Chapter Five: Synchronization

Objectives of the Chapter

- we discuss
  - the issue of synchronization based on time (actual time and relative ordering)
  - how a group of processes can appoint a process as a coordinator; can be done by means of election algorithms
  - distributed mutual exclusion to protect shared resources from simultaneous access by multiple processes
  - distributed transactions, which also do the same thing, but optimize access through advanced concurrency control mechanisms

5.1 Clock Synchronization

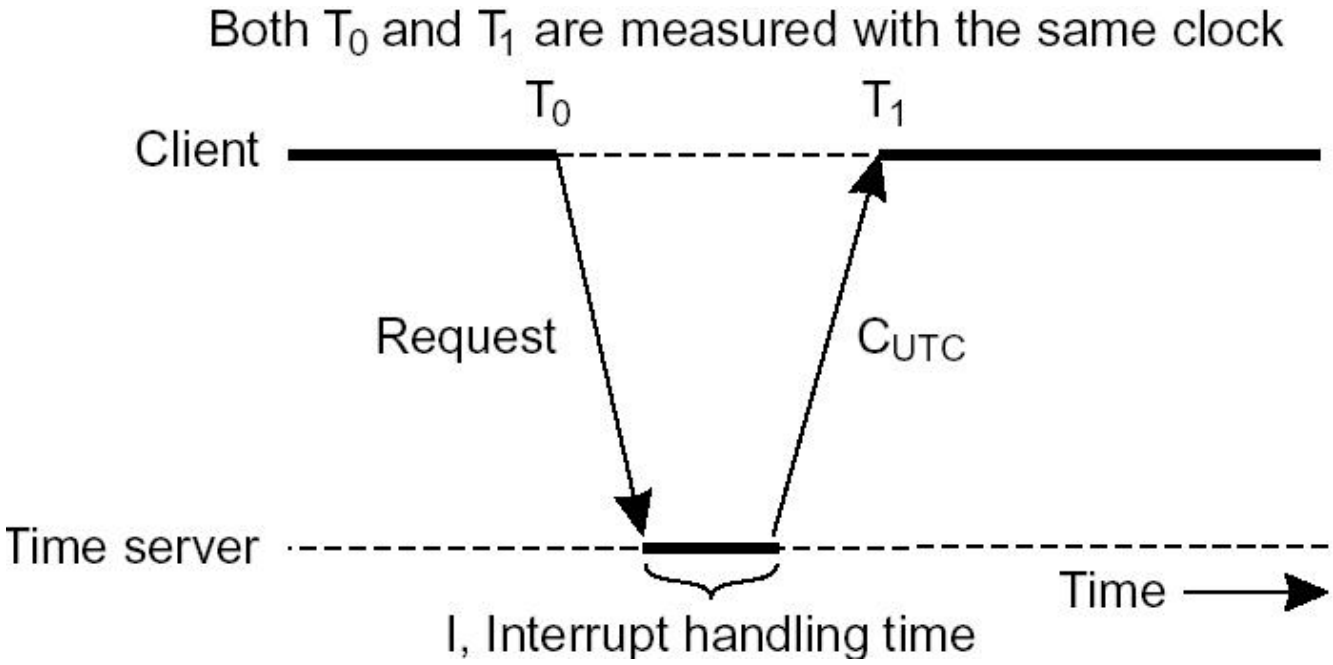
- in centralized systems, time can be unambiguously decided by a system call
- e.g., process A at time t_1 gets the time, say t_A, and process b at time t_2, where $t_1 < t_2$, gets the time, say t_B
  then t_A is always less than t_B
- achieving agreement on time in distributed systems is difficult
- Physical Clocks
  - is it possible to synchronize all clocks in a distributed system?
  - no; even if all computers initially start at the same time, they will get out of synch after some time due to crystals in different computers running at different frequencies, a phenomenon called clock skew

- **how is time actually measured?**
  - **earlier astronomically; based on the amount of time it takes the earth to rotate the sun; 1 solar second = 1/86400th of a solar day (24*3600 = 86400)**
  - **it was later discovered that the period of the earth's rotation is not constant; the earth is slowing down due to tidal friction and atmospheric drag; geologists believe that 300 million years ago there were about 400 days per year; the length of the year is not affected, only the days have become longer**
  - **astronomical timekeeping was later replaced by counting the transitions of the cesium 133 atom and led to what is known as TAI - International Atomic Time**

- TAI was also found to have a problem; 86,400 TAI seconds are behind a solar day by 3 msec, as a result of the day getting longer everyday
- UTC (Universal Coordinated Time) was introduced by having leap seconds whenever the discrepancy between TAI and solar time grows to 800 msec

- Clock Synchronization Algorithms
  - two situations:
    - one machine has a receiver of UTC time, then how do we synchronize all other machines to it
    - no machine has a receiver, each machine keeps track of its own time, then how to synchronize them
  - many algorithms have been proposed

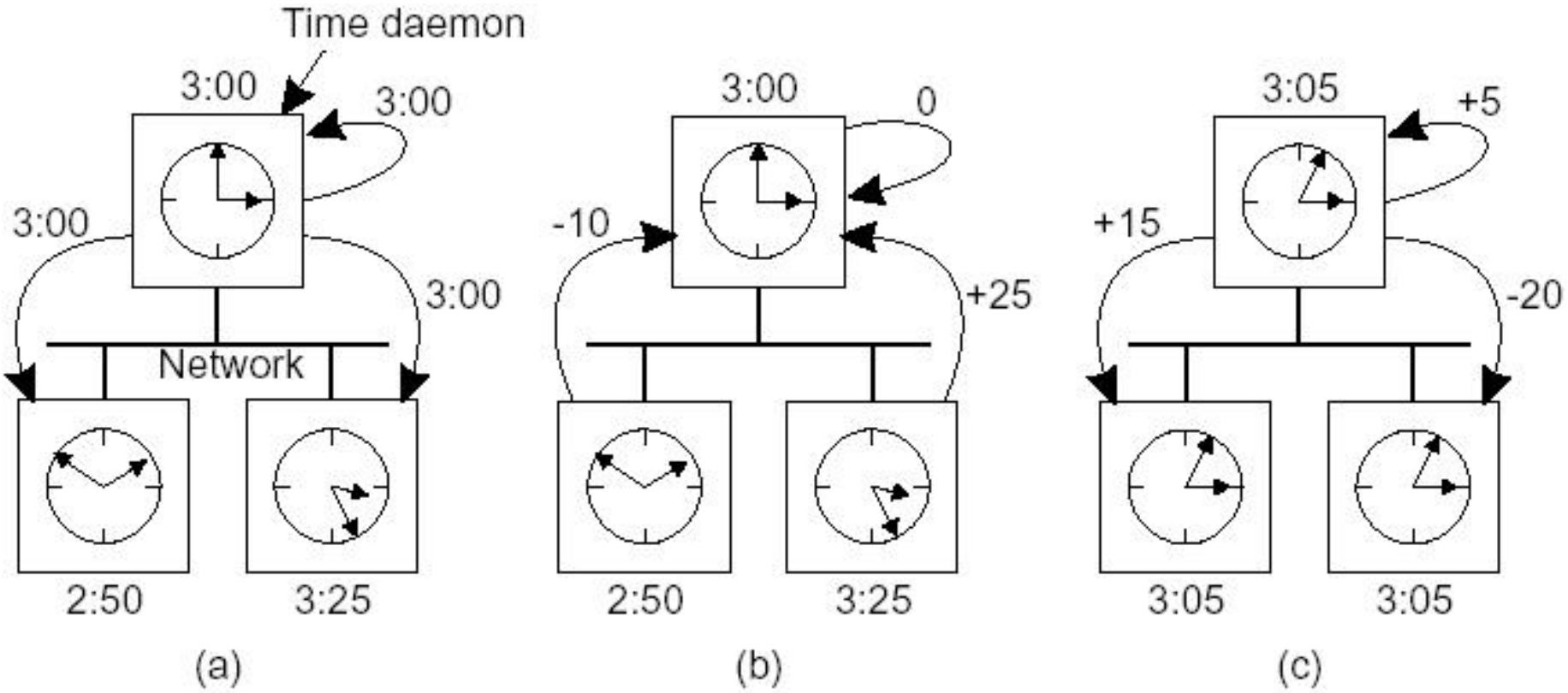
- **Cristian's Algorithm**
  - **suitable when one machine has a UTC receiver; let us call this machine a time server**
  - **every some fixed seconds, each machine sends a message to the time server asking the current time**



getting the current time from a time server

- first approximation: let the client set its clock to C_{UTC}
- problems:
  - time must never run backward
    - solution: introduce change gradually; if a timer is set to generate 100 interrupts per second, it means 10 msec must be added to the time; then make it say 9 (to slow it down) or 11 msec (to advance it gradually)
  - message propagation time
    - solution: try to estimate it as $(T_1-T_0)/2$ and add this to C_{UTC}
    - if we know how long it takes the time server to handle the interrupt, the above can be improved; let the interrupt handling time be I, then the new estimate will be $(T_1-T_0-I)/2$

- **The Berkley Algorithm**
  - in Cristian's algorithm, the time server is passive; only other machines ask it periodically
  - in Berkley UNIX, a time daemon asks every machine from time to time to ask the time
  - it then calculates the average and sends messages to all machines so that they will adjust their clocks accordingly
  - suitable when no machine has a UTC receiver
  - the time daemon's time must be set periodically manually

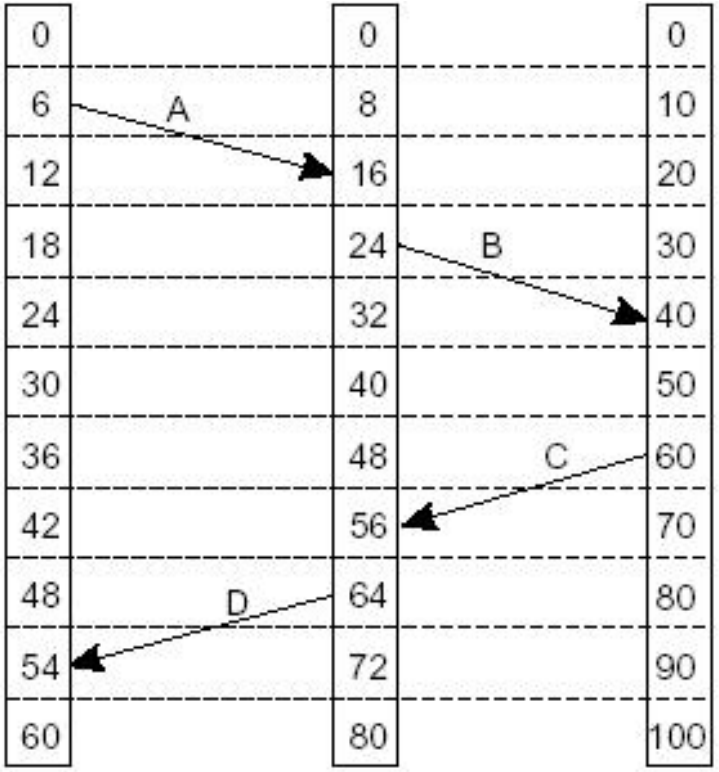


a) *the time daemon asks all the other machines for their clock values*
b) *the machines answer how far ahead or behind the time daemon they are*
c) *the time daemon tells everyone how to adjust their clock*

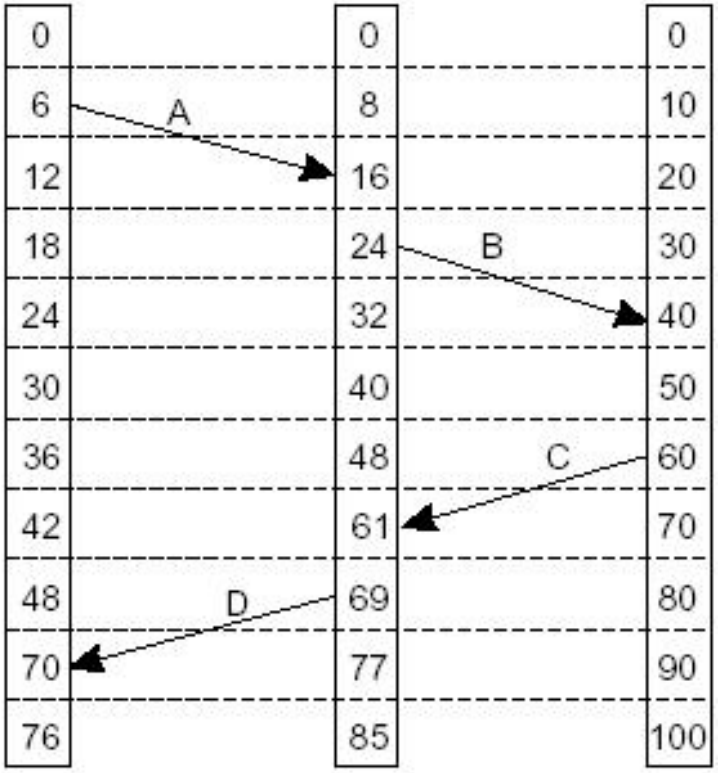
5.2 Logical Clocks

- for some applications, it is sufficient if all machines agree on the same time, rather than with the real time; we need internal consistency of the clocks rather than being close to the real time
- hence the concept of logical clocks
- what matters is the order in which events occur
- Lamport Timestamps
  - Lamport defined a relation called happens before
  - The expression a $\rightarrow$ b is read as "a happens before b"; means all processes agree that first event a occurs, then event b occurs
  - this relation can be observed in two situations
    - if a and b are events in the same process, and a occurs 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 message being received by another process, then a $\rightarrow$ b is also true
    - for every event a, we can assign a time value C(a) on which all processes agree; if a $\rightarrow$ b, then C(a) < C(b)
    - A message can't be received before it is sent

- **Lamport's proposed algorithm for assigning times for processes**
  - **consider three processes each running on a different machine, each with its own clock**
  - **the solution follows the happens before relation; each message carries the sending time; if the receiver's clock shows a value prior to the time the message was sent, the receiver fast forwards its clock to be one more than the sending time**



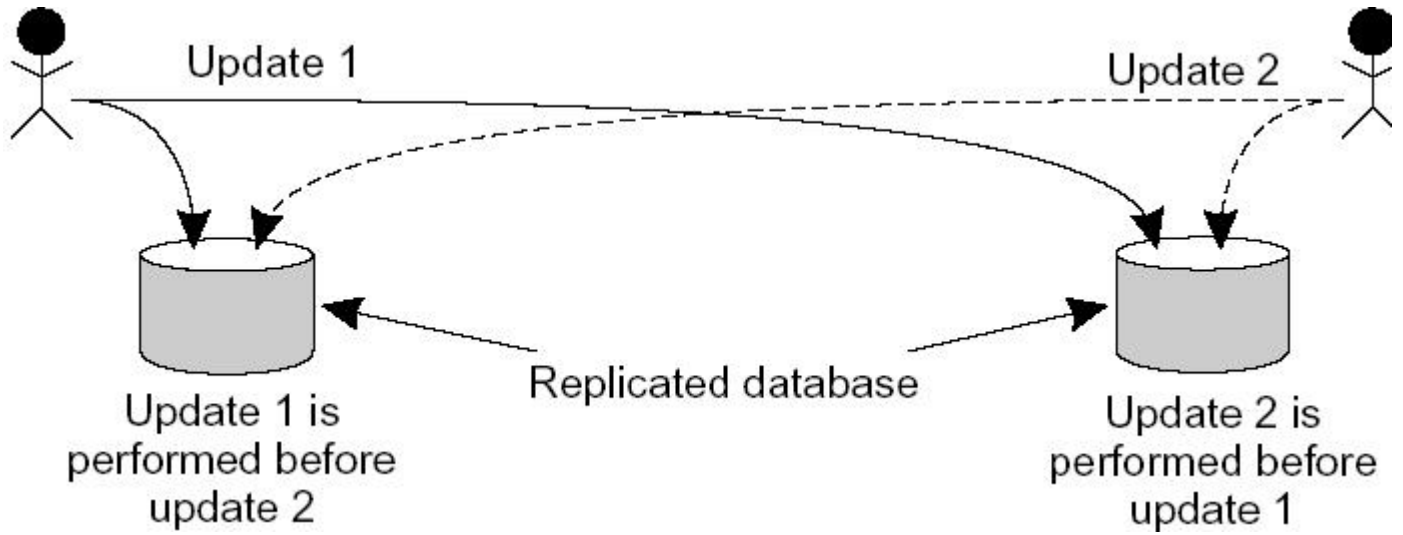
(a)



(b)

a) three processes, each with its own clock; the clocks run at different rates
b) Lamport's algorithm corrects the clocks

- example: **Totally-Ordered Multicasting**
  - assume a bank database replicated across several cities for performance; a query is always forwarded to the nearest copy
  - let us have a customer with $1000 in her account
  - she is in city A and wants to add $100 to her account (update 1)
  - a bank employee in city B initiates an update of increasing accounts by 1 percent interest (update 2)
  - both must be carried out at both copies of the database
  - due to communication delays, if the updates are done in different order, the database will be inconsistent (city A = $1111, city B = $1110)



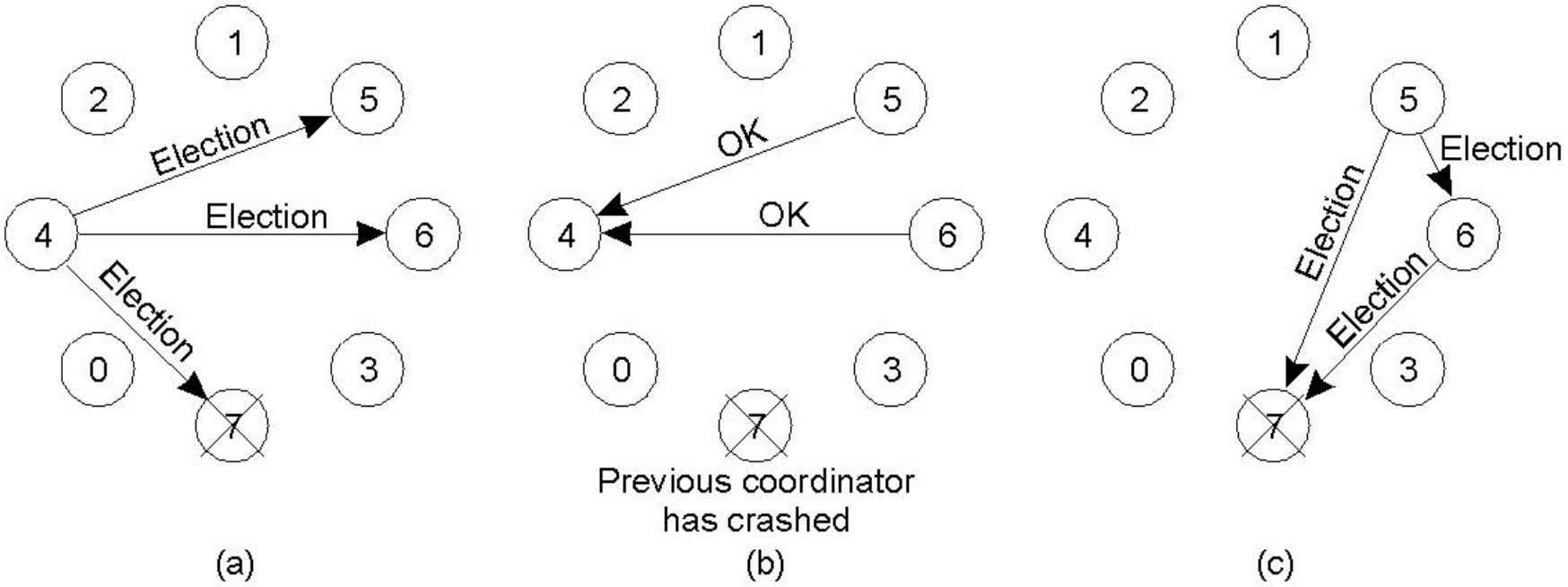
updating a replicated database and leaving it in an inconsistent state

- situations like these require a Totally-Ordered Multicast, i.e., all messages are delivered in the same order to each receiver
- Lamport timestamps can be used to implement totally-ordered multicasts
  - timestamp each message
  - during multicasting, send a message also to the sender
  - assume messages from the same sender are received in the order they are sent and no message is lost
  - all messages are put in a local queue ordered by timestamp
  - each receiver multicasts an acknowledgement
  - then all processes will have the same copy of the local queue
  - a process can deliver a queued message to the application it is running only when that message is at the top of the queue and has been acknowledged by each other process

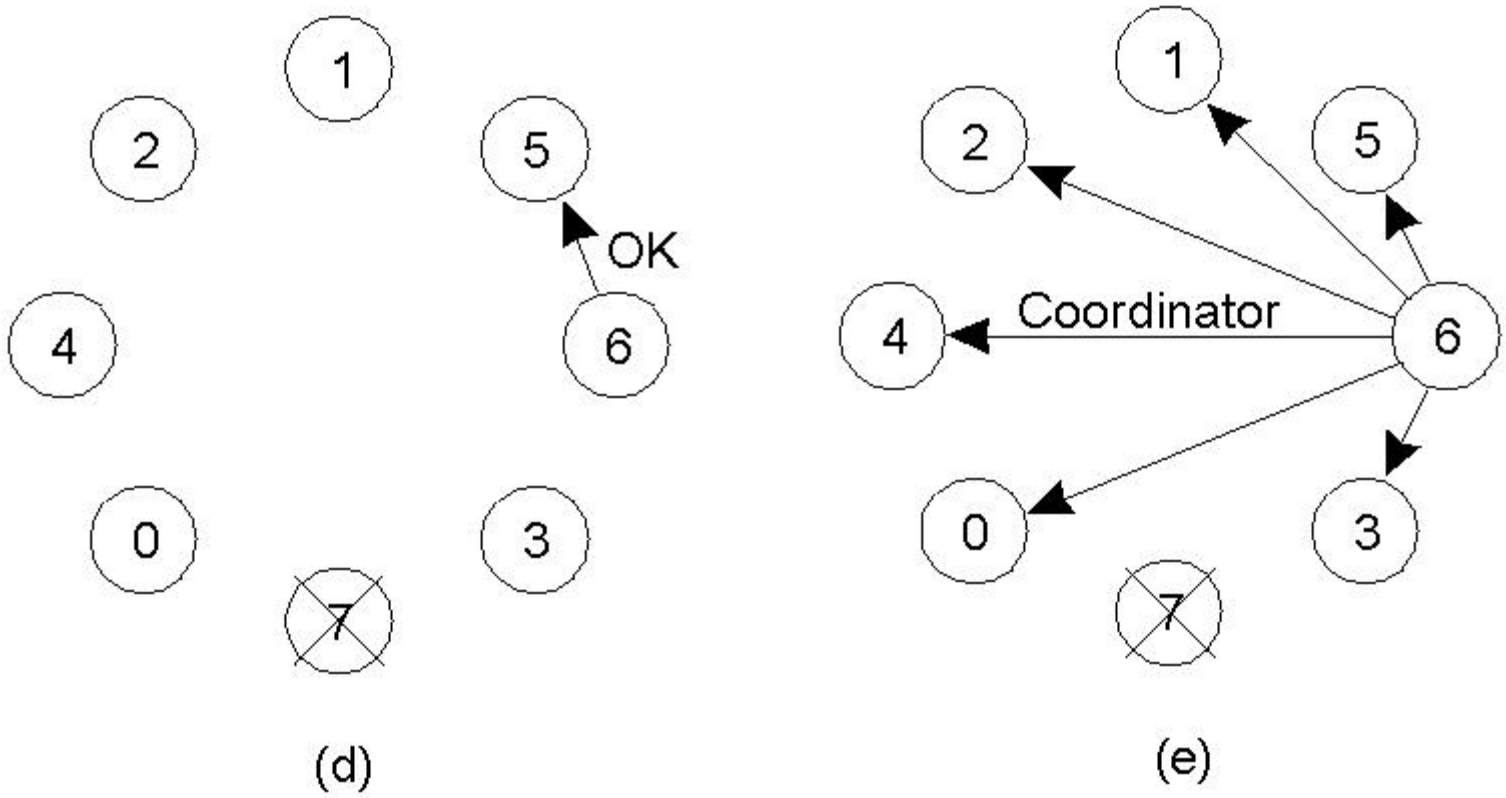
5.4 Election Algorithms

- Devised by Garcia- Molina (1982)
- there are situations where one process must act as a coordinator, initiator, or perform some special task
- assume that
  - each process has a unique number
  - every process knows the process number of every other process, but not the state of each process (which ones are currently running and which ones are down)
- election algorithms attempt to locate the process with the highest process number
- two algorithms: the Bully algorithm and the Ring algorithm

- **The Bully Algorithm** (the biggest person wins)
  - when a process (say P4) notices that the coordinator is no longer responding to requests, it initiates an election as follows
    1. P4 sends an ELECTION message to all processes with higher numbers (P5, P6, P7)
       - if a process gets an ELECTION message from one of its lower-numbered colleagues, it sends an OK message to the sender and holds an election
    2. if no one responds, P4 wins the election and becomes a coordinator
    3. if one of the higher-ups answers, this new process takes over



a) *Process 4 holds an election*
b) *Process 5 and 6 respond, telling 4 to stop*
c) *Now 5 and 6 each hold an election*



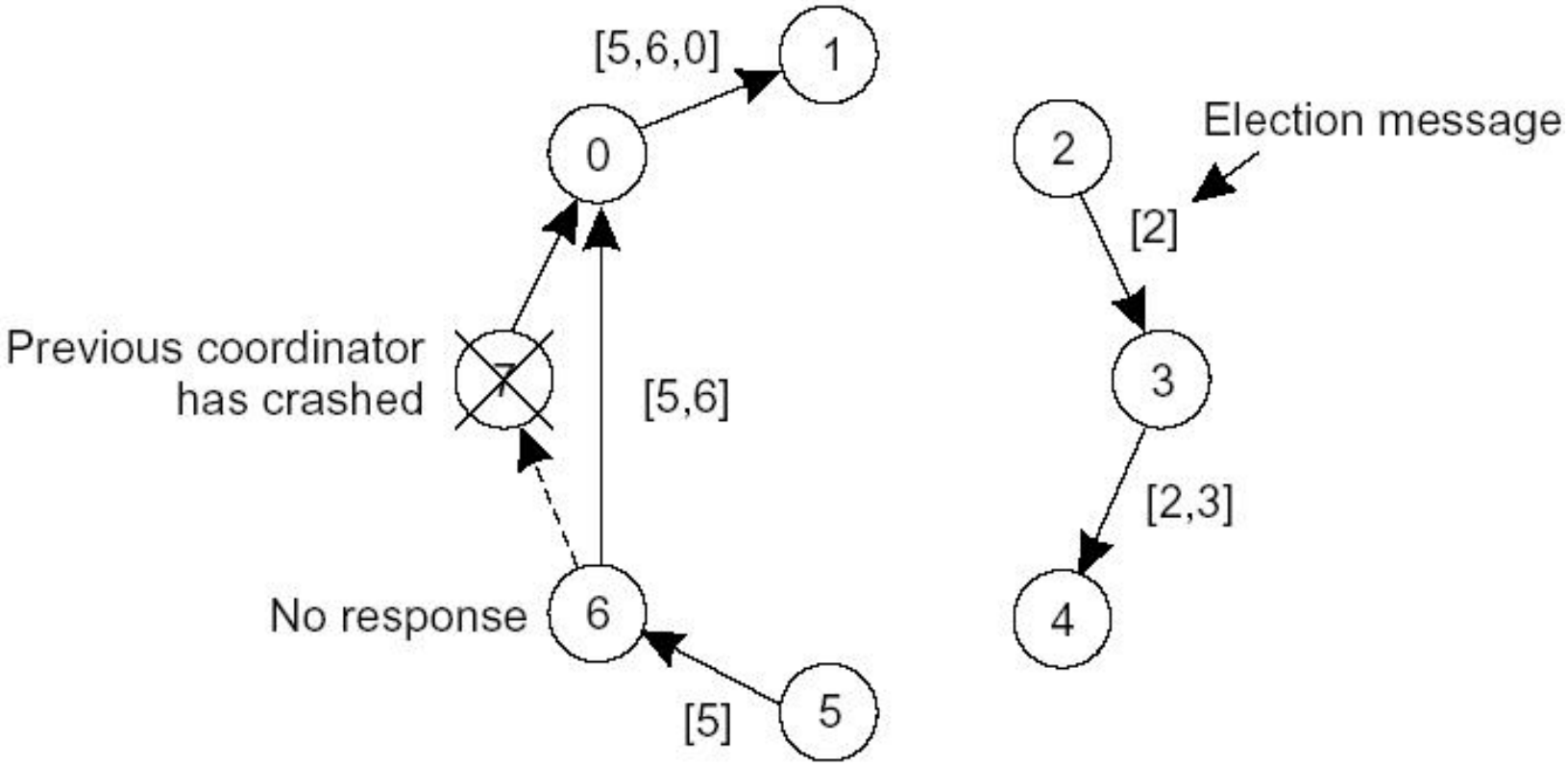
d) Process 6 tells 5 to stop

e) Process 6 wins and tells everyone

- **the last one will send a message to all processes telling them that it is a boss**
- **if a process that was previously down comes back, it holds an election**

- **The Ring Algorithm**
  - based on the use of a ring
  - assume that processes are physically or logically ordered, so that each process knows who its successor is
  - when a process notices that the coordinator is not functioning,
    - it builds an ELECTION message containing its own process number and sends it to its successor
    - if the successor is down, the sender goes to the next number, or the next until a running process is found
    - at each step, the sender adds its own process number to the list in the message making itself a candidate
    - eventually the message gets back to the originating process; the process with the highest number is elected as coordinator, the message type is changed to COORDINATOR and circulated once again to announce the coordinator and who the members of the ring are

- **e.g., processes 2 and 5 simultaneously discover that the previous coordinator has crashed**
- **they build an ELECTION message and a coordinator is chosen**
- **although the process is done twice, there is no problem, only a little bandwidth is consumed**



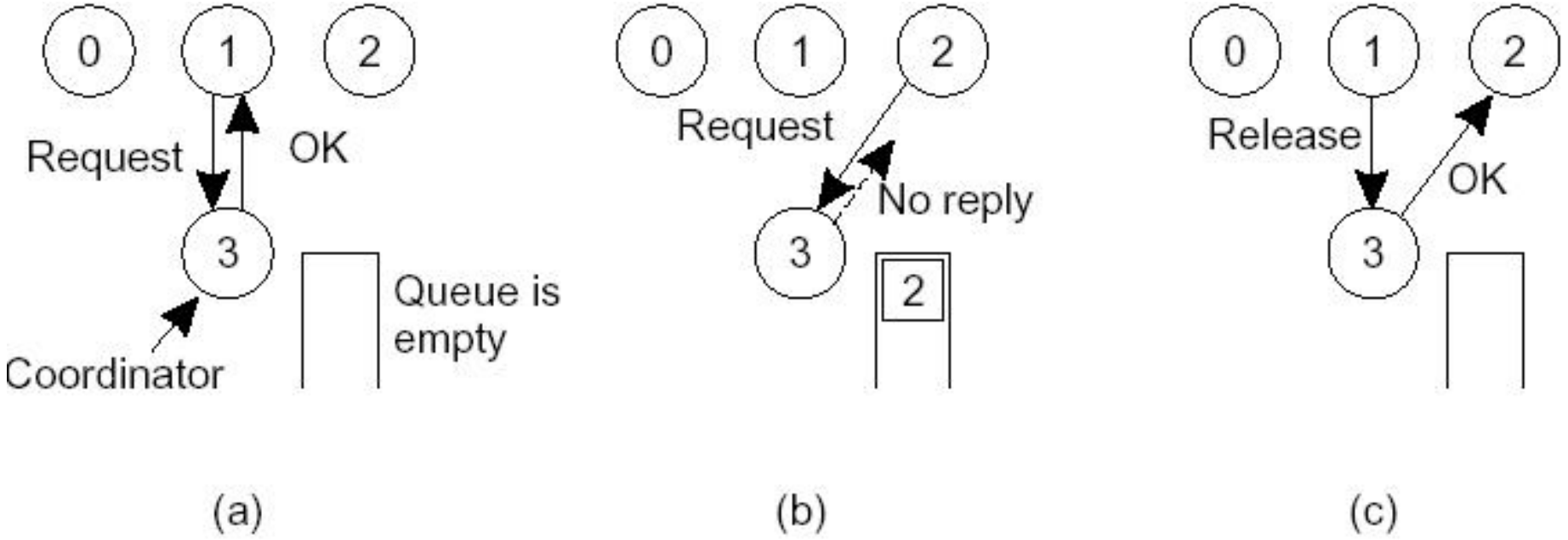
election algorithm using a ring

5.5 Mutual Exclusion

- when a process has to read or update shared data (quite often to use a shared resource such as a printer, a file, etc.), it first enters a critical region to achieve mutual exclusion
- in single processor systems, critical regions are protected using semaphores, monitors, and similar constructs
- how are critical regions and mutual exclusion implemented in distributed systems?
- three algorithms: centralized, distributed, and token ring

- A Centralized Algorithm
  - a coordinator is appointed and is in charge of granting permissions
  - three messages required: request, grant, release

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 (could also send a no message; is implementation dependent), but queues process 2; process 2 is blocked
c) when process 1 exits the critical region, it tells the coordinator, which then replies to 2

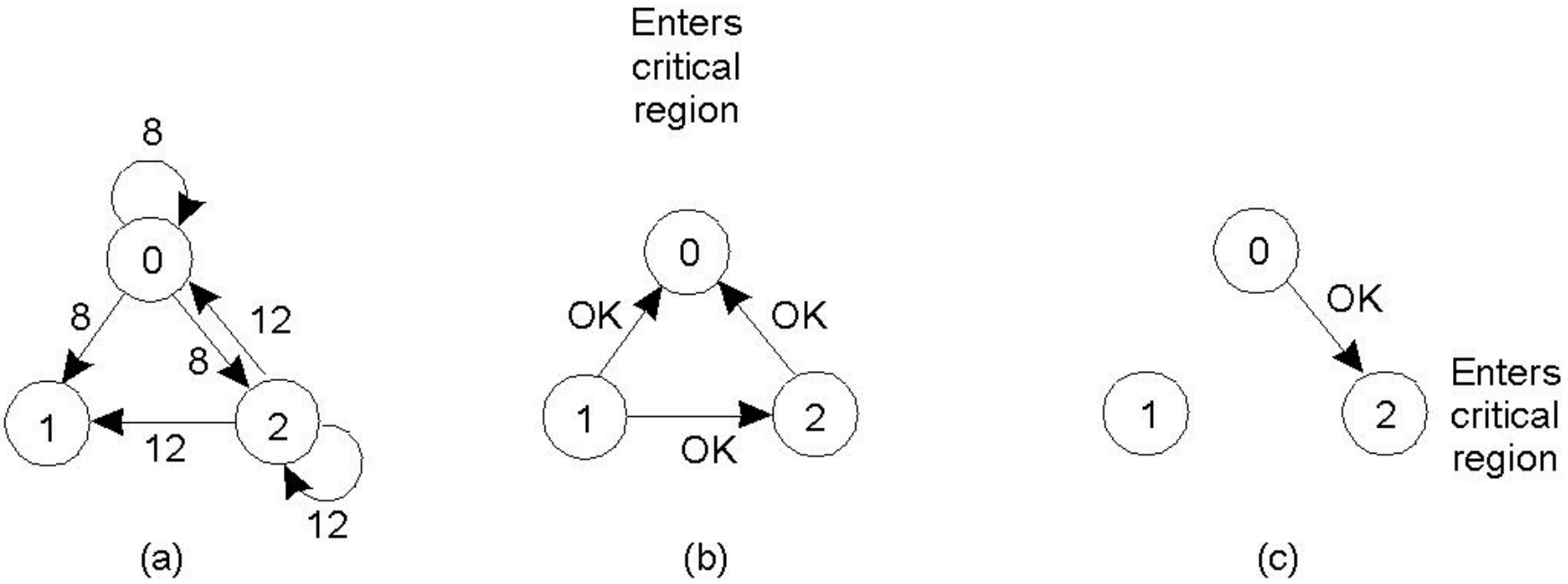


- the algorithm
  - guarantees mutual exclusion
  - is fair - first come first served
  - no starvation
  - easy to implement; requires only three messages: request, grant, release
- shortcoming: a failure of the coordinator crashes the system (specially if processes block after sending a request); it also becomes a performance bottleneck

- A Distributed Algorithm
  - assume that there is a total ordering of all events in the system, such as using Lamport timestamps
  - when a process wants to enter a critical region it builds a message (containing name of the critical region, its process number, current time) and sends it to everyone including itself
  - the sending of a message is assumed to be reliable; i.e., every message is acknowledged

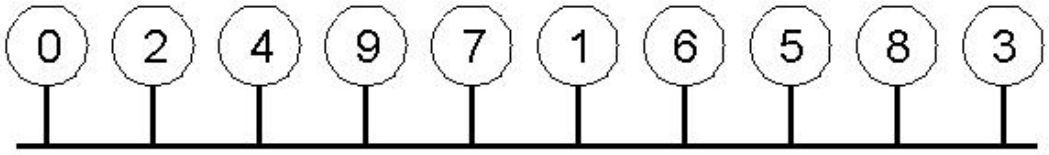
- when a process receives a request message
  1. if the receiver is not in a critical region and does not want to enter it, it sends back an OK message to the sender
  2. if the receiver is already in the critical region, it does not reply; instead it queues the request
  3. if the receiver wants to enter the critical region but has not yet done so, it compares the timestamp of the message it sent with the incoming one; the lowest wins; if the incoming message is lower, it sends an OK message; otherwise it queues the incoming message and does not do anything
- when the sender gets a reply from all processes, it may enter into the critical region
- when it finishes it sends an OK message to all processes in its queue

- **is it possible that two processes can enter into the critical region at the same time if they initiate a message at the same time? NO**
  - **a) two processes (0 and 2) 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 message, so process 2 can now enter into its critical region**

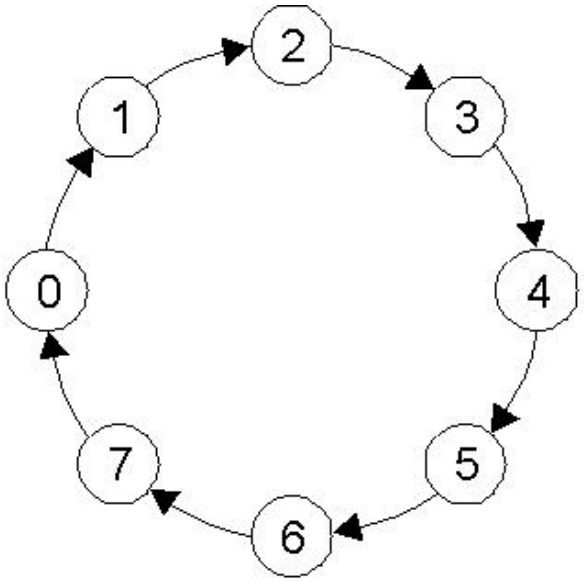


- mutual exclusion is guaranteed
- the total number of messages to enter a critical region is increased to 2(n-1), where n is the number of processes
- no single point of failure; unfortunately n points of failure
- we also have n bottlenecks
- hence, it is slower, more complicated, more expensive, and less robust than the previous one; but shows that a distributed algorithm is possible

- **A Token Ring Algorithm**
  - assume a bus network (e.g., Ethernet); no physical ordering of processes required but a logical ring is constructed by software



(a)



(b)

a) an unordered group of processes on a network
b) a logical ring constructed in software

- when the ring is initialized, process 0 is given a token
- the token circulates around the ring
- a process can enter into a critical region only when it has access to the token
- it releases it when it leaves the critical region
- it should not enter into a second critical region with the same token
- mutual exclusion is guaranteed
- no starvation
- problems
  - if the token is lost, it has to be generated, but detecting that it is lost is difficult
  - if a process crashes
    - can be modified to include acknowledging a token so that the token can bypass a crashed process
    - in this case every process has to maintain the current ring configuration

- **A comparison of the three algorithms (assumes only point-to-point communication channels are used)**

Algorithm	Messages per entry/exit	Delay before entry (in message times)	Problems
Centralized	3 - efficient	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

5.6 Distributed Transactions and Concurrency Control

- transactions are also used to protect shared resources, usually data (similar to mutual exclusion)
- but transactions do much more
  - they also allow a process to access and modify multiple data items as a single atomic operation
  - if a process backs out halfway during the transaction, everything must be restored to the point just before the transaction started (all-or-nothing)
- The Transaction Model
  - the model for transactions comes from the world of business
  - a supplier and a retailer negotiate on
    - price
    - delivery date
    - quality
    - etc.

- until the deal is concluded they can continue negotiating or one of them can terminate
- but once they have reached an agreement they are bound by law to carry out their part of the deal
- transactions between processes is similar with this scenario
- e.g., assume the following banking operation
  - withdraw an amount x from account 1
  - deposit the amount x to account 2
- what happens if there is a problem after the first activity is carried out?
- group the two operations into one transaction; either both are carried out or neither
- we need a way to roll back when a transaction is not completed

- **special primitives are required to program transactions, supplied either by the underlying distributed system or by the language runtime system**
- **exact list of primitives depends on the type of application**

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

- **e.g. reserving a seat from White Plains to Malindi through JFK and Nairobi airports**

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)

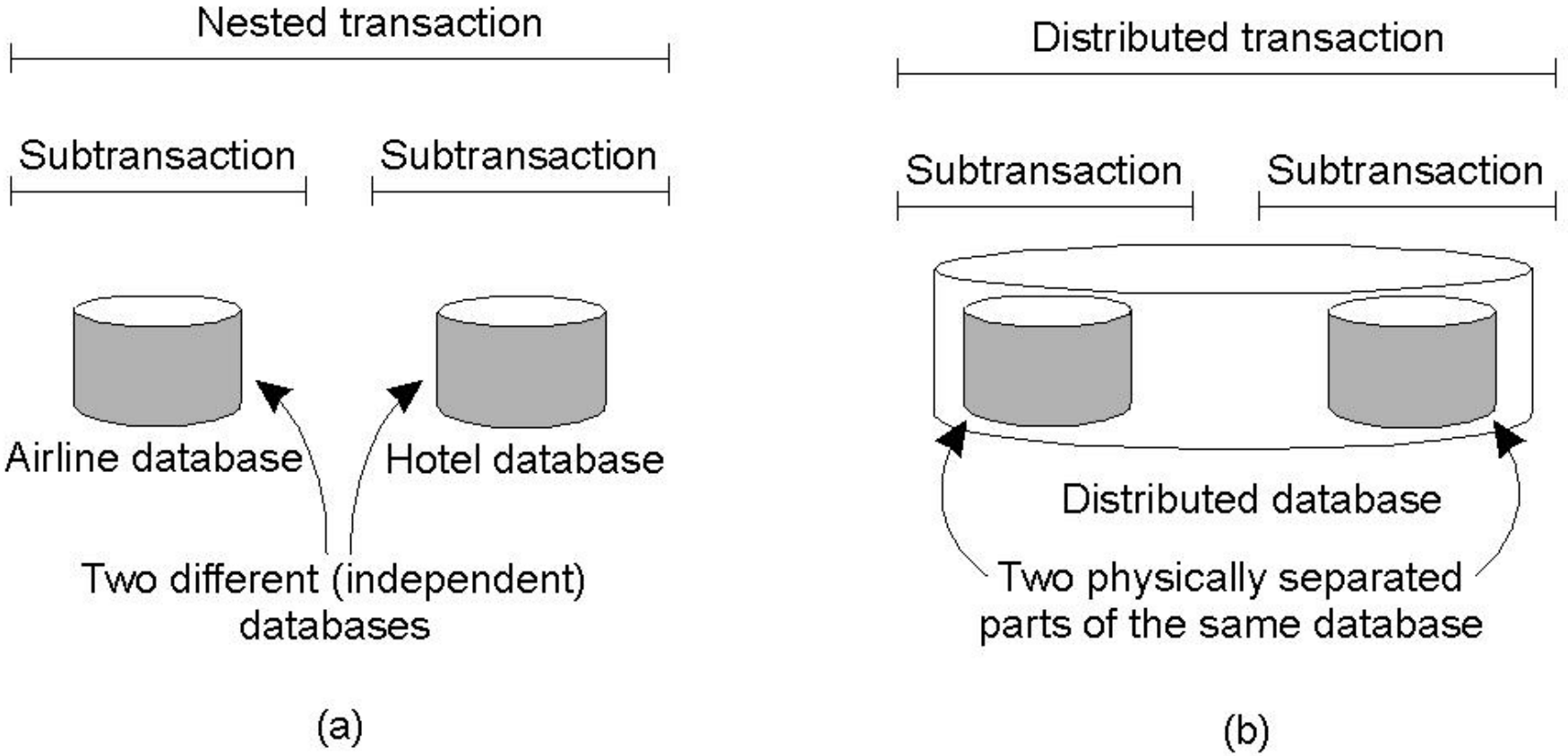
a) transaction to reserve three flights commits

b) transaction aborts when third flight is unavailable

- properties of transactions, often referred to as ACID
  1. Atomic: to the outside world, the transaction happens indivisibly; a transaction either happens completely or not at all; intermediate states are not seen by other processes
  2. Consistent: the transaction does not violate system invariants; e.g., in an internal transfer in a bank, the amount of money in the bank must be the same as it was before the transfer (the law of conservation of money); this may be violated for a brief period of time, but not seen to other processes
  3. Isolated or Serializable: concurrent transactions do not interfere with each other; if two or more transactions are running at the same time, the final result must look as though all transactions run sequentially in some order; see later in this Chapter
  4. Durable: once a transaction commits, the changes are permanent

- **Classification of Transactions**
  - a transaction could be flat, nested or distributed
  - **Flat Transaction**
    - consists of a series of operations that satisfy the ACID properties
    - simple and widely used but with some limitations
      - do not allow partial results to be committed or aborted
        - i.e., atomicity is also partly a weakness
        - in our airline reservation example, we may want to accept the first two reservations and find an alternative one for the last
      - some transactions may take too much time

- **Nested Transaction**
  - constructed from a number of subtransactions; it is logically decomposed into a hierarchy of subtransactions
  - the top-level transaction forks off children that run in parallel, on different machines; to gain performance or for programming simplicity
  - each may also execute one or more subtransactions
  - permanence (durability) applies only to the top-level transaction; commits by children should be undone
- **Distributed Transaction**
  - a flat transaction that operates on data that are distributed across multiple machines
  - problem: separate algorithms are needed to handle the locking of data and committing the entire transaction; see later in this Chapter about distributed locking and in Chapter 7 for distributed commit



a) a nested transaction
b) a distributed transaction

- **Implementation**
  - for simplicity, consider transactions on a file system
  - if each process executing a transaction just updates the file it uses in place, transactions will not be atomic and changes will not vanish if the transaction aborts
  - so we need some other implementation
  - two methods: private workspace and writeahead log

  - **Private Workspace**
    - a process is given a private workspace at the instant it begins a transaction
    - the workspace contains all the files to which it has access
    - all its reads and writes are stored in the private workspace
    - the actual files are not affected until a process commits

- **Writeahead Log**
  - files are actually modified in place, but before any block is changed, a record is written to a log telling
    - which transaction is making the change
    - which file and block is being changed, and
    - what the old and new values are
  - only after the log has been written successfully is the change made to the file
  - if the transaction aborts, the log can be used to back up to the original state, starting at the end and going backwards, called rollback

- example

```
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

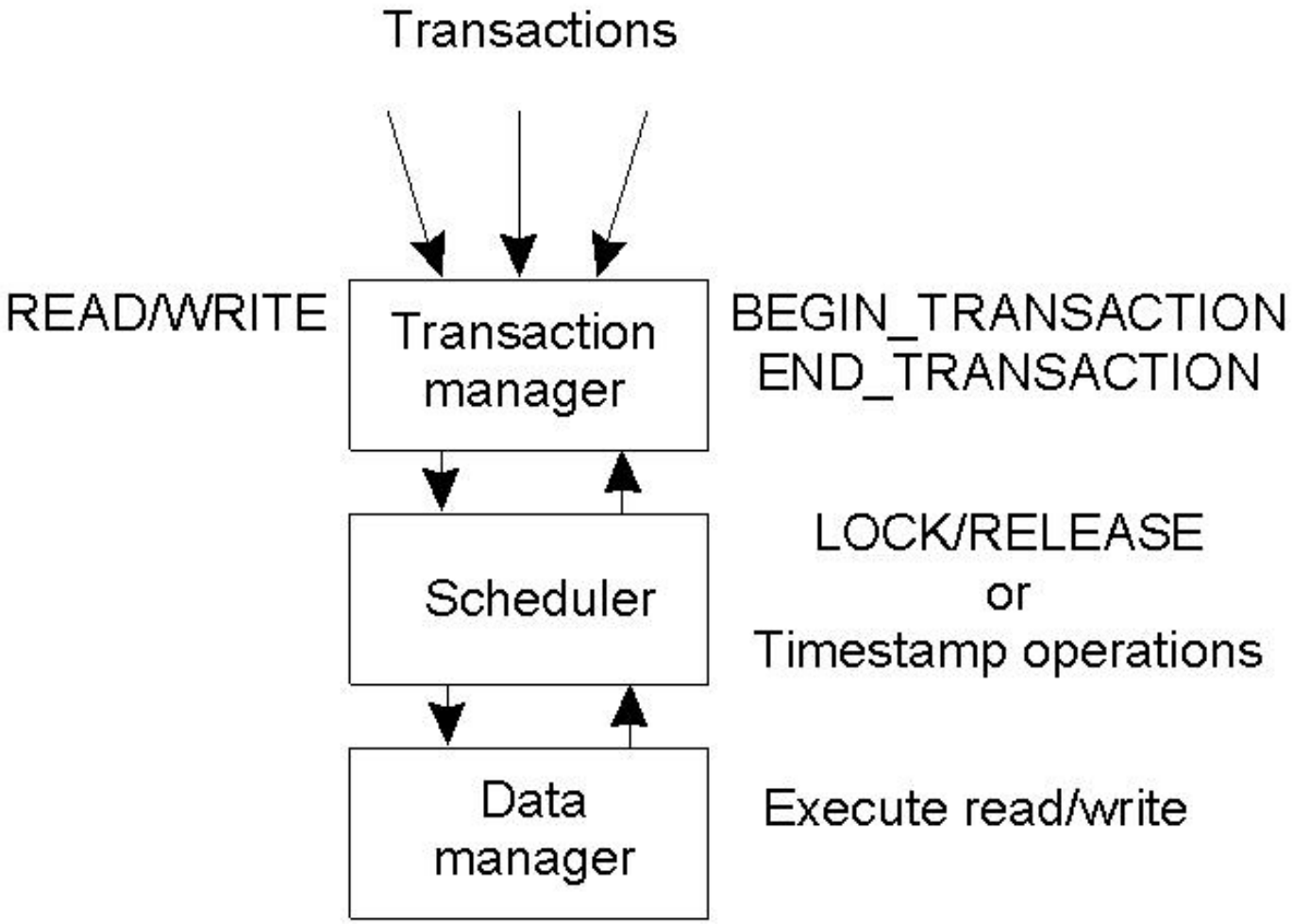
b) – d) the log before each statement is executed

- **Concurrency Control**
  - how to properly control the execution of concurrent transactions; transactions that are executed at the same time on shared data
  - the goal of concurrency control is to allow several transactions to be executed simultaneously, but the collection of data items (such as database records or files) must be left in a consistent state
  - achieved by giving transactions access to data items in a specific order whereby the final result is the same as if all transactions had run sequentially
  - for concurrency control, we can have three different managers organized in a layered fashion (non distributed case)

- **Transaction manager**
  - responsible for guaranteeing atomicity of transactions
  - processes transaction primitives by transforming them into scheduling requests for the scheduler

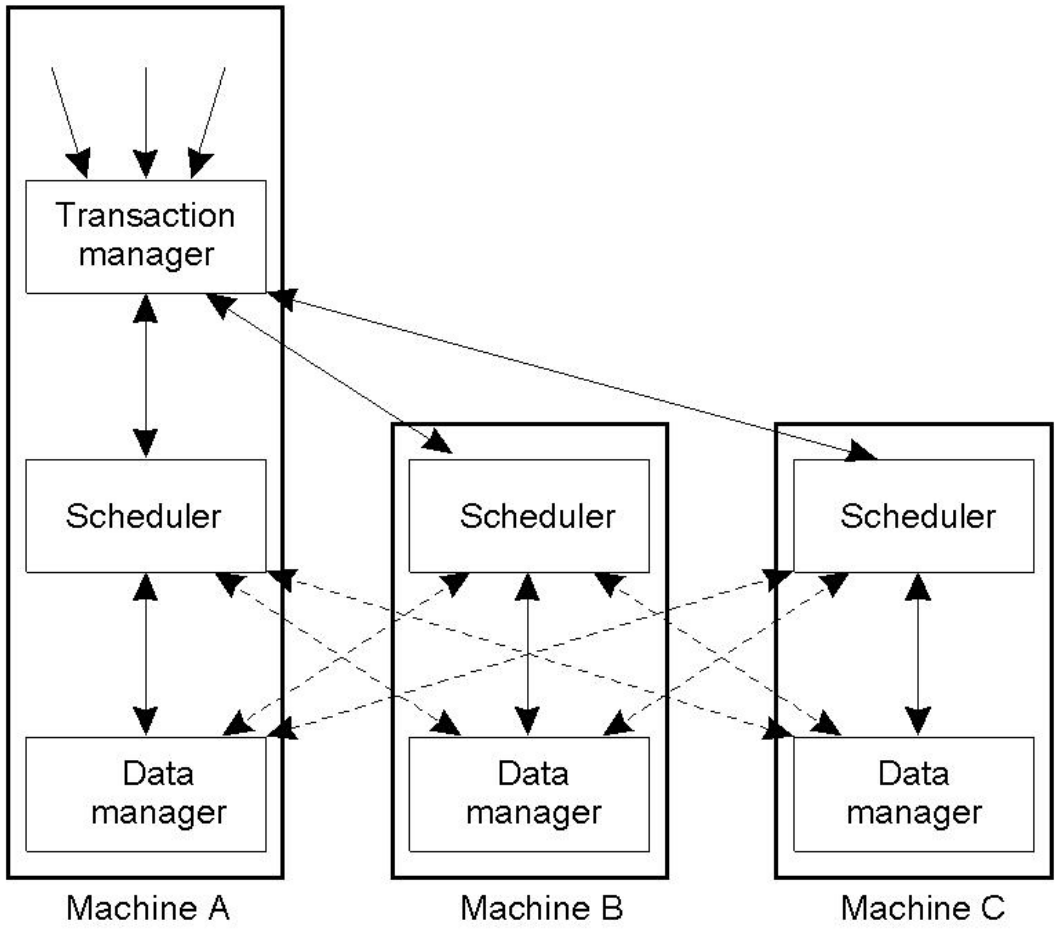
- **Scheduler**
  - responsible for properly controlling concurrency
  - it determines which transaction is allowed to pass a read or write operation to the data manager and at which time
    - the scheduling may be based on locks or timestamps

- **Data manager**
  - performs the actual read and write operations
  - not concerned which transaction is doing the read or write



general organization of managers for handling transactions (non distributed)

- **the above can be adopted for distributed case; each site has its own scheduler and data manager, but a common transaction manager**



general organization of managers for handling distributed transactions

- **Serializability**
  - concurrency control algorithms guarantee that multiple transactions can be executed simultaneously while still being isolated at the same time ➲ the final result should be the same as if the transactions were executed one after the other in some specific order
  - consider the following example where three transactions are executed simultaneously by three separate processes: if the three transactions were run sequentially, the final value of x would be 1, 2, or 3, depending on which one run last; x can be any shared variable, a file, or any other entity

```
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)

- three example schedules (orders)
  - schedule 1 is serialized (a→b→c); schedule 2 is not serialized, but is still legal (result is 3); schedule 3 is not serialized and not legal (result is 5)

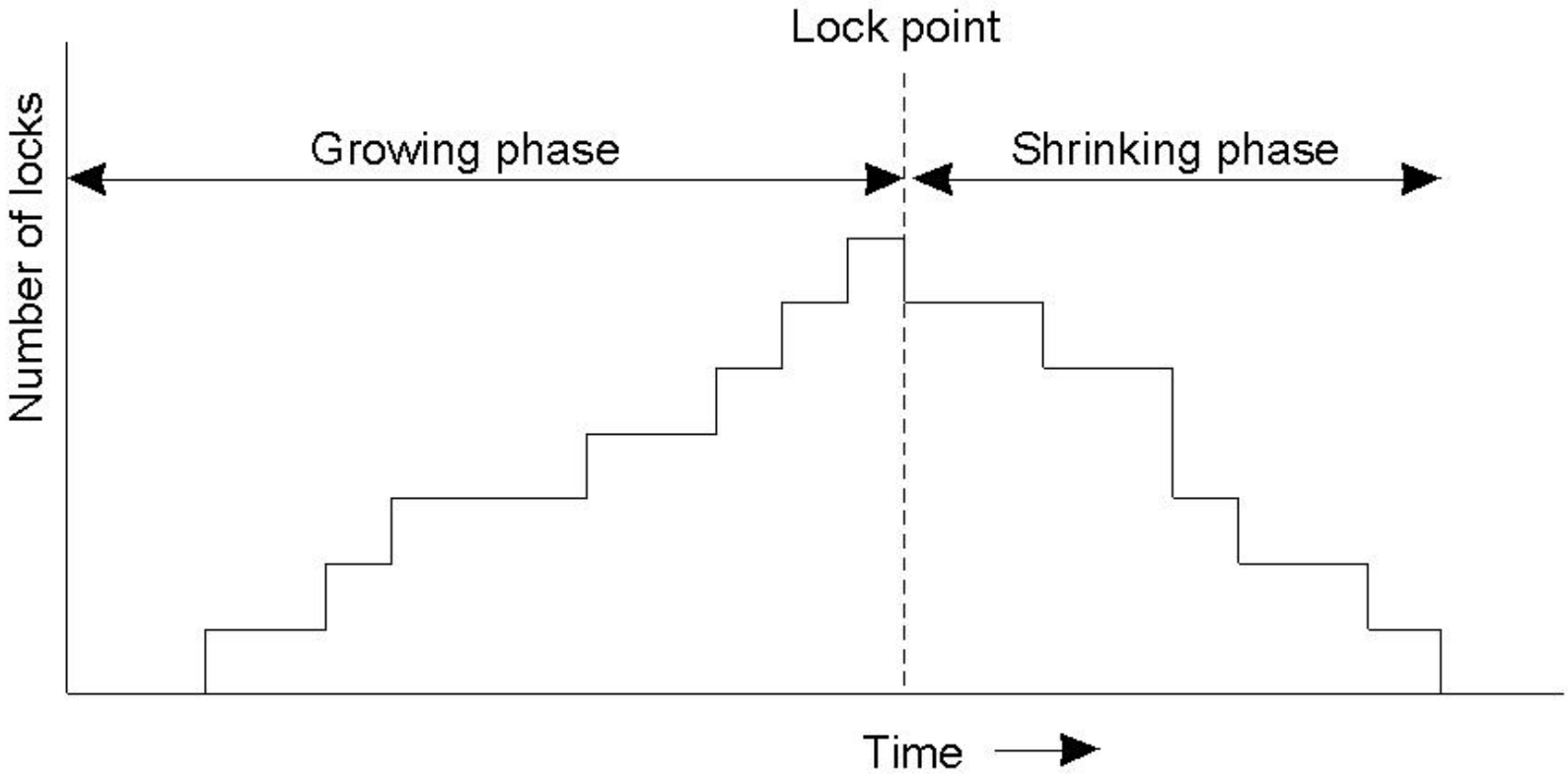
Schedule 1	x = 0; x = x + 1; x = 0; x = x + 2; x = 0; x = x + 3;	Legal
Schedule 2	x = 0; x = 0; x = x + 1; x = x + 2; x = 0; x = x + 3;	Legal
Schedule 3	x = 0; x = 0; x = x + 1; x = 0; x = x + 2; x = x + 3;	Illegal

d) Possible Schedules

- for schedules and concurrency control, we don't need to know the nature of computations; only that the value of x is being changed
- hence, transactions can be represented as a series of read and write operations on specific data items

- then the aim of concurrency control is to properly schedule conflicting operations
- two operations conflict if they operate on the same data item, and if at least one of them is a write operation (read-write conflict or write-write conflict)
- how are read and write operations synchronized? or how are concurrency control algorithms classified?
  - synchronization can take place either through mutual exclusion mechanisms on shared data (locking) or explicitly ordering operations using timestamps
- two types of concurrency control
  - pessimistic approach: if something can go wrong, it will resolve conflicts before they happen; 2PL (and strict 2PL) and pessimistic timestamp ordering
  - optimistic approach: assume nothing will go wrong; synchronization takes place at the end of a transaction; one or more transactions are forced to abort if conflicts occurred; optimistic timestamp ordering

- **Two-Phase Locking (2PL)**
  - the oldest and most widely used concurrency control algorithm is locking
  - when a process needs to access a data item, it requests the scheduler to grant it a lock, when completed it requests the scheduler to release it
  - the scheduler has to use an algorithm to make sure that only valid schedules result (serializable schedules)
  - one such algorithm is two-phase locking - a pessimistic approach
  - it has two phases
    - Growing Phase: the scheduler first acquires all the locks it needs
    - Shrinking Phase: the scheduler releases the locks



two-phase locking

- the following three rules must be obeyed
  1. the scheduler receives an operation oper(T,x) from the transaction manager; checks if it conflicts with another operation for which a lock is granted; if so oper(T,x) is delayed, else it is granted a lock on data item x and passes the operation to the data manager
  2. the scheduler will never release a lock for x until the data manager acknowledges it has performed the operation
  3. once the scheduler has released a lock on behalf of T, it will never grant another lock on behalf of T, no matter for which data item T is requesting a lock; such an attempt is considered a programming error and T is aborted
- it is proved that two-phase locking is serializable; and why it is widely used

- implementing the basic two-phase locking in a distributed system
- assume data are distributed across multiple machines
  - Centralized 2PL
    - a single site is responsible for granting and releasing locks
    - each transaction manager communicates with this centralized lock manager and with its local data manager
  - Primary 2PL
    - each data item is assigned a primary copy; the lock manager on that copy's machine is responsible for granting and releasing locks (distributed locking)
  - Distributed 2PL
    - assume data are replicated
    - the schedulers at each machine are responsible in granting and releasing locks as well as forwarding operations to the local data managers

- **Pessimistic Timestamp Ordering**
  - to each transaction T, assign a timestamp *ts*(T) when it starts (timestamps must be unique, say using Lamport's algorithm)
  - every operation that is part of T is also timestamped with *ts*(T)
  - every data item x in the system also has
    - a read timestamp $ts_{RD}(x)$ - set to the timestamp of the transaction that most recently read x, and
    - a write timestamp $ts_{WR}(x)$ - set to the timestamp of the transaction that most recently changed x
  - if two operations conflict, the data manager processes the one with the lowest timestamp

- **Optimistic Timestamp Ordering**
  - allow all transactions to access data items on the assumption that conflicts are rare
  - to check if there were conflicts, keep track of which data items have been read and written
  - at the point of committing, check all other transactions to see if any of its items have been changed since the transaction started; if so, the transaction is aborted, else, it is committed
  - Implemented based on private workspaces
  - advantages
    - it is also deadlock free
    - allows maximum parallelism since no process has to wait for a lock
  - the disadvantage is the failure of transactions


THANKS