Announcements

- Project 1 (P1) is coming out this week (dates on course website)
  - P1 recitation next week

- We will do the partner matching for those who don't have partners yet
  - Make sure your team has been declared via the survey

- HW1 is coming out today

- For everyone's safety:
  - Please do not congregate after the class for Q/A -- ask questions during the lecture or make use of Piazza and office hours
  - If you are sick, please watch the lecture remotely

- For any private communication, use course staff email < ds-staff-f21-private@lists.andrew.cmu.edu>. Not individual instructor email addresses!

15-440/15-640 Distributed Systems

Distributed Mutual Exclusion

Mutual Exclusion

- Must ensure that only one instance of code is in critical section

```
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 Critical Section 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

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 from different senders may be reordered

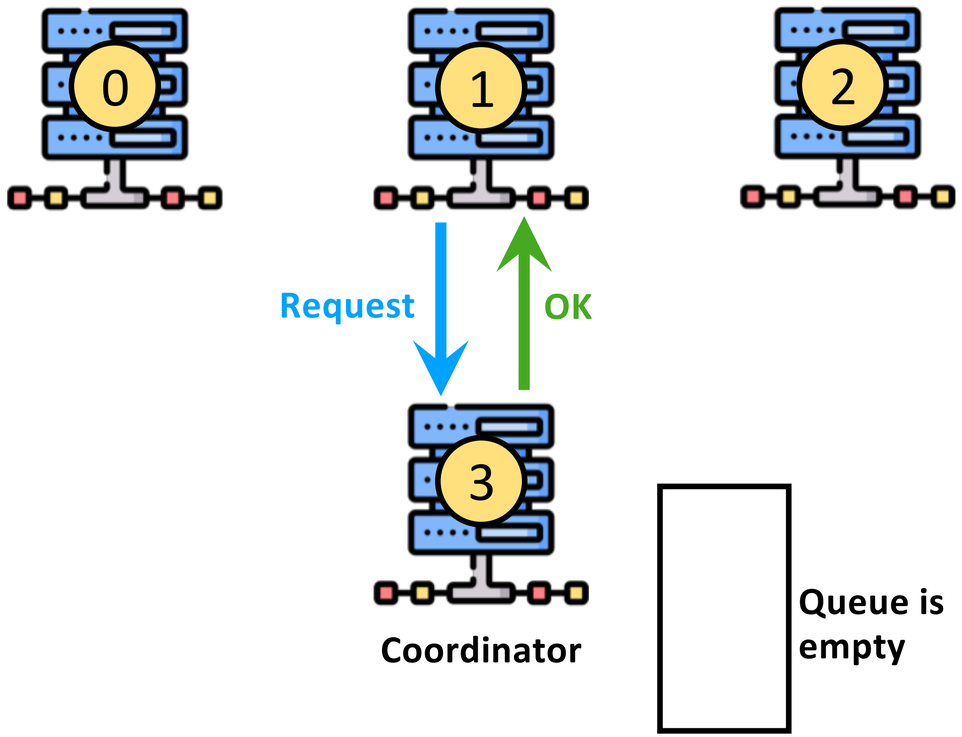
Outline

- Centralized Mutual Exclusion
  - Bully algorithm for leader election
- Decentralized Mutual Exclusion
- Distributed Mutual Exclusion
  - Totally-Ordered Multicast
  - Lamport Mutual Exclusion
  - Ricart & Agrawala Mutual Exclusion
  - Token Ring Mutual Exclusion

Outline

- Centralized Mutual Exclusion
  - Bully algorithm for leader election
- Decentralized Mutual Exclusion
- Distributed Mutual Exclusion
  - Totally-Ordered Multicast
  - Lamport Mutual Exclusion
  - Ricart & Agrawala Mutual Exclusion
  - Token Ring Mutual Exclusion

Centralized Algorithm (1)



@ Server:

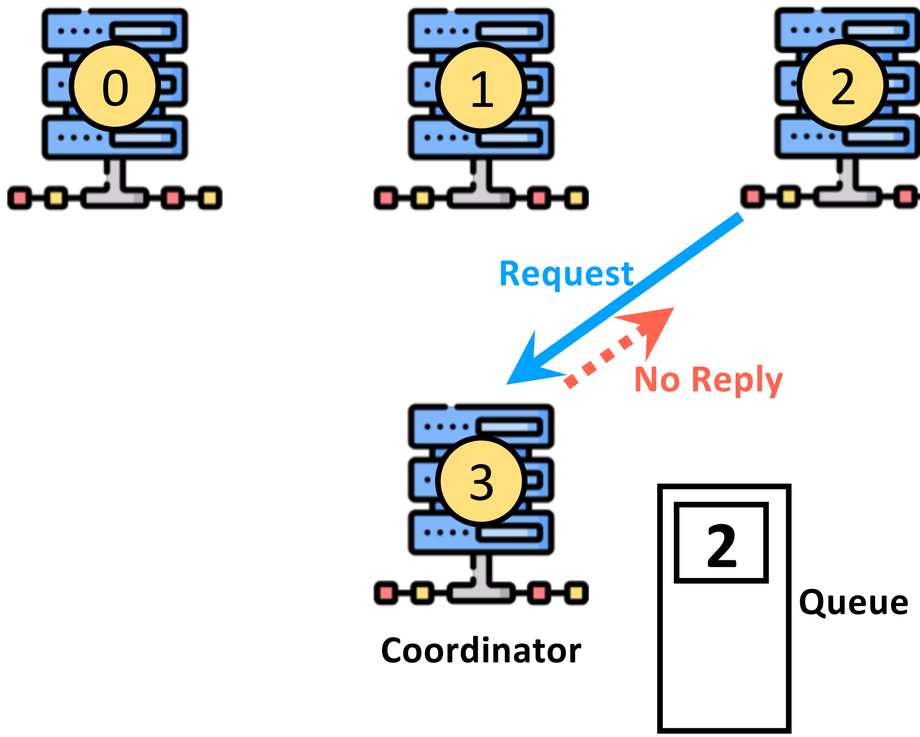
```
while true:
  m = Receive()
  If m == (Request, i):
    If Available():
      Send (Grant) to i
```

@ Client → Acquire:

Send (Request, i) to coordinator

Wait for reply

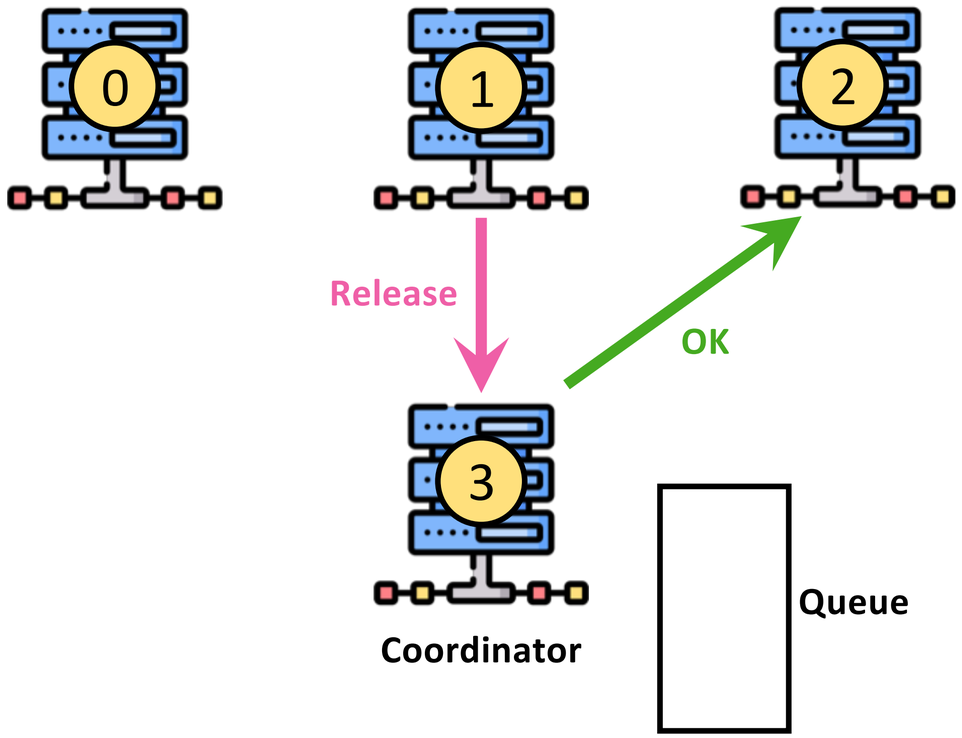
Centralized Algorithm (2)



@ Server:

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

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 → Acquire:
Send (Release) to coordinator

Centralized Algorithm: Summary

- **Correctness:**
  - Clearly safe
  - Fairness depends on queuing policy
    - *Example of an unfair policy?*
- **Performance:**
  - "**cycle**" is a complete round of the protocol with one process *i* requesting access, entering the 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?

Q: What can we do when the coordinator crashes?

Outline

- Centralized Mutual Exclusion
  - Bully algorithm for leader election
- Decentralized Mutual Exclusion
- Distributed Mutual Exclusion
  - Totally-Ordered Multicast
  - Lamport Mutual Exclusion
  - Ricart & Agrawala Mutual Exclusion
  - Token Ring Mutual Exclusion

Selecting a Leader (Elections)

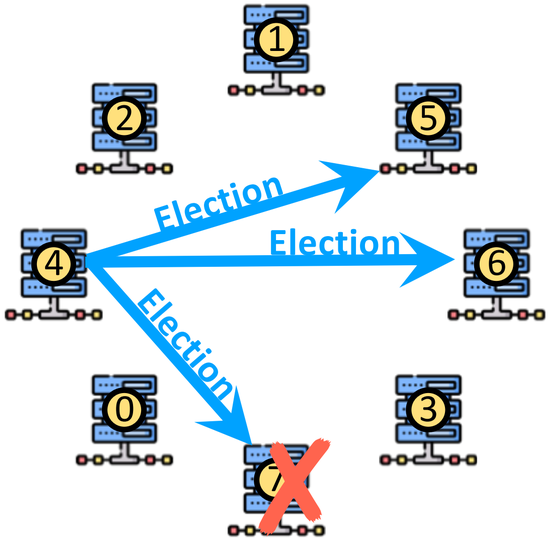
- Select a **unique** process as the leader
- It does not matter which process becomes the leader (all processes identical)
- In general, goal is to identify the process with the largest identifier

Stage 1: Process P notices that leader has failed

Stage 2 (Election Algorithm) The Bully Algorithm

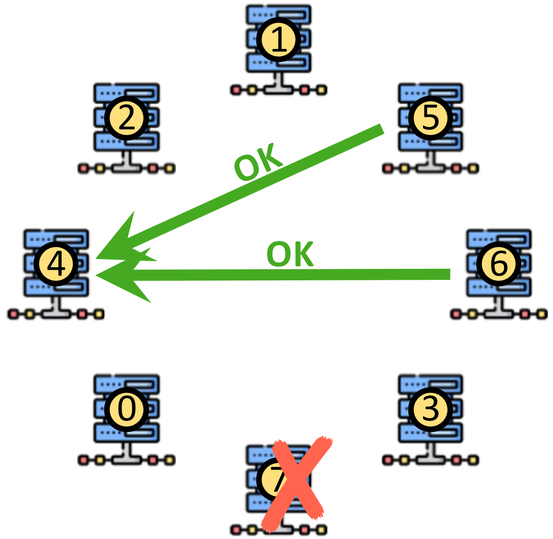
1. *P* sends an *ELECTION* message to all processes with higher numbers.
2. If no one responds, *P* wins the election and becomes coordinator.
3. If one of the higher-ups answers, it takes over. *P*'s job is done.

The Bully Leader-Election Algorithm (1)



(a)

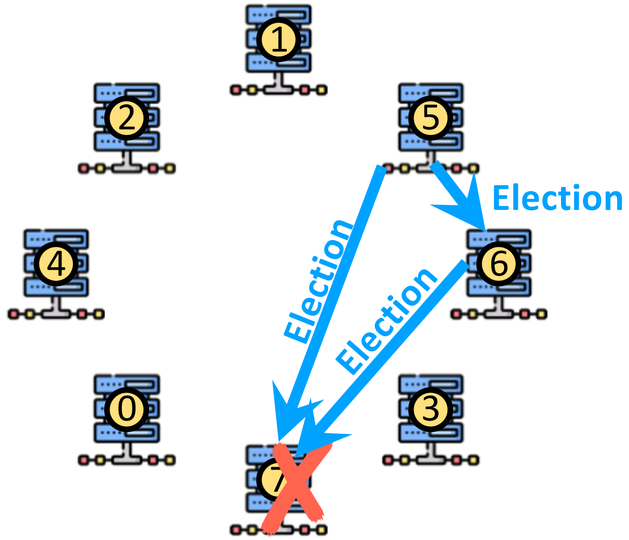
Process 4 holds an election



Previous coordinator has crashed

(b)

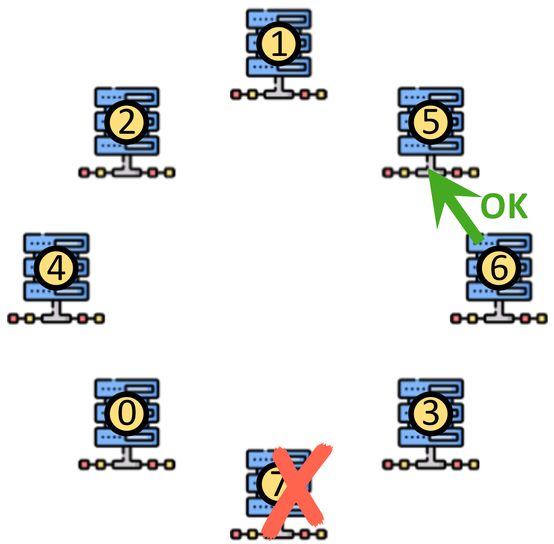
Processes 5 and 6 respond, telling 4 to stop



(c)

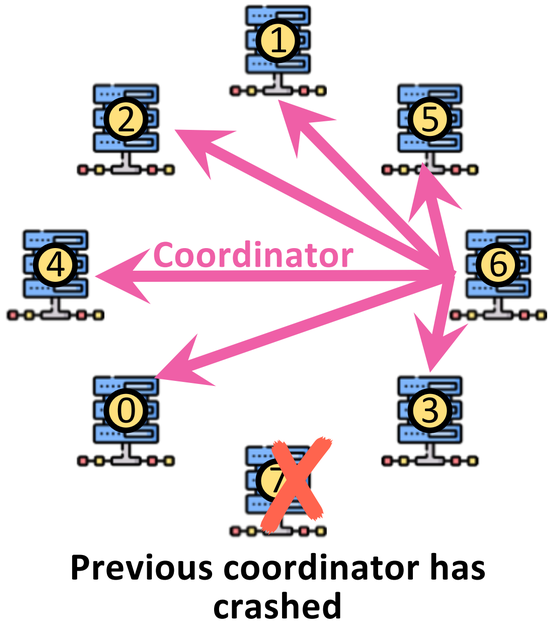
Now 5 and 6 each hold an election

The Bully Leader-Election Algorithm (2)



(a)

Process 6 tells 5 to stop



(b)

Process 6 wins and tells everyone.

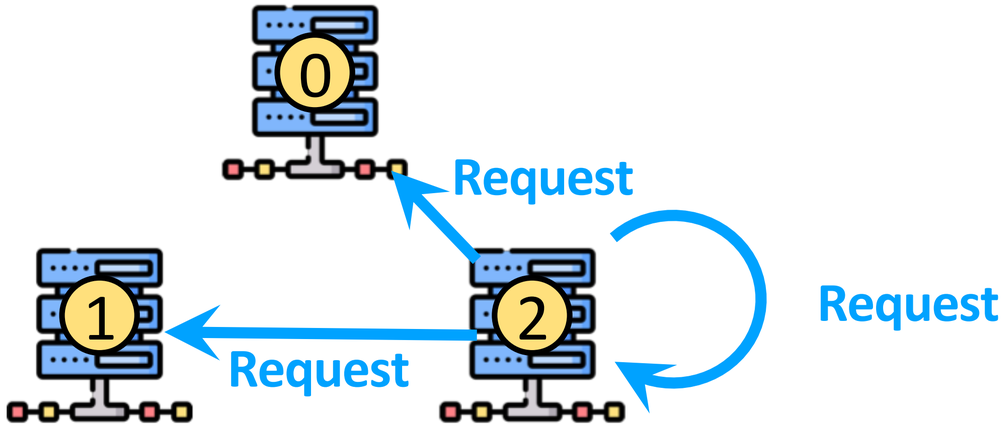
Outline

- Centralized Mutual Exclusion
  - Bully algorithm for leader election
- Decentralized Mutual Exclusion
- Distributed Mutual Exclusion
  - Totally-Ordered Multicast
  - Lamport Mutual Exclusion
  - Ricart & Agrawala Mutual Exclusion
  - Token Ring Mutual Exclusion

Decentralized Algorithm (1)

Opposite extreme to centralized algorithm

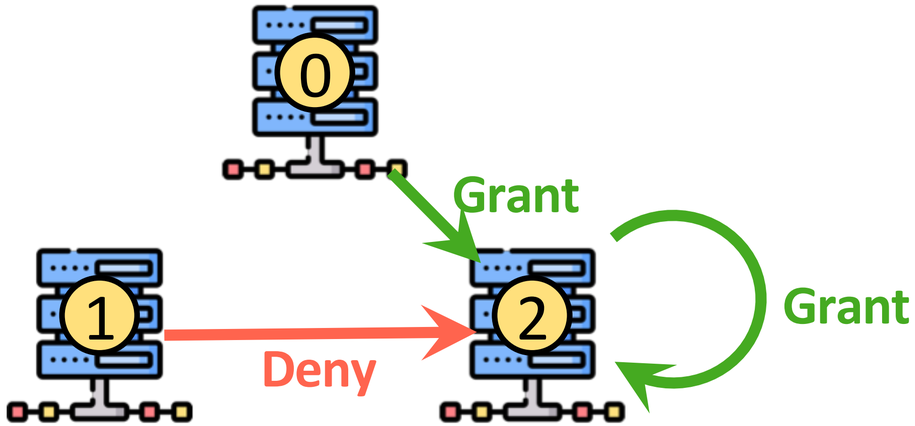
- **Assume that there are n coordinators**
  - Get a majority vote from *m > n/2* coordinators
  - A coordinator replies immediately to a request with *GRANT* or *DENY*



Decentralized Algorithm (2)

Opposite extreme to centralized algorithm

- **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 acces can affect availability
- Starvation!

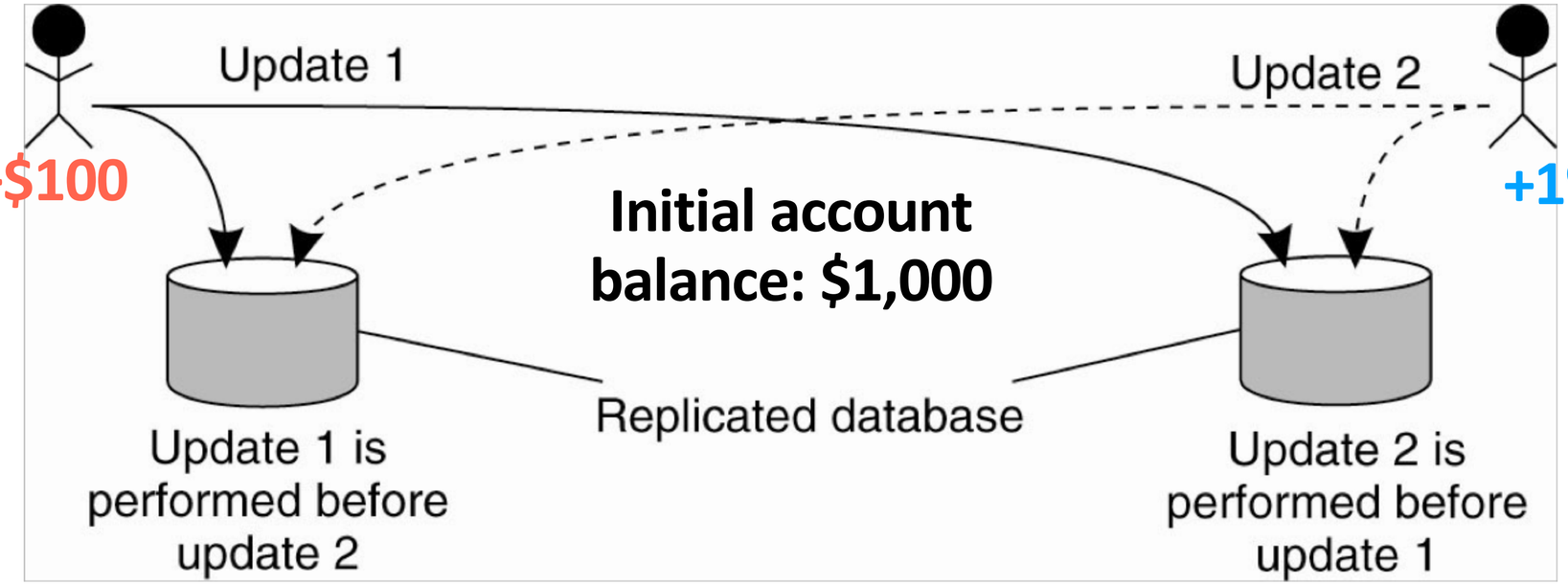
Decentralized Algorithm: Summary

- **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

Outline

- Centralized Mutual Exclusion
  - Bully algorithm for leader election
- Decentralized Mutual Exclusion
- Distributed Mutual Exclusion
  - Totally-Ordered Multicast
  - Lamport Mutual Exclusion
  - Ricart & Agrawala Mutual Exclusion
  - Token Ring Mutual Exclusion

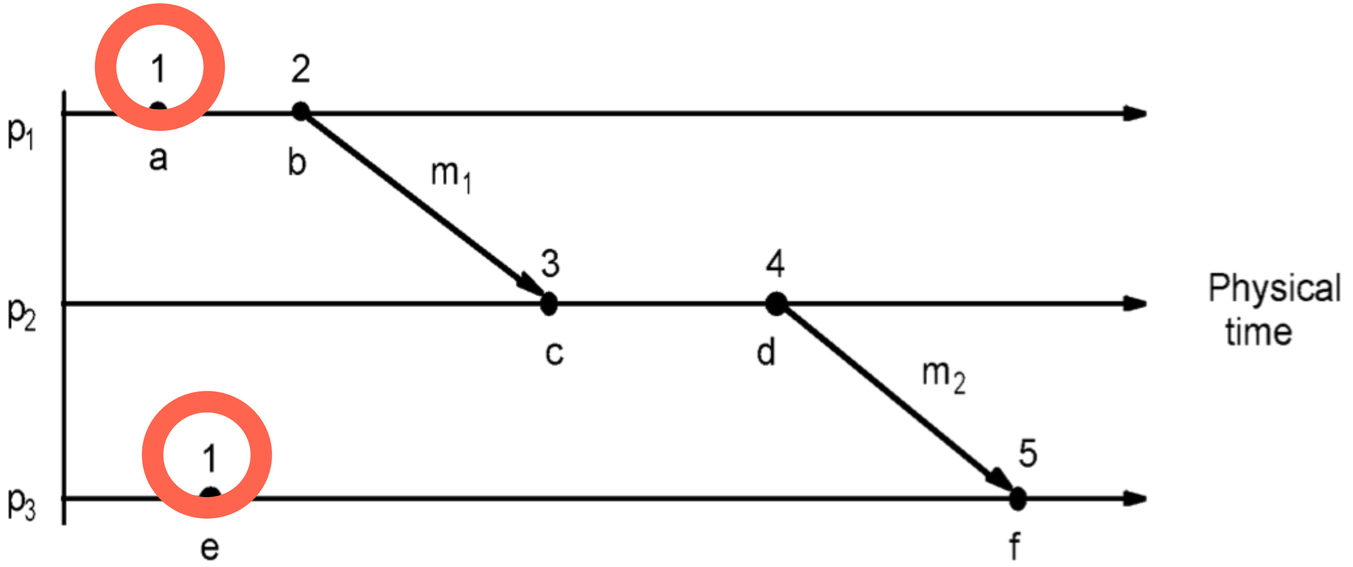
A Total Order Would be Useful



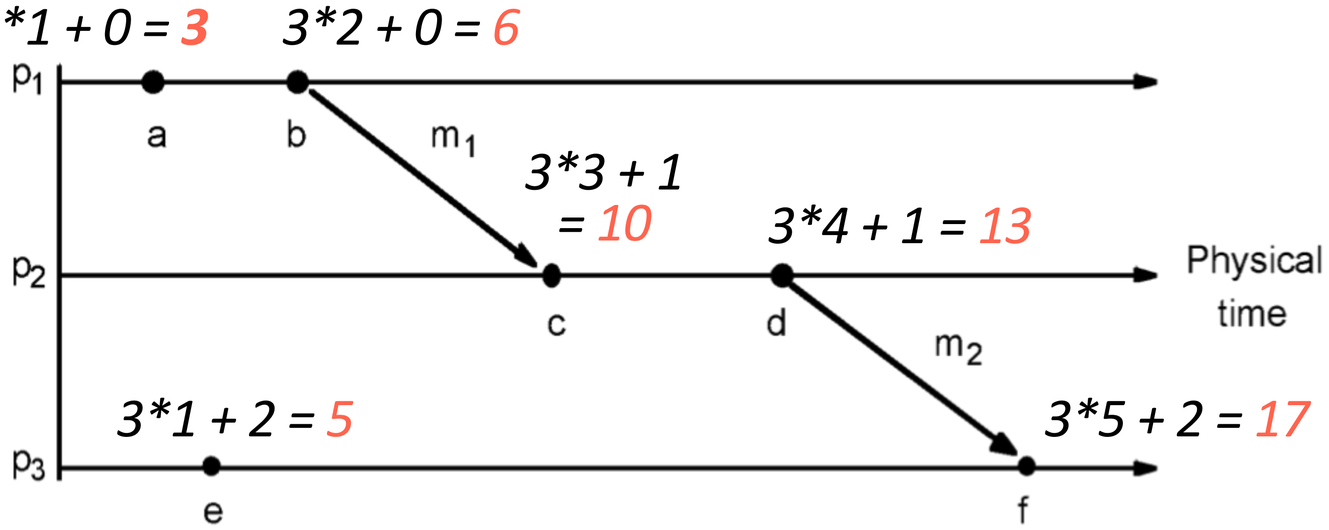
- Can use Lamport's to totally order

RECALL FROM LAST TIME

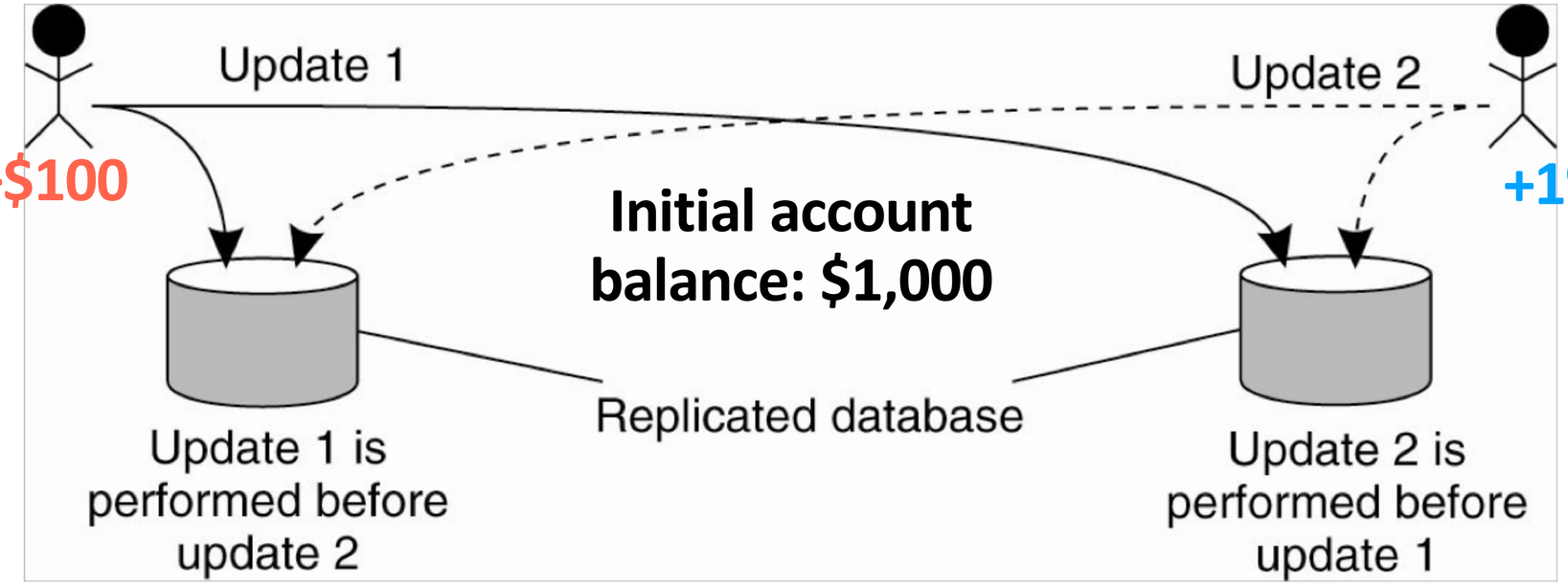
Logical Lamport Clocks



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



A Total Order Would be Useful



- **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?**

Totally-Ordered Multicast

- A **multicast** operation by which all messages are delivered in the same order to each receiver.
  - *Distributed data structure (priority queue)*

Algorithm in a nutshell

1. **Message to be sent is timestamped with sender's logical time**
2. **Message is multicast** *(including to the sender itself)*
3. **When message is received**
   a) It is put into local queue
   b) Ordered according to timestamp
   c) Receiver multicasts acknowledgement

(Assume all messages sent by one sender are received in the order they were sent and that no messages are lost)

Totally-Ordered Multicast

Algorithm using TO-Lamport Clocks

1. **Message to be sent is timestamped with sender's logical time**
2. **Message is multicast** *(including to the sender itself)*
3. **When message is received**
   a) It is put into local queue
   b) Ordered according to timestamp,
   c) Receiver multicasts acknowledgement

But when does the banking application get the message?

Totally-Ordered Multicast

Algorithm in a nutshell

1. **Message to be sent is timestamped with sender's logical time**
2. **Message is multicast** *(including to the sender itself)*
3. **When message is received**
   a) It is put into local queue
   b) Ordered according to timestamp,
   c) Receiver multicasts acknowledgement

But when does the banking application get the message?

Message is delivered to applications only when

- It is at head of queue
- It has been acknowledged by all involved processes

Totally-Ordered Multicast: Summary

- **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.

Outline

- Centralized Mutual Exclusion
  - Bully algorithm for leader election
- Decentralized Mutual Exclusion
- Distributed 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)*

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**

On the sending/requestor side:

- **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

On the receiving/not-requesting side:

- **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 $\leq T_x$
- **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?

Outline

- Centralized Mutual Exclusion
  - Bully algorithm for leader election
- Decentralized Mutual Exclusion
- Distributed Mutual Exclusion
  - Totally-Ordered Multicast
  - Lamport Mutual Exclusion
  - Ricart & Agrawala Mutual Exclusion
  - Token Ring Mutual Exclusion

Ricart & Agrawala Mutual Exclusion

Also, requires a total ordering of all events in the system:
therefore, relies on Lamport totally ordered clocks.

Algorithm Overview:

- 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 Mutual Exclusion

On the receiving processes end, 3 possible cases:

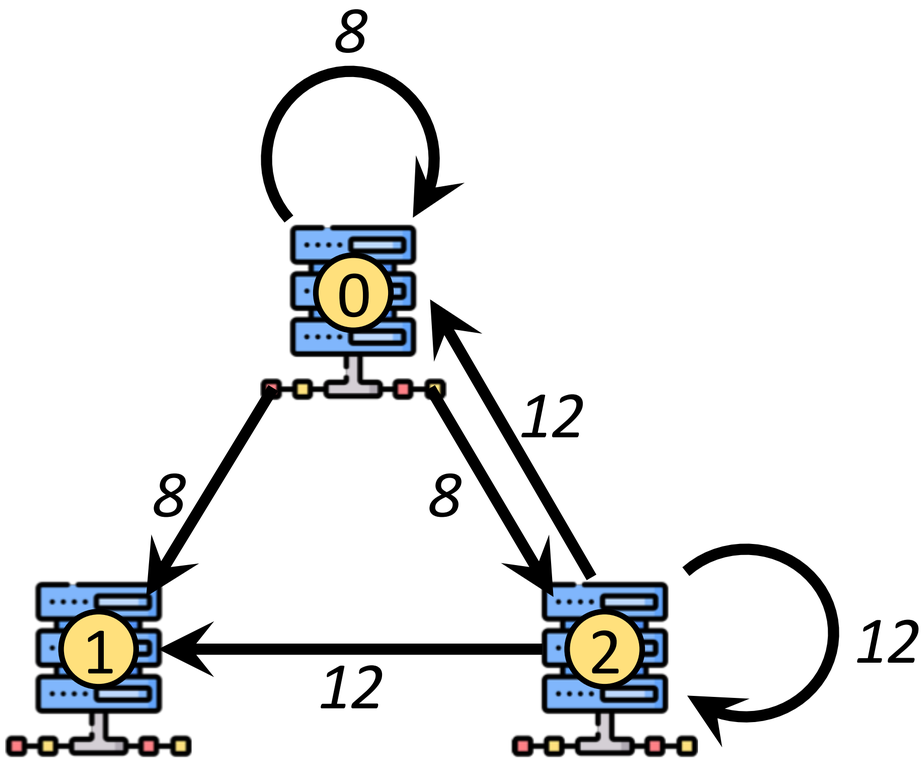
1. **Receiver uninterested in resource.** If the receiver is not accessing the resource and does not want to access it, it sends back an OK message to the sender.

2. **Receiver already has access to resource.** If the receiver already has access to the resource, it simply does not reply. Instead, it queues the request.

3. **Receiver wants access to resource, but doesn't have it yet.** 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. If the incoming message has a lower timestamp, the receiver sends back an OK message. If its own message has a lower timestamp, the receiver queues the incoming request and sends nothing.

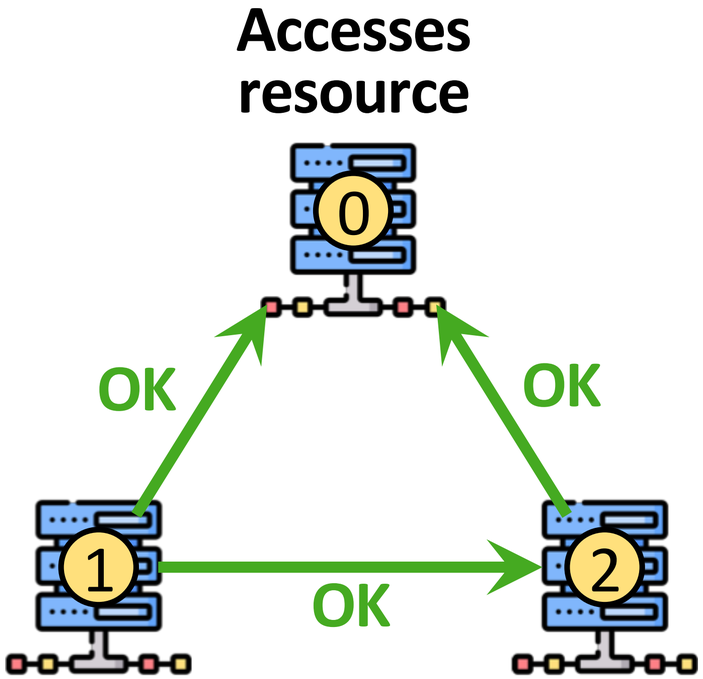
Lowest timestamp wins!

Ricart & Agrawala Mutual Exclusion



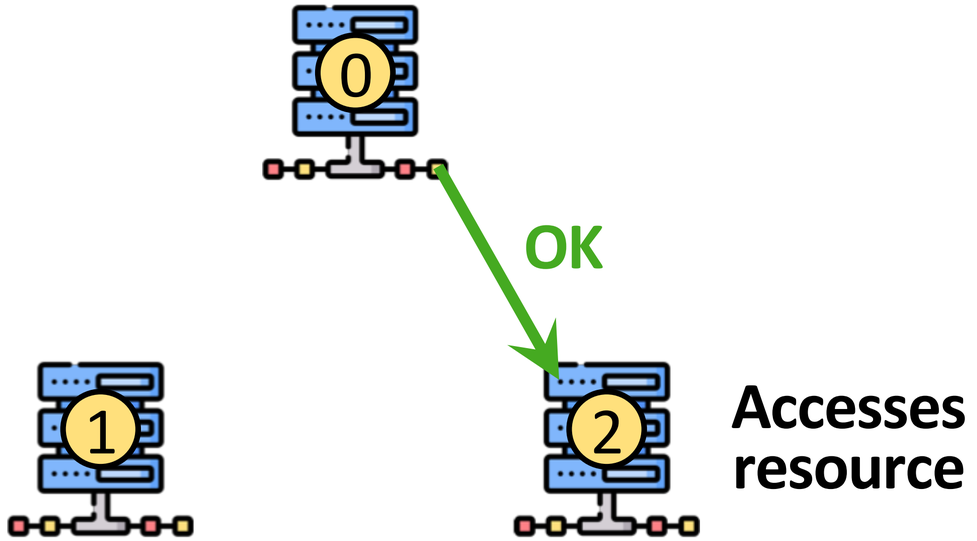
Two processes (0 and 2) want to access a shared resource at the same moment.

Ricart & Agrawala Mutual Exclusion



Process 0 has the lowest timestamp, so it wins.

Ricart & Agrawala Mutual Exclusion



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

Ricart & Agrawala Mutex Summary

- **Correctness:**

  Why is it 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 $T_a < T_b < \ldots < T_z < T_a$, 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 Mutex Summary

- **Correctness:**
  - Case-based argument
  - Deadlock free
  - Starvation free

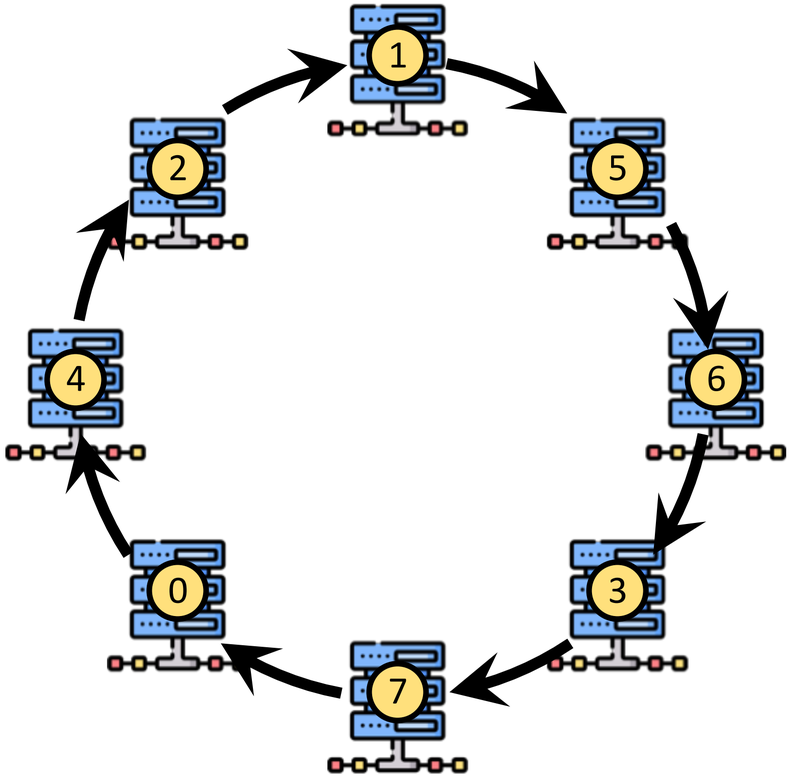
- **Performance:**
  - Each cycle involves *2(n-1)* messages
  - *n-1* **requests** by *i*
  - *n-1* **replies** to *i*

- **Issues:**
  - What if node fails?
  - Performance compared to centralized

Outline

- Centralized Mutual Exclusion
  - Bully algorithm for leader election
- Decentralized Mutual Exclusion
- Distributed 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

Token Ring Algorithm Summary

- **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

PUTTING IT TOGETHER

A Comparison of the 5 Mutex Algorithms

Algorithm	# Messages per cycle	Delay before entry	Problems
Centralized	*3*	*2*	Coordinator Crash
Decentralized	*2mk + m, k≥1*	*2mk*	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

k = number of retries in getting majority

PUTTING IT TOGETHER

Lecture Takeaways

- 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.