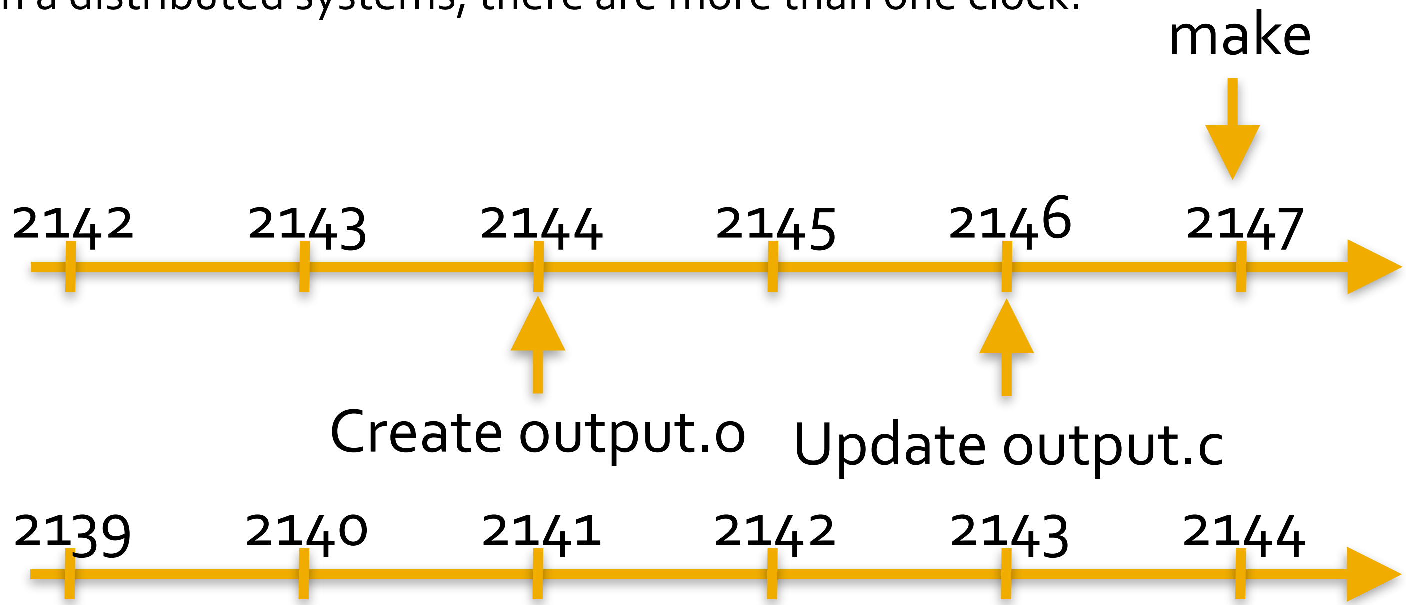
Synchronization

Clock Synchronization

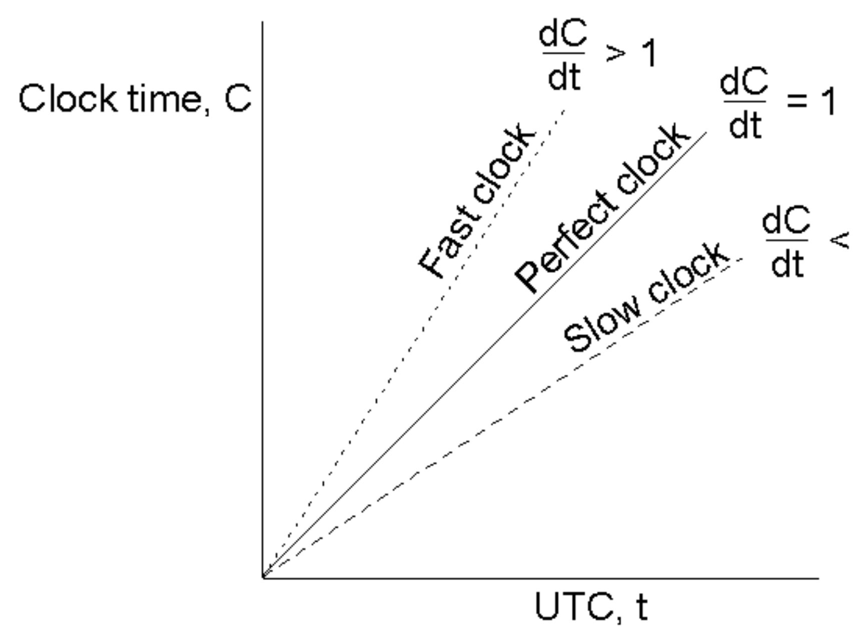
- When each machine has its own clock, an event that occurred after another event may nevertheless be assigned an earlier time.
- Example:
  - make check the dependence based on the last updated time of dependent files.
  - In a uni-processor system, there is only one clock.
  - In a distributed systems, there are more than one clock.



Clock Synchronization Algorithms

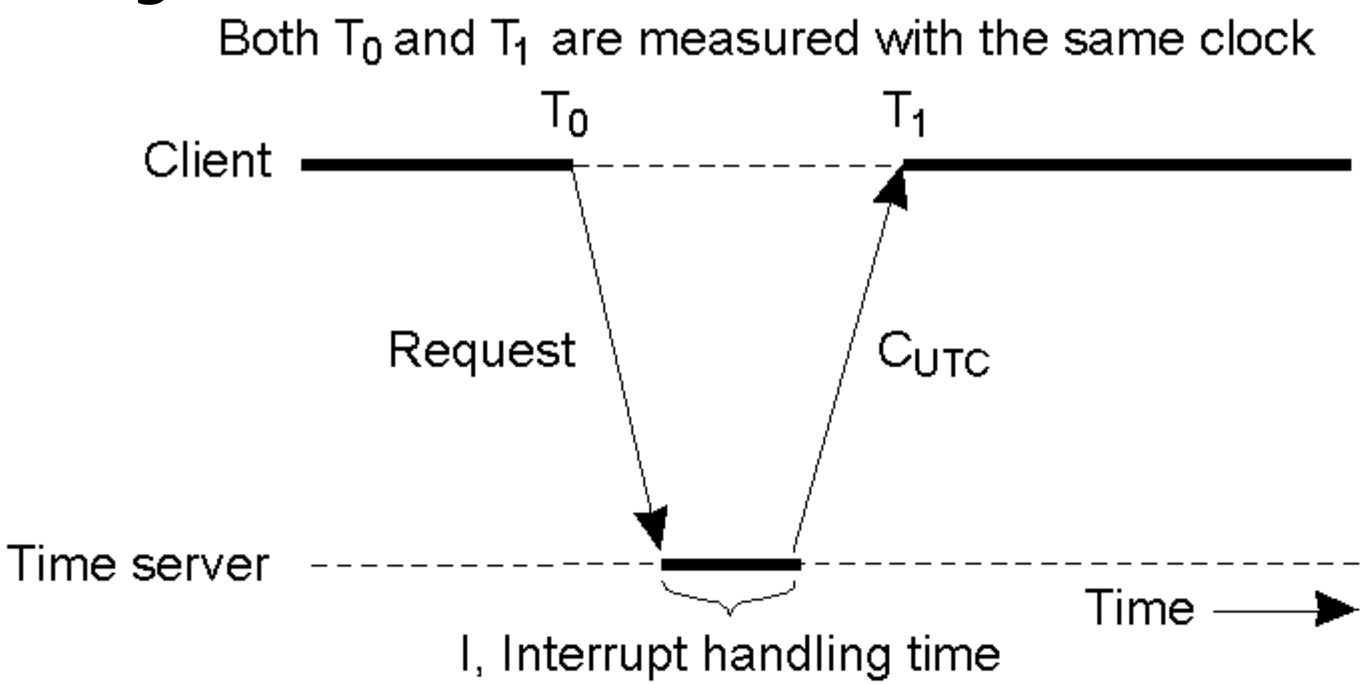
- When no machines have WWV receivers, each machine keeps its own clock.
- When one machine has the WWV receiver, we may still have troubles to synchronize the other machines.
- The relation between clock time and UTC when clocks tick at different rates are given.
  - When the UTC time is t, the value of the clock on machine p is C(t).
  - How often should the computers be synchronized?

If the drift of any two clocks should be no more than δ and the maximum drift rate is ρ, when should the clocks be synchronized?



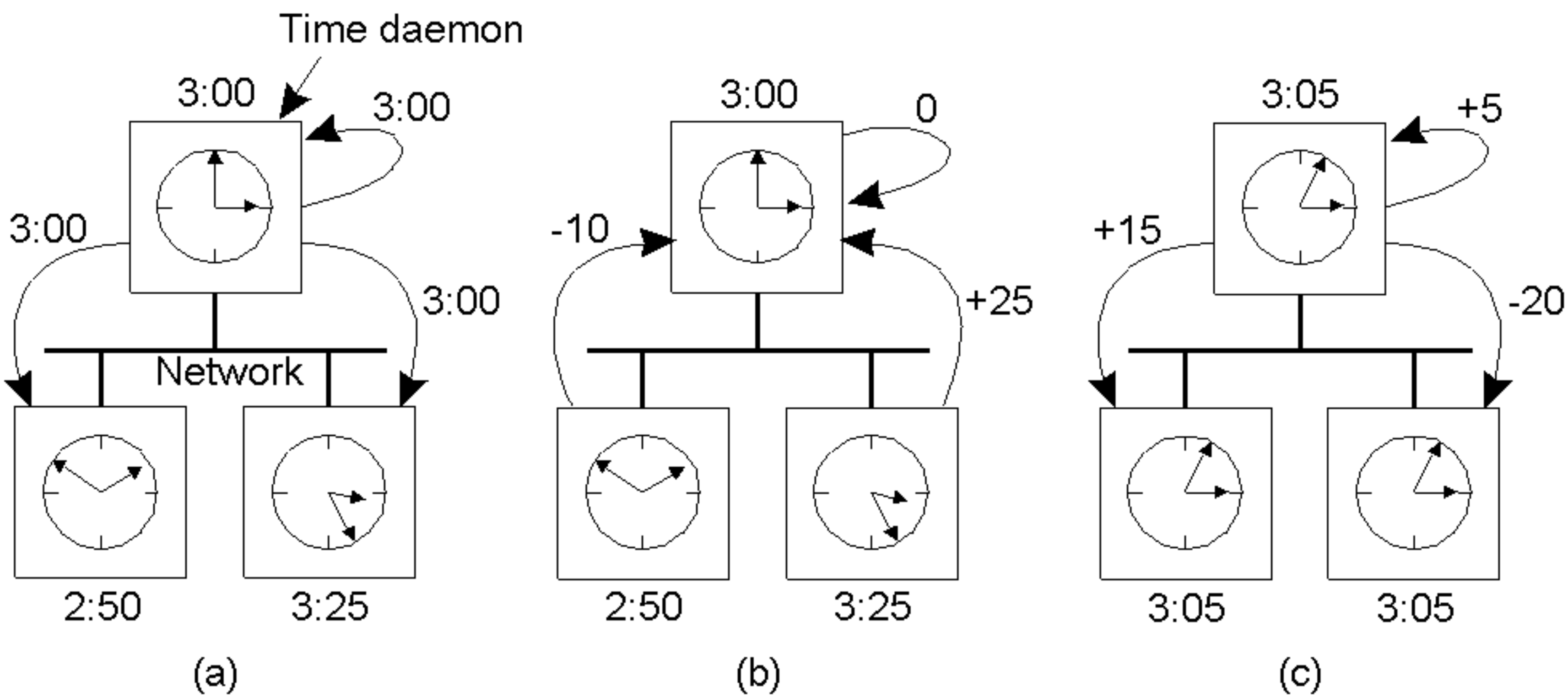
Cristian's Algorithm (External synchronization) - Passive Time Server Centralized Algorithm

- Assume that one machine has a WWV receiver.
- Getting the current time from a time server.



- Problem? One major and one minor problem.

The Berkeley Algorithm (Mutual or internal synchronization) - Active Time Server Centralized Algorithm

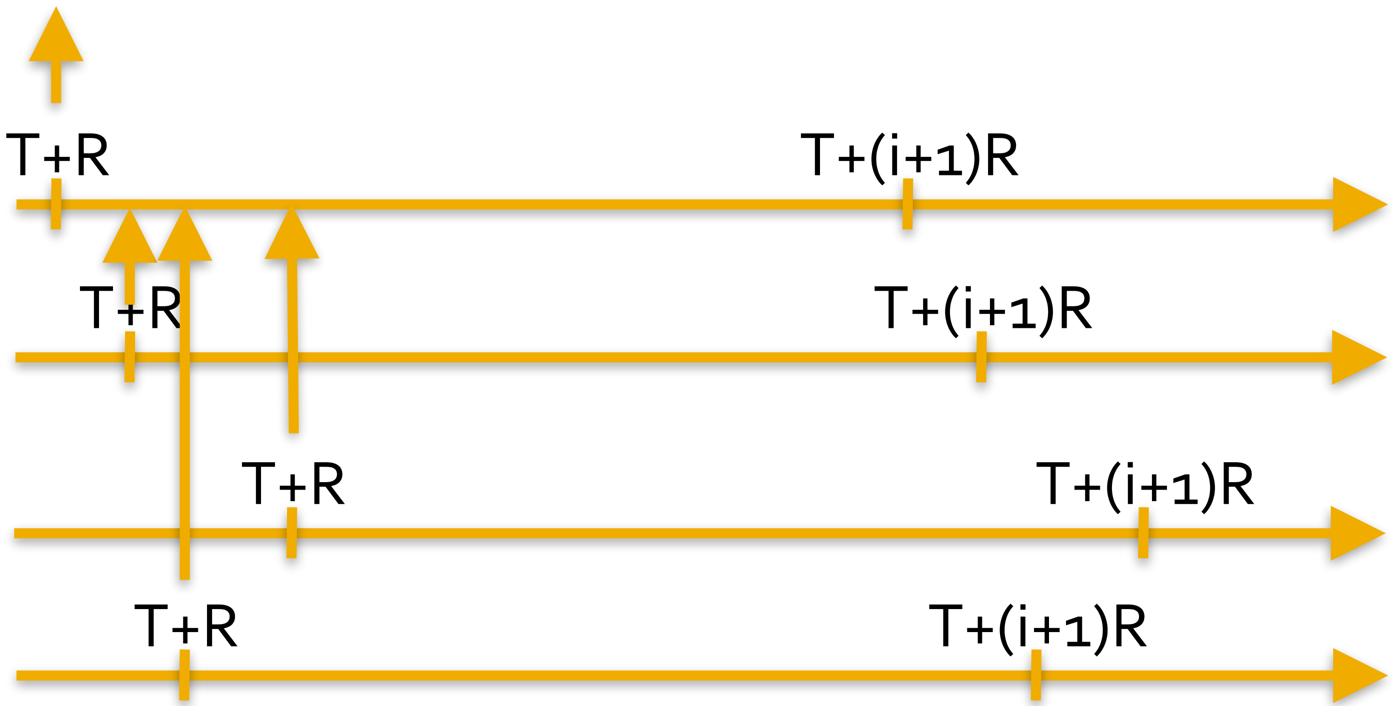


The clock values are selected with a threshold to ignore faulty clocks.

The calculated new time is broadcast to all clocks for synchronization.

Global Averaging Distributed Algorithm

- Distributed clock algorithm to synchronize the local clock with the other clocks.



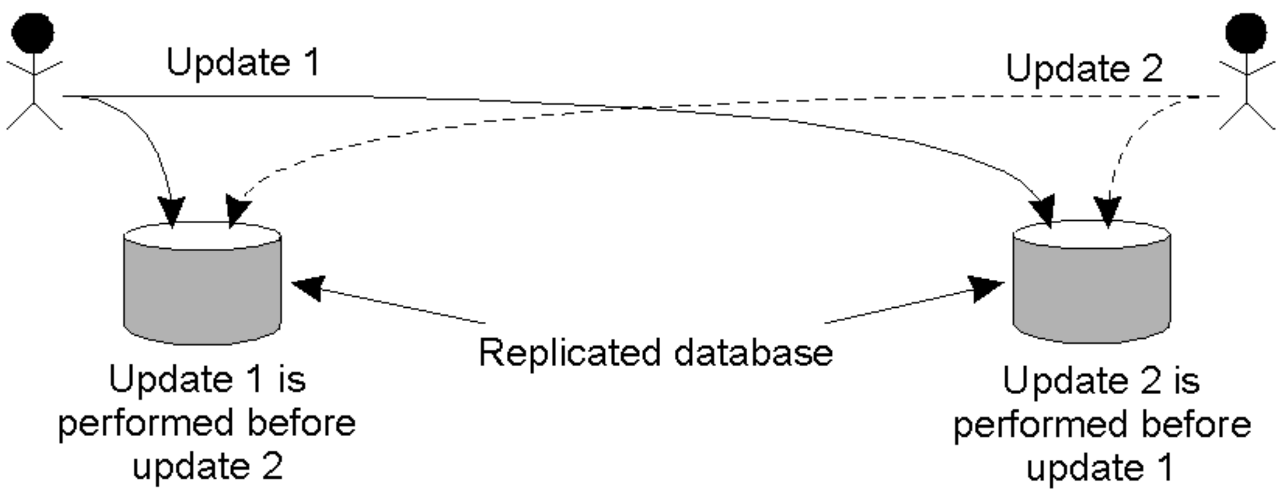
- NTP is accurate in the range of 1 – 50 msec.

Use of Synchronized Clocks

- With the new technology, it is possible to keep millions of clocks synchronized to within a few milliseconds.
- One use scenario is to enforce at-most-once message, e.g., Heart-beat, delivery to a server, even in the face of crashes.
  - Traditional approaches assign each message a unique message number.
  - The server keeps a table for received messages. But, the server may crash and lost the table.
  - With **global timestamp**, each server keeps a global variable:
    - G = CurrentTime –MaxLifeTime – MaxClockSkew
  - The global variable is written to the disk periodically.
  - The message is accepted only when its timestamp is later than the global timestamp. Otherwise, it is rejected because it could be accepted earlier or too old to accept.

Another Example: Totally-Ordered Multicasting

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

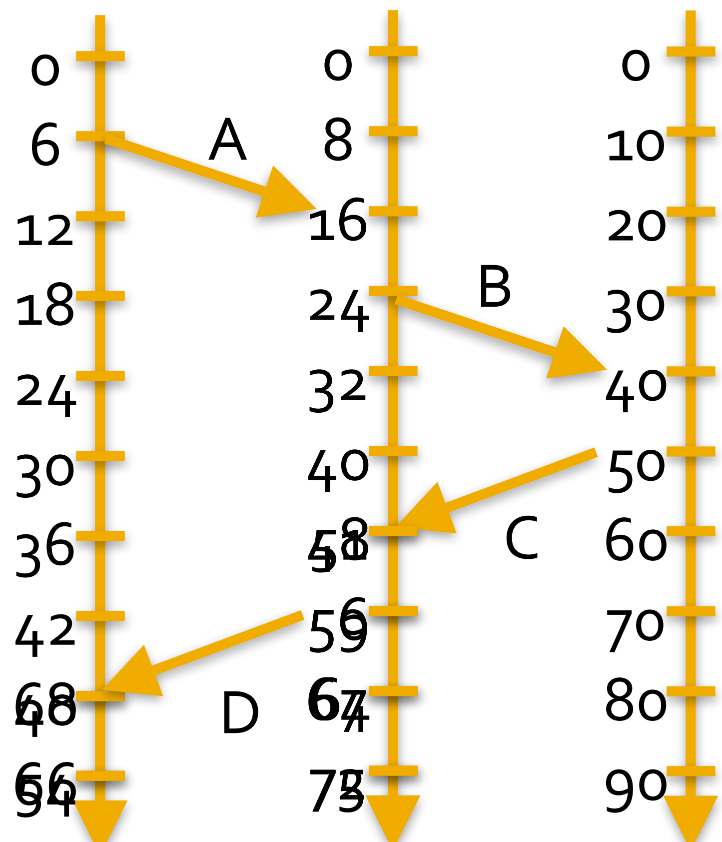


Logical Clocks

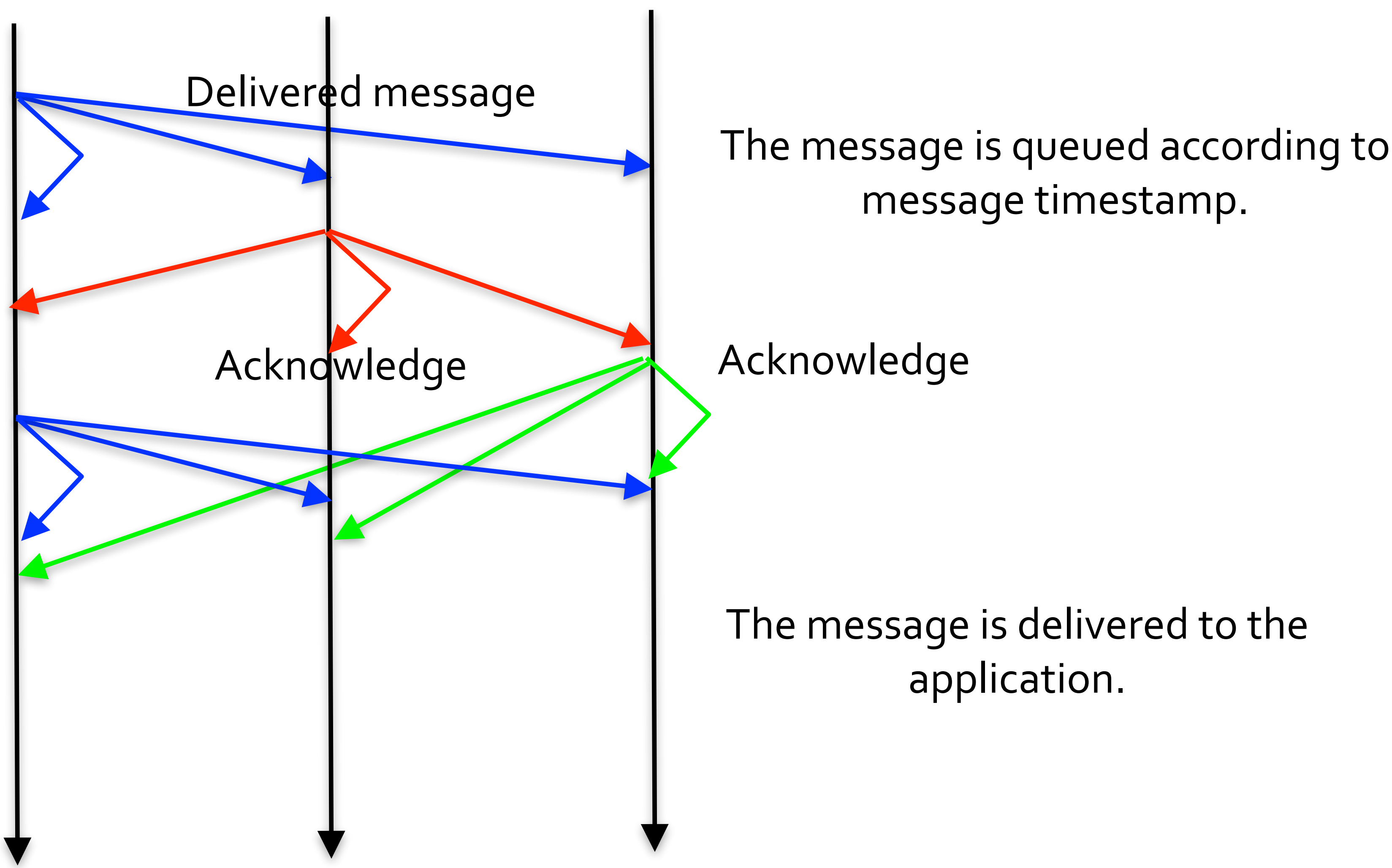
- Not necessary to synchronize all machines to the real time (or clock on the wall).
- It is sufficient that all machines agree on the same time — ***logical clock***.
- Lamport Timestamps:
  - ***a*** -> ***b***: event ***a*** happens before event ***b*** if all processes agree that event ***a*** happens before event ***b***.
  - The relation can be observed directly in two situations:
    - If ***a*** and ***b*** are events in the same process, and ***a*** occurs before ***b***, then ***a*** -> ***b*** is true.
    - If ***a*** is the event of a message being sent by one process, and ***b*** is the event of the message being received by another process, ***a*** -> ***b*** is also true.
  - "Happens before" is a transitive relation.

How to order the events

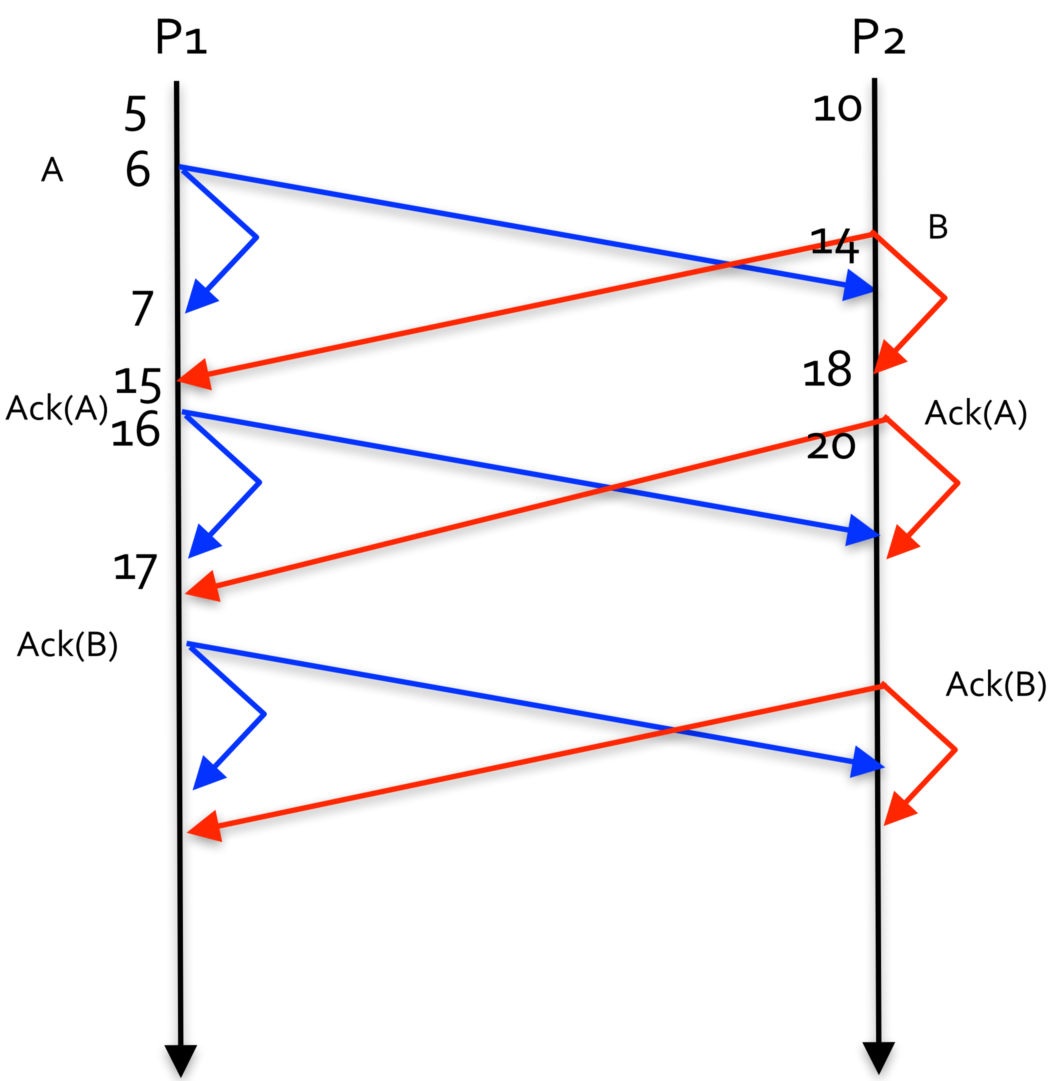
- Clocks on different machines may run at different rates.
- The times for the events need to follow “happens before” condition.



Totally-Ordered Multicasting using Lamport's Timestamp

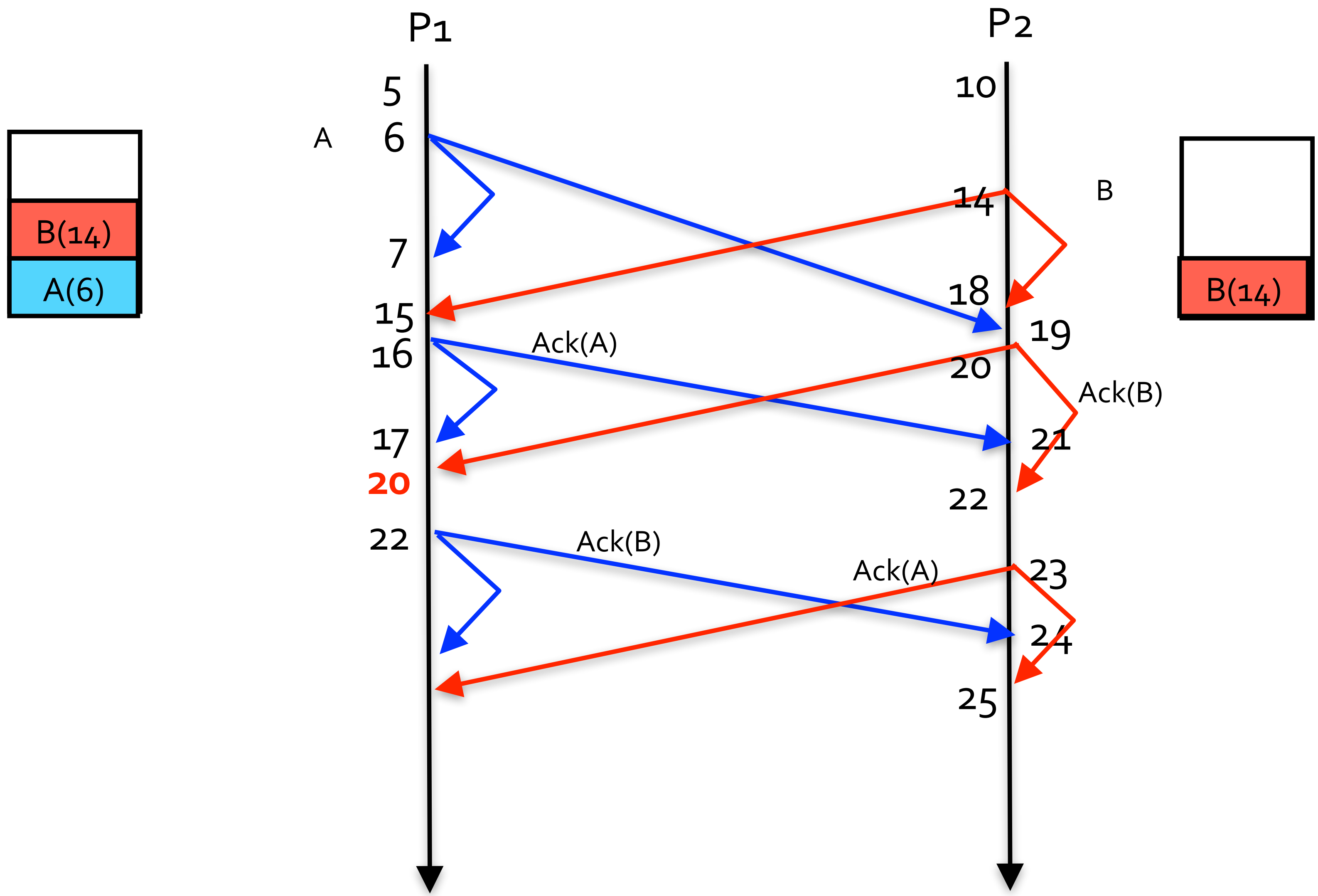


Totally-Ordered Multicasting using Lamport's Timestamp

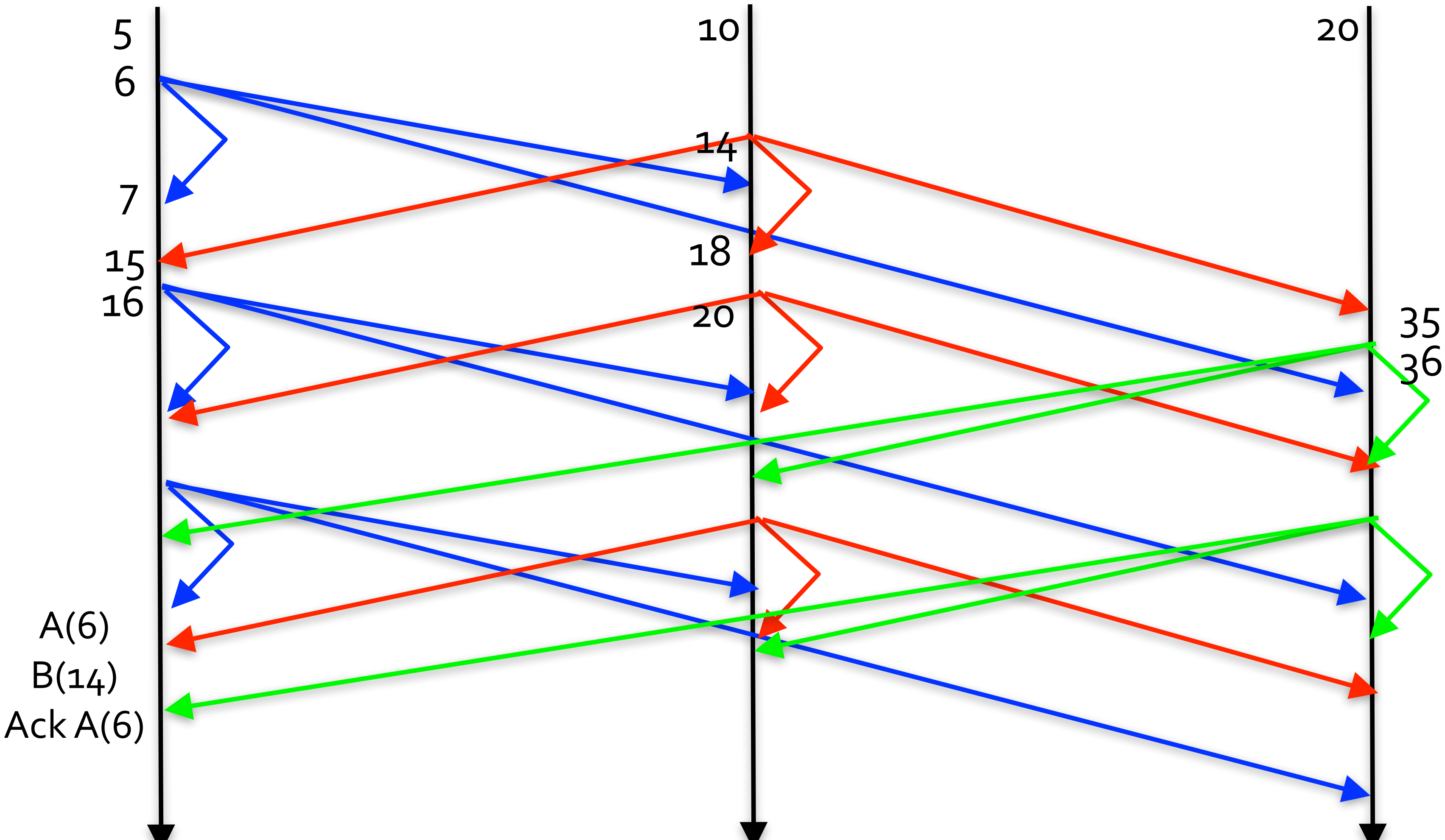


A: Update 1

B: Update 2


B(14)
A(6)
P1
P2
5
10
A
6
14
B
7
18
15
19
16
Ack(A)
20
Ack(B)
17
21
20
22
22
Ack(B)
Ack(A)
23
24
25
B(14)

Totally-Ordered Multicasting using Lamport's Timestamp



Discussion

- When message A and B are independent messages,
  - is it necessary to enforce the totally-ordered multicasting?
- Lamport's happen before defines the order of the events according to the logic clock timestamps.
  - When event ***a*** is the sending event and event ***b*** is the receiving event, it is evident that ***a*** happens before ***b***. Hence, C(***a***) < C(***b***).
  - When C(***a***) < C(***b***), is it necessary that ***a*** happens before ***b***?
  - What if event ***a*** and ***b*** have no ordering relationship?
    - C(***a***) < C(***b***) does not imply that event ***a*** occurs before event ***b***.
    - No causality in Lamport's timestamp.

Vector Timestamps

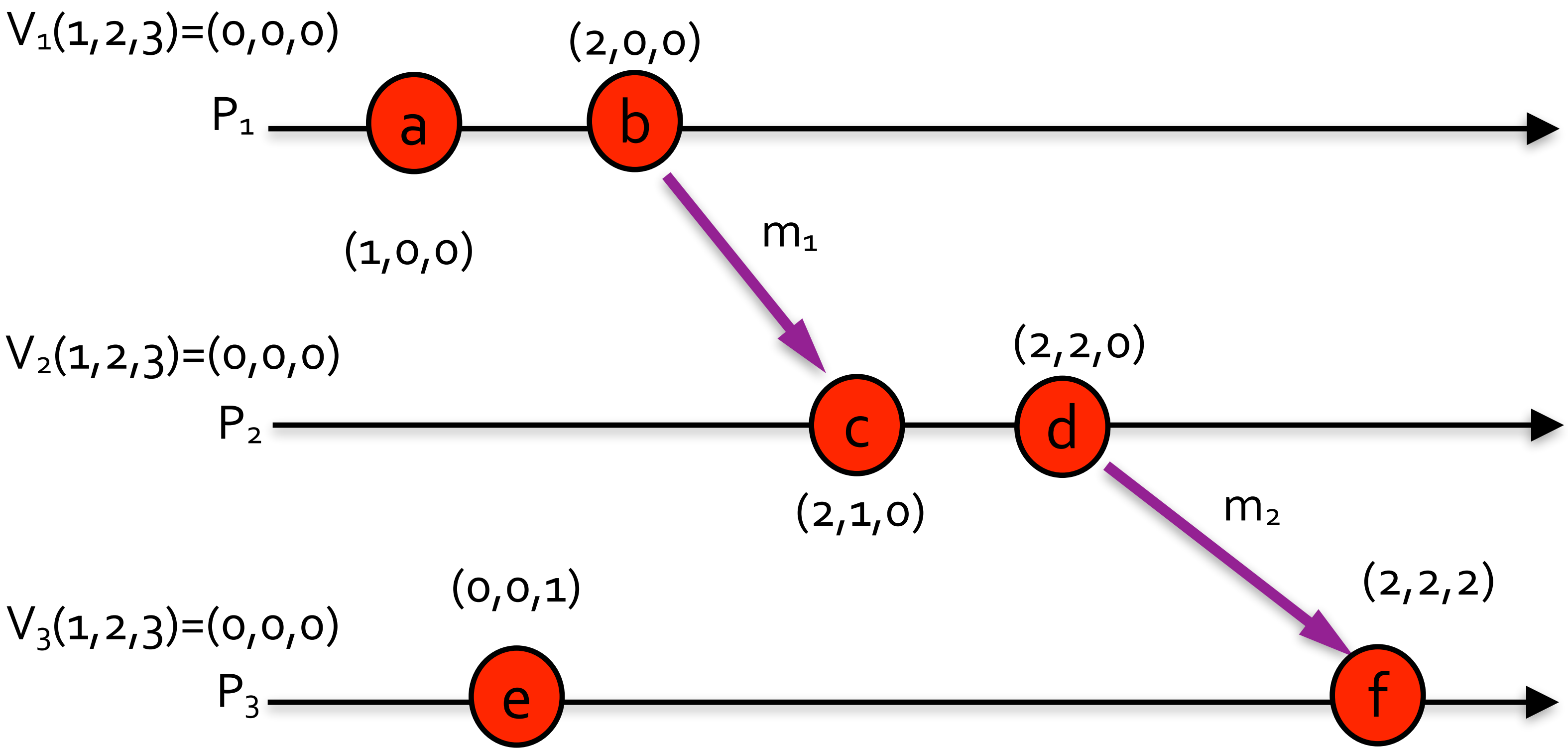
- Each process maintains a vector V_i with the following properties:
  - $V_i[i]$ is the number of events that have occurred so far at P_i.
  - If $V_i[j] = k$ then P_i knows that k events have occurred at P_j.

- Vector Timestamps (VT) assigned to an event ***a*** has the property if VT(***a***) < VT(***b***) for some event b, then event ***a*** is known to causally precede event ***b***.
  - **Def**: VT(***a***) < VT(***b***) if $VT_a[i] <= VT_b[i]$ for all i and $VT_a[j] < VT_b[j]$ for one j.

Vector Logical Clocks

- Vector logical clocks guarantees the following:
  - Each host uses a vector of counters, i-th element is the event(clock) value for host P_i, initially all zeros.
  - Each host P_i, increments the i-th element of its vector upon an event, and assigns the vector to the event.
  - A send (message) event on P_i carries its vector timestamps.
  - For a receive (message) event on P_j,
    - If k is not j, $V_j [k] = Max(V_j[k], V_r[k])$
    - Otherwise, $V_j[k] = V_j[k] + 1$

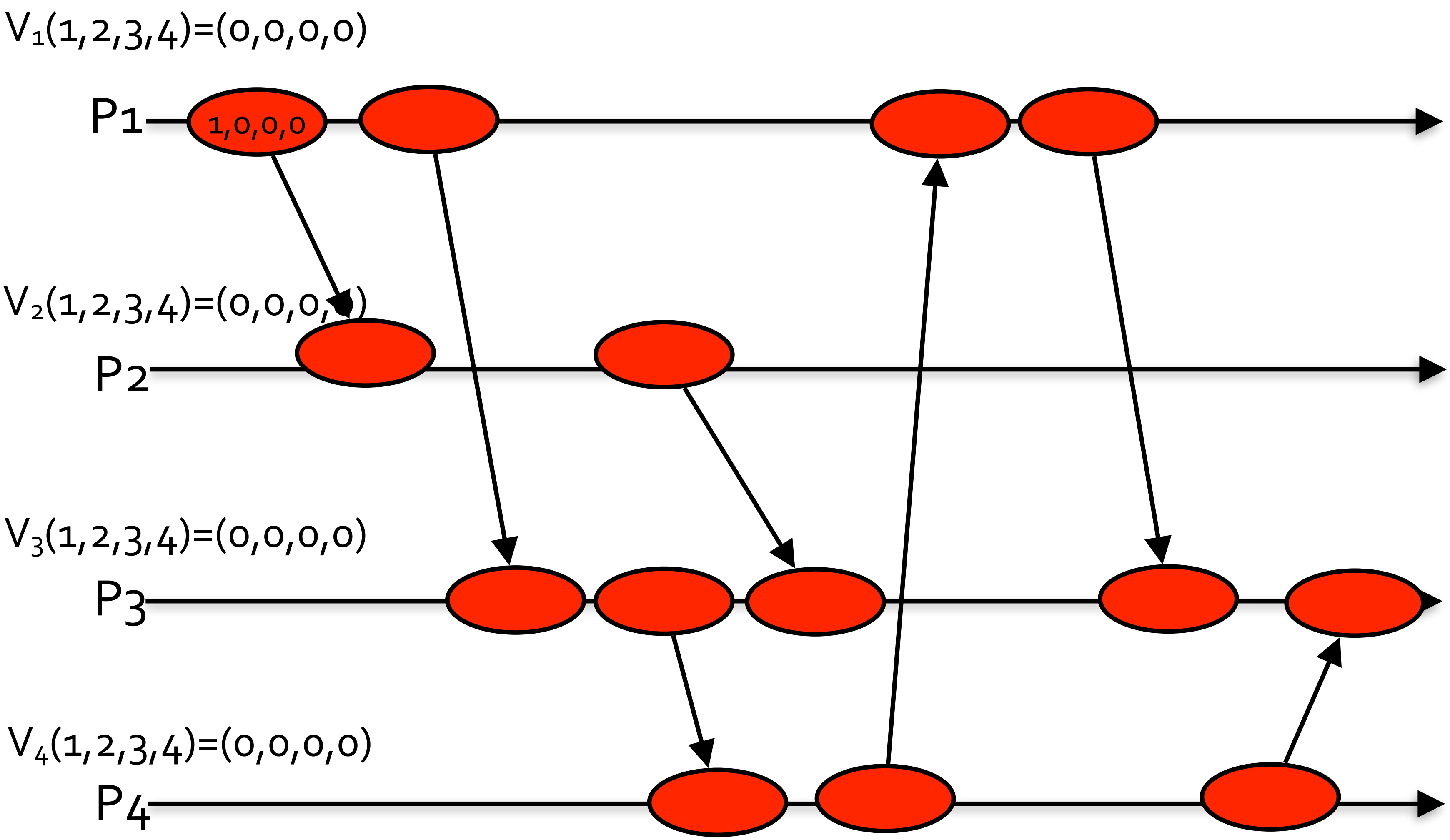
Vector TimeStamp Example



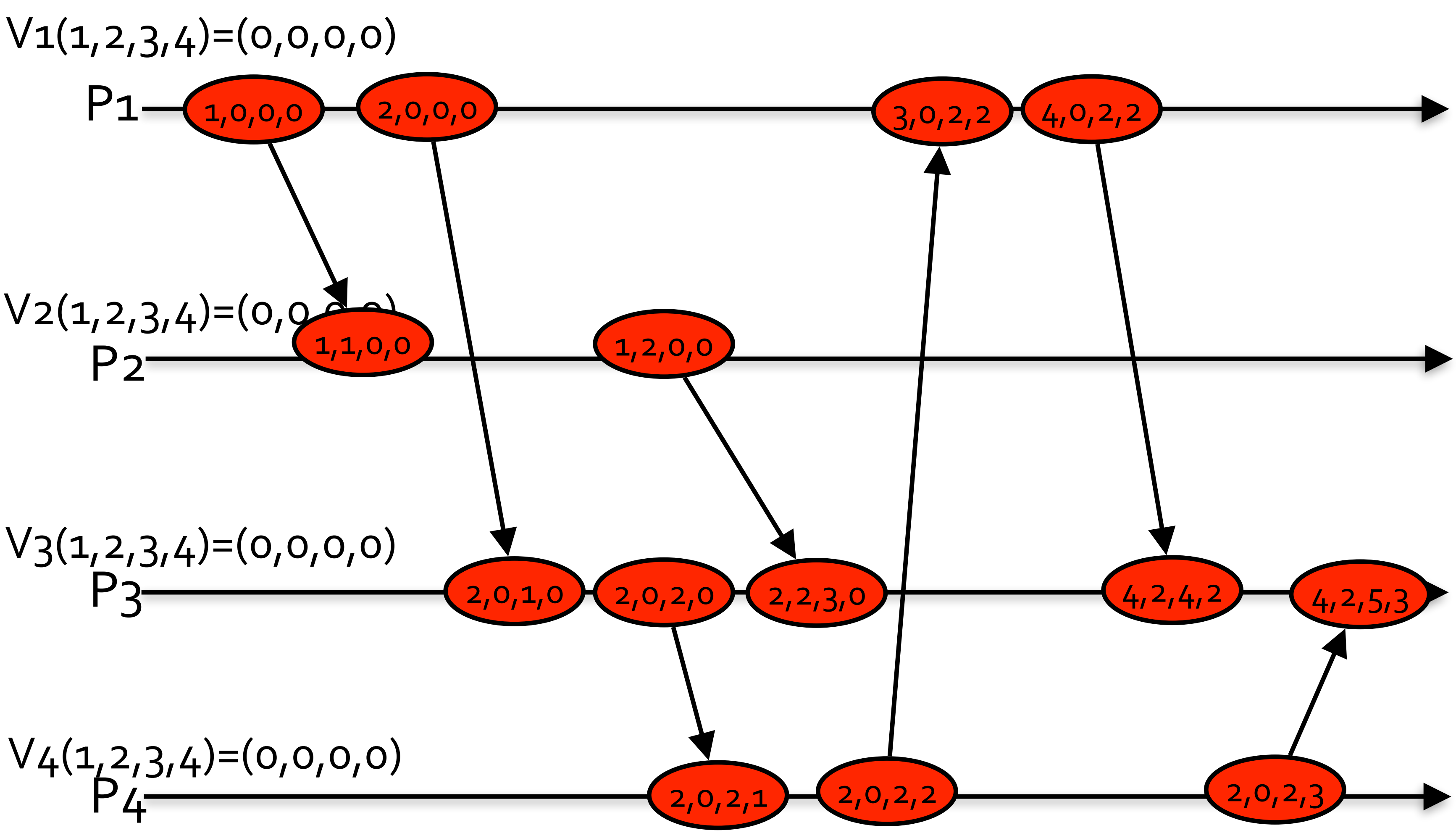
There is no causality between message e and f. But, vector timestamp guarantees that $V_e < V_f$.

What about a and f?

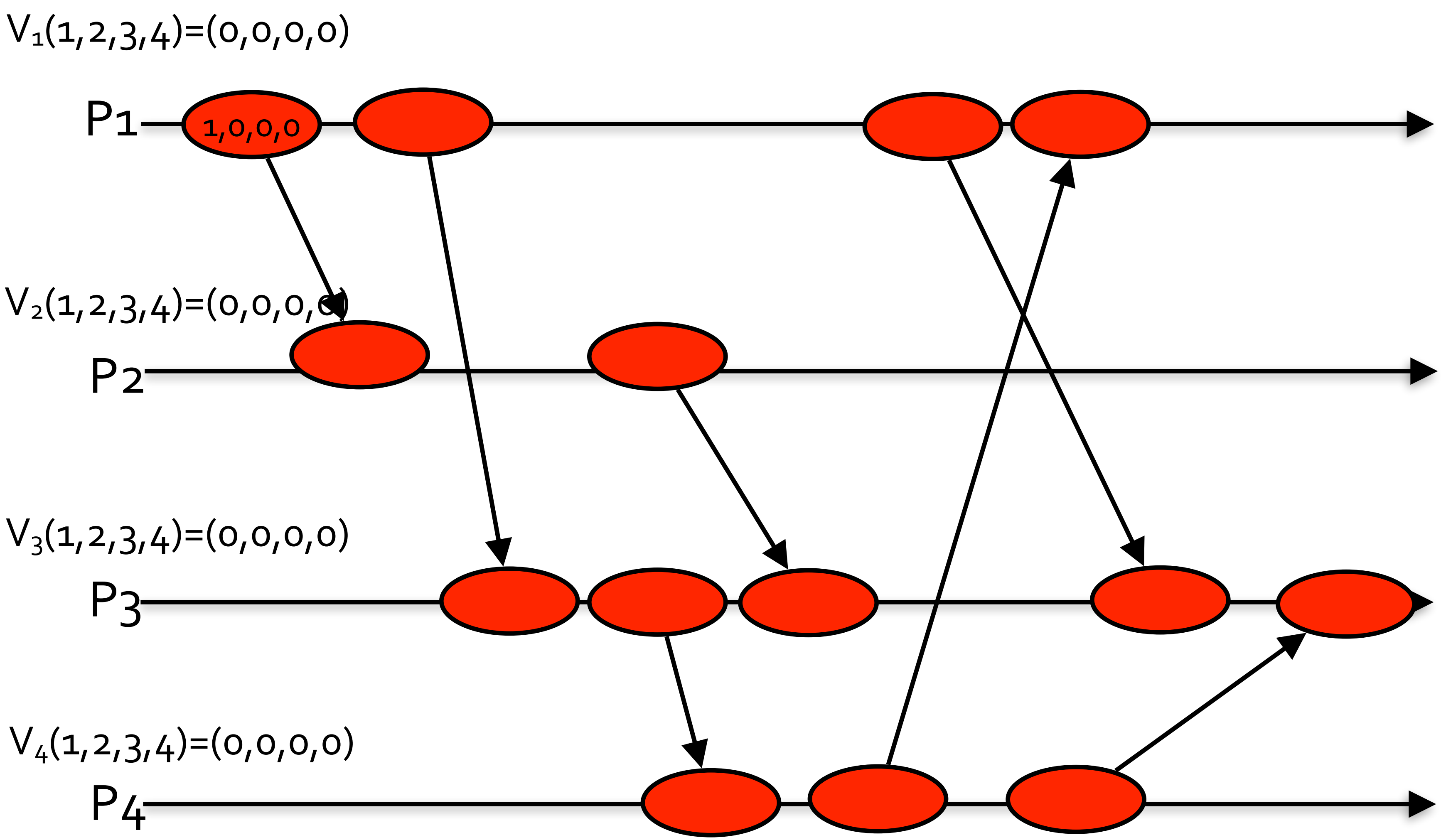
Fill the vector timestamps for all the events



Vector TimeStamp Example



Fill the vector timestamps for all the events (Practice in class)



Multicast Casual Messages

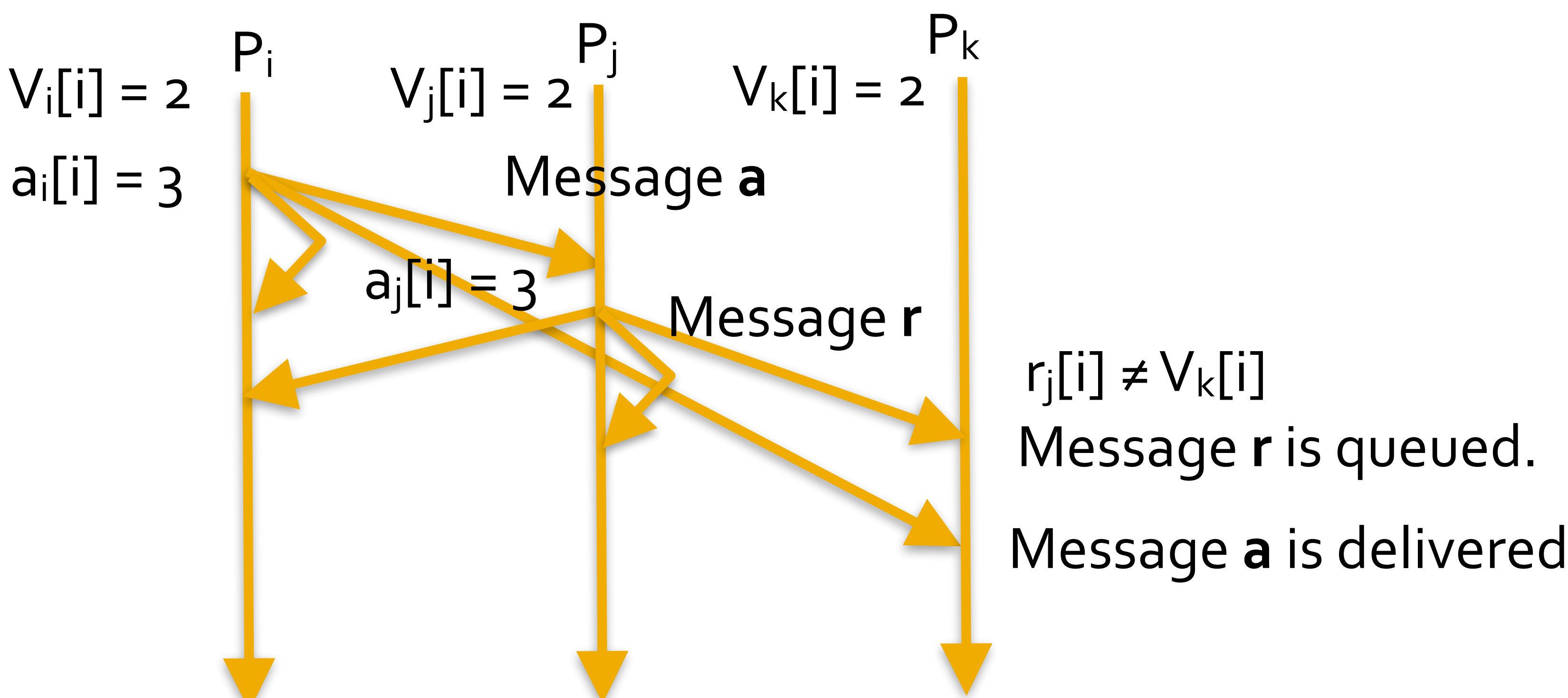
- We now know the causality of messages on uni-cast applications.
- How about applications requiring multi-casting?
  - Considering posting messages on FB:
    - Alice posts a message on her wall.
    - Bob posts a reply to Alice's message.
    - Carlie posts a reply to Bob's reply.
  - How to send the message and replies to their friends?
    - Let FB's servers collect all the messages and broadcast to the friends.
    - Is there a distributed mechanism?

Multicast Casual Messages

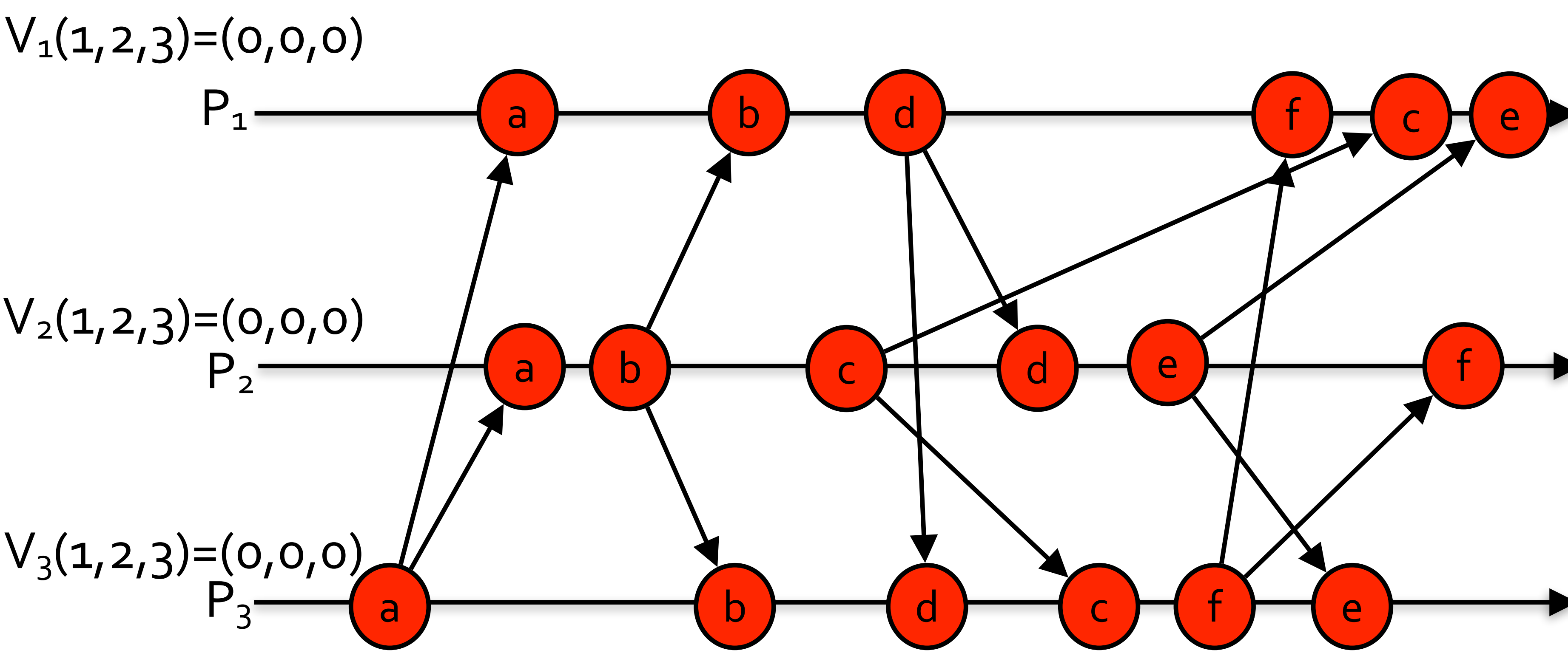
- Can Vector-Timestamp approach be extended for multi-cast messages?
  - We need to assure that all the processes in the group receive the message.
  - The order of message delivery from different processes can be out of order.
    - For same process, the order of message delivery should be in the sending order.

Causality by vector timestamps

- Increment vector only on sending to guarantee casual message delivery.
- Message r is delivered (from P_j to P_k) only if
  - $vt(r)[j]=V_k[j] + 1$: r is the next message that P_k was expecting from process Pj.
  - $vt(r)[i] \leq V_k[i]$ for all $i \neq j$: P_k has not seen any messages that were not seen by P_j when it sent message r.



Example for causality using vector timestamp for msg multicast

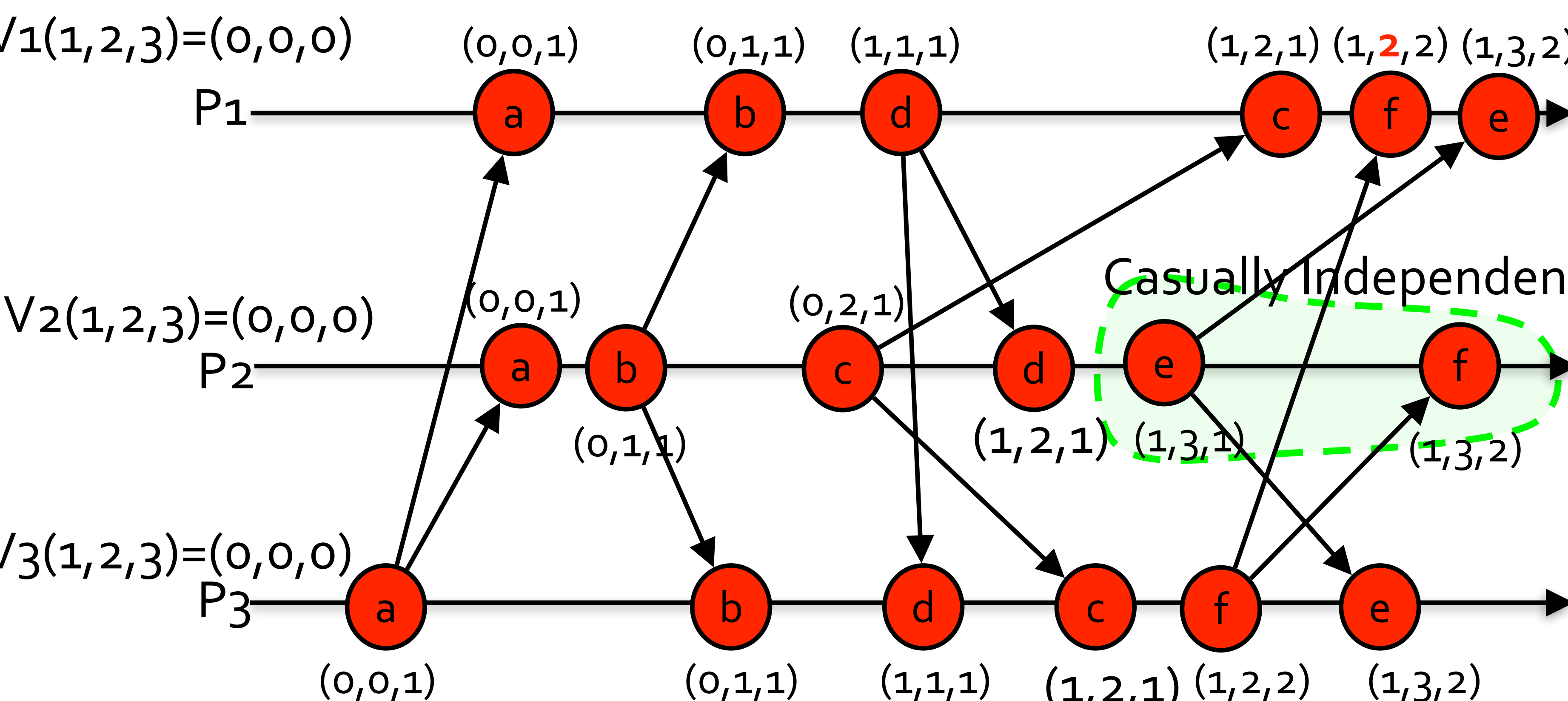


What's the message order on P3?

What's the message order on P1?

Example for causality using vector timestamp

Event f will be held until receiving e.



What's the message order on P3? a, b, d, c, f, e

What's the message order on P1? a, b, d, c, f, e

Global State

- The global state consists of
  - the local state of each process and
  - the messages that are currently in transit, that is, that have been sent but not delivered.
- Global state can be used to detect certain system states such as deadlock and normal termination of a protocol.

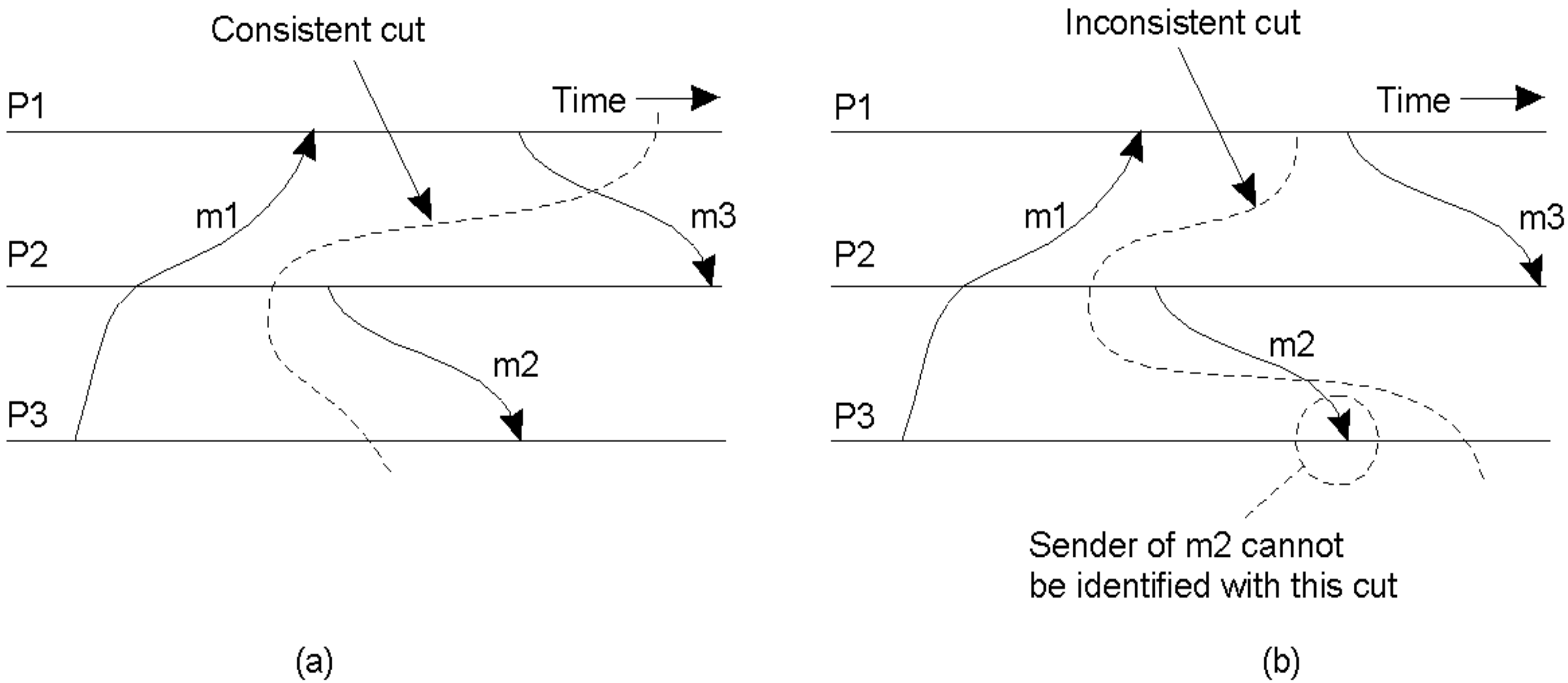
consistent global state

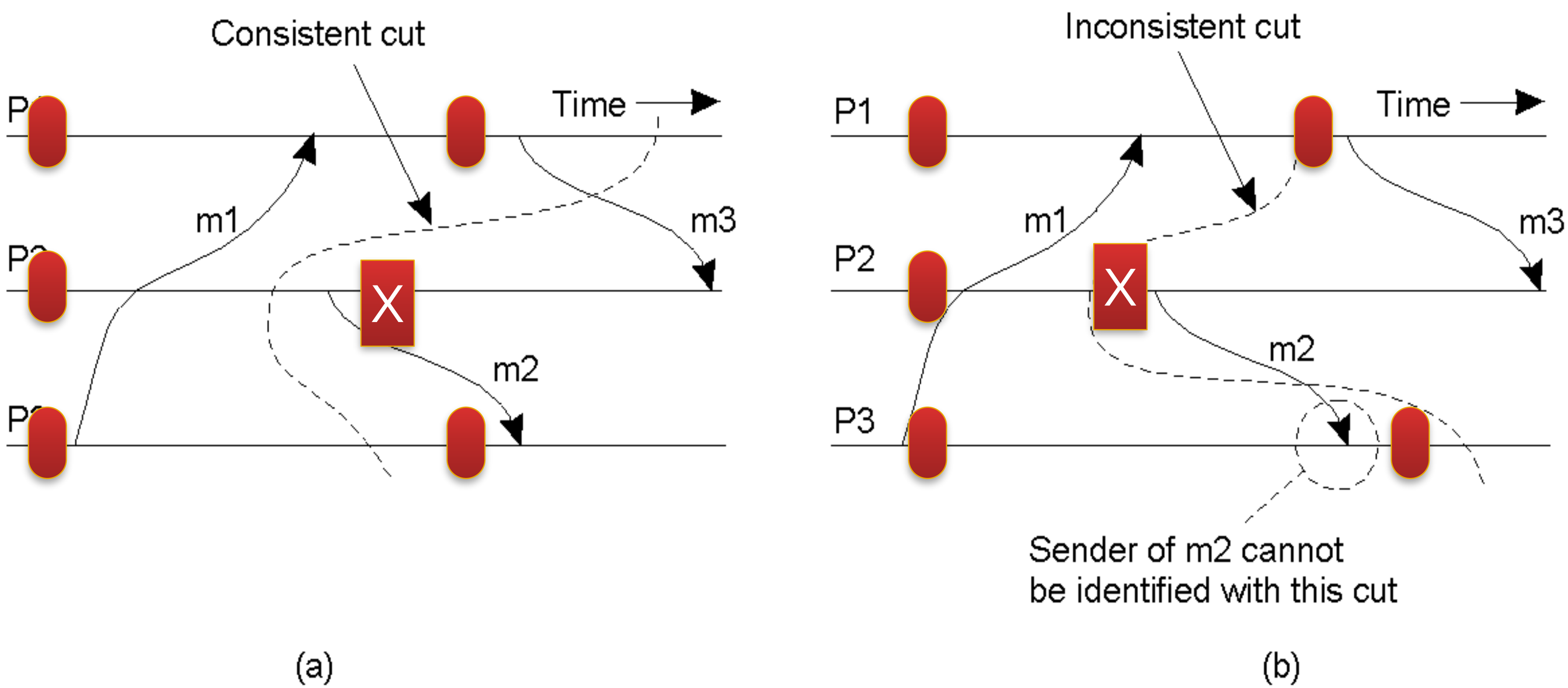
- A cut C is consistent if for all events e and e’

$$\left(e \in C\right) \wedge \left(e' \rightarrow e\right) \Rightarrow e' \in C$$

- Intuitively if an event is part of a cut then all events that happened before it must also be part of the cut
- A consistent cut defines a consistent global state.

Global Snapshot - cut





Recorded state on each processor

X Processor crashes

Channel State Approach

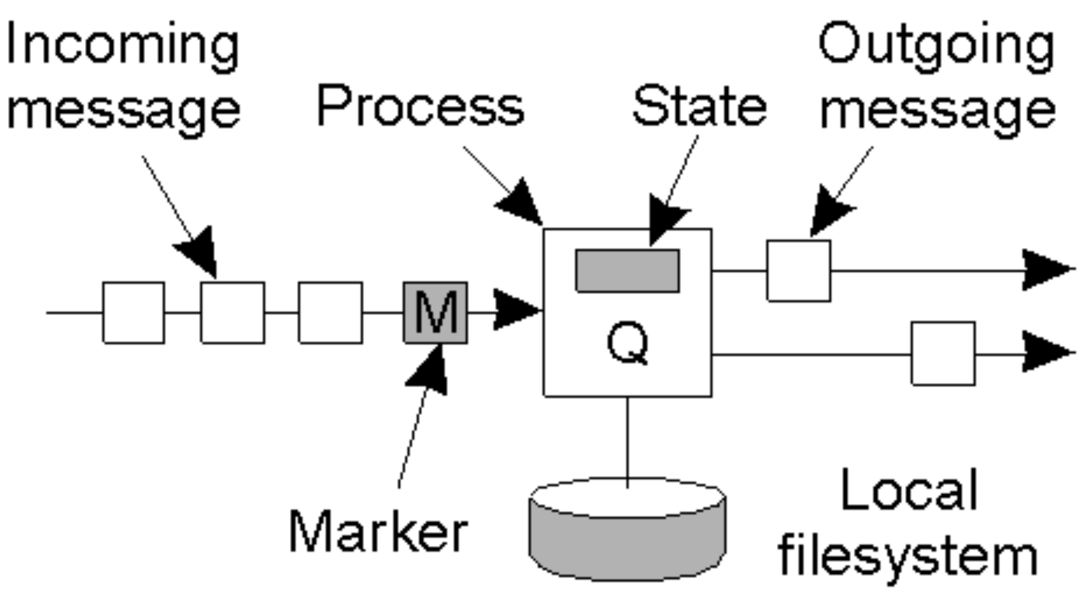
- Each processor records its own state: number of events on the processor.
  - The recored state is periodic written to a server.
- When a processor fails and tries to recover, it tries to recover from its local logged state and works with other processors to know the latest global state.
- Solution here requires
  - additional storage (number of messages)
  - <u>additional computation at recovery time</u> (involving replaying original execution to capture messages sent but not received)

Distributed snapshot by Chaney and Lamport

- It reflects a state in which the distributed system might have been.
- It should reflect the consistent global state. For example, a receiving event cannot exist without a sending event.
- Chaney-Lamport Algorithm (1985):
  - Initiator:
    - Save its local state
    - Send marker tokens on all outgoing edges
  - All other processes:
    - On receiving the first marker on any incoming edges,
      - Save state, and propagate markers on all outgoing edges
      - Resume execution, but also save incoming messages until a marker arrives through the channel
  - Guarantees a consistent global state!

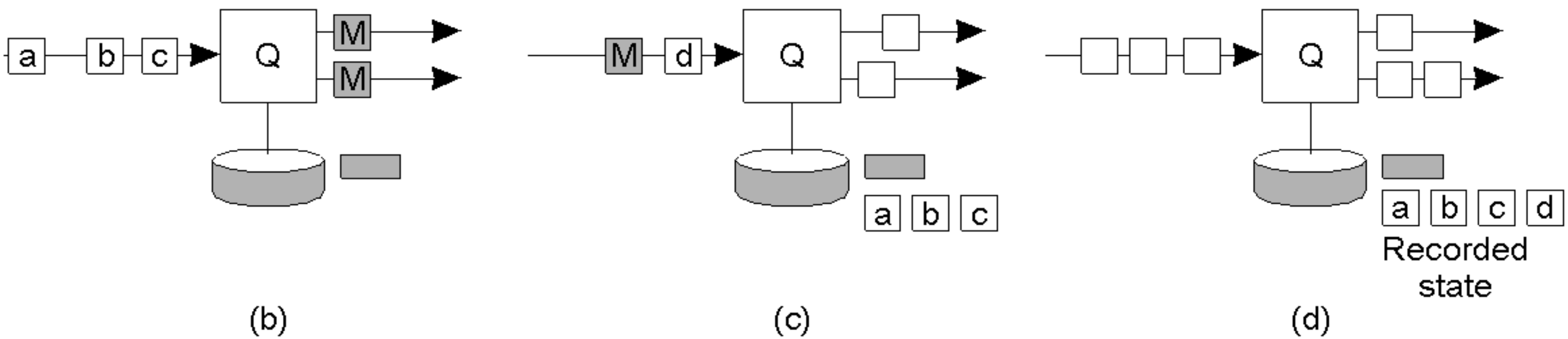
Recording Global State - distributed method.

a) Organization of a process and channels for a distributed snapshot



(a)

Recording Global State

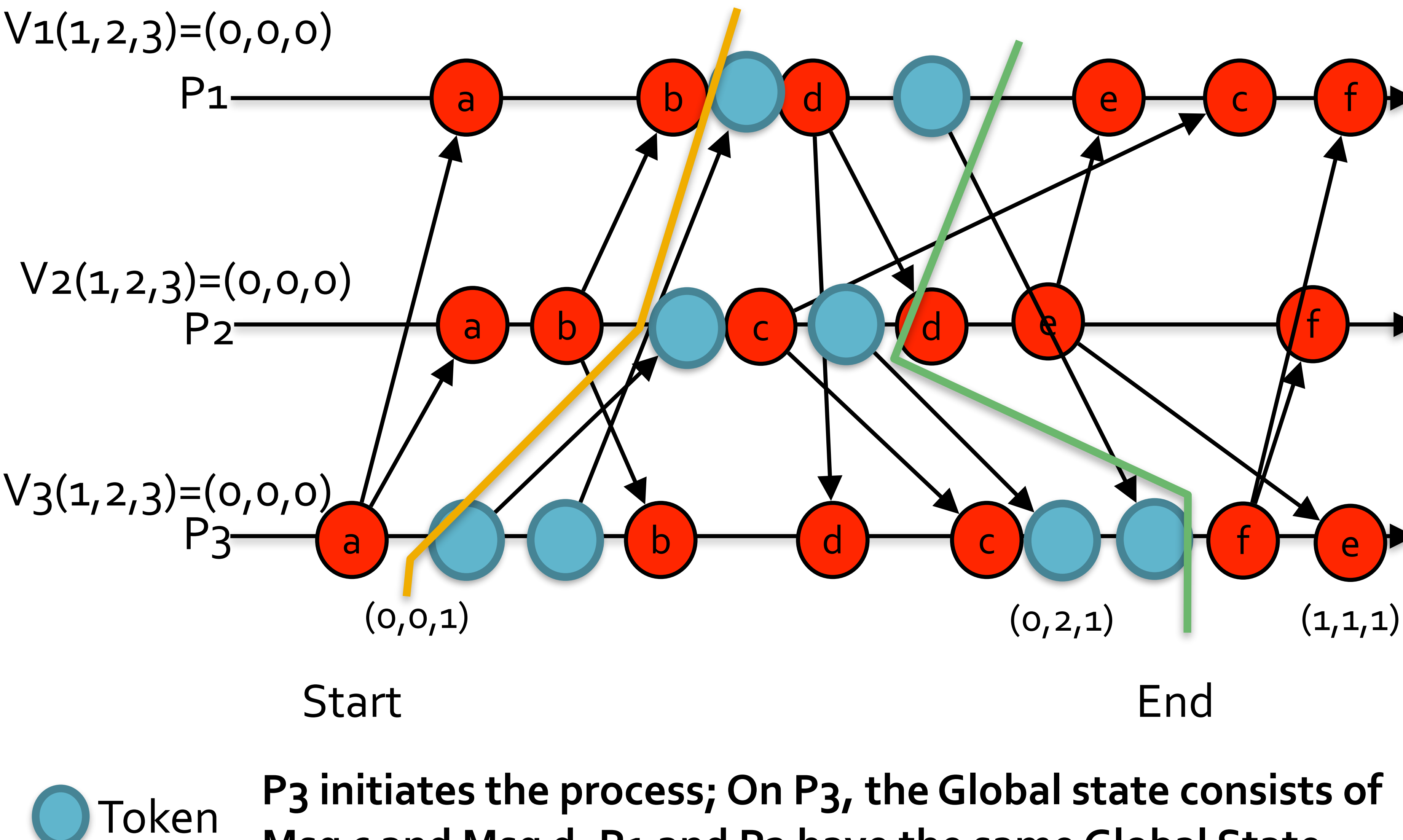


Process Q keeps its local state and starts to record the state of each channel.

Process Q records all the messages at its incoming channels.

Process Q obtains its cut when the markers are received from all the incoming channels.

Distributed snapshot by Chandy and Lamport (1985)

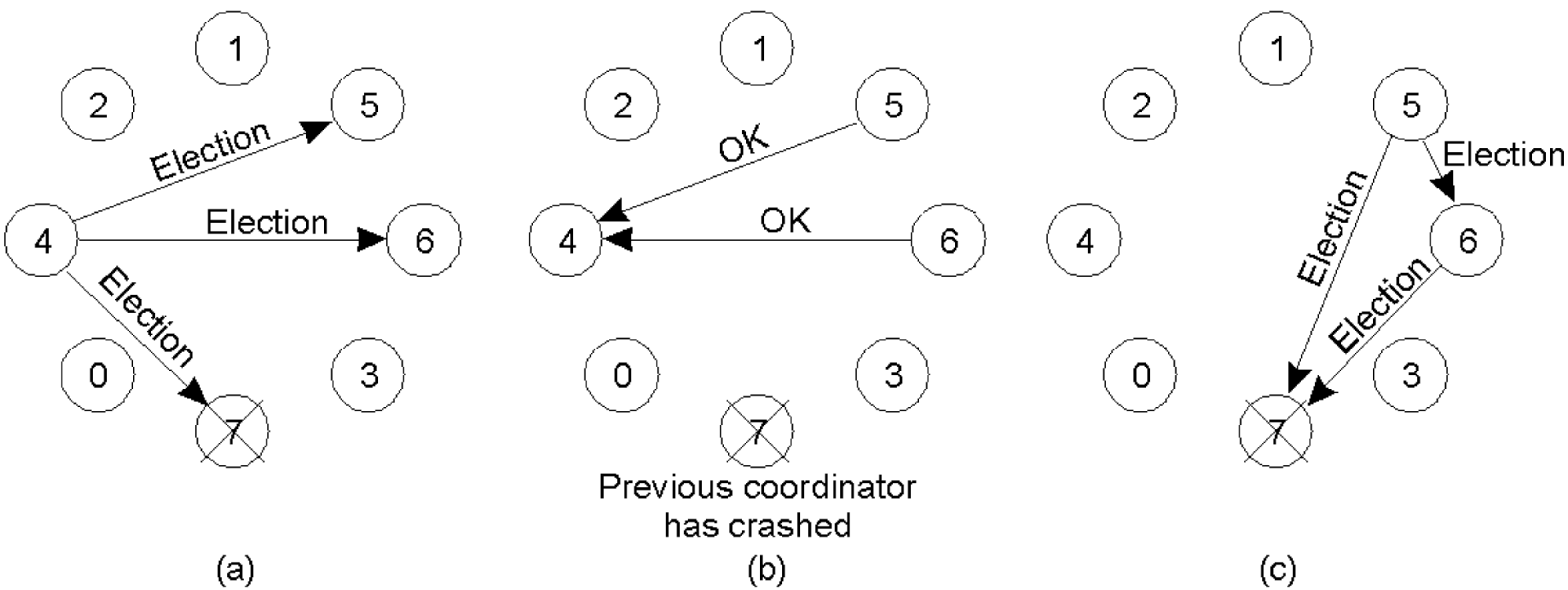


P3 initiates the process; On P3, the Global state consists of Msg c and Msg d. P1 and P2 have the same Global State.

Election Algorithms

- We may need a coordinator in a distributed system. For instance, to lock a variable.
- It is assumed that every process knows the process number of every other process.
  - However, it is not known that which ones are currently up and which ones are currently down.
- The goal is to ensure that when an election starts, it concludes with all processes agreeing on who the new coordinator is.

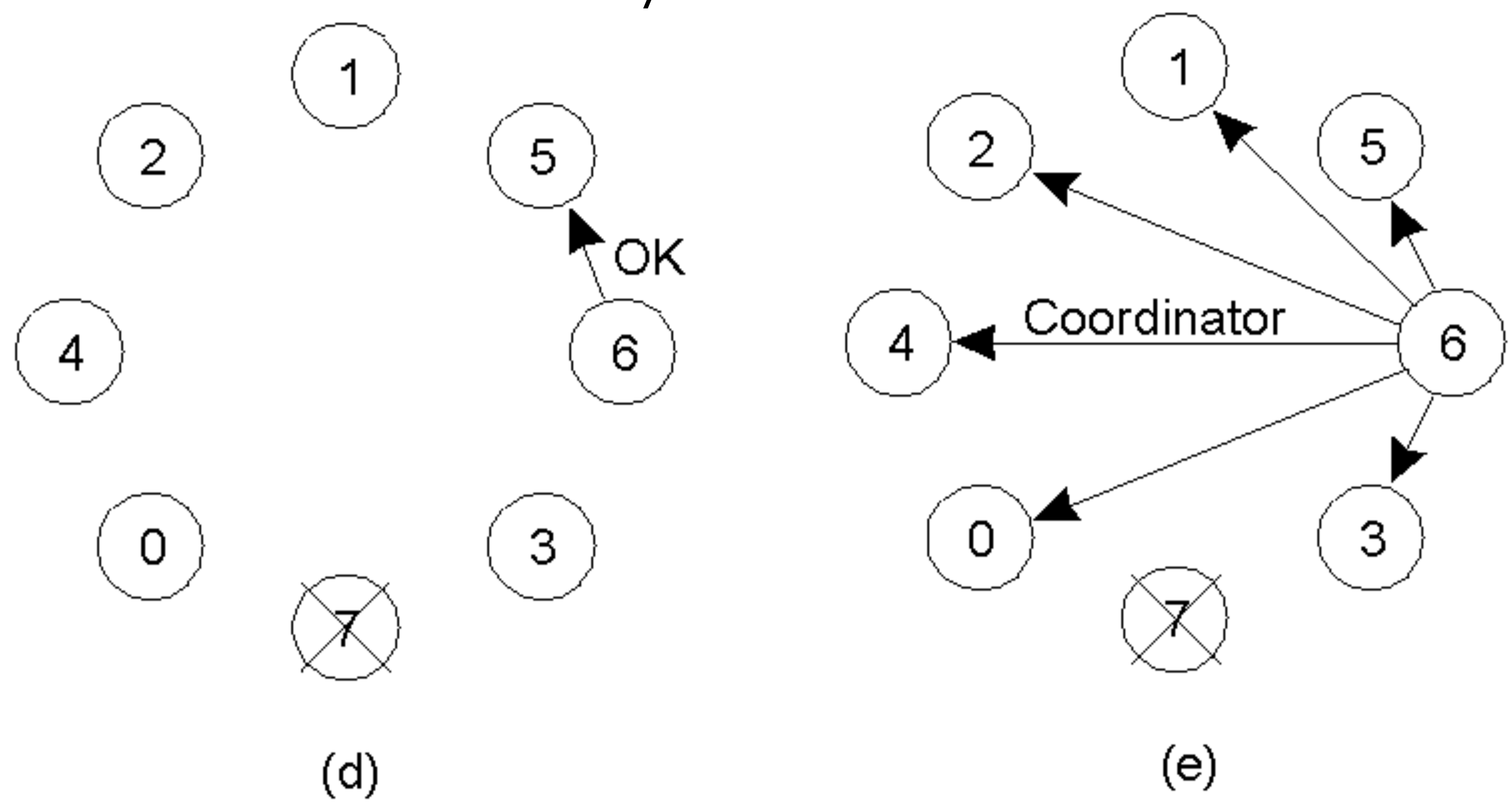
The Bully Algorithm



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

Bully Algorithm

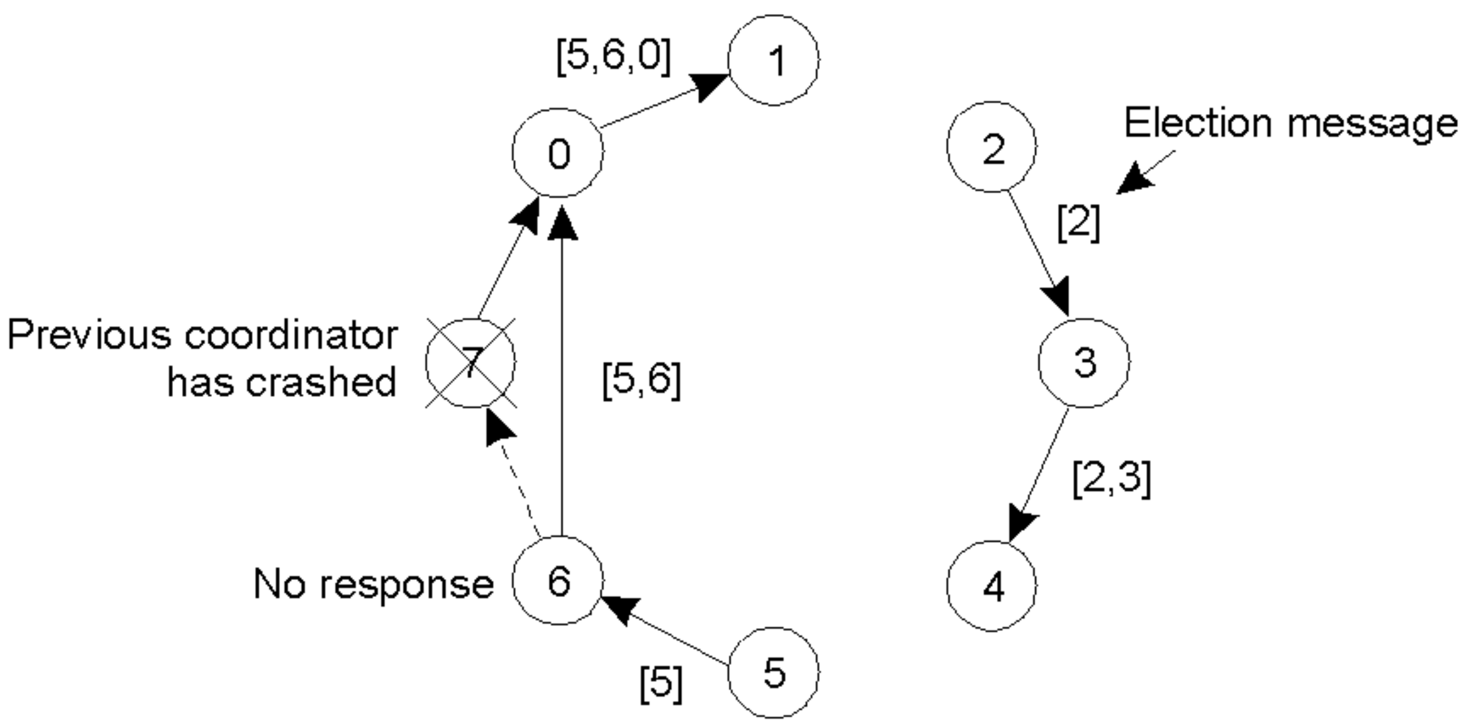
d) Process 6 tells 5 to stop
e) Process 6 wins and tells everyone



- What's the limits of this algorithm? **The communication among live nodes must be robust.**

A Ring Algorithm

- Election algorithm using a ring but not token.



- **The requirements on communication only apply the neighboring nodes on the ring.**

Mutual Exclusion for Distributed Systems

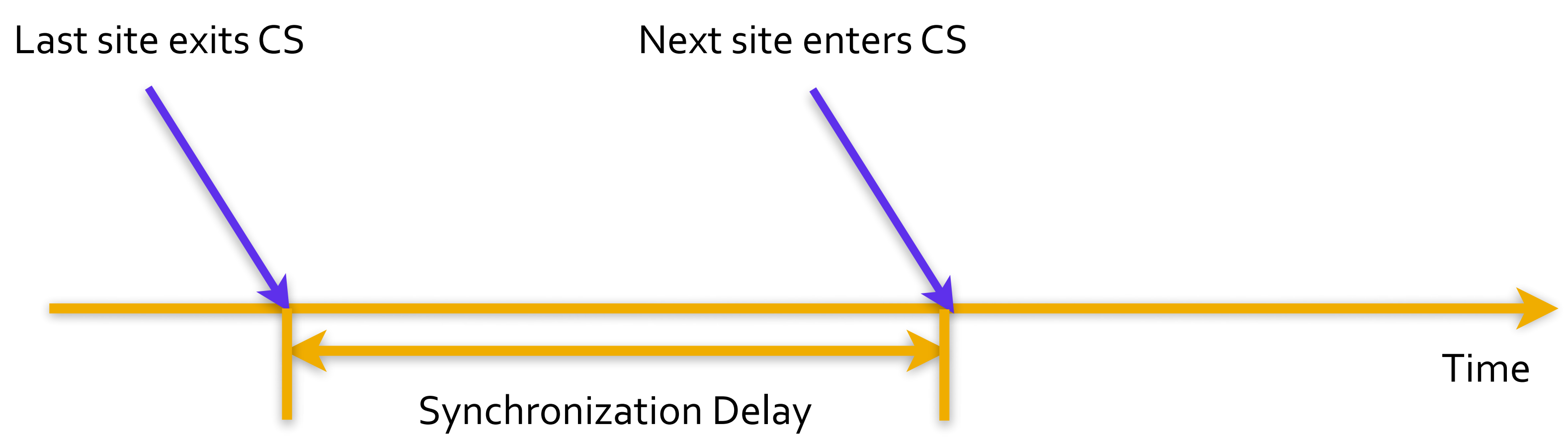
- Mutual exclusion: Concurrent access of processes to a shared resource or data is executed in mutually exclusive manner.
  - Only one process is allowed to execute the critical section (CS) at any given time.
  - In a distributed system, shared variables (semaphores) or a local kernel **cannot** be used to implement mutual exclusion.
- Message passing is the sole means for implementing distributed mutual exclusion.

Requirements

- Requirements of Mutual Exclusion Algorithms
  - **Safety Property**: At any instant, only one process can execute the critical section.
  - **Liveness Property**: This property states the absence of deadlock and starvation. Two or more sites should not endlessly wait for messages which will never arrive.
  - **Fairness**: Each process receives a fair chance to execute the CS. Fairness property generally means the CS execution requests are executed in the order of their arrival (time is determined by a logical clock) in the system.

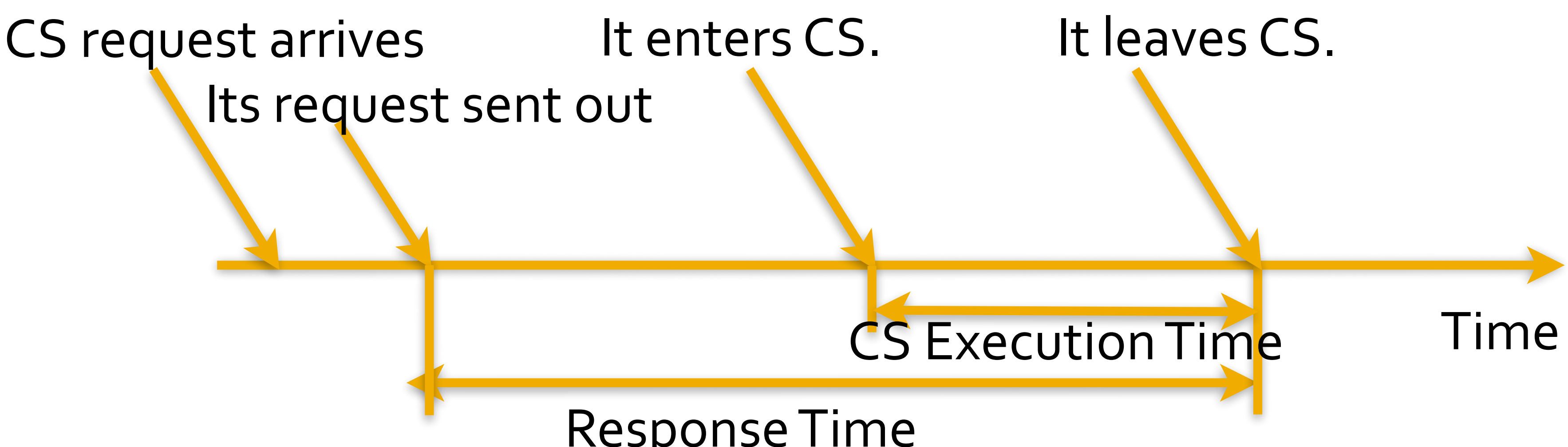
Performance Metrics

- The performance is generally measured by the following four metrics:
  - **Message complexity**: The number of messages required per CS execution by a site.
  - **Synchronization delay**: After a site leaves the CS, it is the time required and before the next site enters the CS.



Performance Metrics

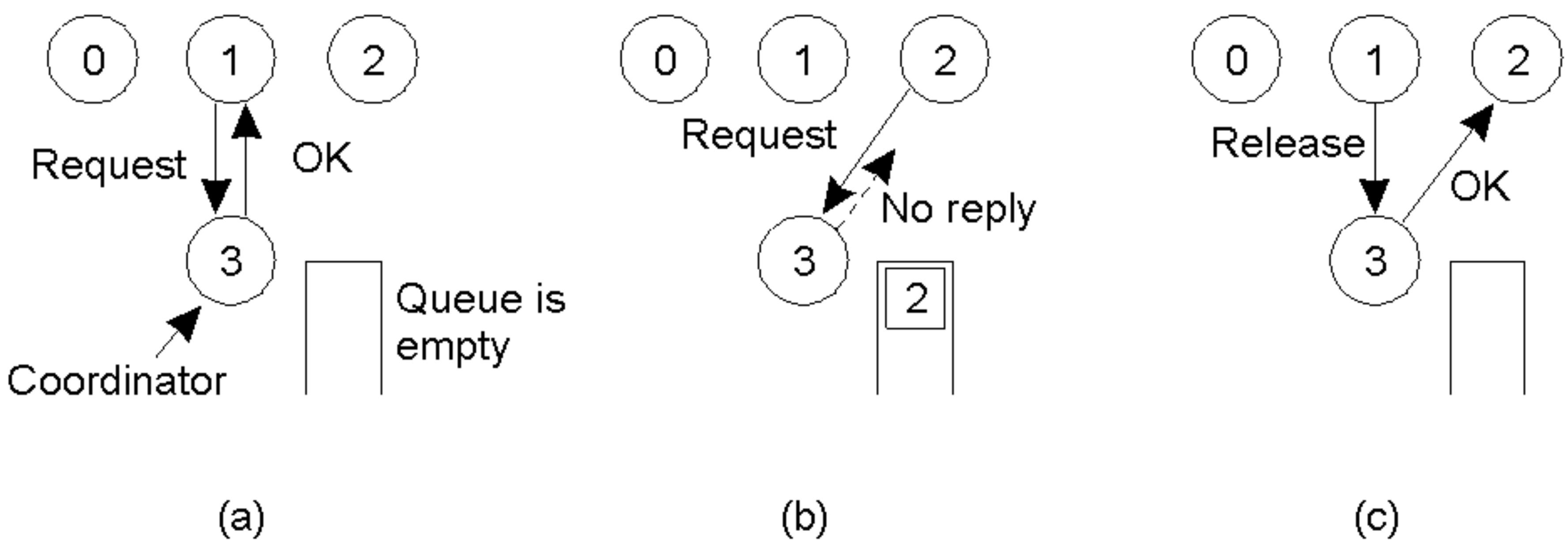
- **Response time**: The time interval a request waits for its CS execution to be over after its request messages have been sent out.



- **System throughput**: The rate at which the system executes requests for the CS.
  - system throughput=1/(SD+E)
  - where SD is the synchronization delay and E is the average critical section execution time.

Mutual Exclusion: A Centralized Algorithm

a) Process 1 asks the coordinator for permission to enter a critical region. Permission is granted
b) Process 2 then asks permission to enter the same critical region. The coordinator does not reply.
c) When process 1 exits the critical region, it tells the coordinator, when then replies to 2



Discussion

- What are the disadvantage of centralized approach?

- Requirements of distributed approach:
  - No single point failure.
  - No global knowledge of the requests
  - Avoid any point failure.
- How to design a distributed approach?

Distributed Mutual Exclusion Algorithms

- Distributed mutual exclusion algorithms must deal with unpredictable message delays and incomplete knowledge of the system state.
- Three basic approaches for distributed mutual exclusion:
  - Token based approach
  - Non-token based approach
  - Quorum based approach

Lamport's Distributed Mutual Exclusive Algorithm

- Requests for CS are executed in the increasing order of timestamps and time is determined by logical clocks.
- Every site S_i keeps a queue, $request\_queue_i$, which contains mutual exclusion requests ordered by their timestamps.
- This algorithm requires communication channels to deliver messages the FIFO order.

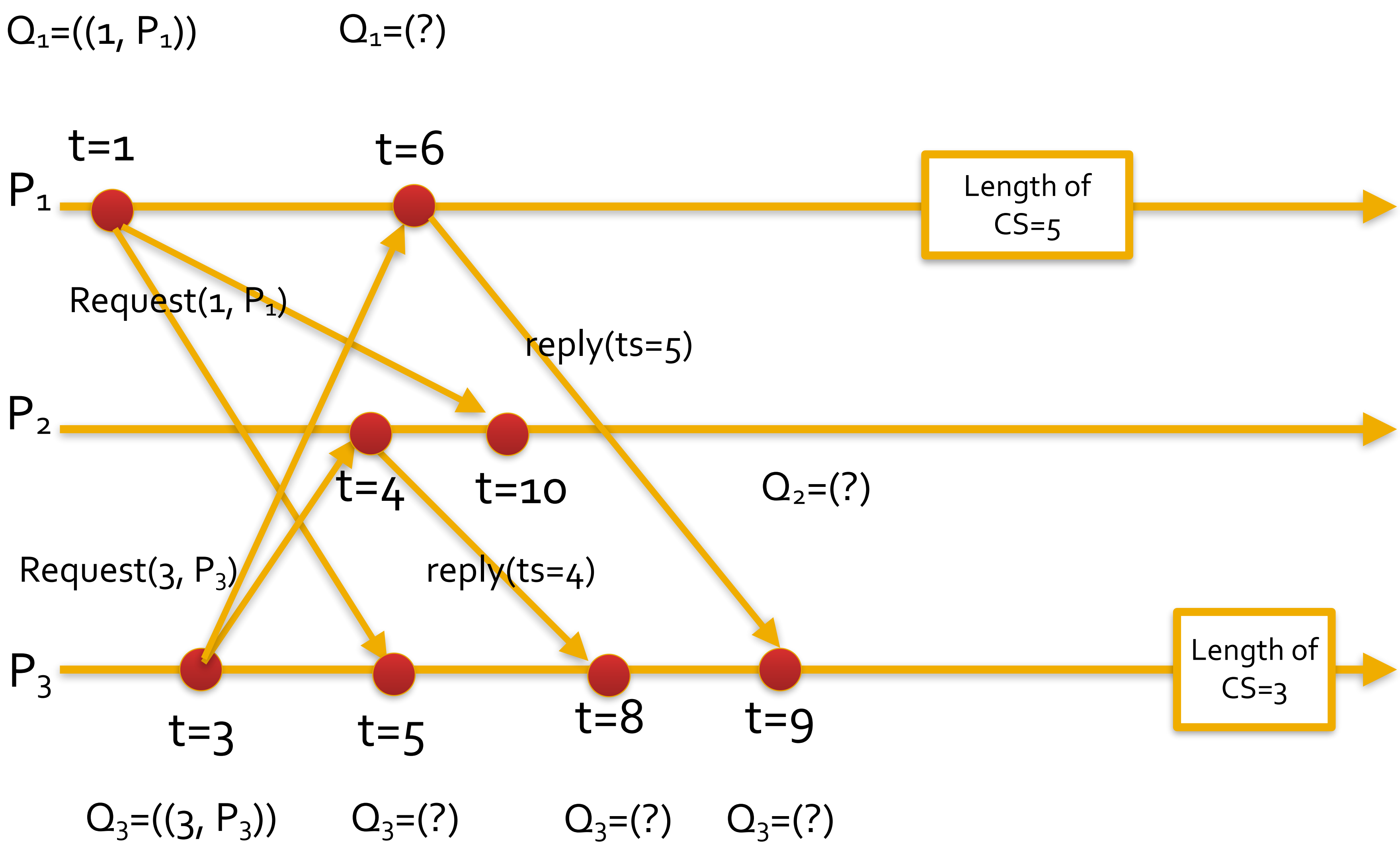
Algorithm: Requesting the critical section

- Requesting the critical section:
  - When a site S_i wants to enter the CS, it broadcasts a REQUEST(ts_i, i) message to all other sites and places the request on *request_queue*$_i$. ((ts_i, i) denotes the timestamp of the request.)
  - When a site S_j receives the REQUEST(ts_i,i) message from site S_i, places site S_i 's request on *request_queue*$_j$ and it returns a timestamped REPLY message to S_i .
  - Executing the critical section: Site S_i enters the CS when the following two conditions hold:
    - L1: S_i has received a REPLY message with timestamp larger than (ts_i, i) from <u>all other sites</u>.
    - L2: S_i 's request is at the top of *request_queue*$_i$.

Algorithm: Releasing the critical section

- Site S_i , upon exiting the CS, removes its request from the top of its request queue and broadcasts a timestamped RELEASE message to all other sites.
- When a site S_j receives a RELEASE message from site S_i , it removes S_i 's request from its request queue.
- When a site removes a request from its request queue, its own request may come at the top of the queue, enabling it to enter the CS.

Example



Q1=((1, P1))
Q1=(?)
t=1
t=6
P1
Length of CS=5
Request(1, P1)
reply(ts=5)
P2
t=4
t=10
Q2=(?)
Request(3, P3)
reply(ts=4)
P3
Length of CS=3
t=3
t=5
t=8
t=9
Q3=((3, P3))
Q3=(?)
Q3=(?)
Q3=(?)

Safety

Theorem: Lamport's algorithm achieves mutual exclusion.

Proof:

- Proof is by contradiction. Suppose two sites S_i and S_j are executing the CS concurrently. For this to happen conditions L1 and L2 must hold at both the sites concurrently.
- This implies that at some instant in time, say t, both S_i and S_j have their own requests at the top of their request queues and condition L1 holds at them. Without loss of generality, assume that S_i's request has smaller timestamp than the request of S_j.
- From condition L1 and FIFO property of the communication channels, it is clear that at instant t the request of S_i must be present in request_queue$_j$ when S_j was executing its CS. This implies that S_j 's own request is at the top of its own request queue when a smaller timestamp request, S_i 's request, is present in the request_queue$_j$ – a contradiction!

Fairness

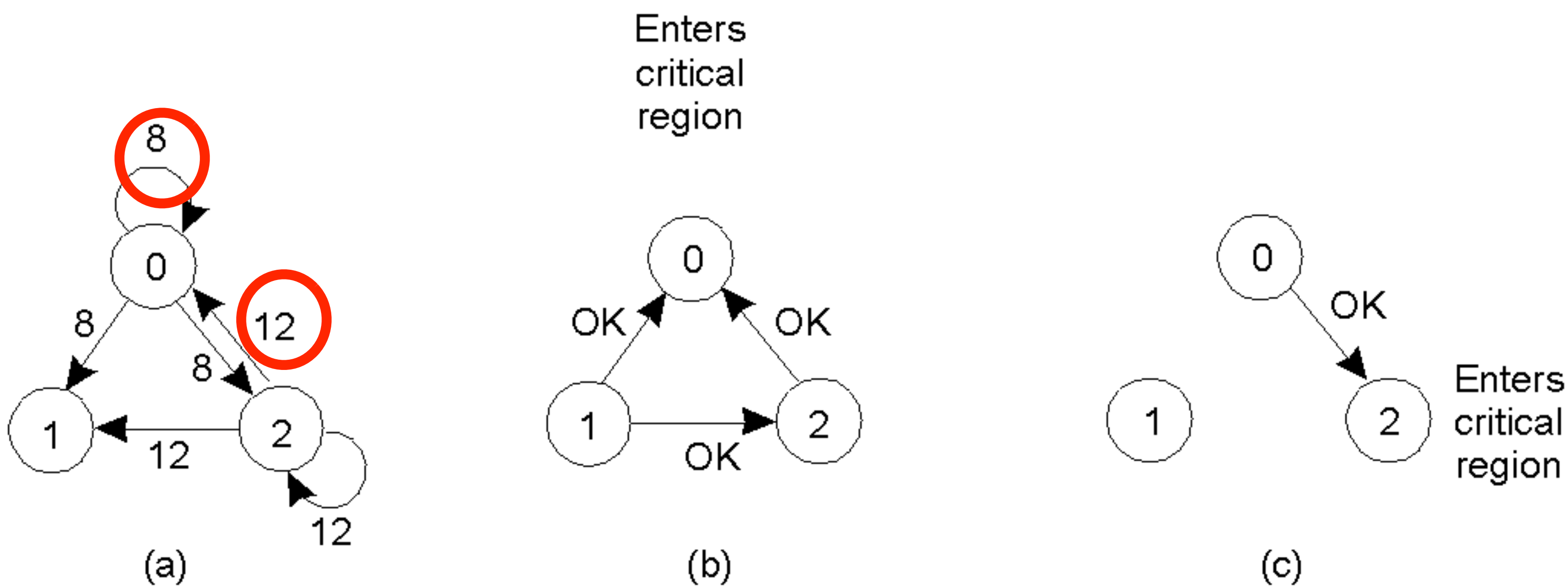
Theorem: Lamport's algorithm is fair.

Proof:

- The proof is by contradiction. Suppose a site S_i 's request has a smaller timestamp than the request of another site S_j and S_j is able to execute the CS before S_i .
- For S_j to execute the CS, it has to satisfy the conditions L1 and L2. This implies that at some instant in time say t, S_j has its own request at the top of its queue and it has also received a message with timestamp larger than the timestamp of its request from all other sites.
- But request queue at a site is ordered by timestamp, and according to our assumption S_i has lower timestamp. So S_i 's request must be placed ahead of the S_j 's request in the $request\_queue_j$. This is a contradiction!

Ricart-Agrawala Algorithm - A Distributed Algorithm

This algorithm is an extension and optimization of Lamport's Distributed Mutual Exclusion Algorithm, by removing the need for RELEASE messages.



Three potential problems of Lamport's algorithm:

- Compared to centralized approach, single point of failure has been replaced by n points of failure.
- A group communication protocol is needed to know which process is in the group.
- Bottleneck problem is not solved.

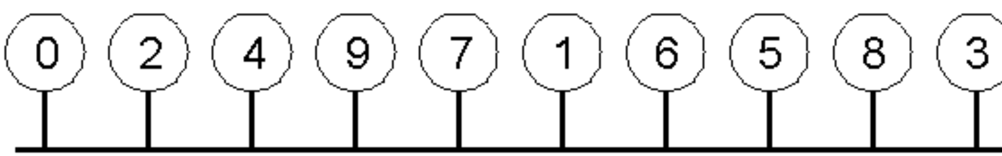
Distributed Mutual Exclusion Algorithm - Ricart–Agrawala algorithm

- The requester builds a message containing the name of the critical region, its process number, and the current time.
- Two cases:
  - The receiver returns OK if
    - the receiver does not want to enter the critical region or
    - the receiver has a request with higher timestamp.
  - The receiver returns nothing and queues the message if
    - it is in the critical region or
    - it wants to enter the critical region and has the lower time stamp.

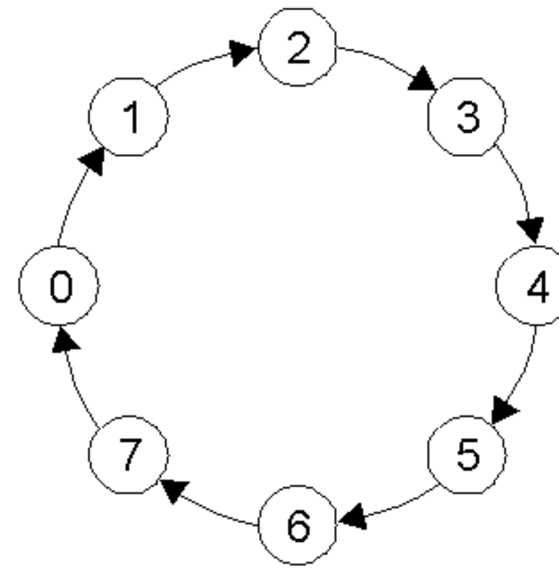
Distributed Mutual Exclusion Algorithm

- Lamport's algorithm:
  - Suffers from multiple point of failure
  - # of messages: 3(N-1) per request
- Ricart–Agrawala algorithm
  - Suffers from multiple point of failure
  - # of messages: 2(N-1) per request

A Token Ring Algorithm



(a)



(b)

- Token-ring Algorithm:
  - A process can enter the critical region only if it holds the token.
  - A token can only be used for one critical region.
  - The token is passed immediately if not interested in entering the critical region.
- Problems:
  a) The token may be lost.
  b) Some process may crash.

Comparison

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

Can we have a better algorithm which does not suffer point failure and low complexity?

Quorum-based Distributed Mutual Exclusion algorithm

- Rationale
  - Overlapping quorum sets are formed in the distributed systems.
    - Any two sites have at least one common neighbor.
  - Requests are sent to the sites in local quorum set only.
  - Requests are granted when grants are received from all the nodes in the quorum set.

Definitions

- A site is any computing device in the network
- For any request of the critical section:
  - The requesting site is the site which is requesting entry into the critical section.
  - The receiving site is every other site which is receiving the request from the requesting site.
- ts refers to the local timestamp of the system according to its logical clock.

Requesting locks

- A requesting site Pi sends a message request(ts,i) to all sites in its quorum set Ri.
- Site Pi enters the critical section on receiving grant messages from all sites in Ri.
- Upon exiting the critical section, Pi sends a release(i) message to all sites in Ri.

Granting or not granting

- Upon reception of a *request*(*ts*,*i*) message, the receiving site P_j will:
  - If site P_j does not have an outstanding grant message, then site P_j sends a grant(j) message to site P_i.
  - If site P_j has an outstanding grant message with a process with <u>higher priority (earlier timestamp)</u> than the request, then site P_j sends a failed(j) message to site P_i and site P_j queues the request from site P_i.
  - If site P_j has an outstanding grant message with a process with <u>lower priority (later timestamp)</u> than the request, then site P_j sends an inquire(j) message to the process which has currently been granted access to the critical section by site P_j.

Inquire, Yield, and manage the queue

- Upon reception of an inquire(j) message, the site P_k:
  - Send a yield(k) message to site P_j if and only if
    - site P_k has received a failed message from some other site, or
    - P_k has sent a yield to some other site but has not received a new grant.
- Upon reception of a yield(k) message, site P_j will:
  - Send a grant(j) message to the request on the top of its own request queue.
  - Place P_k into its request queue.
- Upon reception of a release(i) message, site P_j will:
  - Delete P_i from its request queue.
  - Send a grant(j) message to the request on the top of its request queue.

Define Quorum Set (Maekawa's Algorithm)

- A quorum set must abide by the following properties:
  - $\forall i \forall j \left[R_i \cap R_j \right] \neq \varnothing$
  - $\forall i \left[P_i \in \cap R_i \right]$
  - $\forall i \left[\left| R_i \right| = K \right]$
  - Site P_i is contained in exactly K quorum sets
- Therefore:
  - N=K(K-1)+1 and $\left| R_i \right| \geq \sqrt{N-1}$

Example for quorum sets

- When N=3,
  - K=2
  - R1={1,2}, R2={1,3}, and R3={2,3}.
- When N=7,
  - K=

Example for quorum sets

- When N=3,
  - K=2
  - R1={1,2}, R2={1,3}, and R3={2,3}.
- When N=7,
  - K=3

R1={1, 2, 3}
R4={1, 4, 5}
R6={1, 6, 7}
R2={2, 4, 6}
R5={2, 5, 7}
R7={3, 4, 7}
R3={3, 5, 6}

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
Quorum-based Algorithm	3 \sqrt{N} to 6 \sqrt{N}	2	

Distributed Transactions

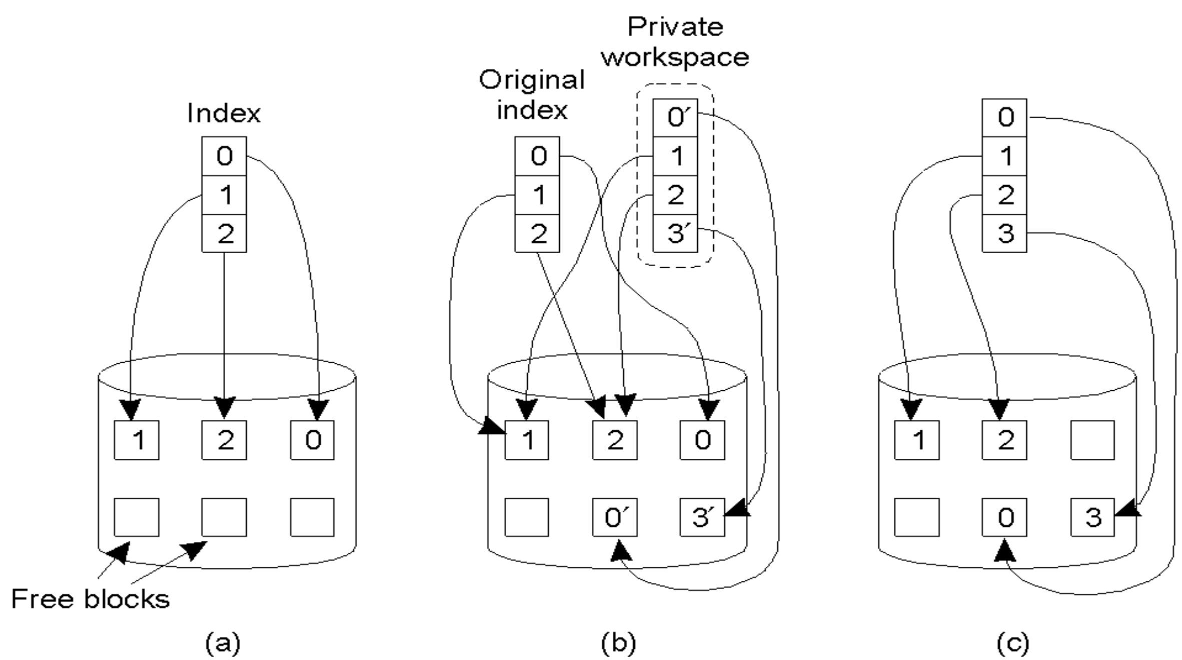
- Transactions are used to protect shared resources.
- ACID properties: Atomic, Consistent, Isolated and Durable.
- One Example:
  - Using e-bank to transfer your money from your checking account to your saving account.
  - What if the connection is broken after the money is withdrew from the checking account but before the money is deposited into your saving account ?

Implementations

- Transactions sound like a good idea, but how to implement them?
- Atomicity and Durability:
  - No failure: private workspace or Writeahead log
  - With failures: we will discuss it later
- Consistency and Isolation: Concurrency Control
  - Serializability
  - Two-Phase Locking

Private Workspace

a) Make a local of the related files
b) Update the files back when the transaction completes



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

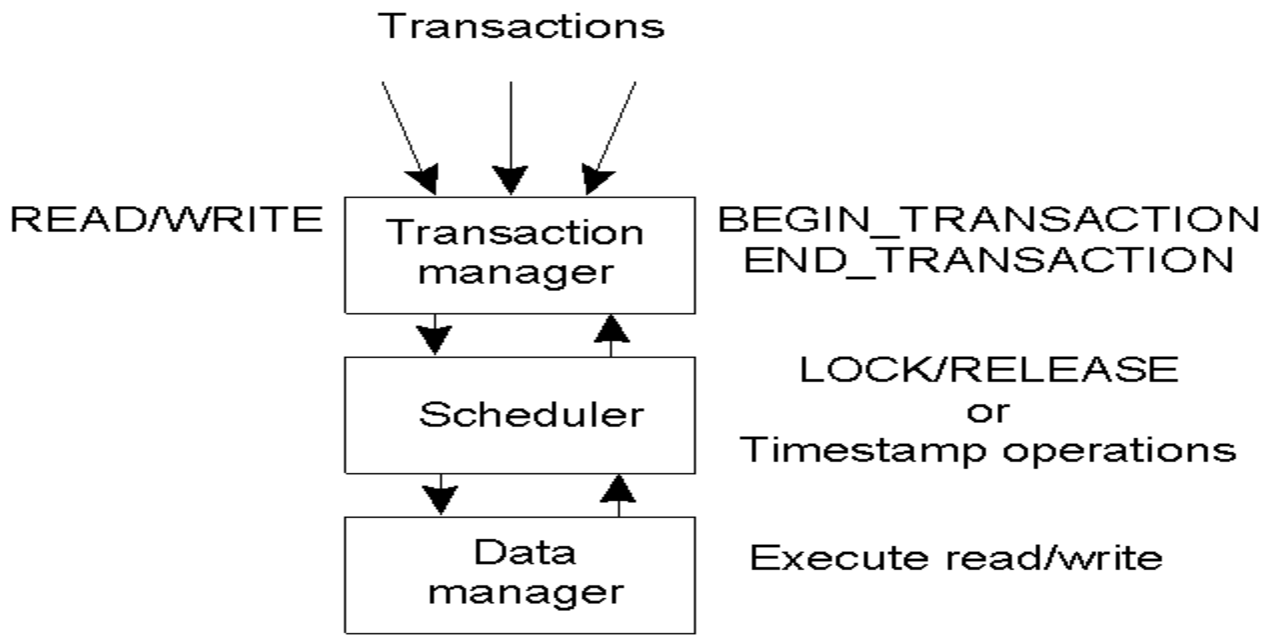
Writeahead Log

a) A transaction
b) – d) The log before each statement is executed

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)*

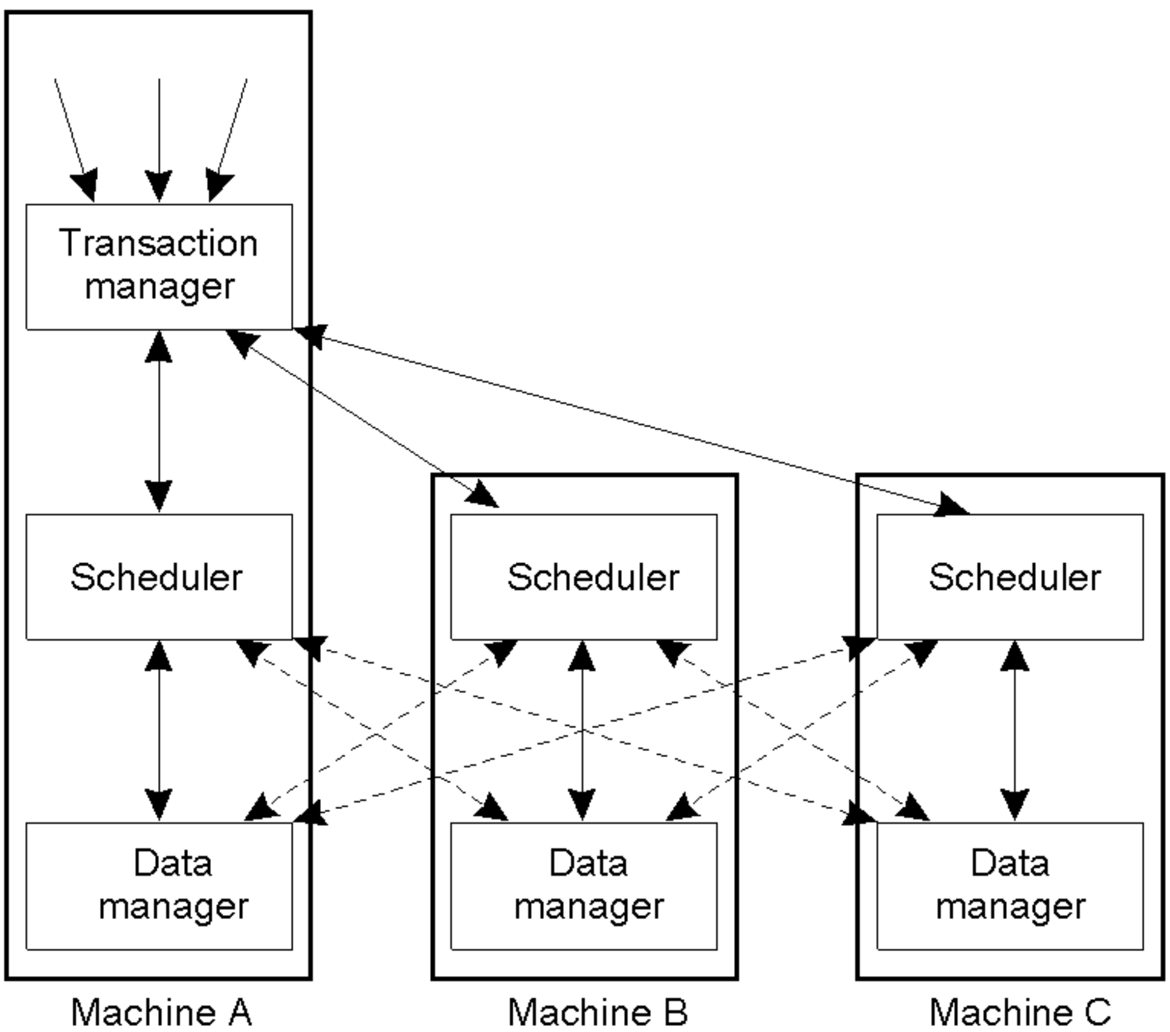
Concurrency Control

- Concurrency control handles the consistency and isolation properties.
- Goal:
  - Allow several transactions to be executed simultaneously.
  - The collection of data items is left in a consistent state.



Concurrency Control for Distributed Systems

- General organization of managers for handling distributed transactions.



Serializability

- Possible schedules

A schedule is **serial** if the actions of the different transactions are not interleaved; they are executed one after another

A schedule is **serializable** if its effect is the same as that of some serial schedule

BEGIN_TRANSACTION	*BEGIN_TRANSACTION*	*BEGIN_TRANSACTION*
x = 0;	*x = 0;*	*x = 0;*
x = x + 1;	*x = x + 2;*	*x = x + 3;*
END_TRANSACTION	*END_TRANSACTION*	*END_TRANSACTION*

Transactions T1, T2, and T3

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;*	*?*

Concurrency Control

- Representation Model:
  - Each write/read operation for variable x by transaction Ti is
    - write(Ti, x); read(Ti, x)
- The challenge is to schedule conflicting operations.
- Two operations conflict if they operate on the same data item, and if at least one of them is a write operation.
  - Read-Write Conflict
  - Write-Write Conflict
- Two types of Concurrency Control: Pessimistic vs. optimistic

Pessimistic Concurrency Control: Two-Phase Locking

If anything will go wrong, it will.

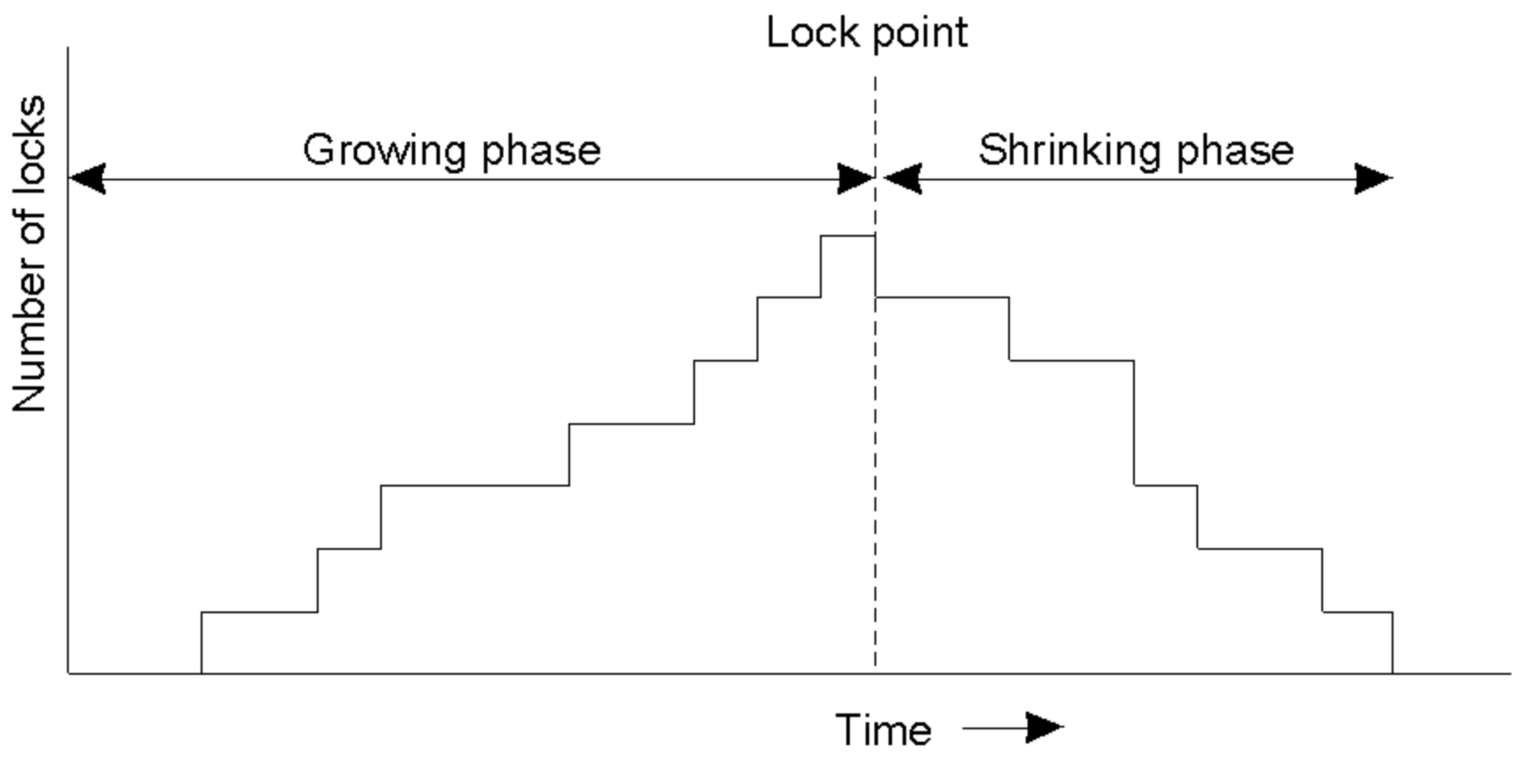
— Murphy's Laws

In nature, nothing is ever right. Therefore, if everything is going right ... something is wrong.

- Lock the data when read/write.
- (Non-Strict) Two-Phase Locking:
  - If a transaction T wants to read/write an object, it must request a shared/exclusive lock on the object.
  - A transaction cannot request additional locks on an object once it releases any lock, and it can release locks at any time.

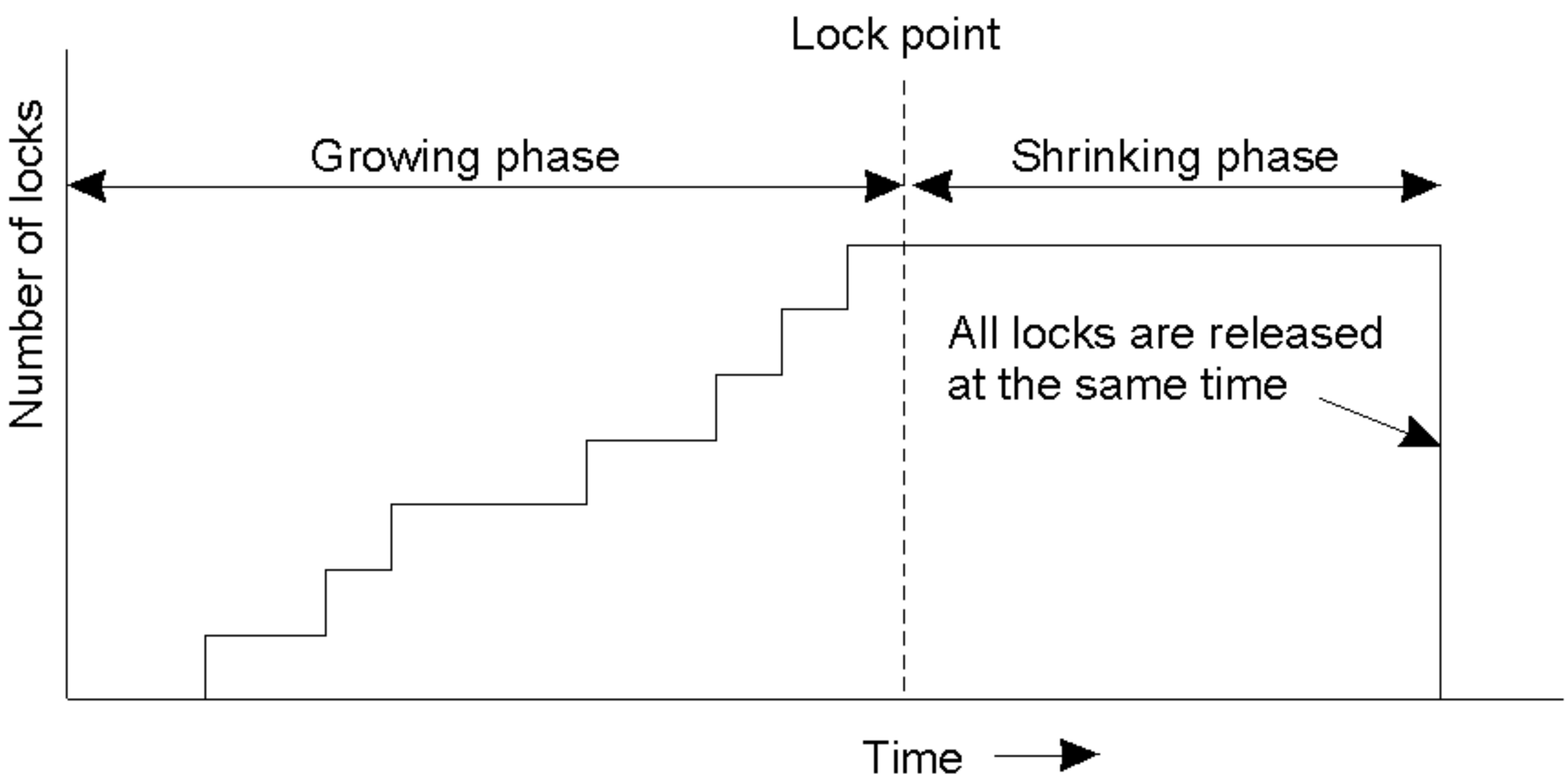
Two-Phase Locking

- Two-phase locking.



Strict Two-Phase Locking

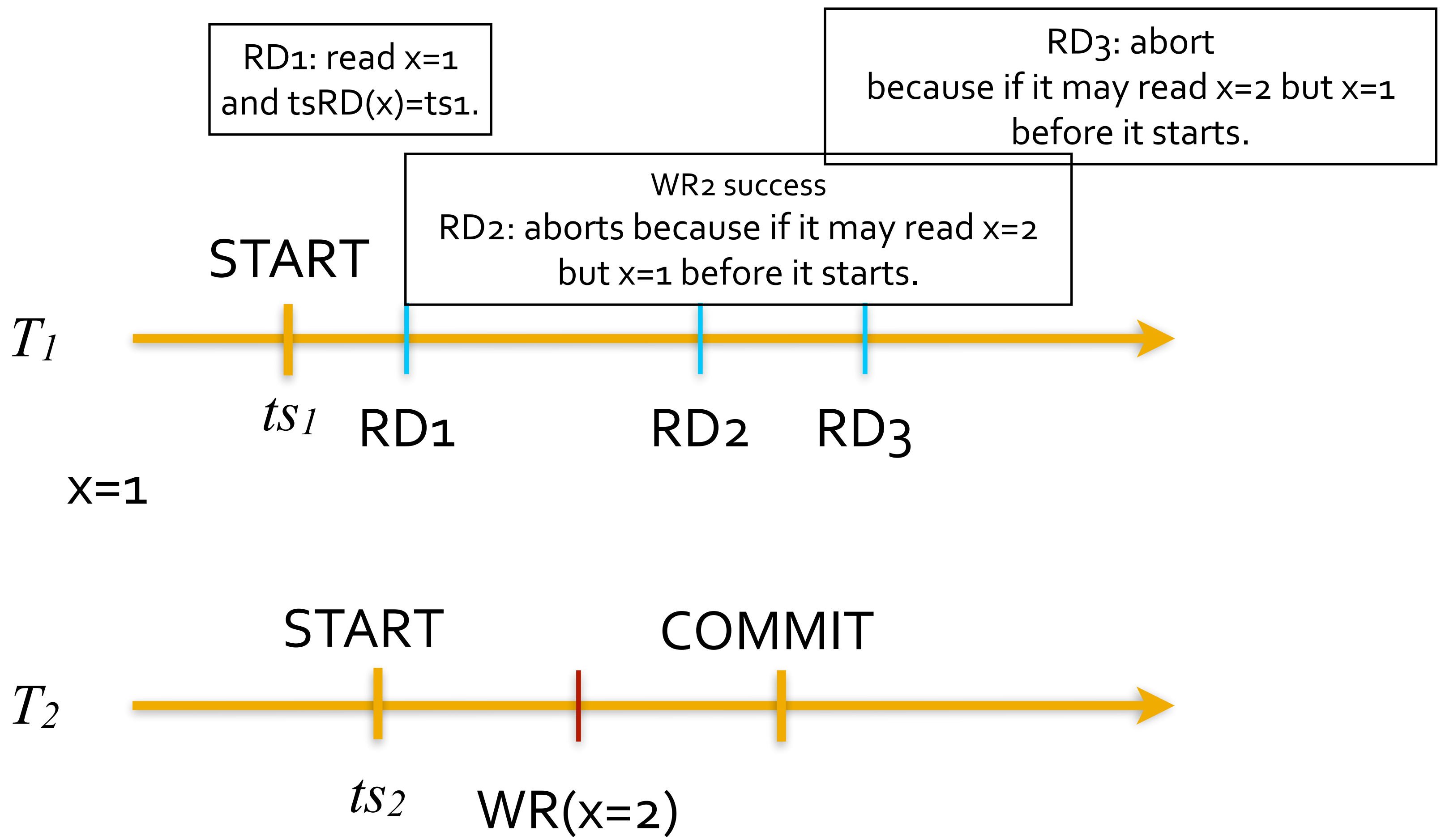
- Strict two-phase locking.



Strict 2PL

- Strict 2PL
  - prevents transactions from
    - reading uncommitted data,
    - overwriting uncommitted data, and
    - unrepeatable reads.
  - It does not guarantee that deadlocks cannot occur,
  - It may additionally be difficult to enforce in distributed data bases, or fault tolerant systems with multiple redundancy.
- A deadlocked schedule supposedly allowed in Strict 2PL.
- Implementation:
  - Centralized 2PL
  - Primary 2PL
  - Distributed 2PL

Read-Write Conflicts



RD1 successes and WR aborts.

Timestamp ordering

- Each transaction T is stamped when it starts.
- Every data item had a read timestamp and write timestamp.
  - tsRD(x): the start time of the transaction that recently reads X and commits.
  - tsWR(x):the start time of the transaction that recently changes X and commits.
  - tstent(x):the start time of the transaction that recently changes x and not yet commits.
  - Timestamps are unique according to Lamport's approach.
- When two operations conflict, the data manager processes the one with the lowest timestamp.

Pessimistic Concurrency Control

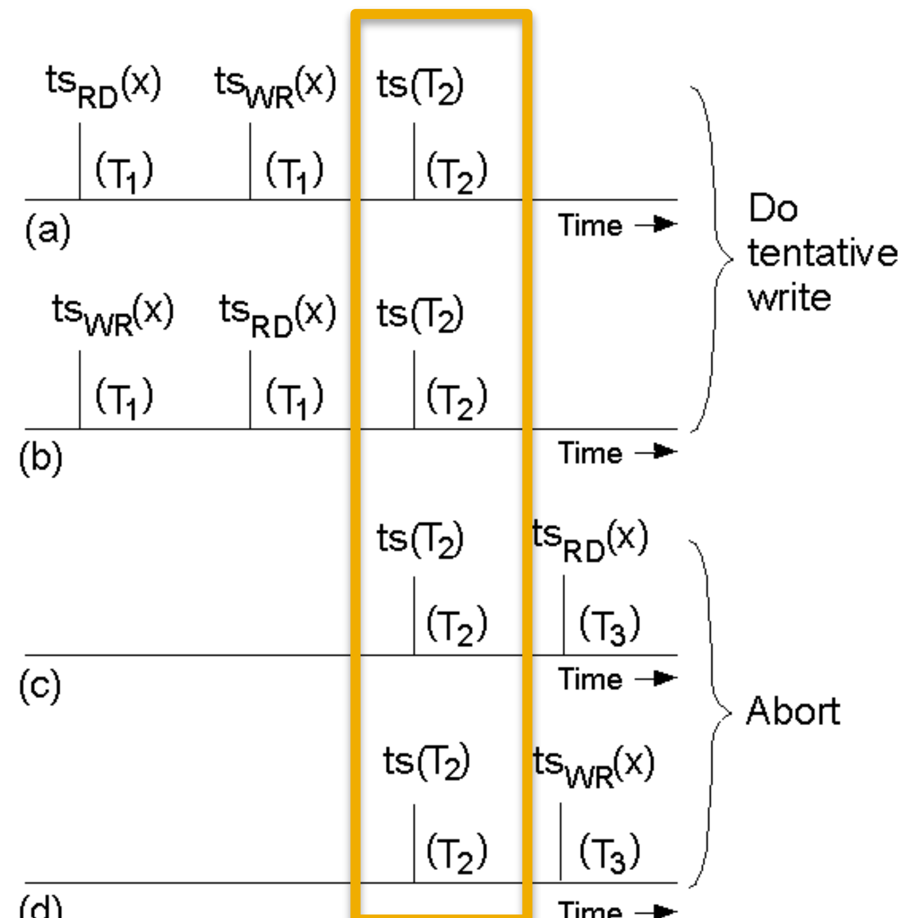
- When a scheduler receives an operation read(T,x) from transaction T with timestamp ts,

- When a scheduler receives an operation write(T,x) from transaction T with timestamp ts,

Pessimistic Timestamp Ordering

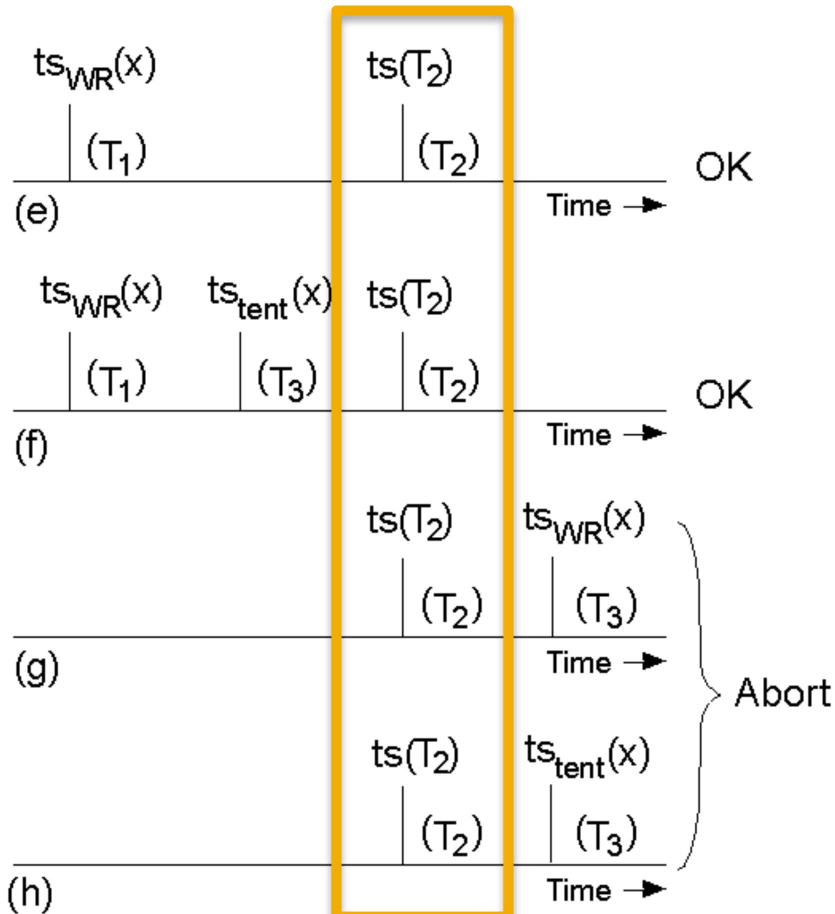
- Three transaction: T1, T2, and T3.
- Transaction T1 starts long time ago and used every data item needed by T2 and T3.
- Transaction T2 and T3 start concurrently with ts(T2) < ts(T3)

(c) and (d): T3 starts later but reads/writes x before T2's operations.



T2 Write

Pessimistic Timestamp Ordering



T2 Read

(e): no conflict.
(f): T2 waits for the interloper to commit and continue.
(g): abort because a later update on x already commits.
(h): abort for the same reason although not yet committed.

Optimistic Timestamp Ordering

- Idea: Just go ahead and do whatever you want to without paying attention to what anybody else is doing. If there is a problem, worry about it later.
- At the end of the transaction, it checks all other transactions to see if any of its items have been changed since the transaction started.
- It is deadlock free.