# Distributed Systems

**15-440/640**

Fall 2018

## 8 – Distributed Mutual Exclusion

Readings: Tanenbaum Book, Chapters 6.3 and 6.4.

# What is “Scalability”?

Ability to easily and rapidly grow the system

A consequence of success    failing systems rarely grow :-)

# How to scale?

Two fundamental approaches:

| <u>Scale Up</u> (aka “vertical scaling”) | <u>Scale Out</u> (aka “horizontal scaling”) |
|---|---|
| **add resources to a single node** e.g., more and faster CPUs, GPUs | **add more nodes** to the distributed system |
| **no application changes** huge win in terms of cost and time | **application has to conform to scale out design** - may involve total rewrite |

Challenges when scaling out?

# Scale-Out Distributed Systems

Distributed Databases

Distributed and Cluster Filesystems

Distributed Computation Frameworks ( → P1)

What if concurrent accesses to shared resources?

How to coordinate access to shared resources?

Inconsistent data, corrupted resources..

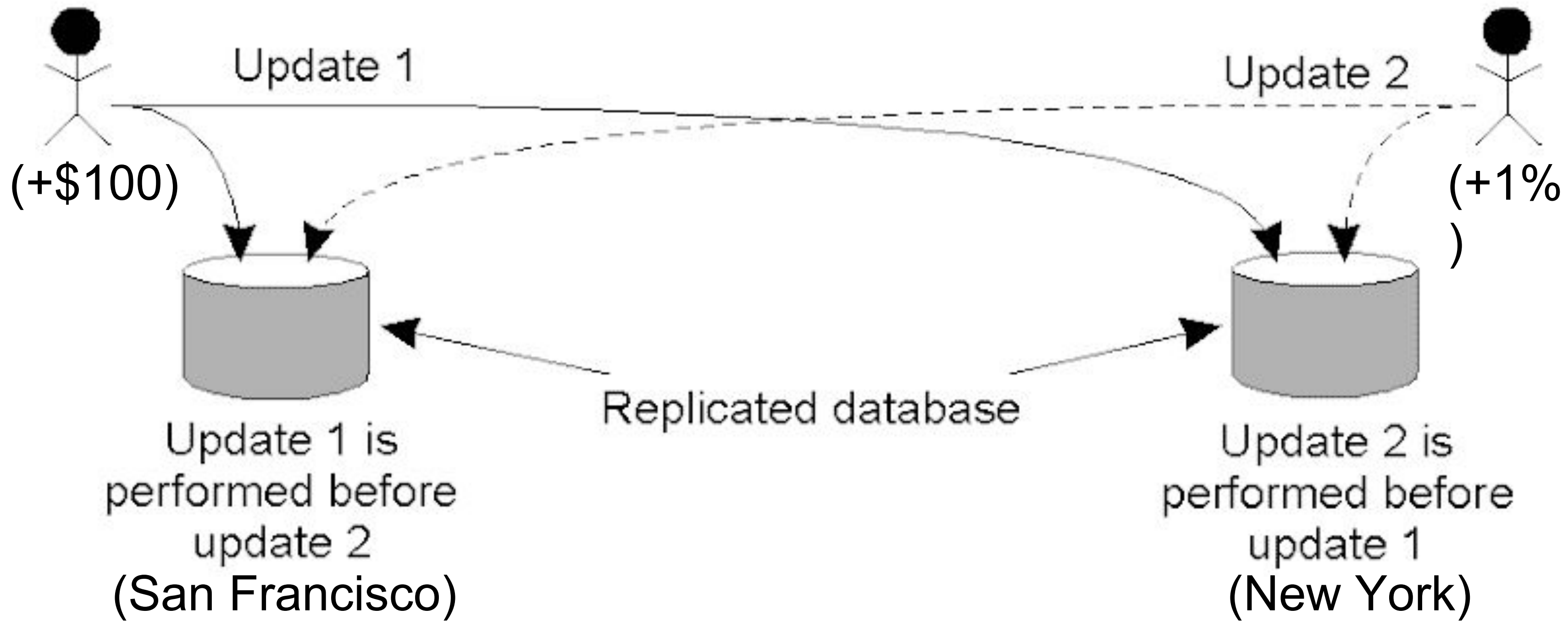
```
while true:
   Perform local operations
Acquire(lock)
   Execute critical section
Release(lock)
```

## Mutex Requirements

1. Correctness/Safety: At most one process holds the lock/enter C.S. at a time

2. Fairness: Any process that makes a request must be granted lock
   - Implies that system must be deadlock-free
   - Assumes that no process will hold onto a lock indefinitely
   - Eventual fairness: Waiting process will not be excluded forever
   - Bounded fairness: Waiting process will get lock within some bounded number of cycles (typically n)

## Distributed Database


Update 1
Update 2
(+$100)
(+1%
)
Replicated database
Update 1 is performed before update 2
(San Francisco)
Update 2 is performed before update 1
(New York)

- San Fran customer adds $100, NY bank adds 1% interest
  - San Fran will have $1,111 and NY will have $1,110
- Updating a replicated database and leaving it in an inconsistent state.

# Distributed Mutex Requirements

No shared memory → message passing.

Focus today

1. Low message overhead
2. No bottlenecks
3. Tolerate out-of-order messages
4. Allow processes to join protocol or to drop out
5. Tolerate failed processes
6. Tolerate dropped messages

Assumptions:

- Total number of processes is fixed at n
- No process fails or misbehaves
- Communication never fails, but messages may be reordered

multiple senders

## Goal of Today's Lecture

Understand trade-offs of main algorithms:

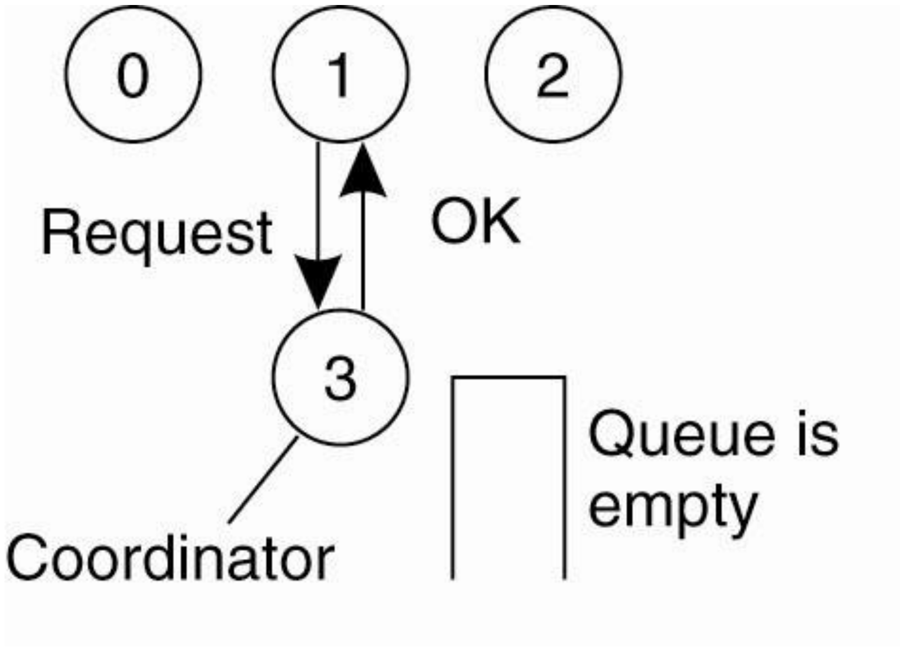
- Centralized Mutual Exclusion
- Bully Leader Election
- Decentralized Mutual Exclusion
- Totally-Ordered Multicast
- Lamport Mutual Exclusion
- Ricart & Agrawala Mutual Exclusion
- Token Ring Mutual Exclusion

## Goal of Today's Lecture

Understand trade-offs of main algorithms:

- Centralized Mutual Exclusion
- Bully Leader Election
- Decentralized Mutual Exclusion
- Totally-Ordered Multicast
- Lamport Mutual Exclusion
- Ricart & Agrawala Mutual Exclusion
- Token Ring Mutual Exclusion

## Centralized Algorithm (1)



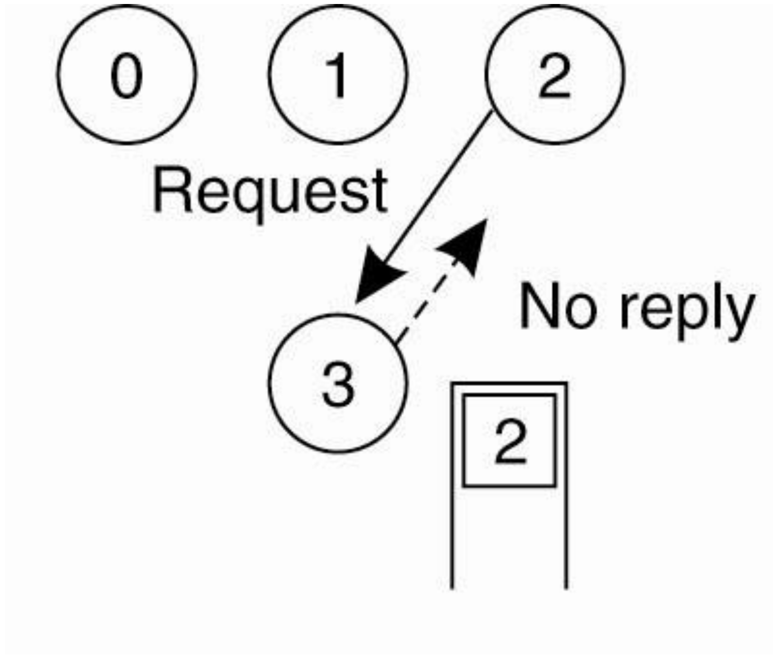
@ Server:

```
while true:
    m = Receive()
    If m == (Request, i):
If Available():
       Send (Grant) to i
```

@ Client → Acquire:

```
Send (Request, i) to coordinator
Wait for reply
```

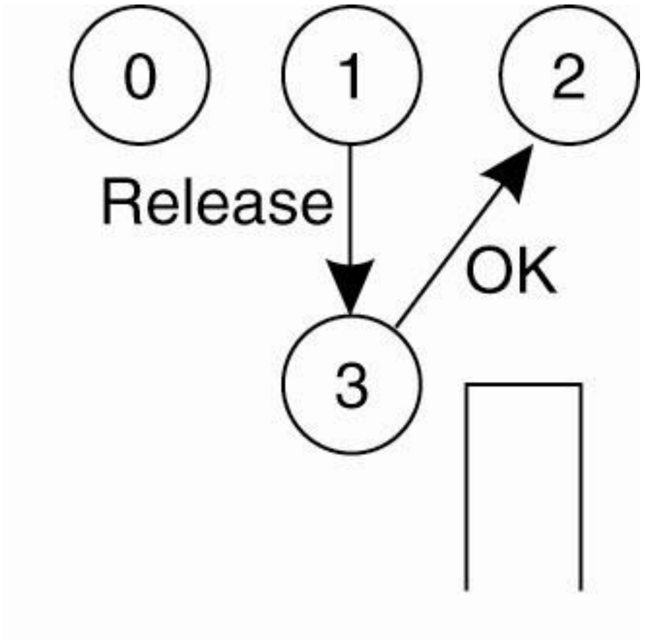
## Centralized Algorithm (2)



@ Server:

```
while true:
    m = Receive()
    If m == (Request, i):
If Available():
        Send (Grant) to I
        else:
            Add i to Q
```

# Centralized Algorithm (3)



## @ Server:

```
while true:
    m = Receive()
    If m == (Request, i):
      If Available():
       Send (Grant) to I
      else:
          Add i to Q
    If m == (Release)&&!empty(Q):
      Remove ID j from Q
      Send (Grant) to j
```

## @ Client → Release:

```
Send (Release) to coordinator
```

## Centralized Algorithm: Summary

- Correctness:
  - Clearly safe
  - Fairness depends on queuing policy.
    - E.g., if always gave priority to lowest process ID, then processes 1 & 2 can lock out 3
- Performance
  - "cycle" is a complete round of the protocol with one process i entering its critical section and then exiting.
  - 3 messages per cycle (1 request, 1 grant, 1 release)
  - Lock server creates bottleneck
- Issues
  - What happens when coordinator crashes?
  - What happens when it reboots?

What can we do when coordinator crashes?

## Goal of Today's Lecture

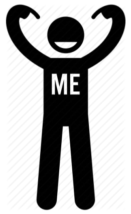
Understand trade-offs of main algorithms:

- Centralized Mutual Exclusion
- Bully Leader Election
- Decentralized Mutual Exclusion
- Totally-Ordered Multicast
- Lamport Mutual Exclusion
- Ricart & Agrawala Mutual Exclusion
- Token Ring Mutual Exclusion

# Selecting a Leader (Elections)

Goal: anyone can trigger election which automatically determines a **unique new leader**

Non-goals: fairness, majorities, selfish notes



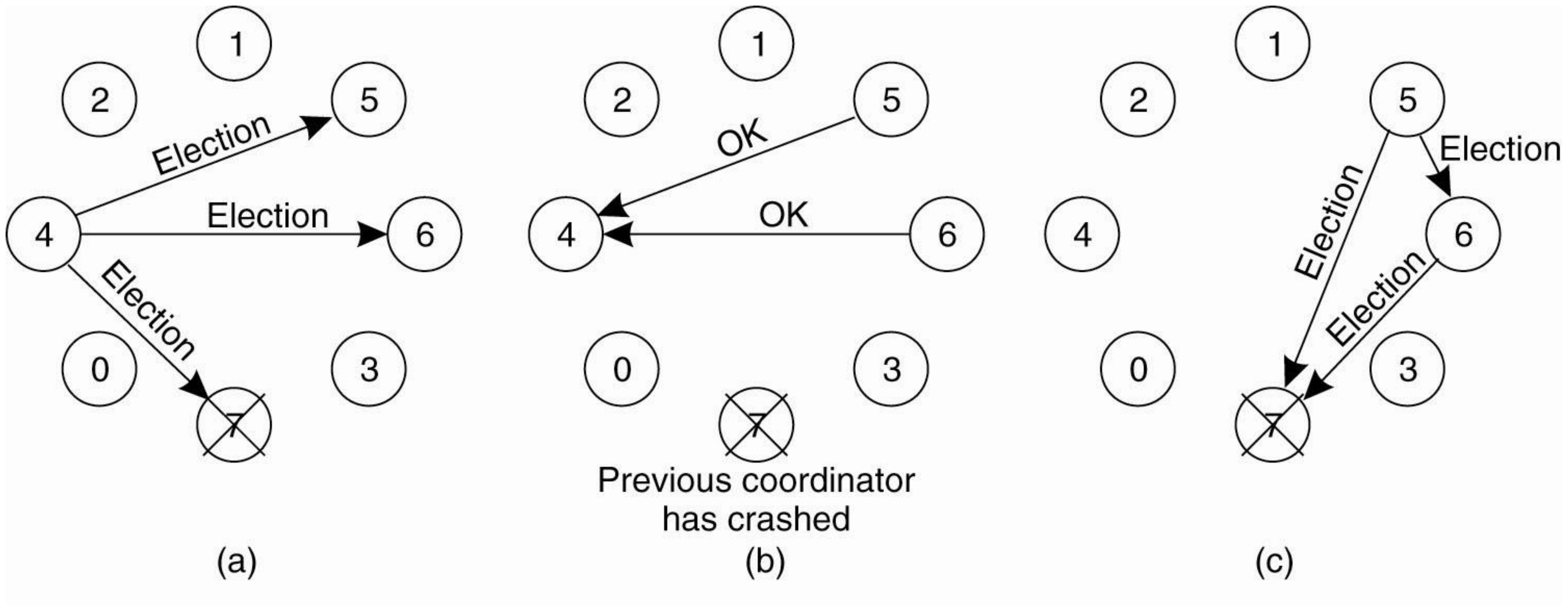
Stage 1: Process P notices that leader has failed

Stage 2: Election Algorithm, e.g., Bully Algorithm

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

# The Bully Leader-Election Algorithm (1)

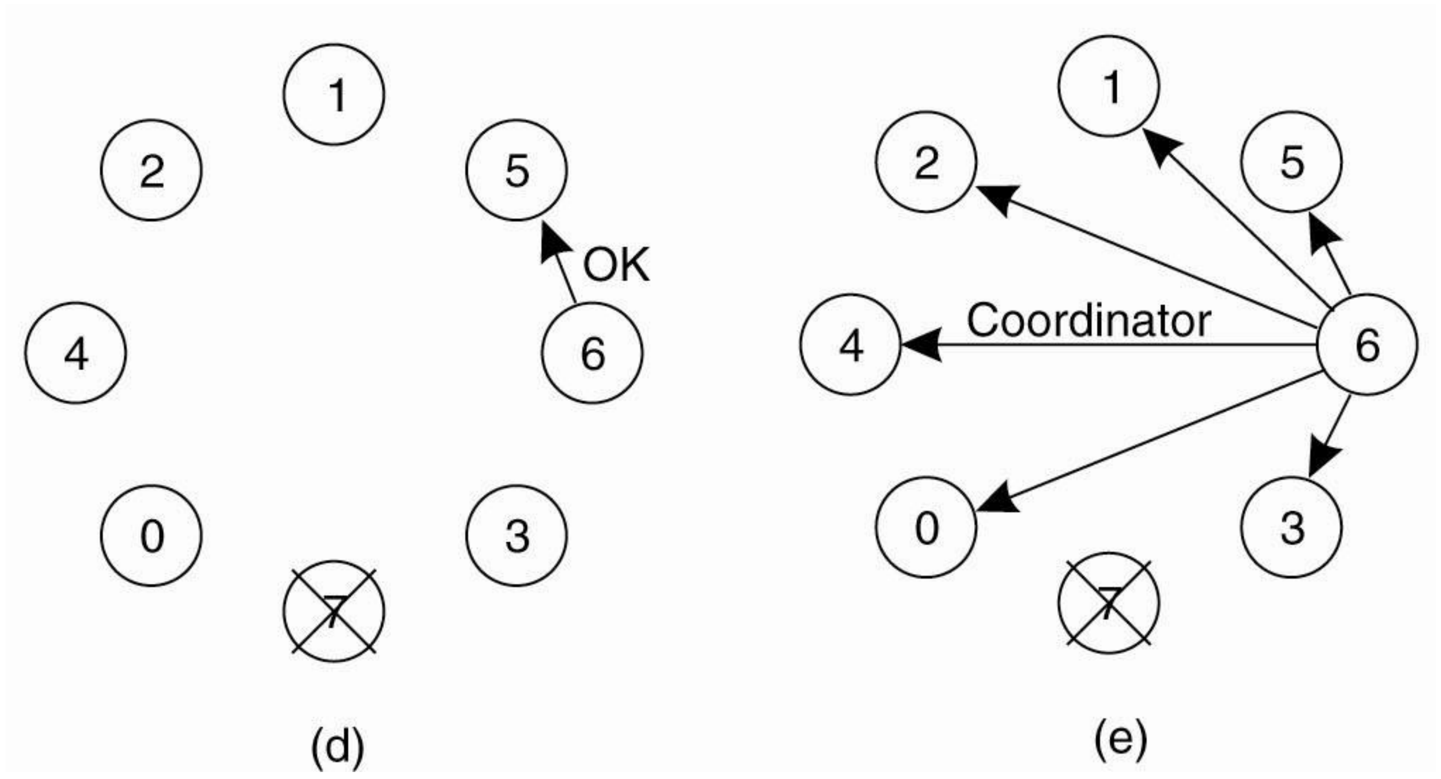
a) Process 4 holds an election
b) Processes 5 and 6 respond, telling 4 to stop
c) Now 5 and 6 each hold an election



## The Bully Leader-Election Algorithm (2)

d) Process 6 tells 5 to stop

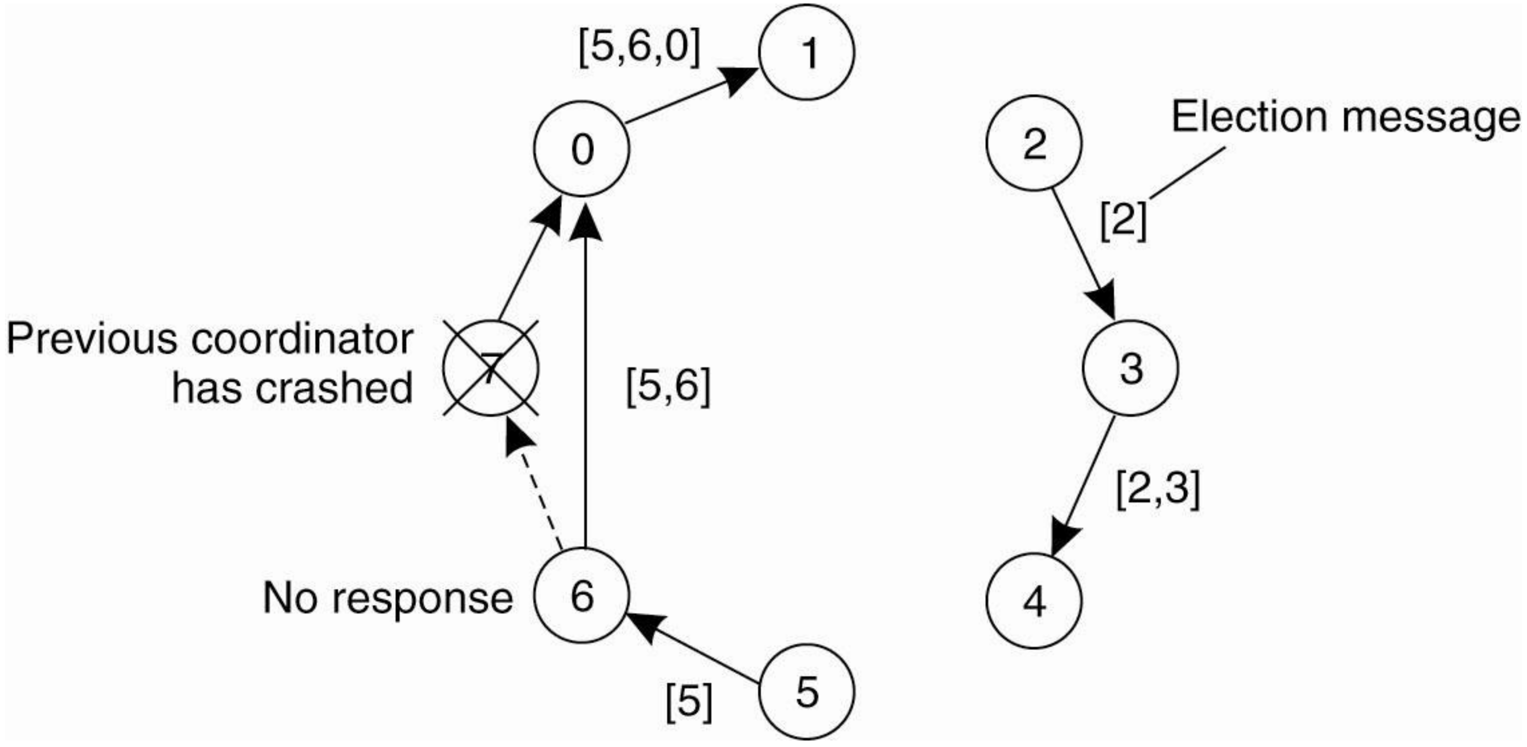
e) Process 6 wins and tells everyone.



Simple but effective. More algorithms in the book.

# A Ring Algorithm

- Election algorithm using a ring.



## Goal of Today's Lecture

Understand trade-offs of main algorithms:

- Centralized Mutual Exclusion
- Bully Leader Election
- Decentralized Mutual Exclusion
- Totally-Ordered Multicast
- Lamport Mutual Exclusion
- Ricart & Agrawala Mutual Exclusion
- Token Ring Mutual Exclusion

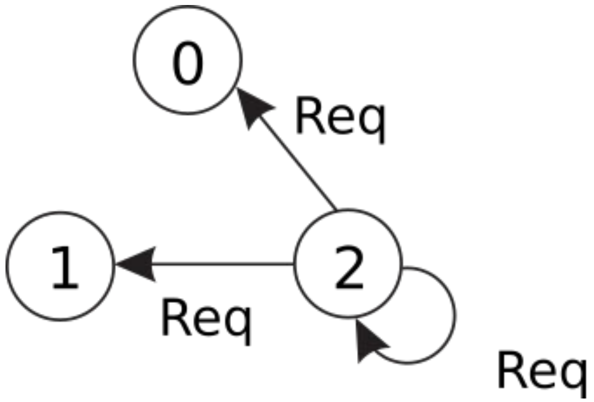
## Decentralized Algorithm (1)

Opposite extreme to centralized algorithm

Minimize state that is distributed across nodes

Assume that there are n coordinators

- Get a majority vote from m > n/2 coordinators
- Reply immediately with GRANT or DENY



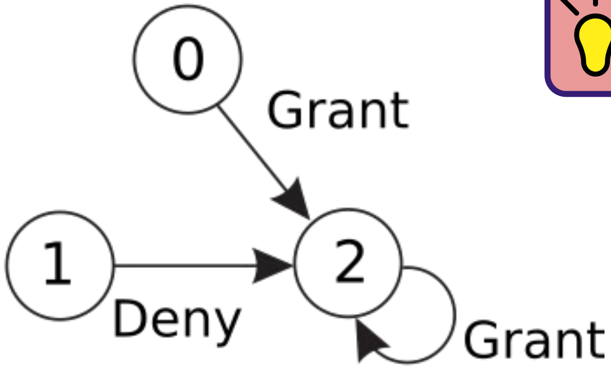
# Decentralized Algorithm (2)

Opposite extreme to centralized algorithm

Minimize state that is distributed across nodes

Assume that there are n coordinators

- Get a majority vote from m > n/2 coordinators
- Reply immediately with GRANT or DENY



What if you get less than m votes?

- Backoff and retry later
- Large numbers of nodes requesting access can affect availability
- Starvation!

## Decentralized Algorithm: Summary

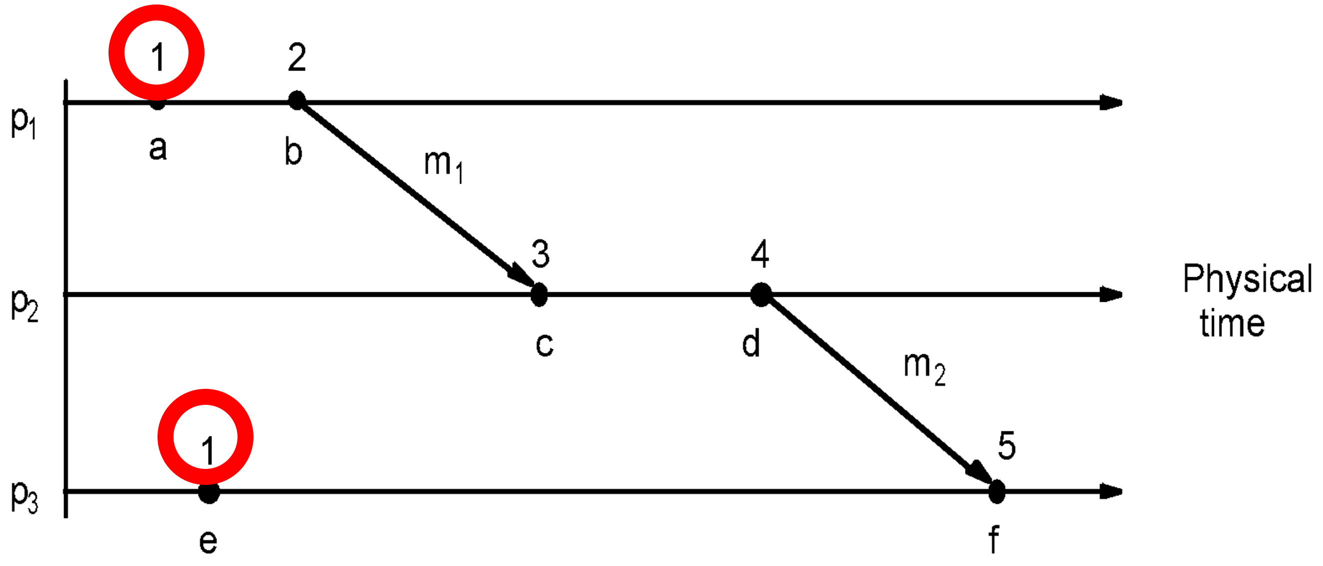
- Correctness:
  - Majority ensures safety
  - Fairness depends on random chance
- Performance
  - 2m + m messages per attempt to get majority
  - unbounded number of messages per cycle
- Issues
  - Node failures are still a problem (forgetting vote on reboot)
  - Backoff and retry problem
  - Starvation

## Goal of Today's Lecture

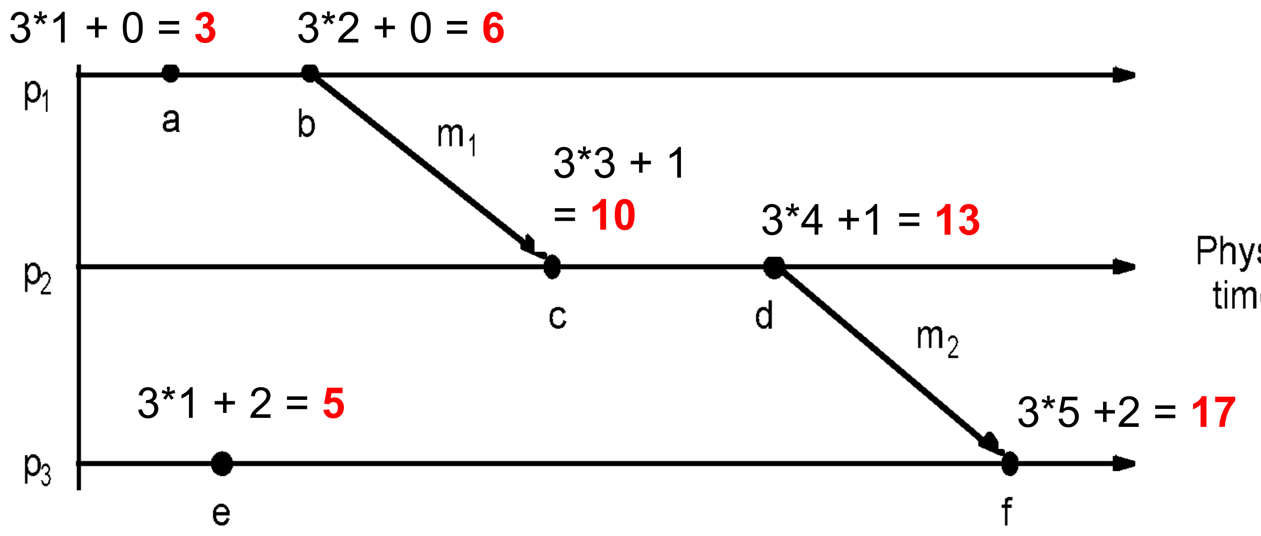
Understand trade-offs of main algorithms:

- Centralized Mutual Exclusion
- Bully Leader Election
- Decentralized Mutual Exclusion
- Totally-Ordered Multicast
- Lamport Mutual Exclusion
- Ricart & Agrawala Mutual Exclusion
- Token Ring Mutual Exclusion

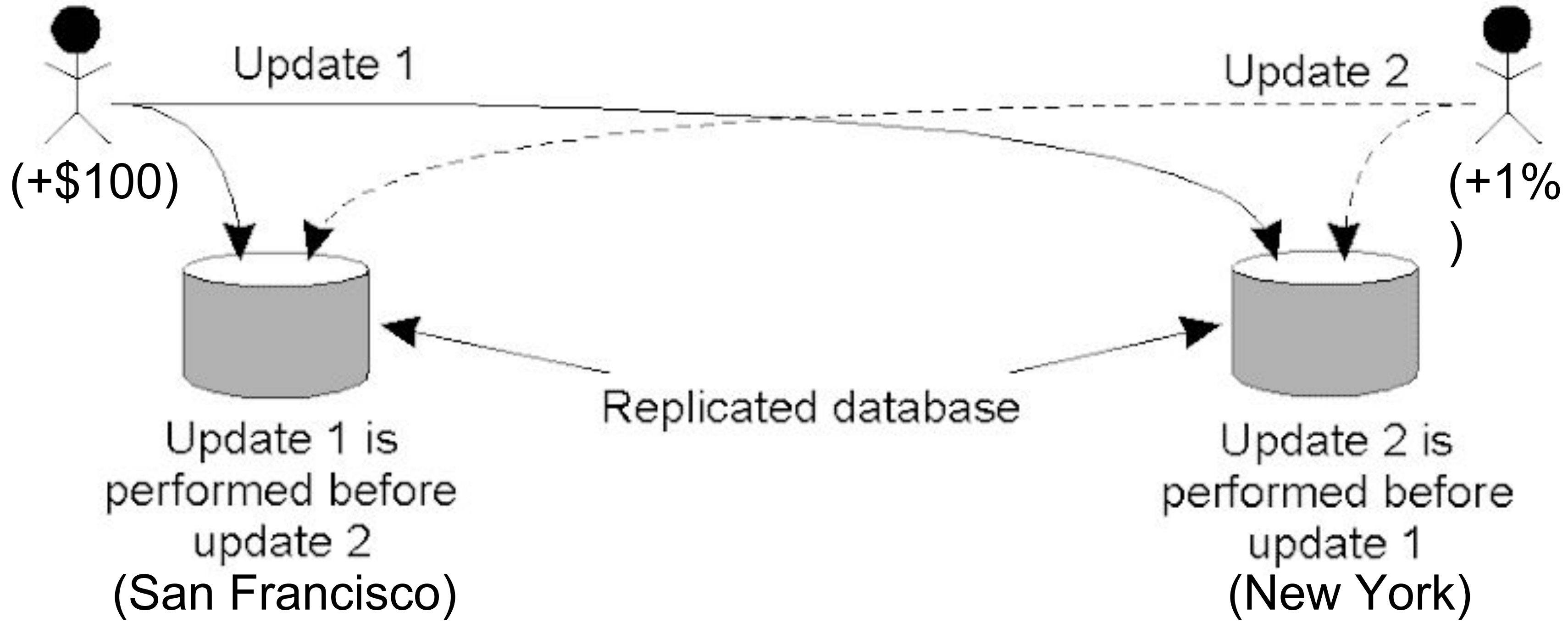
# Recall Logical Lamport Clocks



Total order: break ties using the process ID: $L(e) = M * L_i(e) + i$



## A Total Order Would Be Useful


Update 1
Update 2
(+$100)
(+1%
)
Replicated database
Update 1 is performed before update 2
(San Francisco)
Update 2 is performed before update 1
(New York)

- Can use Lamport’s to totally order
- But would need to be able to roll back events
  - Maybe a large number of them!
- **Could we make sure things are in the right order before processing?**

Book pp. 313

## Totally-Ordered Multicast

- A multicast operation by which all messages are delivered in the same order to each receiver.
- Distributed data structure (priority queue)
- Queue (database) updates until they're ACKed
- Uses TO-Lamport Clocks:
  - Each message is timestamped with the current logical time of its sender.
  - Multicast messages are also sent back to the sender.
  - **<u>Assume all messages sent by one sender are received in the order they were sent and that no messages are lost.</u>**

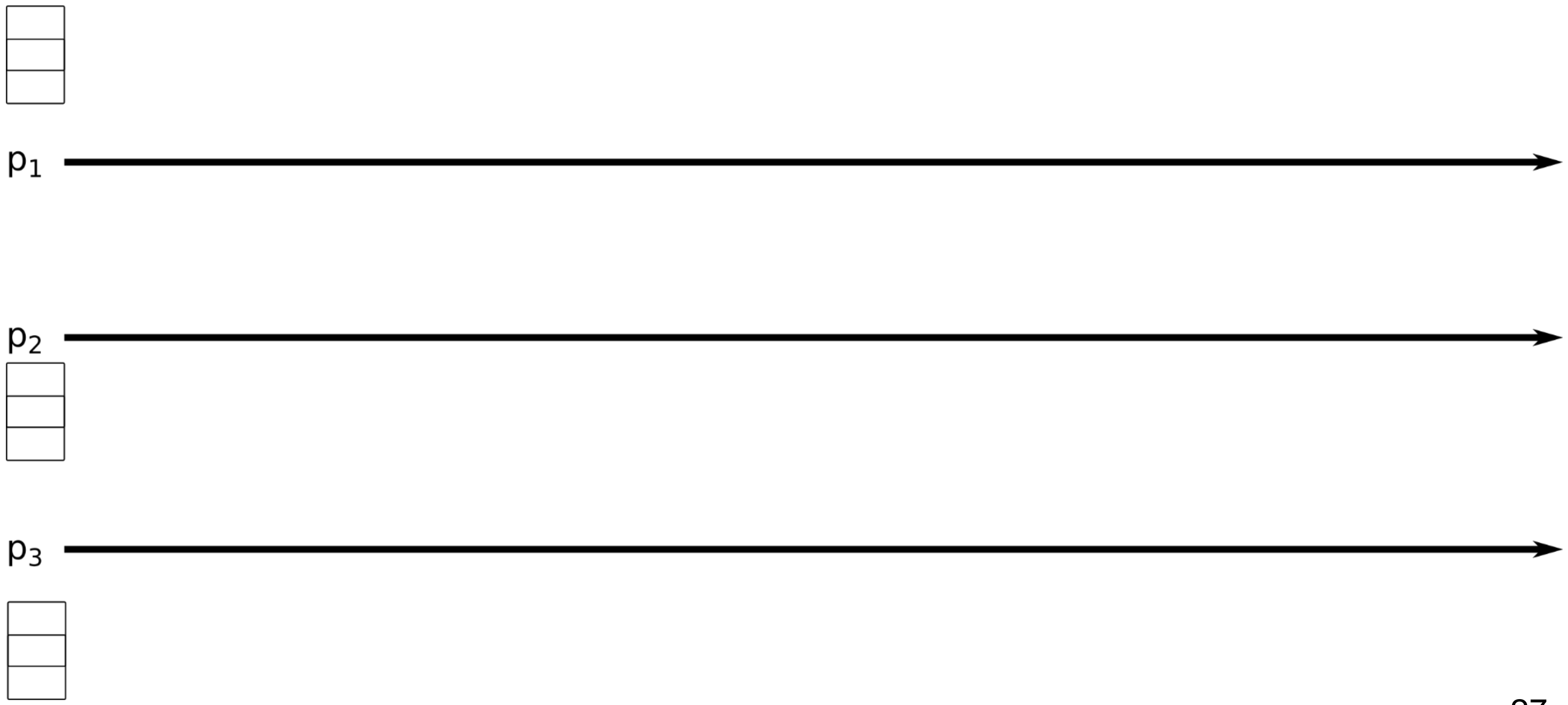
## Totally-Ordered Multicast

- Multicast messages + local timestamp-ordered queue
- Multicasts an ACK to all other processes
- Process only if *both* at queue head and ACK'ed

Long example with three nodes (messages at first and third) and totally crazy message delays.

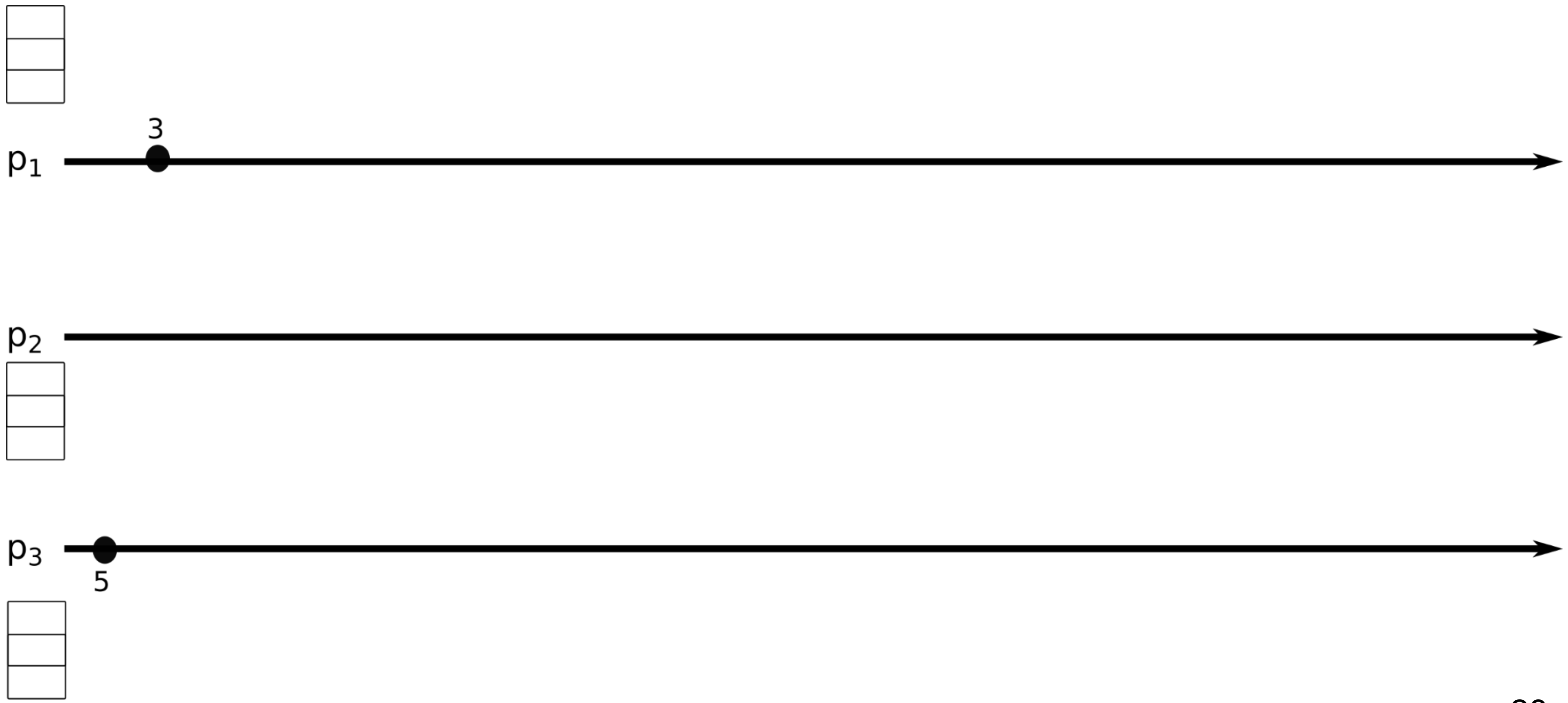
## Totally-Ordered Multicast

- Multicast messages + local timestamp-ordered queue
- Multicasts an ACK to all other processes
- Process only if *both* at queue head and ACK'ed



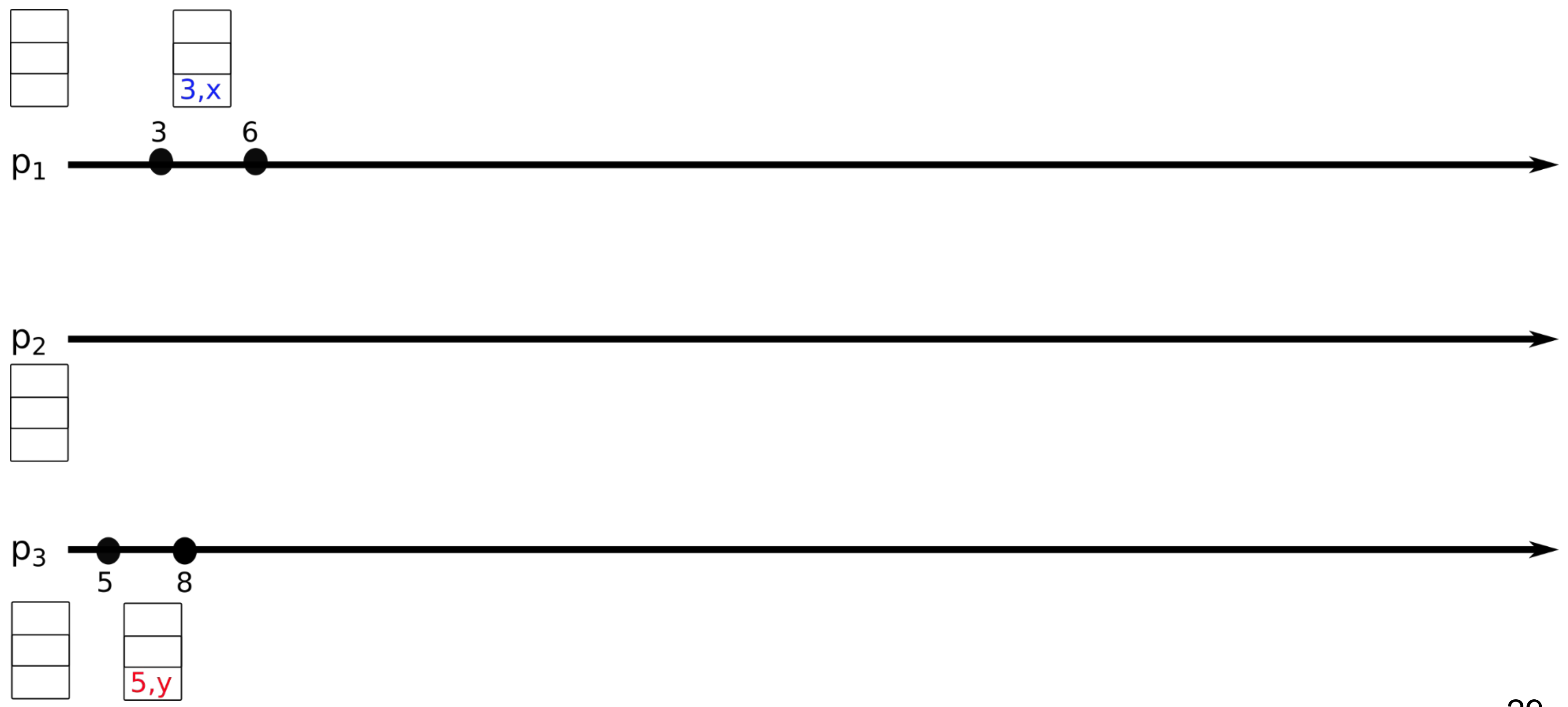
# Totally-Ordered Multicast

- Multicast messages + local timestamp-ordered queue
- Multicasts an ACK to all other processes
- Process only if *both* at queue head and ACK'ed



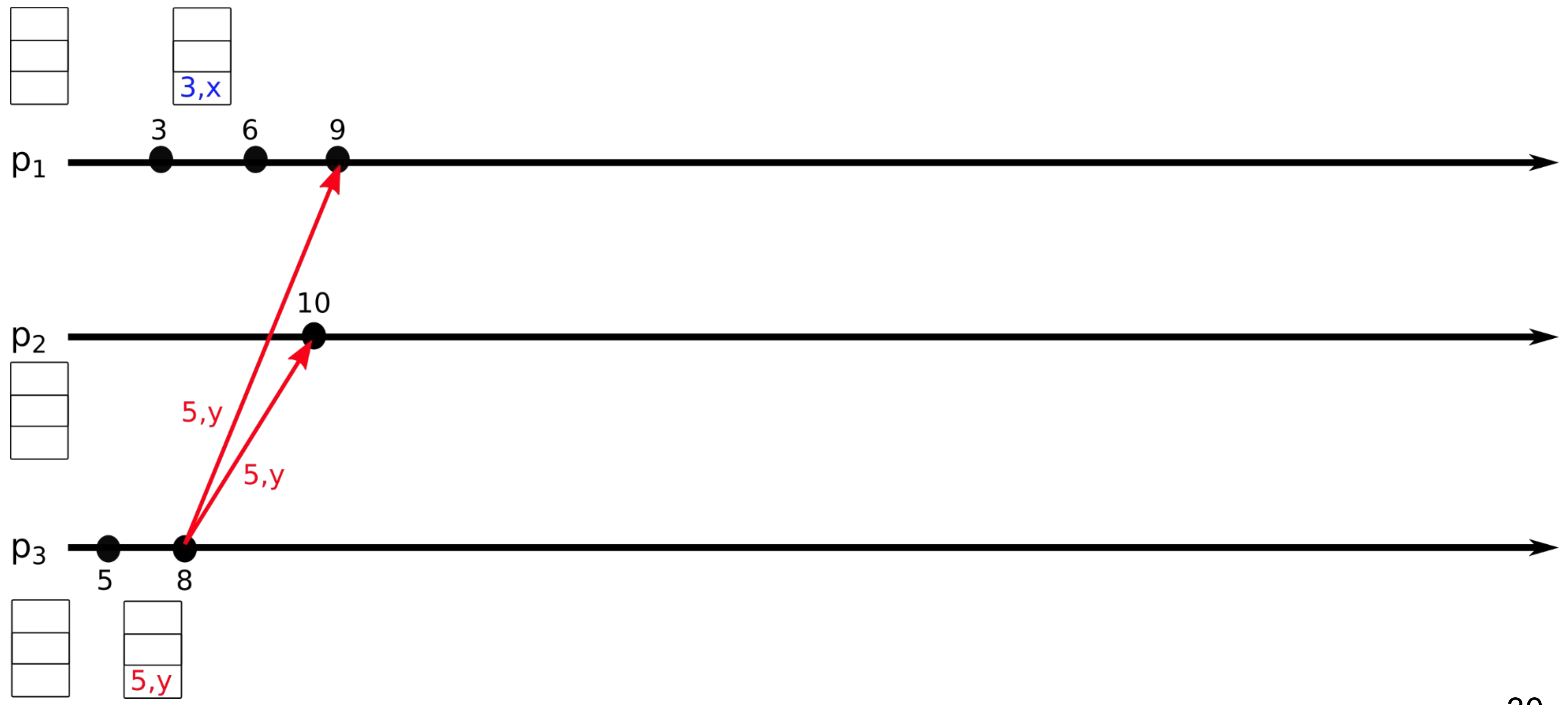
# Totally-Ordered Multicast

- Multicast messages + local timestamp-ordered queue
- Multicasts an ACK to all other processes
- Process only if *both* at queue head and ACK'ed



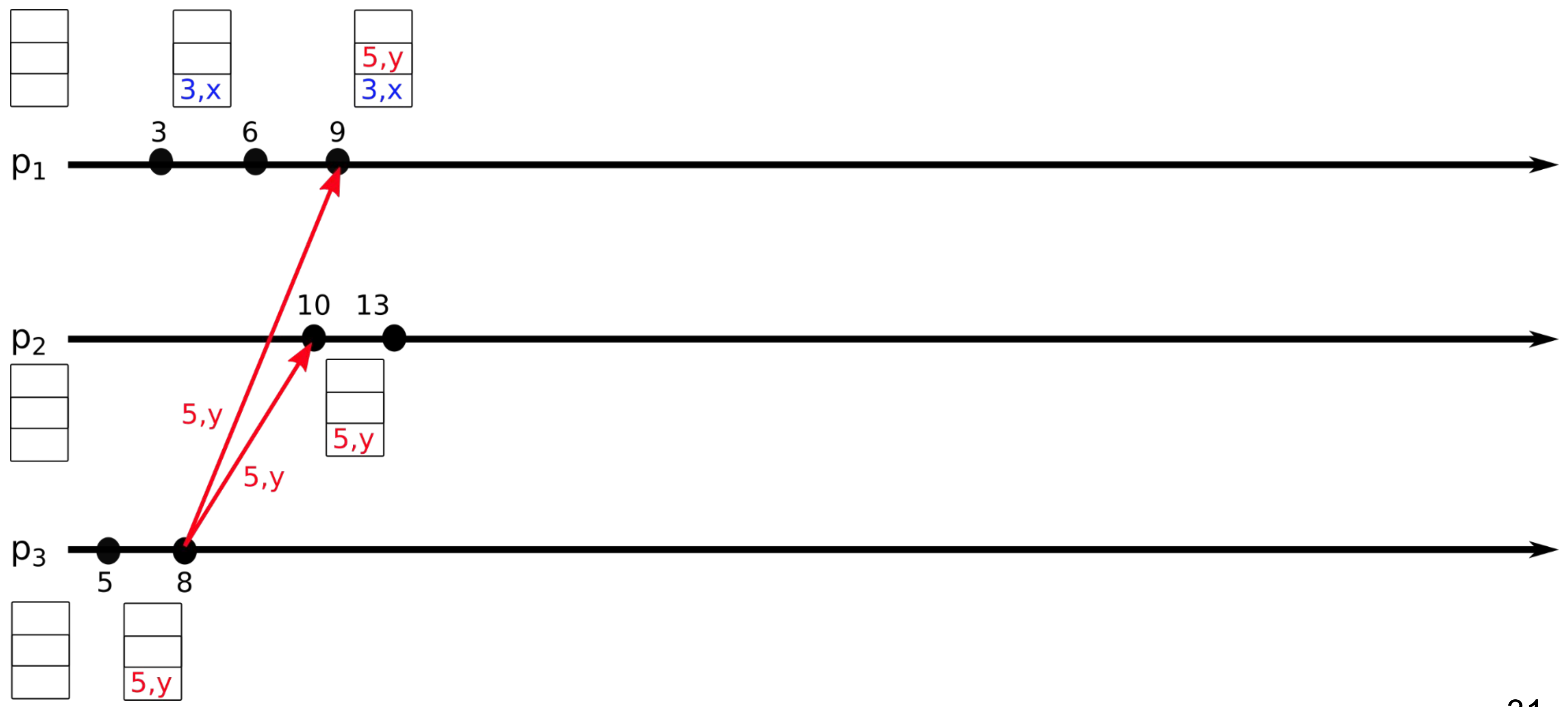
# Totally-Ordered Multicast

- Multicast messages + local timestamp-ordered queue
- Multicasts an ACK to all other processes
- Process only if *both* at queue head and ACK'ed



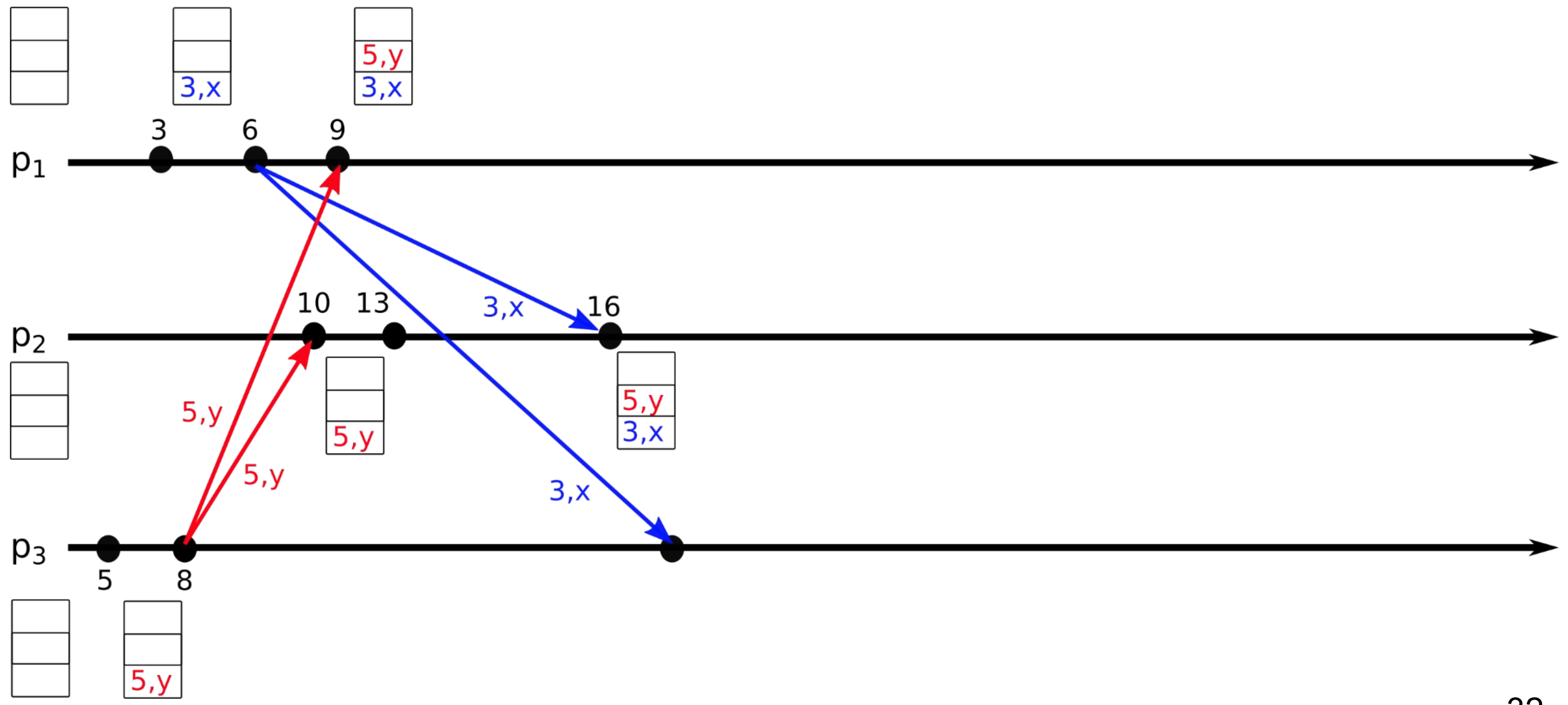
# Totally-Ordered Multicast

- Multicast messages + local timestamp-ordered queue
- Multicasts an ACK to all other processes
- Process only if *both* at queue head and ACK'ed



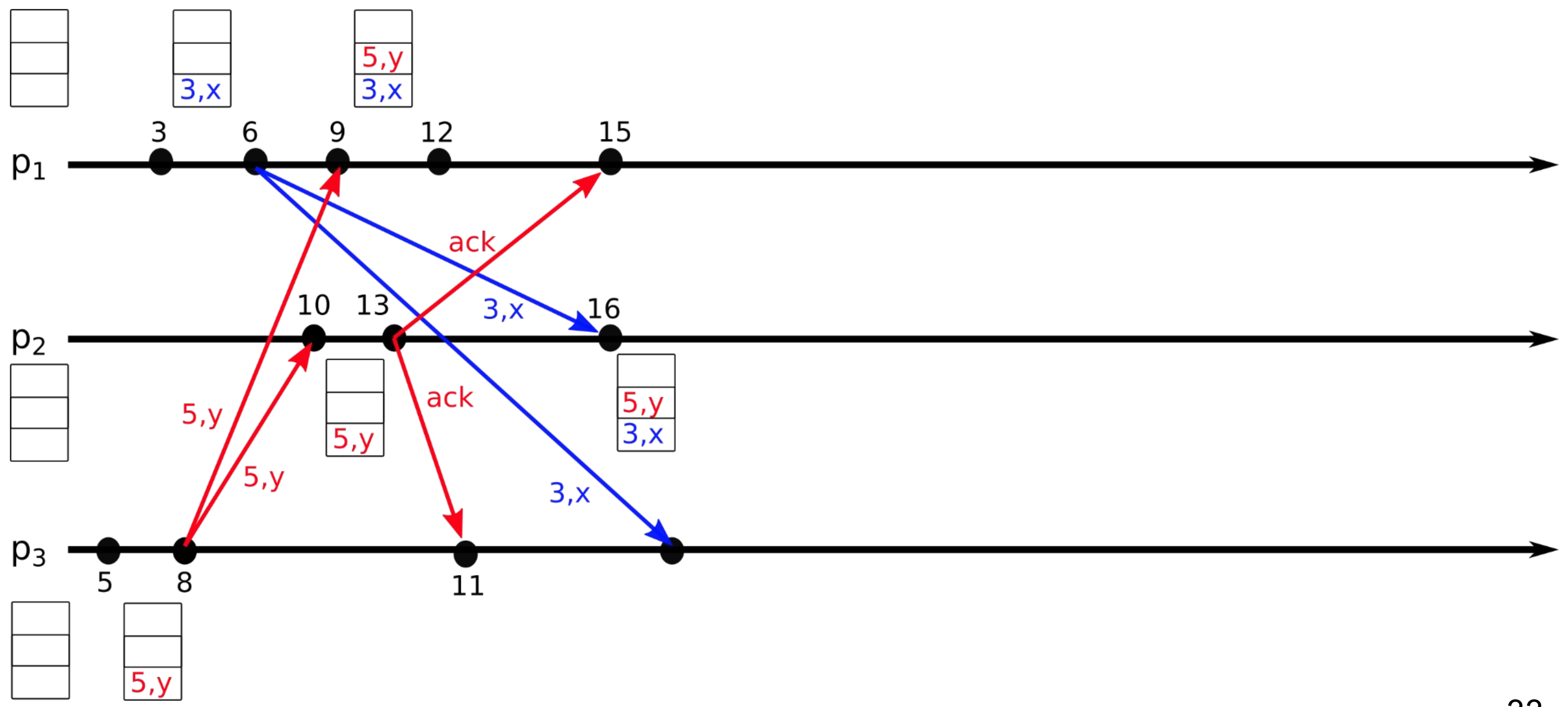
## Totally-Ordered Multicast

- Multicast messages + local timestamp-ordered queue
- Multicasts an ACK to all other processes
- Process only if *both* at queue head and ACK'ed



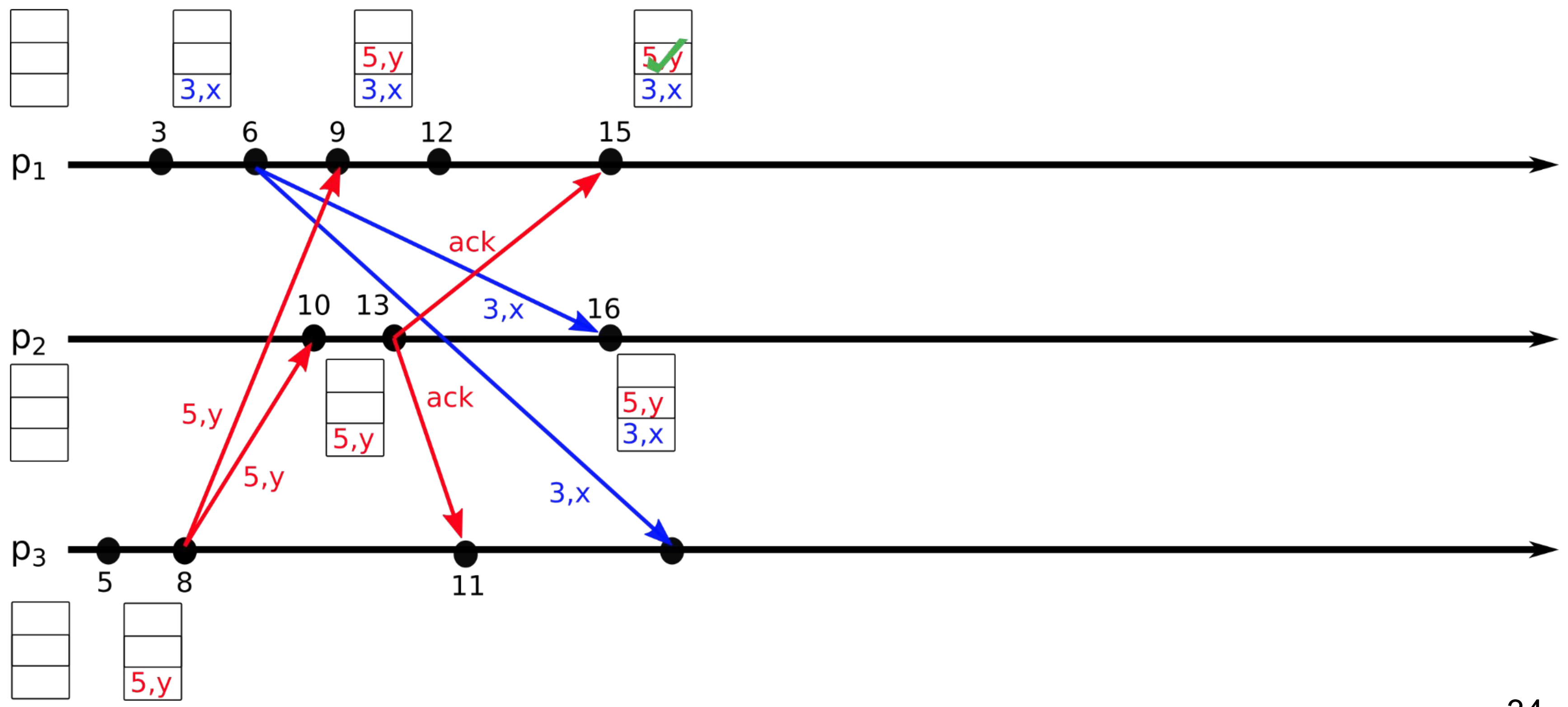
## Totally-Ordered Multicast

- Multicast messages + local timestamp-ordered queue
- Multicasts an ACK to all other processes
- Process only if *both* at queue head and ACK'ed



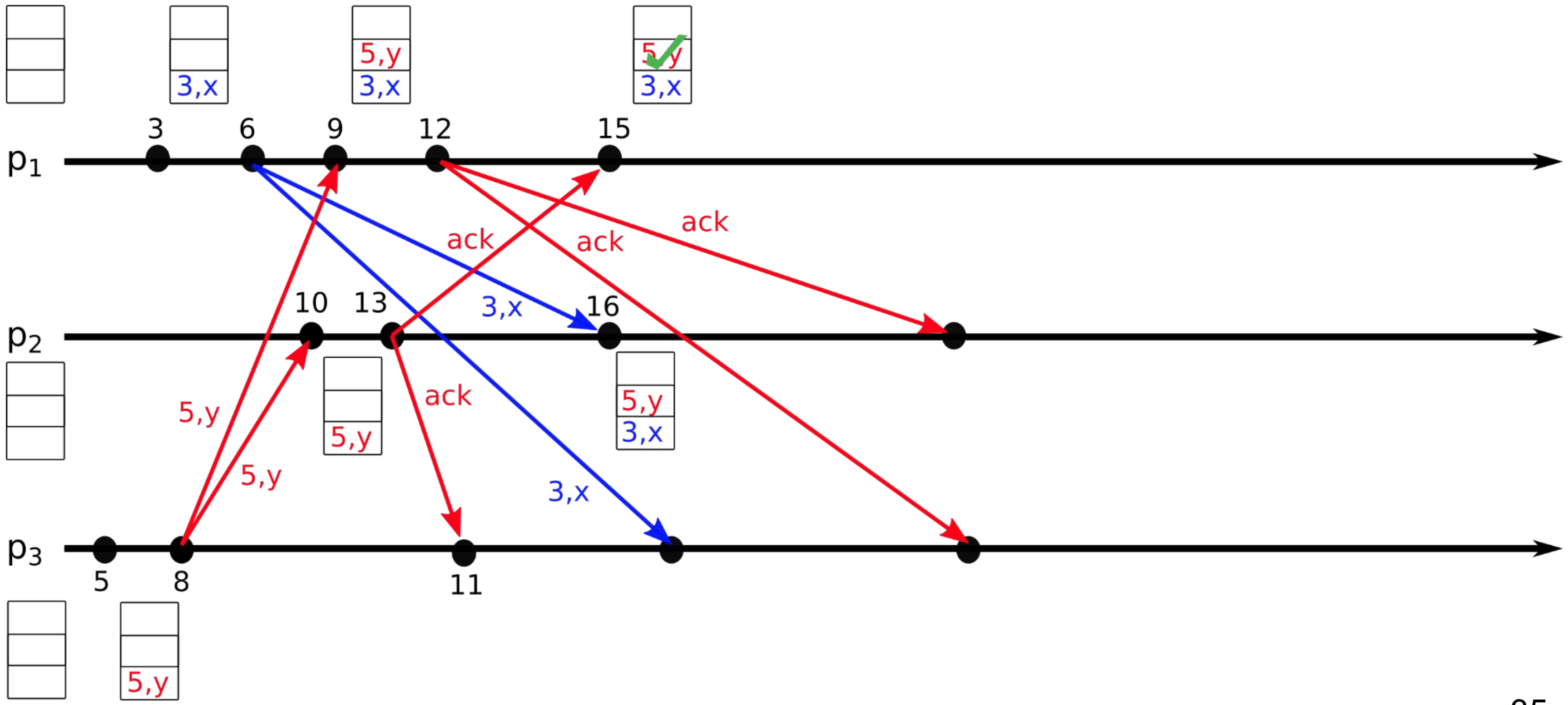
# Totally-Ordered Multicast

- Multicast messages + local timestamp-ordered queue
- Multicasts an ACK to all other processes
- Process only if *both* at queue head and ACK'ed



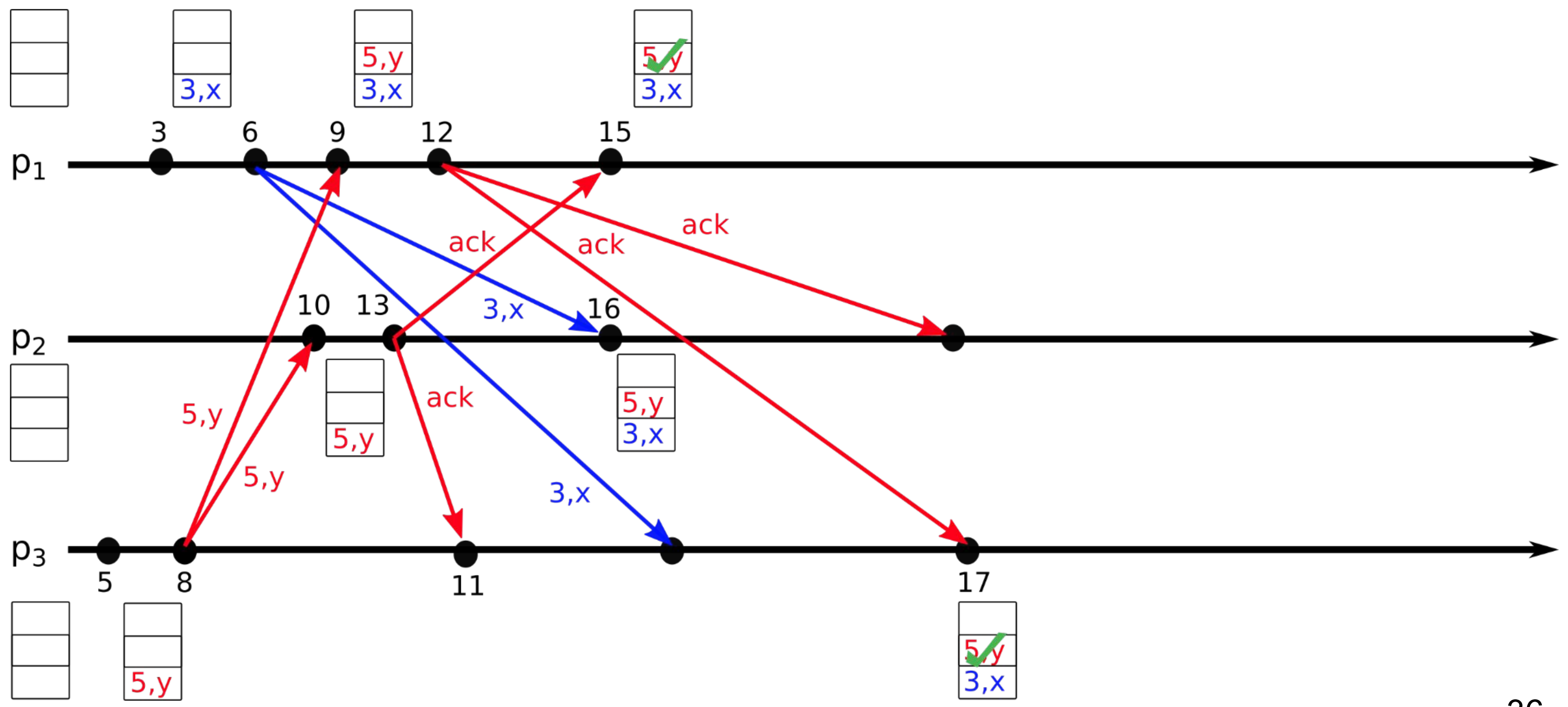
# Totally-Ordered Multicast

- Multicast messages + local timestamp-ordered queue
- Multicasts an ACK to all other processes
- Process only if *both* at queue head and ACK'ed



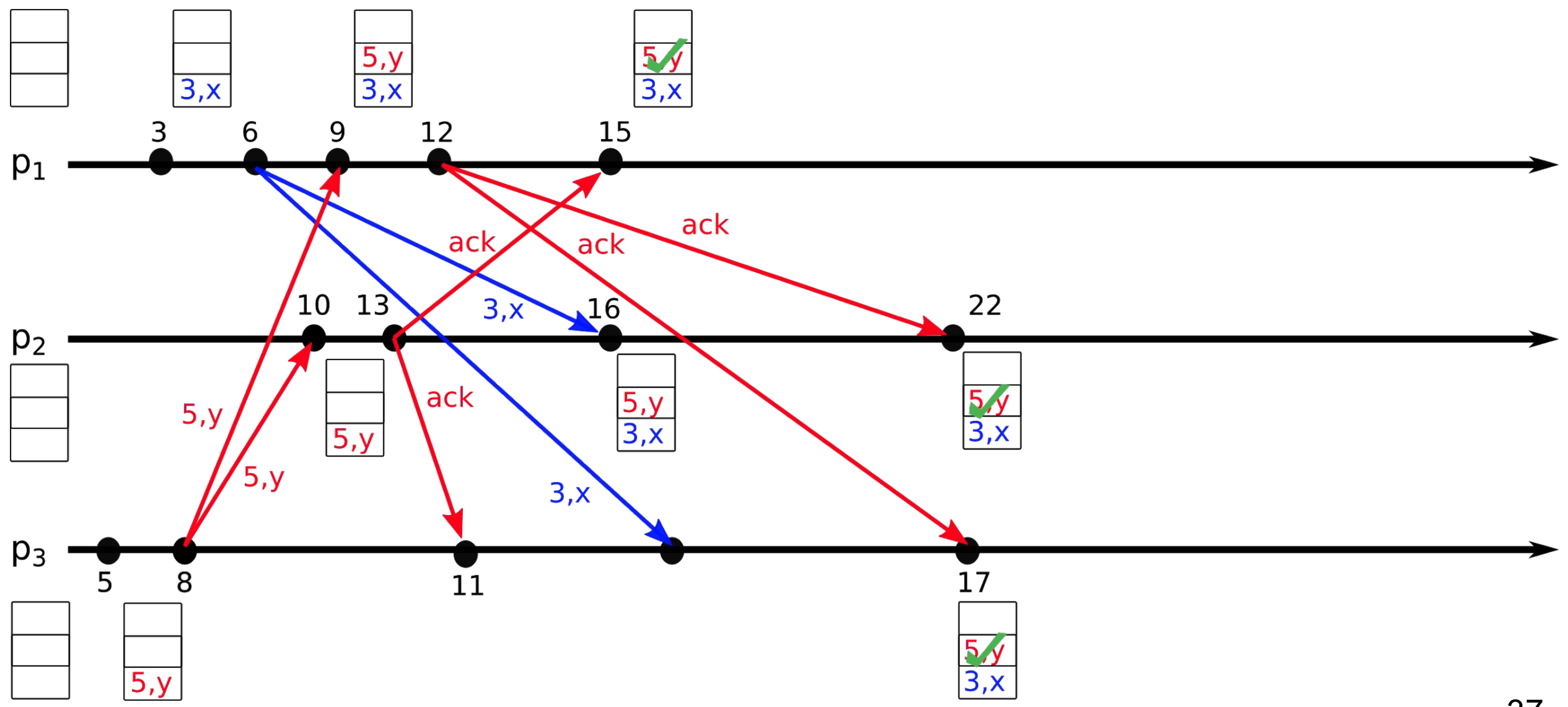
# Totally-Ordered Multicast

- Multicast messages + local timestamp-ordered queue
- Multicasts an ACK to all other processes
- Process only if *both* at queue head and ACK'ed



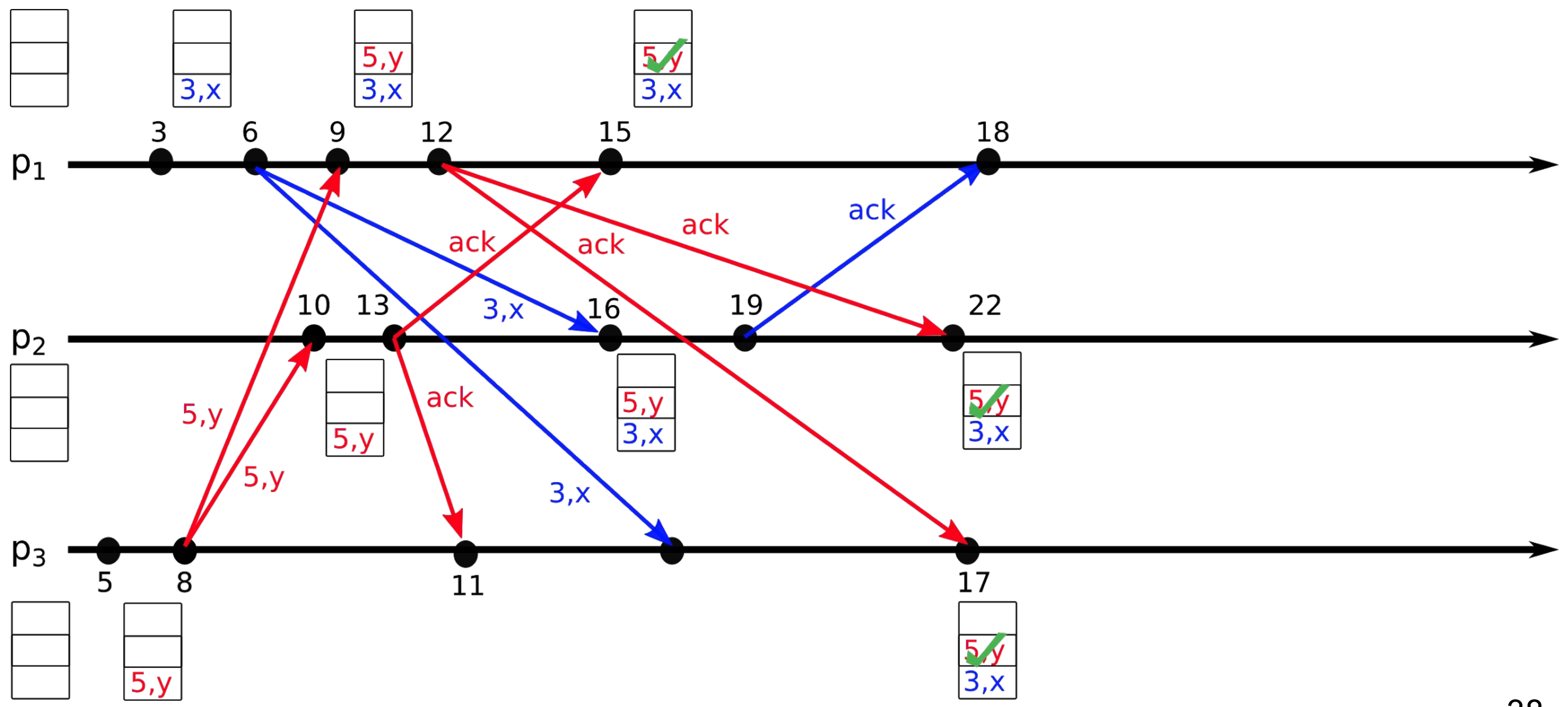
# Totally-Ordered Multicast

- Multicast messages + local timestamp-ordered queue
- Multicasts an ACK to all other processes
- Process only if *both* at queue head and ACK'ed



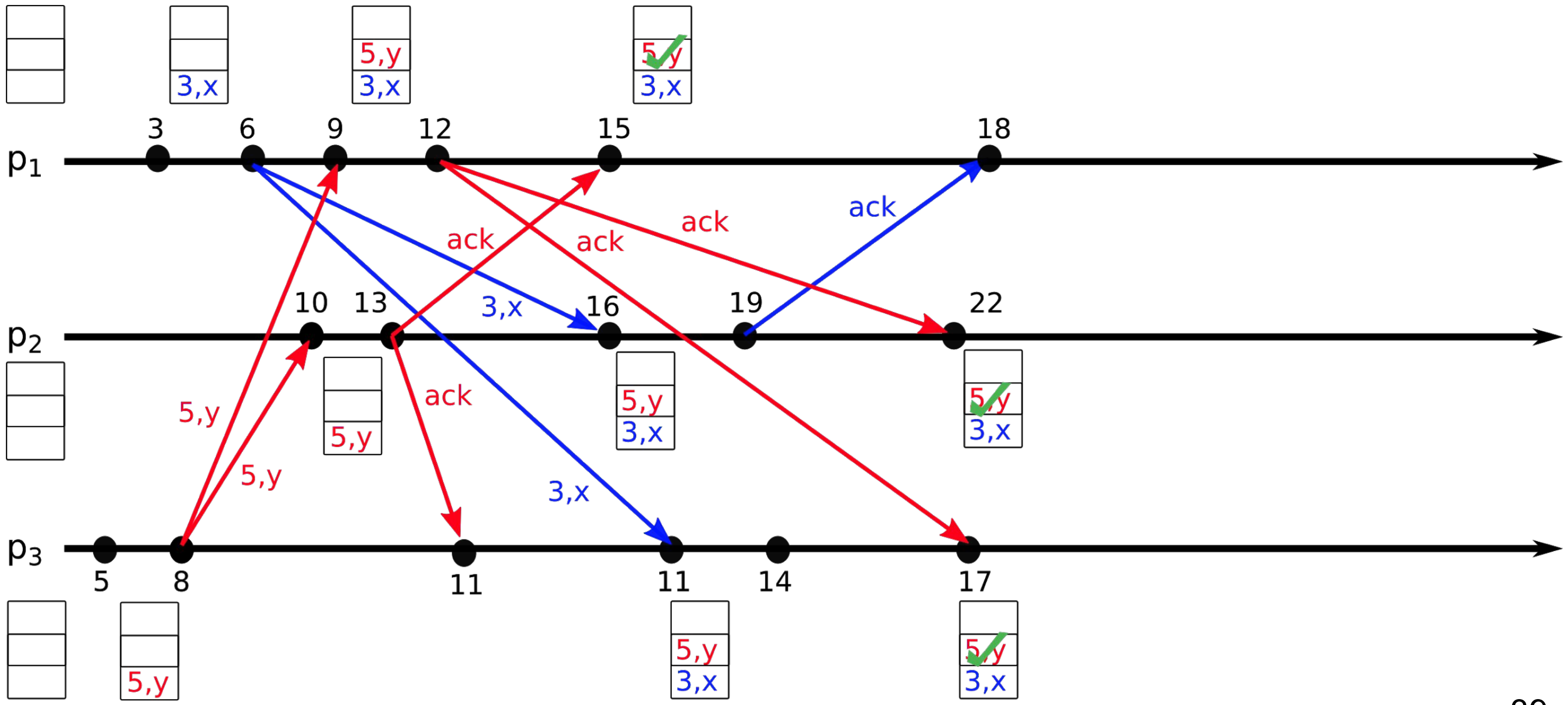
# Totally-Ordered Multicast

- Multicast messages + local timestamp-ordered queue
- Multicasts an ACK to all other processes
- Process only if *both* at queue head and ACK'ed



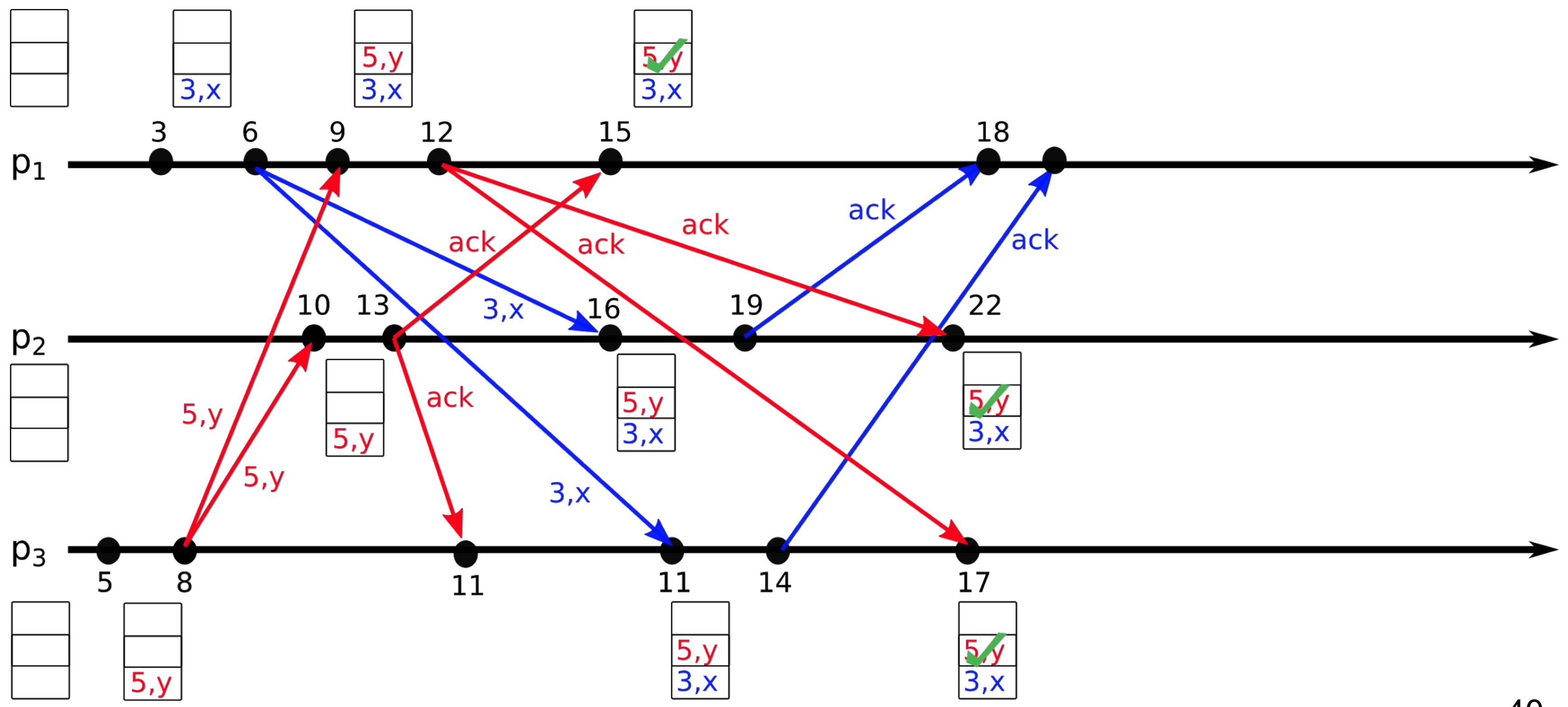
# Totally-Ordered Multicast

- Multicast messages + local timestamp-ordered queue
- Multicasts an ACK to all other processes
- Process only if *both* at queue head and ACK'ed



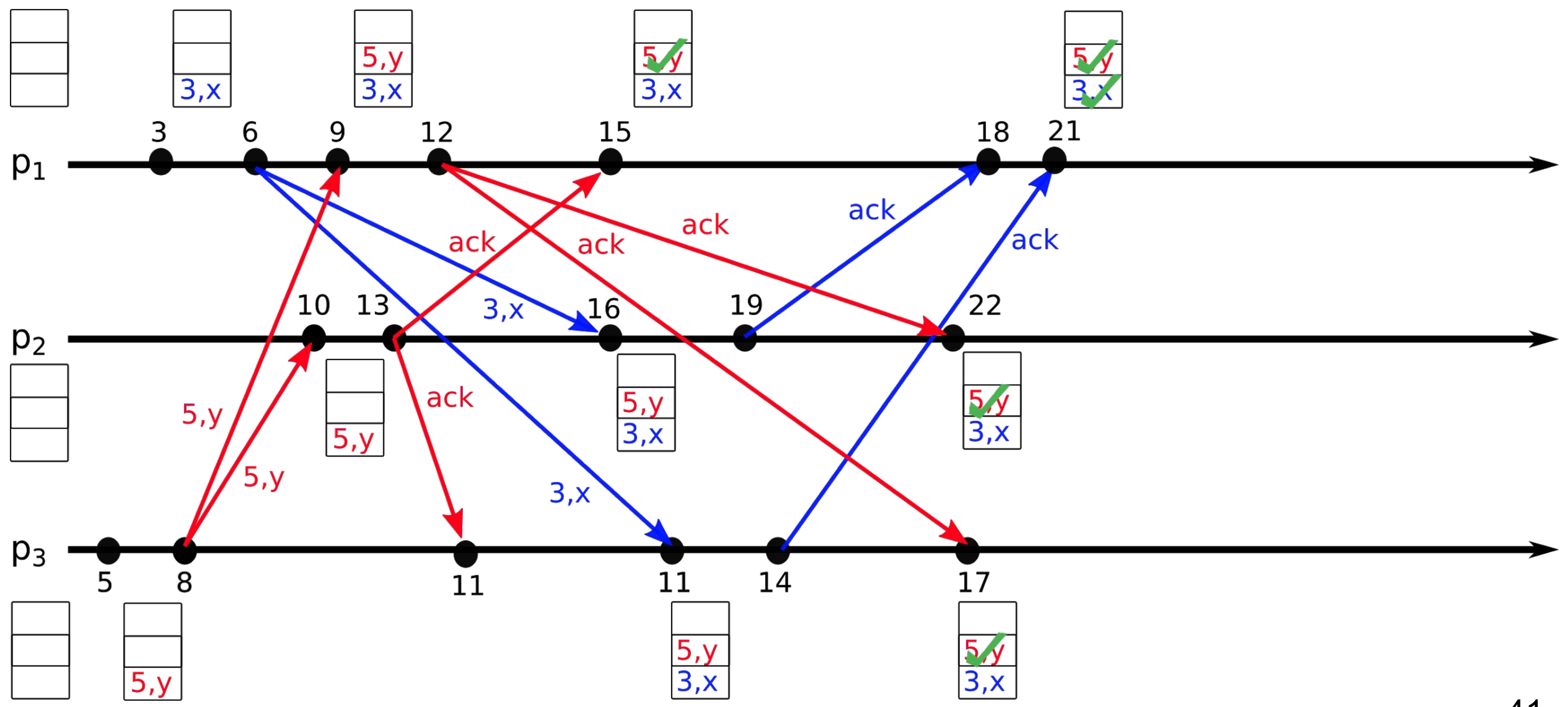
# Totally-Ordered Multicast

- Multicast messages + local timestamp-ordered queue
- Multicasts an ACK to all other processes
- Process only if *both* at queue head and ACK'ed



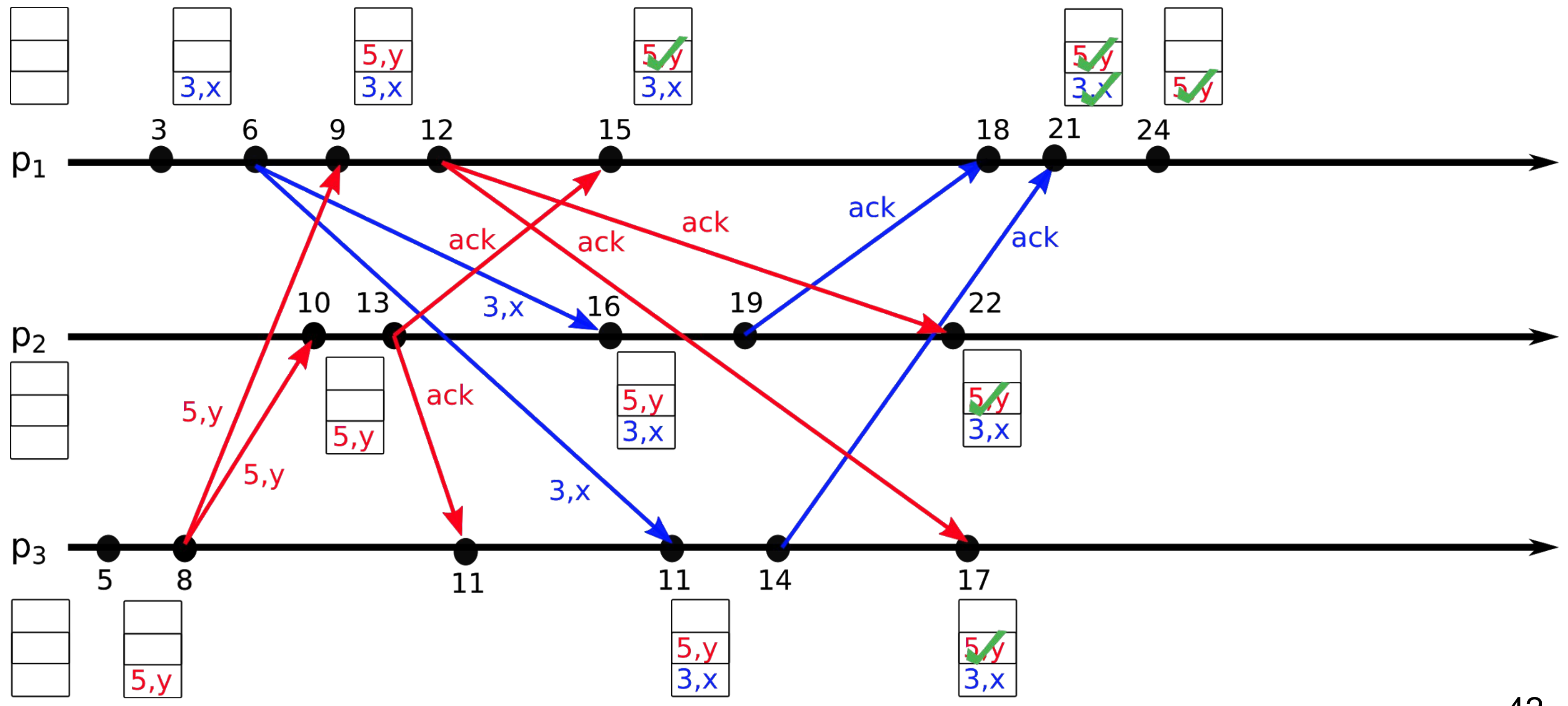
# Totally-Ordered Multicast

- Multicast messages + local timestamp-ordered queue
- Multicasts an ACK to all other processes
- Process only if *both* at queue head and ACK'ed



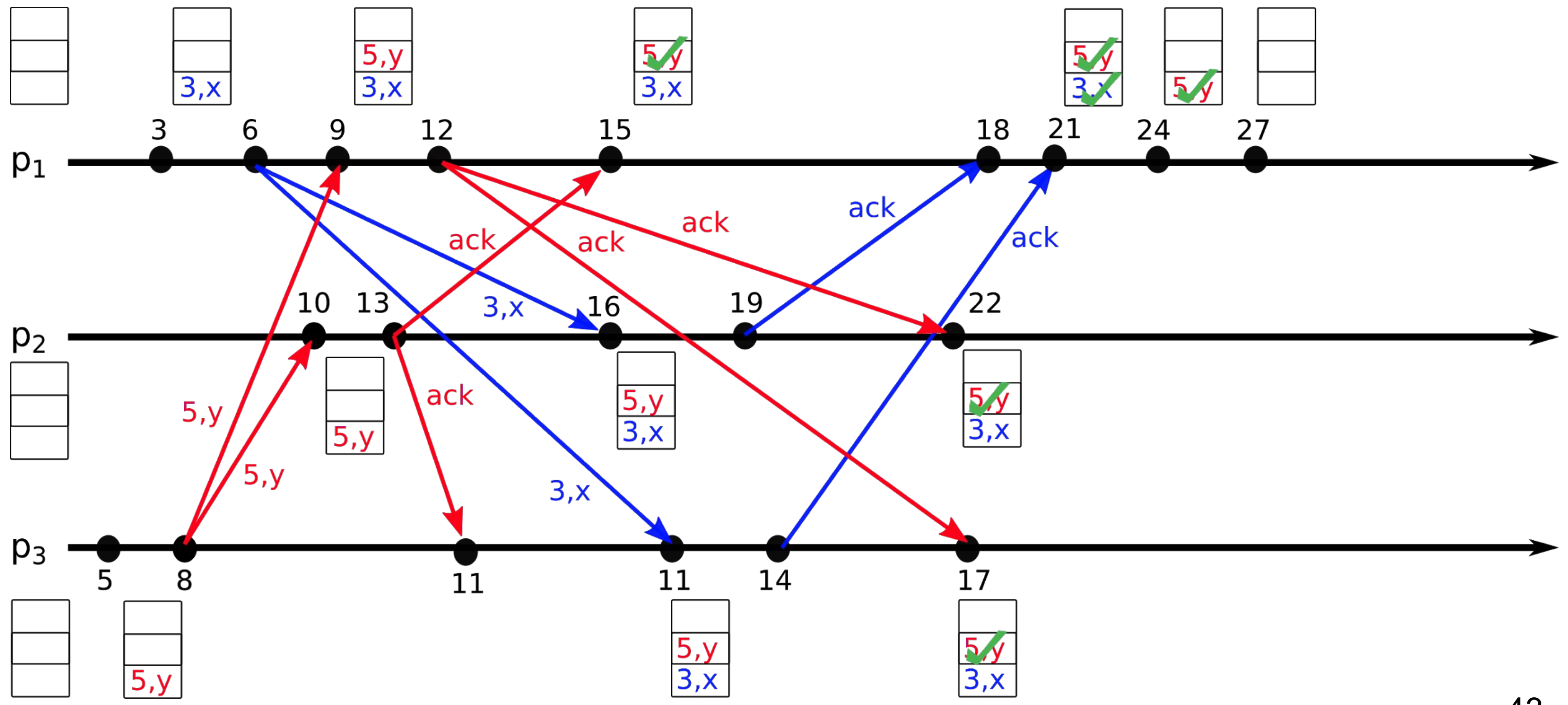
# Totally-Ordered Multicast

- Multicast messages + local timestamp-ordered queue
- Multicasts an ACK to all other processes
- Process only if *both* at queue head and ACK'ed



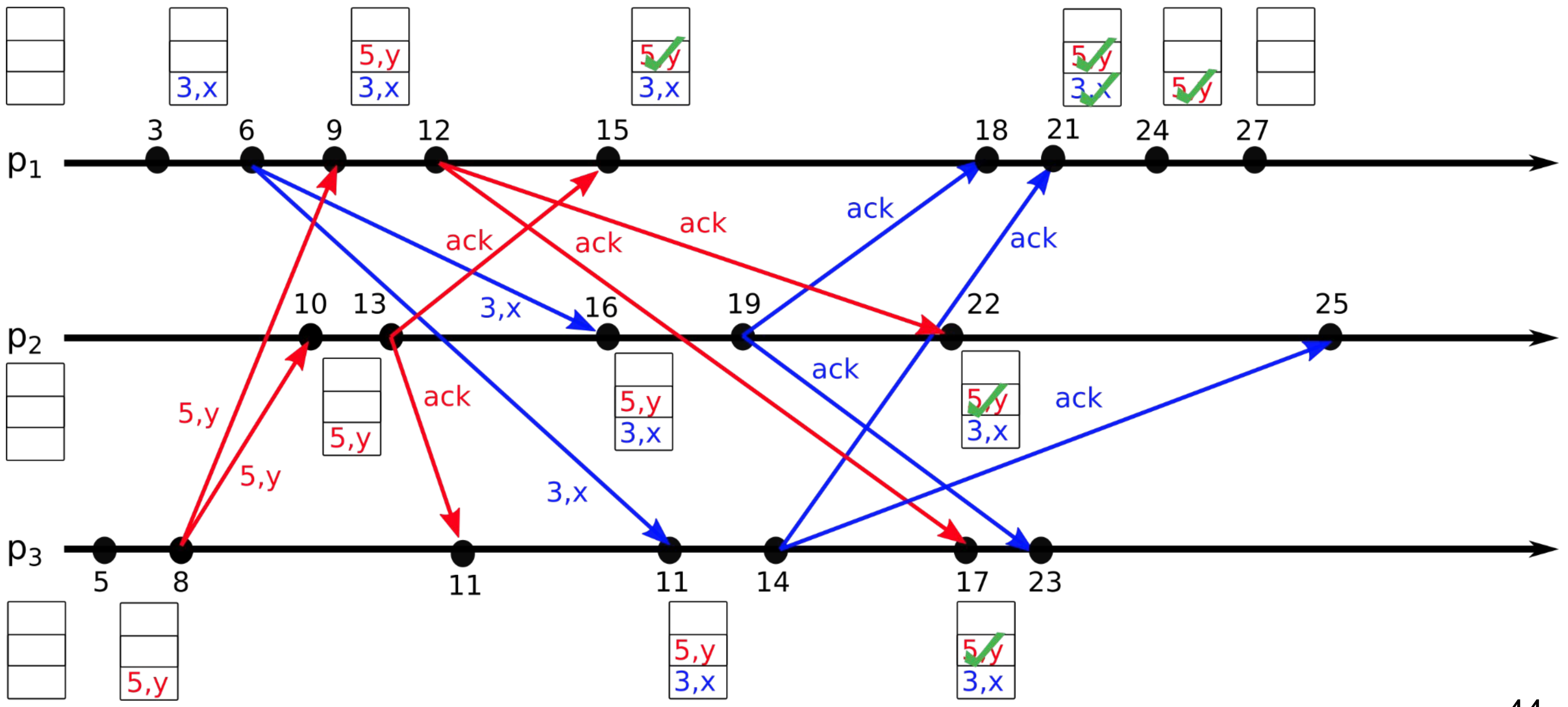
# Totally-Ordered Multicast

- Multicast messages + local timestamp-ordered queue
- Multicasts an ACK to all other processes
- Process only if *both* at queue head and ACK'ed



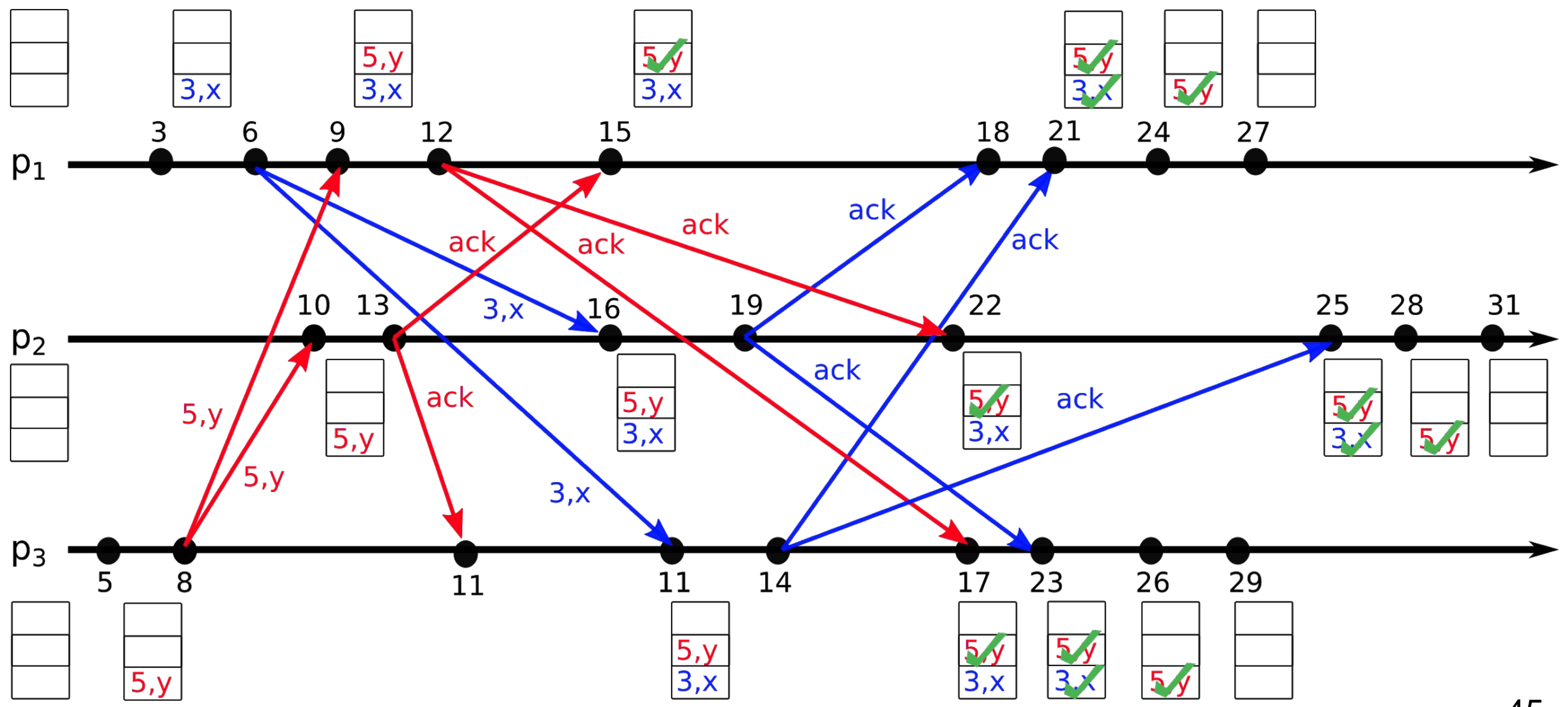
# Totally-Ordered Multicast

- Multicast messages + local timestamp-ordered queue
- Multicasts an ACK to all other processes
- Process only if *both* at queue head and ACK'ed



# Totally-Ordered Multicast

- Multicast messages + local timestamp-ordered queue
- Multicasts an ACK to all other processes
- Process only if *both* at queue head and ACK'ed



# Totally-Ordered Multicast: Summary

- Lamport Details (cont):
  - Local queue ordered according to timestamp.
  - The receiver multicasts an ACK to all other processes.
  - Wait unti head of queue and ack'ed by all participants
- Why does this work?
  - Key observation: by getting an ACK, we must have received all prior messages from this node
  - If that node had messages from before ACK, queue order will ensure correctness.
  - If that node has messages after ACK, their timestamp must be larger than the timestamp of the ACK
  - All processes will eventually have the same copy of the local queue → consistent global ordering.

## Goal of Today's Lecture

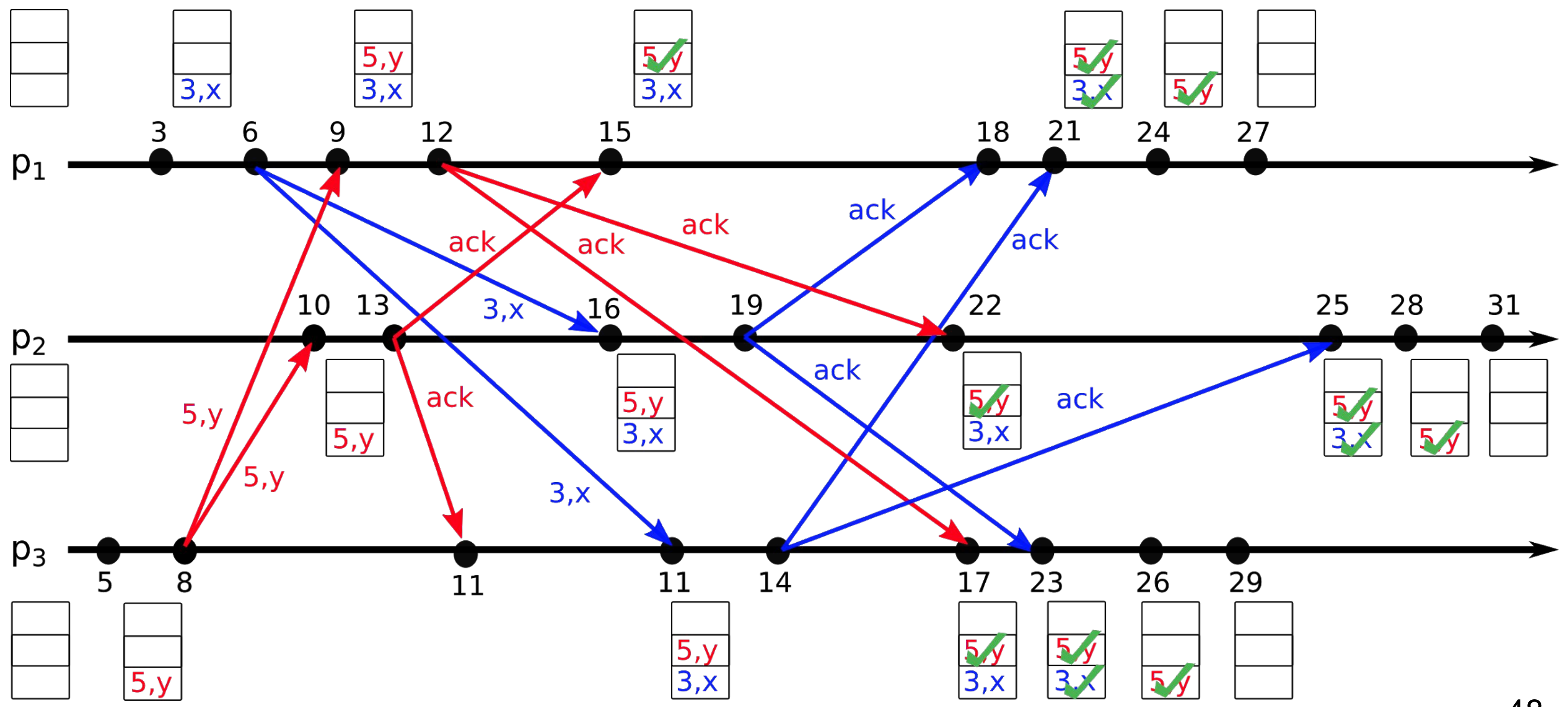
Understand trade-offs of main algorithms:

- Centralized Mutual Exclusion
- Bully Leader Election
- Decentralized Mutual Exclusion
- Totally-Ordered Multicast
- Lamport Mutual Exclusion
- Ricart & Agrawala Mutual Exclusion
- Token Ring Mutual Exclusion

# Lamport Mutual Exclusion (1)

- Based on Lamport TO-multicast
- ACK only to requestor (fewer messages)
- Release after finished (additional message)

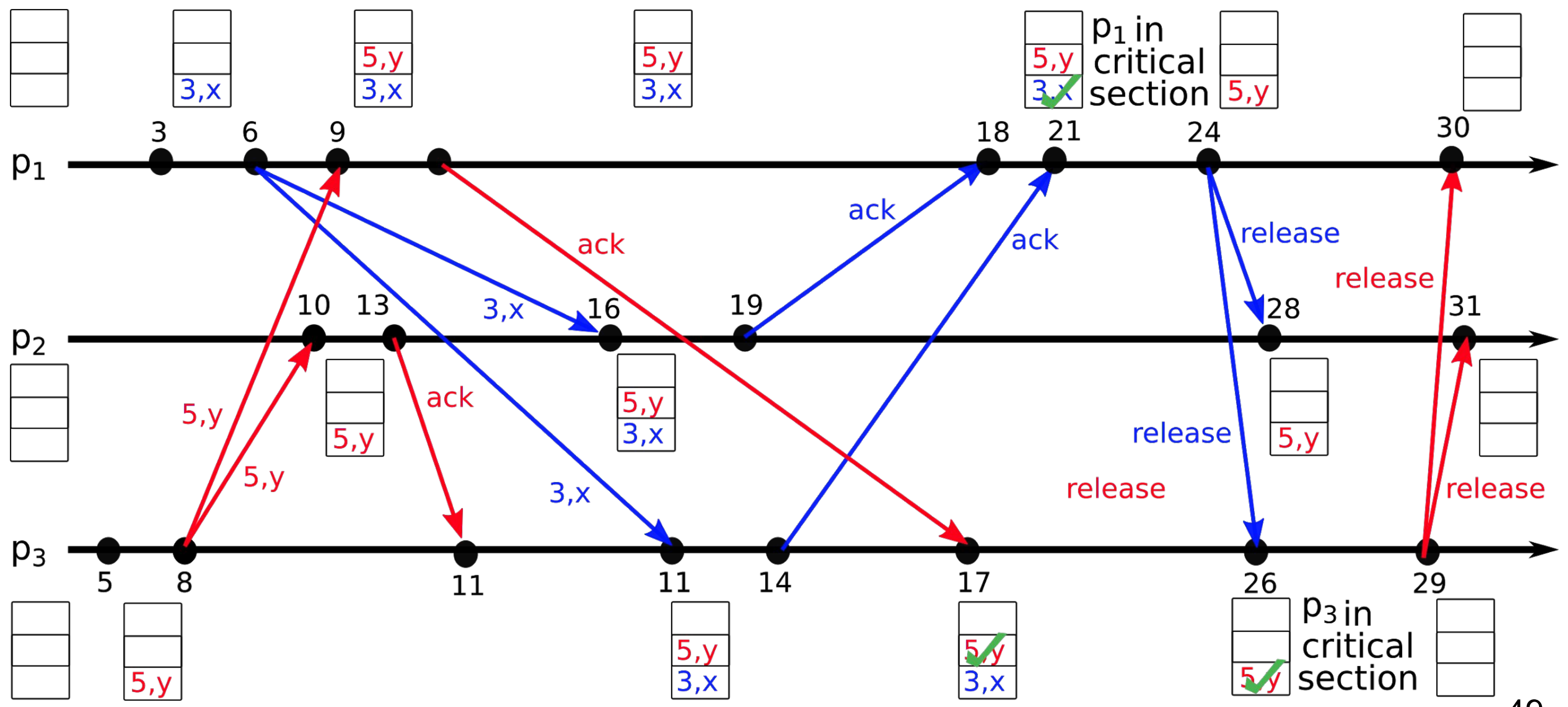
Here's the previous example again (TO Multicast):



## Lamport Mutual Exclusion (2)

- Based on Lamport TO-multicast ⇒ Simplified!
- ACK only to requestor
- Release after finished

Here's the example with Lamport Mutual Exclusion:



## More Details: Lamport Mutual Exclusion

- Every process maintains a queue of pending requests for entering critical section in order. The queues are ordered by virtual time stamps derived from Lamport timestamps
  - For any events e, e' such that e → e' (causality ordering), T(e) < T(e')
  - For any distinct events e, e', T(e) != T(e')

- When node i wants to enter C.S., it sends time-stamped request to all other nodes (including itself)
  - Wait for replies from all other nodes.
  - If own request is at the head of its queue and all replies have been received, enter C.S.
  - Upon exiting C.S., remove its request from the queue and send a release message to every process.

## More Details: Lamport Mutual Exclusion

- Other nodes:
  - After receiving a request, enter the request in its own request queue (ordered by time stamps) and reply with a time stamp.
    - This reply is unicast unlike the Lamport totally order multicast example. Why?
      - Only the requester needs to know the message is ready to commit.
      - Release messages are broadcast to let others to move on
  - After receiving release message, remove the corresponding request from its own request queue.
  - If own request is at the head of its queue and all replies have been received, enter C.S.

## Lamport Mutual Exclusion: Summary

- Correctness
  - When process x generates request with time stamp $T_x$, and it has received replies from all y in $N_x$, then its Q contains all requests with time stamps <= $T_x$
- Performance
  - Process i sends n-1 request messages
  - Process i receives n-1 reply messages
  - Process i sends n-1 release messages
- Issues
  - What if node fails?
  - Performance compared to centralized
  - What about message reordering?

## Goal of Today's Lecture

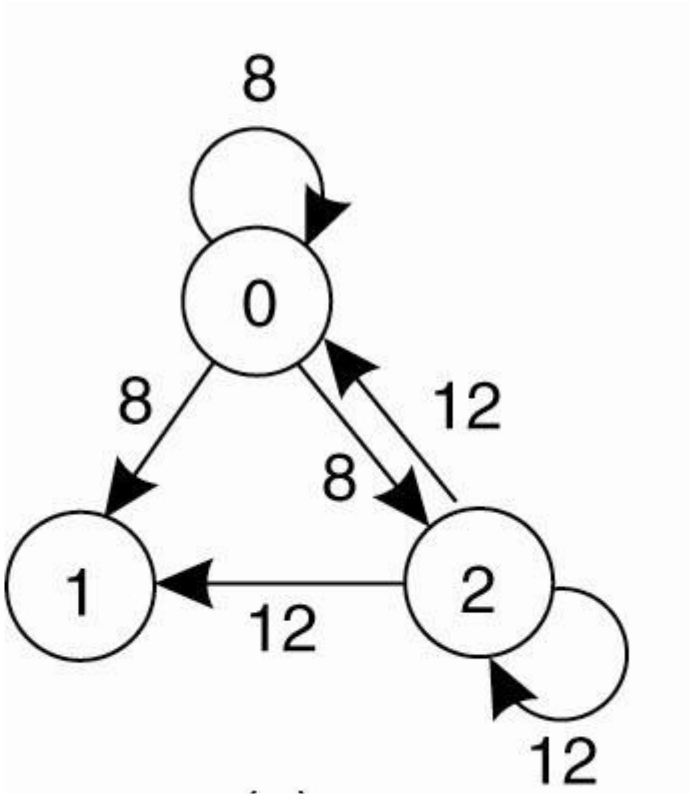
Understand trade-offs of main algorithms:

- Centralized Mutual Exclusion
- Bully Leader Election
- Decentralized Mutual Exclusion
- Totally-Ordered Multicast
- Lamport Mutual Exclusion
- Ricart & Agrawala Mutual Exclusion
- Token Ring Mutual Exclusion

## Ricart & Agrawala Mutex

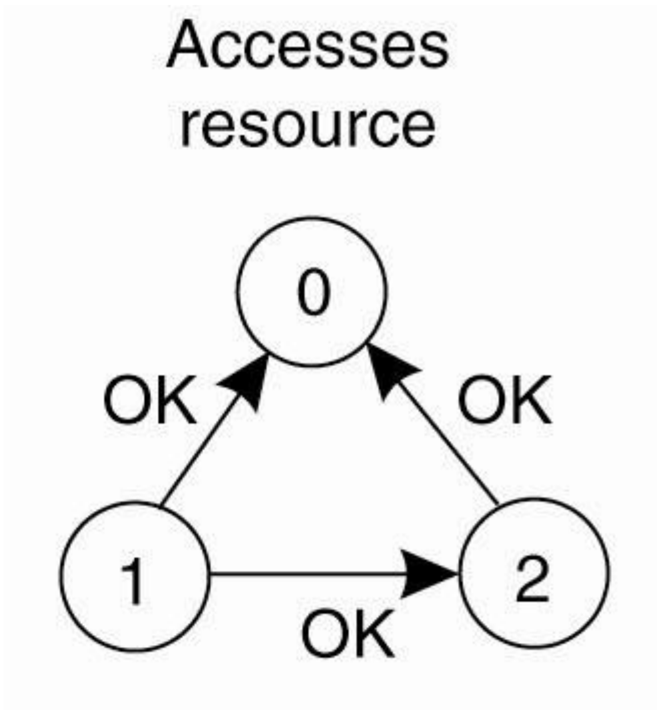
- Also relies on Lamport totally ordered clocks.
- When node i wants to enter C.S., it sends time-stamped request to all other nodes. These other nodes reply (eventually). When i receives n-1 replies, then can enter C.S.
- Trick: Node j having earlier request doesn't reply to i until after it has completed its C.S.

## Ricart & Agrawala Mutex



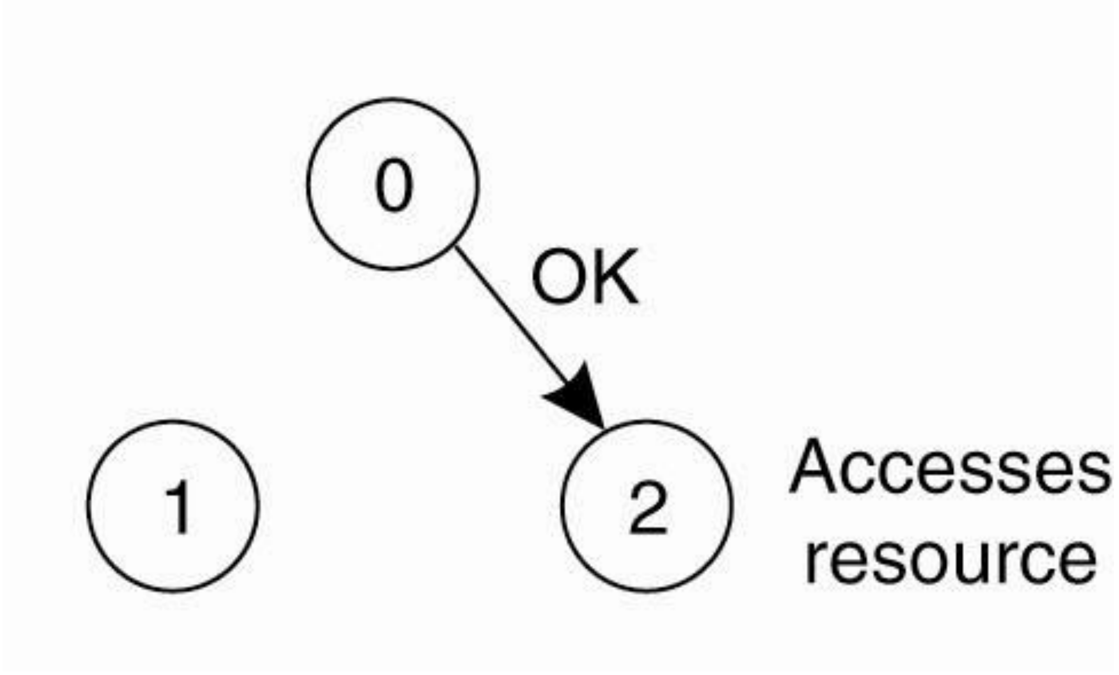
- Two processes (0 and 2) want to access a shared resource at the same moment.

## Ricart & Agrawala Mutex



- Process 0 has the lowest timestamp, so it wins.

## Ricart & Agrawala Mutex

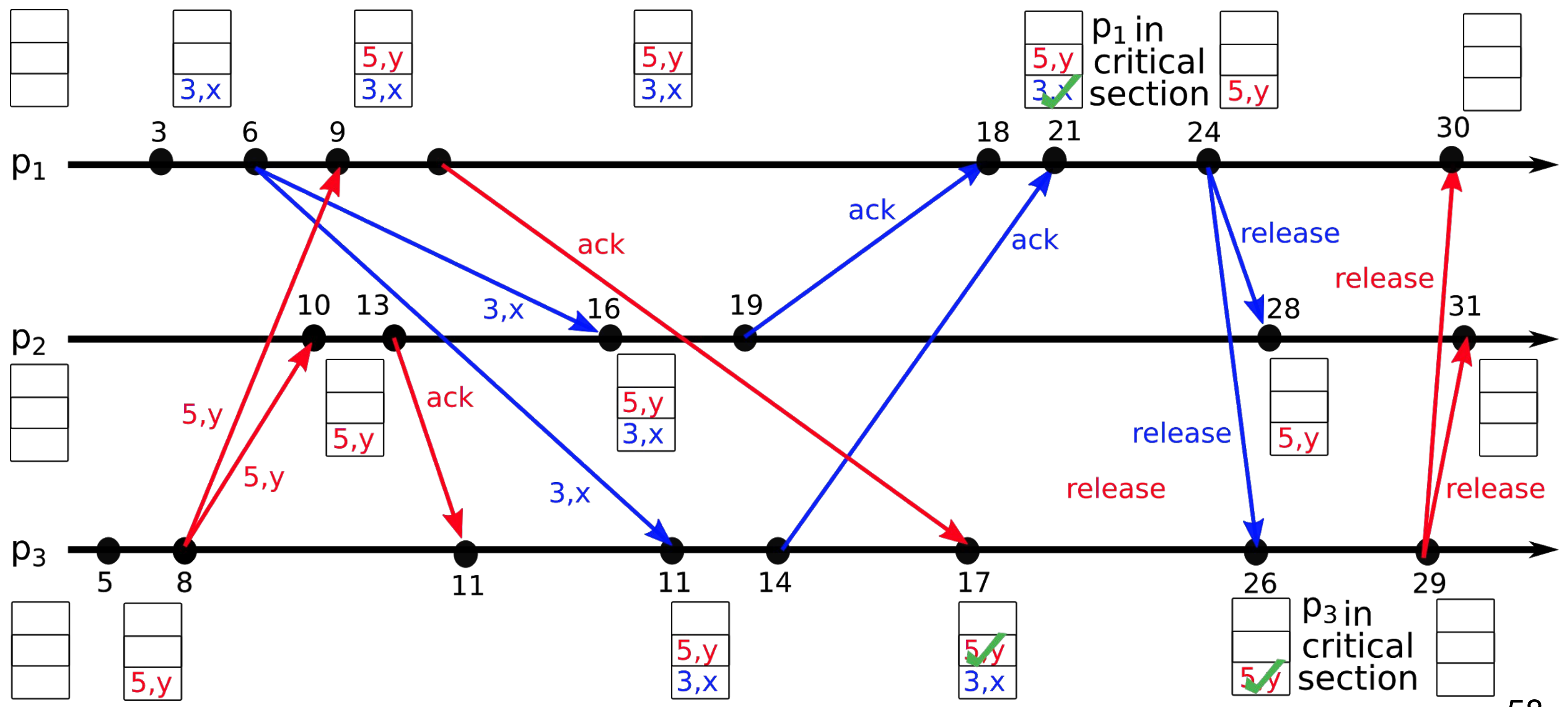


- When process 0 is done, it sends an OK also, so 2 can now go ahead.

# Ricart & Agrawala Mutex (1)

Optimization trick: Node j having earlier request doesn't reply to i until after it has completed its C.S.

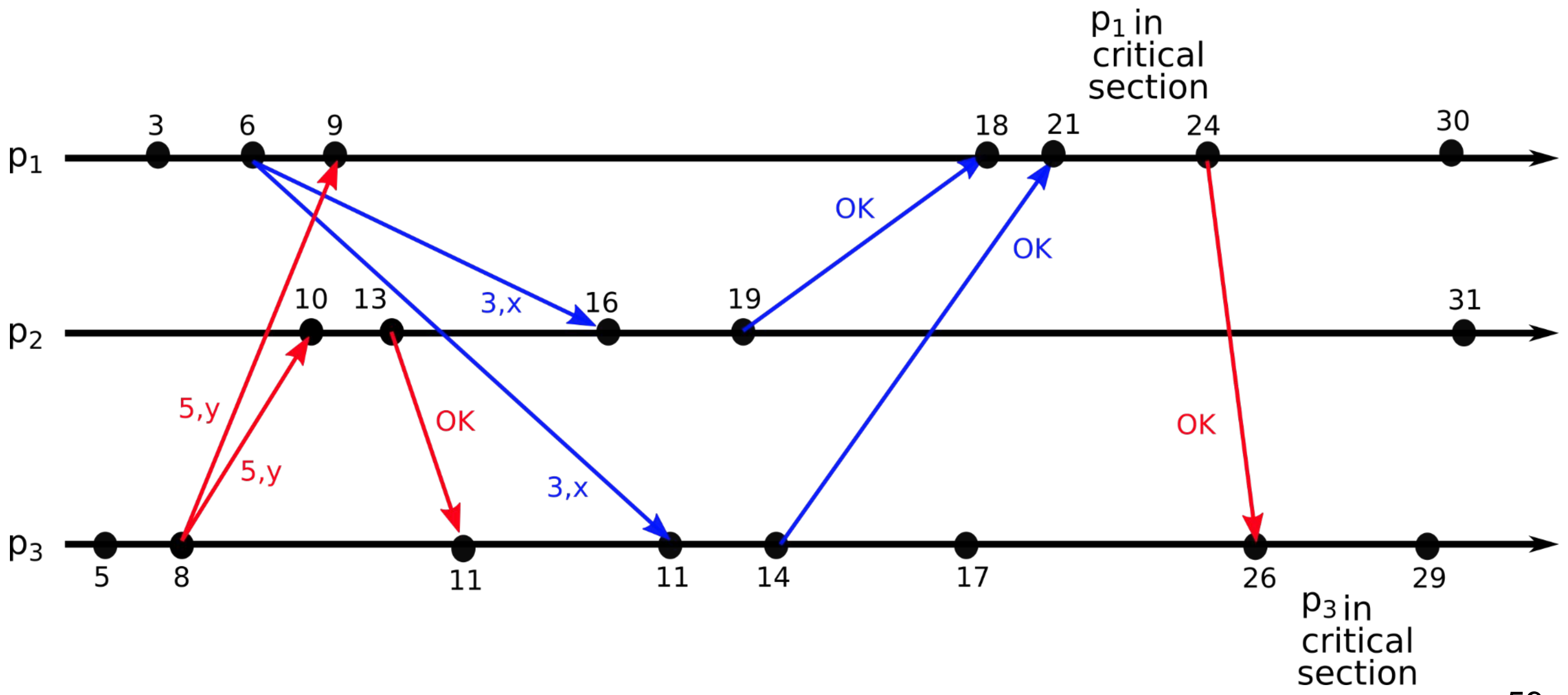
Here's the previous example again with Lamport Mutual Exclusion:



# Ricart & Agrawala Mutex (2)

Optimization trick: Node j having earlier request doesn't reply to i until after it has completed its C.S.

Here's the example with Ricard and Agrawala



## Ricart & Agrawala Mutex: Recap

Three different cases:

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

# Ricart & Agrawala: Summary

- Correctness

  Why correct? (Hint proof by contradiction)

- Performance

- Issues

# Ricart & Agrawala: Correctness

- Look at nodes A & B. Suppose both are allowed to be in their critical sections at same time.
  - A must have sent message (Request, A, $T_a$) & gotten reply (Reply, A).
  - B must have sent message (Request, B, $T_b$) & gotten reply (Reply, B).
- Case 1: One received request before other sent request.
  - E.g., B received (Request, A, $T_a$) before sending (Request, B, $T_b$). Then would have $T_a < T_b$. A would not have replied until after leaving its C.S.
- Case 2: Both sent requests before receiving others request.
  - But still, $T_a$ & $T_b$ must be ordered. Suppose $T_a < T_b$. Then A would not sent reply to B until after leaving its C.S.

## Ricart & Agrawala: Deadlock Free

- Cannot have cycle where each node waiting for some other
- Consider two-node case: Nodes A & B are causing each other to deadlock
  - This would result if A deferred reply to B & B deferred reply to A, but this would require both $T_a < T_b$ & $T_b < T_a$
- For general case, would have set of nodes A, B, C, ..., Z, such that A is holding deferred reply to B, B to C, ... Y to Z, and Z to A. This would require $T_a < T_b < ... < T_z < T_a$, which is not possible

## Ricart & Agrawala: Starvation Free

- If node makes request, it will be granted eventually
- Claim: If node A makes a request with time stamp $T_a$, then eventually, all nodes will have their local clocks > $T_a$
- Justification: From the request onward, every message A sends will have time stamp > $T_a$
  - All nodes will update their local clocks upon receiving those messages.
  - So, eventually, A's request will have a lower time stamp than any other node's request, and it will be granted.

## Ricart & Agrawala: Summary

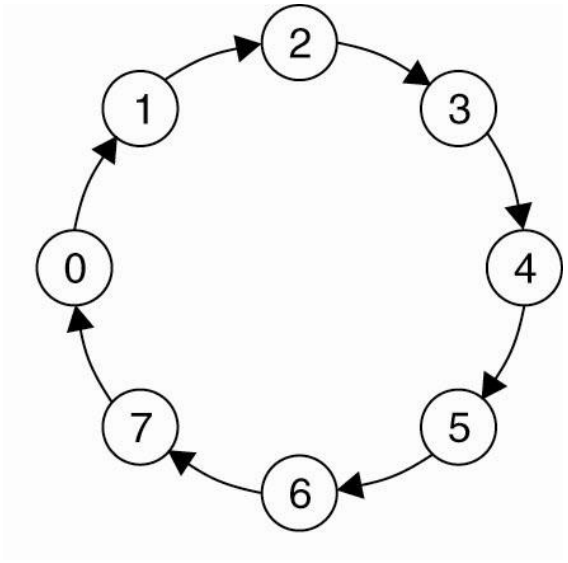
- Correctness
  - Case-based argument
  - Deadlock free
  - Starvation free
- Performance
  - Each cycle involves 2(n-1) messages
    - n-1 requests by I
    - n-1 replies to I
- Issues
  - What if node fails?
    Performance compared to centralized

## Goal of Today's Lecture

Understand trade-offs of main algorithms:

- Centralized Mutual Exclusion
- Bully Leader Election
- Decentralized Mutual Exclusion
- Totally-Ordered Multicast
- Lamport Mutual Exclusion
- Ricart & Agrawala Mutual Exclusion
- Token Ring Mutual Exclusion

# A Token Ring Algorithm



- Organize the processes involved into a logical ring
- One token at any time → passed from node to node along ring

# A Token Ring Algorithm

- Correctness:
  - Clearly safe: Only one process can hold token
- Fairness:
  - Will pass around ring at most once before getting access.
- Performance:
  - Each cycle requires between 1 - ∞ messages
  - Latency of protocol between 0 & n-1
- Issues
  - Lost token

# A Comparison of the 5 Mutex Algorithms

| Algorithm | # Messages per cycle | Delay before entry | Problems |
|---|---|---|---|
| Centralized | 3 | 2 | Coordinator crash |
| Decentralized | 2 m k + m, k≥1 | 2m | Starvation |
| Lamport | 3 (N-1) | 2 (N-1) | Crash of any process, inefficient |
| Ricart & Agrawala | 2 (N-1) | 2 (N-1) | Crash of any process |
| Token ring | 1 to infinite | 0 to (N-1) | Lost token, process crash |

- Which one would you choose?
- What happens with crashes?

## Summary

- Lamport algorithm demonstrates how distributed processes can maintain consistent replicas of a data structure (the priority queue).
- Ricart & Agrawala's algorithm demonstrate utility of logical clocks.
- Centralized & ring based algorithms have much lower message counts
- None of these algorithms can tolerate failed processes or dropped messages.