Module 6 - Synchronization

Synchronization Problem

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

What will we study

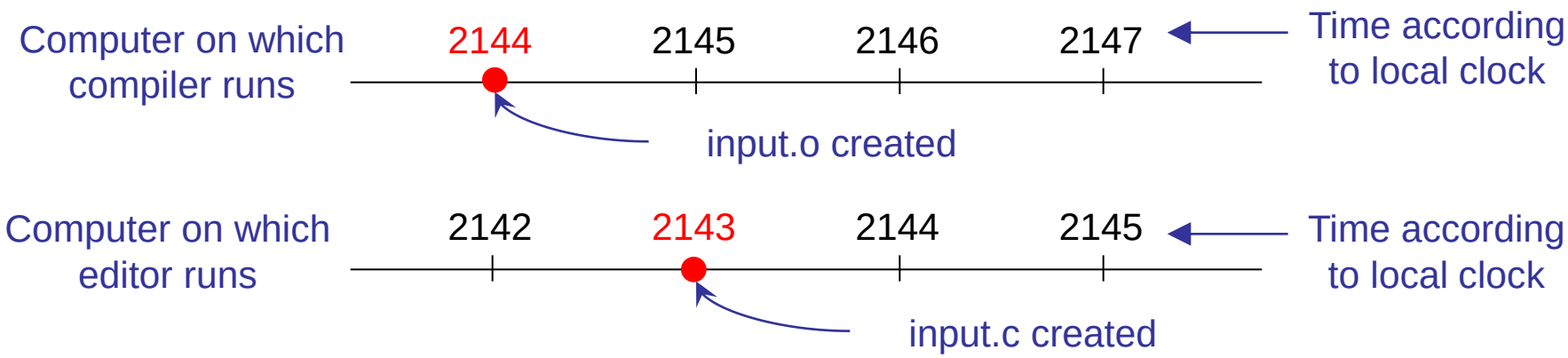
- Part 1 - Clocks
  - How to synchronize events based on actual time?
    - → Clock synchronization
  - How to determine relative ordering?
    - → Logical clocks
- Part 2 - Global State and Election
  - What is the “global state” of a distributed system?
  - How do we determine the “coordinator” of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

Clock synchronization

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

Clock synchronization (2)

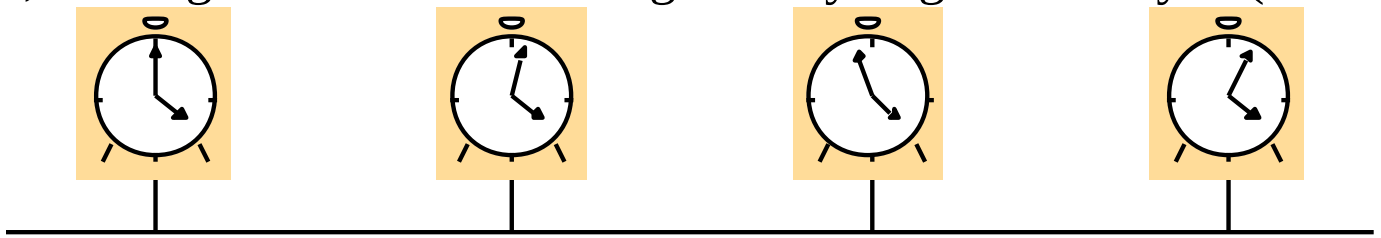
- In a distributed system:
  - Achieving agreement on time is not trivial!



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

Clocks

- A computer has a timer, not really a clock.
- Timer is a precisely machined quartz crystal oscillating at a frequency that depends on how the crystal was cut and the amount of tension
- Two registers are associated with the crystal: counter & holding register
  - Each oscillation of the crystal decrements the counter by one. When counter gets to zero, an interrupt is generated and the counter reloaded from holding register.
  - In this way, it is possible to program a timer to generate an interrupt 60 times a second, or any other desired frequency. Each interrupt is called a clock tick
  - At each clock tick, the interrupt procedure adds 1 to the time stored in memory (this keeps the software clock up to date)
- On each of *n* computers, the crystals will run at slightly different frequencies, causing the software clocks gradually to get out of sync (clock skew).



Logical vs Physical Clocks

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

Physical Clocks

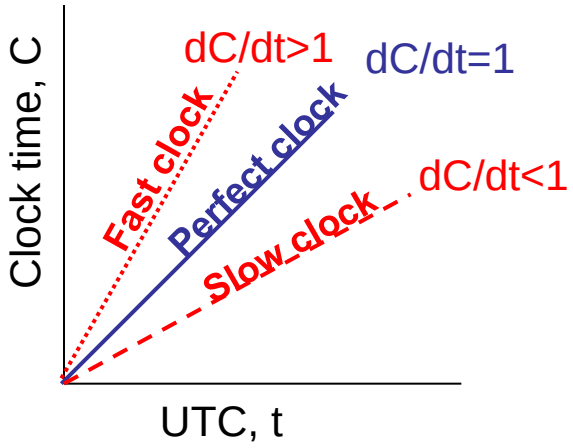
- Since the invention of mechanical clocks in the 17th century, time has been measured astronomically (mean solar day, mean solar second)
- With the invention of the atomic clock in 1948, it becomes possible to measure the time more accurately
  - Labs around the world have atomic clocks and each of them periodically tells the BIH (Bureau International de l'Heure) in Paris how many times its clock ticked.
  - The BIH averages these to produce the TAI (Temps Atomique International).
  - Originally the atomic time is computed to make the atomic second equal to the mean solar second

Universal Coordinated Time

- Because the mean solar day gets longer all the time, the BIH made necessary correction (by introducing leap seconds) called UTC (Universal Coordinated Time)
- UTC is provided to people who need precise time:
  - National Institute of Standard Time (NIST) operates a shortwave radio station with call letters WWV from Fort Collins, Colorado: ± 1 msec accurate.
  - From an earth station using GEOS: accurate to 0.5 msec
  - By telephone from NIST: cheaper but less accurate.

Physical Clock Synchronization

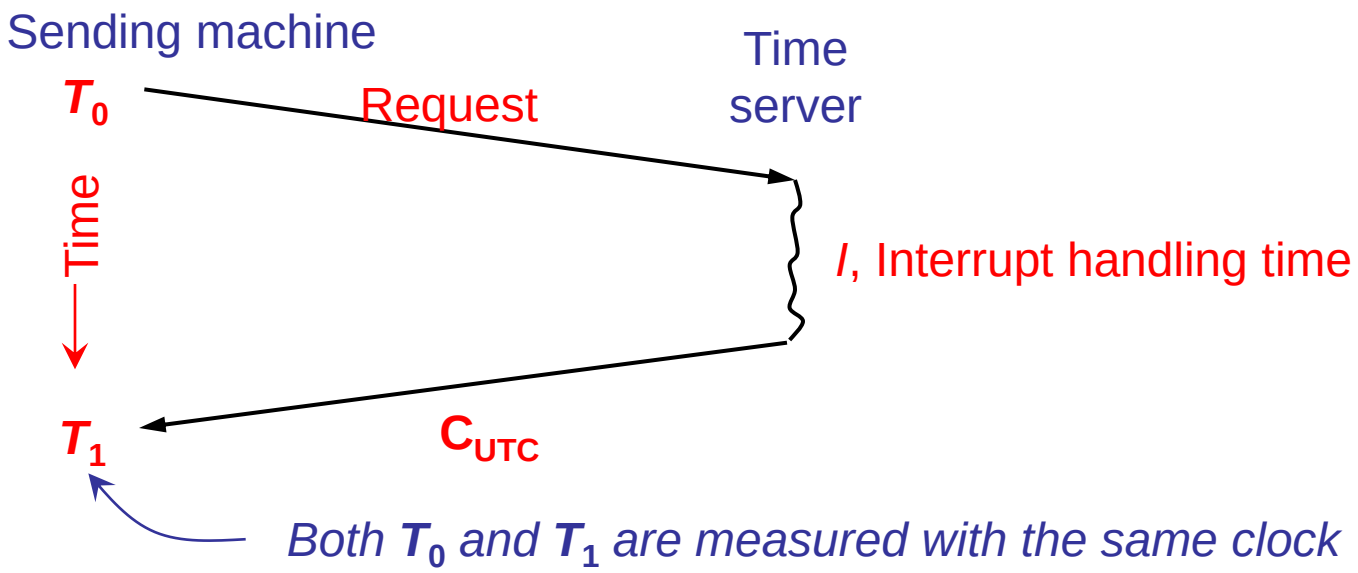
- Model of the system
  - Each machine timer causes an interrupt *H* times a second (theoretically)
  - When timer goes off, the interrupt handler adds 1 to the software clock
  - When UTC is *t*, the value of the clock on machine *p* is $C_p(t)$
  - In a perfect world, $C_p(t) = t$ for all *p* and all *t*, i.e., $dC/dt = 1$
  - In practice, timers do not interrupt exactly *H* times a second. The relative error obtainable with modern chips is 10^{-5}
  - Timer is said to be working within its specification if:
  - $1-\rho \le dC/dt \le 1+\rho$, where constant ρ is the maximum drift rate (specified by manufacturer)



- If after synchronization, 2 clocks are drifting from UTC in the opposite direction, they may be as much as $2\rho\Delta t$ apart at time Δt
- For clocks not to differ by more than δ, they must be resynchronized every $\delta / 2\rho$ seconds

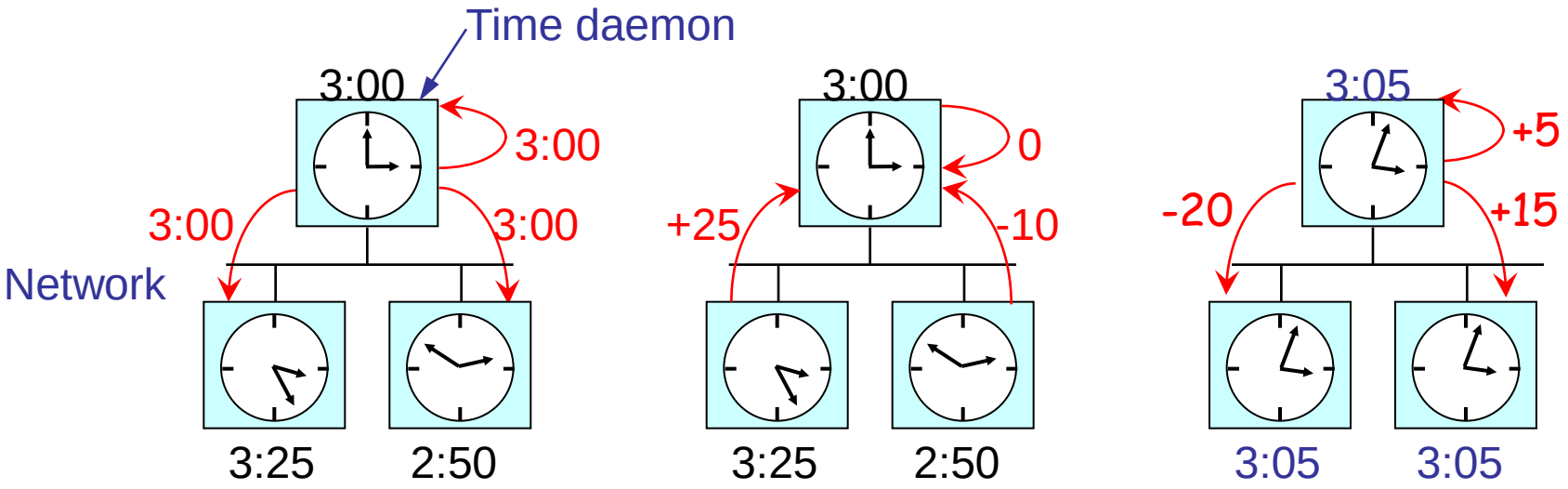
Cristian's Algorithm

- One time server (WWV receiver); all other machines stay synchronized with the time server.
- Cannot set T_1 to C_{UTC} because time must never run backwards. Changes are introduced gradually by adding more or less seconds for each interrupt.
- The propagation time is included in the change.
  - Estimated as: $(T_1 - T_0 - I)/2$ (can be improved by continuous probing & averaging)



Berkeley Algorithm

- Suitable when no machine has a WWV receiver
- The time server (a time daemon) is active:
  - Time daemon polls every machine periodically to ask what time is there
  - Based on the answers, it computes an average time
  - Tells all other machines to advance their clocks to the new time or slow their clocks down until some specified reduction has been achieved
- The time daemon's time is set manually by operator periodically

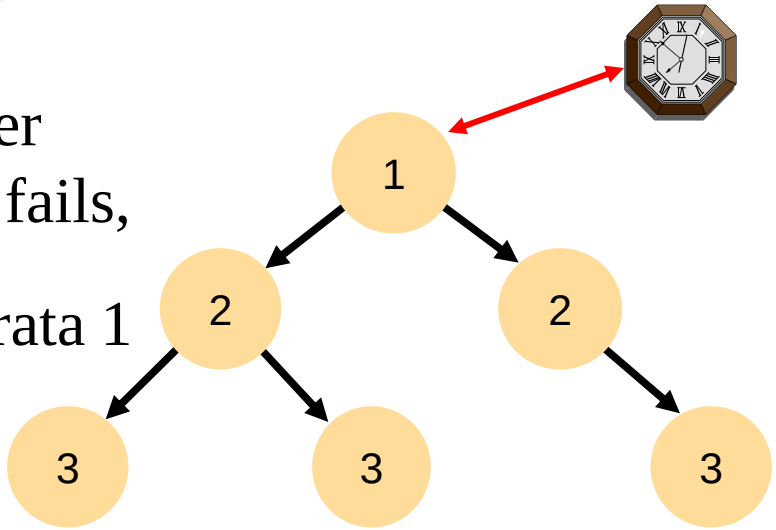


Averaging Algorithm

- A decentralized algorithm:
  - Divide the time into fixed length resynchronization intervals
  - The i-th interval starts at T_0+iR and runs until $T_0+(i+1)R$, where T_0 is an agreed upon moment in the past, and R is a system parameter
  - At the beginning of each interval, each machine broadcasts the current time according to its own clock (broadcasts are not likely to happen simultaneously because the clocks on different machine do not run at exactly the same speed)
  - After a machine broadcasts its time, it starts a timer to collect all other broadcasts that arrive during some interval S
  - When all broadcasts arrive, an algorithm is run to compute the new time from them
    - → Simplest algorithm: Average the values from all other machines
  - Variations: discard m highest and m lowest values and average the rest; Add to each message an estimate of the propagation time from source

The Internet Network Time Protocols (NTP)

- Layered client-server architecture, based on UDP message passing
- Synchronization at clients with higher strata number less accurate due to increased latency to strata 1 time server
- Failure robustness: if a strata 1 server fails, it may become a strata 2 server that is being synchronized though another strata 1 server
- Modes:
  - Multicast
    - → One computer periodically transmits time
  - Procedure call
    - → Similar to Cristian's algorithm
  - Symmetric
    - → To be used where high accuracy is needed



Use of Synchronized Clocks

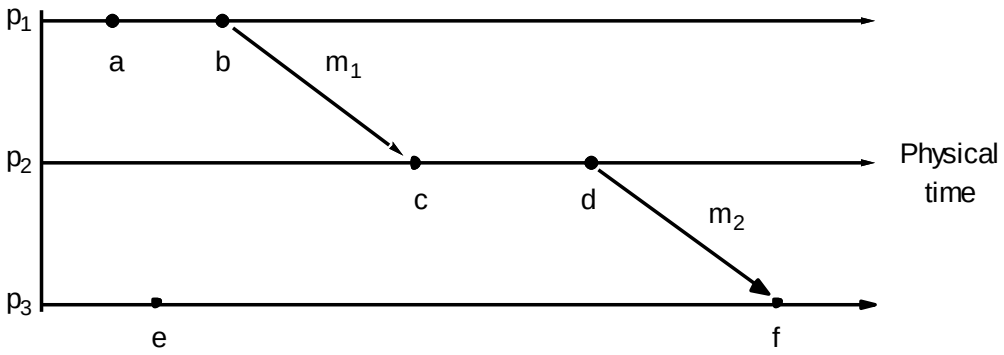
- Hardware and Software for synchronizing clocks on a wide scale are available (e.g., over the entire Internet)
- It is possible to keep millions of clocks synchronized to within a few milliseconds of UTC
- Algorithms that utilize synchronized clocks appeared
- Two examples (Liskov, 1993)
  - At-Most-Once Message Delivery
  - Clock-Based Cache Consistency

At-Most-Once Message Delivery

- Enforce at-most-once message delivery to a server, even in the face of crashes
- Algorithm:
  - Every message carries a connection identifier and a timestamp
  - For each connection, the server records in a table the most recent timestamp it has seen
  - If an incoming message for a connection is lower than timestamp stored for that connection, the message is rejected as a duplicate
  - Periodically the current time is written to disk.
  - When server crashes and then reboots, it reloads the stored time value
    - → Any message with a timestamp older than this time value is rejected as duplicate
    - → Consequence:
      - every message that might have been accepted before the crash is rejected.
      - Some new messages may be incorrectly rejected, but at-most-once semantics is always guaranteed.

Logical Clock Synchronization

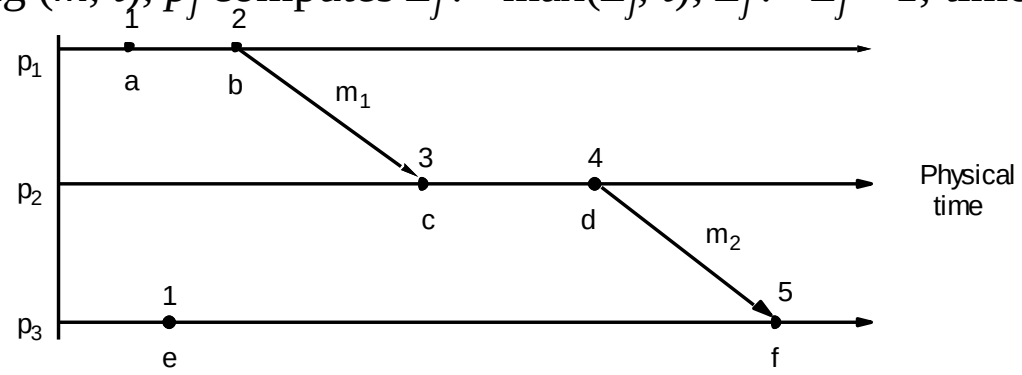
- Happens-before relation
  - If a and b are events in the same process, and a happens-before b, then $a \rightarrow b$ is true
  - If a is the event of a message being sent by one process, and b is the event of the same message being received by another process, then $a \rightarrow b$ is also true
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



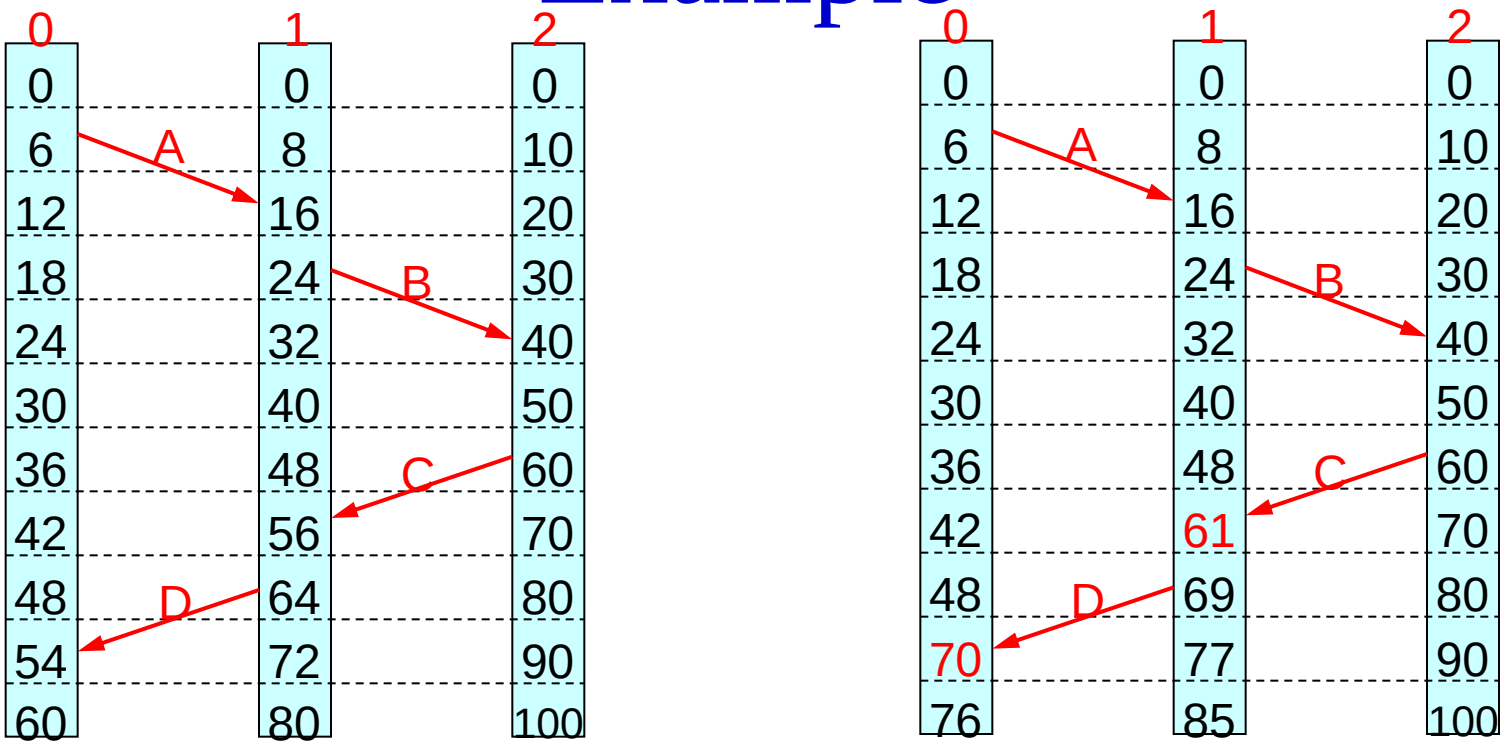
From Coulouris, Dollimore and Kindberg, Distributed Systems: Concepts and Design, 3rd ed.

Lamport's Algorithm

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



Example

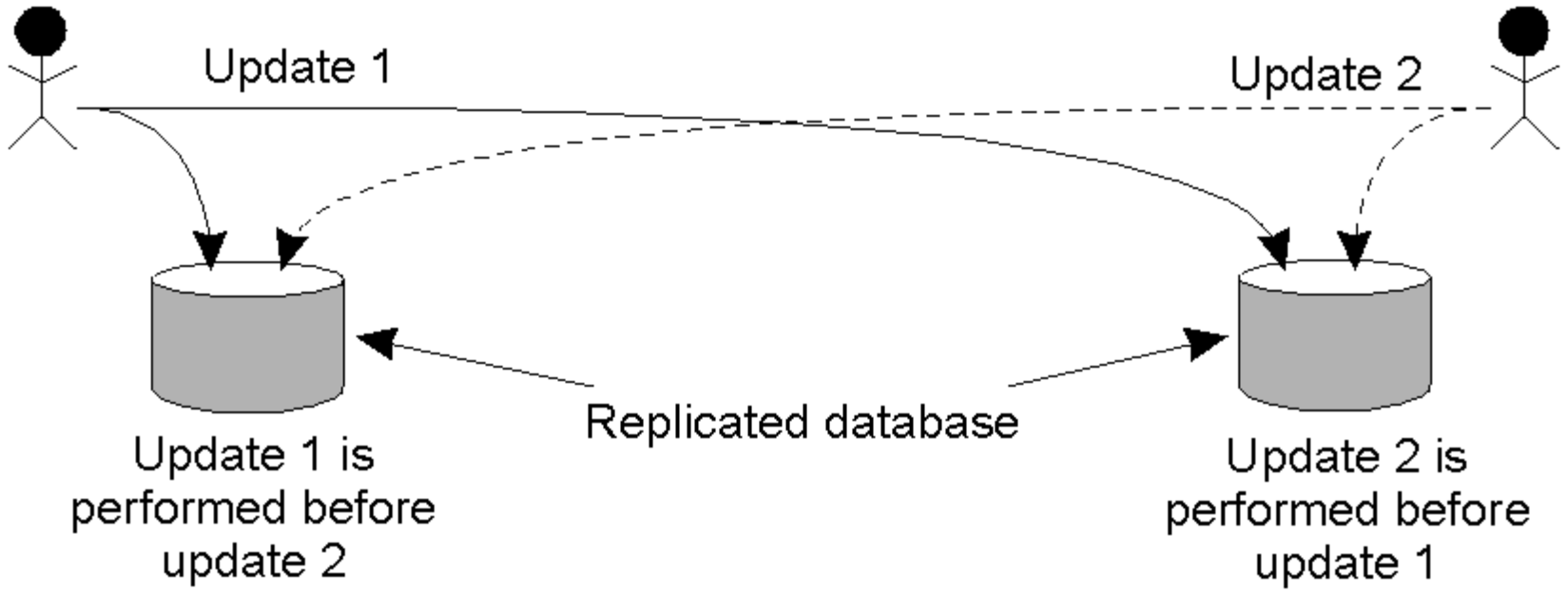


a) Three processes, each with its own clock. The clocks run at different rates.

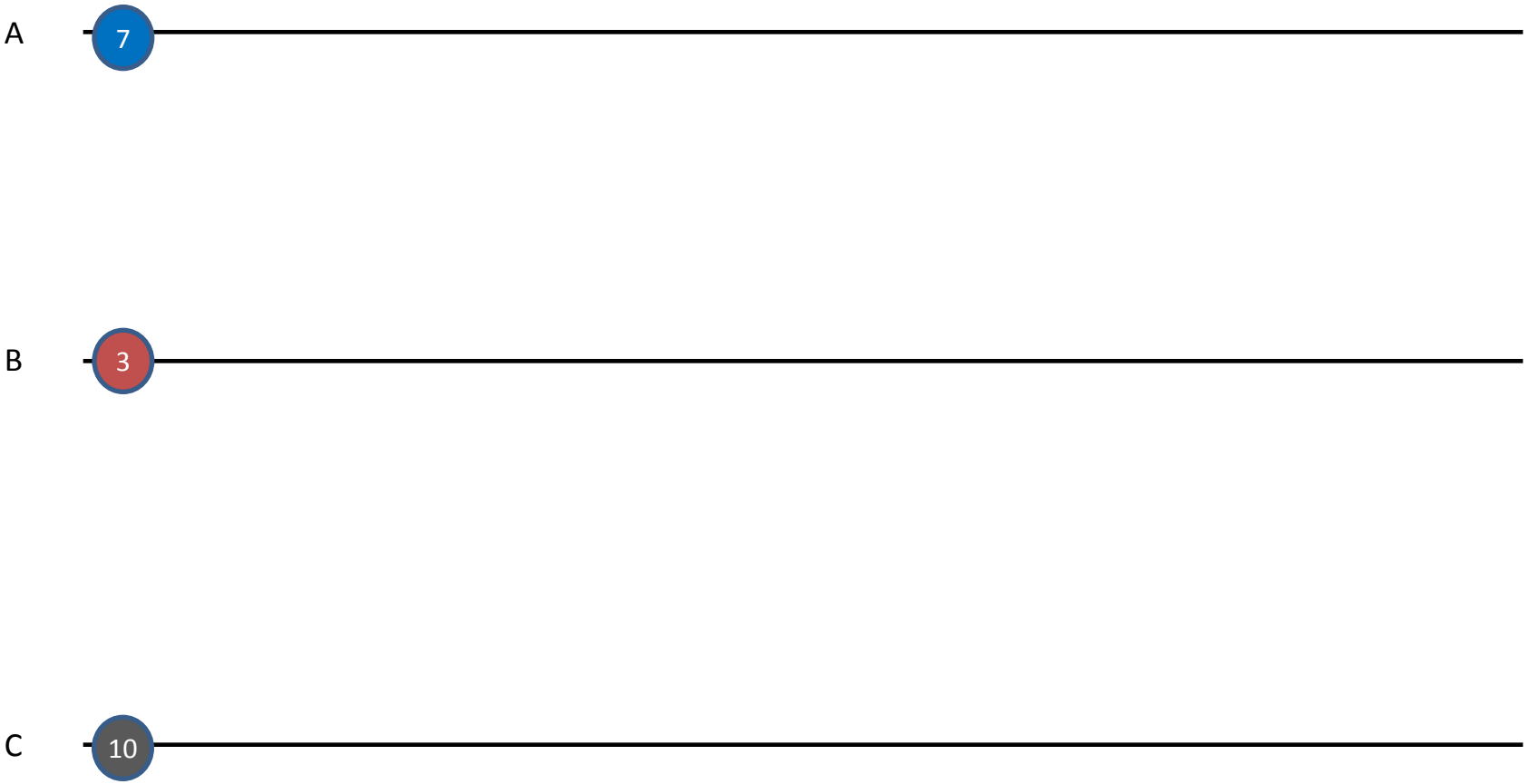
b) Lamport's algorithm corrects the clocks.

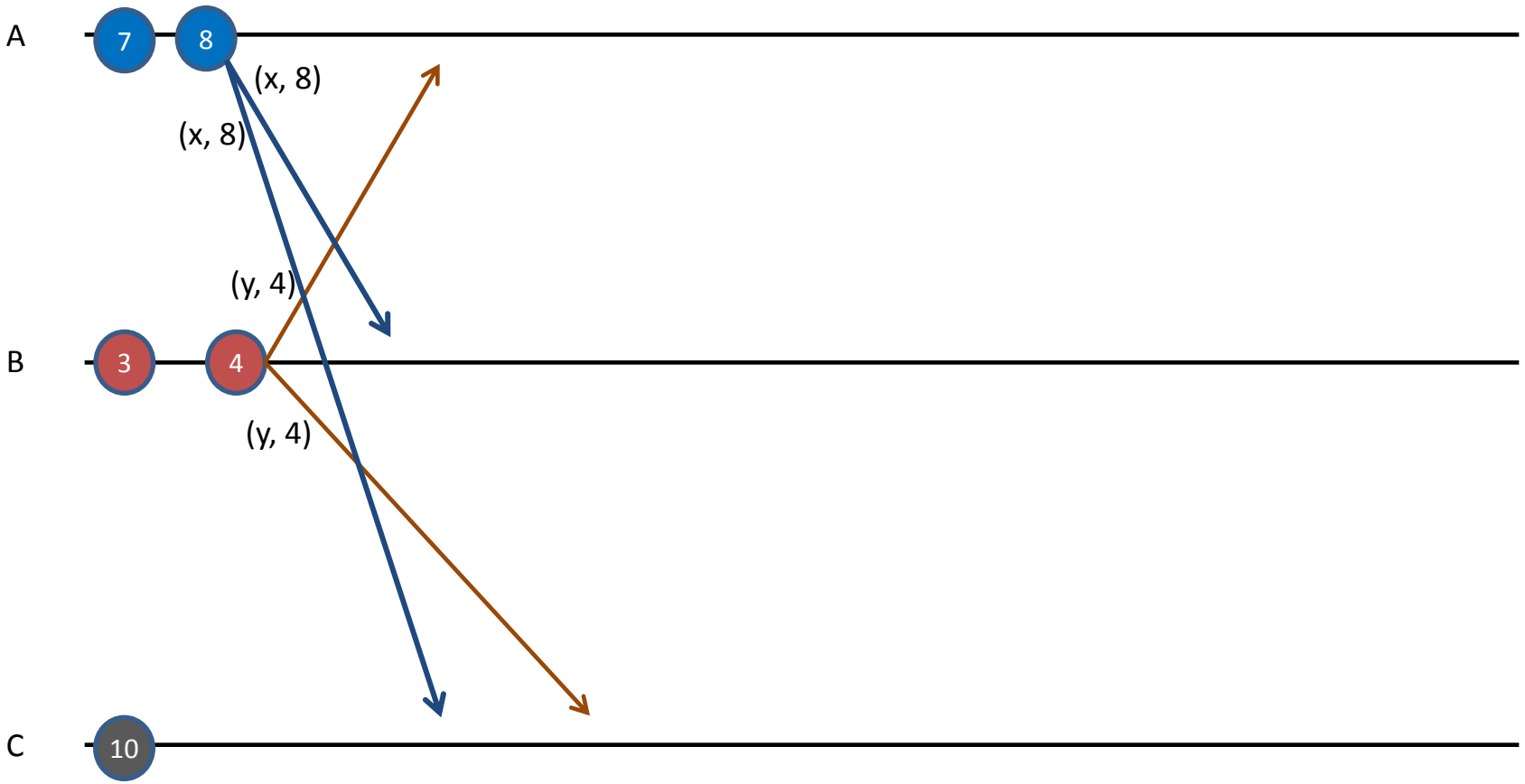
- Lamport solution:
  - Between every two events, the clock must tick at least once
  - No two events occur at exactly the same time. If two events happen in processes 1 and 2, both with time 40, the former becomes 40.1 and the latter becomes 40.2

Lamport Timestamps



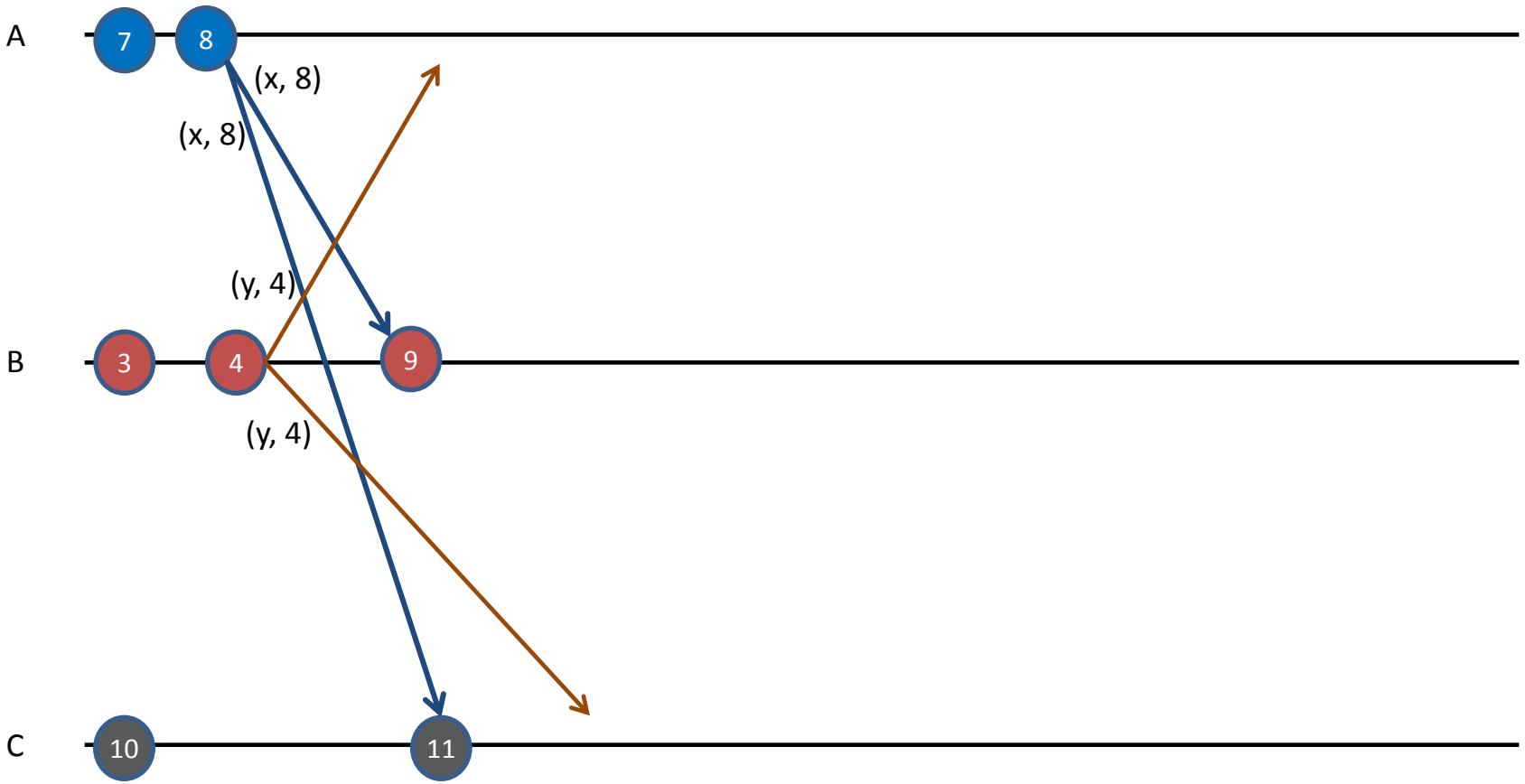
a) The updates have to be ordered the same way across the replicas.
b) This can be achieved by a totally ordered multicast algorithm.
c) Totally ordered multicasting can be achieved by means of Lamport clocks.

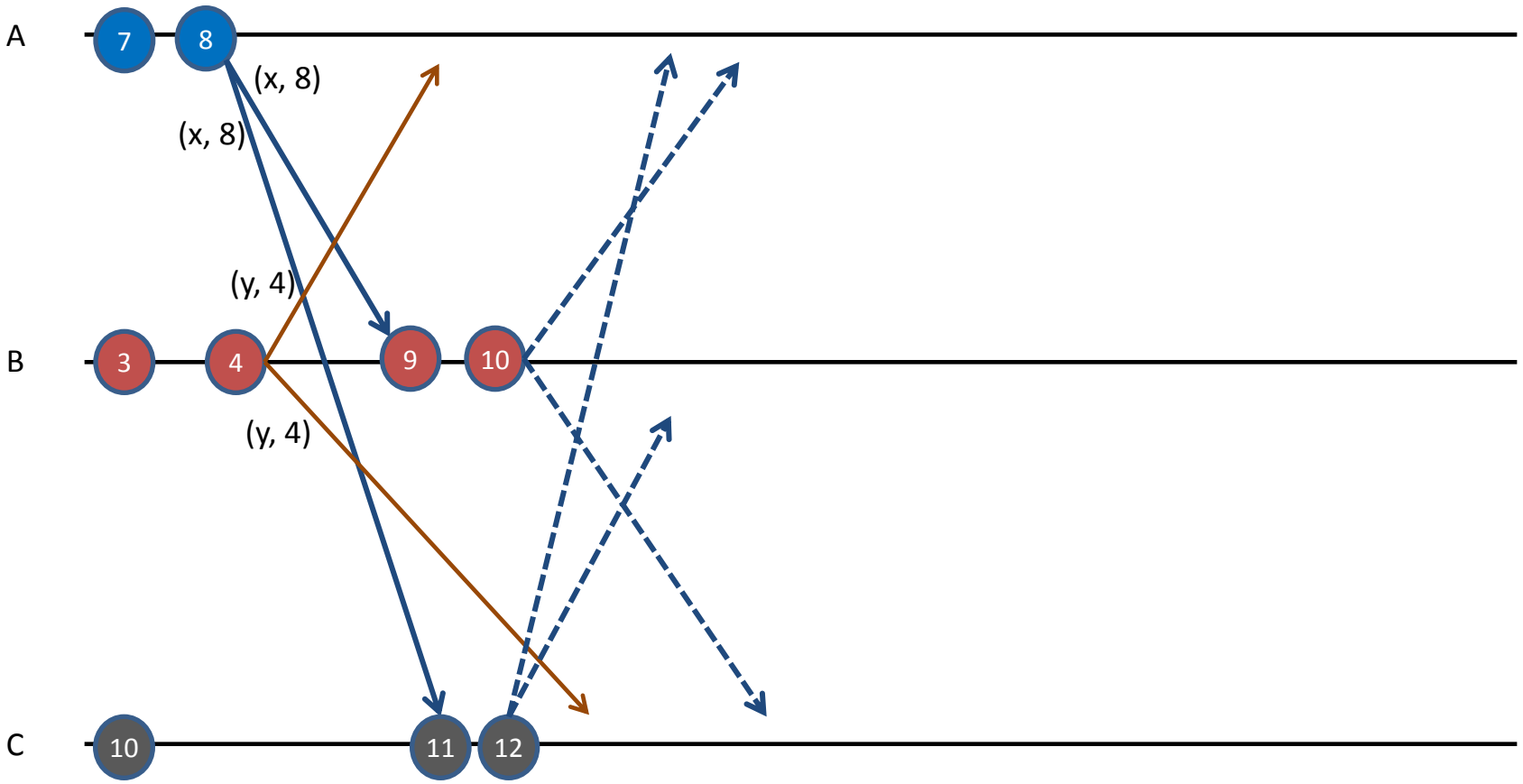

A
7
B
3
C
10

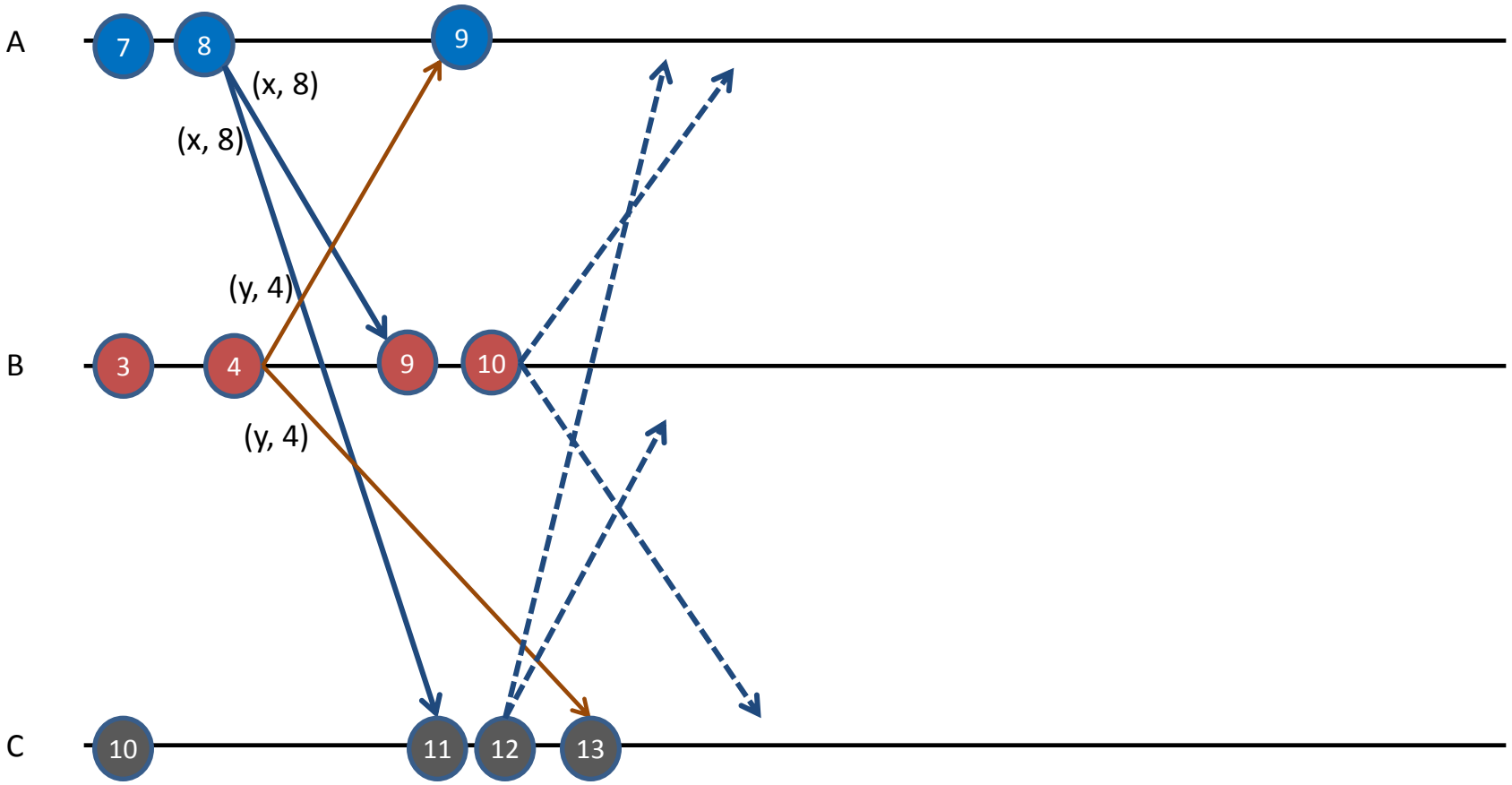


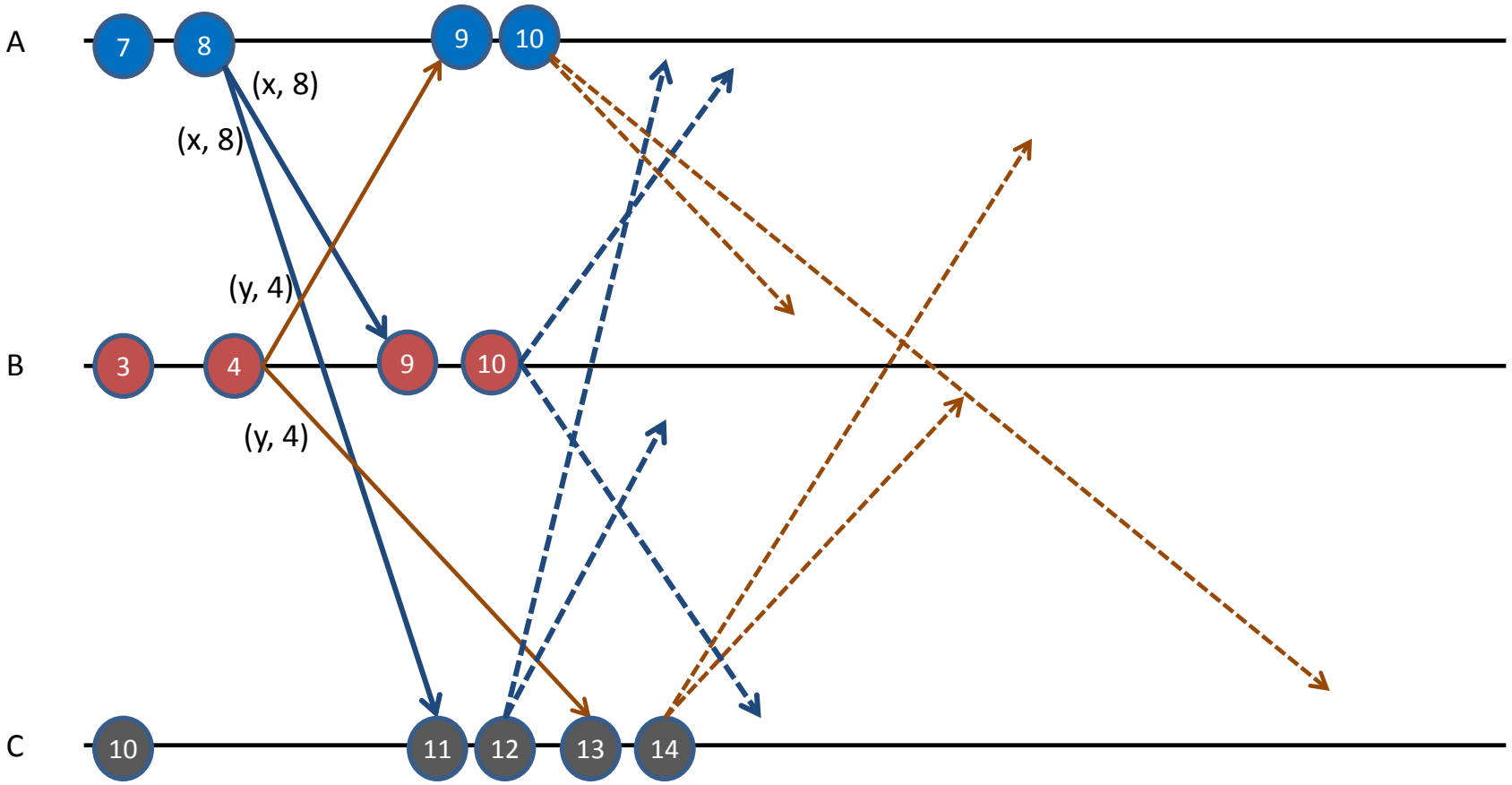
2 criteria before a node can commit a write:

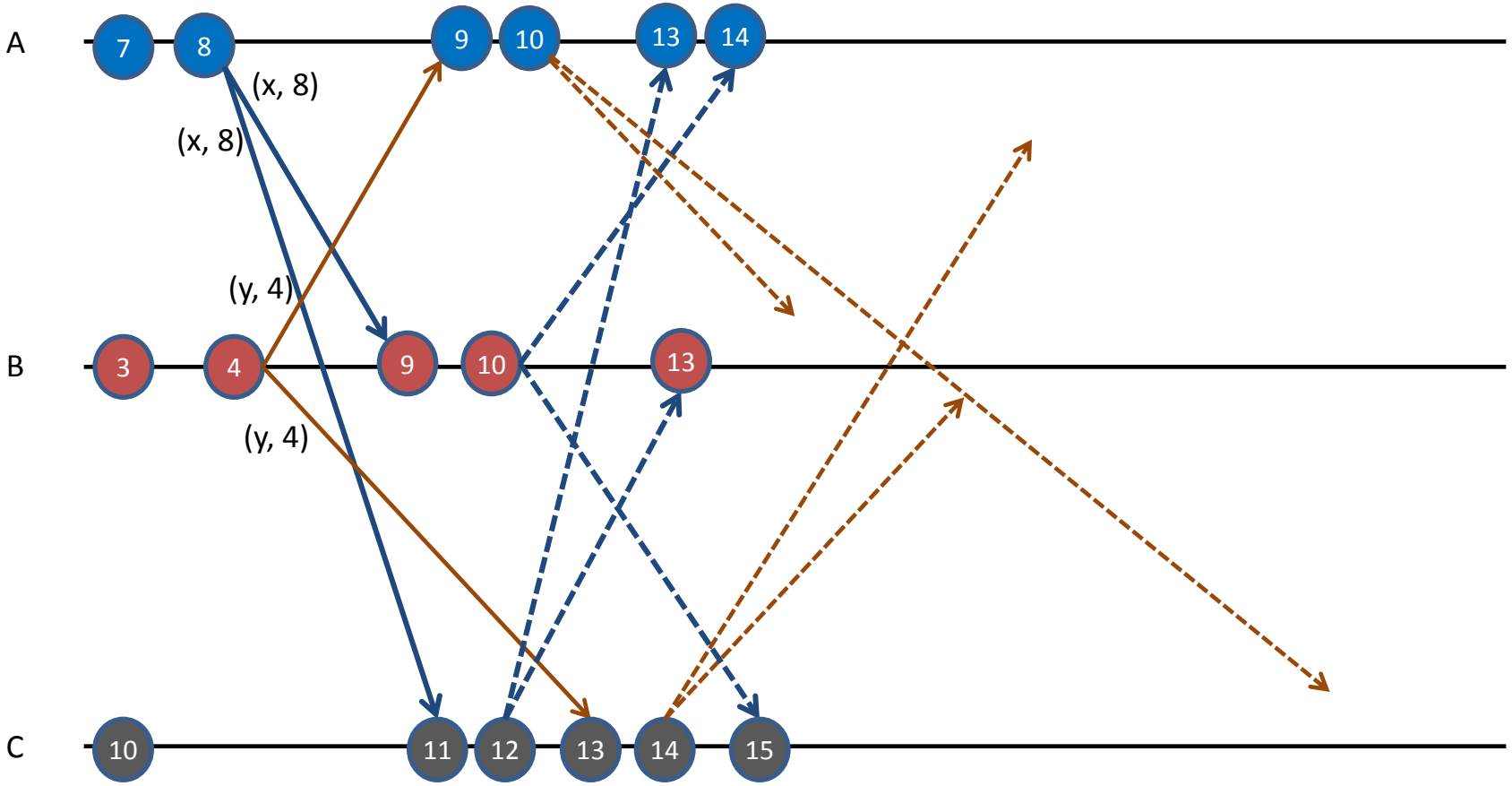
- All write requests with a smaller timestamp has been committed.
- It has received a write-request acknowledgement from every node.


A
7
8
(x, 8)
(x, 8)
(y, 4)
B
3
4
9
(y, 4)
C
10
11

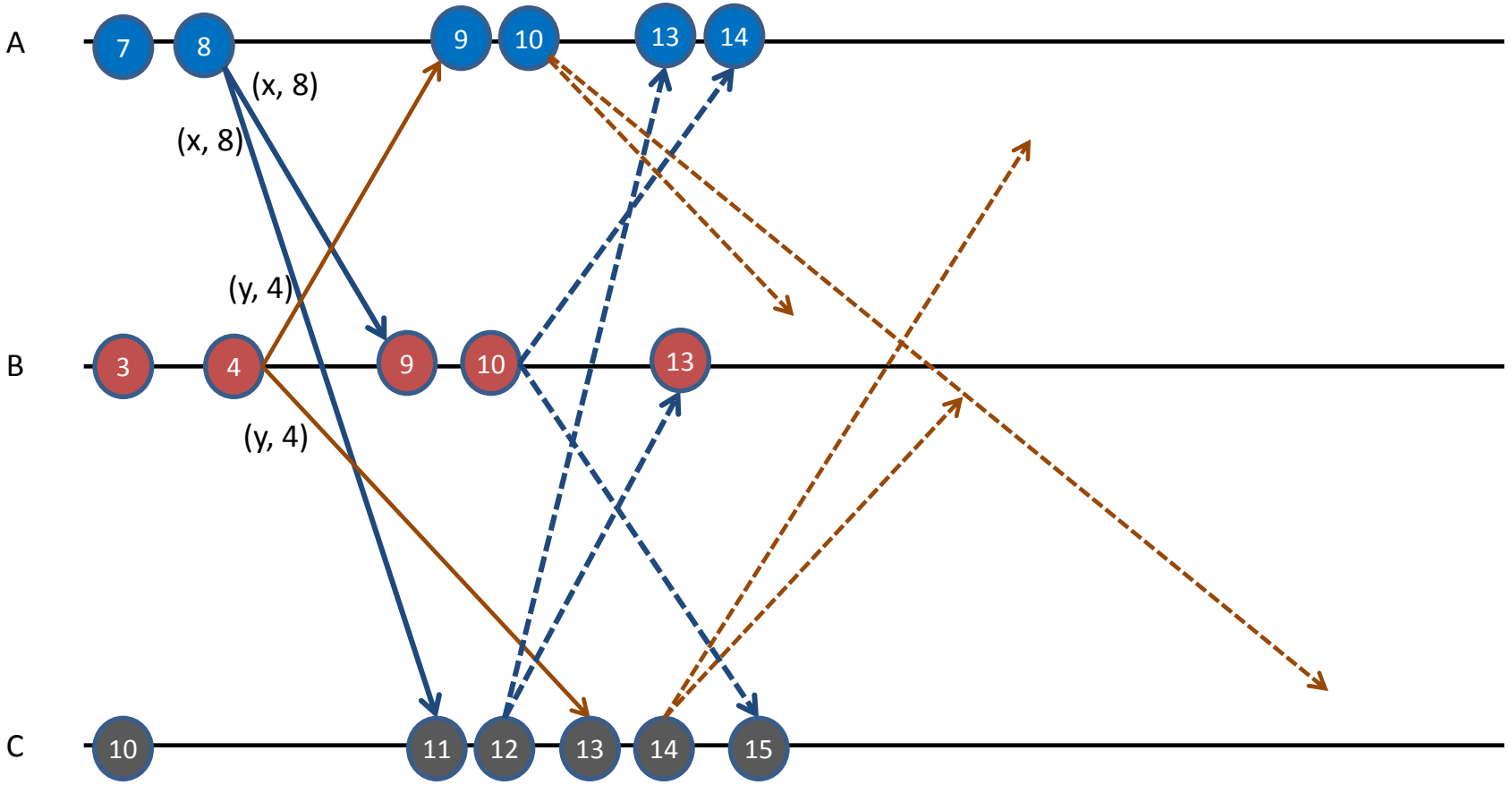

A
B
C
7
8
(x, 8)
(x, 8)
(y, 4)
3
4
9
10
(y, 4)
10
11
12


A
B
C
7
8
9
(x, 8)
(x, 8)
(y, 4)
3
4
9
10
(y, 4)
10
11
12
13


A
B
C
7
8
9
10
3
4
9
10
10
11
12
13
14
(x, 8)
(x, 8)
(y, 4)
(y, 4)

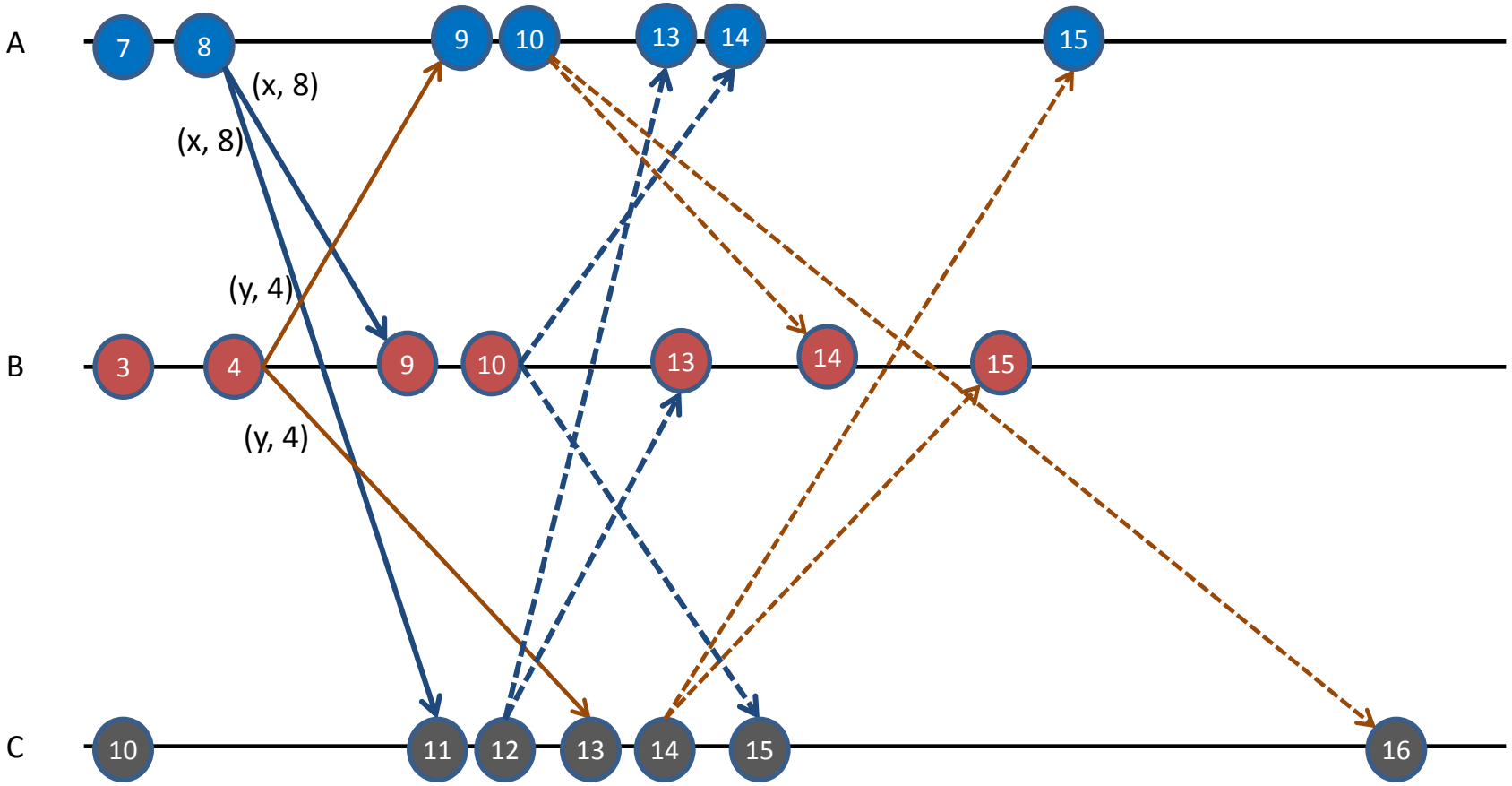

A
B
C
7
8
9
10
13
14
3
4
9
10
13
10
11
12
13
14
15
(x, 8)
(x, 8)
(y, 4)
(y, 4)

Can not commit (x, 8) until (y, 4) is committed as the nodes have already seen the (y, 4) message.



Node *B* will always send the (y, 4) message before sending an acknowledgement for (x, 8).

Can now commit (y, 4) and (x, 8) in that order.



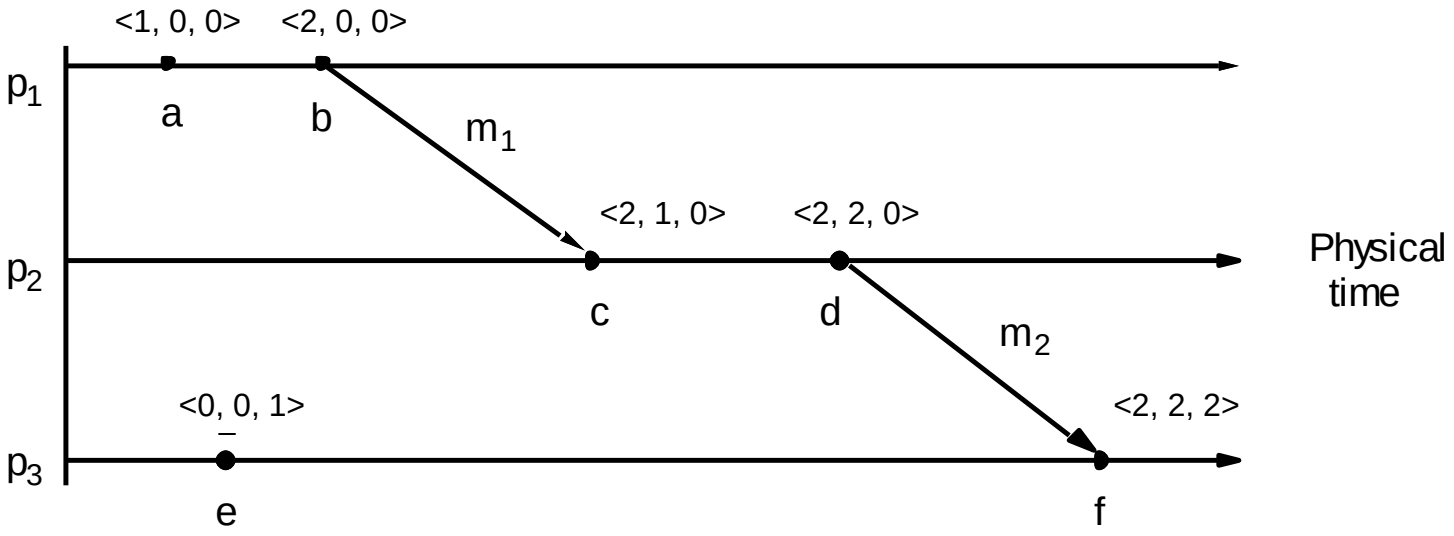
Vector Timestamp

- With Lamport timestamps:
  - e -> f then timestamp(e) < timestamp(f)
  - But timestamp(e) < timestamp(f) does not imply that e -> f
  - Does not capture causality

- Vector timestamp:
  - All hosts use a vector of logical clocks, where the *i*-th element is the clock value for the *i*-th host
  - Timestamp elements for other hosts are initially set to zero

Vector Timestamp

- For every non-communication event on host *i*
  - Increment the *i*-th element in the vector
- Send message sends the entire vector
- On receiving a message from host *j* (on host *i*), update vector *j* by:
  - Incrementing the *i*-th element
  - For the remaining elements, take the max value between vector *i* and *j*



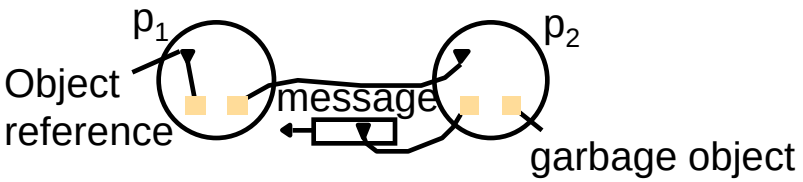
Vector Timestamp

- Two timestamps are equal iff all elements are equal
  - VT1[i] = VT2[i], i = 1, …, n
- VT1 <= VT2
  - iff VT1[i] <= VT2[i], i = 1,..., n
- VT1 < VT2
  - iff VT1 <= VT2 and VT1 != VT2
- Otherwise, VT1 is concurrent with VT2
  - We can't tell from by the message communication pattern whether one happened before the other.

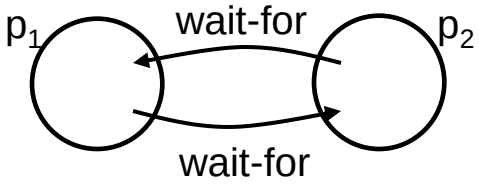
Part 2
Global State & Election Algorithms

Global State

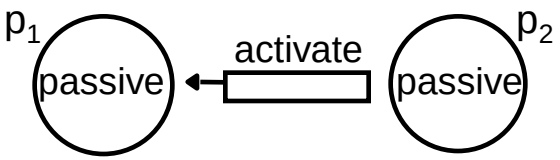
- Sometimes it is useful to know the global state of a distributed system
  - Garbage collection



  - Deadlocks



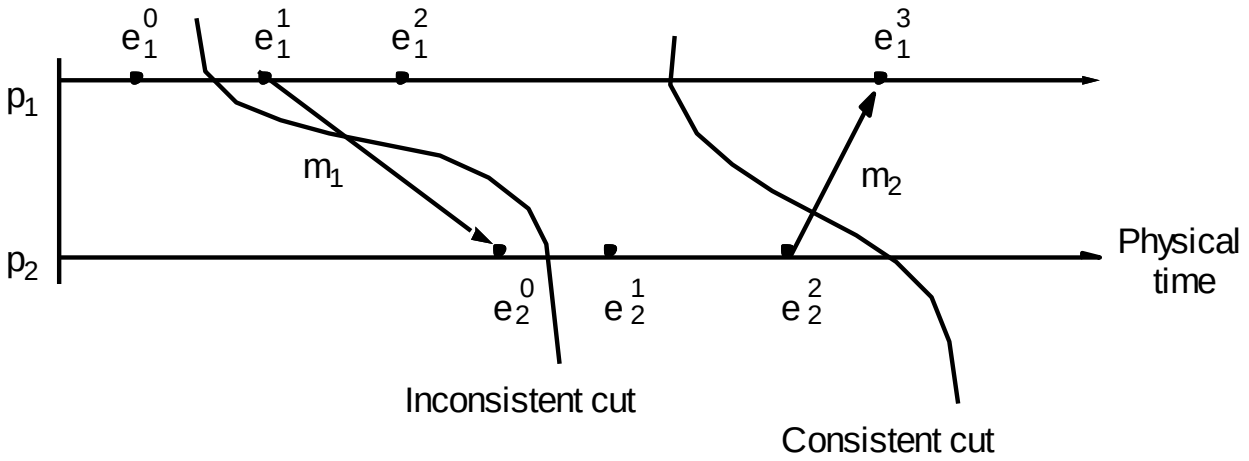
  - Termination



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

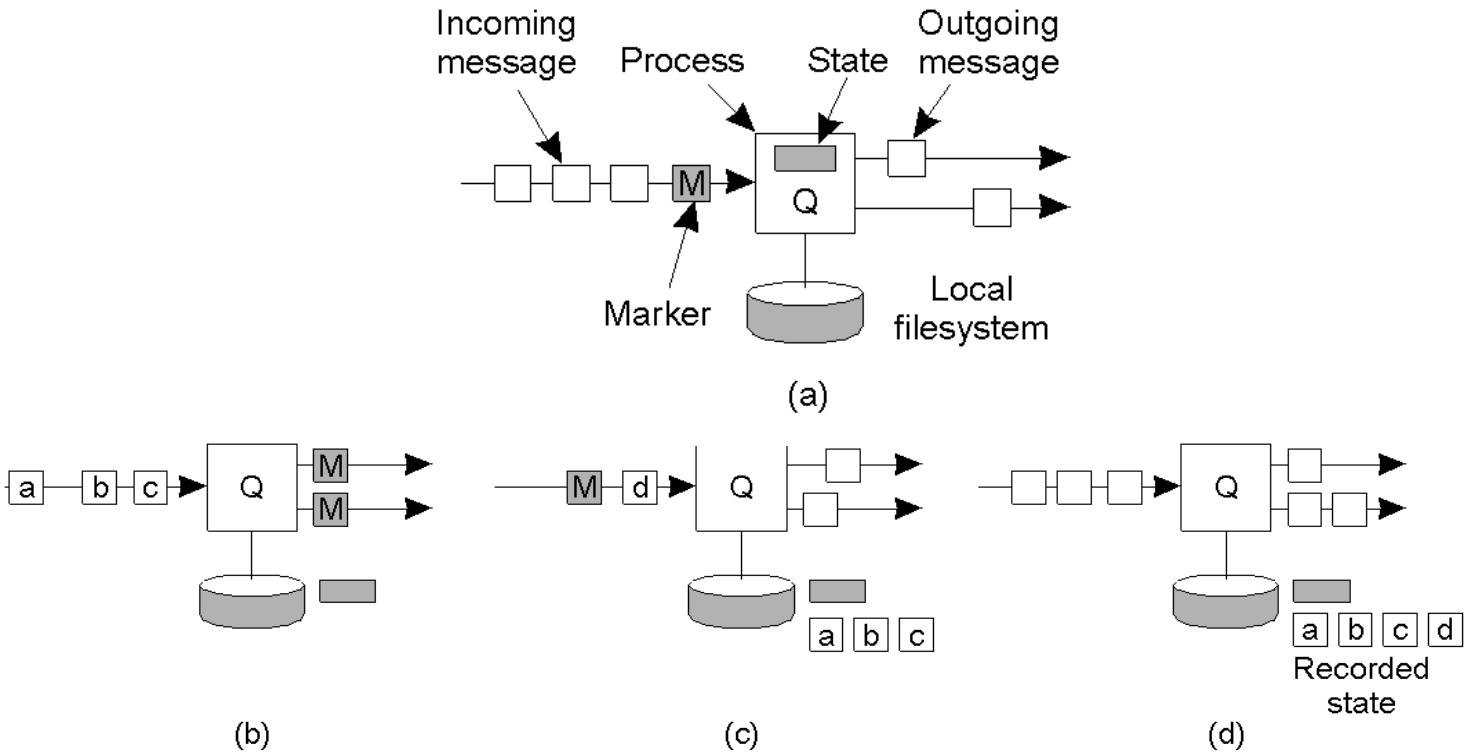
Distributed Snapshot

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

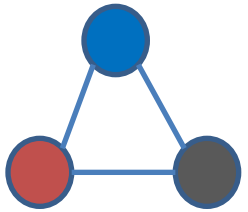


- Cut of the system's execution: subset of its global history identifying set of events *e*
  - Cut C is consistent iff $\forall e \in C, f \rightarrow e \Rightarrow f \in C$

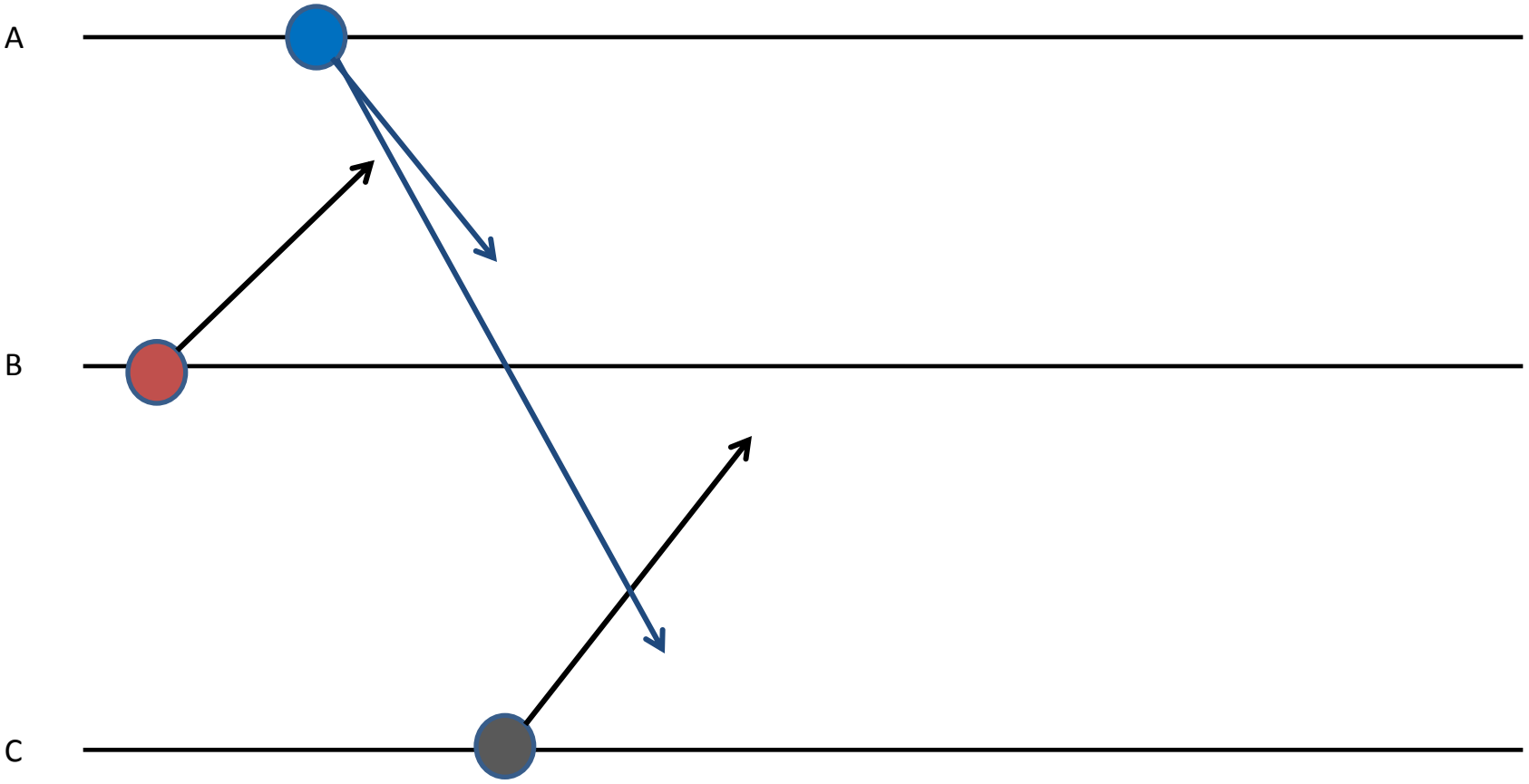
Distributed Snapshot Algorithm

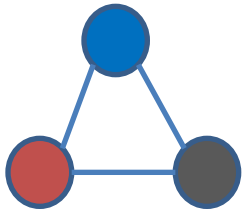


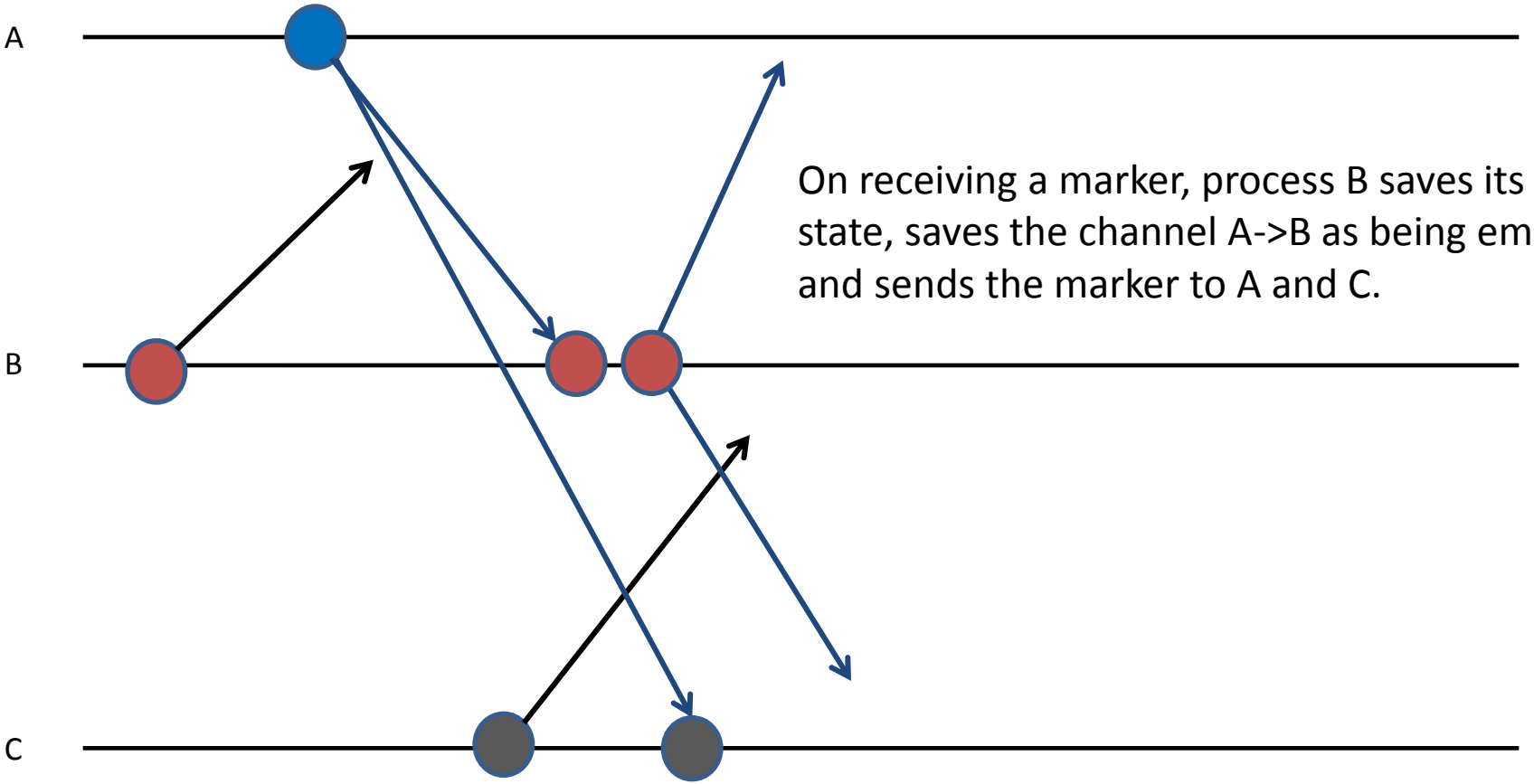
a) Organization of a process and channels for a distributed snapshot
b) Process Q receives a marker for the first time and records its local state
c) Q records all incoming message
d) Q receives a marker for its incoming channel and finishes recording the state of the incoming channel



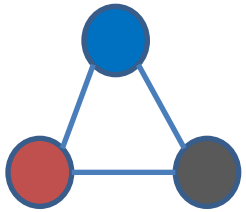
Initiates distributed snapshot. Process A saves it local state and sends a marker to B and C.

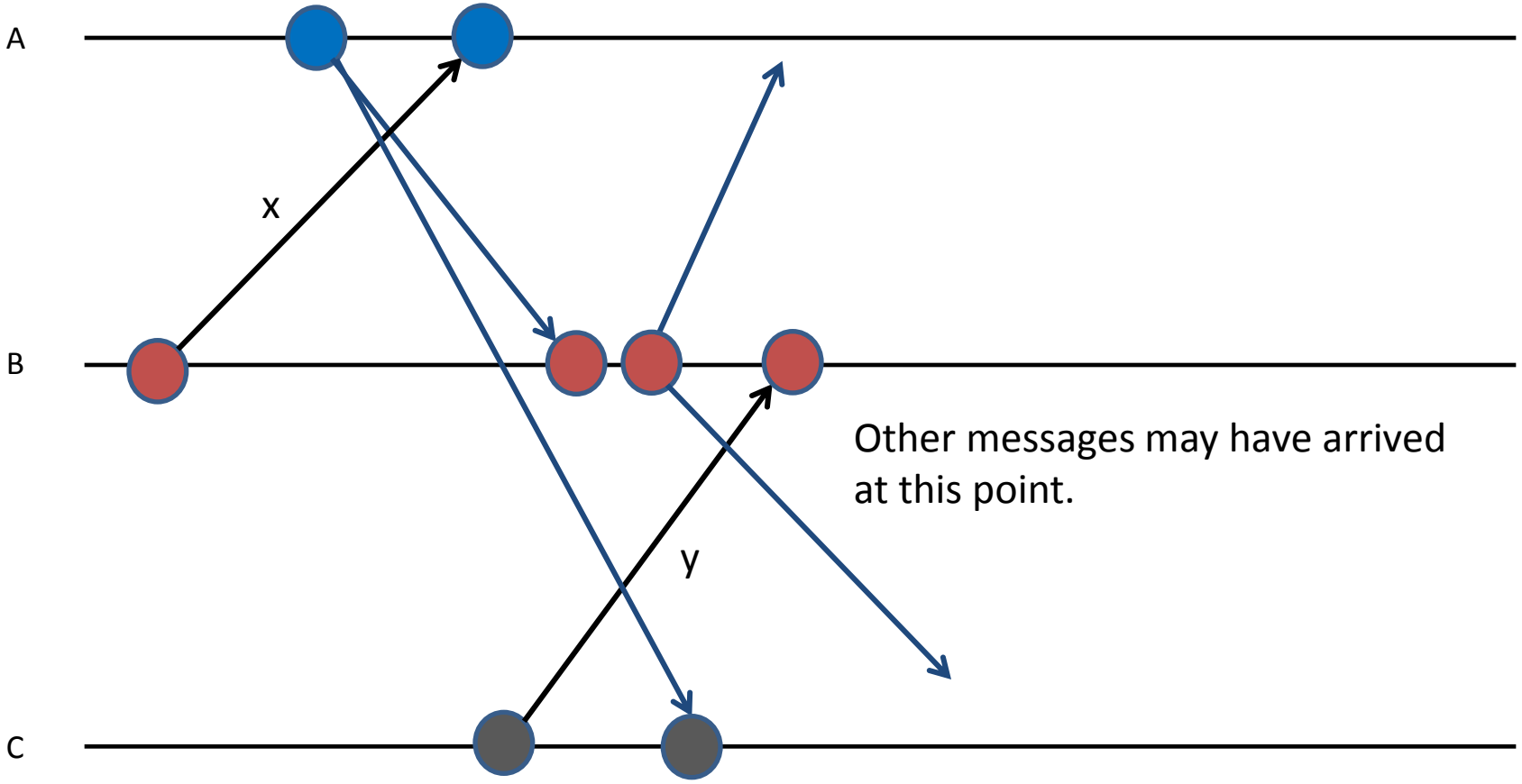




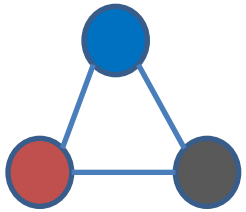


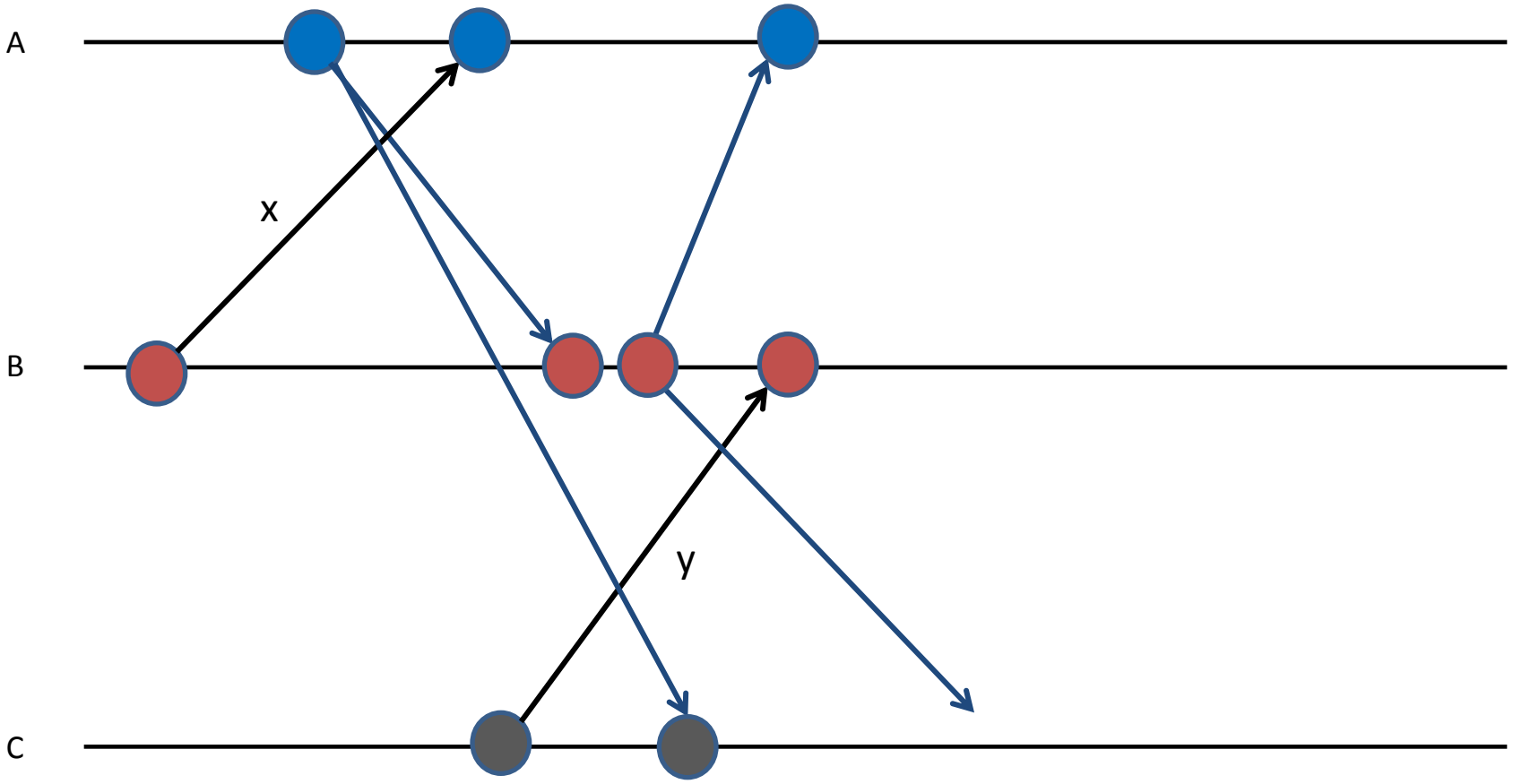
On receiving a marker, process B saves its local state, saves the channel A->B as being empty, and sends the marker to A and C.

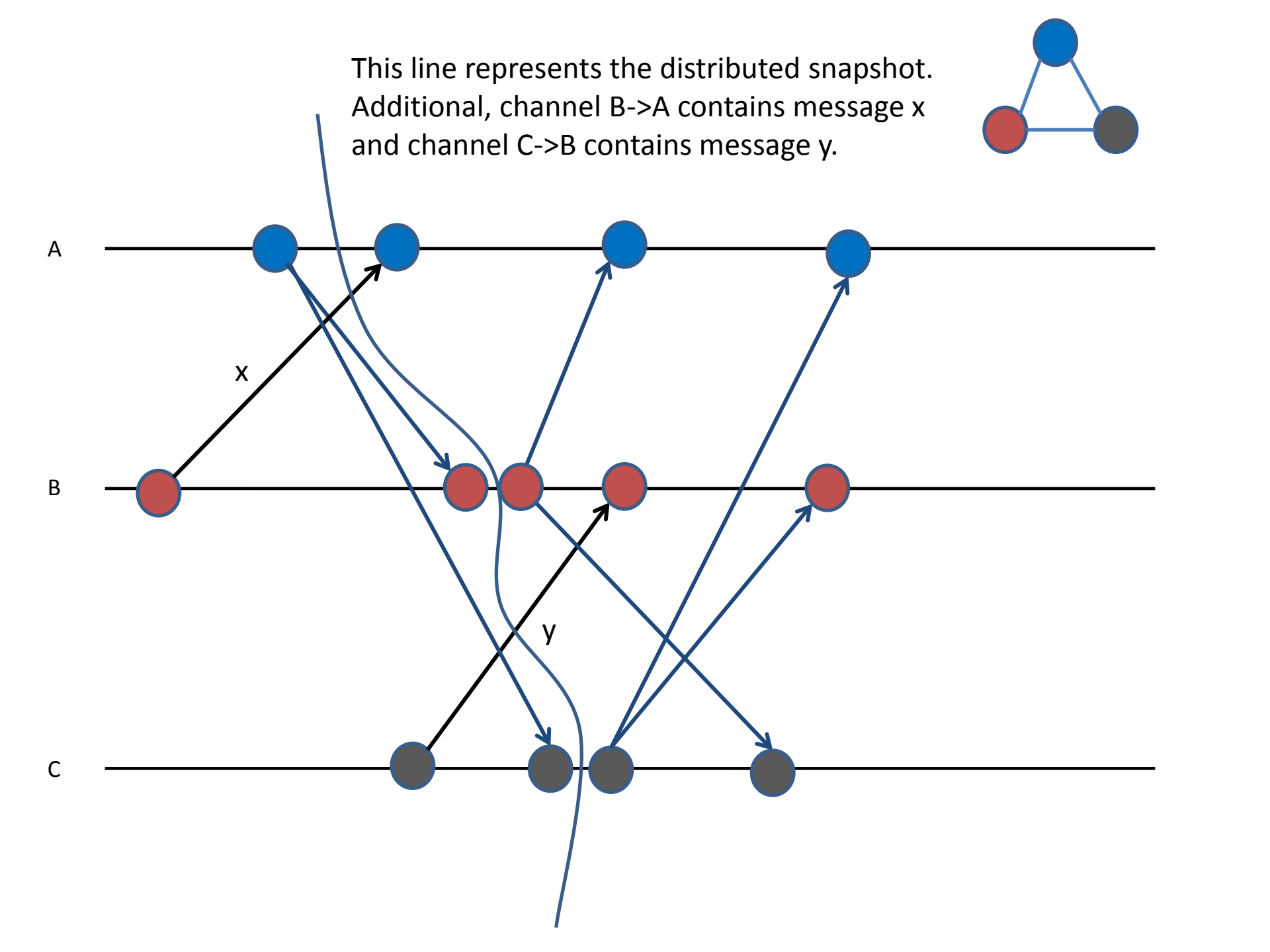



A
B
C
x
y
Other messages may have arrived
at this point.

When a node that has already recorded a local snapshot receives a marker on a channel, it records the state of the channel as having all messages that have arrived since the local snapshot. For example, channel B->A contains message x.






This line represents the distributed snapshot.
Additional, channel B->A contains message x
and channel C->B contains message y.
A
B
C
x
y

Election Problem

- Many algorithms require 1 process to act as a coordinator
  - → Example: Coordinator in the centralized mutual exclusion algorithm
- In general, it does not matter which process takes on this special responsibility, but one has to do it.
- How to elect a coordinator?

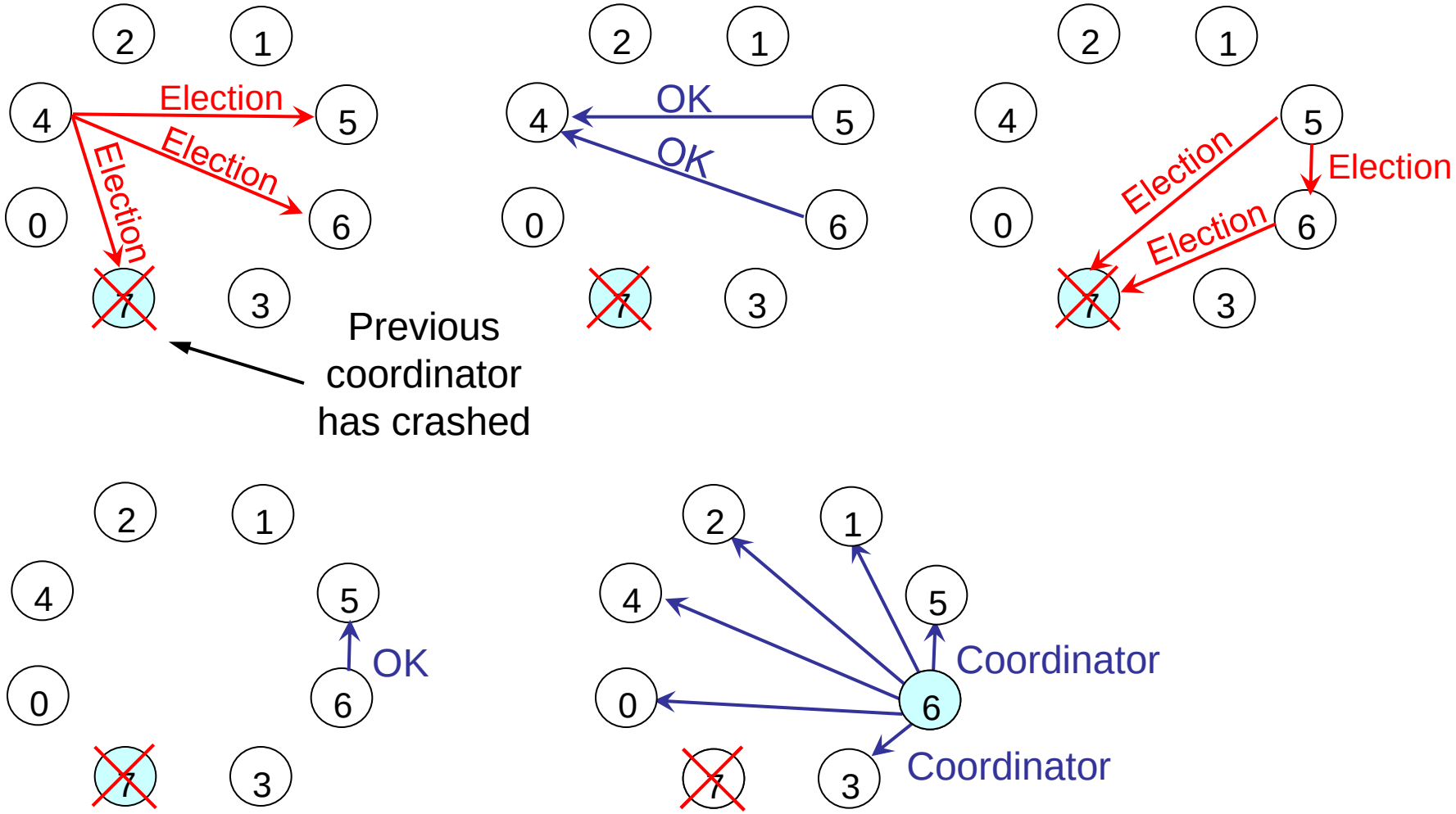
General Approach

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

The Bully Algorithm

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

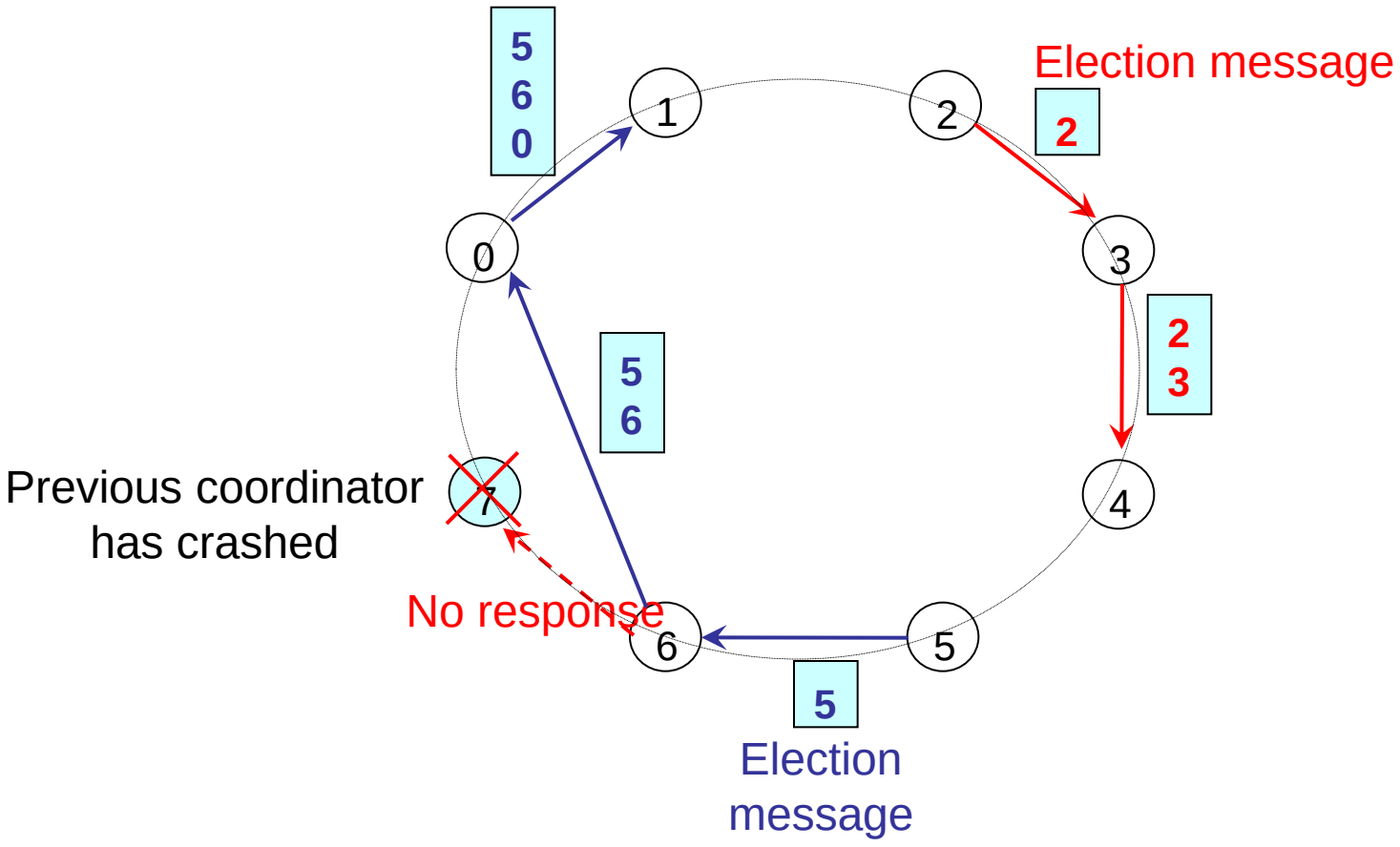
Example



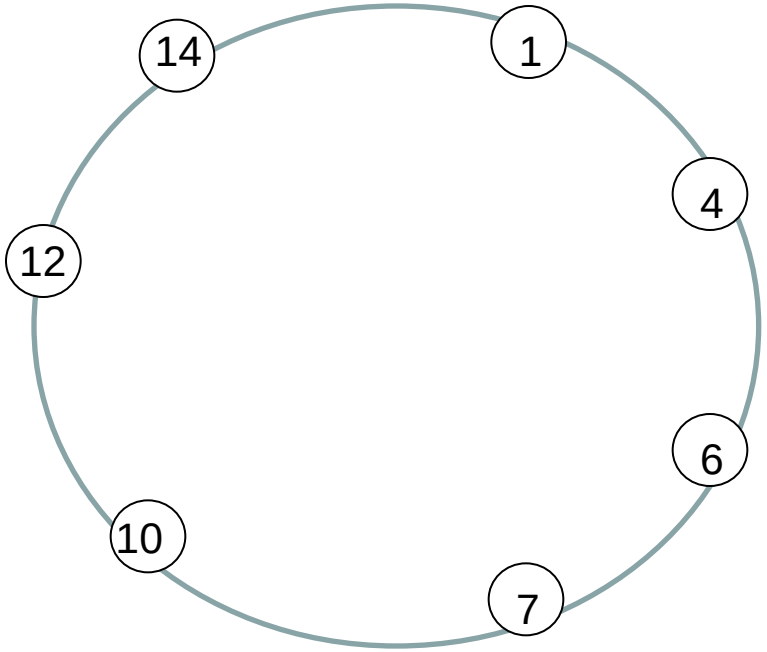
A Ring Algorithm

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

Example



Leader Election in a DHT



- Rather than select the highest ID node as the leader, select the node closest to a pre-determined ID in the ID space (such as 0).
- Resilient DHT routing should deliver requests to the closest node even with node failures.

Part 3
Sharing in Distributed Systems

Mutual Exclusion Problem

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

Protocols & Requirements

- Application-level protocol

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

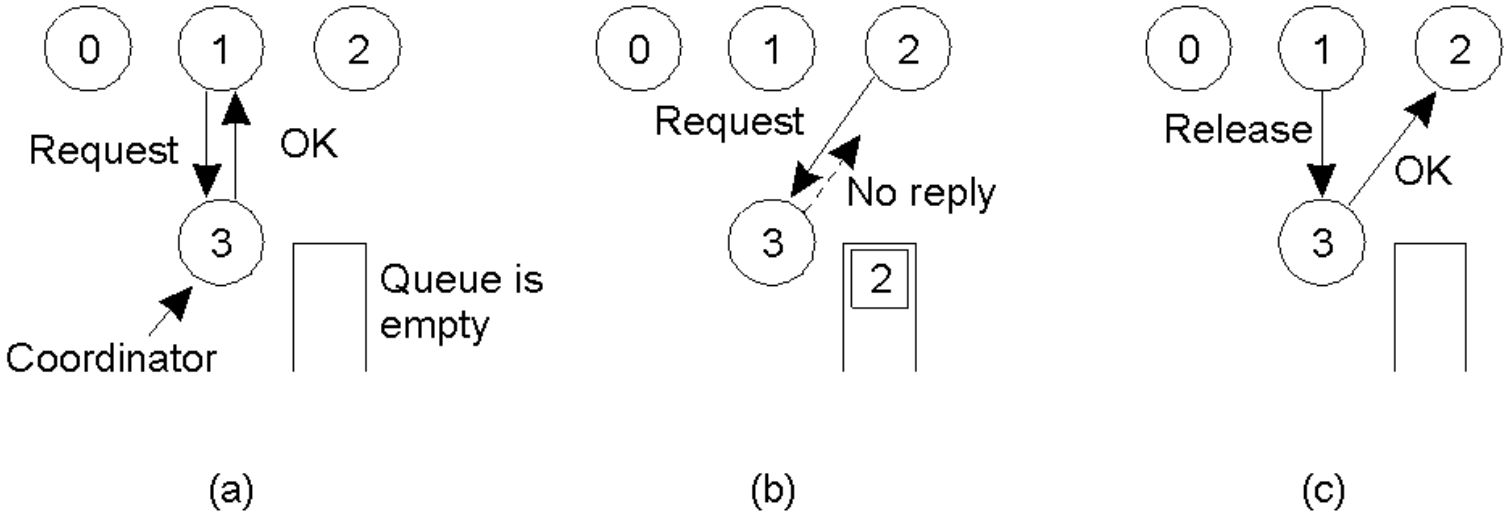
  resourceAccesses() // access shared resources in critical section

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

- Requirements

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

A Centralized Algorithm (Central Server Algorithm)



One process is elected as the coordinator:

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

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

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

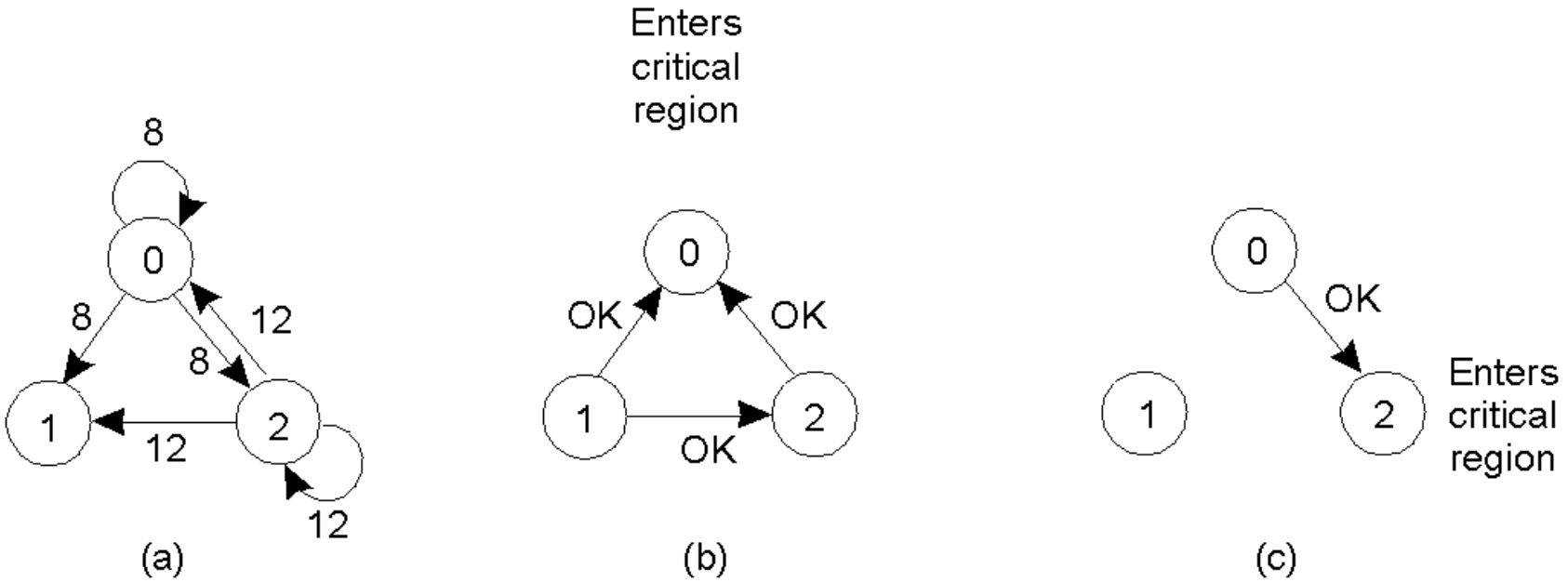
A Centralized Algorithm (2)

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

A Distributed Algorithm

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

Example



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

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

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

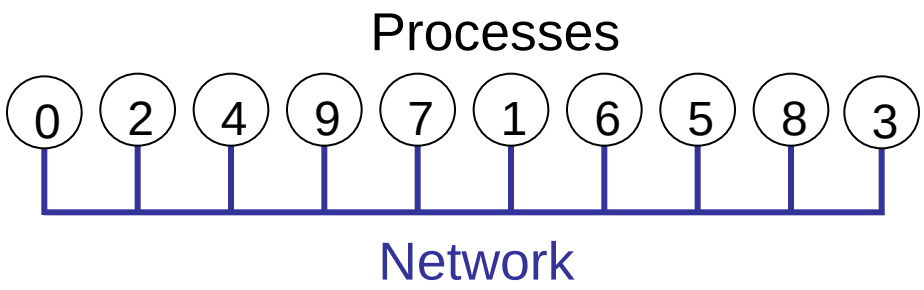
Problems

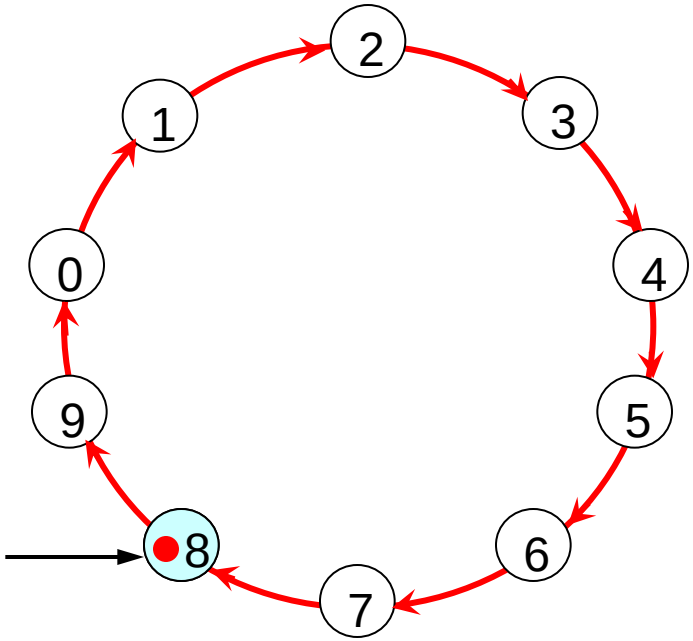
- The single point of failure has been replaced by *n* points of failure: if any process crashes, it will fail to respond to requests. This silence will be interpreted (incorrectly) as denial of permission, thus blocking all subsequent attempts by all processes to enter all critical regions

- If reliable multicasting is not available, each process must maintain the group membership list itself, including processes entering the group, leaving the group, and crashing

$\Rightarrow$ Slower, more complicated, more expensive, and less robust that centralized algorithm!

A Token Ring Algorithm





Token holder may enter critical region or pass the token

A Token Ring Algorithm (2)

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

Comparison of the 3 Algorithms

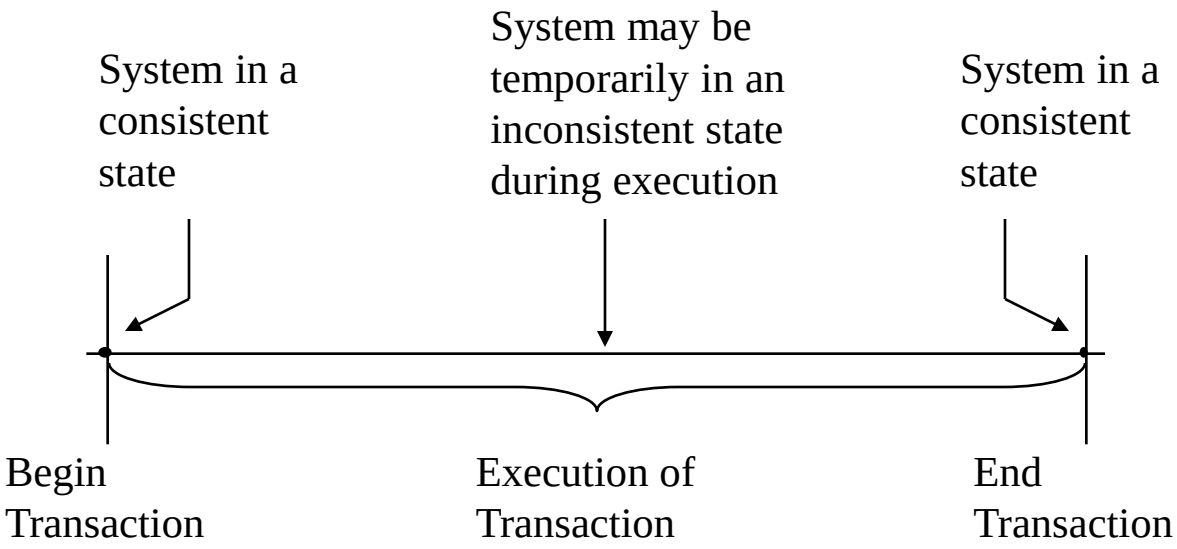
Algorithm	Messages per entry/exit	Delay before entry (in message times)	Problems
Centralized	3	2	Coordinator crash
Distributed	2 (n – 1)	2 (n – 1)	Crash of any process
Token ring	1 to ∞	0 to n – 1	Lost token, process crash

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

Transaction

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

- concurrency transparency
- failure transparency



Example Transaction

```
begin
    input(flight_no, date, customer_name);
    Begin_transaction Reservation
    begin
        Write(flight(date).stsold++);
        Write(flight(date).cname, customer_name);
        Commit
    end. {Reservation}
    output("reservation completed")
…
end
```

Example Transaction (2)

What if there are no free seats?

```
begin
    input(flight_no, date, customer_name);
    Begin_transaction Reservation
    begin
        temp ← Read(flight_no(date).stsold);
        if temp = flight(date).cap then
        begin
            cond ← Abort;
            Abort
        end
        else begin
            Write(flight(date).stsold, temp + 1);
            Write(flight(date).cname, customer_name);
            cond ← Commit;
            Commit
        end
    end {Reservation}
    if cond=Abort then output("no free seats")
        else output("reservation completed");
...
end
```

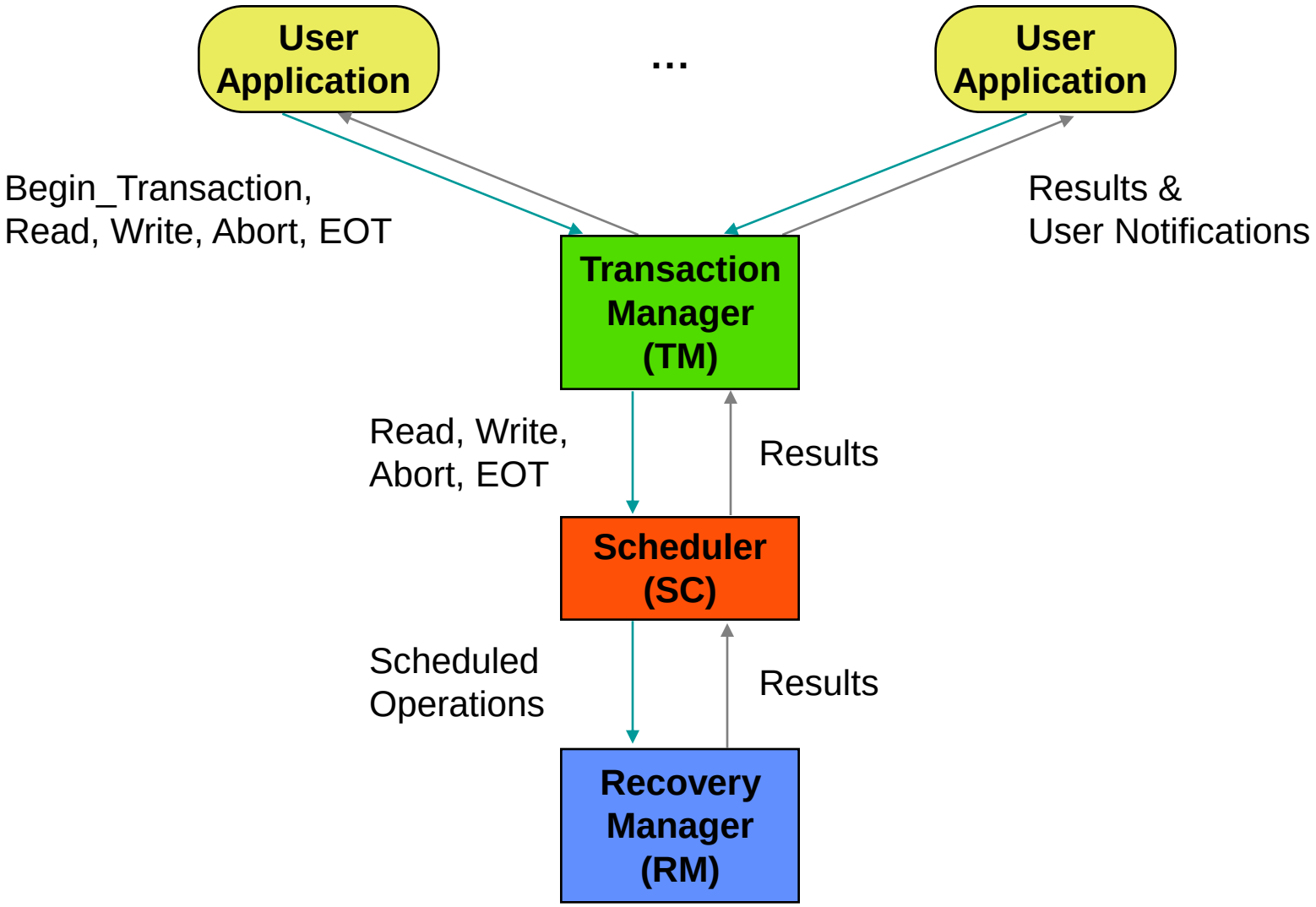
Characterization

- Read set (RS)
  - The set of objects that are read by a transaction
- Write set (WS)
  - The set of objects whose values are changed by this transaction
- Base set (BS)
  - RS $\cup$ WS

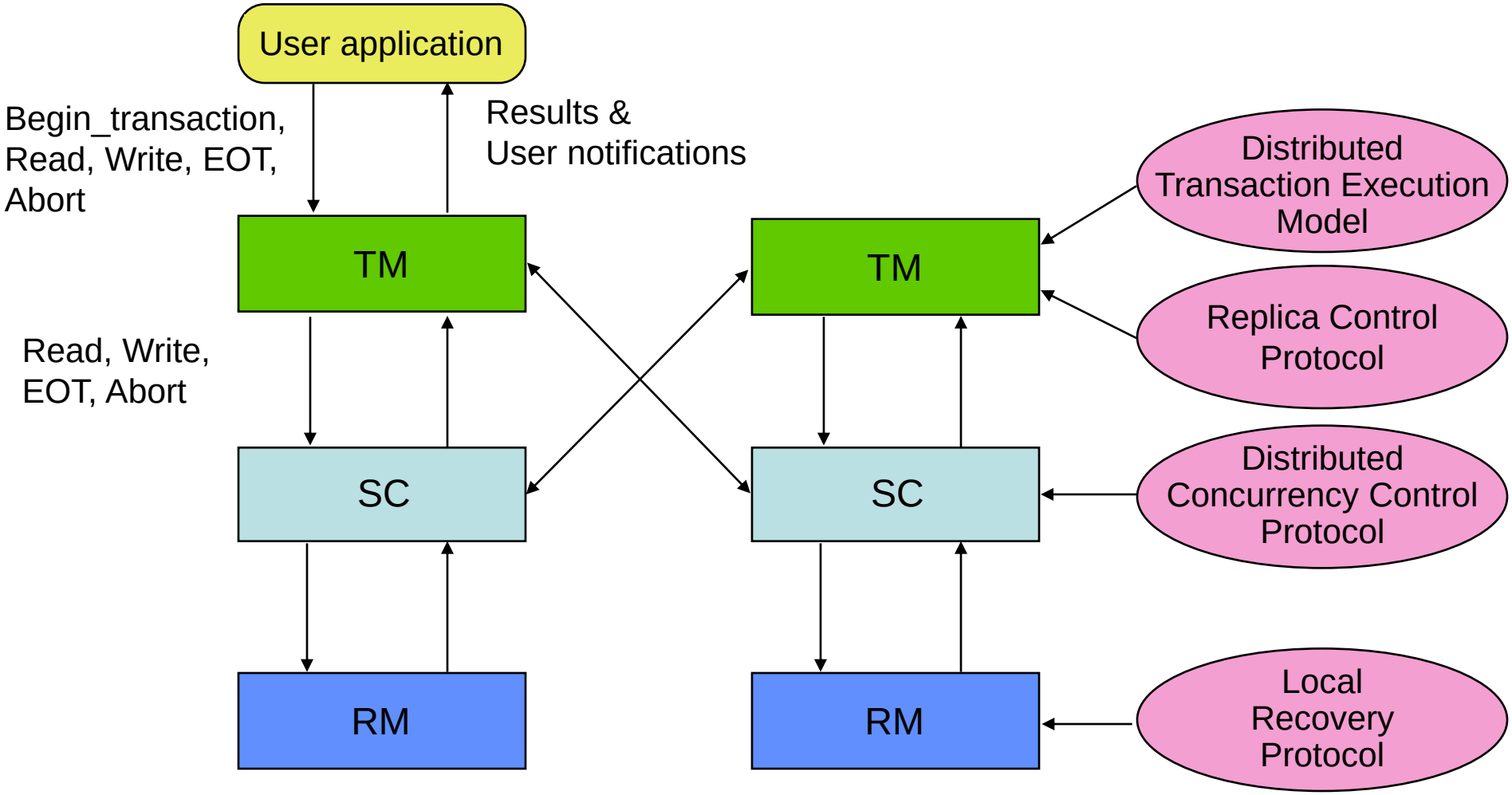
Transaction Primitives

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

Centralized Transaction Execution



Distributed Transaction Execution



Properties of Transactions

ATOMICITY

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

CONSISTENCY

- No violation of integrity constraints
  - → System specific rules. In a distributed database, the index should always reflect the data, foreign key constraints are not violated, triggers are issued, replicas have the same value, etc.
- Transactions are correct programs

ISOLATION ⇐ Our focus in this module

- Concurrent changes invisible → serializable

DURABILITY

- Committed updates persist
- Database recovery

Transactions Provide…

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

Isolation

- Isolation is based on execution histories (sequences).
- Consider the following two transactions:

T_1:	Read(x)	T_2:	Read(x)
	$x \leftarrow x+1$		$x \leftarrow x+1$
	Write(x)		Write(x)
	Commit		Commit

- Possible execution sequences:

T_1:	Read(x)	T_1:	Read(x)
T_1:	$x \leftarrow x+1$	T_1:	$x \leftarrow x+1$
T_1:	Write(x)	T_2:	Read(x)
T_1:	Commit	T_1:	Write(x)
T_2:	Read(x)	T_2:	$x \leftarrow x+1$
T_2:	$x \leftarrow x+1$	T_2:	Write(x)
T_2:	Write(x)	T_1:	Commit
T_2:	Commit	T_2:	Commit

Isolation Problems

- Serializability
  - If several transactions are executed concurrently, the results must be the same as if they were executed serially in some order.
- Incomplete results
  - An incomplete transaction cannot reveal its results to other transactions before its commitment.
  - Necessary to avoid cascading aborts.
  - Anomalies:
    - → Lost updates
      - The effects of some transactions are not reflected on the database.
    - → Inconsistent retrievals
      - E.g. a transaction, if it reads the same object more than once, should always read the same value.

Lost Update Problem

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

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

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

Inconsistent Retrievals Problem

Initial values: A=200, B=200

Transaction *V*:		**Transaction *W*:**	
a.withdraw(100) *b.deposit(100)*		*aBranch.branchTotal()*	
a.withdraw(100);	$100		
		total = a.getBalance()	$100
		total = total+b.getBalance()	$300
		total = total+c.getBalance()	
b.deposit(100)	$300	⋮	

Execution Schedule (or History)

- An order in which the operations of a set of transactions are executed.
- A schedule (history) can be defined as a partial order over the operations of a set of transactions.

T_1:	T_2:	T_3:
Read(x)	Write(x)	Read(x)
Write(x)	Write(y)	Read(y)
Commit	Read(z)	Read(z)
	Commit	Commit

$$H_1=\{W_2(x),R_1(x), R_3(x),W_1(x),C_1,W_2(y),R_3(y),R_2(z),C_2,R_3(z),C_3\}$$

Serial History

- All the actions of a transaction occur consecutively.
- No interleaving of transaction operations.
- If each transaction is consistent (obeys integrity rules), then the database is guaranteed to be consistent at the end of executing a serial history.

T_1:	T_2:	T_3:
Read(x)	Write(x)	Read(x)
Write(x)	Write(y)	Read(y)
Commit	Read(z)	Read(z)
	Commit	Commit

$$H_s=\{W_2(x),W_2(y),R_2(z),C_2,R_1(x),W_1(x),C_1,R_3(x),R_3(y),R_3(z),C_3\}$$

Serializable History

- Transactions execute concurrently, but the net effect of the resulting history upon the database is equivalent to some serial history.
- Equivalent with respect to what?
  - Conflict equivalence: the relative order of execution of the conflicting operations belonging to non-aborted transactions in two histories are the same.
  - Conflicting operations: two incompatible operations (e.g., Read and Write) conflict if they both access the same object.
    - → Incompatible operations of each transaction is assumed to conflict; do not change their execution orders.
    - → If two operations from two different transactions conflict, the corresponding transactions are also said to conflict.

Conflict Rules

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

Serializable History

T_1:	T_2:	$T3$:
Read(x)	Write(x)	Read(x)
Write(x)	Write(y)	Read(y)
Commit	Read(z)	Read(z)
	Commit	Commit

The following are not conflict equivalent (but H_1 is serializable)

$$H_s=\{W_2(x),W_2(y),R_2(z),C_2,R_1(x),W_1(x),C_1,R_3(x),R_3(y),R_3(z),C_3\}$$

$$H_1=\{W_2(x),R_1(x),\ R_3(x),W_1(x),C_1,W_2(y),R_3(y),R_2(z),C_2,R_3(z),C_3\}$$

H_2 is not conflict equivalent to H_s and not serializable

$$H_2=\{W_2(x),R_1(x),\ R_3(x),W_1(x),C_1,\ R_3(y),\ W_2(y),\ R_2(z),C_2,R_3(z),C3\}$$

H_3 is conflict equivalent to H_s and, therefore, is *serializable*.

$$H_3=\{W_2(x),R_1(x),W_1(x),C_1,R_3(x),W_2(y),R_3(y),R_2(z),C_2,R_3(z),C_3\}$$

Resolving Lost Update Example

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

Resolving Inconsistent Retrieval Problem

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

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

Distributed Transaction Serializability

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

Global Non-serializability

T_1: Read(x)
$x \leftarrow x+5$
Write(x)
Commit

T_2: Read(x)
$x \leftarrow x*15$
Write(x)
Commit

The following two local histories are individually serializable (in fact serial), but the two transactions are not globally serializable.

$$LH_1=\{R_1(x),W_1(x),C_1,R_2(x),W_2(x),C_2\}$$

$$LH_2=\{R_2(\mathrm{x}),W_2(\mathrm{x}),C_2,R_1(\mathrm{x}),W_1(\mathrm{x}),C_1\}$$

Concurrency Control

- The problem of synchronizing concurrent transactions such that the consistency of the database is maintained while, at the same time, maximum degree of concurrency is achieved.
- Algorithms:
  - Two-Phase Locking-based (2PL)
    - → Centralized (primary site) 2PL
    - → Distributed 2PL
  - Timestamp Ordering (TO)
    - → Basic TO
    - → Multiversion TO

Locking-Based Algorithms

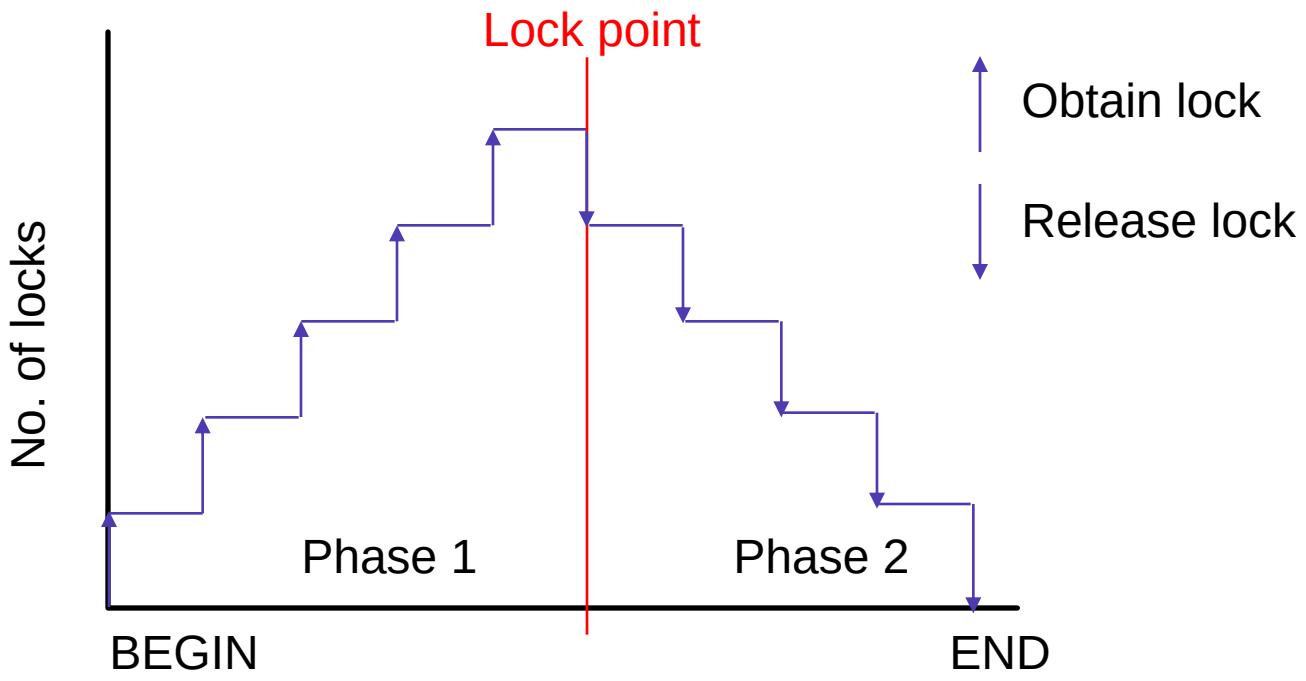
- Transactions indicate their intentions by requesting locks from the scheduler (called lock manager).
- Locks are either read lock (*rl*) [also called shared lock] or write lock (*wl*) [also called exclusive lock]
- Read locks and write locks conflict (because Read and Write operations are incompatible)

	rl	*wl*
rl	yes	no
wl	no	no

- Locking works nicely to allow concurrent processing of transactions.

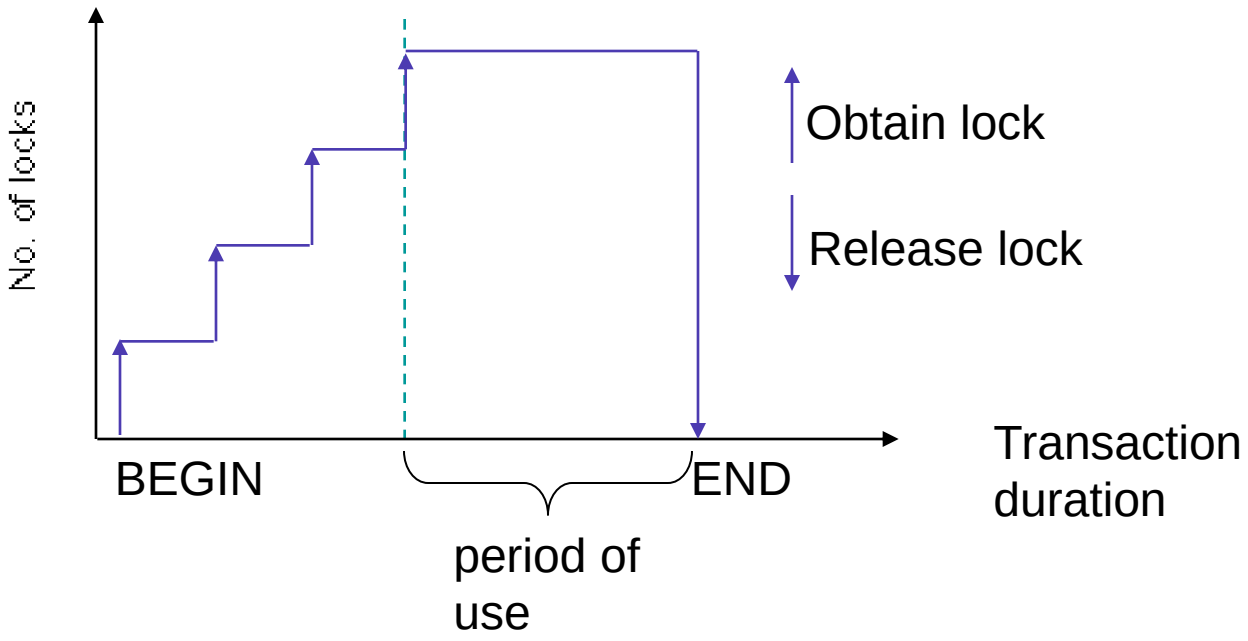
Two-Phase Locking (2PL)

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



Strict 2PL

Hold locks until the end.



Locking Example

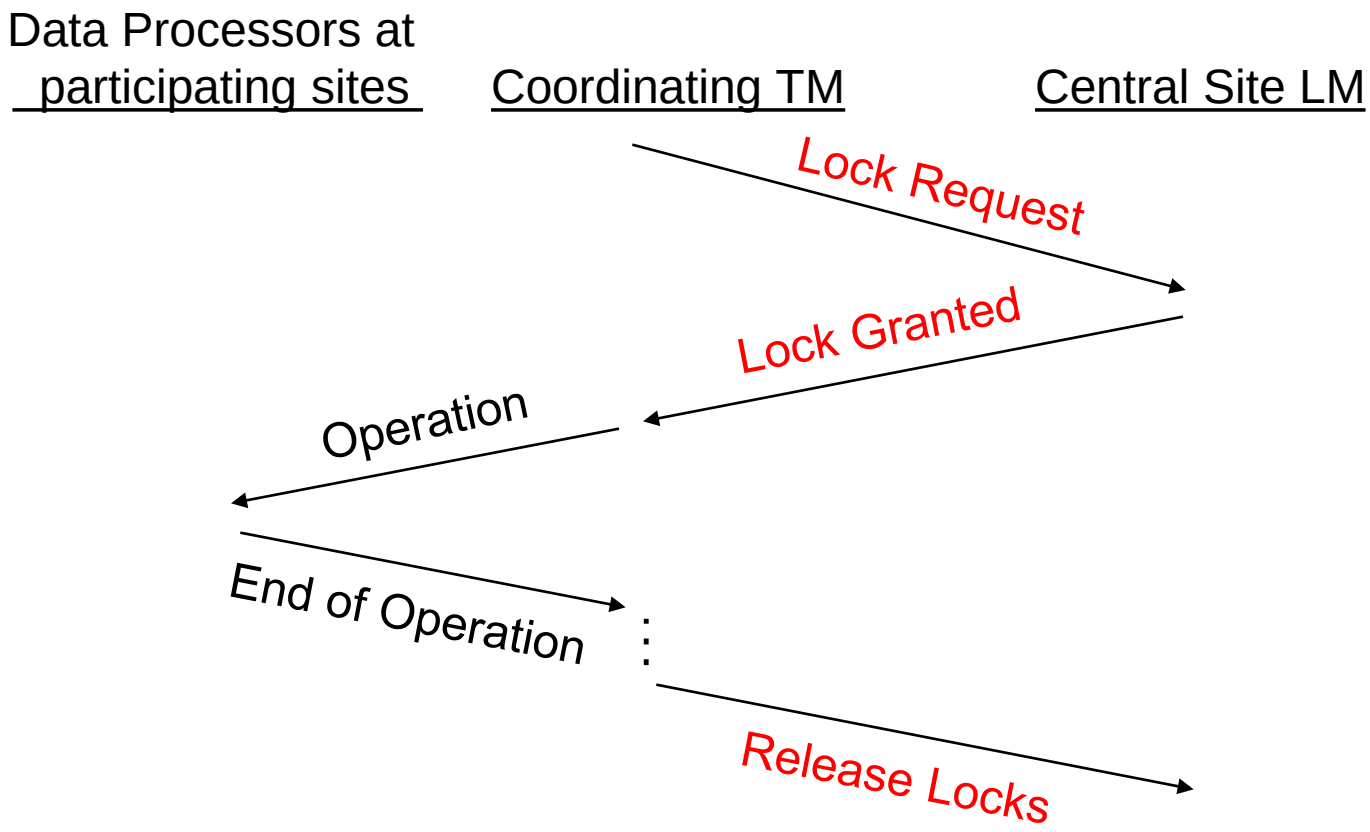
Transaction *T*:		**Transaction *U*:**	
balance = b.getBalance() *b.setBalance(bal*1.1)* *a.withdraw(bal/10)*		*balance = b.getBalance()* *b.setBalance(bal*1.1)* *c.withdraw(bal/10)*	
Operations	Locks	Operations	Locks
openTransaction			
bal = b.getBalance()	write lock *B*		
*b.setBalance(bal*1.1)*		*openTransaction*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

Locking Actions

1. When an operation accesses an object within a transaction:
   (a) If the object is not already locked, it is locked and the operation proceeds.
   (b) If the object has a conflicting lock set by another transaction, the transaction must wait until it is unlocked.
   (c) If the object has a non-conflicting lock set by another transaction, the lock is shared and the operation proceeds.
   (d) If the object has already been locked in the same transaction, the lock will be promoted if necessary and the operation proceeds. (Where promotion is prevented by a conflicting lock, rule (b) is used.)
2. When a transaction is committed or aborted, the server unlocks all objects it locked for the transaction.

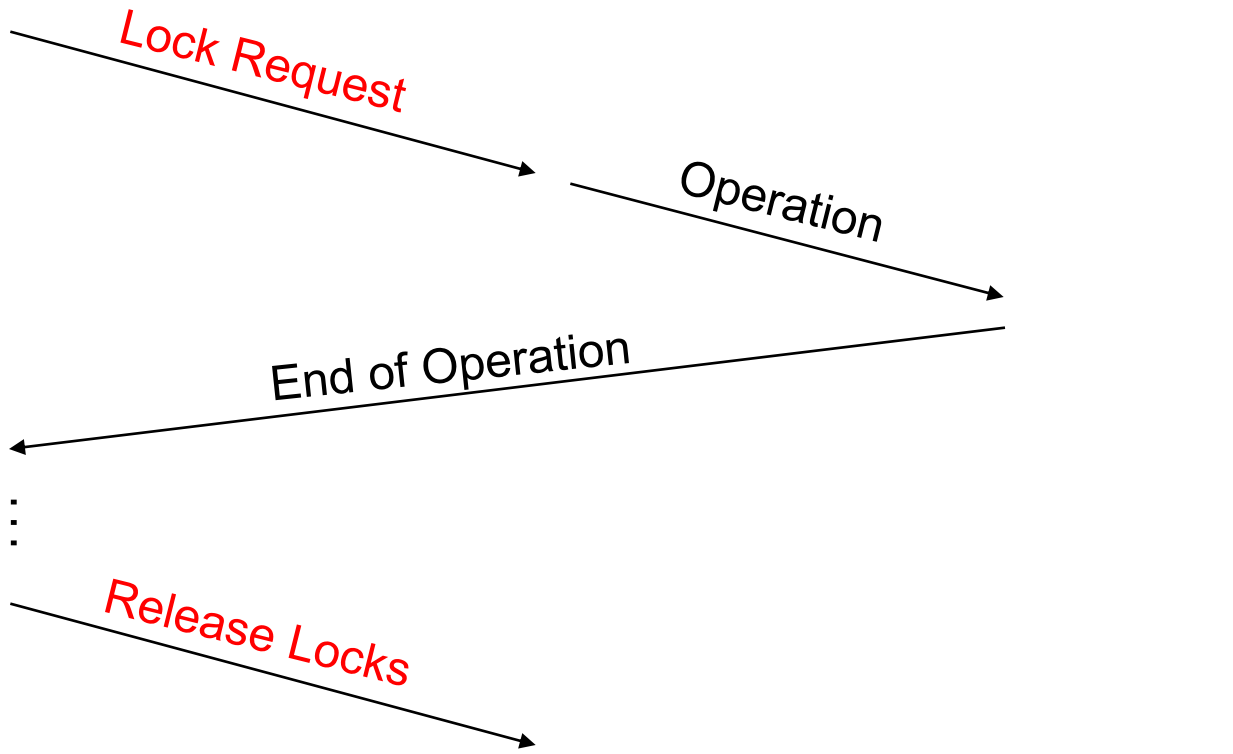
Centralized 2PL

- There is only one 2PL scheduler in the distributed system.
- Lock requests are issued to the central scheduler.



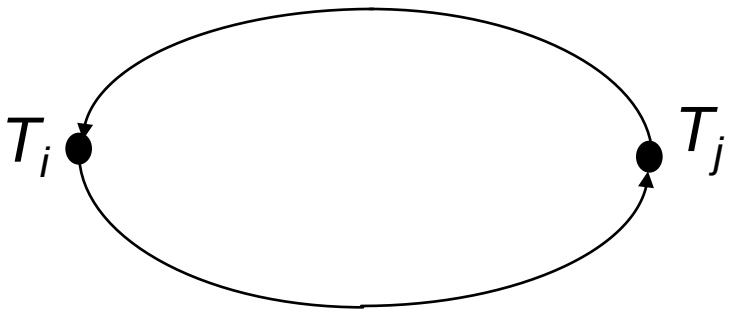
Distributed 2PL

- 2PL schedulers are placed at each site. Each scheduler handles lock requests for objects at that site.



Deadlock

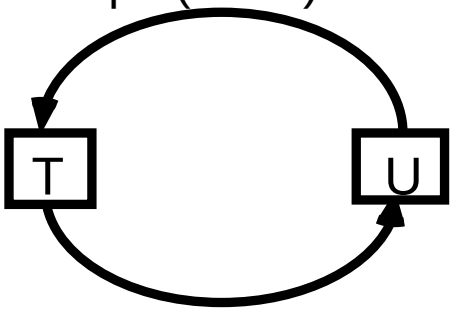
- A transaction is deadlocked if it is blocked and will remain blocked until there is intervention.
- Locking-based CC algorithms may cause deadlocks.
- Wait-for graph
  - If transaction T_i waits for another transaction T_j to release a lock on an entity, then $T_i \rightarrow T_j$ in WFG.

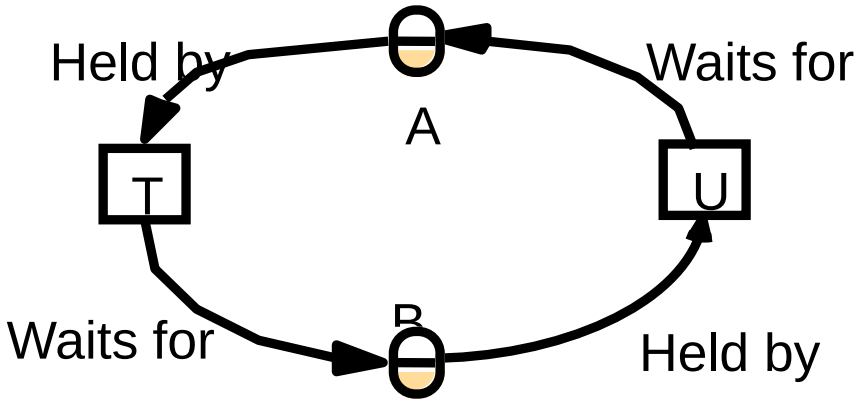


Deadlock Due To Locking

Transaction *T*		Transaction *U*	
Operations	**Locks**	**Operations**	**Locks**
a.deposit(100);	write lock *A*		
		b.deposit(200)	write lock *B*
b.withdraw(100)			
• • •	waits for *U*'s lock on *B*	*a.withdraw(200);*	waits for *T*'s lock on A
		• • •	
• • •		• • •	

Wait-for Graph (WFG)





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

Resolution of Deadlock

Transaction *T*		Transaction *U*	
Operations	**Locks**	**Operations**	**Locks**
a.deposit(100);	write lock *A*		
		b.deposit(200)	write lock *B*
b.withdraw(100)			
•••	waits for *U*'s lock on *B*	*a.withdraw(200);*	waits for *T*'s lock on *A*
	(timeout elapses) *T*'s lock on *A* becomes vulnerable, unlock *A*, abort *T*	•••	
		•••	
		a.withdraw(200);	write locks *A* unlock *A*, *B*

Local versus Global WFG

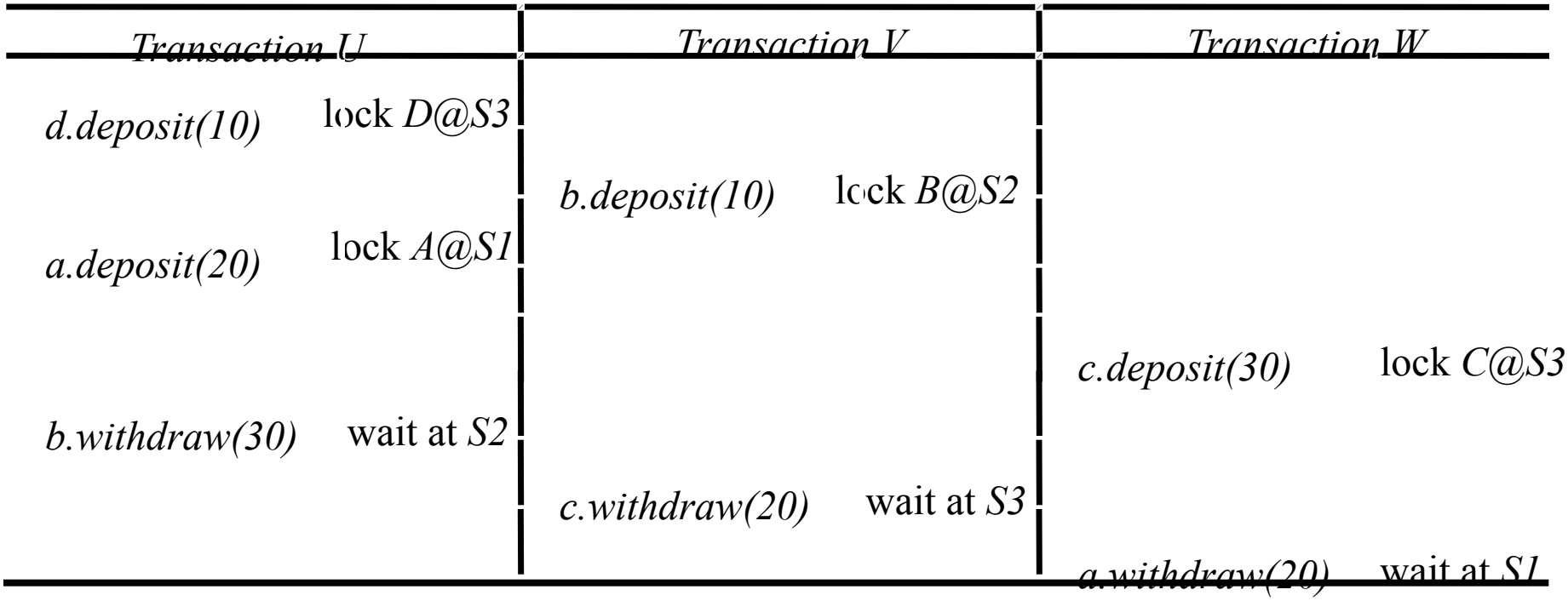
Assume T_1 and T_2 run at site 1, T_3 and T_4 run at site 2. Also assume T_3 waits for a lock held by T_4 which waits for a lock held by T_1 which waits for a lock held by T_2 which, in turn, waits for a lock held by T_3.

Local WFG

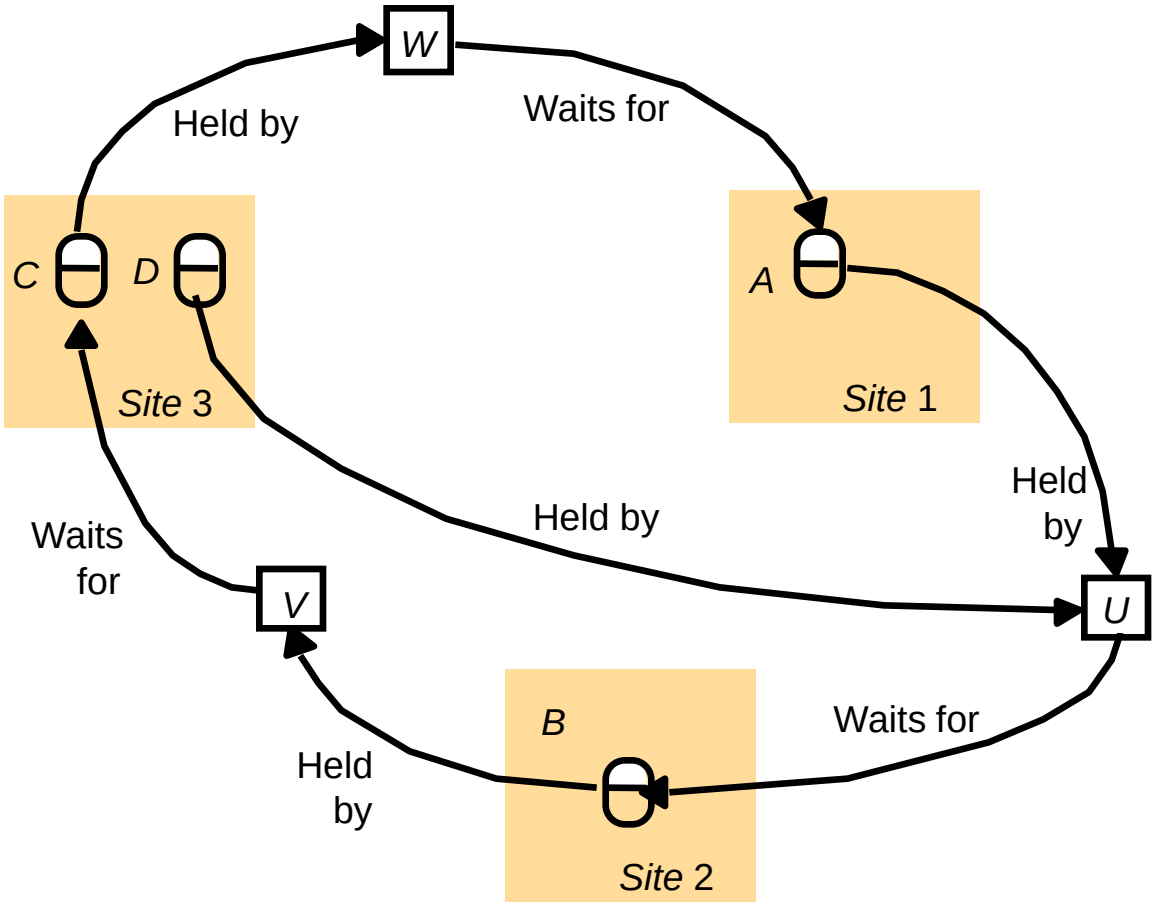
Global WFG

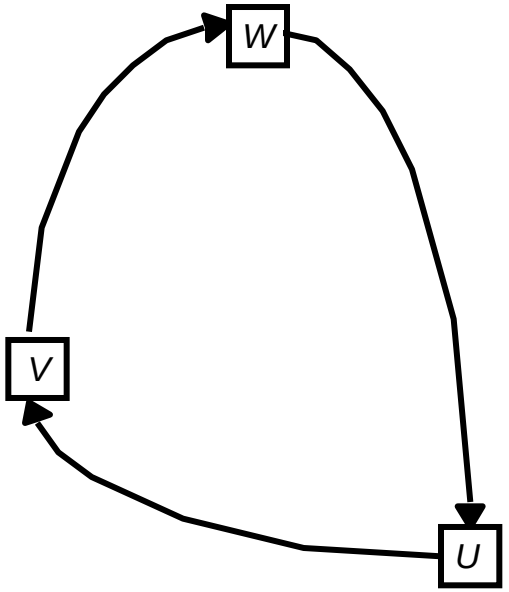
Distributed Deadlock Example

Site 1 stores object A, Site 2 stores object B, Site 3 stores objects C, D



Distributed Deadlock Example (2)





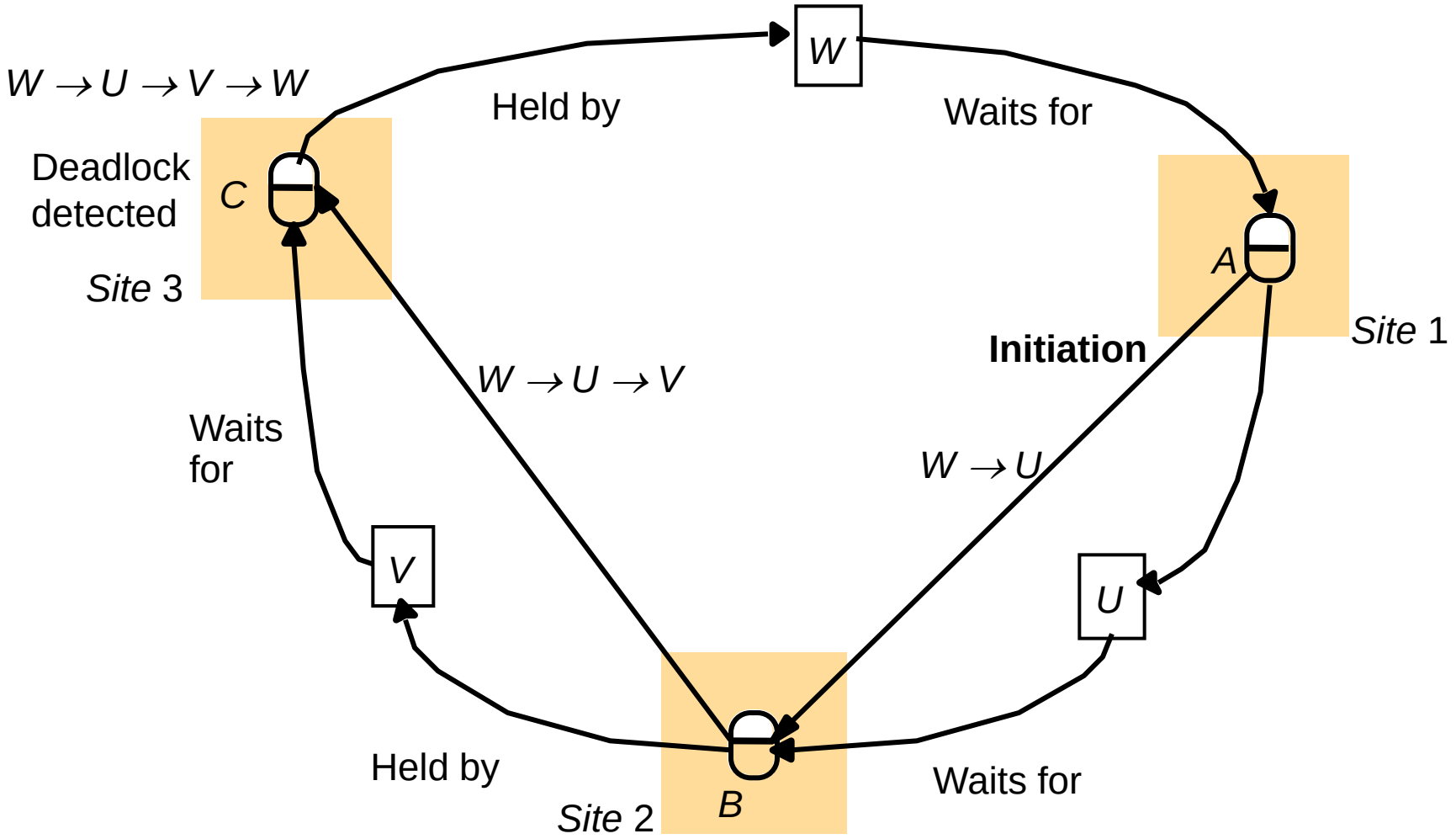
Deadlock Management

- Prevention
  - Guaranteeing that deadlocks can never occur in the first place. Check transaction when it is initiated. Requires no run time support.
  - All resources which may be needed by a transaction must be predeclared
- Avoidance
  - Detecting potential deadlocks in advance and taking action to insure that deadlock will not occur. Requires run time support.
  - Order either the objects or the sites and always request locks in that order.
- Detection and Recovery
  - Allowing deadlocks to form and then finding and breaking them. As in the avoidance scheme, this requires run time support.
  - Centralized
  - Hierarchical
  - Distributed

Distributed Deadlock Detection

- Sites cooperate in detection of deadlocks.
- One example:
  - The local WFGs are formed at each site and passed on to other sites. Each local WFG is modified as follows:
    - ☆ Since each site receives the potential deadlock cycles from other sites, these edges are added to the local WFGs
    - 🕐 The edges in the local WFG which show that local transactions are waiting for transactions at other sites are joined with edges in the local WFGs which show that remote transactions are waiting for local ones.
  - Each local deadlock detector:
    - → looks for a cycle that does not involve the external edge. If it exists, there is a local deadlock which can be handled locally.
    - → looks for a cycle involving the external edge. If it exists, it indicates a potential global deadlock. Pass on the information to the next site.

Probing



Timestamp Ordering

- Transaction (T_i) is assigned a globally unique timestamp $ts(T_i)$.
- Transaction manager attaches the timestamp to all operations issued by the transaction.
- Each object is assigned a write timestamp (*wts*) and a read timestamp (*rts*):
  - $rts(x)$ = largest timestamp of any read on x
  - $wts(x)$ = largest timestamp of any write on x
- Conflicting operations are resolved by timestamp order.

Basic T/O:

for $R_i(x)$
if $ts(T_i) < wts(x)$
then reject $R_i(x)$
else {accept $R_i(x)$
$rts(x) \leftarrow ts(T_i)$ }

for $W_i(x)$
if $ts(T_i) < rts(x)$ **or** $ts(T_i) < wts(x)$
then reject $W_i(x)$
else {accept $W_i(x)$
$wts(x) \leftarrow ts(T_i)$ }

Conflicts in Timestamp Ordering

Rule	T_c	T_i	
1.	*write*	*read*	T_c must not *write* an object that has been *read* by any T_i where $T_i > T_c$ this requires that $T_c \geq$ the maximum read timestamp of the object.
2.	*write*	*write*	T_c must not *write* an object that has been *written* by any T_i where $T_i > T_c$ this requires that $T_c >$ write timestamp of the committed object.
3.	*read*	*write*	T_c must not *read* an object that has been *written* by any T_i where $T_i > T_c$ this requires that $T_c >$ write timestamp of the committed object.

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

Multiversion Timestamp Ordering

- Do not modify the values in the database, create new rules
  - Each data item *x* has a sequence of version $<x_1, x_2, \ldots x_m>$
- $wts(x_k)$: timestamp of the transaction that created x_k
- $rts(x_k)$: largest timestamp of a transaction that successfully read x_k
- When a new x_k is created by T_i, the $wts(x_k)$ is initialize to the timestamp of T_i
- $rts(x_k)$ updated whenever a transaction with a bigger timestamp reads it.

Multiversion Timestamp Ordering

- If transaction T_i issues a read(x), then the system returns the x_k with largest write timestamp amongst those with write timestamps that are less than or equal to $TS(T_i)$.
- If transaction T_i issues a write(x) and if $TS(T_i) < rts(x_k)$, then abort. Otherwise, create a new version of x.

- Reads always succeed!
- Writes by T_i is rejected if some other transaction T_j that should read T_i's write had already read an older version.

Atomic Commit Protocols

- Atomicity property:
  - Either all operations must be carried out, or none of the operations
- For distributed transactions, this involves multiple servers

- Simple solution is have the transaction coordinator broadcast the commit/abort message
  - Why is this inadequate?
    - → It prevents individual servers from making unilateral decision to abort a transaction
    - → For example, if we are using locking, a server cannot abort a transaction to resolve a deadlock
    - → Server might have crashed without the coordinator knowing.

Two-Phase Commit Protocol

- Trying to reach consensus on whether to commit the transaction.