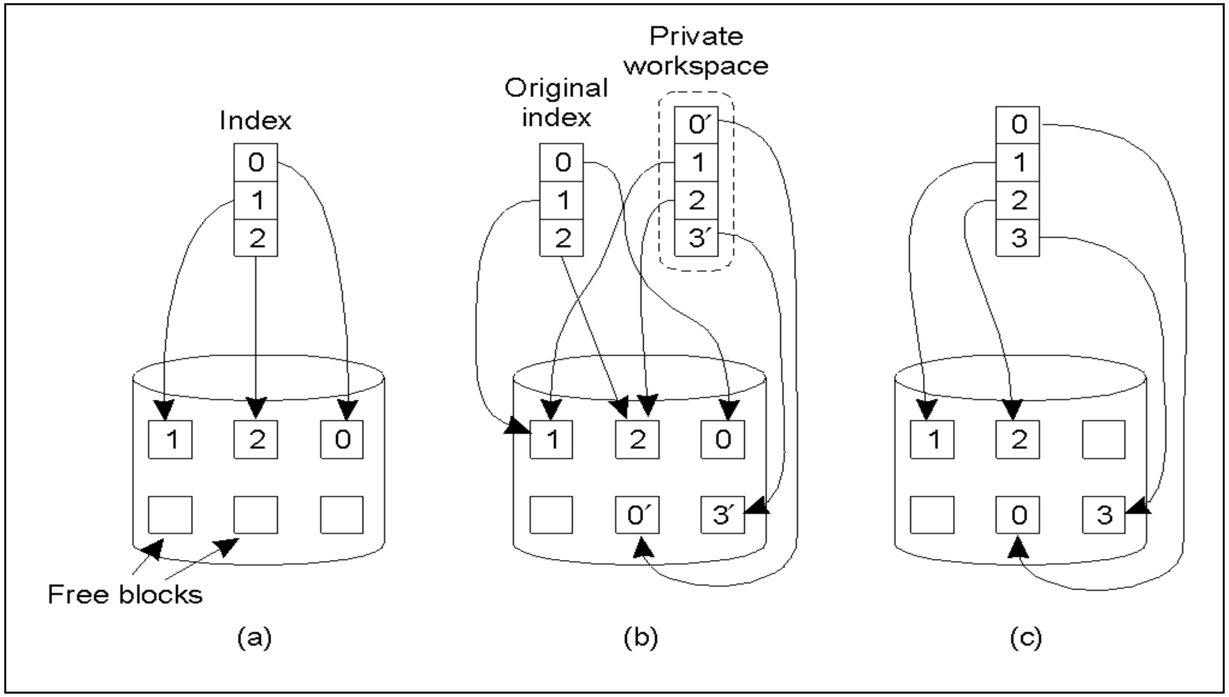
Distributed Transactions



(a) A nested transaction
(b) A distributed transaction

Private Workspace

Private workspace is a method to implement **atomic** transactions.



(a) The file index and disk blocks for a three-block file

(b) The situation after a transaction has modified block 0 and appended block 3

(c) After committing

Writeahead Log

Writeahead log is another method to implement atomic transactions.

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)

(a) A transaction

(b) – (d) The log before each statement is executed

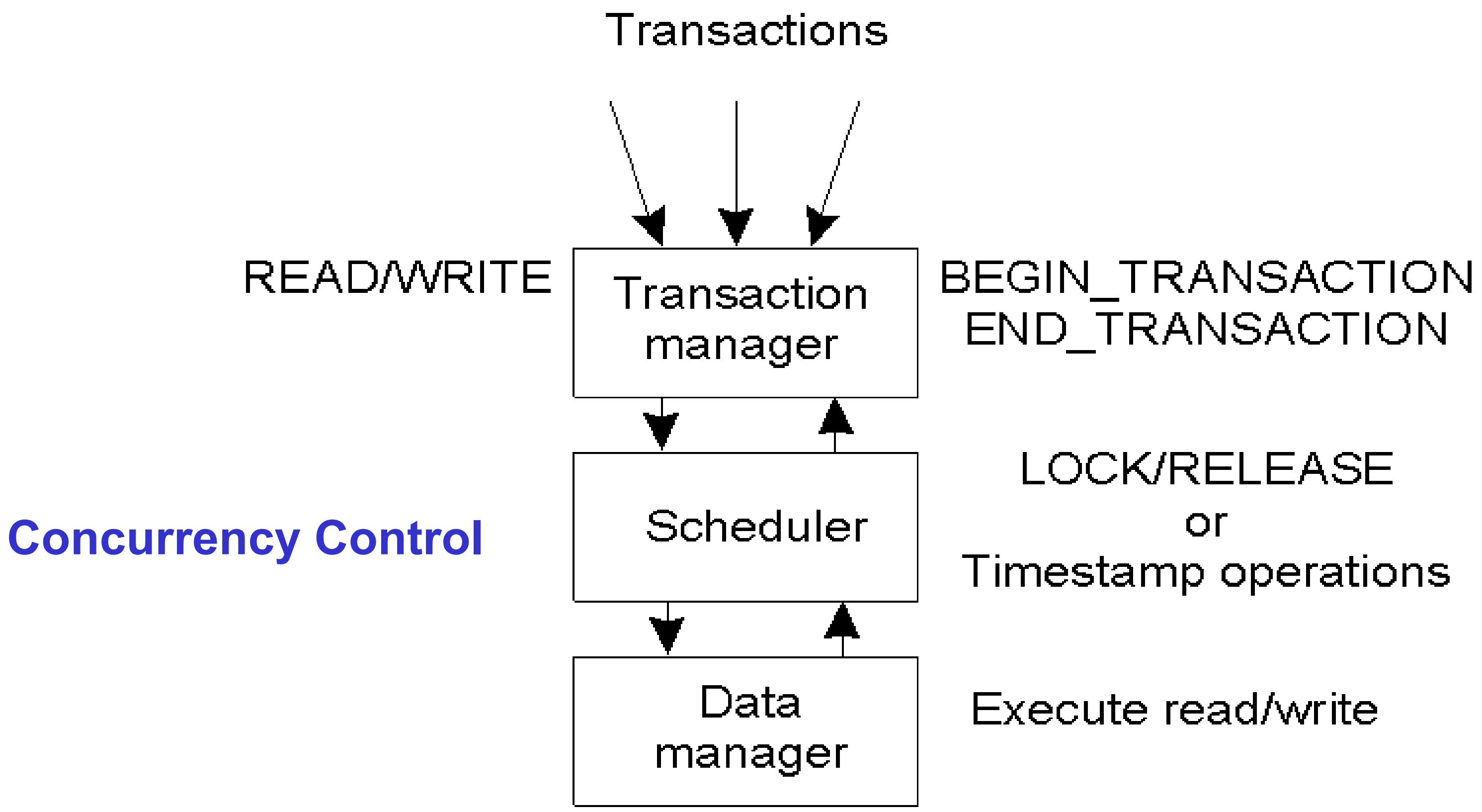
Rollback is executed in case of an abort.

Concurrency Control (1)

- **Concurrency control** is used to assure **SERIALIZABILITY** : concurrent transactions do not interfere with each other.

- The final result should be the same as if the transaction s were executed one after the other in some specific sequential order.

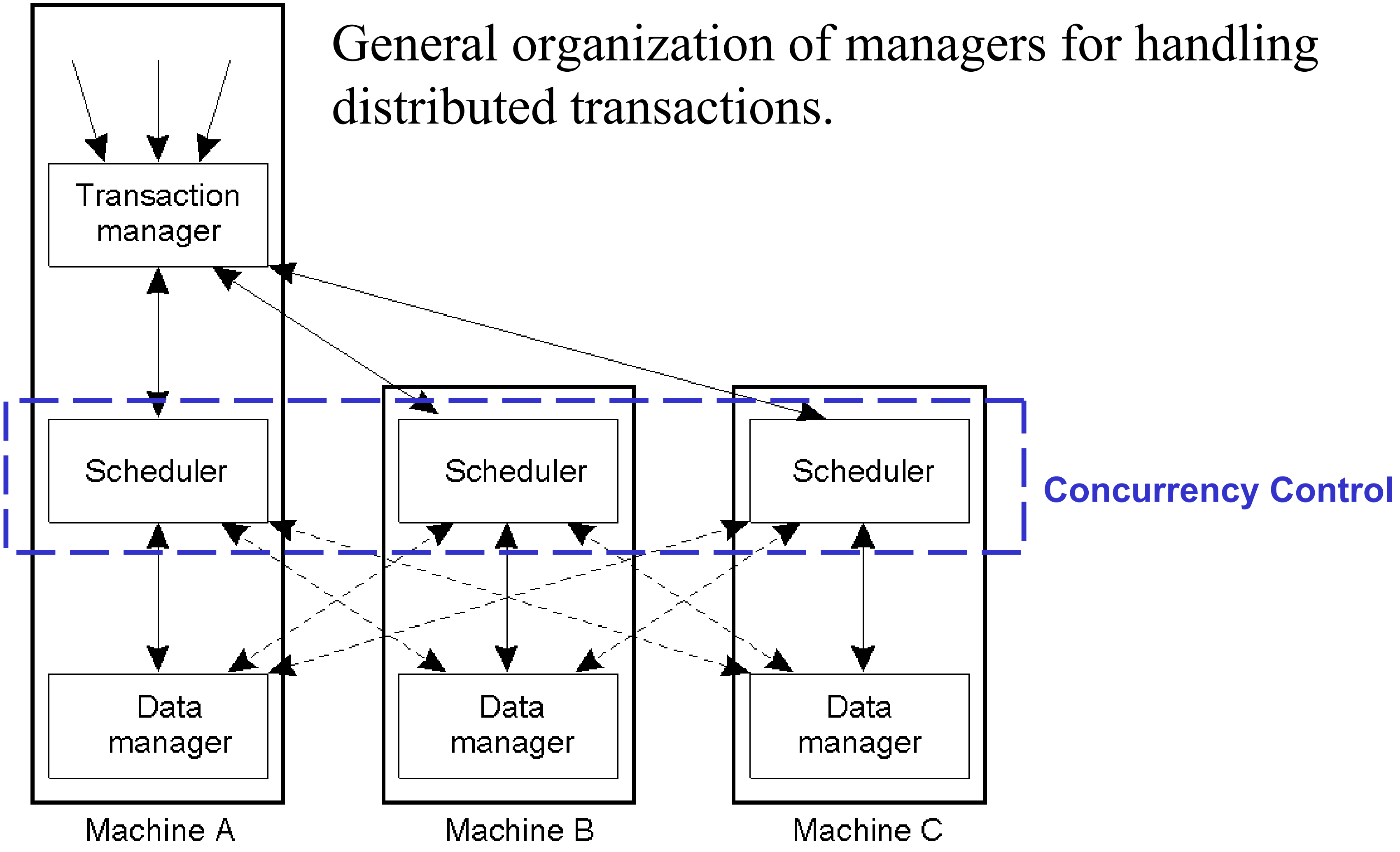
Concurrency Control (2)



General organization of managers for handling transactions.

Concurrency Control (3)

General organization of managers for handling distributed transactions.



Serializability

BEGIN_TRANSACTION
  x = 0;
  x = x + 1;
END_TRANSACTION
 	
(a)
	BEGIN_TRANSACTION
	  x = 0;
	  x = x + 2;
	END_TRANSACTION
	
	(b)
		BEGIN_TRANSACTION
		  x = 0;
		  x = x + 3;
		END_TRANSACTION
		
		(c)

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**

Time -->

(d)

(a) – (c) Three transactions T_1, T_2, and T_3

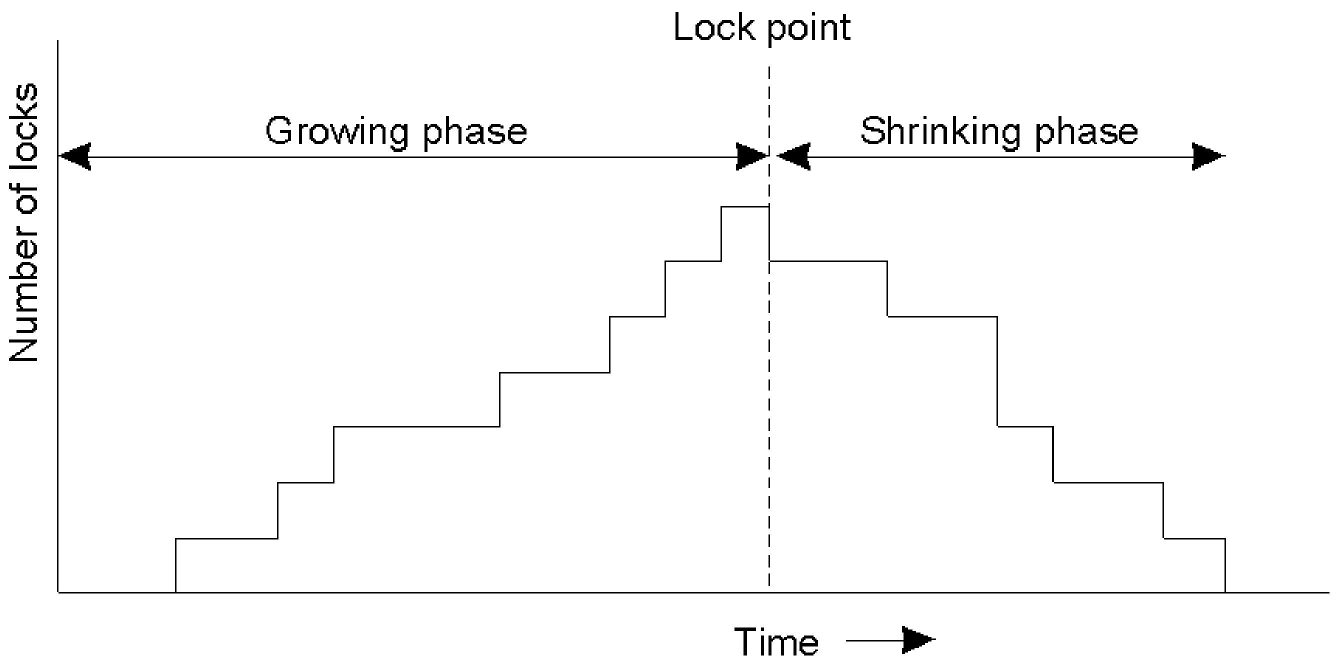
(d) Possible schedules

Conflicting Operations

- Two operations conflict if they operate on same data item and at least one of them is a **write**.
- Concurrency control must find a proper schedule for **conflicting operations** (by a correct **synchronization**).
- Used techniques:
    - *Two-phase locking*
    - *Timestamp ordering*

Two-Phase Locking (1)

- In **Two-phase locking** the scheduler first acquires all the locks it needs during the **growing phase** and then release them in the **shrinking phase**.



Two-Phase Locking (2)

Basic rules

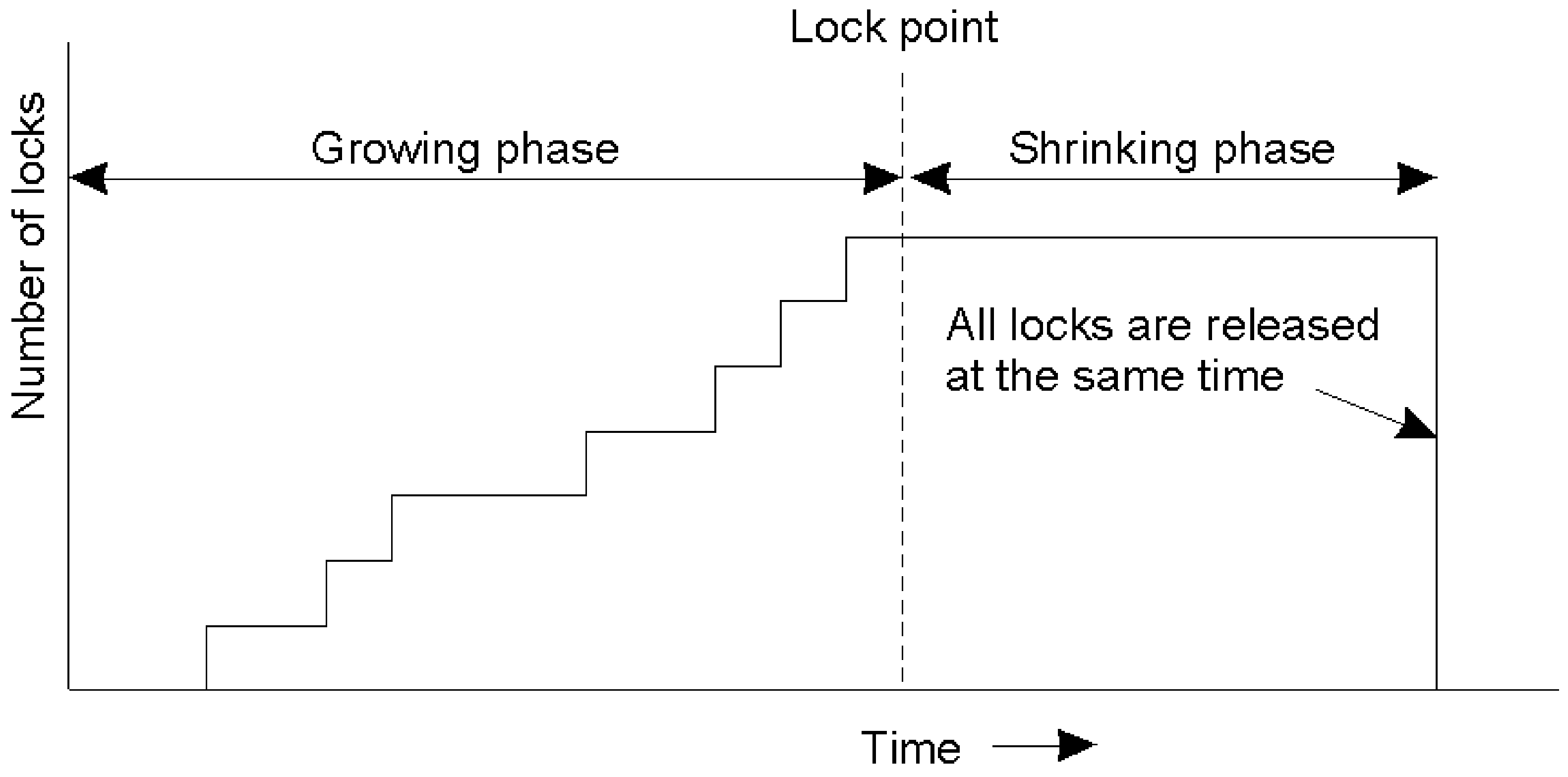
1. When the scheduler receives an operation on *x* is check if the operations conflicts with any other operation for which it already granted a lock. If there no conflict the scheduler grants a lock for *x* and asks the data manager to run the operation.

2. The scheduler will never release the lock for x until the data manager has executed the operation.

3. Once the scheduler released a lock on behalf of T it will never grant another lock on behalf of T.

These three rules guarantee serializability.

Strict Two-Phase Locking



In Strict two-phase locking locks are released when a transaction is finished.

Pessimistic Timestamp Ordering (1)

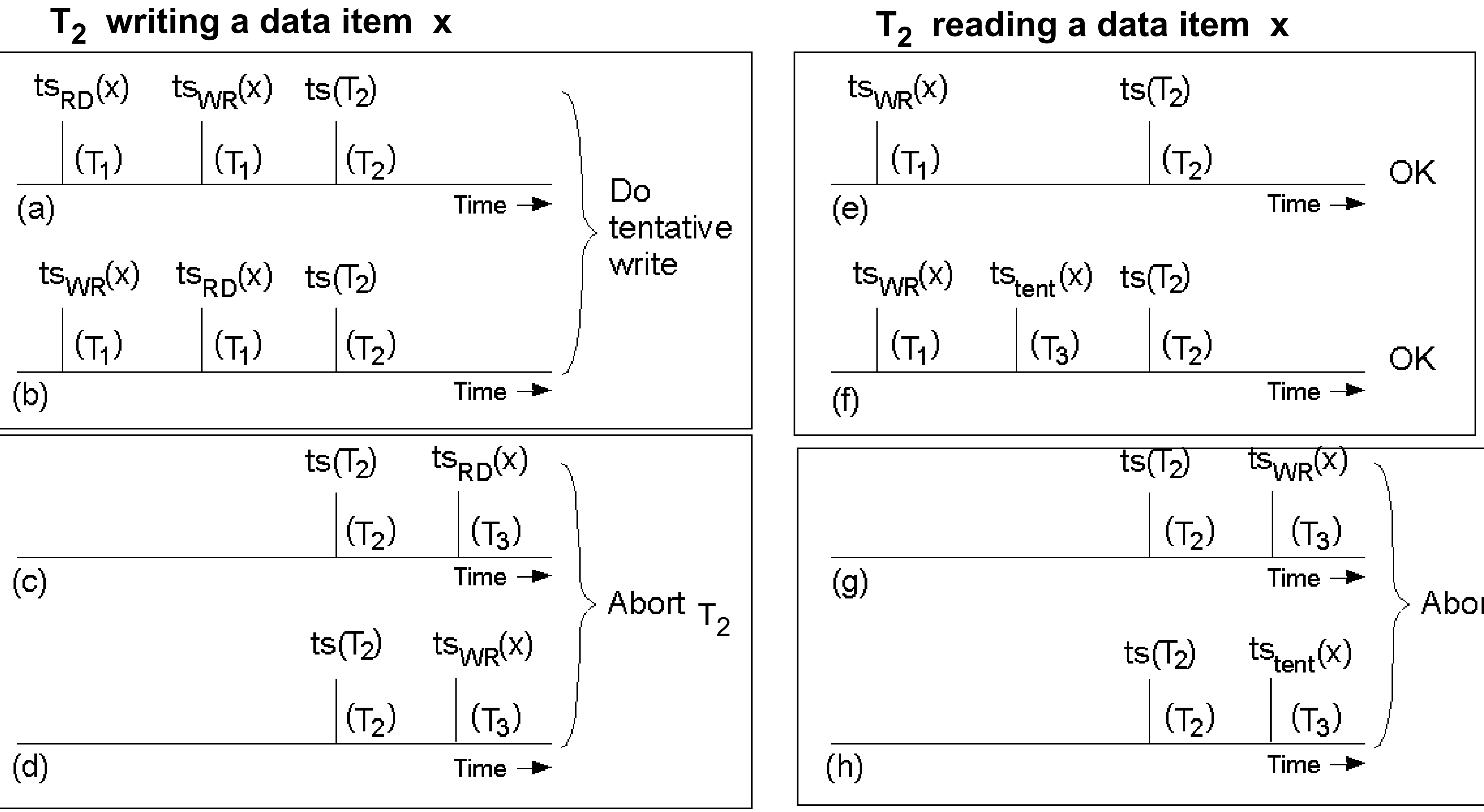
- In concurrency control using timestamps each transaction has a timestamp ***ts(T)***.

- Every data item has a **read timestamp** $\boldsymbol{ts_{RD}(T)}$ and a **write timestamp** $\boldsymbol{ts_{WR}(T)}$

- If two operations conflict the data manager processes the one with the lowest timestamp.

- Timestamps are used to abort operations.

Pessimistic Timestamp Ordering (2)



Examples of concurrency control using timestamps.

2PL and Timestamp Ordering

- Two-phase locking can lead to deadlock, so deadlock detection is needed.
- Timestamp ordering is deadlock free.
- **Optimistic concurrency control** is an alternative approach to pessimistic strategy.