Synchronization in Distributed Systems

Introduction

- Semaphores require processes to access a shared variable. Other synchronization mechanisms may also be based on shared memory (monitors, for example)
- In distributed systems, interprocess communication is via messages (or RPC) so mutual exclusion mechanisms must be based on this approach. Logical clocks are an example.

Message Passing

- Processes may use broadcast (transmit to any interested process) or multicast (transmit to all members of a specific group) message passing.
- Blocking send/receive protocols are often used, because it can be guaranteed that the sender and receiver are synchronized when the message is exchanged
  - Sender will not continue until message has reached receiver

Example

- Process 1 at Site 1

  write to file

  Send (p2, message)

  If the sender wants to rendezvous with receiver, the Send can also be blocking – or a blocking Receive can be executed immediately after the send.

- Process 2 at Site 2

  Receive* (p1, message)

  read from file

  Note similarity to solution that used semaphore to force one process to wait for another.

  * blocking

Distributed Mutual Exclusion Characterized

- Processes communicate only through messages – no shared memory or clock.
- Processes must expect unpredictable message delays.
- Processes coordinate access to shared resources (printer, file, etc.) that should only be used in a mutually exclusive manner.

Example: Overlapped Access to Shared File

- Airline reservation systems maintain records of available seats.
- Suppose two people buy the same seat, because each checks and finds the seat available, then each buys the seat.
- Overlapped accesses generate different results than serial accesses – race condition.

Desirable Characteristics for Distributed Mutex Solutions

(1) no deadlocks – no set of sites should be permanently blocked, waiting for messages from other sites in that set.
(2) no starvation – no site should have to wait indefinitely to enter its critical section, while other sites are executing the CS more than once
(3) fairness - requests honored in the order they are made. This means processes have to be able to agree on the order of events. (Fairness prevents starvation.)
(4) fault tolerance – the algorithm is able to survive a failure at one or more sites.

Distributed Mutex – Overview

- **Token-based solution**: processes share a special message known as a token
  - Token holder has right to access shared resource
  - Wait for/ask for (depending on algorithm) token; enter Critical Section when it is obtained, pass to another process on exit or hold until requested (depending on algorithm)
  - If a process receives the token and doesn’t need it, just pass it on.

Overview - Token-based Methods

- Advantages:
  - Starvation can be avoided by efficient organization of the processes
  - Deadlock is also avoidable
- Disadvantage: token loss
  - Must initiate a cooperative procedure to recreate the token
  - Must ensure that only one token is created!

Overview

- **Permission-based** solutions: a process that wishes to access a shared resource must first get permission from one or more other processes.
- Avoids the problems of token-based solutions, but is more complicated to implement.

Basic Algorithms

- Centralized
- Decentralized
- Distributed
  - Distributed with “voting” – for increased fault tolerance
- Token Ring

Centralized Mutual Exclusion

Central coordinator manages requests

FIFO queue to guarantee no starvation

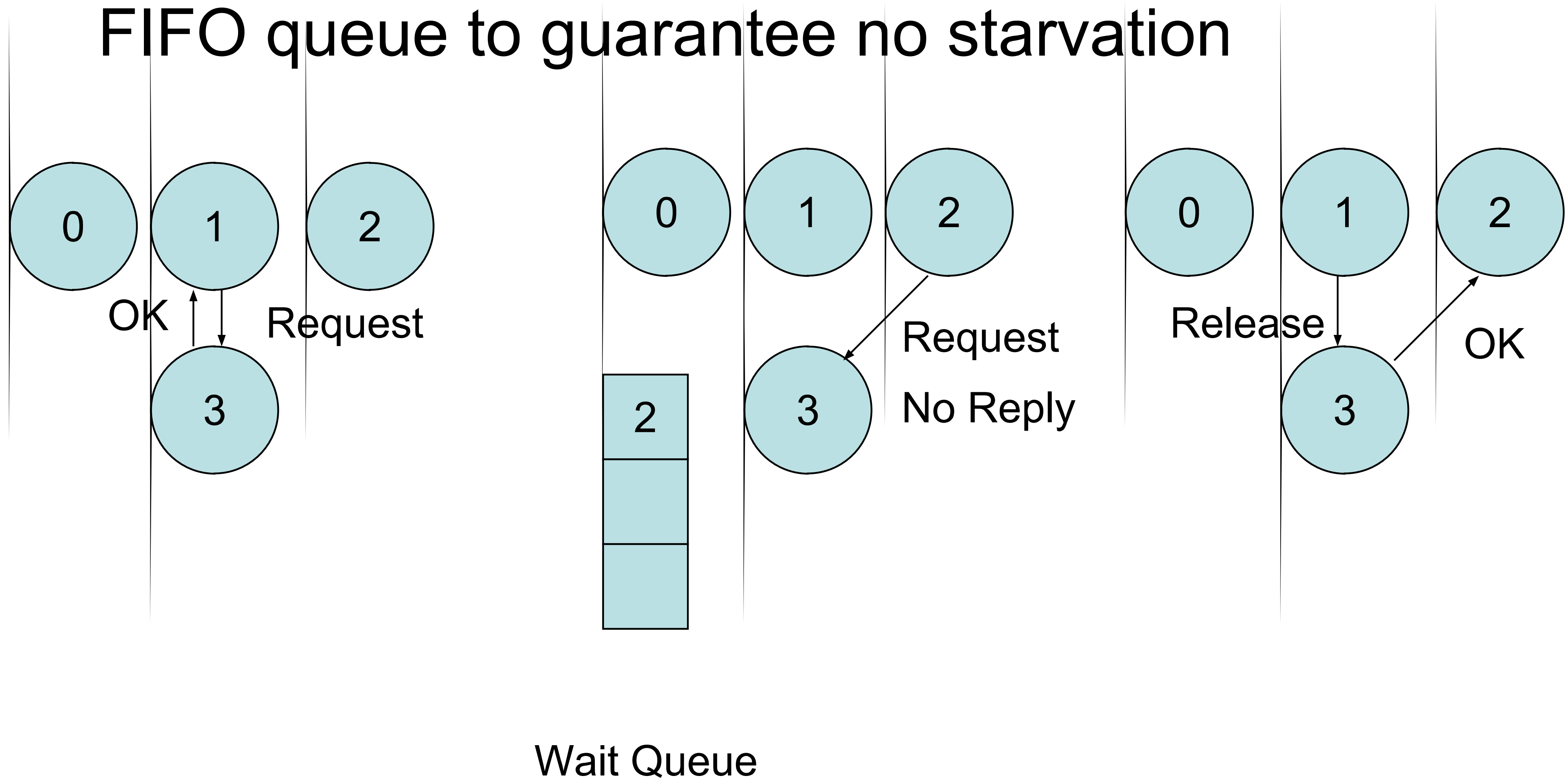


Figure 6-14

Performance Analysis

- Guarantees mutual exclusion
- No starvation/fair
  - If requests are honored in order
- No deadlock
- Fault tolerant?
  - Single point of failure
  - Blocking requests mean client processes have difficulty distinguishing crashed coordinator from long wait
  - Bottlenecks
- Simplicity is a big plus

Decentralized Mutex Algorithm

- More fault-tolerant than the centralized approach.
- Uses the Distributed Hash Table (DHT) approach to locate objects/replicas
  - Object names are hashed to find the node where they are stored (*succ* function)
- *n* replicas of each object are placed on n successive nodes
  - Hash object name to get addresses
- Now every <u>replica</u> has a coordinator that controls access

The Decentralized Algorithm

- Coordinators respond to requests at once: Yes or No
- For a process to use the resource it must receive permission from $m > n/2$ coordinators.
  - If the requester gets fewer than m votes it will wait for a random time and then ask again.
- If a request is denied, or when the CS is completed, notify the coordinators who have sent OK messages, so they can respond again to another request. (Why is this important?)

Analysis

- More robust than the central coordinator approach. If one coordinator goes down others are available.
  - If a coordinator fails and resets then it will not remember having granted access to one requestor, and may then give access to another. According to the authors, it is highly unlikely that this will lead to a violation of mutual exclusion. (See the text for a probabilistic argument.)

Analysis

- If a resource is in high demand, multiple requests will be generated by different processes.
- High level of contention
- Processes may wait a long time to get permission - Possibility of starvation exists
- Resource usage drops.

Distributed Mutual Exclusion

- Probabilistic algorithms do not <u>guarantee</u> mutual exclusion is correctly enforced.
- Many other algorithms do, including the following.
- Originally proposed by Lamport, based on his logical clocks and total ordering relation
- Modified by Ricart-Agrawala

The Algorithm

- Two message types:
  - Request access to shared resource: sent <u>to</u> all processes in the group
  - Reply/OK: A message eventually received at the request site, S_i, <u>from</u> all other sites.
- Messages are time-stamped based on Lamport's total ordering relation, with logical clock.process id. Reason: to provide an unambiguous order of all relevant events.

Requesting

- When a process P_i wants to access a shared resource it builds a message with the resource name, pid and current timestamp: *Request* (r_a, ts_i, i)
  - A request sent from P_3 at "time" 4 would be time-stamped (4.3). Send the message to all processes, including yourself.
- Assumption: message passing is reliable.

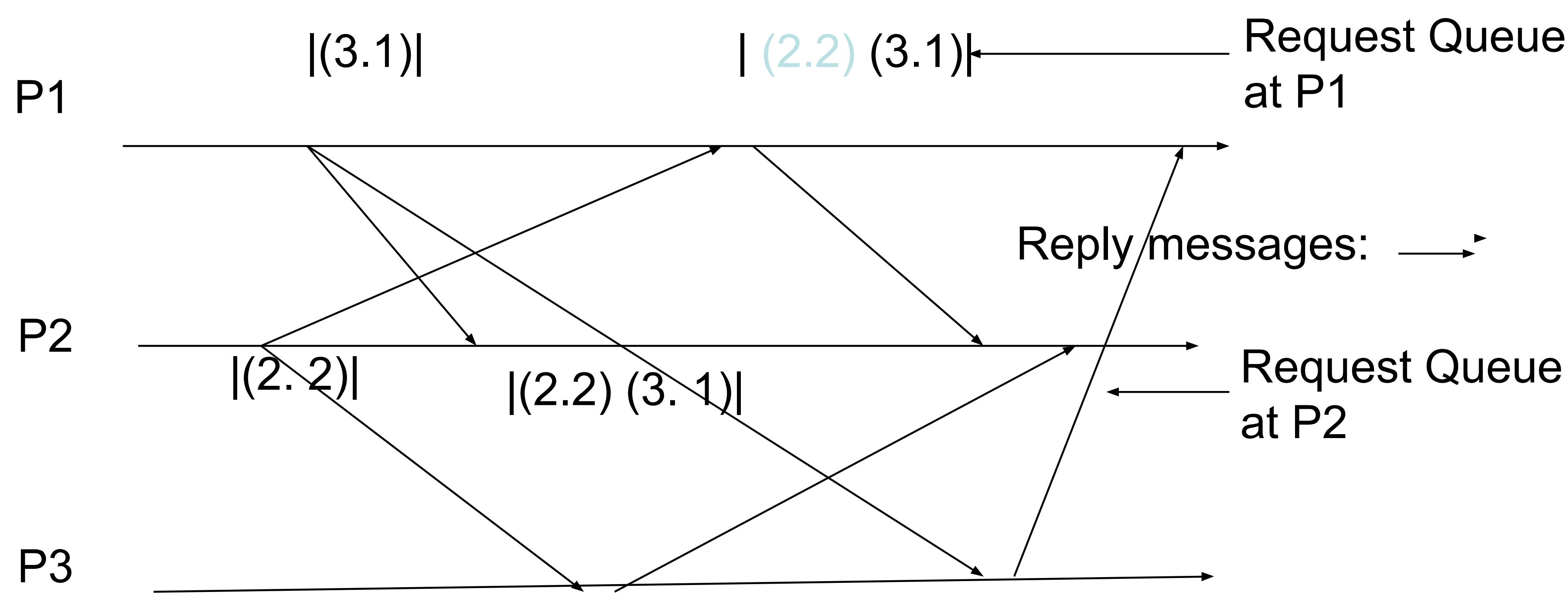
Processing a Request

- P_i sends a Request (r_a, ts_i, i) to all sites.
- When P_k receives the request it acts based on its own status relative to the critical section.
  - sends a Reply (OK) if it (P_k) is not in the critical section and doesn't want the critical section
  - queues the request locally if it is in its critical section, but does not reply
  - if it isn't in the CS but would like to be, sends a Reply (OK) if the incoming request has a lower timestamp than its own, otherwise queues the request and does not reply. In this case the incoming request has lower priority than P_k's request priority.

Executing the Critical Section

- P_i can enter its critical section when it has received an OK Reply from every other process.
- No undelivered higher priority request can exist. It would have arrived before the OK reply.

Based on Figure 6.4 – Singhal and Shivaratri



Process 2 can enter the critical section.
It has received messages from site 1 and site 3
P2 did not reply to P1 because P1’s Request has a larger timestamp.

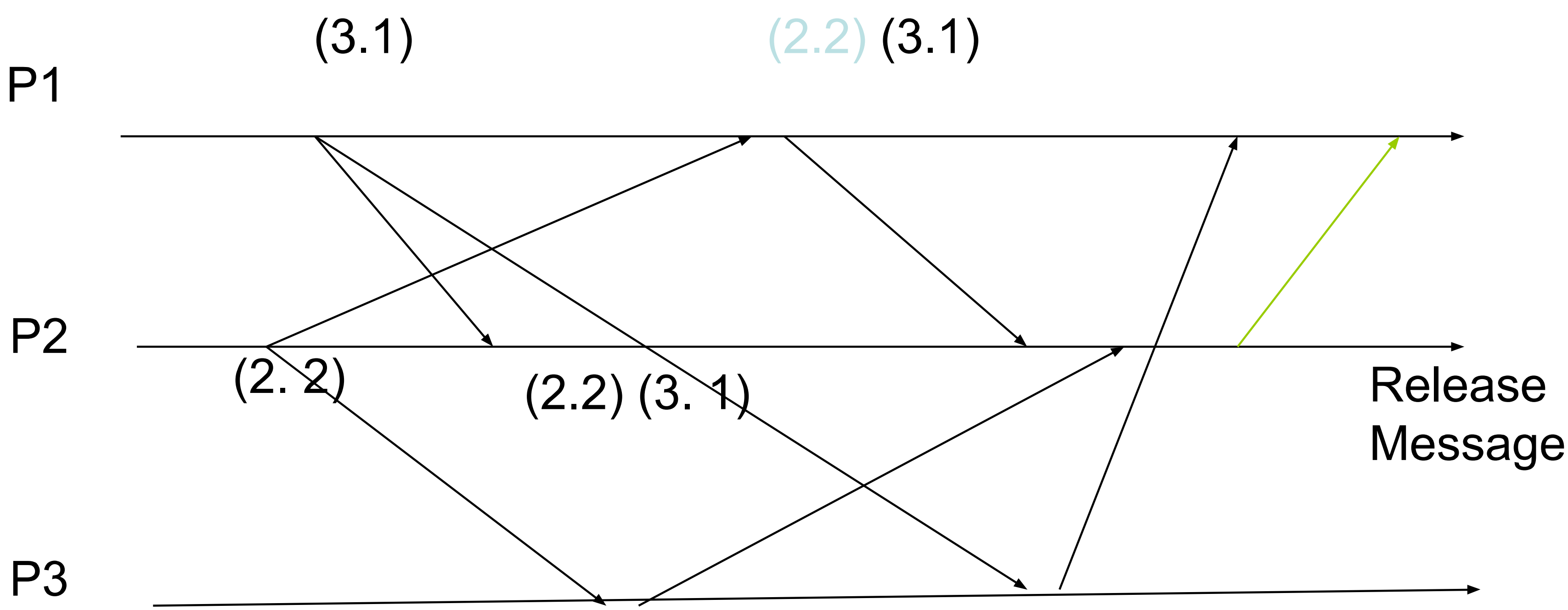
Releasing the Critical Section

- When a processor completes its CS, it sends a Release message to all processors. (the Release message acts as an OK Reply to any process waiting for one.)
- Its request is removed from all queues at this time.
- If other processes are waiting to execute CS’s, one of them will now be able to proceed.

Comments

- Purpose of REPLY from node *i* to *j:* ensures that *j* has seen all requests from *i* prior to sending the REPLY (and therefore, possibly any request of *i* with timestamp lower than *j*'s 's request)
- Requires FIFO channels.
- 2(n – 1) messages per critical section
- Synchronization delay = one message transmission time
- Requests are granted in order of increasing timestamps

Based on Figure 6.4



When Process 2 leaves the critical section it sends a Release Message to Process 1 and now the next site can enter mutual exclusion.

Analysis

- Deadlock and starvation are not possibilities since the queue is maintained in order, and OKs are eventually sent to all processes with requests.
- Message passing overhead is not excessive: 2(n-1) per critical section
- However – Consider fault tolerance.
  - What happens if a processor in the system fails?

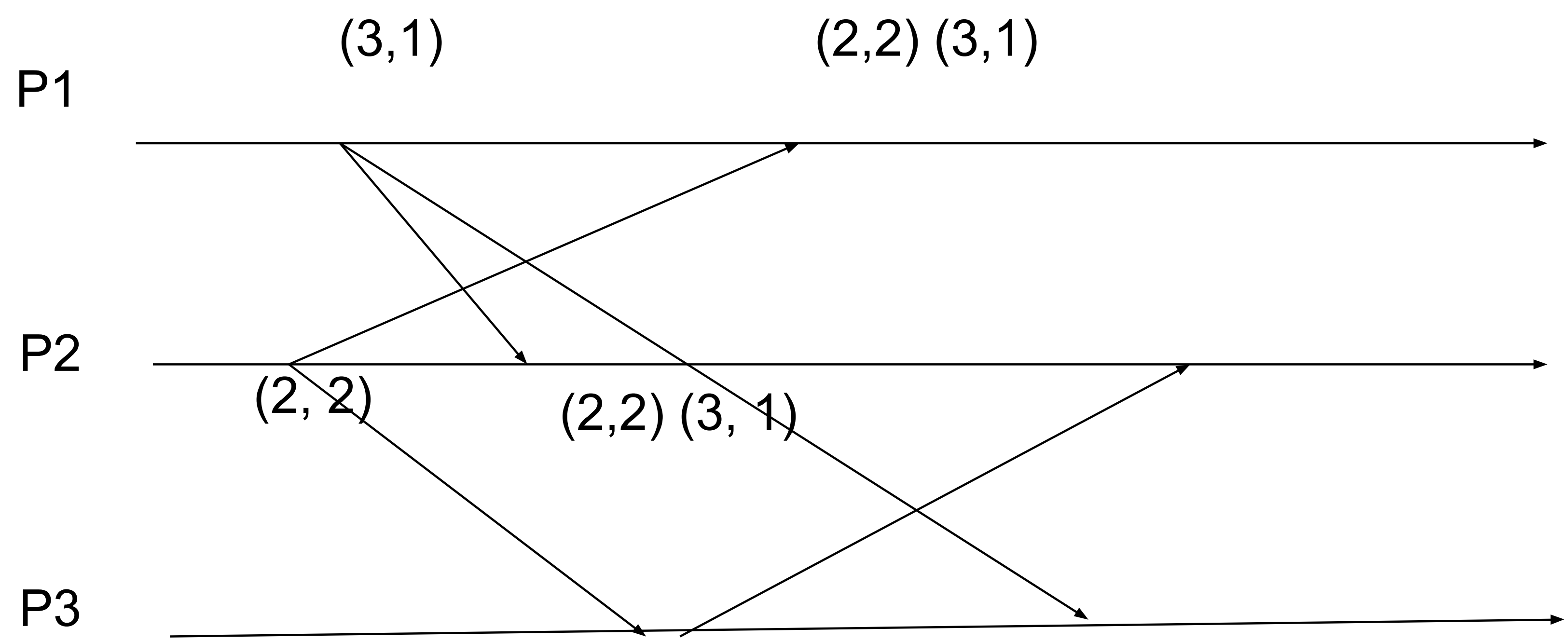
Dealing With Failure – Additional Messages

- When a request comes in, the receiver always sends a reply (Yes or No)
- If a reply doesn't come in a reasonable amount of time, repeat the request.
  - Continue until a reply arrives or until the sender decides the processor is dead.
- Lots of message traffic, *n* bottlenecks instead of one, must keep track of all group members, etc.

Dealing with Failure - A Voting Scheme Modification

- But it really isn’t necessary to get 100% permission
- When a processor receives a Request it sends a reply (vote) only if it hasn't "voted" for some other process. In other words, **you can only send out one OK Reply at a time**.
- There should be at most N votes in the system at any one time.

Voting Solution to Mutual Exclusion



Process 3 receives two requests; it “votes” for P2 because it received P2’s request first. P2 votes for itself.

A Further Problem

- Voting improves fault tolerance but what about deadlock?
- Suppose a system has 10 processes
- Also assume that three Requests are generated at about the same time and that two get 3 votes each, one gets 4 votes, so no process has a majority.

Tweaking the Voting Solution to Prevent Deadlock

- A processor can change its vote if it receives a Request with an earlier time stamp than the request it originally voted for.
  - Additional messages: Retract and Relinquish.
- If P_j receives a Retract message from P_k and has not yet entered its critical section, it sends P_k a Relinquish message and no longer counts that vote.

Tweaking the Voting Solution

- When P_k gets a Relinquish message, it can vote again.
- Eventually, the processor with the earliest timestamp will get enough votes to begin executing the critical section.
- This is still an O(N) algorithm although the message traffic is increased.

A Token Ring Algorithm

- Previous algorithms are permission based, this one is token based.
- Processors on a bus network are arranged in a logical ring, ordered by network address, or process number (as in an MPI environment), or some other scheme.
- Main requirement: that the processes know the ordering arrangement.

Algorithm Description

- At initialization, process 0 gets the token.
- The token is passed around the ring.
- If a process needs to access a shared resource it waits for the token to arrive.
- Execute critical section & release resource
- Pass token to next processor.
- If a process receives the token and doesn’t need a critical section, hand to next processor.

Analysis

- Mutual exclusion is guaranteed trivially.
- Starvation is impossible, since no processor will get the token again until every other processor has had a chance.
- Deadlock is not possible because the only resource being contended for is the token.
- The problem: lost token

Lost Tokens

- What does it mean if a processor waits a long time for the token?
  - Another processor may be holding it
  - It’s lost
- No way to tell the difference; in the first case continue to wait; in the second case, regenerate the token.

Crashed Processors

- This is usually easier to detect – you can require an acknowledgement when the token is passed; if not received in bounded time,
  - Reconfigure the ring without the crashed processor
  - Pass the token to the new “next” processor

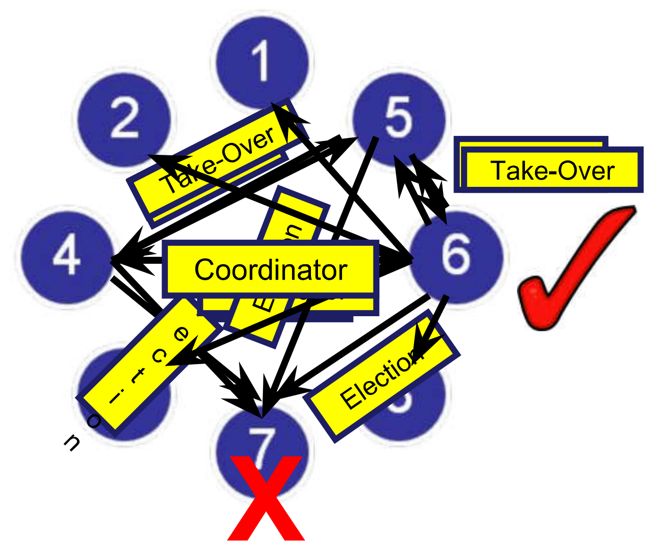
Election Algorithms

- We will study two election algorithms
  1. Bully Algorithm
  2. Ring Algorithm

1. Bully Algorithm

- A process initiates election algorithm when it notices that the existing coordinator is not responding
- Process P_i calls for an election as follows:

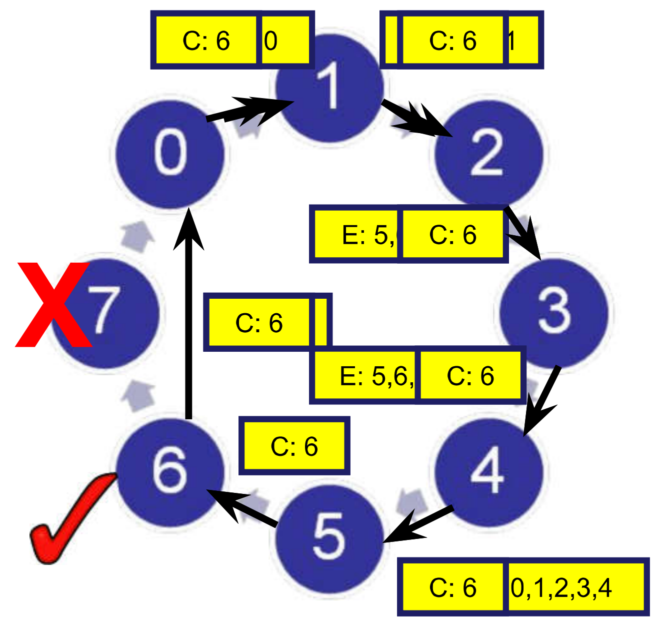
1. P_i sends an "Election" message to all processes with higher process IDs
2. When process P_j with $j>i$ receives the message, it responds with a "Take-over" message. P_j no more contests in the election
   i. Process P_j re-initiates another call for election. Steps 1 and 2 continue
3. If no one responds, P_i wins the election. P_i sends "Coordinator" message to every process



2. Ring Algorithm

- This algorithm is generally used in a ring topology
- When a process P_i detects that the coordinator has crashed, it initiates an election algorithm

1. P_i builds an "Election" message (E), and sends it to its next node. It inserts its ID into the Election message
2. When process P_j receives the message, it appends its ID and forwards the message
   i. If the next node has crashed, P_j finds the next alive node
3. When the message gets back to the process that started the election:
   i. it elects process with highest ID as coordinator, and
   ii. changes the message type to "Coordination" message (C) and circulates it in the ring



Comparison of Election Algorithms

Algorithm	Number of Messages for Electing a Coordinator	Problems
Bully Algorithm	$O(n^2)$	• Large message overhead
Ring Algorithm	$2n$	• An overlay ring topology is necessary

- Assume that:
  - n = Number of processes in the distributed system

Summary of Election Algorithms

- Election algorithms are used for choosing a unique process that will coordinate an activity
- At the end of the election algorithm, all nodes should uniquely identify the coordinator
- We studied two algorithms for election
  - Bully algorithm
    - Processes communicate in a distributed manner to elect a coordinator
  - Ring algorithm
    - Processes in a ring topology circulate election messages to choose a coordinator

Comparison

Algorithm	Messages per entry/exit	Delay before entry	Synch ‡‡ Delay	Problems
Centralized	3	2	2	coordinator crash
Decentralized	3mk*	2m‡	2m	starvation, low efficiency
		(if no competition)		
Distributed	2(n – 1)	2(n – 1)**	1***	crash of <u>any</u> process
Token ring	1 to ∞	0 to n -1	1 to n-1	lost token, process crash

* m: number of coordinators contacted; k: number of attempts

** (n-1) requests and (n-1) replies where n is the number of processes

** n-1 release messages; sent one after the other

*** 1 message to the next process,

‡ Textbook is inconsistent: 2m in the figure, 3mk in the discussion

‡‡ Delay Before Entry: Assumes no other process is in critical section
Synchronization Delay: After one process leaves, how long before next enters
DBE and SD figures assume messages are sent one after another, not broadcast

Summary

- The centralized method is the simplest. If crashes are infrequent, probably the best.
- Also … can couple with a leader election algorithm to improve fault tolerance.
- All of these algorithms work best (and are most likely to be needed) in smallish systems.