Synchronization

Distributed Algorithms

- Algorithms that are intended to work in a distributed environment
- Used to achieve tasks such as:
  - Communication
  - Accessing resources
  - Allocating resources
  - etc.
- Synchronization and Coordination linked to distributed algorithms
  - Achieved using distributed algorithms

Why Synchronize?

- It is important to *control access* to a single, shared resource and agree on the *ordering of events* such as whether message *m1* from process *P* was sent before or after message *m2* from process *Q.*
- Synchronization in Distributed Systems is much more difficult than in uniprocessor systems.
- How processes cooperate and synchronize with each other?

Synchronization And Coordination

- making sure that processes doing the right thing at the right time.

➢ Two fundamental issues:

- Coordination (the right thing)
- Synchronization (the right time)

- allowing processes to synchronize and coordinate their actions

Synchronization

- Synchronization is coordination with respect to time, and refers to the **ordering** of events and execution of instructions in time.
- “Ordering of all actions”

- Total ordering of events
  (whether message m1 from process P was sent before or after message m2 from process Q)
- Ordering of access to resources
- Examples of synchronization - Ordering distributed events and ensuring that a process performs an action at a particular time.

Coordination

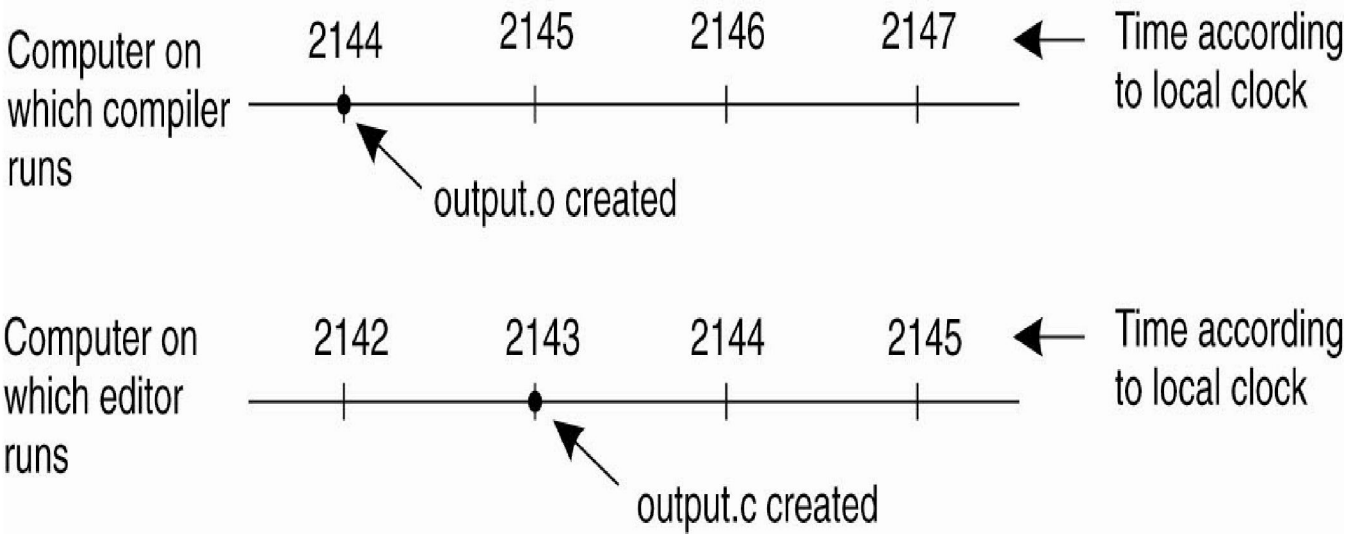
- Coordination refers to coordinating the actions of separate processes relative to each other and allowing them to agree on global state.
- "**Coordinate actions and agree on values.**"

- Coordinate Actions:
- What actions will occur
- Who will perform actions
- Agree on Values:
- Agree on global value
- Agree on environment
- Agree on state
- Examples of coordination include ensuring that processes agree on –
- what actions will be performed?
- who will be performing actions?
- state of the system?

Main Issues

- Time and Clocks: synchronizing clocks and using time in distributed algorithms
- Global State: how to acquire knowledge of the system's global state
- Concurrency Control: coordinating concurrent access to resources
- Coordination: when do processes need to coordinate and how do they do it

Clock Synchronization



For make command in Unix/Linux: When each machine has its own clock, an event that occurred after another event may nevertheless be assigned an earlier time

Clock Synchronization

- Synchronization based on “Actual Time” is really easy on a uniprocessor system.
- Achieving agreement on time in a DS is not trivial.
- Is it even possible to synchronize all the clocks in a Distributed System?
- Skew- Disagreement on reading of two clocks
- Drift – Difference in the rate at which two clocks count the time(due to crystal material, heat, voltage, humidity, etc.)
- 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”).
- 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.

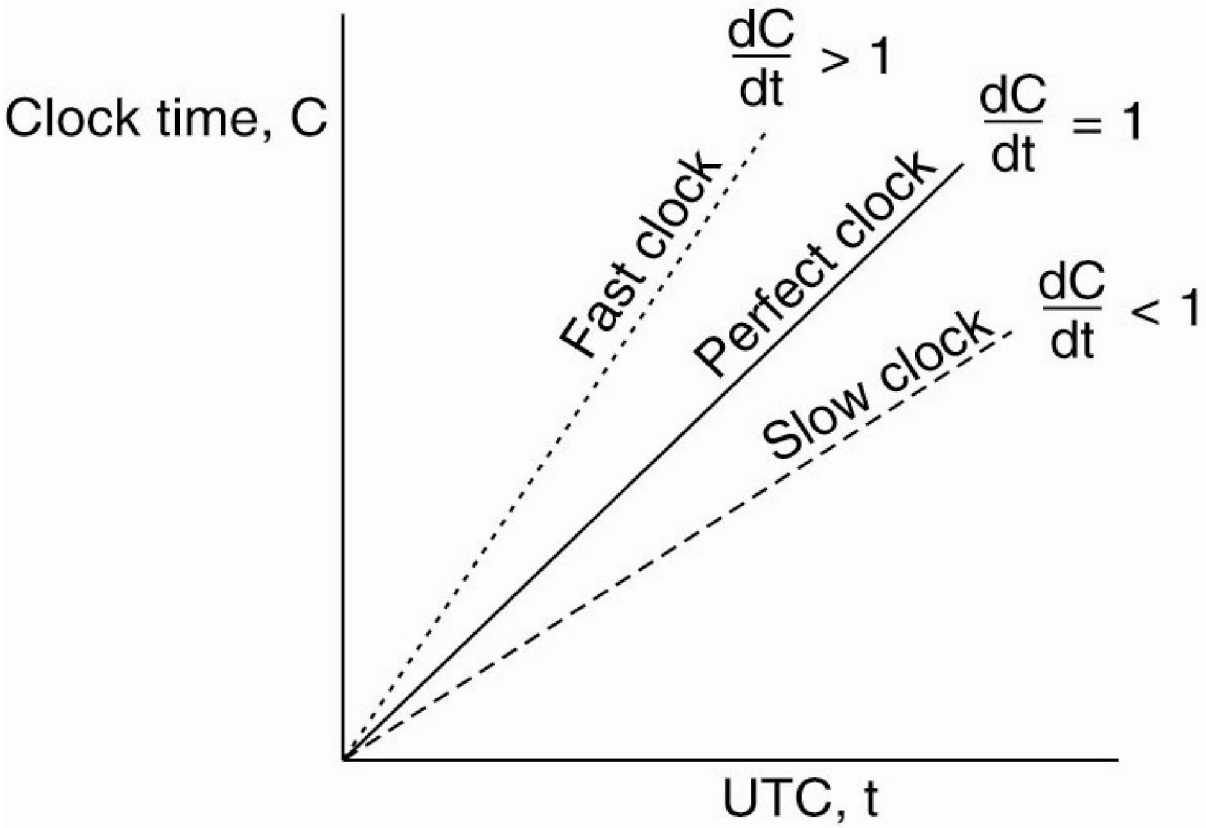
Clock synchronization algorithms

- UTC – Universal Coordinated Time
- When UTC time is t, the value of the clock on machine p is Cp(t) = t (in perfect world)
- That means dC/dt=1
- If there exists some constant q,

$1-q <= dC/dt <= 1+q$

Where q is maximum drift rate

Clock synchronization algorithms

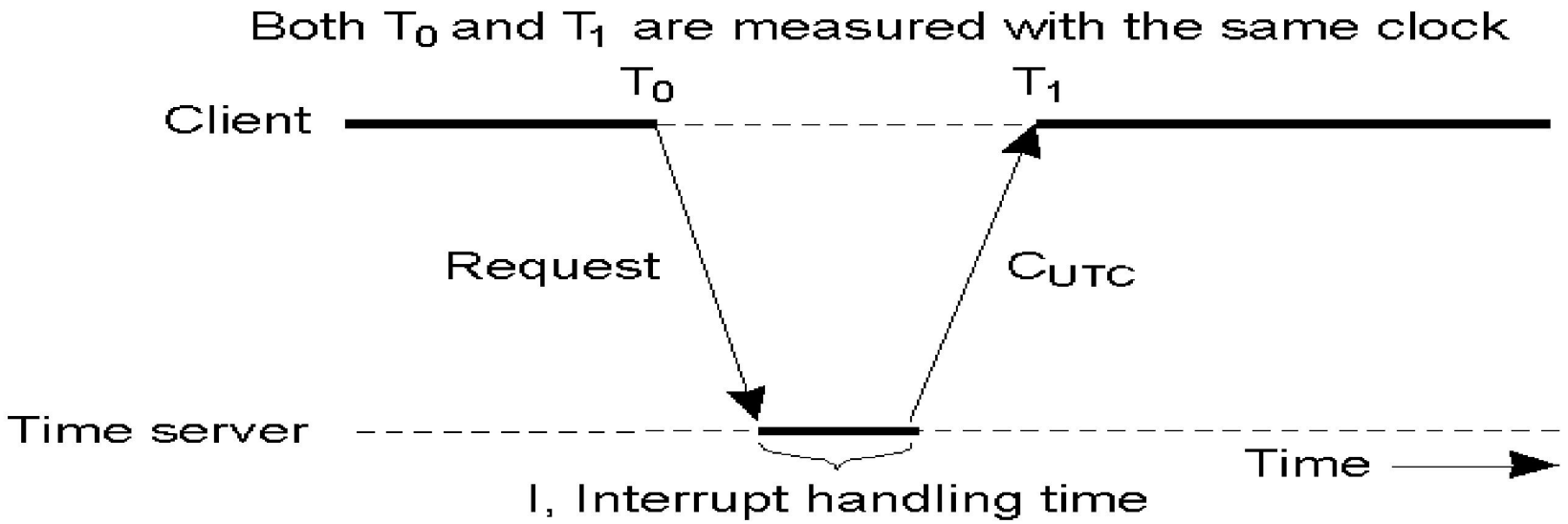


The relation between clock time and UTC when clocks tick at different rates(slow, perfect, fast)

Clock synchronization algorithms

- There exists a *time server* receiving signals from a UTC source
  - Cristian's algorithm
- There is no UTC source available
  - Berkley's algorithm
- Exact time *does not matter!*
  - Lamport's algorithm

Cristian's Algorithm



- Cristian's algorithm requires clients to periodically synchronize with a central time server (typically a server with a UTC receiver)
- Every computer periodically asks the "time server" for the current time
- The server responds ASAP with the current time C_{UTC}
- The client sets its clock to C_{UTC}

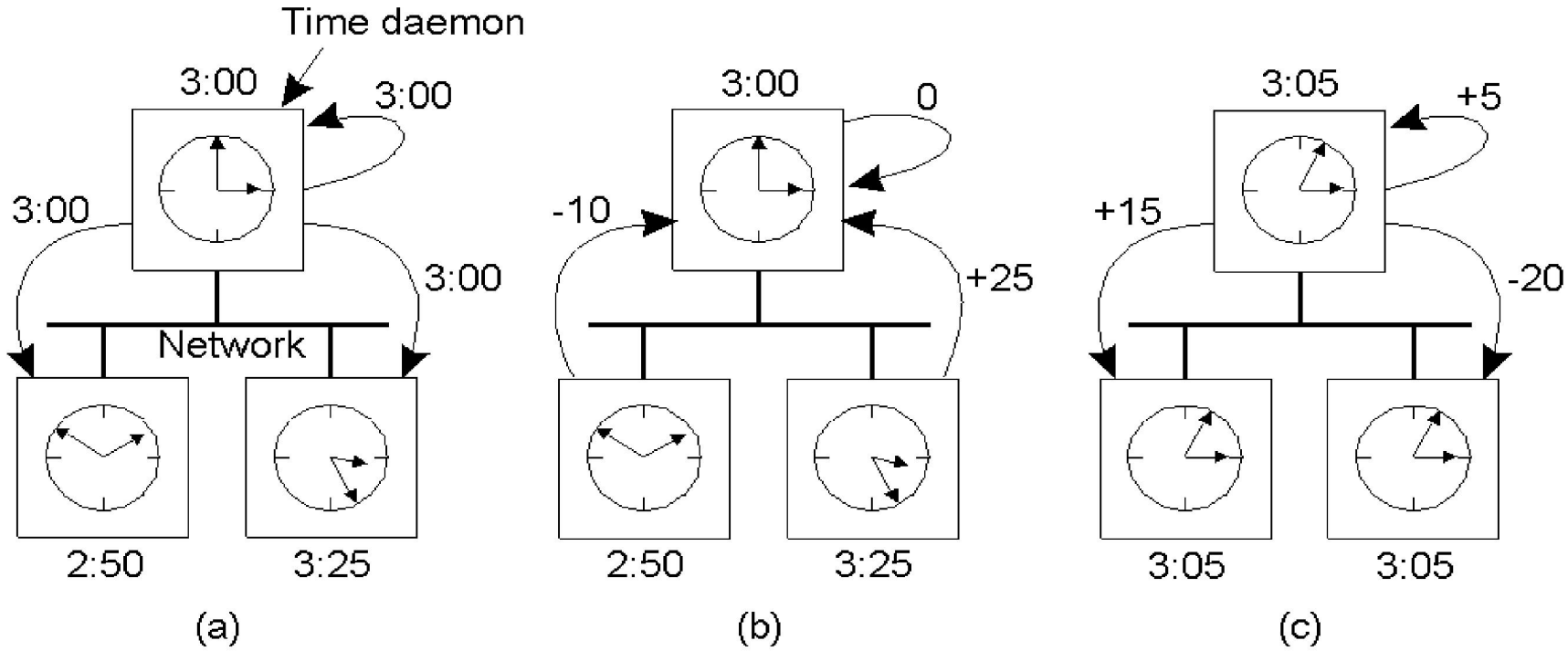
Cristian's Algorithm

- **Major problem -** if time from time server is less than the client – resulting in time running backwards on the client! (Which cannot happen – time does not go backwards).
- **Minor problem** - results from the delay introduced by the network request/response latency.
- Estimate message propagation time – (T1-T0)/2
- One way propagation time – (T1-T0-I)/2
- To improve accuracy – take series of measurements and discard those whose T1-T0 exceeds threshold value

Berkeley Algorithm

- In Christian's algorithm – Time server is passive
- An algorithm for internal synchronization of a group of computers
- A *master* polls to collect clock values from the others (*slaves*)
- The master uses round trip times to estimate the slaves' clock values
- It takes an average
- It sends the required adjustment to the slaves
- If master fails, can elect a new master to take over

The Berkeley Algorithm



a) The time daemon asks all the other machines for their clock values

b) The machines answer

c) Takes an average & tells everyone how to adjust their clock

Berkeley Algorithm

Computers	Clock Reading
A (daemon)	3:00
B (left)	2:50
C (right)	3:25

Computers	Ahead/Behind
A (daemon)	0:00
B (left)	-0:10
C (right)	+0:25

Berkeley Algorithm

- Average time: 3:05

Computers	Needed Adjustment
A (daemon)	+0:05
B (left)	+0:15
C (right)	-0:20

Other Clock Sync. Algorithms

- Both Cristian's and the Berkeley Algorithm are centralized algorithms.
- Decentralized algorithms also exist- Averaging algorithms, Internet's Network Time Protocol (NTP)
- NTP achieves worldwide accuracy in the range of 1-50 msec through the use of advanced clock synchronization algorithms.
- Multiple external time sources
- Use of synchronized clocks – using timestamp

  - G = CurrentTime – MaxLifeTime – MaxClockSkew

  - G is global variable, MaxLifeTime is max time a message can live, and MaxClockSkew is how far from UTC the clock might be at worst

  - any timestamp older than G can safely be removed

Averaging algorithms

- Divide time into fixed-length resynchronization intervals
- ith interval starts at T0+iR and runs until T0+(i+1)R, where T0 is an agreed-upon moment in the past, R is a system parameter
- At the beginning of each interval, every machine broadcasts the current time as per its clock
- After broadcast, each machine will start a local timer to collect all other broadcasts arrive during some interval S
- Average the values received from others
- Variation 1 – discard highest and lowest values and average the rest
- Variation 2 – correct each message by adding estimate propagation time from the source

Logical Clocks

- For may purposes, it is sufficient that all machines agree on the same time and not essential to be real time.
- In make, if all machines agree on particular time irrespective of actual time, then it is fine.
- Lamport(1978) showed in his paper that - for many DS algorithms, associating an event to an absolute real time is not essential, we only need to know an unambiguous order of events
    - Lamport's timestamps
    - Vector timestamps

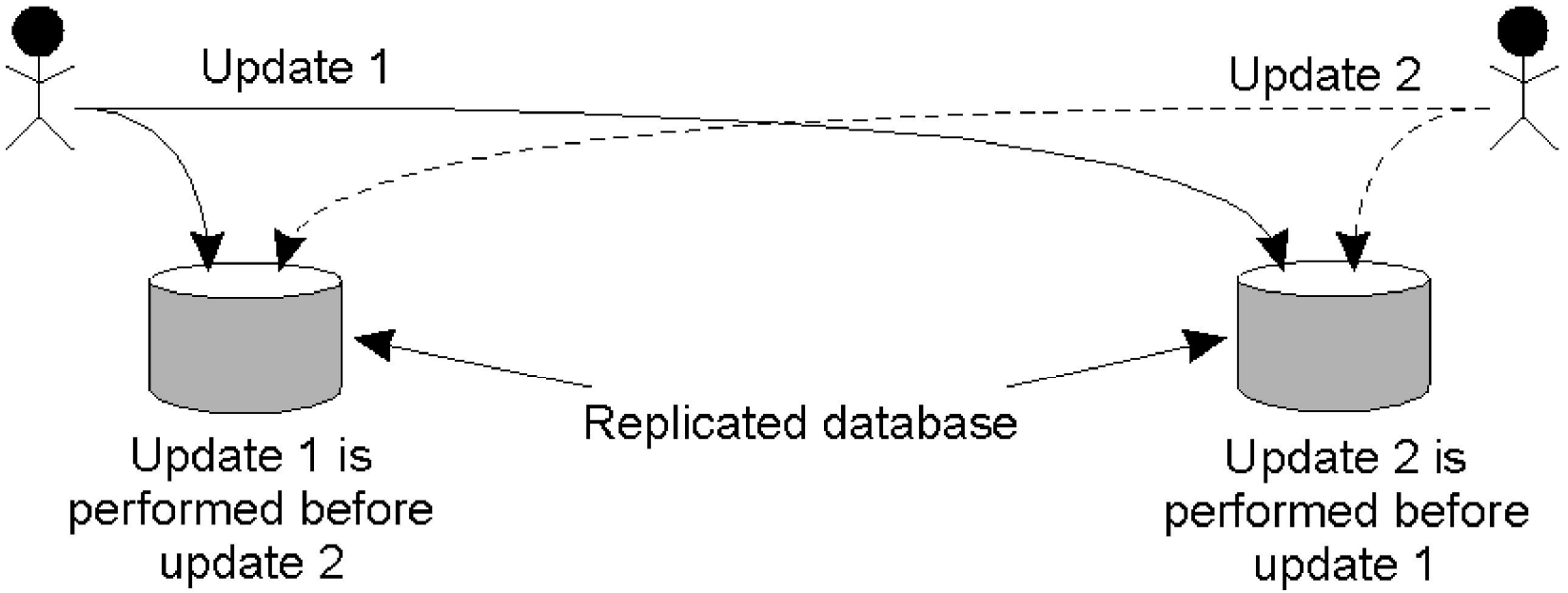
Logical Clocks

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

Logical Clocks (Cont.)

- Synchronization based on “relative time”.
  - Example: Unix make (Is output.c updated after the generation of output.o?)
- “relative time” may not relate 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*.

Example: Why Order Matters?



- Replicated accounts in New York(NY) and San Francisco(SF)
- Two updates occur at the same time
  - Current balance: $1,000
  - Update1: Add $100 at SF; Update2: Add interest of 1% at NY
  - Whoops, inconsistent states!

Ordering Events

- Event ordering linked with concept of ***causality***:
  - Saying that event *a* happened before event *b* is same as saying that event *a* causally affects event *b*
  - If events *a* and *b* happen on processes that do not exchange any data, their exact ordering is not important

The “Happens-Before” Relation

- 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.
- 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

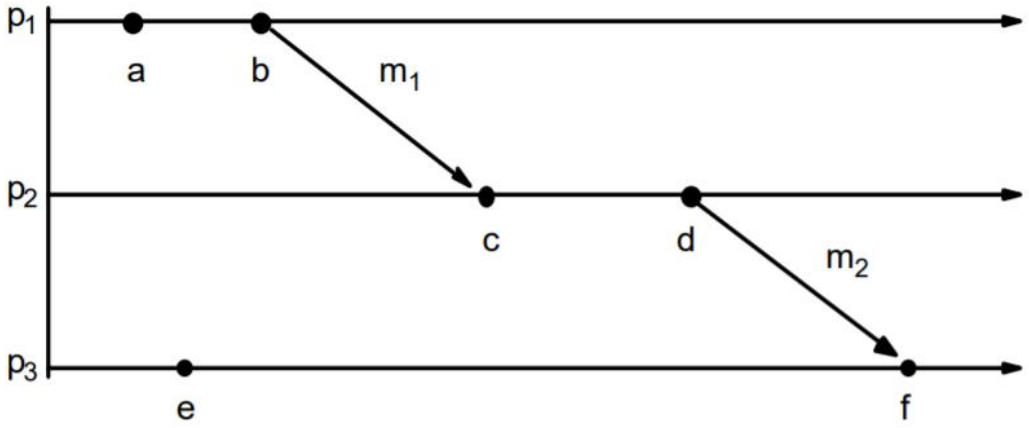
- If A “happens-before” B and B “happens-before” C, then it follows that A “happens-before” C.
- If the “happens-before” relation holds, deductions about the current clock “value” on each DS component can then be made.
- It therefore follows that if C(A) is the time on A, then C(A) < C(B), and so on.
- If two events on separate sites have same time, use unique PIDs to break the tie.

The “Happens-Before” Relation

- Now, 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

- a -> b : Event a occurred before event b. Events in the same process p1.
- b -> c : If b is the event of sending a message m1 in a process p1 and c is the event of receipt of the same message m1 by another process p2.
- a -> b, b -> c, then a -> c; “->” is transitive.



The “Happens-Before” Relation

- How can some event that “happens-before” some other event possibly have occurred at a later time? NO
- So, Lamport’s solution - 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 synchronized state. The clocks are all kept in sync *relative to each other*.

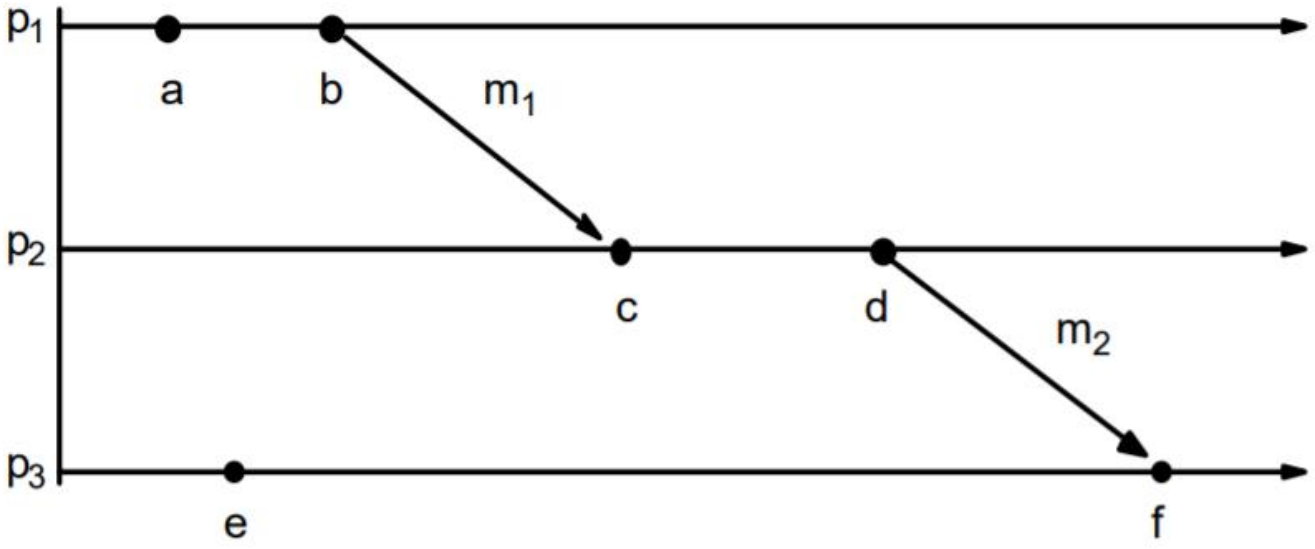
Lamport's Logical Clocks

- The "happens-before" relation $\rightarrow$ can be observed directly in two situations:

1. If *a* and *b* are events in the same process, and *a* occurs before *b*, then $a \rightarrow b$ is true.
2. 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 $a \rightarrow b$.

Lamport's Logical Clocks

- Causally Ordered Events
  – a -> b : Event a "causally" affects event b
- Concurrent Events
  – a || e: if a !-> e and e !-> a

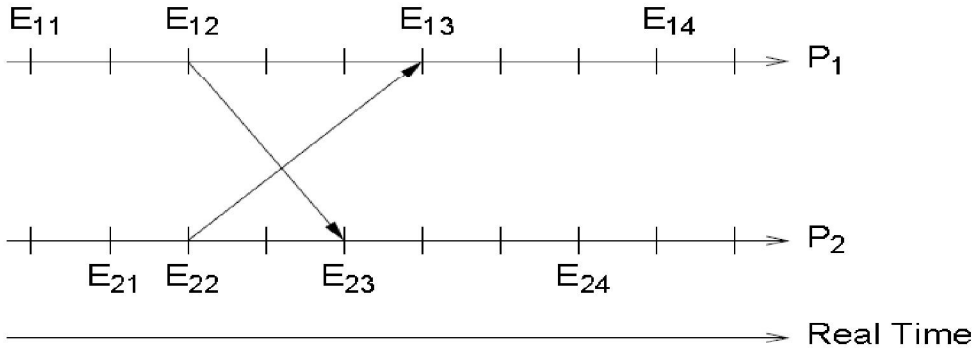


Lamport's Logical Clocks

The relation $\rightarrow$ is a partial order:

- If $a \rightarrow b$, then a causally affects b
- We consider unordered events to be concurrent:

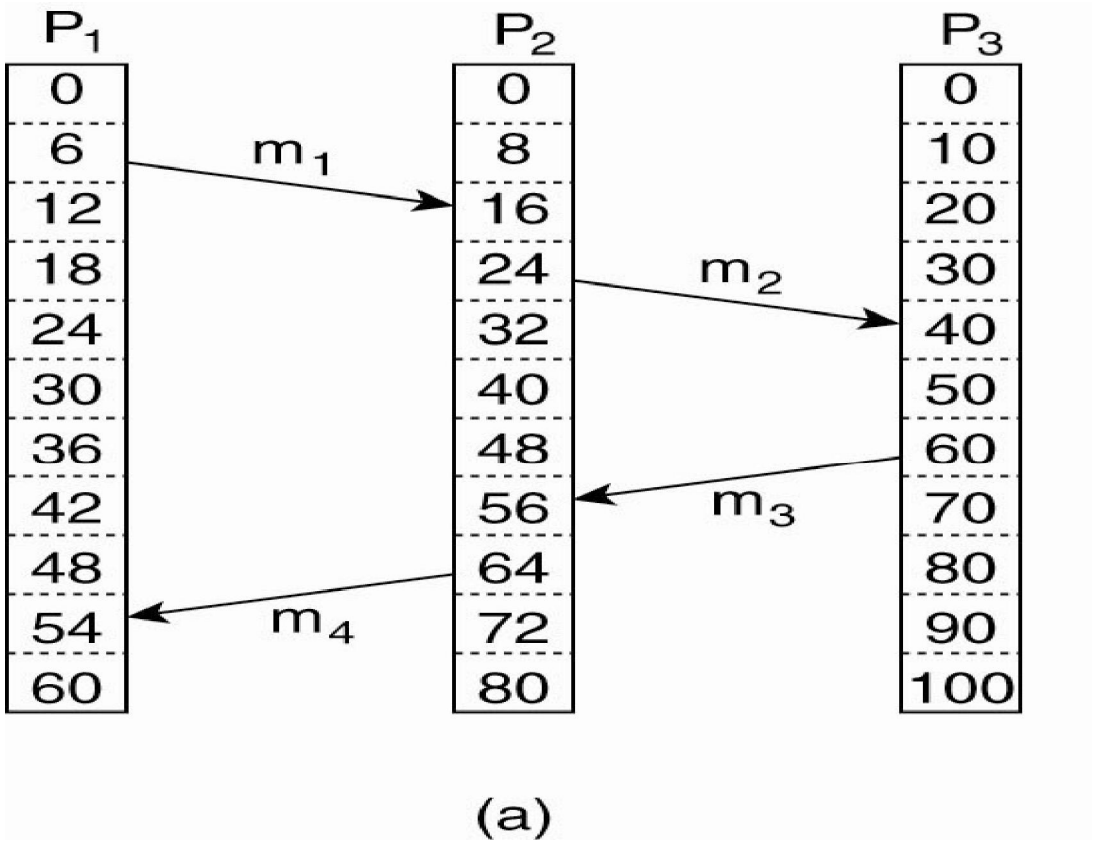
Example: $a \not\rightarrow b$ and $b \not\rightarrow a$ implies $a \parallel b$



- Causally related: $E_{11} \rightarrow E_{12}, E_{13}, E_{14}, E_{23}, E_{24}, \ldots$

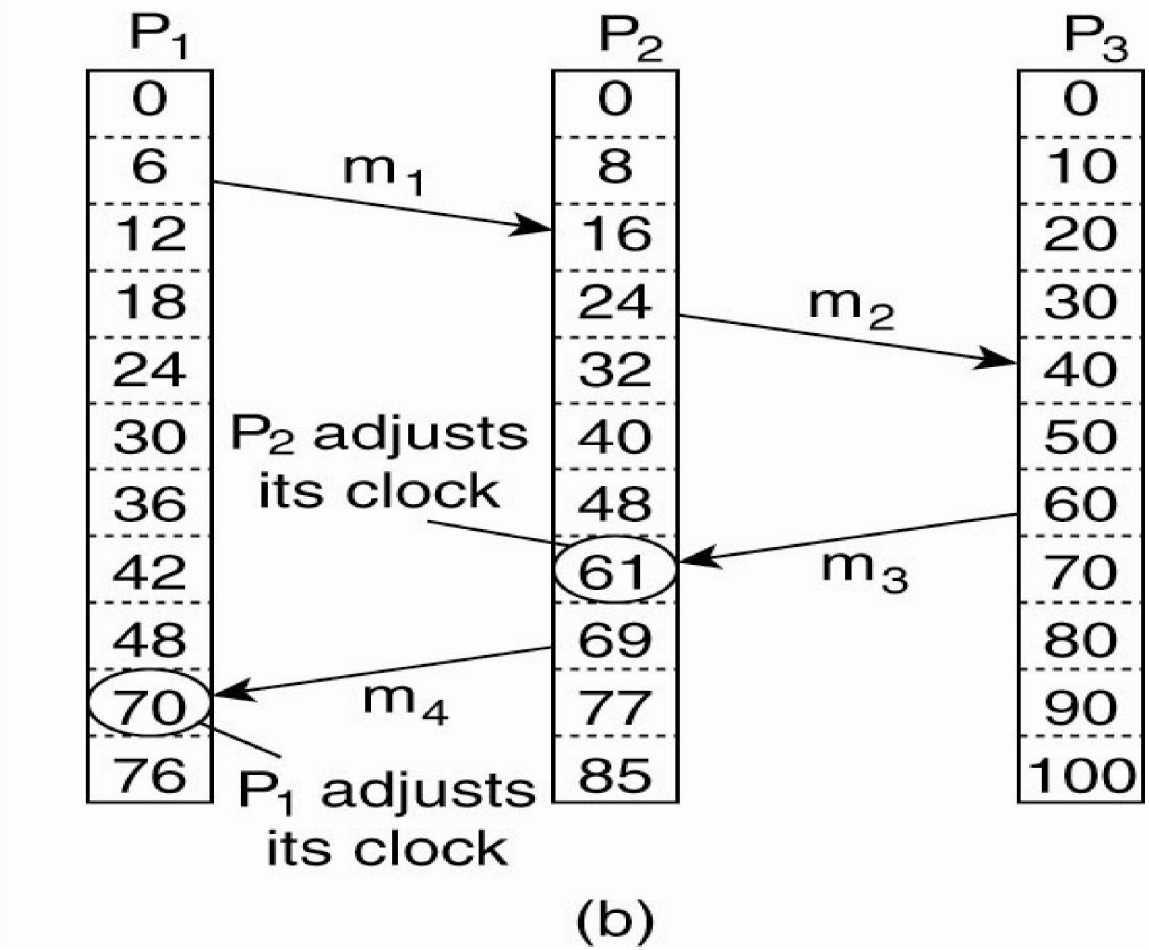
  $E_{21} \rightarrow E_{22}, E_{23}, E_{24}, E_{13}, E_{14}, \ldots$
- Concurrent: $E_{11} \| E_{21}$, $E_{12} \| E_{22}$, $E_{13} \| E_{23}$, $E_{11} \| E_{22}$, $E_{13} \| E_{24}$, $E_{14} \| E_{23}, \ldots$

Lamport's Logical Clocks



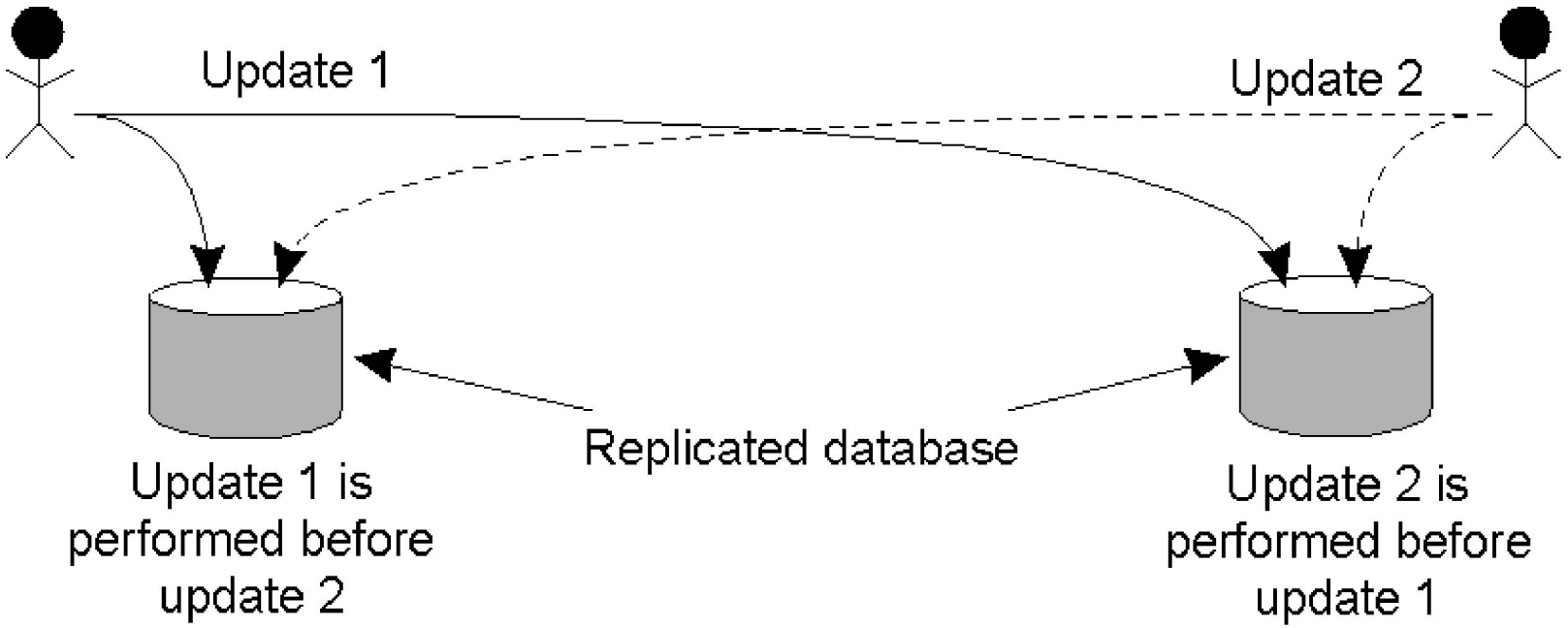
(a) Three processes, each with its own clock. The clocks run at different rates.

Lamport's Logical Clocks



(b) Lamport's algorithm corrects the clocks

Example: Totally-Ordered Multicasting



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!

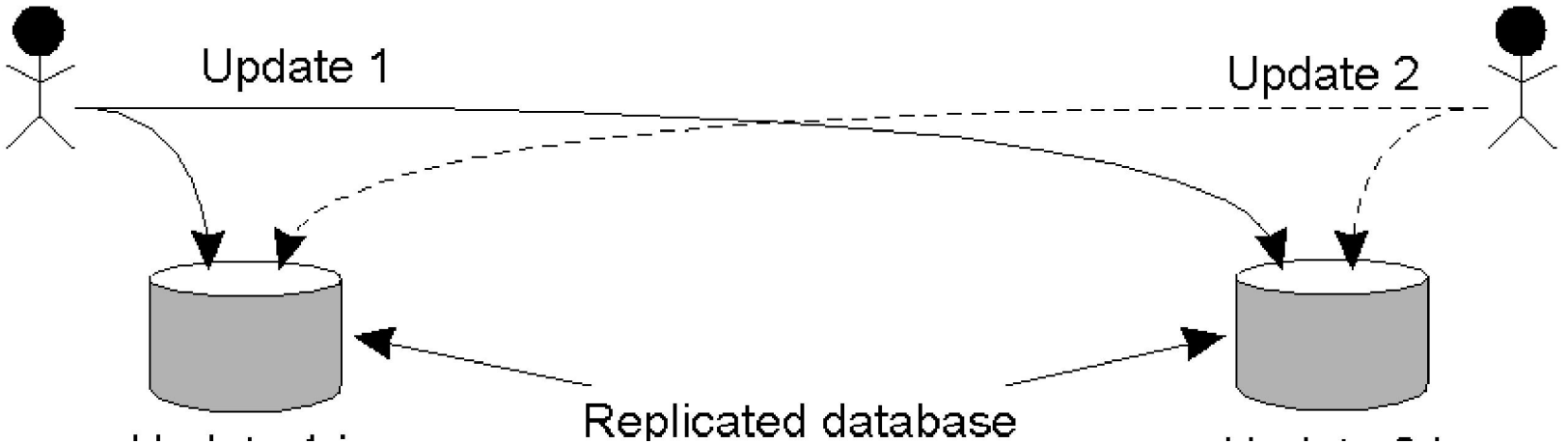
Solution: Totally-Ordered Multicasting

- 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, the timestamp of the acknowledgement is always greater than that of the original message.

Totally Ordered Multicasting

- Only when a message is marked as acknowledged by all the other processes will 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.
- All messages are delivered/processed in the same order everywhere, and updates can occur in a consistent manner.

Totally-Ordered Multicasting



- Update 1 is time-stamped and multicast. Added to local queues.
- Update 2 is time-stamped and multicast. Added to local queues.
- Acknowledgements for Update 2 sent/received. Update 2 can now be processed.
- Acknowledgements 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.)

Lamport Algorithm

- Updating counter C_i for process P_i :

1. Before executing an event P_i executes $C_i \leftarrow C_i + 1$.
2. When process P_i sends a message m to P_j, it sets *m*'s timestamp *ts(m)* equal to C_i after having executed the previous step.
3. Upon the receipt of a message *m*, process P_j adjusts its own local counter as $C_j \leftarrow \max\{C_j , ts(m)\}$, after which it then executes the first step and delivers the message to the application.

Lamport Algorithm

- Each process increments local clock between any two successive events
- Message contains a timestamp
- Upon receiving a message, if received timestamp is ahead, receiver fast forward its clock to be one more than sending time

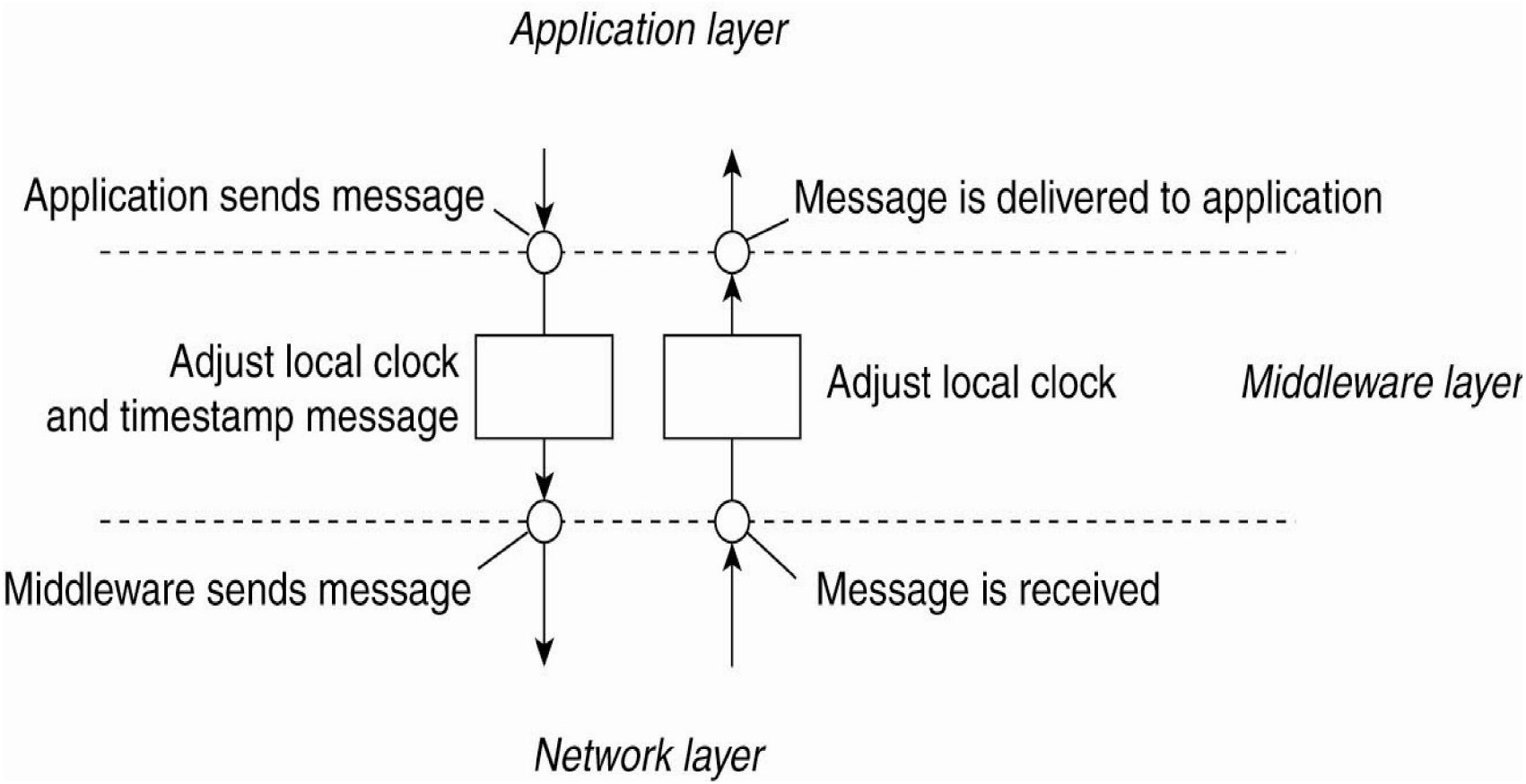
- Sending end

```
time = time+1;
time_stamp = time;
send(message, time_stamp);
```

- Receiving end

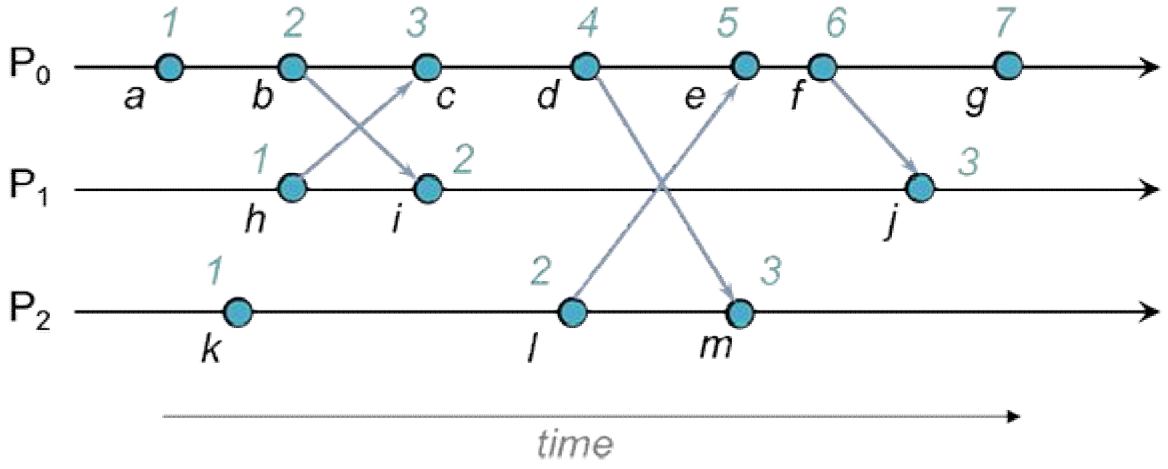
```
(message, time_stamp) = receive();
time = max(time_stamp, time)+1;
```

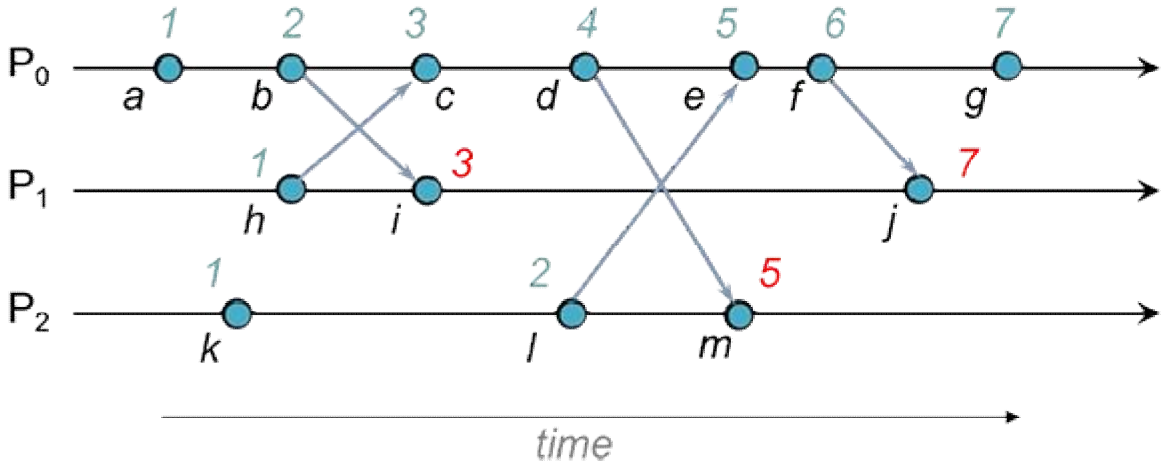
Lamport Algorithm



The positioning of Lamport's logical clocks in distributed systems

Example

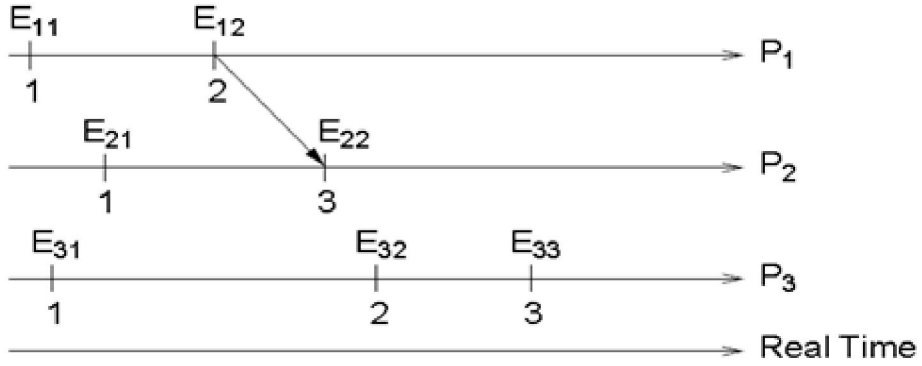

P0
P1
P2
1
2
3
4
5
6
7
a
b
c
d
e
f
g
h
i
j
k
l
m
time


P0
P1
P2
1
2
3
4
5
6
7
a
b
c
d
e
f
g
h
i
j
k
l
m
time

Vector Clocks

Main shortcoming of Lamport's clocks:

- $L(a) < L(b)$ does not imply $a \rightarrow b$
- We cannot deduce causal dependencies from time stamps:



- We have $L_1(E_{11}) < L_3(E_{33})$, but $E_{11} \not\rightarrow E_{33}$
- Why?
  - Clocks advances independently or via messages
  - There is no history as to where advances come from

Vector Clocks Algorithm

- Vector clocks are constructed by letting each process P_i maintain a vector VC_i with the following two properties:

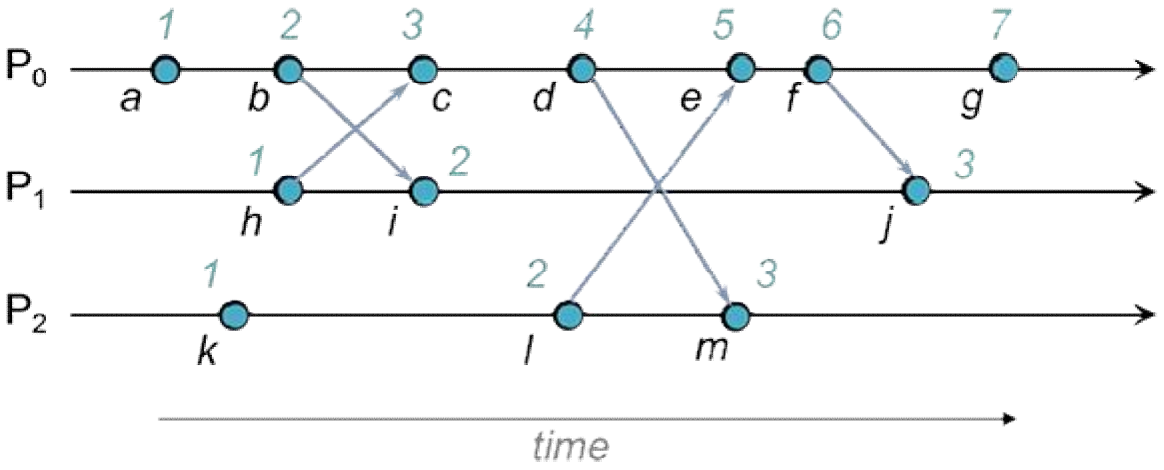
1. VC_i [*i*] is the number of events that have occurred so far at P_i. In other words, VC_i [*i*] is the local logical clock at process P_i .
2. If VC_i [*j*] = k then P_i knows that k events have occurred at P_j. It is thus P_i's knowledge of the local time at P_j .

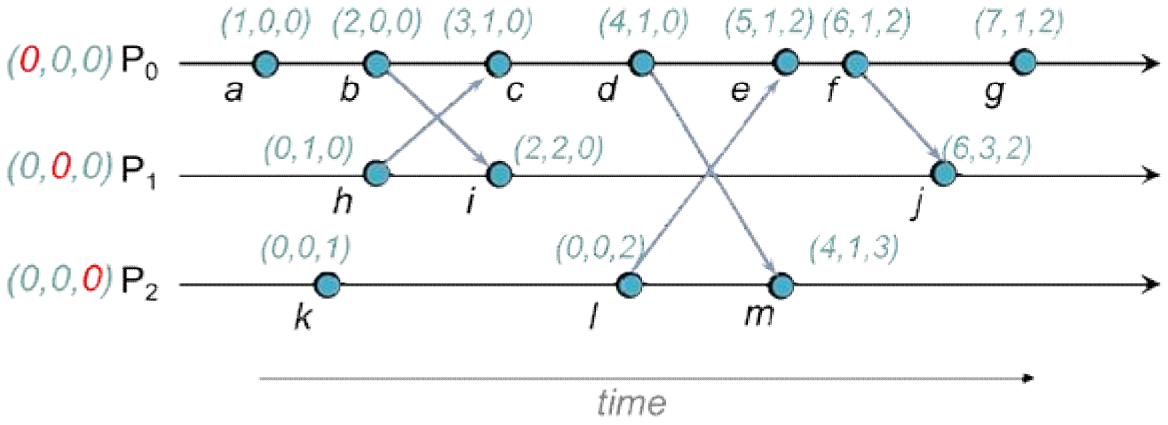
Vector Clocks Algorithm

- Steps carried out to accomplish property 2 of previous slide:

1. Before executing an event, P_i executes $VC_i[i] \leftarrow VC_i[i] + 1$.
2. When process P_i sends a message m to P_j, it sets *m*'s (vector) timestamp *ts(m)* equal to VC_i after having executed the previous step.
3. Upon the receipt of a message *m*, process P_j adjusts its own vector by setting $VC_j[k] \leftarrow \max\{VC_j[k], ts(m)[k]\}$ for each *k*, after which it executes the first step and delivers the message to the application.

Example


P0
P1
P2
1
2
3
4
5
6
7
a
b
c
d
e
f
g
1
2
3
h
i
j
1
2
3
k
l
m
time


(0,0,0) P0
(0,0,0) P1
(0,0,0) P2
(1,0,0)
(2,0,0)
(3,1,0)
(4,1,0)
(5,1,2)
(6,1,2)
(7,1,2)
a
b
c
d
e
f
g
(0,1,0)
(2,2,0)
(6,3,2)
h
i
j
(0,0,1)
(0,0,2)
(4,1,3)
k
l
m
time

Lamport Vs Vector

Lamport's timestamps

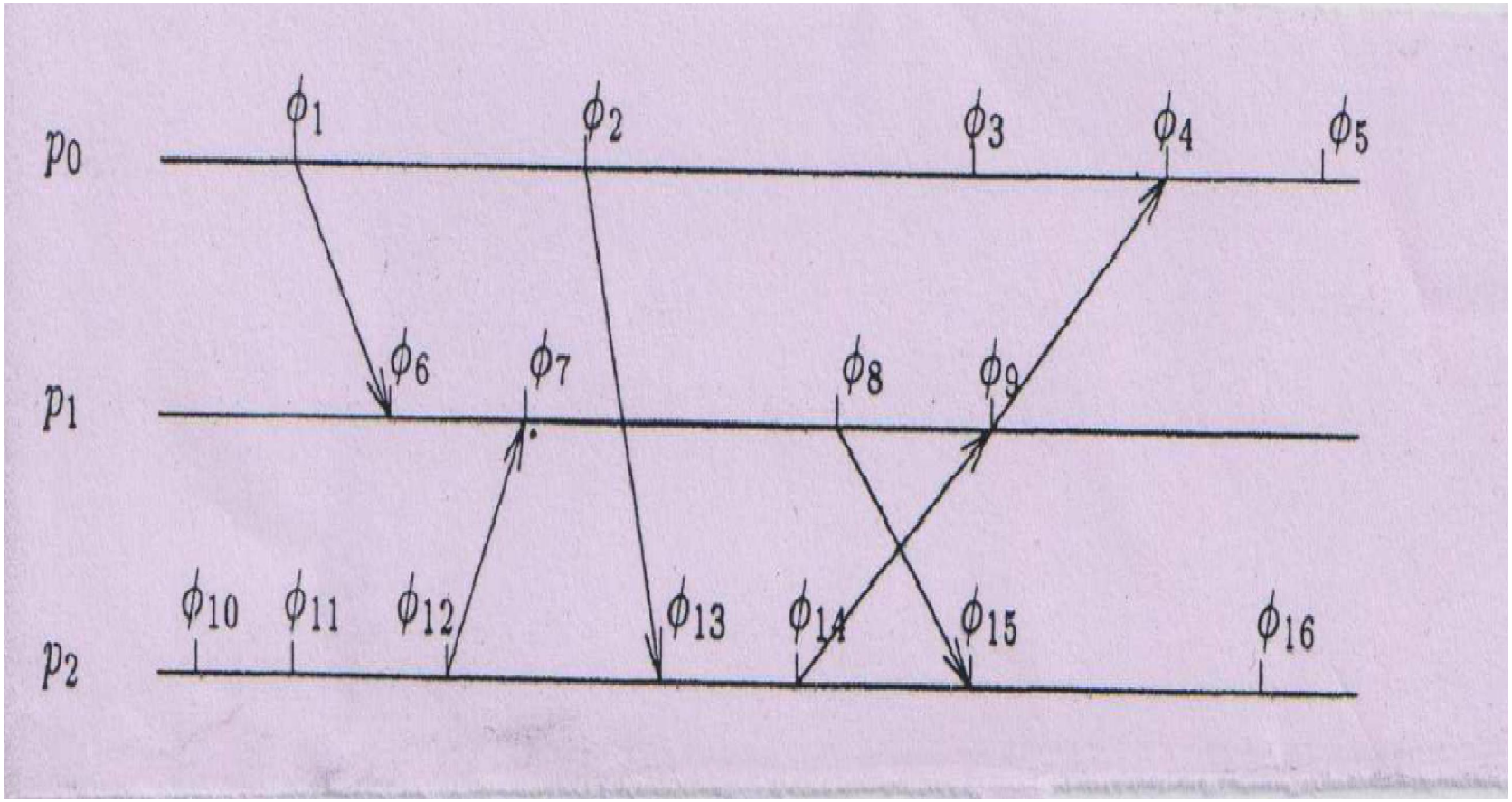
- Integer clocks assigned to events
- Obeys causality
- Cannot distinguish concurrent events

Vector timestamps

- Obeys causality
- By using more space, can also identify concurrent events

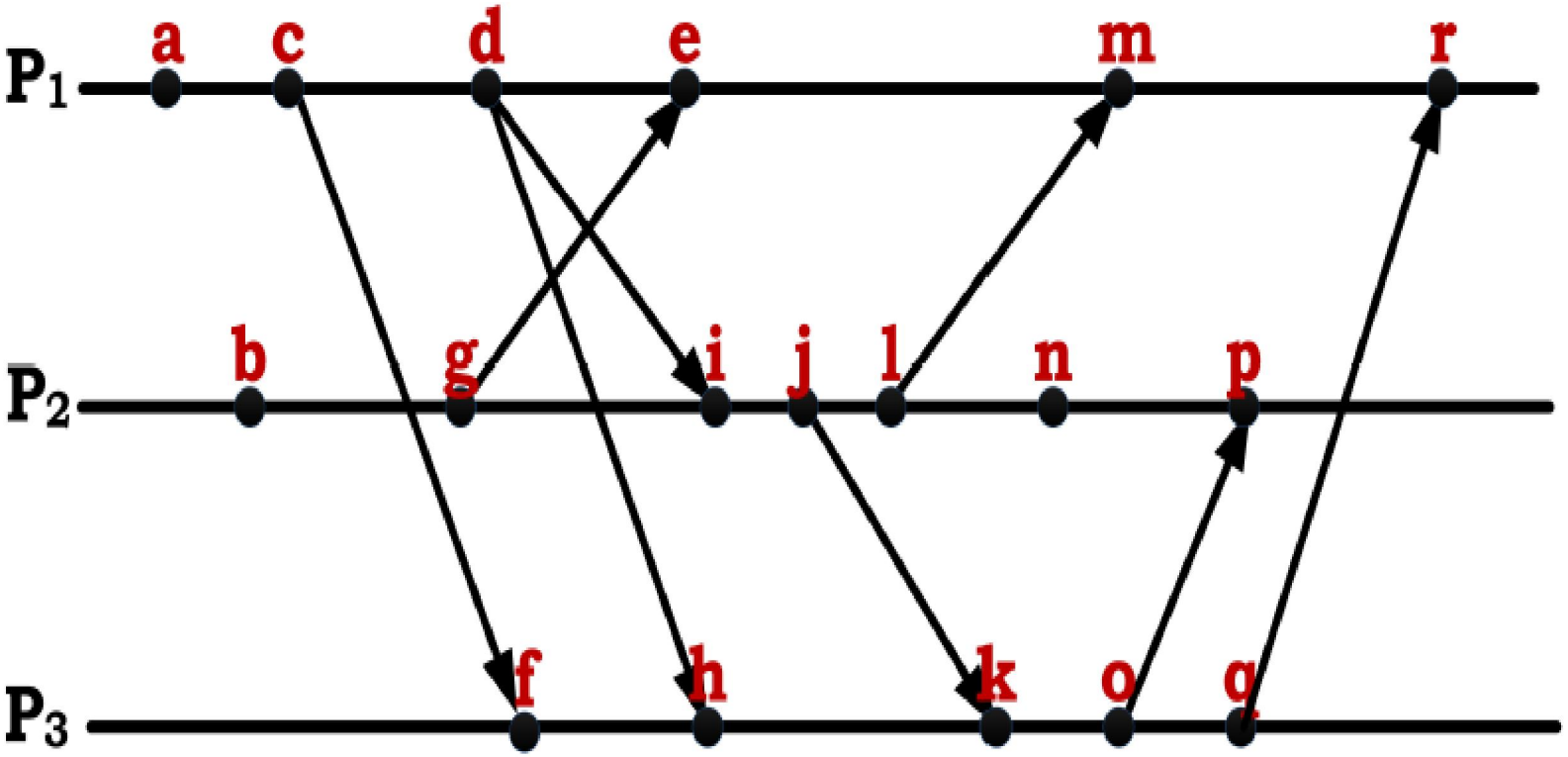
Tutorial- Lamport and Vector Clock

- For the figure below, write down the lamport clock and vector clock value at each event.



Tutorial- Lamport and Vector Clock

For the figure below, write down the lamport clock and vector clock value at each event.



GLOBAL STATE

Global State

- *"The global state of a distributed computation is the set of local states of all individual processes involved in the computation plus the state of the communication channels."*
- Knowing the global state of distributed system may be useful for many reasons.

Global State

Determining global properties:

often difficult, but essential for some applications.

- Distributed garbage collection:

  Do any references exist to a given object?:

  Collects remote server objects that are no longer referenced by any client in the network

- Distributed deadlock detection:

  Do processes wait in a cycle for each other?

- Distributed algorithm termination detection:

  Did a set of processes finish all activity? (Consider messages in transit)

All these properties are stable: once they occur, they do not disappear without outside intervention.

Analysis of Global State

(Distributed Snapshot)

- A distributed snapshot reflects a state in which the distributed system might have been.
- A snapshot reflects a consistent global state
- Example: if we have recorded that a process Q has received a message from another process P,
- then we should also recorded that process P has actually sent that message
- Otherwise, a snapshot will contain the recording of messages that have been received but never sent,
- Which indicate the inconsistent global state
- The reverse condition (P has sent a message that Q has not yet received) is allowed

More on States

- process state
  - memory state + register state + open files + kernel buffers + …

Or

  - application specific info like transactions completed, functions executed etc,.
- channel state
  - "*Messages in transit*" *i.e*. those messages that have been sent but not yet received

Why global state determination is difficult in Distributed Systems?

- Distributed State :

  Have to collect information that is spreaded across several machines!!

- Only Local knowledge :

  A process in the computation does not know the state of other processes.

Difficulties due to Non Determinism

- Deterministic Computation
  - At any point in computation there is at most one event that can happen next.

- Non-Deterministic Computation
  - At any point in computation there can be more than one event that can happen next.

Deterministic Computation Example

A Variant of producer-consumer example

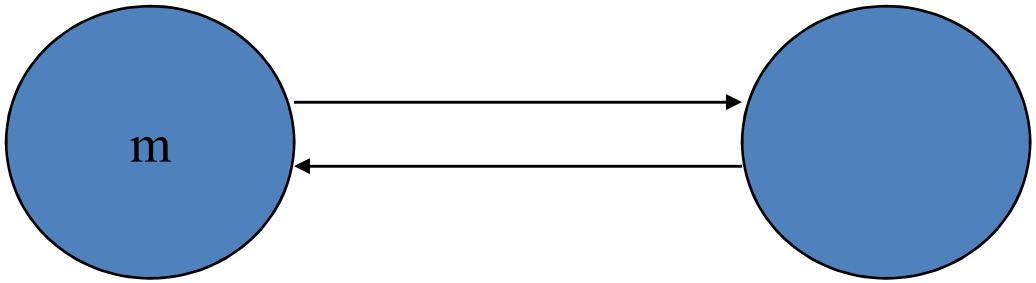
- Producer code:

```
while (1)
{
        produce m;
        send m;
        wait for ack;
}
```

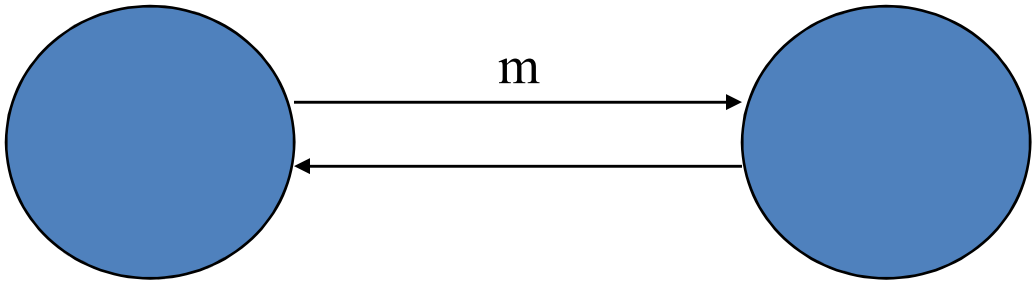
- Consumer code:

```
while (1)
{
        recv m;
        consume m;
        send ack;
}
```

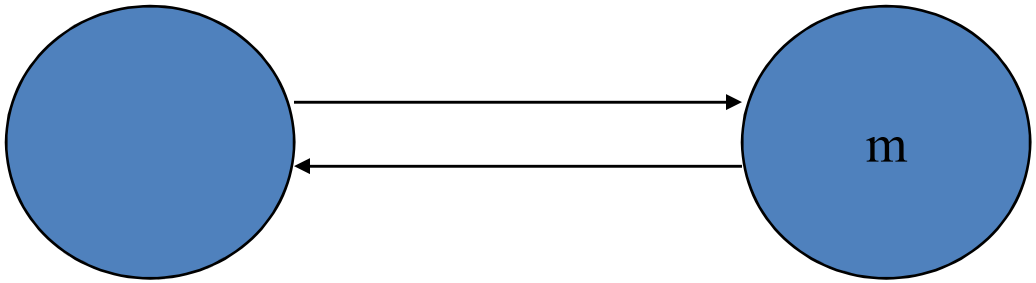
Example: Initial State



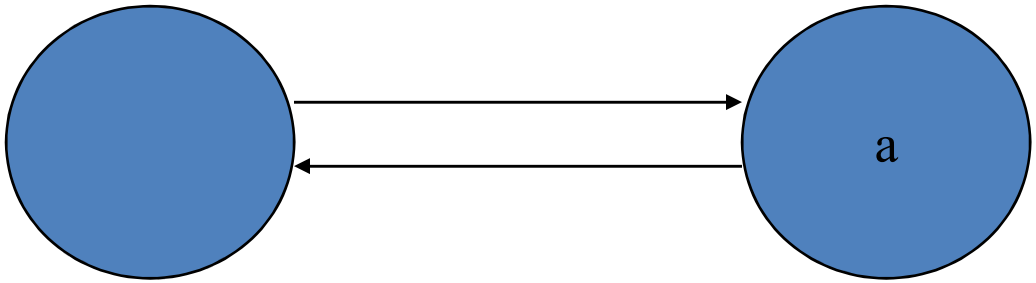
Example



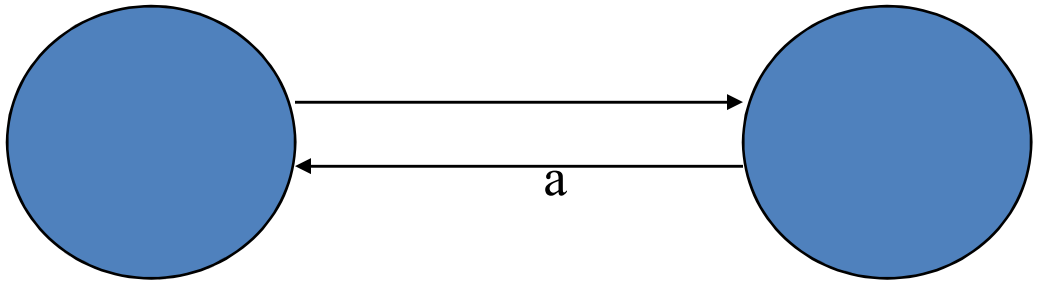
Example



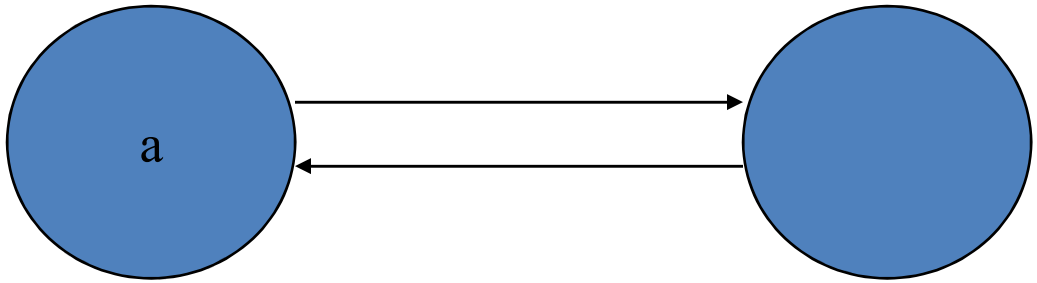
Example



Example



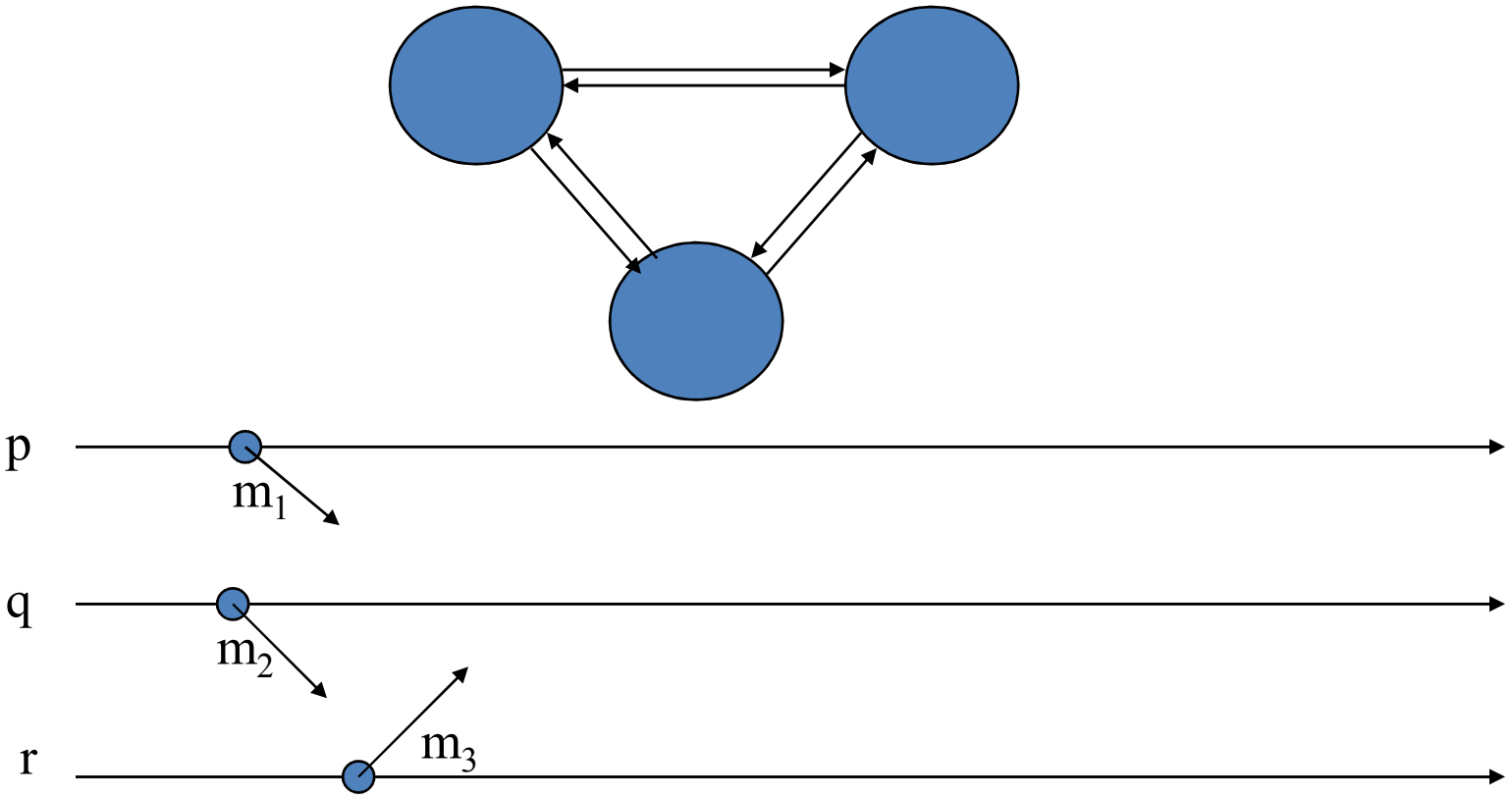
Example



Deterministic state diagram

Non-deterministic computation

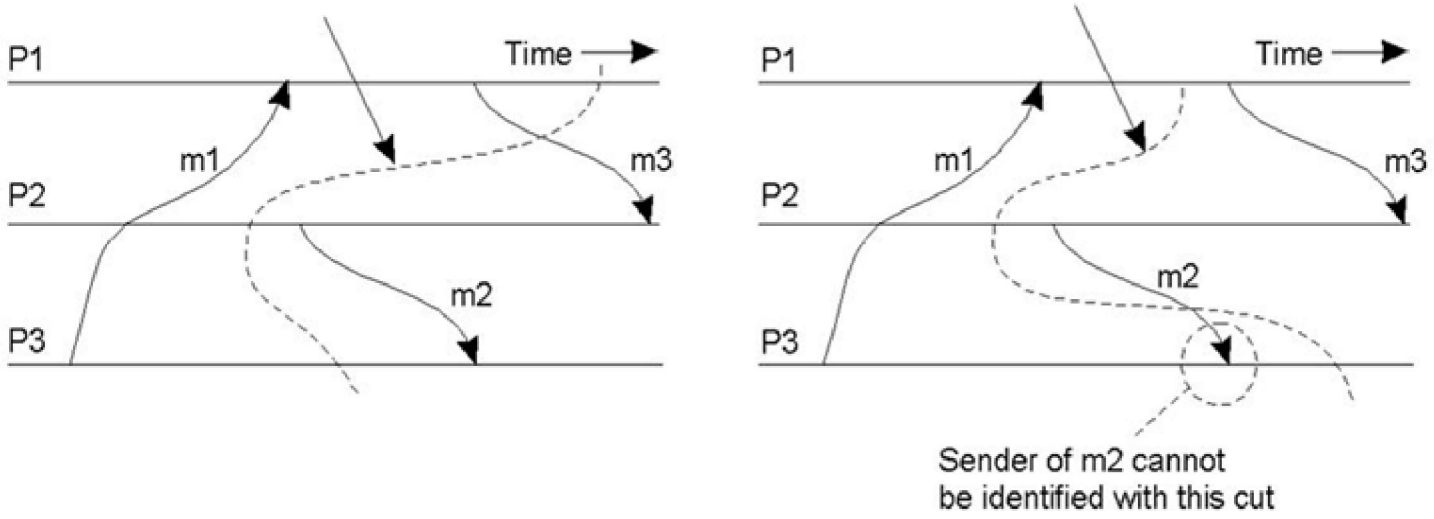
3 processes



Cut

- Global state can be graphically represented by cut
- It represents the last event recorded for each process
- 2 types of cut: consistent, inconsistent
- Consistent cut: all recorded message receipts have a corresponding recorded send event
- Inconsistent cut: no corresponding send event

Cut


P1
Time
m1
m3
P2
m2
P3
P1
Time
m1
m3
P2
m2
P3
Sender of m2 cannot
be identified with this cut

Election Algorithms

- Many Distributed Systems 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, assume the following:
  - Processes each 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

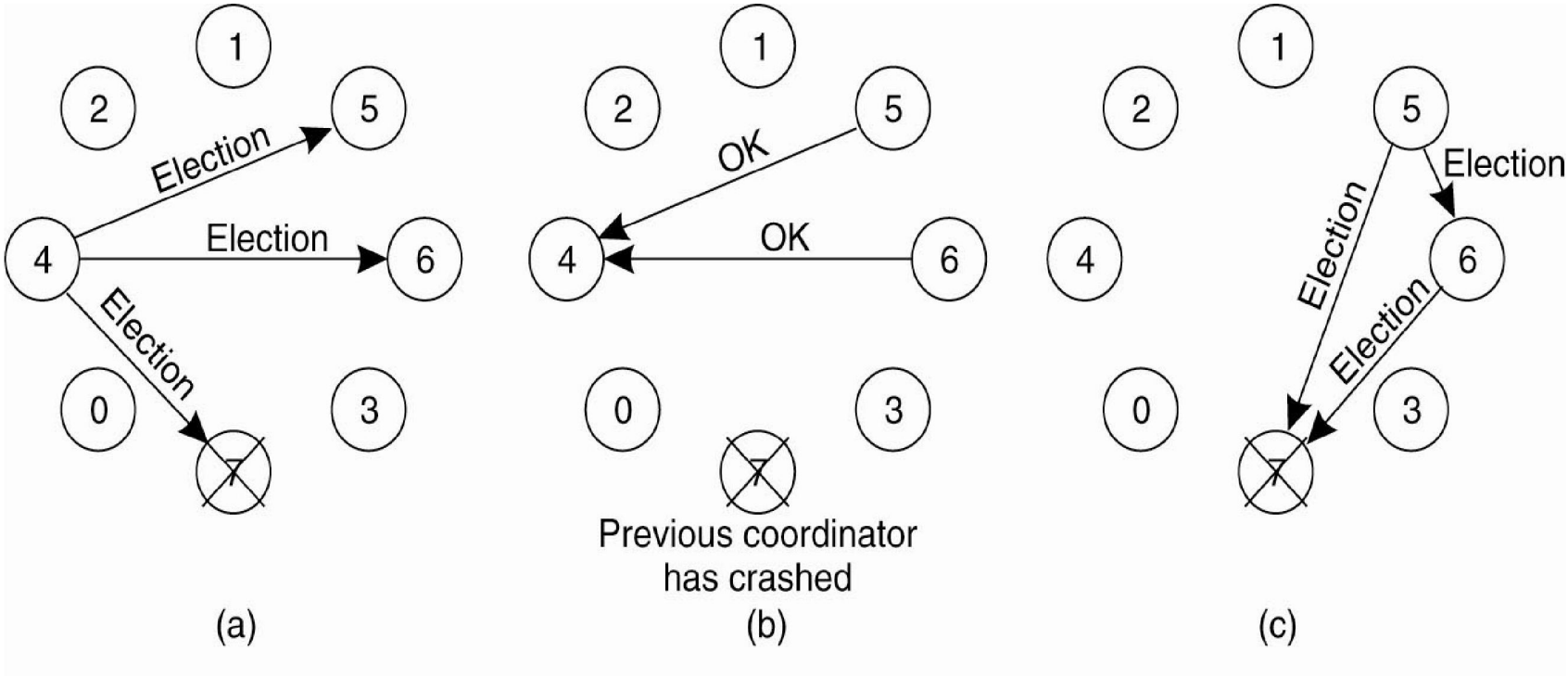
- The goal of all election algorithms is to have all the processes in a group *agree* on a coordinator.
- There are two types of election algorithm:

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

The Bully Algorithm

1. *P* sends an *ELECTION* message to all processes(with higher numbers)
2. If no one responds, *P* wins the election and becomes coordinator.
3. If one of the higher-ups answers, it takes over. *P*'s job is done.

The Bully Algorithm



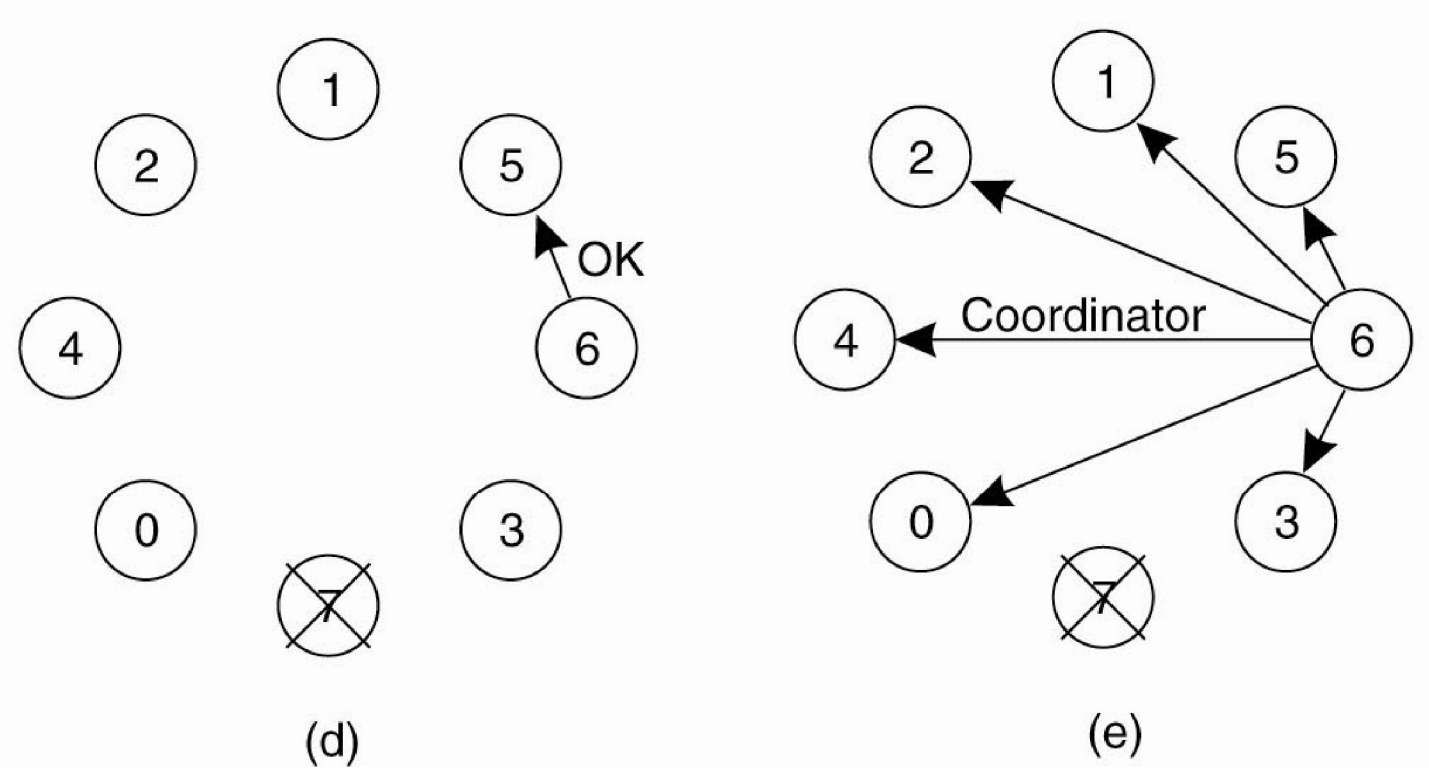
The bully election algorithm:

(a) Process 4 holds an election.

(b) 5 and 6 respond, telling 4 to stop.

(c) Now 5 and 6 each hold an election.

The Bully Algorithm



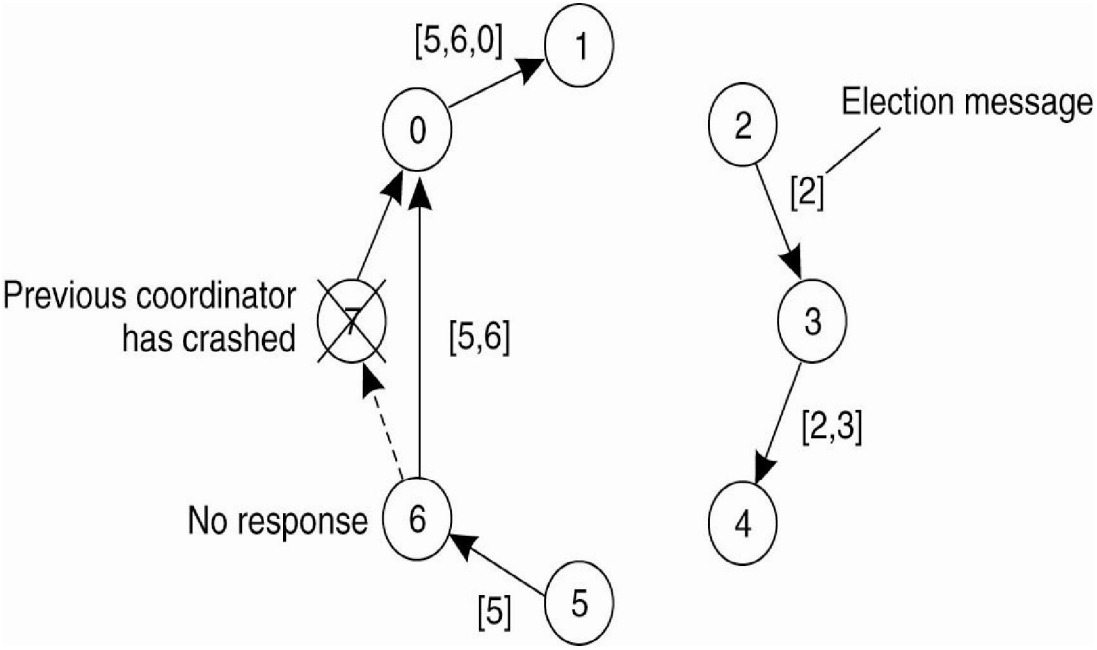
(d) Process 6 tells 5 to stop.

(e) Process 6 wins and tells everyone.

The “Ring” Election Algorithm

- The processes are ordered in a “logical/physical 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.

The Ring Algorithm



- Both process 2,and 5 discovered that coordinator 7 has been crashed – both will start ELECTION message and then COORDINATOR message - finalized 6 as coordinator.

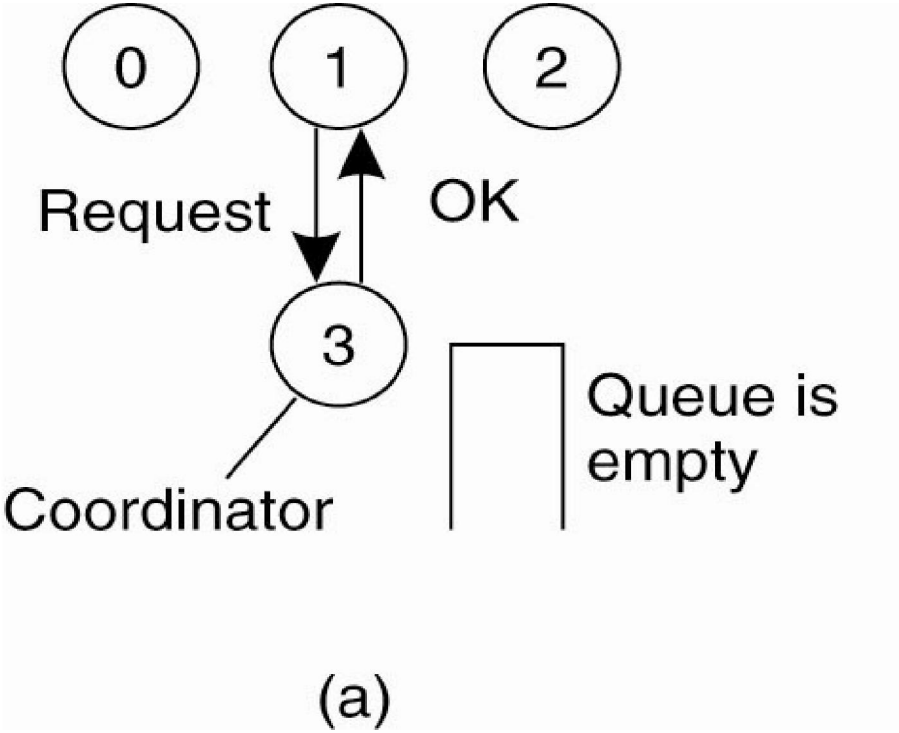
Mutual Exclusion in Distributed Systems

- It is necessary to protect a *shared resource* within a Distributed System using “mutual exclusion” – for example, it might be necessary to 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 – thus achieving *mutual exclusion*.
- These techniques have been adapted to Distributed Systems …

DS Mutual Exclusion Techniques

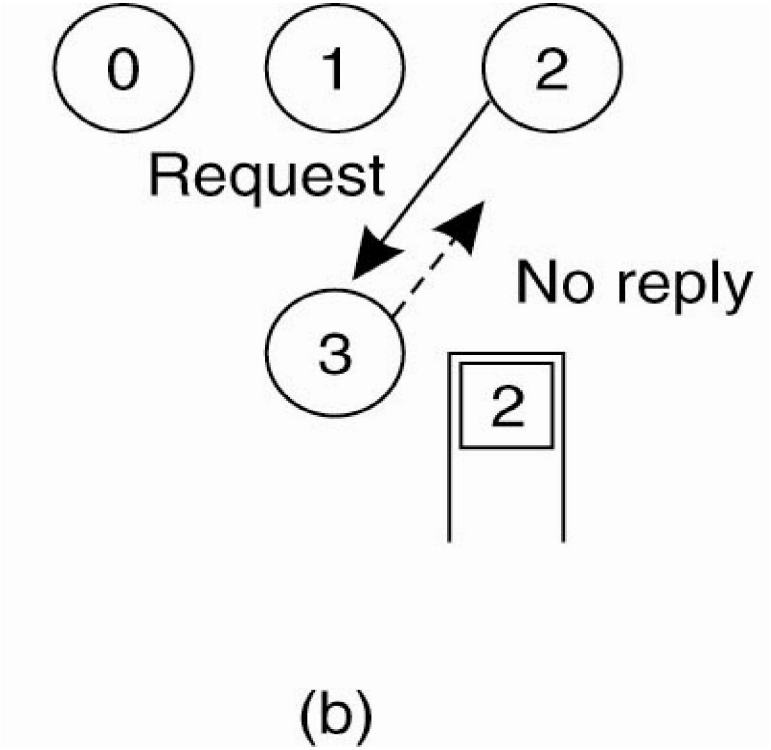
- **Centralized**: a single coordinator controls whether a process can enter a critical region.
- **Distributed**: the group *confers* to determine whether or not it is safe for a process to enter a critical region.

A Centralized Algorithm



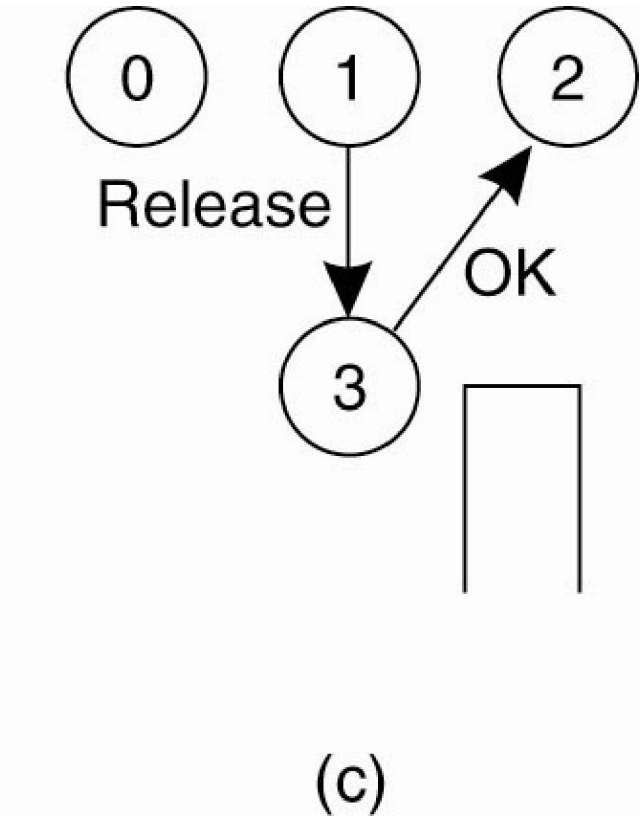
(a) Process 1 asks the coordinator for permission to access a shared resource. Permission is granted.

A Centralized Algorithm



b) Process 2 then asks permission to access the same resource. The coordinator does not reply.

A Centralized Algorithm



(c) When process 1 releases the resource, it tells the coordinator, which then replies to 2

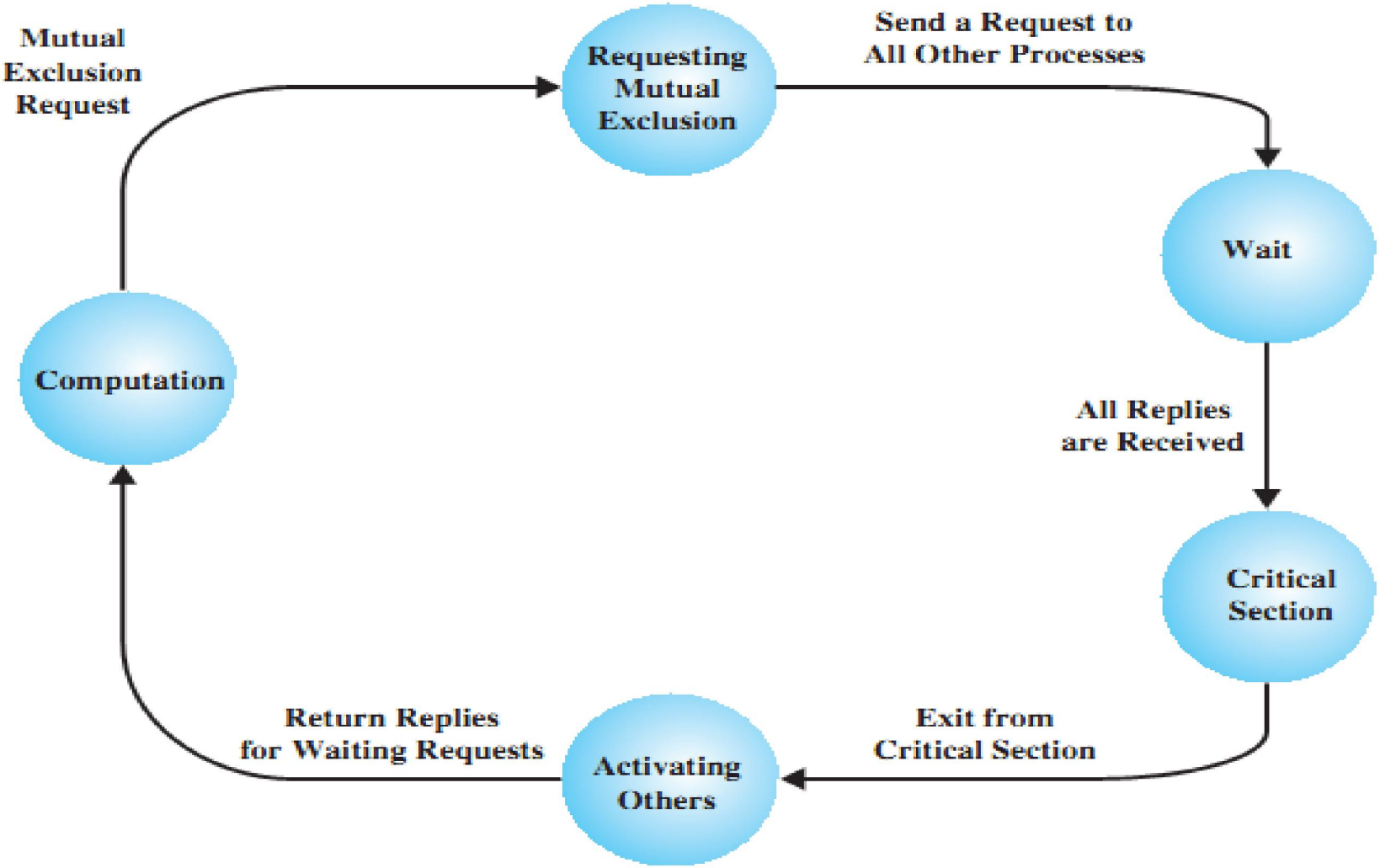
A Centralized Algorithm

- ***Advantages***:
  - There's no process starvation.
  - Easy to implement.
- ***Disadvantages***:
  - There's a single point of failure!
  - The coordinator is a bottleneck on busy systems.
- When there is no reply, does this mean that the coordinator is "dead" or just busy?

Distributed Algorithm

- Based on work by Ricart and 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.

State Diagram for a Process



Distributed Algorithm

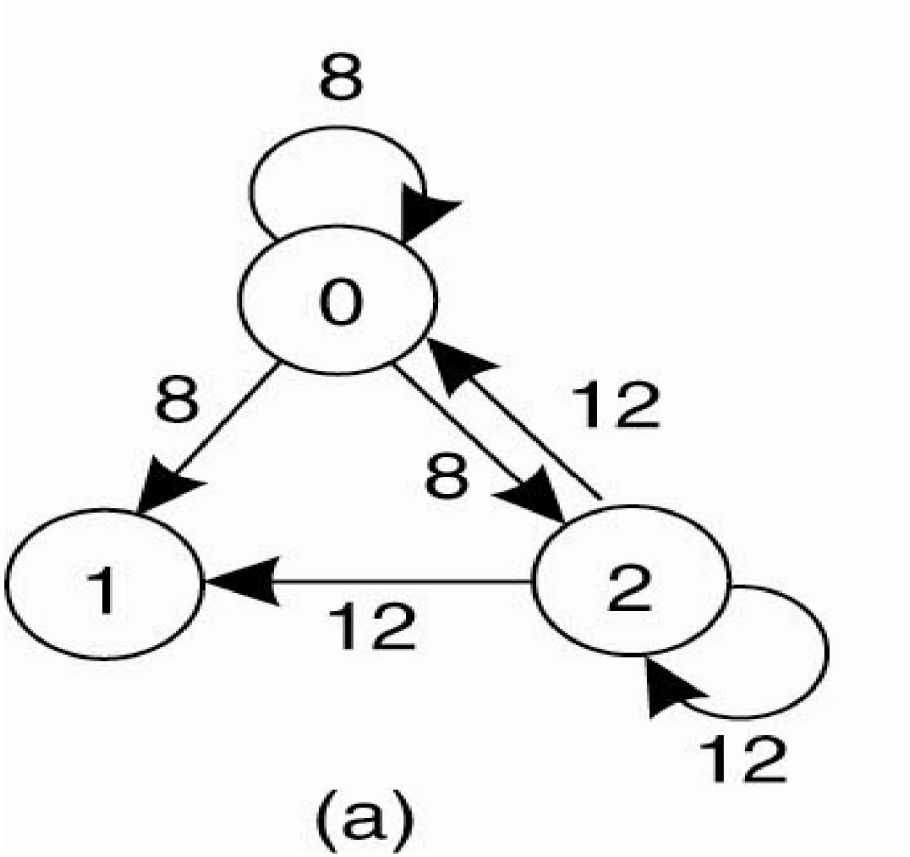
1. When a process (the “requesting process”) decides to enter a critical region, a message is sent to all processes in the Distributed System (including itself).
2. What happens at each process depends on the “state” of the critical region.
3. If not in the critical region (and not waiting to enter it), a process sends back an OK to the requesting process.
4. If in the critical region, a process will queue the request and will not send a 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 processes in its queue, then empties its queue.

Distributed Algorithm

- Three different cases:

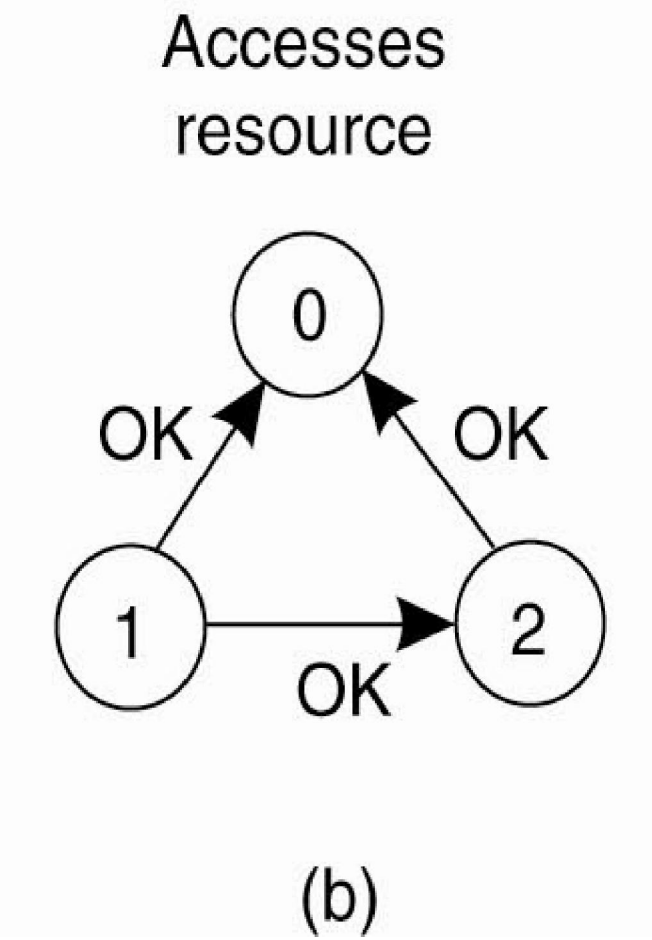
1. If the receiver is not accessing the resource and does not want to access it, it sends back an OK message
   to the sender.
2. If the receiver already has access to the resource, it simply does not reply. Instead, it queues the request.
3. If the receiver wants to access the resource as well but has not yet done so, it compares the timestamp of the incoming message with the one contained in the message that it has sent everyone. The lowest one wins.

Distributed Algorithm



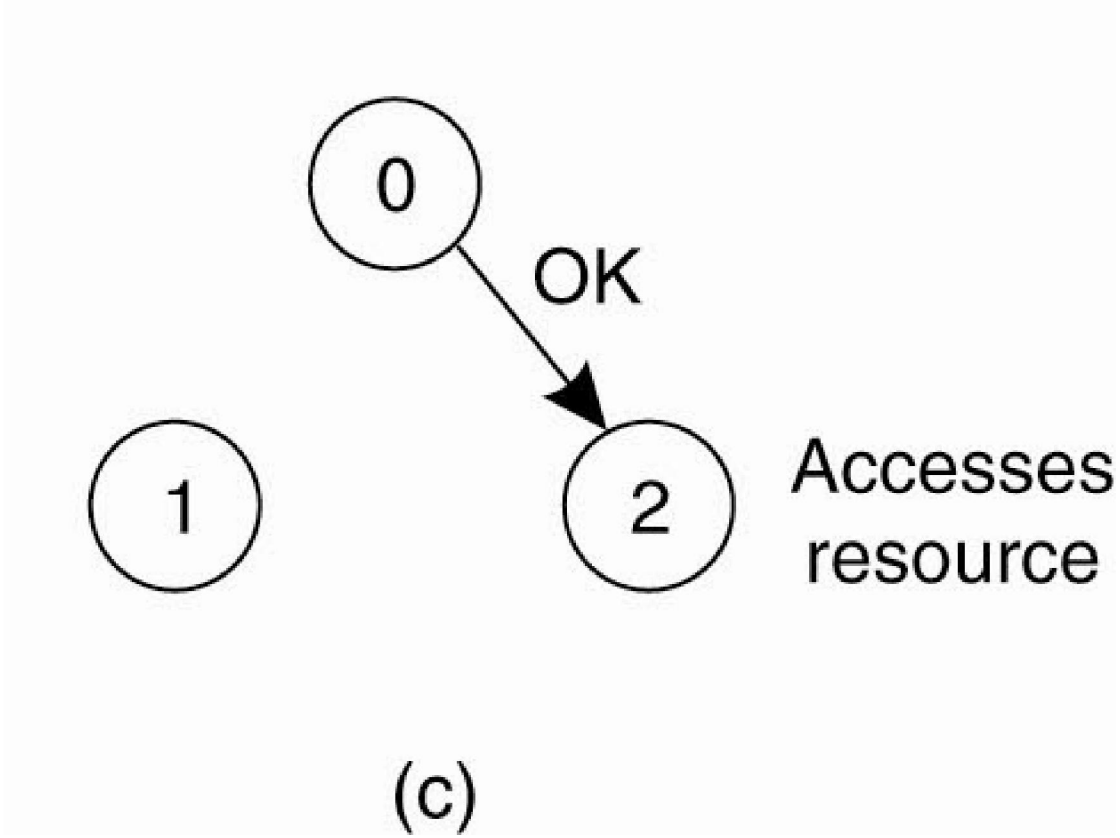
(a) Two processes want to access a shared resource at the same moment

Distributed Algorithm



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

Distributed Algorithm



(c) When process 0 is done, it sends an OK also, so 2 can now go ahead

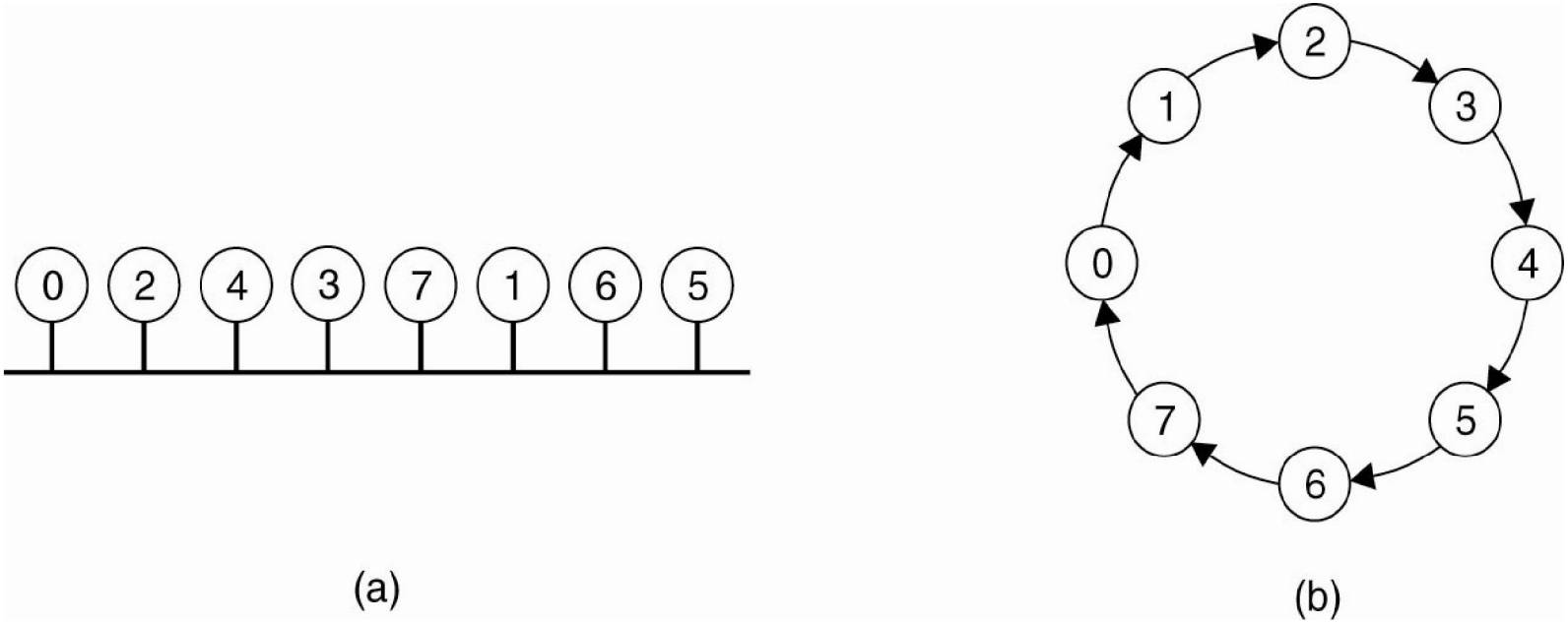
Distributed Algorithm

- 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**:
  - 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.

Distributed Algorithm

- It is always best to implement a distributed algorithm when a *reasonably good* centralized solution exists.
- What's good *in theory* (or on paper) may not be so good *in practice*.
- Think of all the message traffic this distributed algorithm is generating
- Every process is involved in the decision to enter the critical region, whether they have an interest in it or not.

Token Ring Algorithm



(a) An unordered group of processes on a network.

(b) A logical ring constructed in software.

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

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

- The “Centralized” algorithm is simple and efficient, but suffers from a single point-of-failure.
- The “Distributed” algorithm is slow, complicated, inefficient of network bandwidth, and not very robust.
- The “Token-Ring” algorithm sometimes take a long time to reenter a critical region having just exited it.

DISTRIBUTED TRANSACTIONS

Introduction to Transactions

- Related to Mutual Exclusion, which protects a “shared resource”.
- Transactions protect “shared data”.
- 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 to a previously “good state” whenever a transaction prematurely aborts.

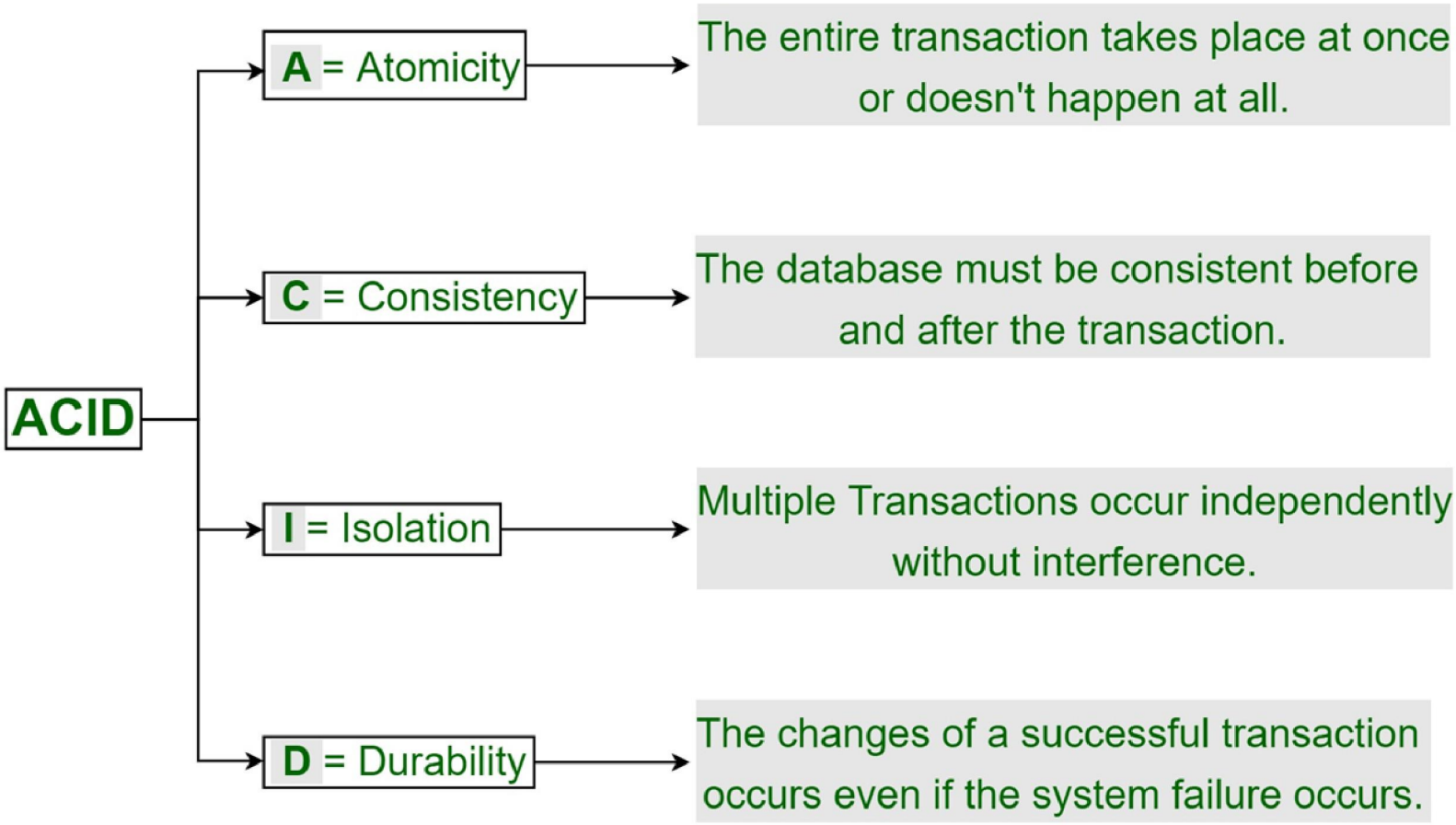
Transactions

- Mutual Exclusion :
  - Protect shared data against simultaneous access
  - Allow multiple data items to be modified in single atomic action

- Transaction Model:

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

ACID Properties



Classification Of Transactions

- **Flat Transaction**: this is the model that we have looked at so far. sequence of operations that satisfies ACID

  Disadvantage: too rigid…. Partial results cannot be committed.

- **Nested Transaction**: *hierarchy* of transactions. A main, parent transaction spawns child sub-transactions to do the real work.

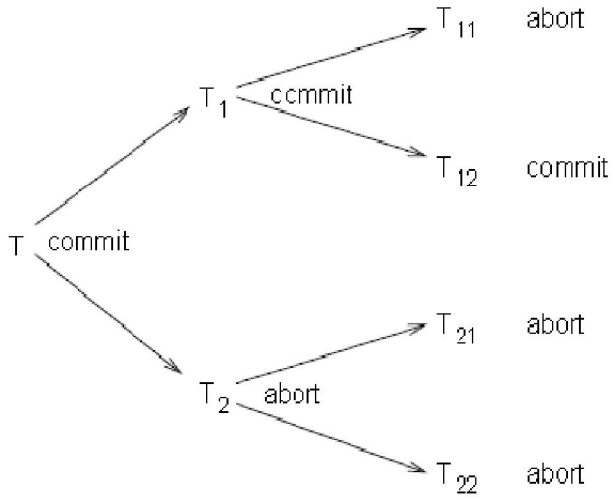
  Disadvantage: problems result when a sub-transaction commits and then the parent aborts the main transaction.

- **Distributed Transaction**: this is sub-transactions operating on distributed data stores.

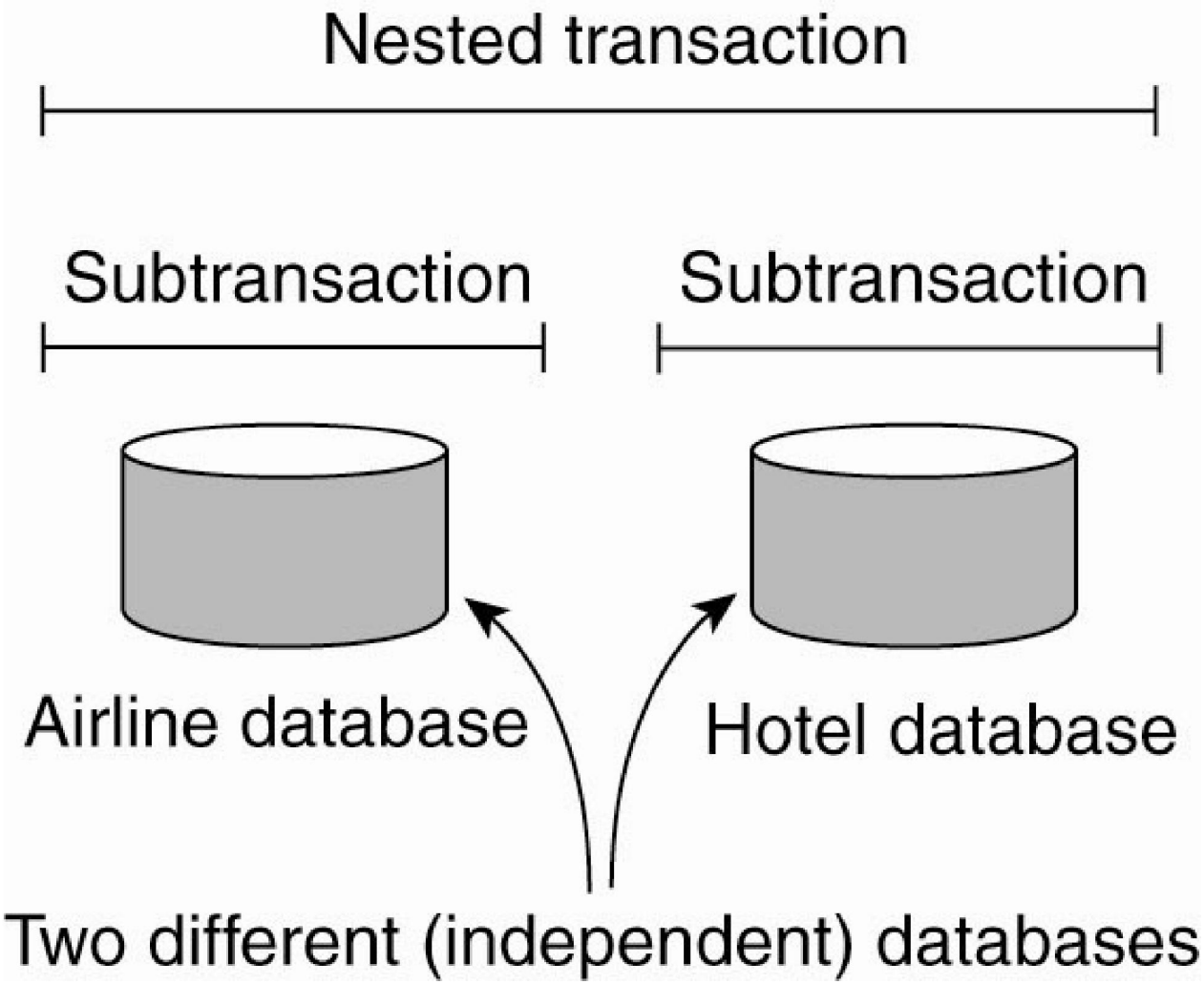
  Disadvantage: complex mechanisms required to lock the distributed data, as well as commit the entire transaction.

Nested Transaction

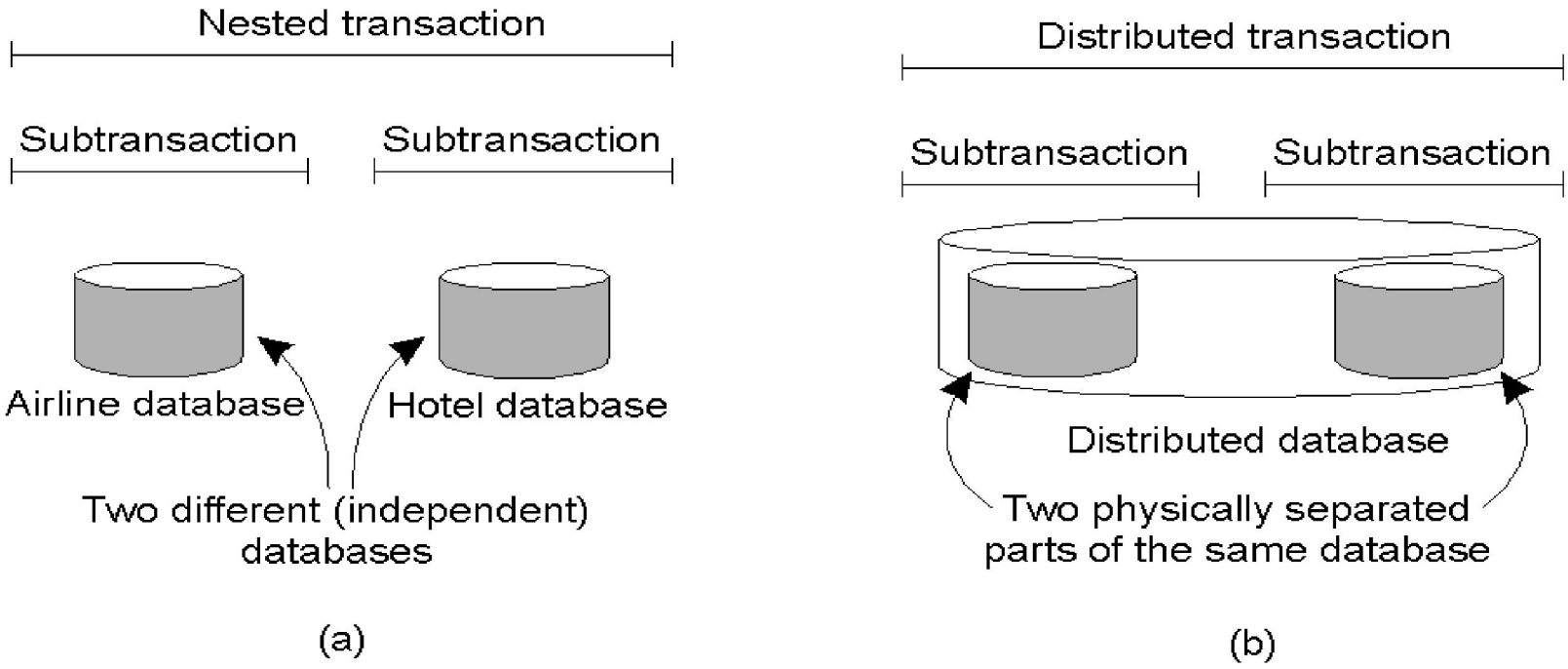
- Subdivide a complex transaction into many smaller *sub-transactions*
- As a result, if a single *sub-transaction* fails, the work that went into others is not necessary wasted
- Example:
  - Booking a flight
    - Sydney Manila
    - Manila Amsterdam
    - Amsterdam Toronto
- What to do?
  - Abort whole transaction
  - Partially commit transaction and try alternative for aborted part
  - Commit non aborted parts of transaction



Nested Transaction



Nested vs. Distributed Transactions



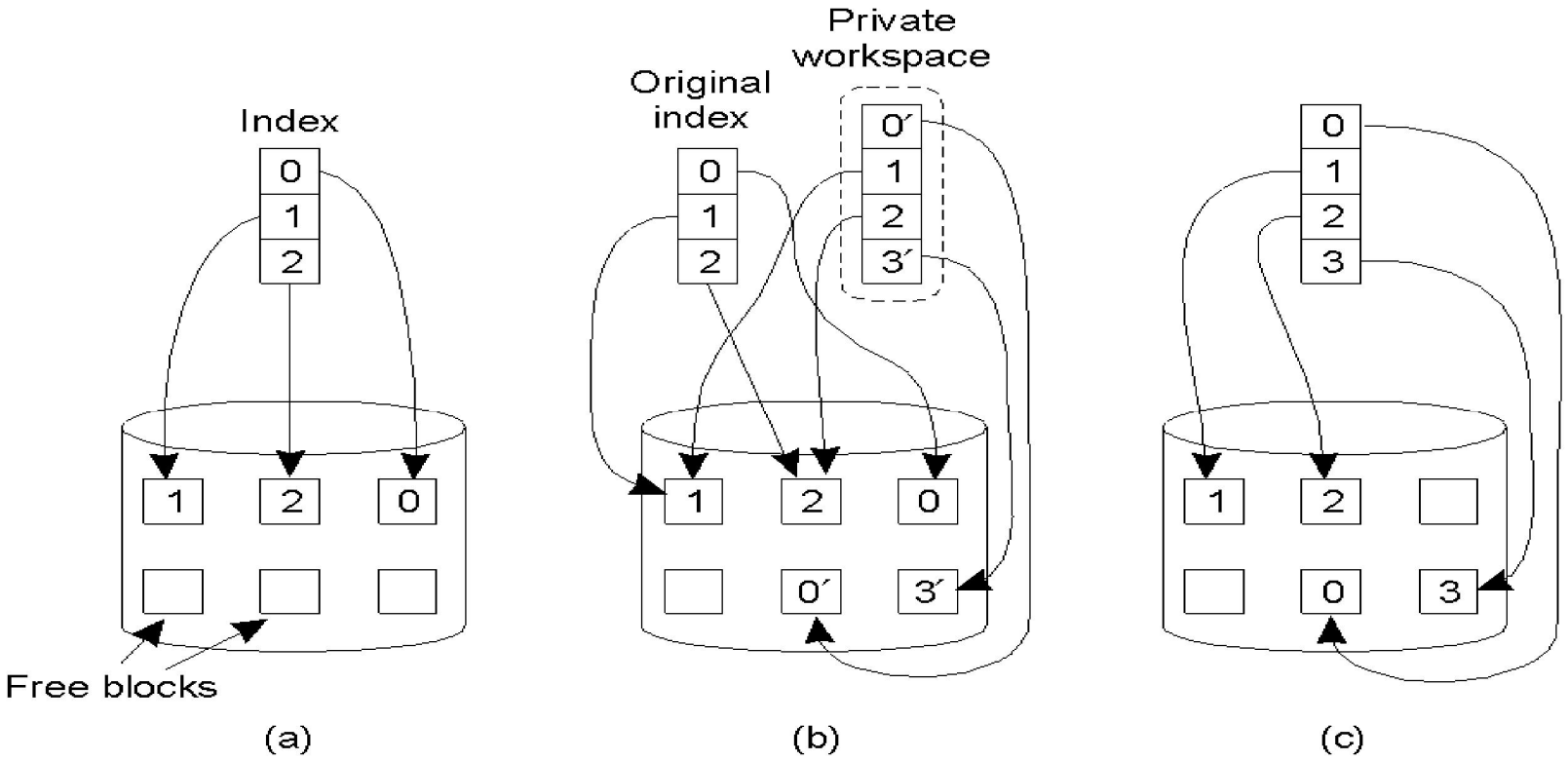
- A nested transaction is a transaction that is logically decomposed into a hierarchy of sub transactions.
- A distributed transaction is logically a flat, indivisible transaction that operates on distributed data

Transaction Implementation

Private Workspace

- When a process starts a transaction, it is given private workspace containing all the files to which it has access
- Until the transaction commits/aborts, all the reads/writes go to the private workspace instead of file system.

Private Workspace



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

- Actual files get modified
- Before any block is changed, a record is written to a log
- Log is reverted when a transaction Aborts

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(old/new values) before each statement is executed

Concurrency Control

- It is often necessary to allow transactions to occur simultaneously (for example, to allow multiple travel agents to simultaneously reserve seats on the same flight, Clients accessing bank accounts).
- Due to the consistency and isolation properties of transactions, concurrent transaction must not be allowed to interfere with each other.
- **Concurrency control algorithms** for transactions guarantee that multiple transactions can be executed simultaneously while providing a result that is the same as if they were executed one after another.(consistent state)

Concurrency Control

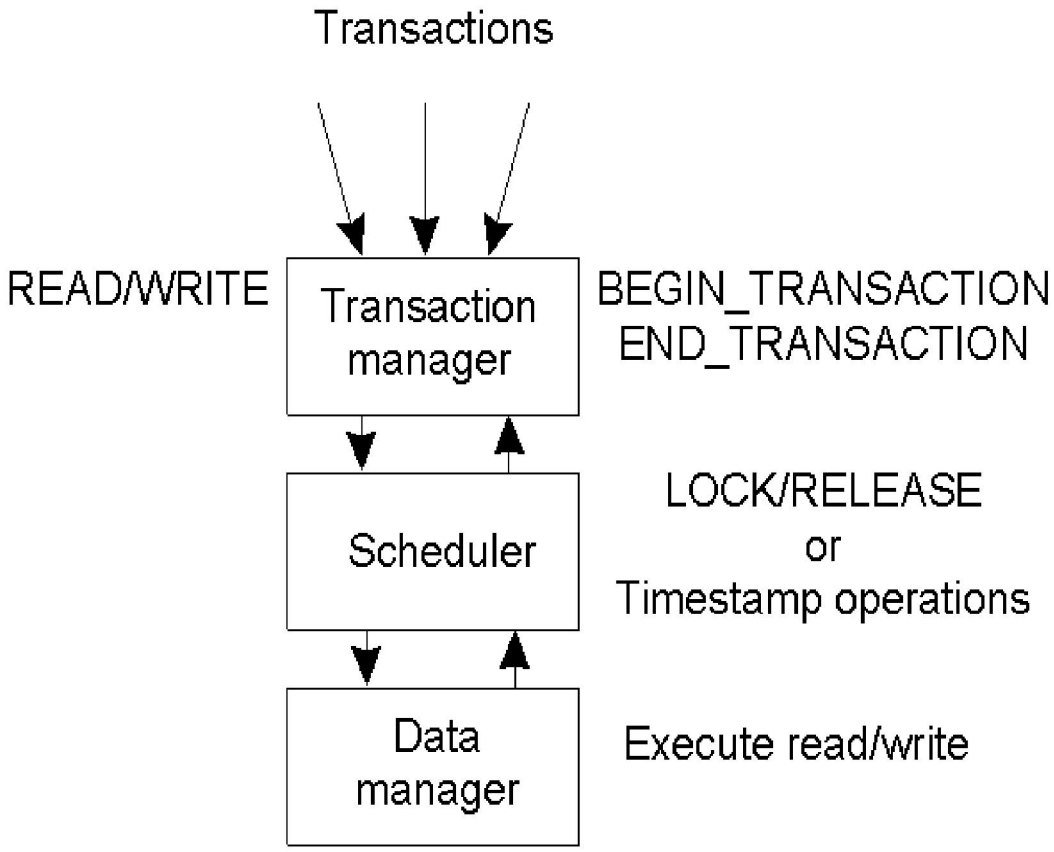
- Concurrency control is best understood in terms of three different managers

(1) Data manager – performs actual read and write operations

(2) Scheduler – mainly responsible to control concurrency in such a way that isolation and consistency are met

(3) Transaction manager – mainly responsible for guaranteeing atomicity of transactions

Concurrency Control



General organization of managers for handling transactions.

Conflicts

- **Idea behind concurrency control – properly schedule conflicting operations**
- **conflict:** operations (from the same, or different transactions) that operate on same data
- **read-write conflict:** one of the operations is a write
- **write-write conflict:** more than one operation is a write
- **Schedule*:***
  - Total ordering (interleaving) of operations
  - Legal schedules provide results as though the transactions were serialized (i.e., performed one after another) (*serial equivalence*)

Serializability

- Main purpose of concurrency control algorithms is to 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 other in some specific order
- Which of the following schedules are serialized? A->b->c, order is different…c->b->a

```
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	?
Schedule 2	x = 0; x = 0; x = x + 1; x = x + 2; x = 0; x = x + 3;	?
Schedule 3	x = 0; x = 0; x = x + 1; x = 0; x = x + 2; x = x + 3;	?

(d)

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

b) d) Possible schedules

Need for concurrency control

- **Lost update problem**

Time	T_1	T_2	Value
1	read_item(X);		X = 80
2	X:=X-N;		X =80–5=75 (which is not written into database)
3		read_item(X);	X = 80 (T2 still reads in the original value of X, the updated value of X is lost.)
4		X:=X+M	X=80+4=84
5	write_item(X);		X=75 is written into database
6	read_item(Y);		
7		write_item(X);	X=84 over writes X=75, a wrong record is written in database
8	Y:=Y+N;		
9	write_item(Y);		

- **Uncommitted dependency/dirty read**

Time	T_1	T_2	Comment
1	read_item(X);		
2	X:=X-N;		
3	write_item(X);		X is temporarily updated
4		read_item(X);	
5		X:=X+M	
6		write_item(X);	
7		**COMMIT**	**T_2 loses update chance**
8	read_item(Y);		
...	...	...	
	ROLLBACK		**T_2 depends on uncommitted value and loses an update at time step 7**

Need for concurrency control

- **Inconsistent analysis**

Time	T_1	T_3	Comment
1		sum:=0;	
2		read_item(A);	
3		sum:=sum+A;	
4	read_item(X);		
5	X:=X-N;		
6	write_item(X);		
7		read_item(X);	**T_3 reads X after N is subtracted from it**
8		sum:=sum+X;	
9		read_item(Y);	**T_3 reads Y before N is added to it**
10		sum:=sum+Y;	**A wrong summary is resulted (off by N)**
11	read_item(Y);		
12	Y:=Y+N;		
13	write_item(Y);		

Dealing with Concurrency

- Concurrency control algorithms can be classified by looking at the way read and write operations are synchronized.
- Synchronization can take place either through mutual exclusion mechanisms on shared data(i.e. locking) or explicitly ordering operations using timestamps.

Dealing with Concurrency

- Pessimistic vs Optimistic concurrency control approaches
- Pessimistic – Murphy's law: if something can go wrong, it will. Operations are synchronized/ conflicts are resolved before they carried out/happen.
- Optimistic – based on idea: nothing will go wrong. Synchronization takes place at the end of transaction. If there is conflict, one or more transactions are forced to abort!

- Locking (mutual exclusion) - pessimistic
- Timestamp Ordering (explicit ordering)- pessimistic
- Optimistic Control

Locking

- Pessimistic approach(Murphy's law): prevent illegal schedules
  - Operations are synchronized before they are carried out
  - Process requests the scheduler to grant a lock for data item and when no longer needed it is requested to release a lock.
  - The scheduler is responsible for granting and releasing locks in such a way that legal schedules are produced.
  - Ensures that only valid schedules result

Basic Two Phase Locking (2PL)

- In two-phase locking(2PL), the scheduler first acquires all the locks in needs during the growing phase and then releases them during shrinking phase with following 3 rules:

(1) When the scheduler receives an operation oper(T,x) from the transaction manager, it tests whether that operation conflicts with any other operation for which it already granted a lock. If there is a conflict, oper(T,x) is delayed and hence transaction T also. If no conflict, the scheduler grants a lock for data item x, and passes the operation to the data manager.

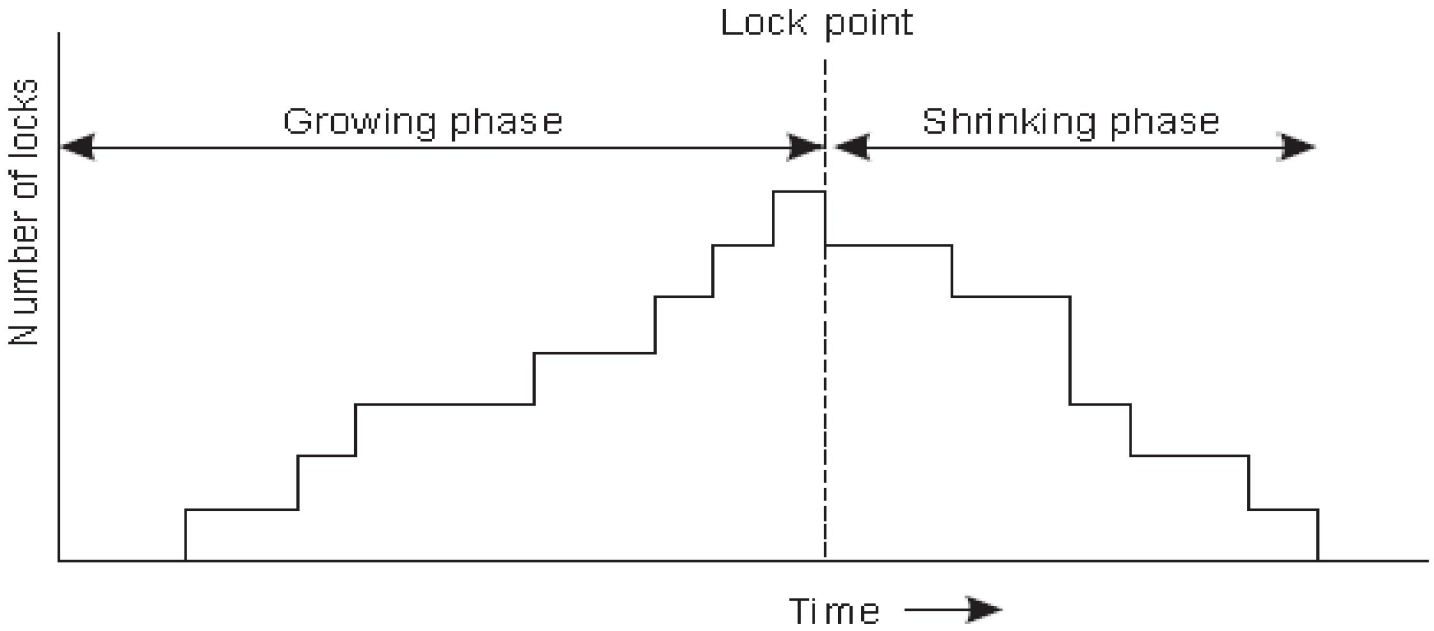
(2) The scheduler will never release a lock for data item x, until the data manager acknowledges it has performed the operation for which the lock was set.

(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.

- **Conservative/Static 2PL** – lock all the items it accesses before the transaction begins execution

Basic Two Phase Locking (2PL)

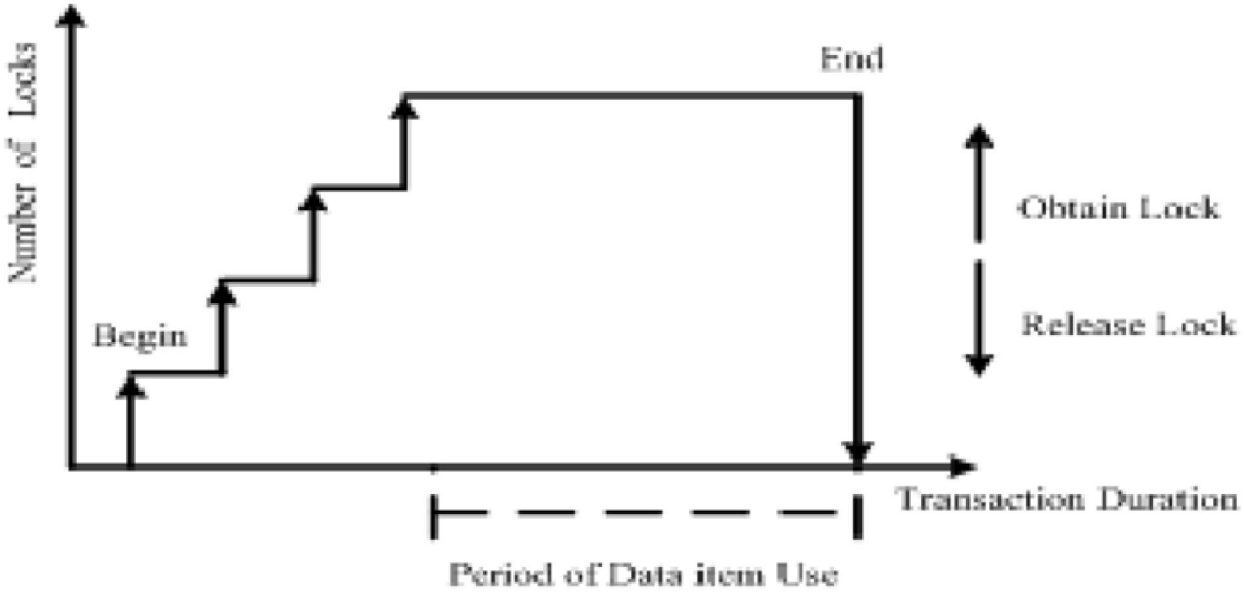
- Scheduler first acquires all the locks it needs during the growing phase
- It is proven by Eswaran et al. in 1976 that if all transactions use 2PL, all schedules formed by interleaving them are serializable.



Problems With Locking

- Cascaded Aborts: If a transaction (T1) reads the results of a write of another transaction (T2) that is subsequently aborted, then the first transaction (T1) will also have to be aborted.
- Exclusive(r&w) and shared(r) locks
- Solution – strict 2PL, where shrinking phase(wrt exclusive locks) does not take place until the transaction has finished running and has either committed or aborted.
- Rigorous 2PL – both locks will be released at the end
- Problem in above both: Deadlock- two processes try to acquire the same pair of locks in ?

Strict two-phase locking



Timestamp Ordering

- Lock methods maintains serializability by mutual exclusion
- Timestamp ordering defines the serialization order of transactions -> generate serial schedule
- Each transaction Ti is assigned with unique timestamp (*ts*(*Ti*)) when it starts.
- Timestamps are unique – using Lamport's algorithm
- Each data item has two timestamps:
  - read timestamp: $ts_{RD}(x)$ - the timestamp of the last read x
  - write timestamp: $ts_{WR}(x)$ - the timestamp of the last committed write x

- Timestamp ordering rule:
  - write request only valid if $TS(W) > ts_{WR}$ and $TS(W) \geq ts_{RD}$
  - read request only valid if $TS(R) > ts_{WR}$
- Conflict resolution:
  - Operation with lower timestamp executed first

Optimistic Control

- Assume that no conflicts will occur.
  - Detect & resolve conflicts at commit time
  - A transaction is split into three phases:
    - Working
    - Validation
    - Update
  - In the working phase operations are carried out on shadow copies with no attempt to detect or order conflicting operations.

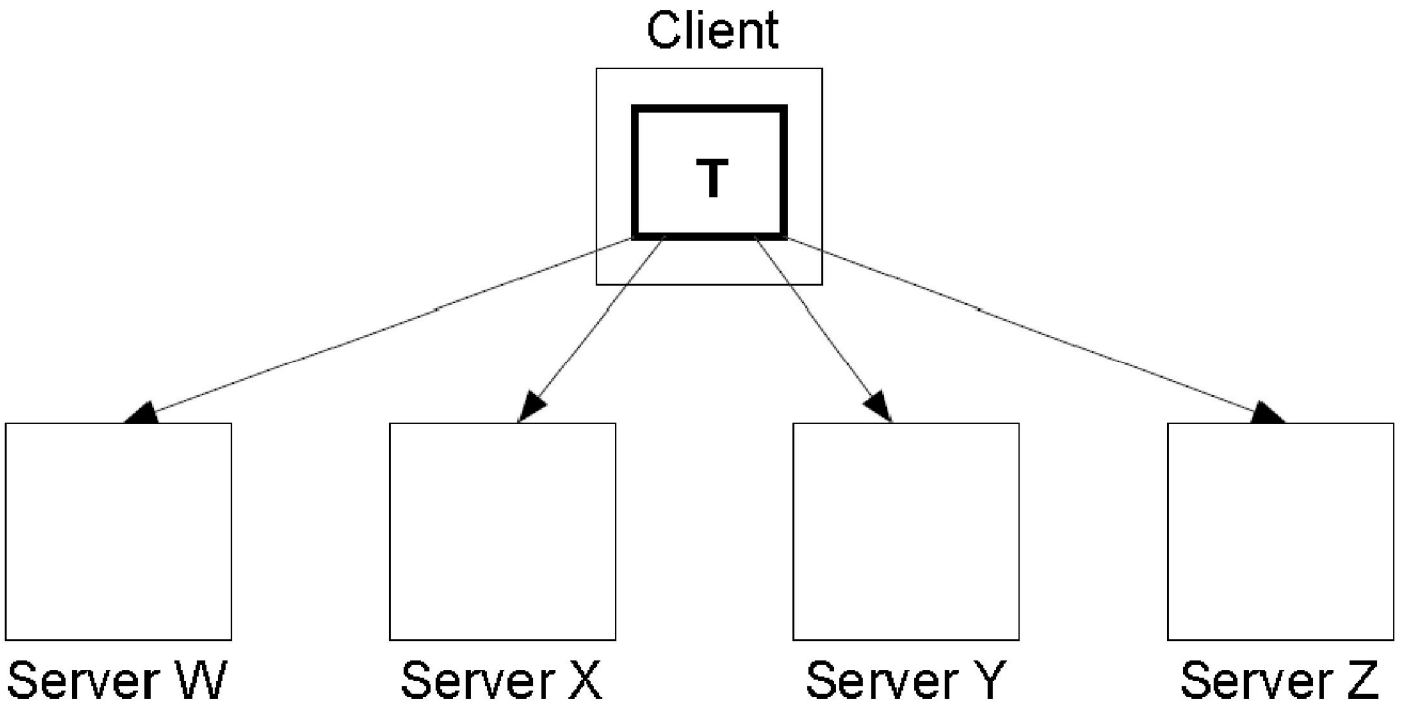
Cont….

- In the validation phase the scheduler attempts to detect conflicts with other transactions that were in progress during the working phase.
- If conflicts are detected then one of the conflicting transactions is aborted.
- In the update phase, assuming that the transaction was not aborted, all the updates made on the shadow copy are made permanent.

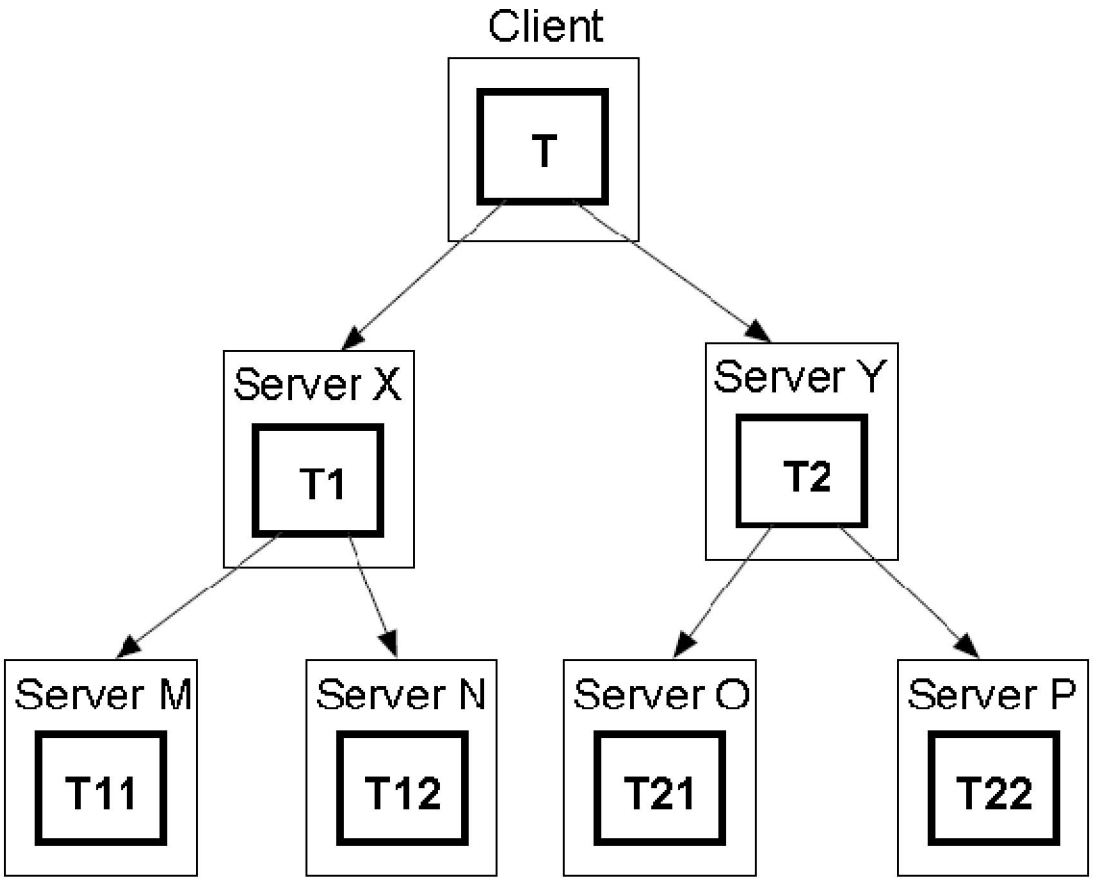
Distributed Transactions

- In distributed system, a single transaction will, in general, involve several servers:
  - transaction may require several services,
  - transaction involves files stored on different servers
- All servers must agree to *Commit* or *Abort*, and do this atomically.

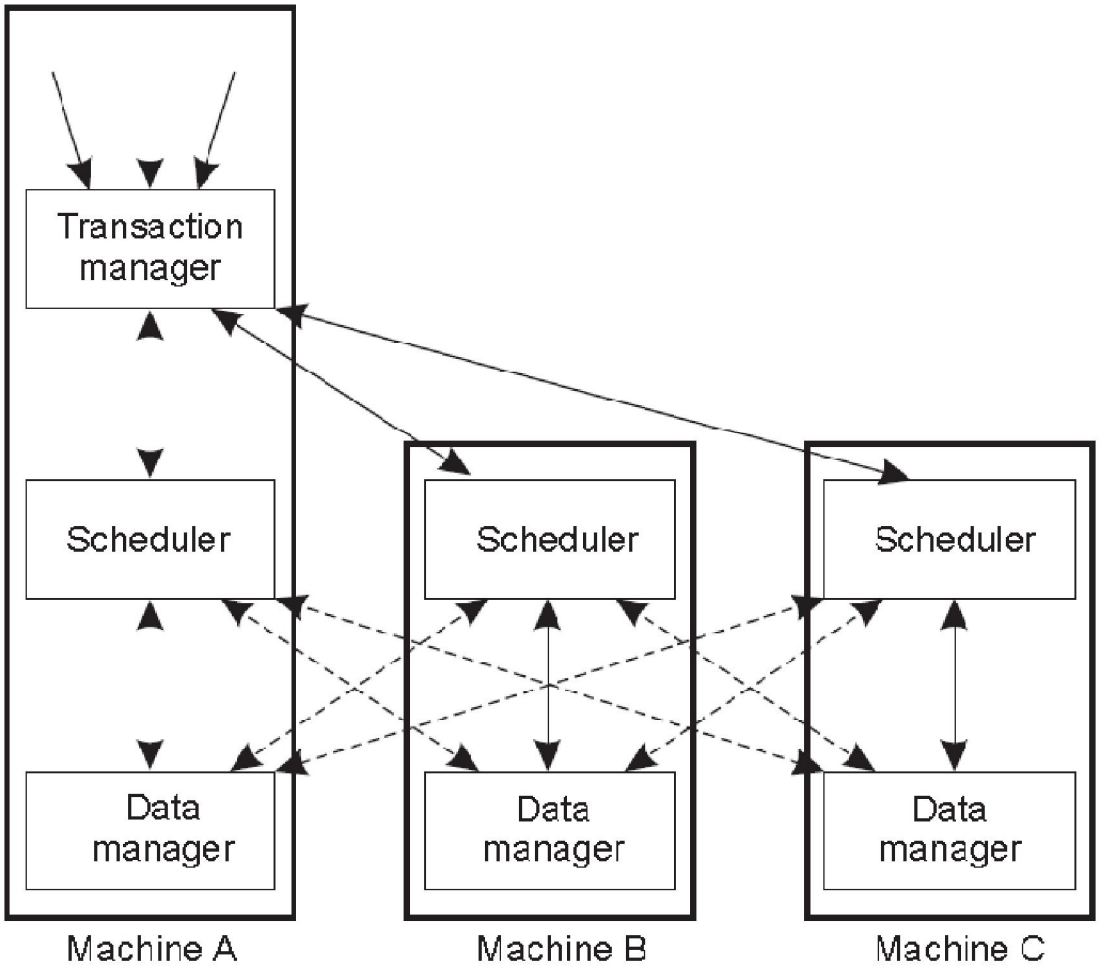
Distributed Flat Transaction



Distributed Nested Transaction



Distributed Concurrency Control



Distributed Transaction Organization

- Each distributed transaction has a coordinator, the server handling the initial **BeginTransaction** call
- Coordinator maintains a list of workers, i.e. other servers involved in the transaction
- Each worker needs to know coordinator
- Coordinator is responsible for ensuring that whole transaction is atomically committed or aborted
  - Require a distributed commit protocol.

Distributed Atomic Commit – 2PC Protocol

- Transaction may only be able to commit when all workers are ready to commit (e.g. validation in optimistic concurrency)
- Two phase commit (2PC) is the standard protocol for making commit and abort atomic
- Distributed commit requires at least two phases:
  1. Voting phase: all workers vote on commit, coordinator then decides whether to commit or abort.
  2. Completion phase: all workers commit or abort according to decision.
- Basic protocol is called two-phase commit (2PC)
- Example – transfer amount X from bank A to bank B

The Protocol

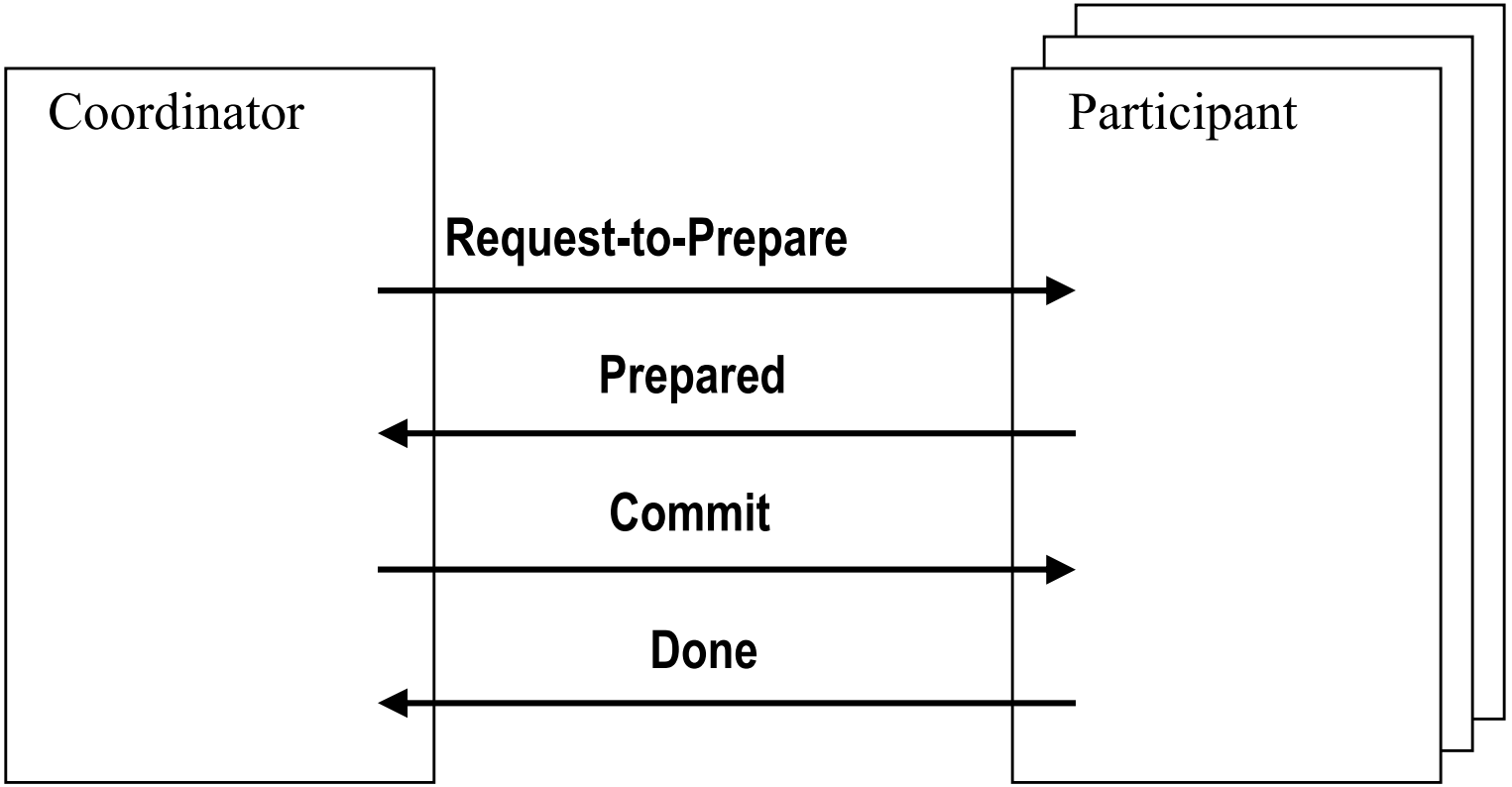
1. (Begin Phase 1) The coordinator sends a Request-to-Prepare message to each participant
2. The coordinator waits for all participants to vote
3. Each participant
   - votes Prepared if it's ready to commit
   - may vote No for any reason
   - may delay voting indefinitely
4. (Begin Phase 2) If coordinator receives Prepared from all participants, it decides to commit.
   (The transaction is now committed.)
   Otherwise, it decides to abort.

The Protocol

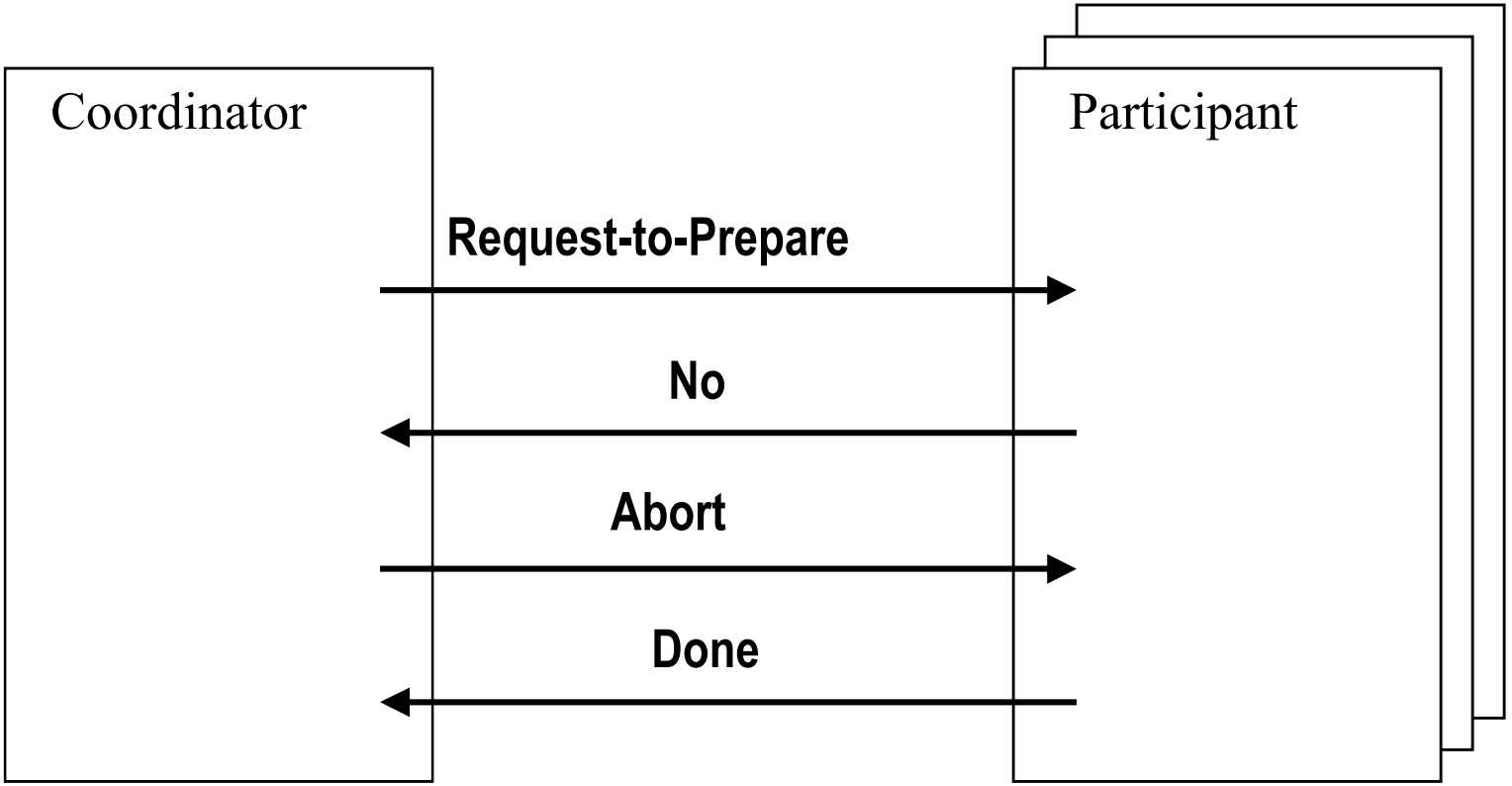
5 The coordinator sends its decision to all participants (i.e. Commit or Abort)

6 Participants acknowledge receipt of Commit or Abort by replying Done.

Case 1: Commit



Case 2: Abort



Two-phase commit: Coordinator

1. Sends CanCommit, receives yes, abort;
2. Sends DoCommit, DoAbort

   If *yes* messages are received from all workers, the coordinator sends a **DoCommit message** to all workers, which then implement the Commit. However, if any of the workers replies with **abort** instead of *yes*, the coordinator sends a **DoAbort message** to all workers to trigger a collective Abort.

Two-phase commit: Worker

1. Receives CanCommit, sends yes, abort;
2. Receives DoCommit, DoAbort

A worker on receiving a **CanCommit message** answers with either a yes or abort message, depending on the workers internal state. Afterward, it commits or aborts depending on the instructions from the coordinator.

The Protocol: Failure

- server failures/hosts:
  - If a host fails in the 2PC protocol, then, after being restarted, it aborts all transactions
  - restarting worker aborts all transactions
- Failure of communication channels/network:
  - coordinator aborts after timeout.

Two-phase commit with timeouts: Worker

- On *timeout* sends GetDecision
- Whenever a worker that is ready to commit when receiving a CanCommit message times out while waiting for the decision from the coordinator, it sends a special GetDecision message that triggers the coordinator to resend the decision on whether to Commit or Abort.
- These messages are handled by the coordinator once the decision between committing and aborting has been made.
- Coordinator resends CanCommit messages if a worker does not issue a timely reply.

Two-phase commit with timeouts: Coordinator

- On timeout re-sends CanCommit,
- On GetDecision repeats decision

Limitations(2PC)

- Once node voted “yes”, cannot change its mind, even if subsequently crashes.
- If coordinator crashes, all workers may be blocked

Distributed Nested Transactions

- Each sub-transaction starts after its parent and finishes before it.
- When a sub-transaction completes, it makes an independent decision about commit provisionally or abort.
- Difference between provisional commit and prepared to commit :

  Provisional commit: it's not saved on permanent storage; It only means it has finished correctly and will agree to commit when it is asked to.

  Prepared commit: guarantees a sub-transaction will be able to commit
- If the parent aborts, it aborts all transactions on the provisional commit list. Otherwise, if the parent is ready to commit, it lets all sub-transactions commit.
- After all sub-transactions are completed, the provisionally committed sub-transactions participate in a two-phase commit protocol.
- When a worker receives a CanCommit message, there are two alternatives:

  1. If it has no recollection of the sub-transactions involved in the committing transaction, it votes abort, as it must have recently crashed.

  2. Otherwise, it saves the information about the provisionally committed sub-transaction to a persistent store and votes yes.