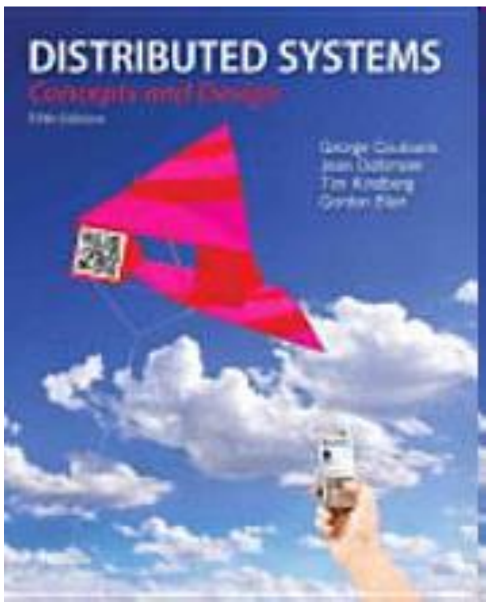
Slides for Chapter 15: Coordination and Agreement



From **Coulouris, Dollimore and Kindberg**
Distributed Systems: Concepts and Design

Objectives

- Understanding of the problems of ***coordination*** and ***agreement*** in distributed systems

- Learning algorithms for ***distributed mutual exclusion*** and ***election*** algorithms, for ***multicast communication***, ***consensus*** and related problems

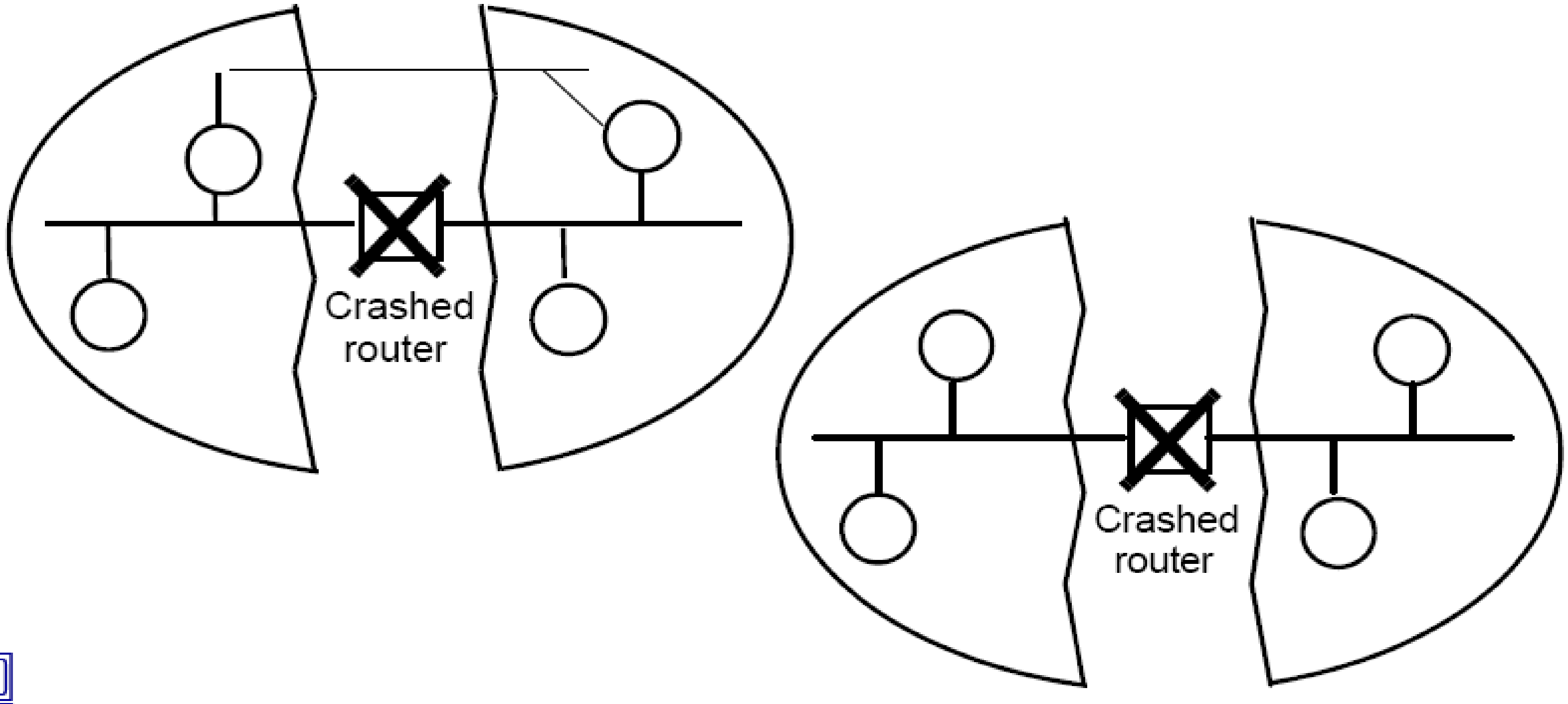
Acknowledgement

- Prof. Narayanan
- Stevens Institute of Technology Computer Science Department

What about Failure?

- Failure Assumptions
  - Reliable channels
    - **retransmission and redundancy to handle needs**
  - Independent Process Communication Capabilities
  - Process failure as crash
    - **correct process**
      - no failures
- Failure Properties
  - failure may partition network
  - connectivity may be asymmetric
  - connectivity may be intransitive
  - inability to communicate at same time
  - all of the above may be temporary

Failure Examples



Failure Detection

- Failure Detector
  - Has a process failed?
  - Possible answers
    - **Unsuspected**
      - reason to believe it is up
    - **Suspected**
      - reason to believe it has failed
    - **Failed**
      - know it has failed

- How practical is failure detection?
  - Synchronous system
    - **can reliably answer question**
    - **few real life systems are synchronous**
  - Asynchronous system
    - **inaccurate**
      - suspect running process
    - **incomplete**
      - unsuspecting of a failed process

Failure Detection

- Unreliable Failure Detection
  - each process p sends a "p is alive" message periodically
  - use time period and maximum transmission delay to detect
    - **OK**
      - if message received
    - **Suspected**
      - if message not received
  - timeout based on network load/delay
- Reliable Failure Detection
  - works only in synchronous systems
  - use time period and maximum transmission delay to detect
    - **OK**
      - if message received
    - **Failed**
      - if message not received

Distributed Mutual Exclusion

- Think of multiple processes accessing a shared resource
- What is the result if multiple updates are executed on the shared resource?
  - e.g. bank account with a withdrawal and a deposit
    - **it is usual for servers that manage resources to provide mutual exclusion mechanisms**
- What is general issue?

Mutual Exclusion Issues

- Resource
  - may need to be accessed exclusively
  - need for mutual exclusion to assure validity
- Critical Section
  - part of process where shared resources are accessed
- Critical Section Problem
  - need to execute critical sections under mutual exclusion
  - this assures validity since at most one process at a time can modify resource

New Challenges for Distributed Mutual Exclusion?

- There is no shared variables or facilities supplied by a single local kernel can be used
- Message passing is the only way to be relied on
- Sometimes there’s even no server (P2P)

General Issue

- Imagine a serverless situation
  - peers attempting to negotiate mutual exclusion
    - **e.g. Ethernet**
    - **e.g. bouncers at exits of bar trying to keep track of how full the bar is**
- Generic mechanism for mutual exclusion is needed

Mutual Exclusion Algorithms

- Consider a set of N processes
  - $p_1, p_2, \ldots, p_N$
  - each process has a critical section
  - critical section is protected as follows:
    - **enter()**
    - **critical section**
    - **exit()**
- Assumptions
  - system is asynchronous
  - processes do not fail
  - message delivery is reliable
    - **eventual delivery**
    - **exactly once**
    - **unaltered**
- What are the requirements on solution?

Mutual Exclusion Requirements

ME1 (safety)	At most one process may execute in the critical section at a time
ME2 (liveness)	Request to enter and exit the critical section eventually succeed
ME3 ($\rightarrow$ ordering)	Entry into the critical section must respect the $\rightarrow$ order of the requests

Properties

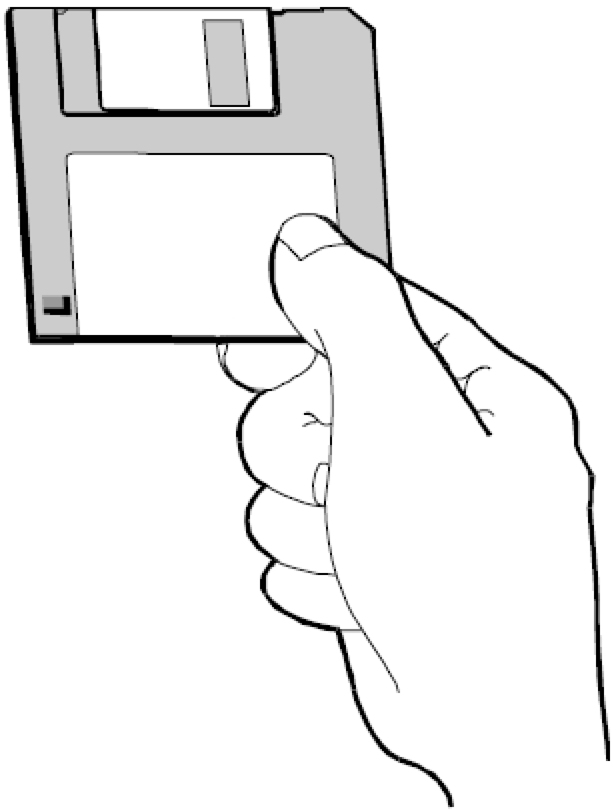
- mutual exclusion by ME1
- deadlock avoided by ME2
- starvation avoided by ME2
- fairness by ME3 (Note the use of $\rightarrow$)

Evaluating Performance

- How can we compare algorithms for mutual exclusion?
  - bandwidth
    - **number of messages sent in each *entry* and *exit* operation**
  - client delay
    - **waiting time at each *entry* and *exit***
  - synchronization delay
    - **time between *exit* and *entry* by next process**
    - **impacts *throughput***

Central Server Algorithm

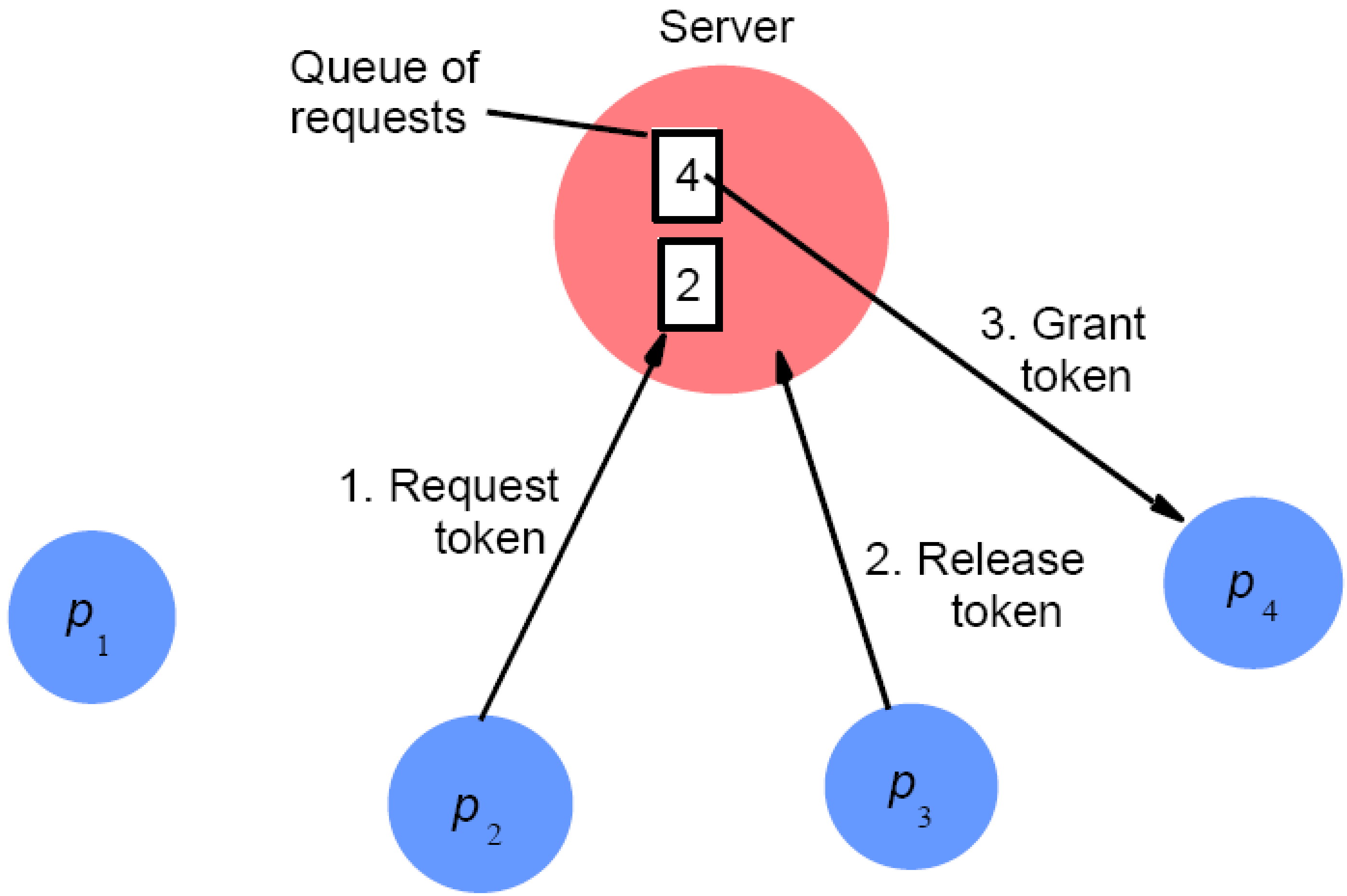
- Server grants permission to enter critical section
  - to enter critical section
    - request entry
    - enter when reply (diskette) is received
  - to exit a critical section
    - return diskette



Server Behavior

- No pending requests
  - wait for request
- Pending requests
  - queued in FIFO order
  - token absent
    - wait until token is received
  - token present
    - remove head of queue and hand over token

Mutual Exclusion Token Server



Characterization of Token Server Algorithm

- ME1
  - number of processes in critical section bounded by number of tokens (= 1)
- ME2
  - FIFO queue
  - no failures
  - process entering queue will be served eventually
- ME3
  - server is ignorant of → relation
  - processes served in order in which messages are received
  - violations of ME3 can occur
- Characteristics
  - entry
  - exit
  - client delay
  - synchronization delay

Token Server Algorithm

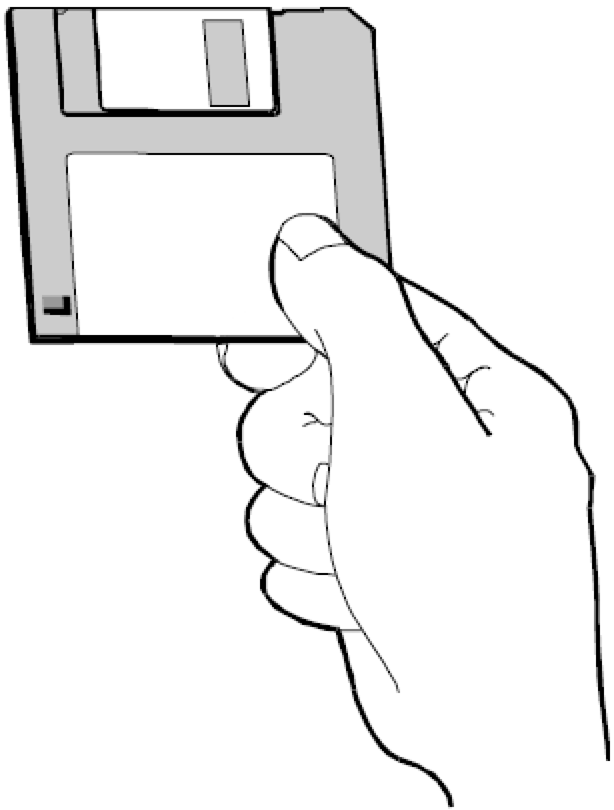
- Characteristics
  - entry
    - request message, receive token (2 messages)
  - exit
    - return token (1 message)
  - client delay
    - depends on size of FIFO queue on the server
    - worst case, linear in number of processes
  - synchronization delay
    - return token and other process receives token (2 messages)

Problem for The Central Server Algorithm

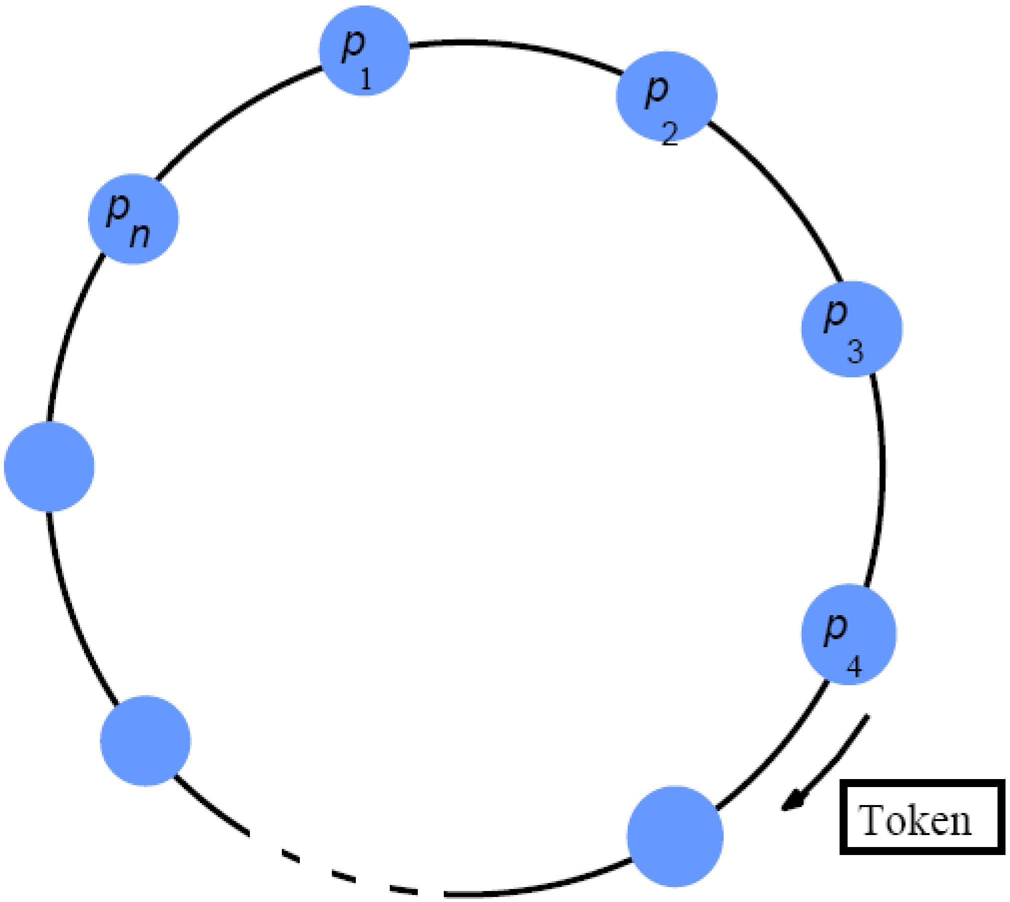
- The server may become a performance bottleneck for the system as a whole.

Ring-Based Algorithm

- Peer processes arranged in a ring
- token cycles in ring
  - to enter critical section
    - enter when token (diskette) is received
  - to exit a critical section
    - pass on token (diskette)



Mutual Exclusion Token Ring



Characterization of Token Ring Algorithm

- ME1
  - number of processes in critical section bounded by number of tokens (= 1)
- ME2
  - ring topology with cycling token
  - no failures
  - process seeking entry will receive token eventually
- ME3
  - token location is independent of $\rightarrow$ relation
  - processes served in token received order
  - violations of ME3 can occur
- Characteristics
  - entry
  - exit
  - client delay
  - synchronization delay

Token Ring Algorithm

- Characteristics
  - entry
    - wait for token to come to the process
    - worst case, average case - linear in number of processes
    - bandwidth occupied even when no one is trying to enter critical section
  - exit
    - single message
  - client delay
    - similar to entry
  - synchronization delay
    - similar to entry

Ricart and Agrawala Algorithm

- N peer processes using multicasting and logical clocks
- entry to critical section
  - multicast a request
  - enter only when reply is received from all processes
  - reply conditions are geared to ensuring that ME1, ME2, and ME3 are met

Algorithm Basics

- Basics
  - each process has a unique numeric identifier (tiebreaker)
  - each process maintains a Lamport clock (see ch.10)
  - request message is of form <T,p_i>
  - <S,p> < <T,q>
    - **S < T or if (S = T) and p < q**
- Assumptions
  - no process failures
  - no message failures
- Process States
  - RELEASED
    - **outside critical section**
  - WANTED
    - **waiting to enter CS**
  - HELD
    - **in the CS**

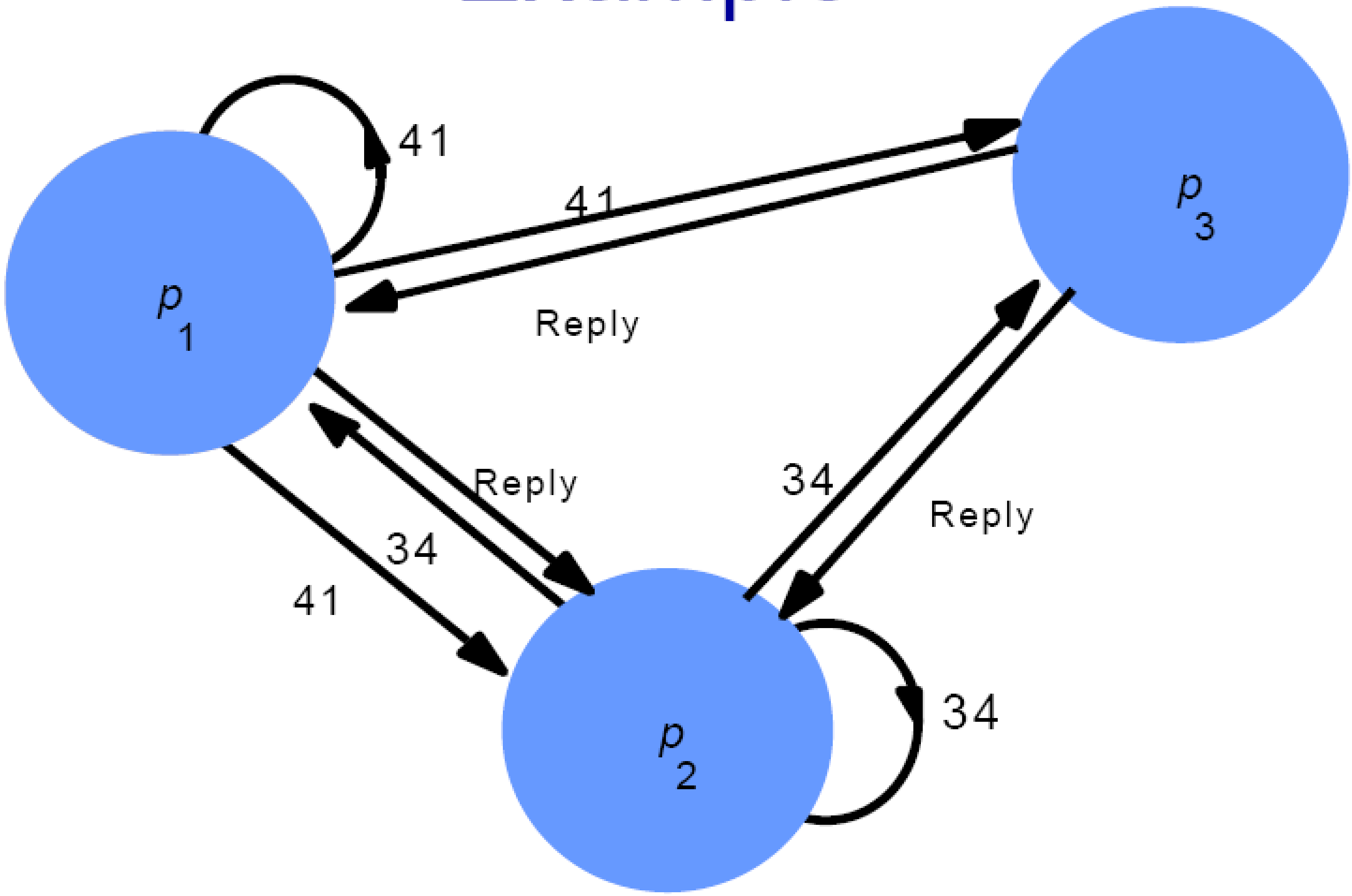
Ricart & Agrawala Algorithm

On initialization
    state := RELEASED;
To enter the section
    state := WANTED;
    Multicast *request* to all processes;
    T := request's timestamp;
    Wait until (number of replies received = $(N - 1)$);
    state := HELD;

request processing deferred here (applies to: *state* := WANTED; Multicast *request* to all processes; T := request's timestamp;)

On receipt of a request $<T_i, p_i>$ *at* p_j $(i \neq j)$
    if (*state* = HELD or (*state* = WANTED *and* $(T, p_j) < (T_i, p_i)$))
    then
        queue *request* from p_i without replying;
    else
        reply immediately to p_i;
    end if
To exit the critical section
    state := RELEASED;
    reply to any queued requests;

Multicast Synchronization Example



Characterization of Algorithm

- Messages <S,p> are totally ordered
  - consider <S,p> and <T,q>
    - **S < T - then <S,p> < <T,q>**
    - **S = T**
      - p < q - then <S,p> < <T,q>
      - p = q - then <S,p> = <T,q>
      - p > q - then <S,p> > <T,q>
    - **S > T - then <S,p> > <T,q>**

ME1 Satisfied
Proof by contradiction

- Suppose two processes p and q enter critical section at the same time
  - <S,p> sent from p to q
  - <T,q> sent from q to p
  - p entered CS
    - **q replied to p. Therefore, <S,p> < <T,q>**
  - q entered CS
    - **p replied to q. Therefore, <T,q> < <S,p>**
  - a contradiction

ME2 is Satisfied

- If p sends entry request message <S,p> to q, all subsequent events in q (after the receipt of that message) have a time stamp greater than S
  - obviously, given the logical clock rules
- If p sends entry request message <S,p>, then other processes can enter the CS before p at most once before p becomes first in total ordering

- Entry request
  - suppose p is indefinitely postponed
  - some process q must exist which does not reply to p
  - if q received the entry request from p
    - **reply immediately OR**
    - **reply upon exit from CS**
  - q must not have received the message
  - contradicts no failure

Characterization of Ricart and Agrawala Algorithm

- Characteristics
  - entry
  - exit
  - client delay
  - synchronization delay
- ME3 - Exercise

Ricart and Agrawala Algorithm

- Characteristics
  - entry
    - one multicast request, N-1 reply messages
  - exit
    - size of queue
  - client delay
    - number of processes wishing to enter ahead of you in precedence (worst case, average case - linear)
  - synchronization delay
    - next process (in precedence) has already received N-2 messages. one more message will allow it to enter.