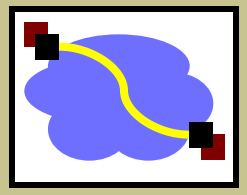
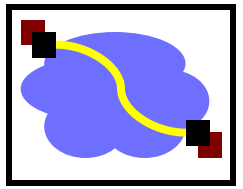


15-440 Distributed Systems

Mutual Exclusion

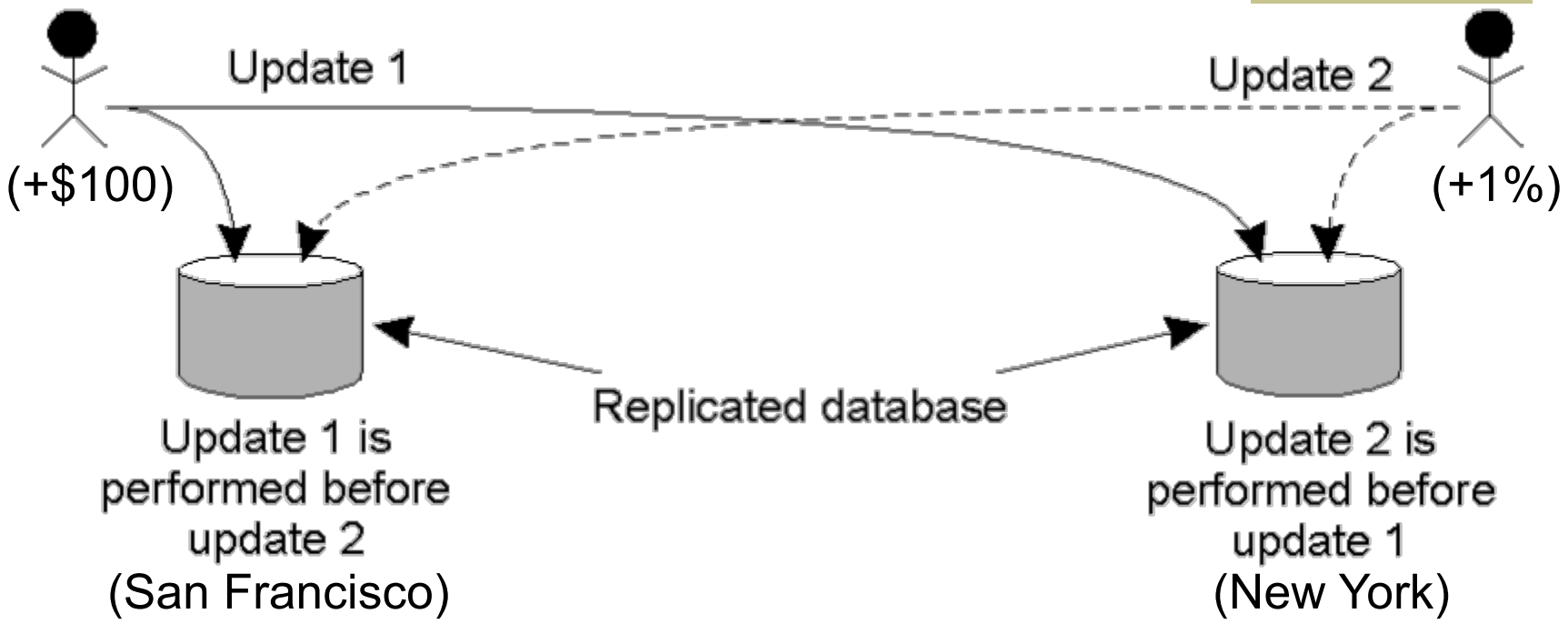
HW1 due and P1 update



- P1 due date error
- Checkpoint: 10/04
- Part A: 10/13
- **Part B: 10/25**

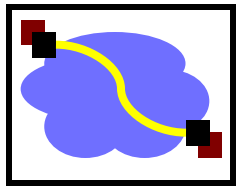
- **Note that there are 3 separate due dates!**

Distributed Database



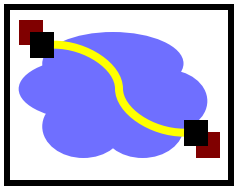
- San Fran customer adds $100, NY bank adds 1% interest
  - San Fran will have $1,111 and NY will have $1,110
- Updating a replicated database and leaving it in an inconsistent state.

Today's Lecture



- Centralized Mutual Exclusion
- Totally Ordered Multicast
- Distributed Mutual Exclusion

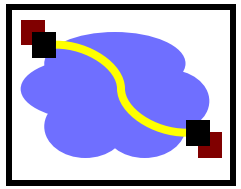
Mutual Exclusion



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

- Must ensure that only one instance of code is in critical section
- Whereas multithreaded systems can use shared memory, we assume that processes can only coordinate via message passing.

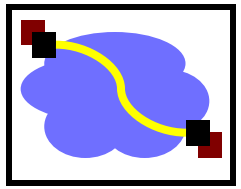
Requirements



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)

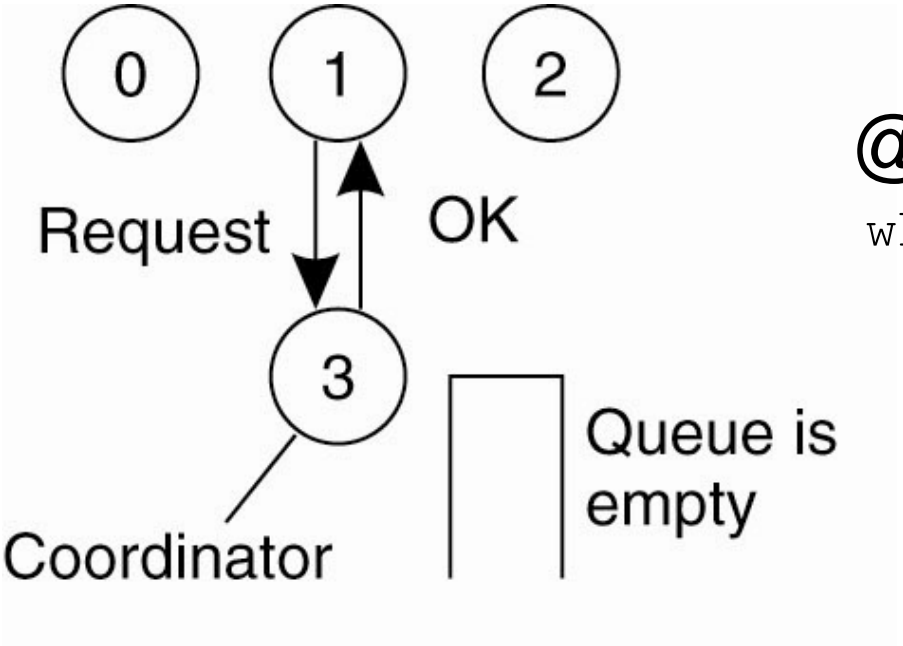
Other Requirements



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

- Today, will focus on 1-3
  - Total number of processes is fixed at n
  - No process fails or misbehaves
  - Communication never fails, but messages may be delivered out of order.

Mutual Exclusion
A Centralized Algorithm (1)



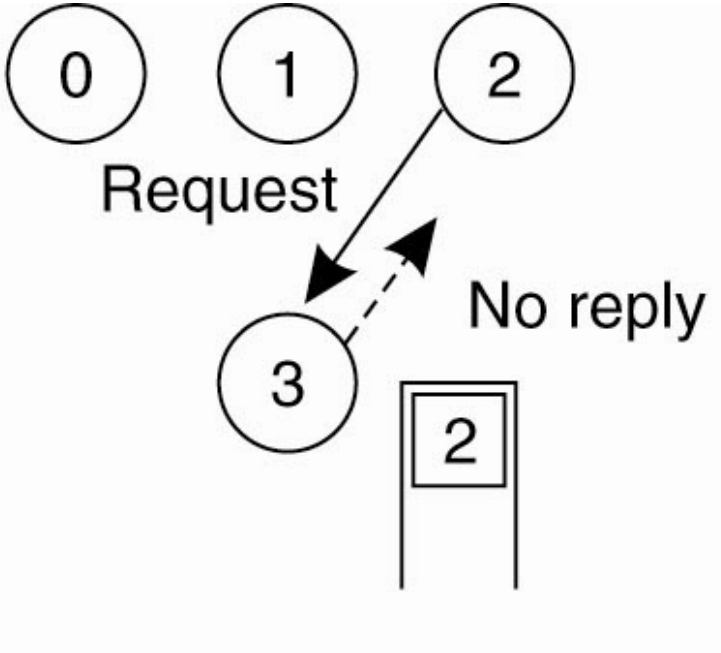
@ Server:

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

@ Client → Acquire:

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

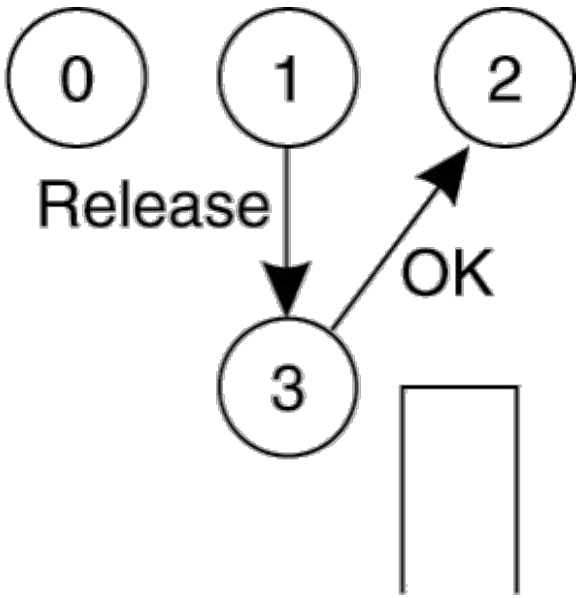
Mutual Exclusion
A Centralized Algorithm (2)



@ Server:

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

Mutual Exclusion
A Centralized Algorithm (3)



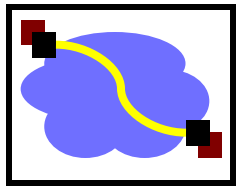
@ Server:

```
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:

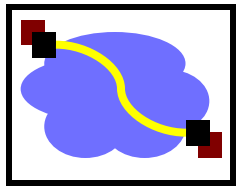
```
Send (Release) to coordinator
```

Mutual Exclusion
A Centralized Algorithm (4)



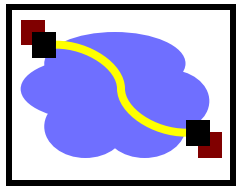
- Correctness:
  - Clearly safe
  - Fairness depends on queuing policy.
    - E.g., if always gave priority to lowest process ID, then processes 1 & 2 lock out 3
- Performance
  - "cycle" is a complete round of the protocol with one process i entering its critical section and then exiting.
  - 3 messages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

Selecting a Leader (Elections)

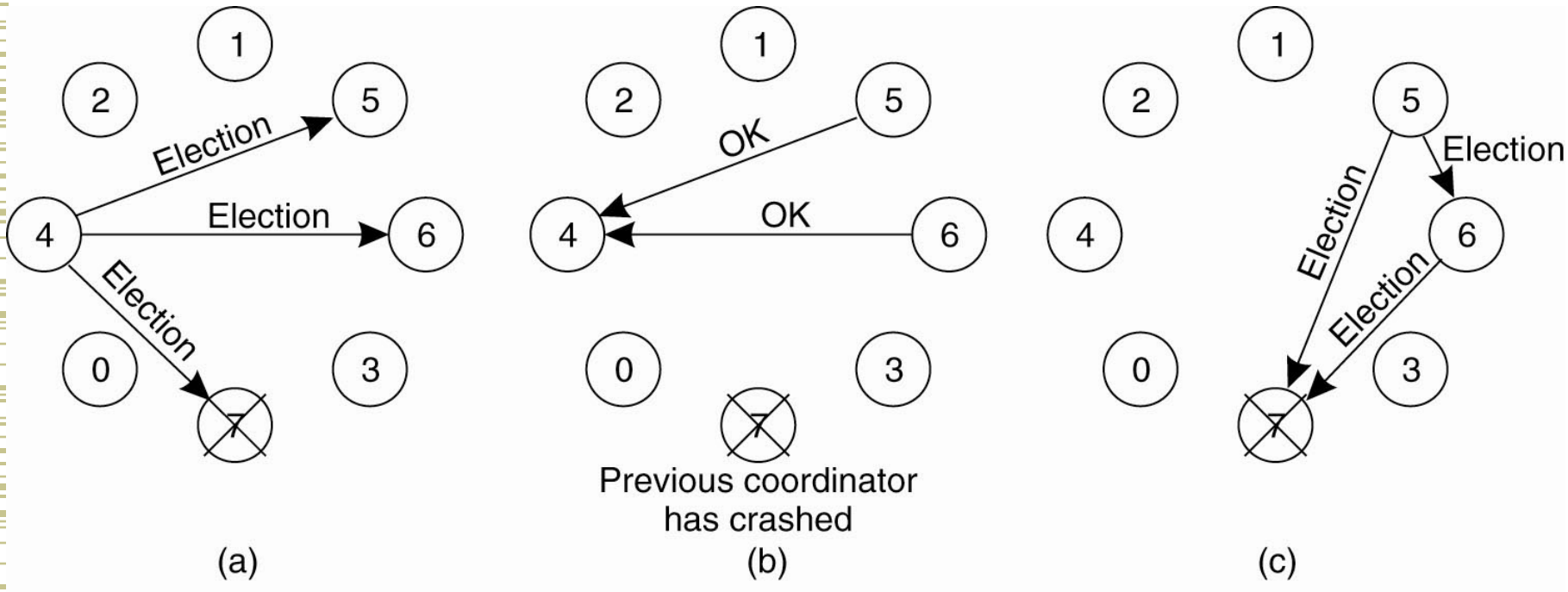


- The Bully Algorithm
  1. P sends an ELECTION message to all processes with higher numbers.
  2. If no one responds, P wins the election and becomes coordinator.
  3. If one of the higher-ups answers, it takes over. P's job is done.

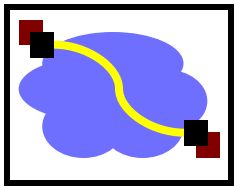
The Bully Algorithm (1)



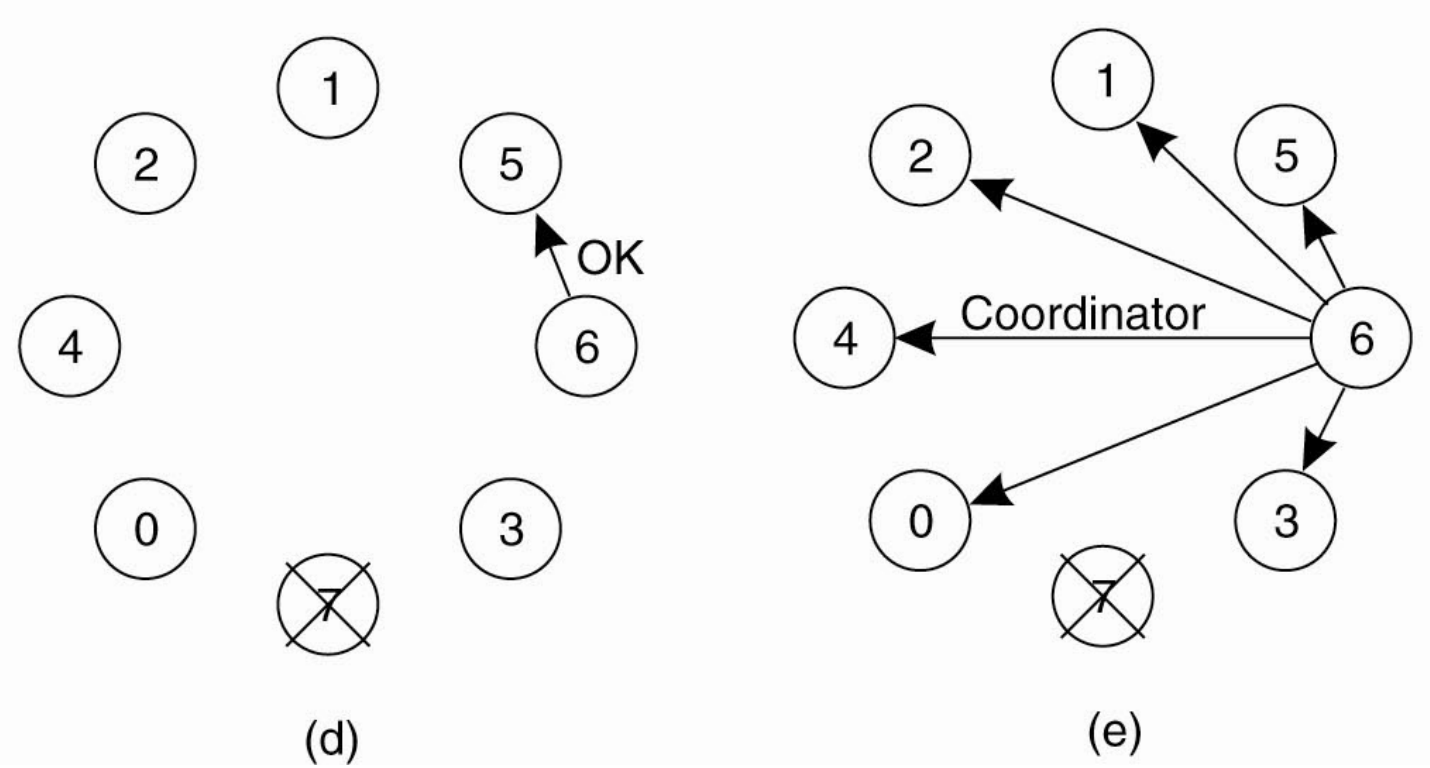
- The bully election algorithm. (a) Process 4 holds an election. (b) Processes 5 and 6 respond, telling 4 to stop. (c) Now 5 and 6 each hold an election.



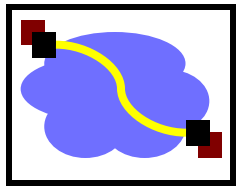
The Bully Algorithm (2)



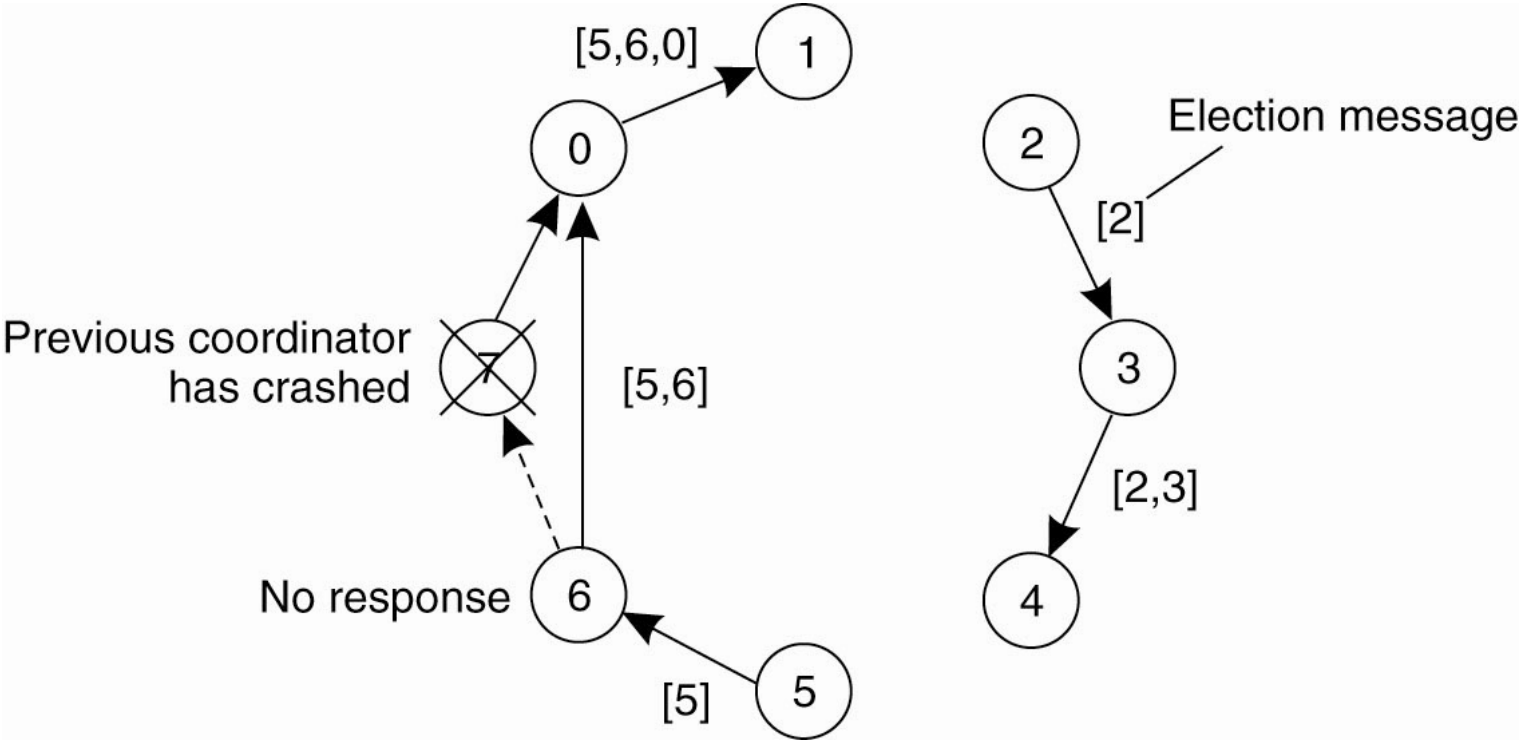
- The bully election algorithm. (d) Process 6 tells 5 to stop. (e) Process 6 wins and tells everyone.



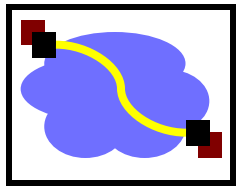
A Ring Algorithm



- Election algorithm using a ring.

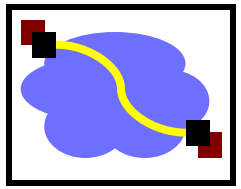


Today's Lecture



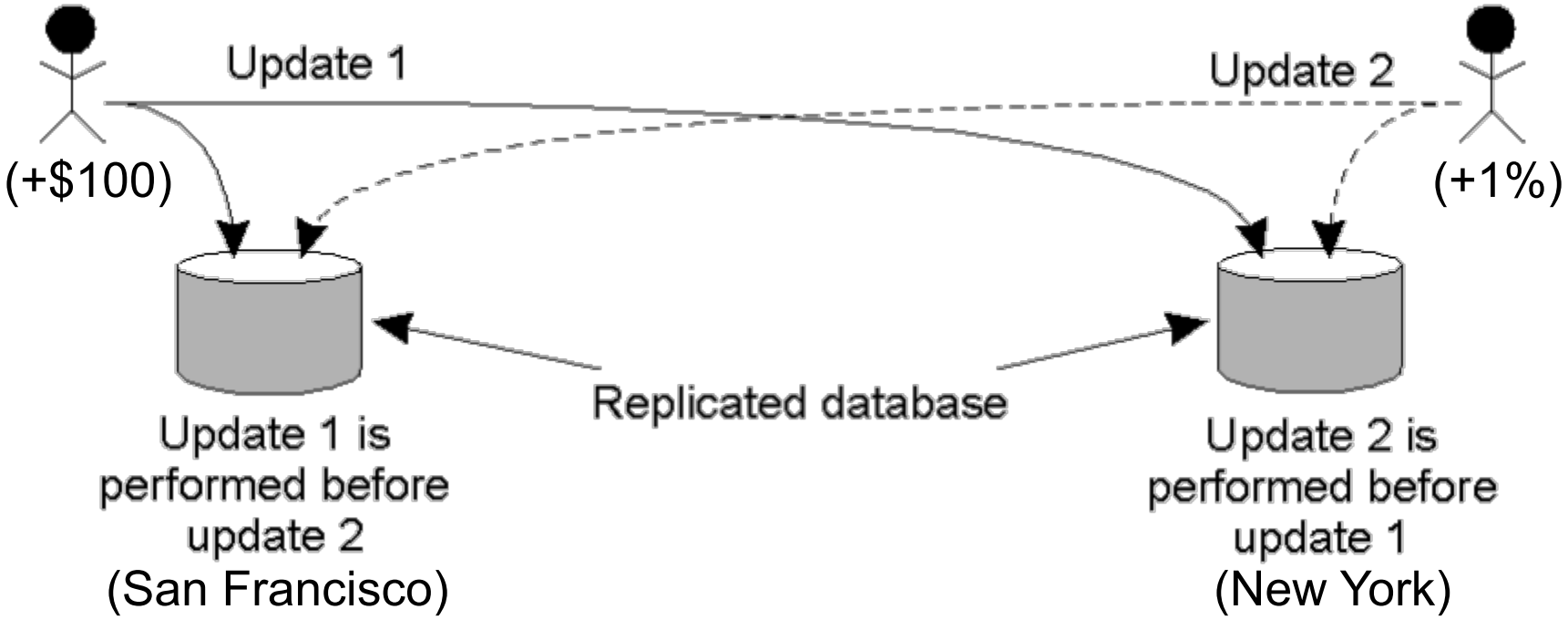
- Centralized Mutual Exclusion
- Totally Ordered Multicast
- Distributed Mutual Exclusion

Decentralized Algorithm Strawman



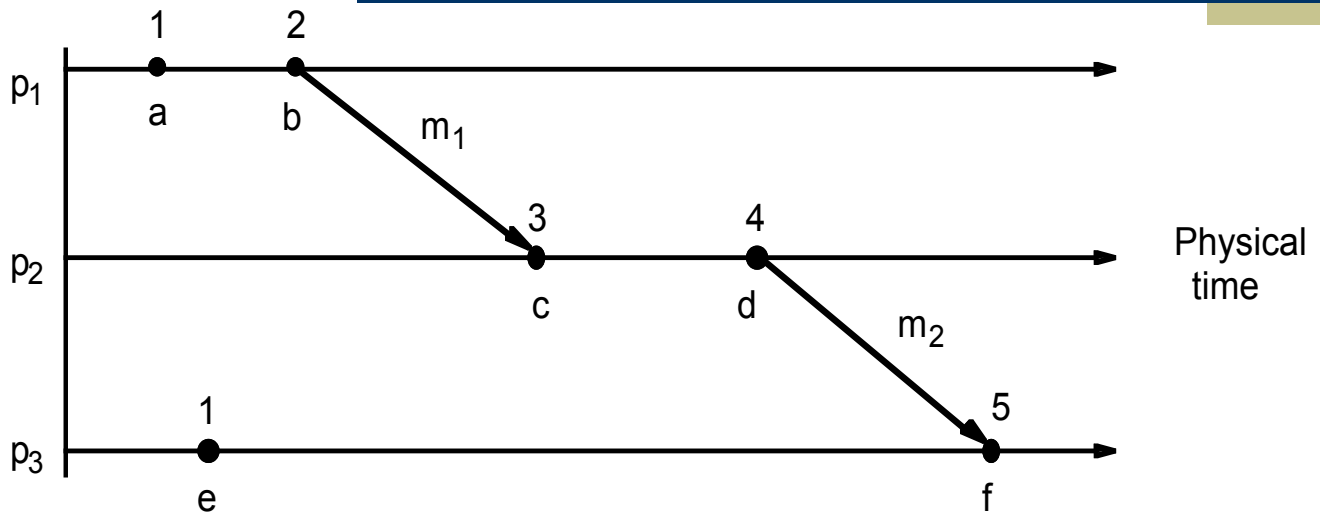
- Assume that there are n coordinators
  - Access requires a majority vote from m > n/2 coordinators.
  - A coordinator always responds immediately to a request with GRANT or DENY
- Node failures are still a problem
  - Coordinators may forget vote on reboot
- What if you get less than m votes?
  - Backoff and retry later
  - Large numbers of nodes requesting access can affect availability
  - Starvation!

Example: Totally-Ordered Multicasting



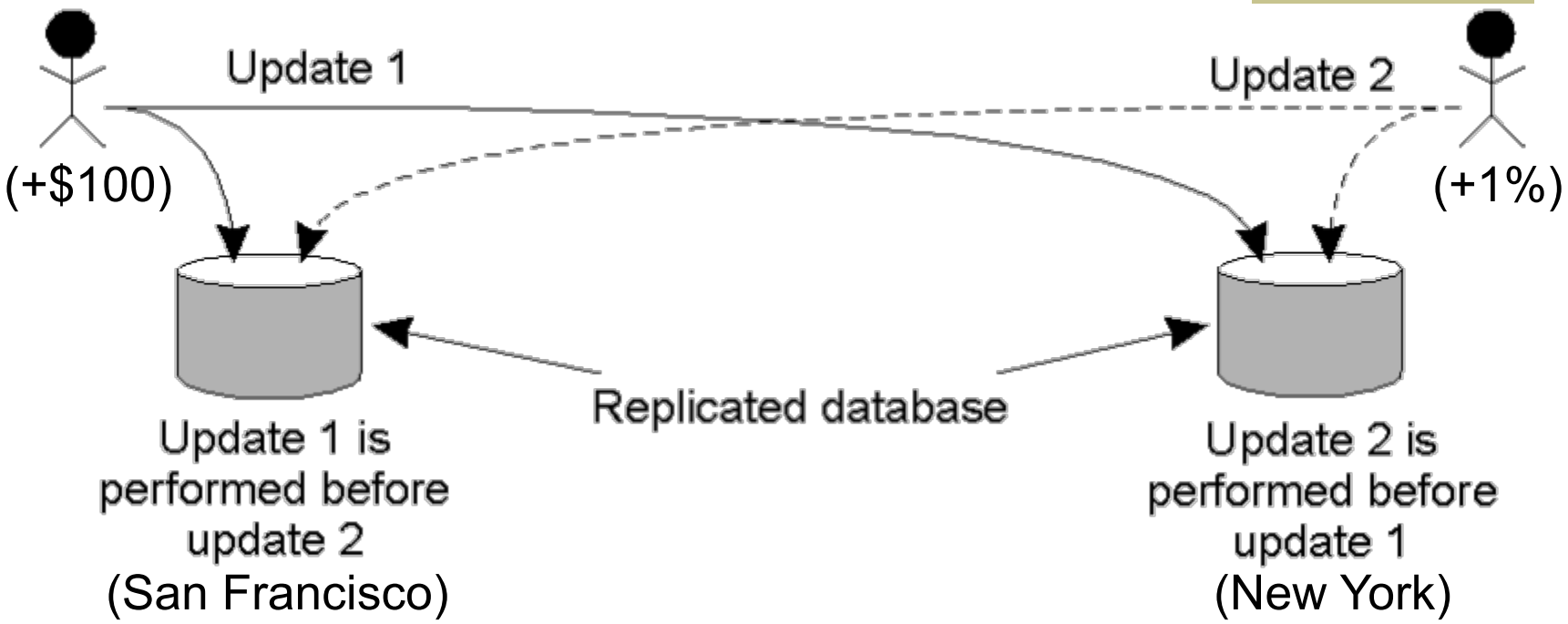
- San Fran customer adds $100, NY bank adds 1% interest
  - San Fran will have $1,111 and NY will have $1,110
- Updating a replicated database and leaving it in an inconsistent state.
- Can use Lamport's to totally order

Lamport Clock (1)



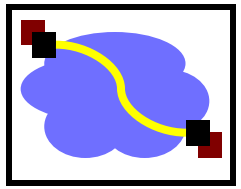
- Rule 1: Li is incremented by 1 before each event at process pi
- Rule 2:
  - (a) when process pi sends message m, it piggybacks t = Li
  - (b) when pj receives (m,t) it sets Lj := max(Lj, t) and applies rule 1 before timestamping the event receive (m)
- Use Lamport's algorithm, but break ties using the process ID
  - L(e) = M * Li(e) + i
    - M = maximum number of processes
    - i = process ID

Example: Totally-Ordered Multicasting



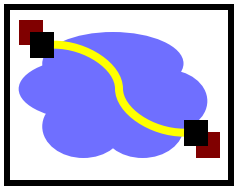
- Can use Lamport's to totally order
- But would need to be able to roll back events
  - Maybe a large number of them!
- Could we make sure things are in the right order before processing?

Totally-Ordered Multicast



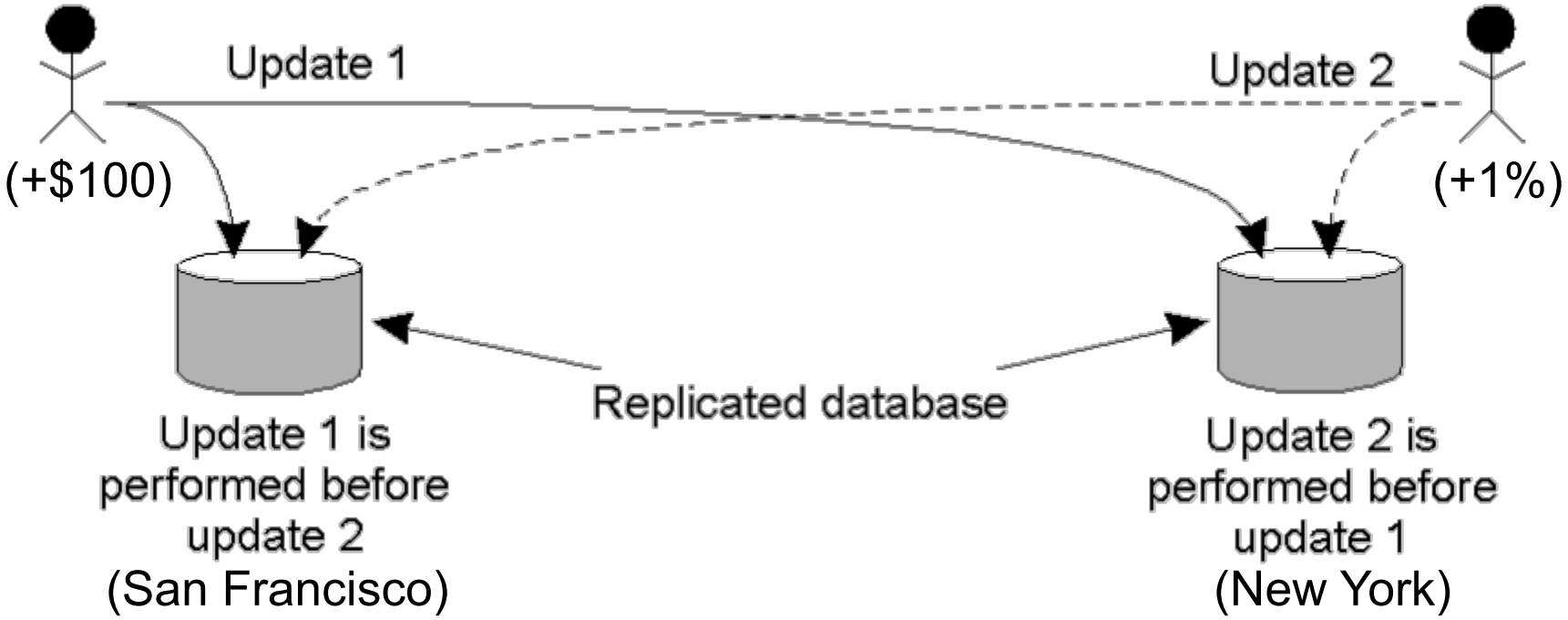
- A multicast operation by which all messages are delivered in the same order to each receiver.
- Lamport Details:
  - Each message is timestamped with the current logical time of its sender.
  - Multicast messages are also sent back to the sender.
  - **Assume all messages sent by one sender are received in the order they were sent and that no messages are lost.**

Totally-Ordered Multicast



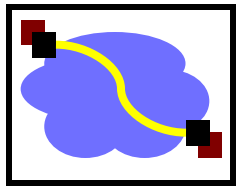
- Lamport Details (cont):
  - Receiving process puts a message into a local queue ordered according to timestamp.
  - The receiver multicasts an ACK to all other processes.
  - Only deliver message when it is *both* at the head of queue and ack'ed by all participants

Example: Totally-Ordered Multicasting



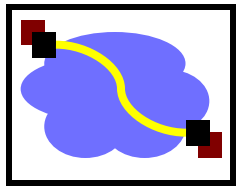
- What are the timestamps of the updates and ACKs?

Totally-Ordered Multicast



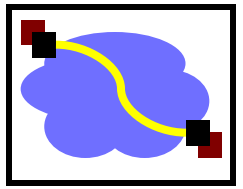
- Lamport Details (cont):
  - Receiving process puts a message into a local queue ordered according to timestamp.
  - The receiver multicasts an ACK to all other processes.
  - Only deliver message when it is *both* at the head of queue and ack'ed by all participants
- Why does this work?
  - Key point from Lamport: the timestamp of the received message is lower than the timestamp of the ACK.
  - All processes will eventually have the same copy of the local queue → consistent global ordering.

Today's Lecture



- Centralized Mutual Exclusion
- Totally Ordered Multicast
- Distributed Mutual Exclusion

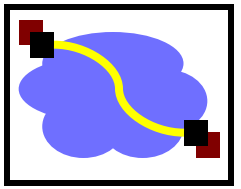
A Distributed Algorithm (Lamport Mutual Exclusion)



- 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')

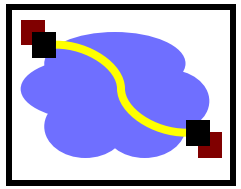
- When node i wants to enter C.S., it sends time-stamped request to all other nodes (including itself)
  - Wait for replies from all other nodes.
  - If own request is at the head of its queue and all replies have been received, enter C.S.
  - Upon exiting C.S., remove its request from the queue and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

A Distributed Algorithm



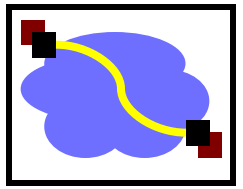
- Other nodes:
  - After receiving a request, enter the request in its own request queue (ordered by time stamps) and reply with a time stamp.
    - This reply is unicast unlike the Lamport totally order multicast example. Why?
      - Only the requester needs to know the message is ready to commit.
      - 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.

A Distributed Algorithm



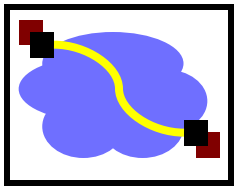
- Correctness
  - When process x generates request with time stamp T_x, and it has received replies from all y in N_x, then its Q contains all requests with time stamps <= T_x.
- Performance
  - Process i sends n-1 request messages
  - Process i receives n-1 reply messages
  - Process i sends n-1 release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

A Distributed Algorithm (take 2) (Ricart & Agrawala)



- 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.

A Distributed Algorithm



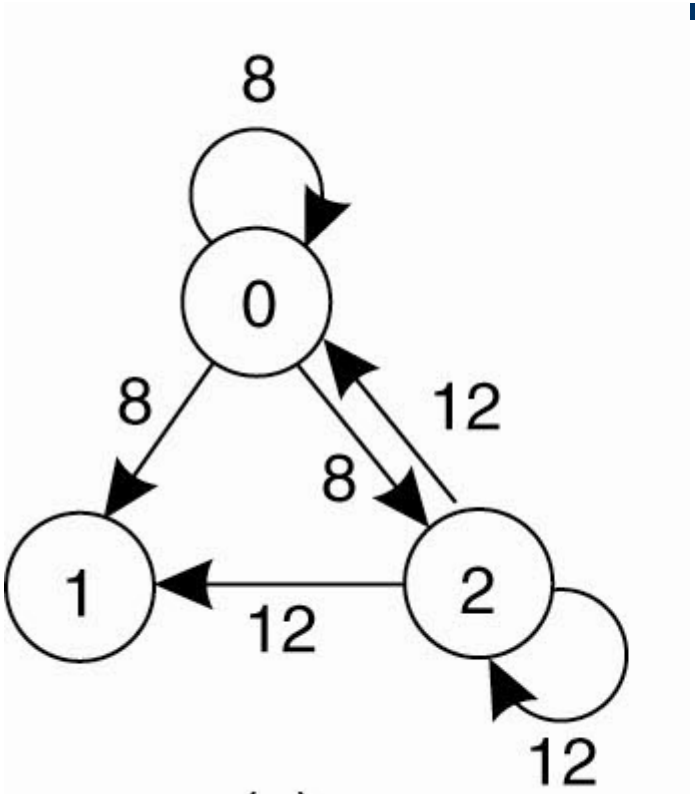
Three different cases:

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

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

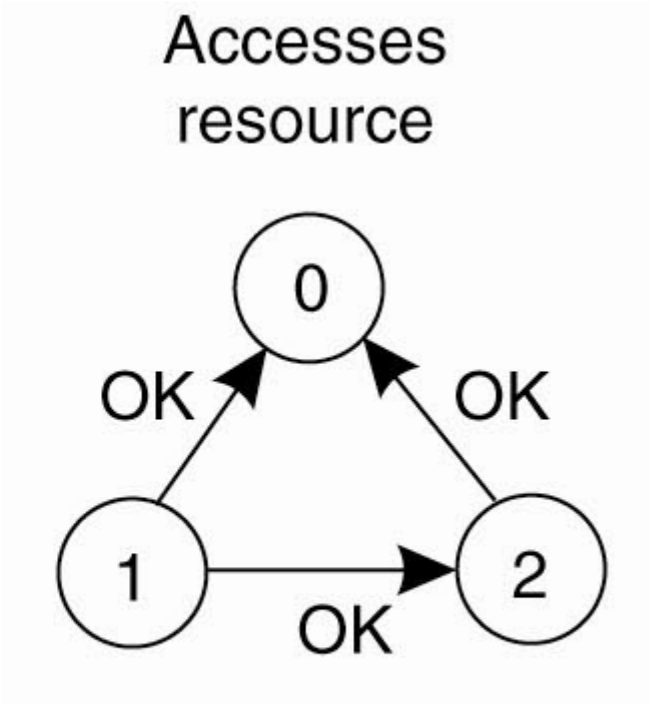
3. If the receiver wants to access the resource as well but has not yet done so, it compares the timestamp of the incoming message with the one contained in the message that it has sent everyone. The lowest one wins.

A Distributed Algorithm



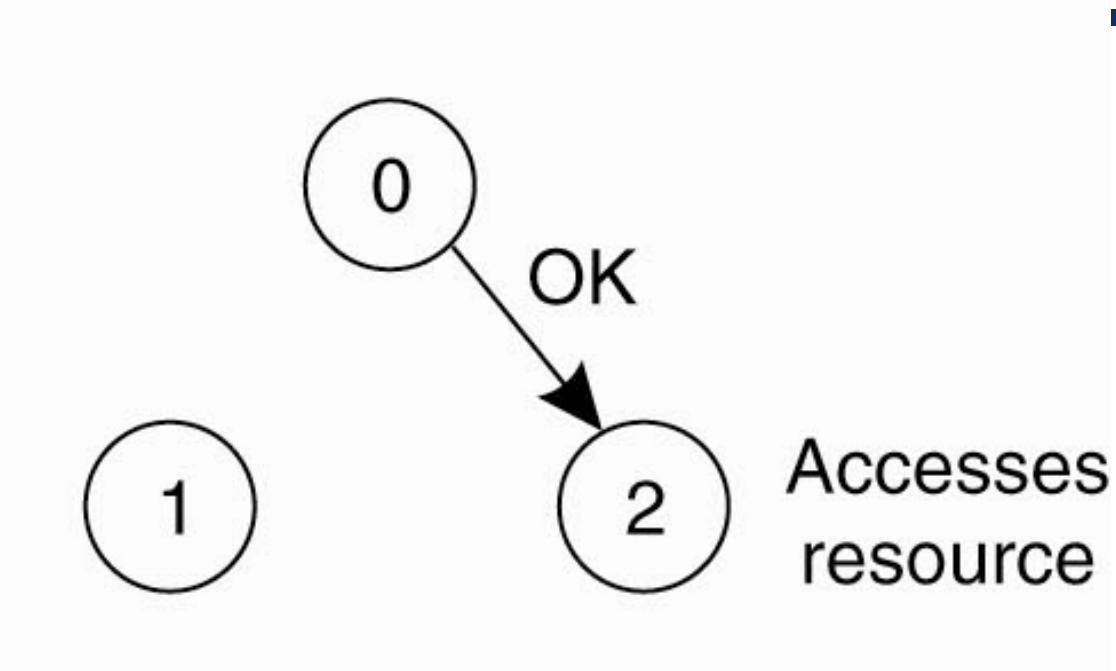
- Two processes (0 and 2) want to access a shared resource at the same moment.

A Distributed Algorithm



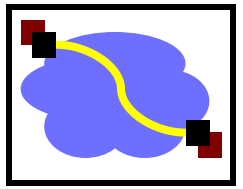
- Process 0 has the lowest timestamp, so it wins.

A Distributed Algorithm (4)



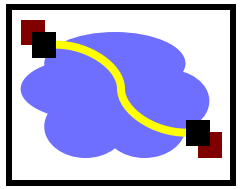
- When process 0 is done, it sends an OK also, so 2 can now go ahead.

Correctness



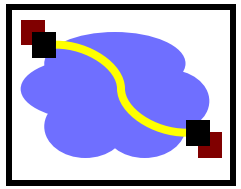
- Look at nodes A & B. Suppose both are allowed to be in their critical sections at same time.
  - A must have sent message (Request, A, Ta) & gotten reply (Reply, A).
  - B must have sent message (Request, B, Tb) & gotten reply (Reply, B).
- Case 1: One received request before other sent request.
 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

Deadlock Free



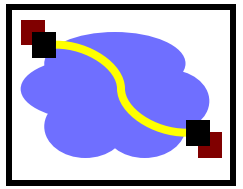
- Cannot have cycle where each node waiting for some other
- Consider two-node case: Nodes A & B are causing each other to deadlock.
  - This would result if A deferred reply to B & B deferred reply to A, but this would require both Ta < Tb & Tb < Ta.
- For general case, would have set of nodes A, B,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

Starvation Free



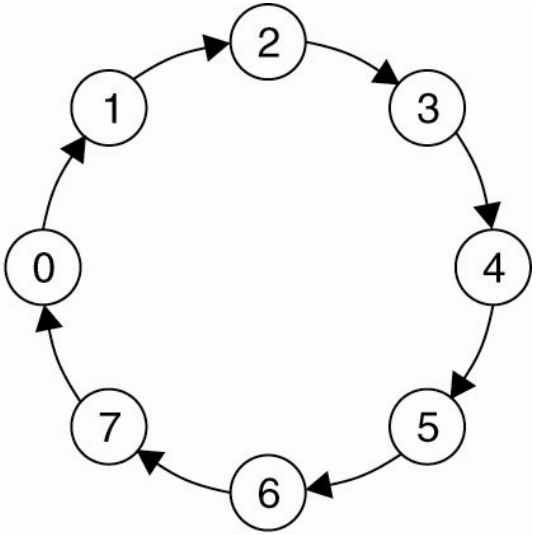
- If node makes request, it will be granted eventually
- Claim: If node A makes a request with time stamp Ta, then eventually, all nodes will have their local clocks > Ta.
- Justification: From the request onward, every message A sends will have time stamp > Ta.
  - All nodes will update their local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

Performance



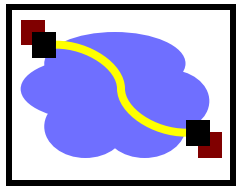
- Each cycle involves 2(n-1) messages
  - n-1 requests by I
  - n-1 replies to I
- Issues
  - What if node fails?
  - Performance compared to centralized

A Token Ring Algorithm



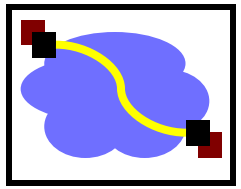
- Organize the processes involved into a logical ring
- One token at any time → passed from node to node along ring

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

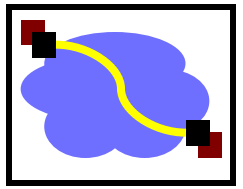
A Comparison of the Four Algorithms



Algorithm	Messages per entry/exit	Delay before entry (in message times)	Problems
Centralized	3	2	Coordinator crash
Decentralized	3mk, k = 1,2,...	2 m	Starvation, low efficiency
Distributed	2 (n – 1)	2 (n – 1)	Crash of any process
Token ring	1 to ∞	0 to n – 1	Lost token, process crash

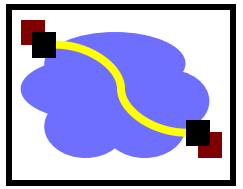
- What happens with crashes?

Summary



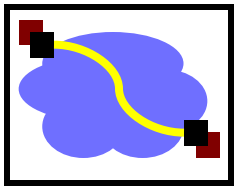
- Lamport algorithm demonstrates how distributed processes can maintain consistent replicas of a data structure (the priority queue).
- Ricart & Agrawala's algorithms demonstrate utility of logical clocks.
- Centralized & ring based algorithms much lower message counts
- None of these algorithms can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

Ricart & Agrawala Example



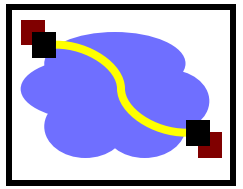
- Processes 1, 2, 3. Create totally ordered clocks by having process ID compute timestamp of form T(e) = 10*L(e)+id, where L(e) is a regular Lamport clock.
- Initial timestamps → P1: 421, P2: 112, P3: 143
- Action types:
  - R m: Receive message m
  - B m: Broadcast message m to all other processes
  - S m to j: Send message m to process j

Timeline



Process	T1	T2	T3	Action
3			153	B (Request, 3, 153)
2		162		R (Request, 3, 153)
1	431			R (Request, 3, 153)
1	441			S (Reply, 1) to 3
2		172		S (Reply, 2) to 3
3			453	R (Reply, 1)
3			463	R (Reply, 2)
3			473	Enter critical section
1	451			B (Request, 1, 451)
2		182		B (Request, 2, 182)
3			483	R (Request, 1, 451)
3			493	R (Request, 2, 182)
1	461			R (Request, 2, 182)
2		462		R (Request, 1, 451) # 2 has D = {1}
1	471			S (Reply, 1) to 2 # 2 has higher priority
2		482		R (Reply, 1)
3			503	S (Reply, 3) to 1 # Release lock
3			513	S (Reply, 3) to 2
1	511			R (Reply, 3) # 1 has R = {2}
2		522		R (Reply, 3) # 2 has R = {}
2		532		Enter critical section
2		542		S (Reply, 2) to 1 # Release lock
1	551			R (Reply, 2) # 1 has R = {}
1	561			Enter critical section

Overall Flow in Example



- P1 and P2 compete for lock after it is released by P3.
- P1's request has timestamp 451, while P2's request has timestamp 182.
- P2 defers reply to P1, but P1 replies to P2 immediately.
- This allows P2 to proceed ahead of P1.