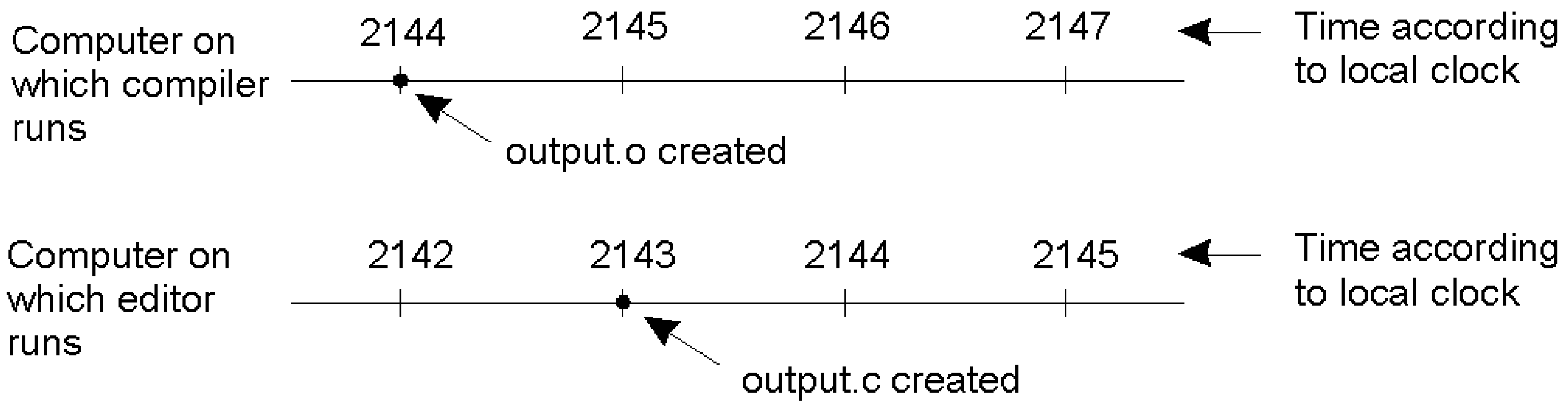
Synchronization

Why Synchronize?

- To *control access* to a single, shared resource.
- To agree on the *ordering of events*.
- Synchronization in Distributed Systems is much more difficult than in uniprocessor systems:

1. Synchronization based on “Actual Time”.
2. Synchronization based on “Relative Time”.
3. Synchronization based on Co-ordination (with Election Algorithms).
4. Distributed Mutual Exclusion.
5. Distributed Transactions.

Clock Synchronization – The Problem


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 on another remote device.

In the above e.g., **MAKE** will **not call the compiler** for the newer version of the "output.c" program, even though it is "newer".

Clock Synchronization

- Synchronization based on “Actual Time”.
  - time sync is really easy on a uniprocessor system.

- Achieving agreement on time in a DS is not trivial.

Question: is it even possible to synchronize all the clocks in a Distributed System?

With multiple computers, “clock skew” ensures that no two machines have the same value for the “current time”. But, how do we measure time?

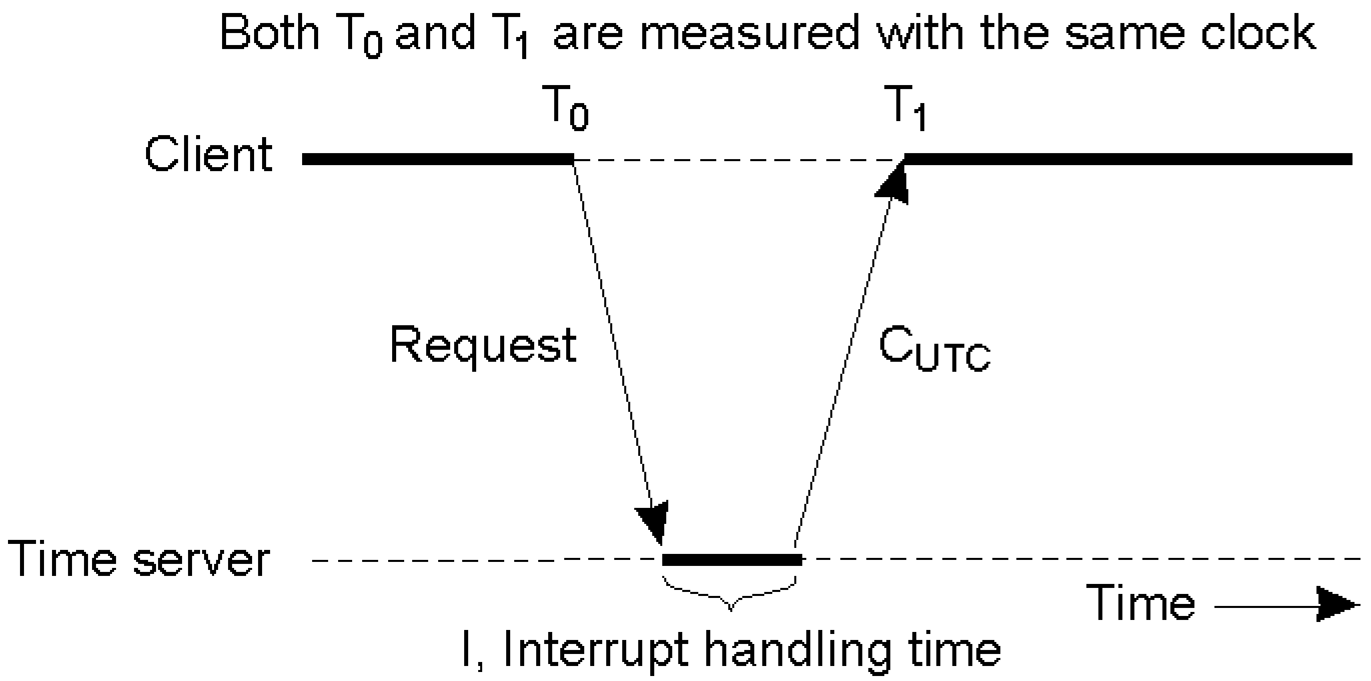
How Do We Measure Time?

- Turns out that we have only been measuring time accurately with a “global” atomic clock since *Jan. 1st, 1958* (the “beginning of time”).
  - Refer to pages 243-246 of the textbook for all the details – it’s quite a story.

Bottom Line: measuring time is not as easy as one might think it should be.

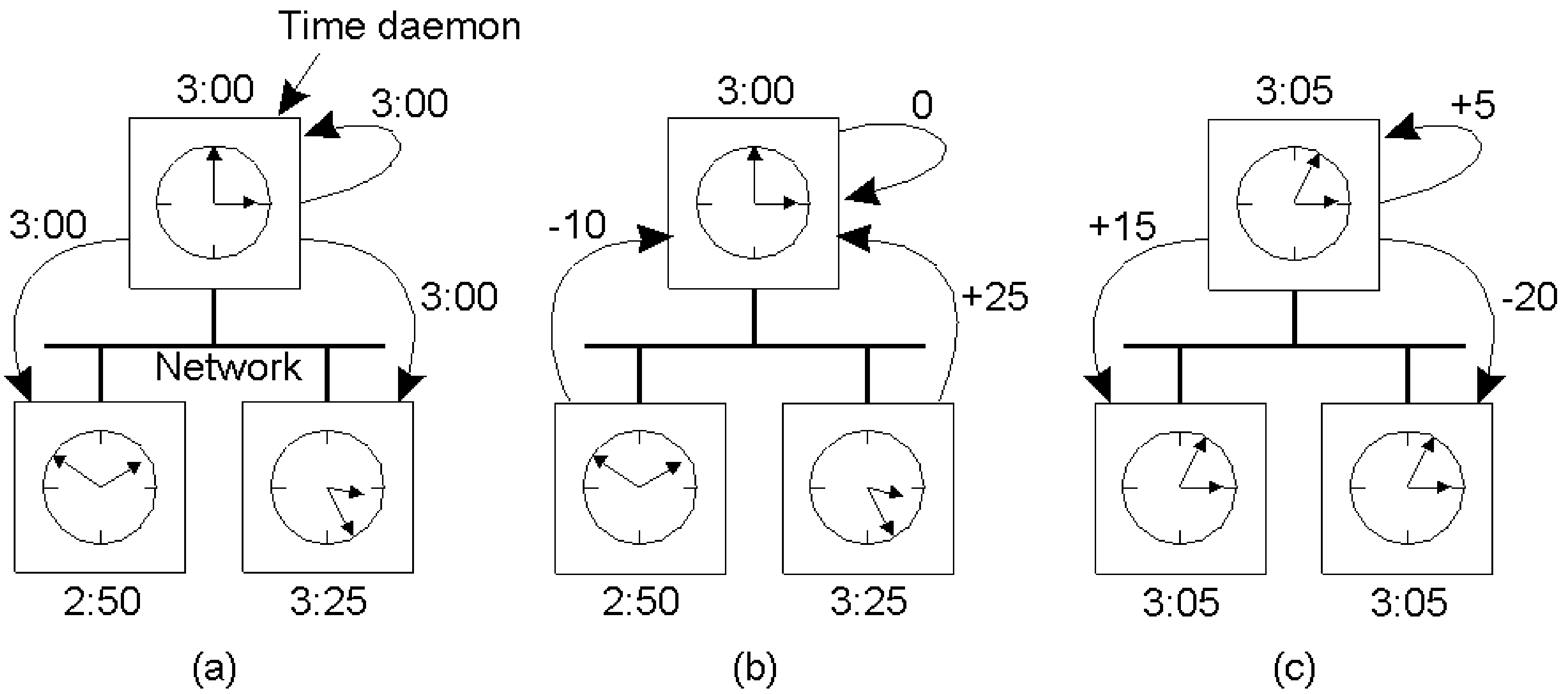
- Algorithms based on the current time (from some **Physical Clock**) have been devised for use within a DS.

Clock Sync. Algorithm: Cristian's


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" - using periodic client requests:
  - **Major problem** if time from time server is less than the client – resulting in time running backwards on the client! (cannot happen – time does not go backwards).
  - **Minor problem** results from the delay introduced by the network request/response latency.

The Berkeley Clock Sync. Algorithm



a) The time daemon asks all the other machines for their clock values using a **polling mechanism** (and providing the “current” time).

b) The machines answer, indicating how they differ from the time sent.

c) The time daemon tells everyone how to adjust their clock based on a calculation of the “**average time value**”.

Clocks that are running fast are slowed down. Clocks running slow jump forward.

Other Clock Sync. Algorithms

- Both Cristian’s and the Berkeley Algorithm are centralised algorithms.

- Decentralised algorithms also exist, and the Internet’s Network Time Protocol (NTP) is the best known and most widely implemented.

  - NTP can synchronize clocks to within a 1-50 msec accuracy.

Logical Clocks

- Synchronization based on "relative time".

- Note that (with this mechanism) there is no requirement for "relative time" to have any relation to the "real time".

- What's important is that the processes in the Distributed System *agree on the **ordering** in which certain **events** occur*.

Such "clocks" are referred to as ***Logical Clocks***.

Lamport's Logical Clocks

- **First point**: if two processes do not interact, then their clocks do not need to be synchronized – they can operate *concurrently* without fear of interferring with each other.

- **Second (critical) point**: it does not matter that two processes share a common notion of what the "real" current time is. What does matter is that the processes have some agreement on the order in which certain events occur.
  - Lamport used these two observations to define the "**happens-before**" relation (also often referred to within the context of *Lamport's Timestamps*).

The “Happens-Before” Relation (1)

- If A and B are events in the same process, and A occurs before B, then we can state that:
  - *A “happens-before” B is true.*

- Equally, 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 “happens-before” B is also true.

(NB: a message cannot be received before it is sent, since it takes a finite, nonzero amount of time to arrive… and, of course, time is not allowed to run backwards).

The “Happens-Before” Relation (2)

- Obviously, if A “happens-before” B and B “happens-before” C, then it follows that
  - A “happens-before” C.
- If the “happens-before” relation holds, deductions about the current clock “value” on each DS component can then be made.
  - It therefore follows that if C(A) is the time on A, then C(A) < C(B), and so on.

The “Happens-Before” Relation (3)

- Assume three processes are in a DS: A, B and C.
- All have their own physical clocks (which are running at differing rates due to “clock skew”, etc.).
- A sends a message to B and includes a “timestamp”.
- If this sending timestamp is less than the time of arrival at B, things are OK, as the “happens-before” relation still holds (i.e. A “happens-before” B is true).
- However, if the timestamp is more than the time of arrival at B, things are NOT OK (as A “happens-before” B is not true, and this cannot be as the receipt of a message has to occur *after* it was sent).

The "Happens-Before" Relation (4)

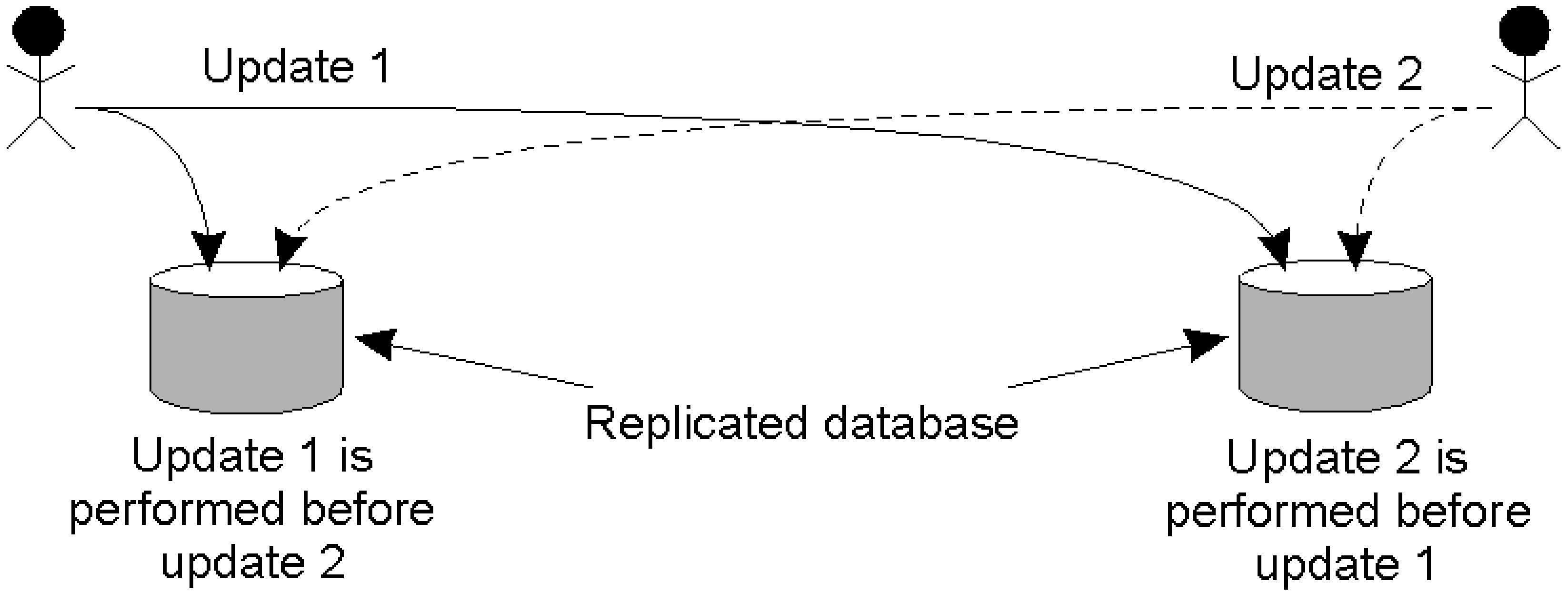
- **The question to ask is**:
  - How can some event that "happens-before" some other event possibly have occurred at a later time?

- **The answer is**: it can't!
  - So, Lamport's solution is to have *the **receiving process adjust its clock forward** to one more than the sending timestamp value*.
  - This allows the "happens-before" relation to hold, and also keeps all the clocks running in a synchronised state.
  - The clocks are all kept in sync *relative to each other*.

Example of Lamport's Timestamps

Diagram on page 254 of textbook.

Problem: Totally-Ordered Multicasting


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: Update 1 adds 100 euro to an account, Update 2 calculates and adds 1% interest to the same account.
- Due to network delays, the updates may not happen in the correct order… whoops!

Solution: Totally-Ordered Multicasting

- A multicast message is sent to all processes in the group, including the sender, together with the sender’s timestamp.
- At each process, the received message is added to a local queue, ordered by timestamp.
- Upon receipt of a message, a multicast acknowledgement/timestamp is sent to the group.
- Due to the “happens-before” relationship holding, the timestamp of the acknowledgement is always greater than that of the original message.

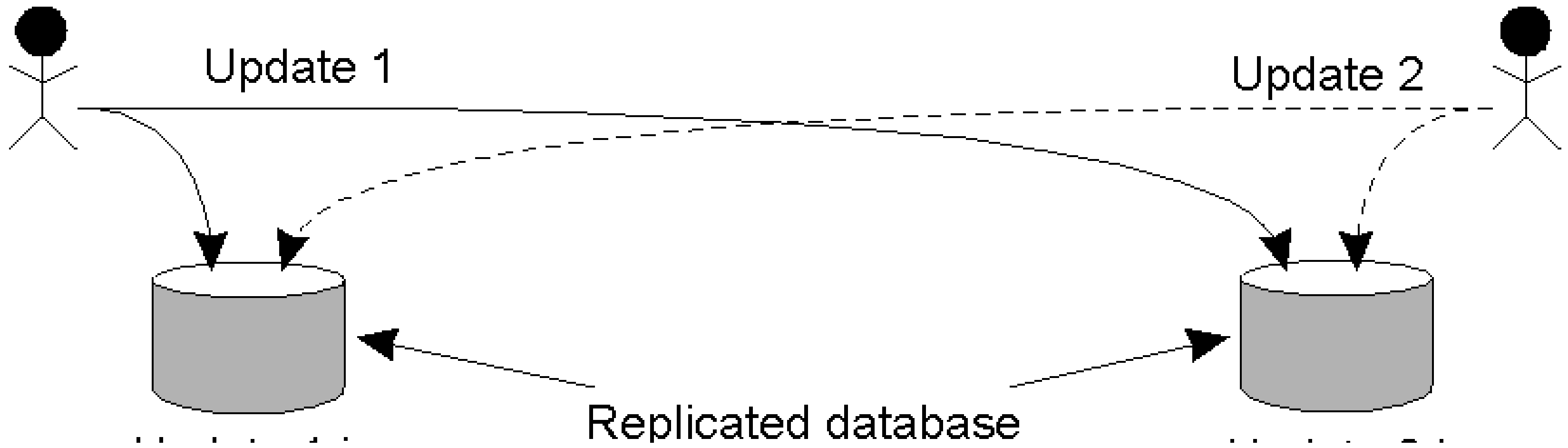
More Totally Ordered Multicasting

- Only when a message is marked as acknowledged by all the other processes will it be removed from the queue and delivered to a waiting application.

- Lamport’s clocks ensure that each message has a unique timestamp, and consequently, the local queue at each process eventually contains the same contents.

- In this way, all messages are delivered/processed in the same order everywhere, and updates can occur in a consistent manner.

Totally-Ordered Multicasting, Revisited



Update 1 is time-stamped and multicast. Added to local queues.

Update 2 is time-stamped and multicast. Added to local queues.

Ack for Update 2 sent/received. Update 2 can now be processed.

Ack for Update 1 sent/received. Update 1 can now be processed.

(Note: all queues are the same, as the timestamps have been used to ensure the "happens-before" relation holds.)

Election Algorithms

- Many DS require a process to act as *coordinator* (for various reasons).
  - The selection of this process can be performed automatically by an “**election algorithm**”.
- For simplicity, we assume the following:
  - Processes have a unique, positive identifier.
  - All processes know all other process identifiers.
  - The process with the highest valued identifier is duly elected coordinator.
  - When an election “concludes”, a coordinator has been chosen and is known to all processes.

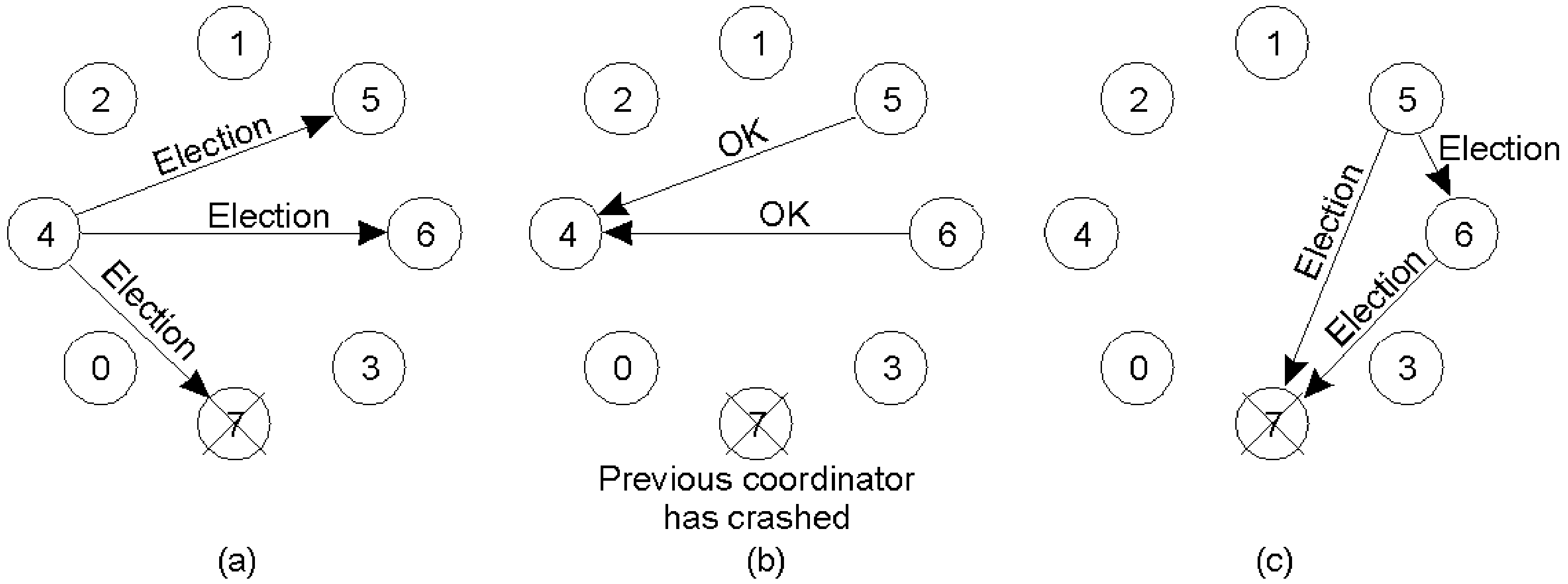
Goal of Election Algorithms

- The overriding goal of all election algorithms is to have all the processes in a group ***agree*** on a **coordinator**.

- There are two types of algorithm:

1. **Bully**: “the biggest guy in town wins”.
2. **Ring**: a logical, cyclic grouping.

The “Bully” Election Algorithm



- When a process “notices” that the current coordinator is no longer responding (4 deduces that 7 is down), it sends out an **ELECTION** message to any higher numbered process.
- If none respond, it (4) becomes the coordinator (sending out a **COORDINATOR** message to all other processes informing them of this change of coordinator).
- If a higher numbered process responds to the ELECTION message with an OK message, the election is cancelled and the higher-up process starts its own election (5 and 6 in this example both start, with 6 eventually winning).
- When the original coordinator (7) comes back on-line, it simply sends out a COORDINATOR message, as it is the highest numbered process (and it knows it).
- Simply put: the process with the highest numbered identifier *bullies* all others into submission.

The “Ring” Election Algorithm

- The processes are ordered in a “logical ring”, with each process knowing the identifier of its successor (and the identifiers of all the other processes in the ring).
- When a process “notices” that a coordinator is down, it creates an **ELECTION** message (which contains its own number) and starts to circulate the message around the ring.
- Each process puts itself forward as a candidate for election by adding its number to this message (assuming it has a higher numbered identifier).
- Eventually, the original process receives its original message back (having circled the ring), determines who the new coordinator is, then circulates a **COORDINATOR** message with the result to every process in the ring.
- With the election over, all processes can get back to work.

Mutual Exclusion within Distributed Systems

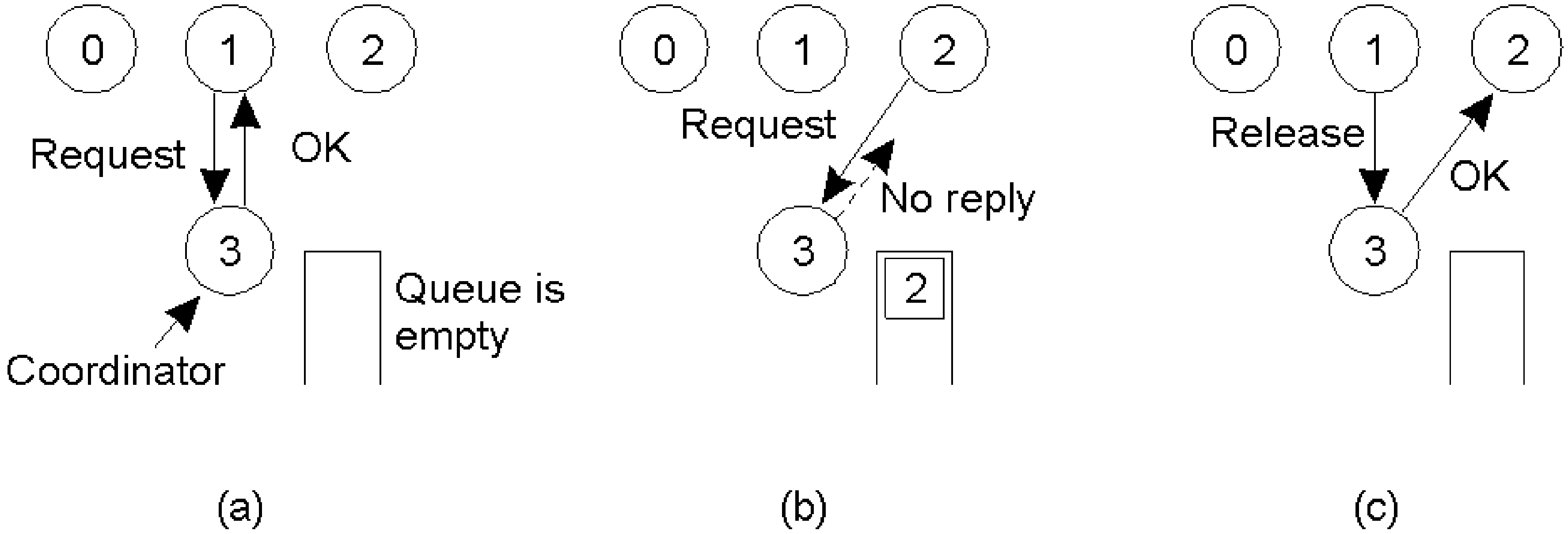
- It is often necessary to protect a *shared resource* within a DS using "**mutual exclusion**"
  - e.g., ensure that no other process changes a shared resource while another process is working with it.
- In non-distributed, uniprocessor systems, we can implement "**critical regions**" using techniques such as semaphores, monitors and similar constructs – achieve *mutual exclusion*.
  - These techniques have been adapted to Distributed Systems…

DS Mutual Exclusion Techniques

- **Centralized**: a single coordinator controls whether a process can enter a critical region.

- **Distributed**: the group *confers* to determine whether or not it is safe for a process to enter a critical region.

Mutual Exclusion: Centralized Algorithm



- P1 asks the coordinator permission to enter a critical region. Permission is granted by an OK message (assuming it is, of course, OK).
- P2 then asks permission to enter the same critical region. The coordinator does not reply (but adds 2 to a queue of processes waiting to enter the critical region). No reply is interpreted as a “busy state” for the critical region.
- When P1 exits the critical region, it tells the coordinator, which then replies to P2 with an OK message.

Comments: The Centralized Algorithm

- ***Advantages***:
  - It works.
  - It is fair.
  - There’s no process starvation.
  - Easy to implement.
- ***Disadvantages***:
  - There’s a single point of failure!
  - The coordinator is a bottleneck on busy systems.

Critical Question: when there is no reply, does this mean that the cordinator is “dead” or just busy?

Distributed Mutual Exclusion

- Based on work by Ricart &Agrawala (1981).

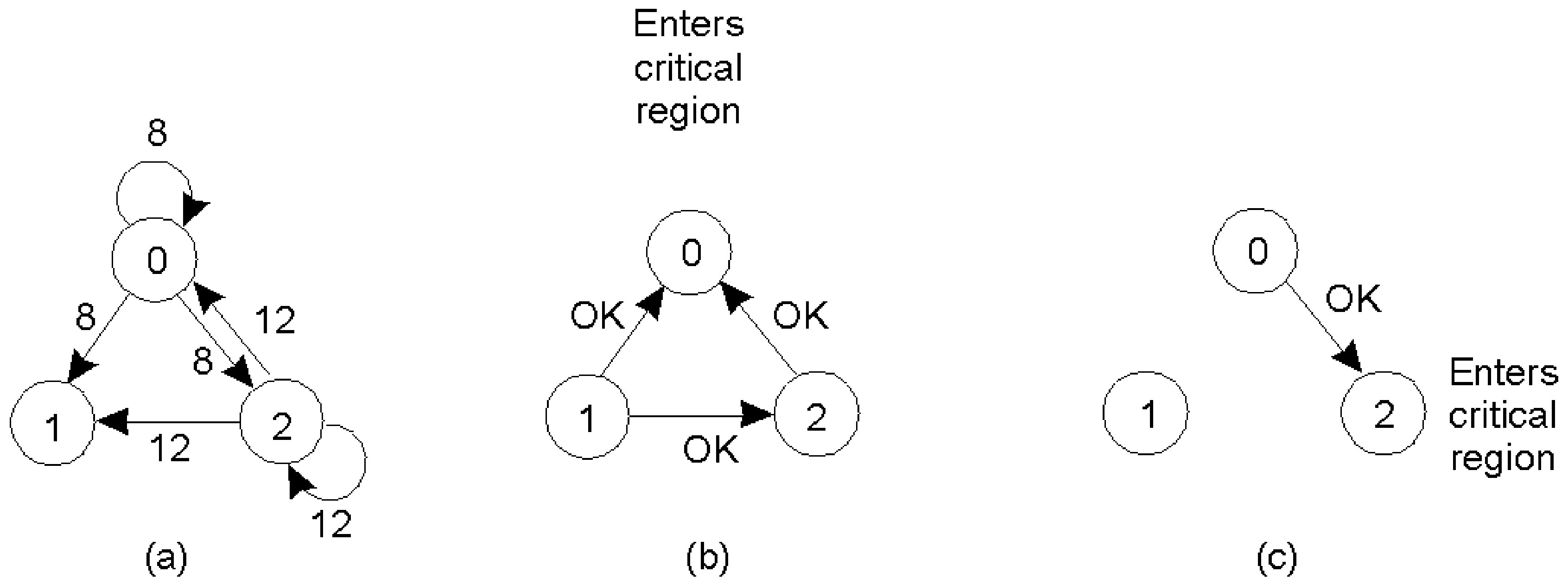
- Requirement of their solution: *total ordering* of all events in the distributed system (which is achievable with Lamport's timestamps).

- Note that messages in their system contain three pieces of information:
  1. The critical region ID.
  2. The requesting process ID.
  3. The current time.

Mutual Exclusion: Distributed Algorithm

1. When a process ("requesting process") decides to enter a critical region, a message is sent to all processes in the DS (including itself).
2. What happens at each process depends on the "state" of the critical region.
3. If not in the critical region (and not waiting to enter it), a process sends back an OK to the requesting process.
4. If in the critical region, a process will queue the request and send back *no reply* to the requesting process.
5. If **waiting** to enter the critical region, a process will:
   a) Compare the timestamp of the new message with that in its queue (note that the lowest timestamp wins).
   b) If the received timestamp wins, an OK is sent back, otherwise the request is queued (and no reply is sent back).
6. When all the processes send OK, the requesting process can safely enter the critical region.
7. When the requesting process leaves the critical region, it sends an OK to all the process in its queue, then empties its queue.

The Distributed Algorithm in Action



P0 and P2 wish to enter the critical region "at the same time".
P0 wins as it's timestamp is lower than that of P2.
When P0 leaves the critical region, it sends an OK to P2.

Comments: The Distributed Algorithm

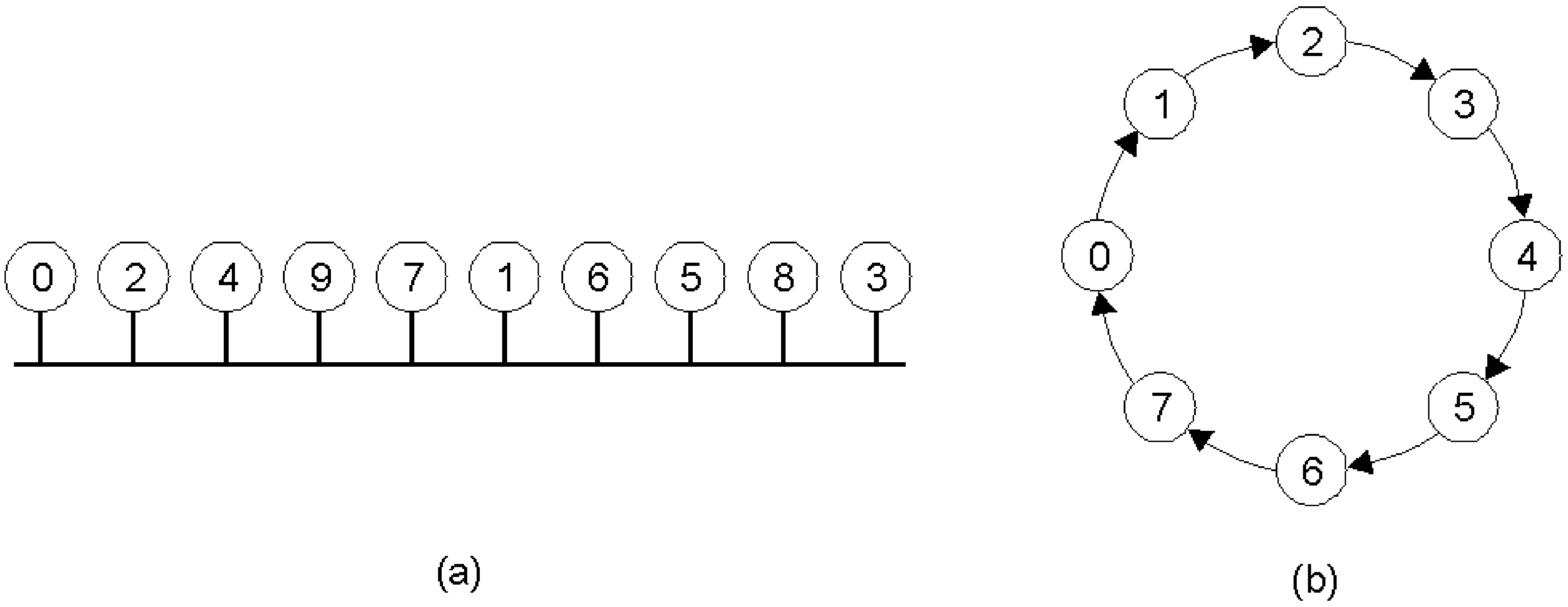
- The algorithm works because in the case of a conflict, the lowest timestamp wins as everyone agrees on the total ordering of the events in the distributed system.
- **Advantages**:
  - It works.
  - There is no single point of failure
- **Disadvantages**:
  - We now have multiple points of failure!!!
  - A "crash" is interpreted as a *denial of entry* to a critical region.
  - (A patch to the algorithm requires all messages to be ACKed).
  - Worse is that all processes must maintain a list of the current processes in the group (and this can be tricky)
  - Worse still is that one overworked process in the system can become a *bottleneck* to the entire system – so, everyone slows down.

Which Just Goes To Show …

- It is NOT always best to implement a distributed algorithm when a *reasonably good* centralised solution exists.
- Also, what's good *in theory* (or on paper) may not be so good *in practice*.

- Finally, think of all the message traffic this distributed algorithm is generating (especially with all those ACKs). **Remember**: every process is involved in the decision to enter the critical region, whether they have an interest in it or not. (Oh dear …)

Mutual Exclusion: Token-Ring Algorithm



- An unordered group of processes on a network. Note that each process knows the process that is next in order on the ring after itself.
- A logical ring is constructed in software, around which a token can circulate – a **critical region** can only be entered when the **token** in held.
- When the critical region is exited, the token is released.

Comments: Token-Ring Algorithm

- **Advantages**:
  - It works (as there’s only one token, so mutual exclusion is guaranteed).
  - It’s fair (everyone gets a shot at grabbing the token at some stage).
- **Disadvantages**:
  - Lost token… how is the loss detected (it is in use or is it lost)? How is the token regenerated?
  - Process failure can cause problems – a broken ring!
  - Every process is required to maintain the current logical ring in memory – not easy.

Comparison: Mutual Exclusion 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

None are perfect – they all have their problems!

- "Centralized" algorithm is simple and efficient, but suffers from a single point-of-failure.
- "Distributed" algorithm has nothing going for it – it is slow, complicated, inefficient of network bandwidth, and not very robust.
- "Token-Ring" algorithm suffers from the fact that it can sometimes take a long time to reenter a critical region having just exited it.
- All perform poorly when a process crashes, and they are all generally poorer technologies than their non-distributed counterparts. Only in situations where crashes are very infrequent should any of these techniques be considered.

Introduction to Transactions

- Related to Mutual Exclusion since it protects a “shared resource”.

- Transactions also protect “shared data”.
  - Often, a single transaction contains a collection of data accesses/modifications.
  - The collection is treated as an “atomic operation” – either all the collection complete, or none of them do.
  - Mechanisms exist for the system to revert (rollback) to a previously “good state” whenever a transaction prematurely aborts.

The Transaction Model (1)

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 (2)

```
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.

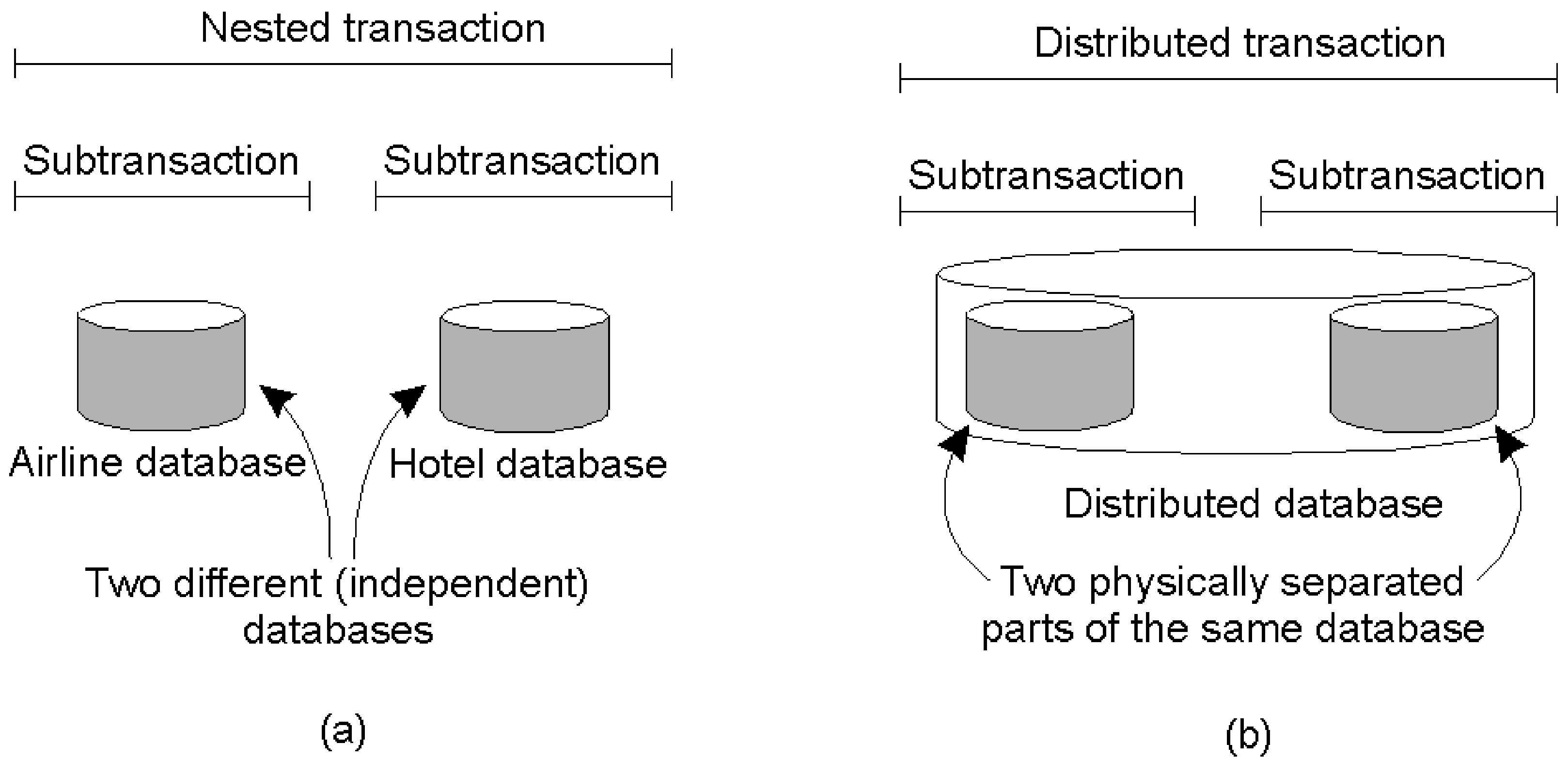
A.C.I.D.

- Four key transaction characteristics:
  - **Atomic**: the transaction is considered to be one thing, even though it may be made of up many different parts.
  - **Consistent**: “invariants” that held before the transaction must also hold after its successful execution.
  - **Isolated**: if multiple transactions run at the same time, they must not interfere with each other. To the system, it should look like the two (or more) transactions are executed sequentially (i.e., that they are serializable).
  - **Durable**: Once a transaction commits, any changes are permanent.

Types of Transactions

- **Flat Transaction**: model that we have looked at so far. Disadvantage: too rigid. Partial results cannot be committed. That is, the “atomic” nature of Flat Transactions can be a downside.
- **Nested Transaction**: a main, parent transaction spawns child sub-transactions to do the real work. Disadvantage: problems result when a sub-transaction commits and then the parent aborts the main transaction. Things get messy.
- **Distributed Transaction**: this is sub-transactions operating on distributed data stores. Disadvantage: complex mechanisms required to lock the distributed data, as well as commit the entire transaction.

Nested vs. Distributed Transactions



a) A nested transaction – logically decomposed into a hierarchy of sub-transactions.
b) A distributed transaction – logically a flat, indivisible transaction that operates on distributed data.

Summary

- **Synchronization**: doing the right thing at the right time.
  - No real notion of a globally shared "physical" clock.
  - Clock-synchronization algorithms exist to try and provide such a facility.
  - Often not necessary to know the time, more important to be sure that things happen in the correct order… this leads to the notion of "logical" clocks (**Lamport's Timestamps**).
- Synchronization also achieved by one process acting as **coordinator**.
  - Dynamic "election algorithms" have been developed to allow DS's to automatically select the coordinator.
- **Distributed Mutual Exclusion** is possible (in theory), but performs poorly on "real" networks.
- Related to Mutual Exclusion is the notion of a **Transaction**
  - there are three main types: flat, nested and distributed.
  - Important characteristics: **A.C.I.D.** (Atomic, Consistent, Isolated and Durable).