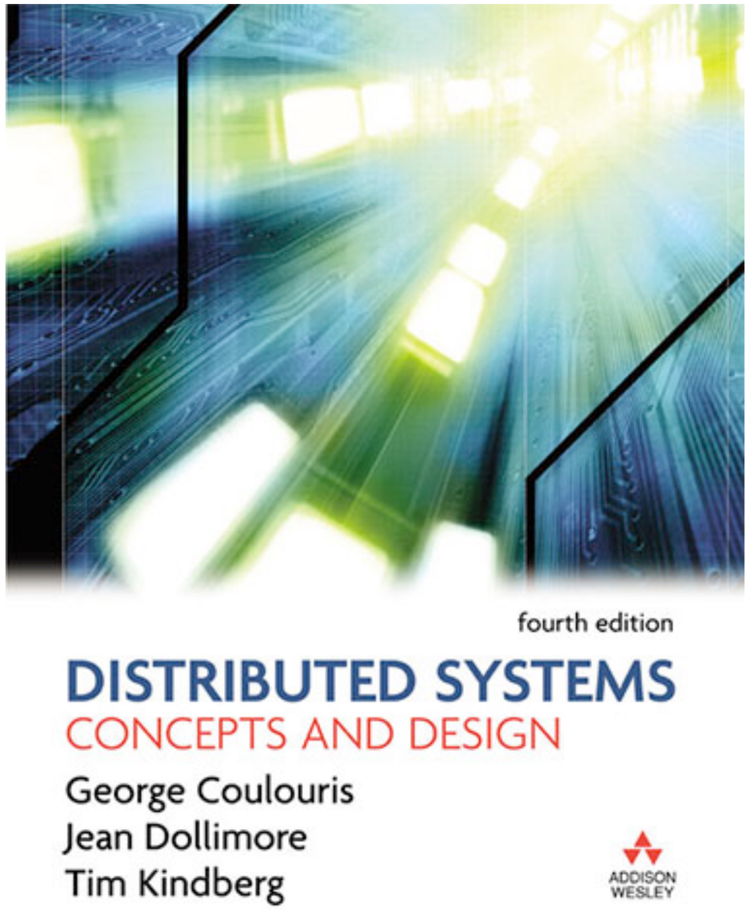
Coordination and Agreement



AIM: Coordination and/or Agreement

- Collection of algorithms whose goals vary

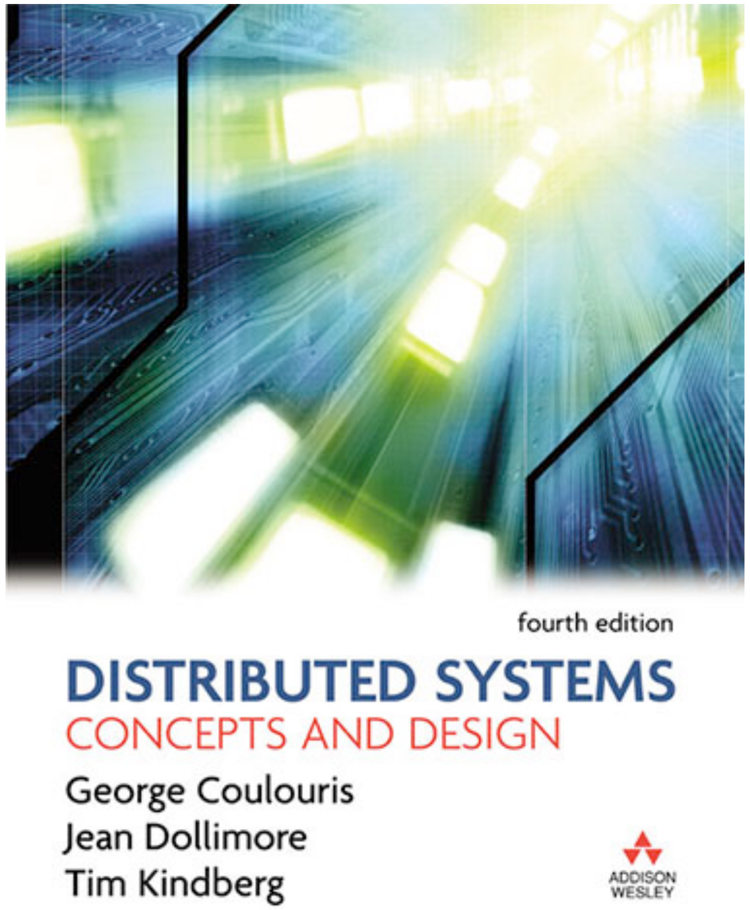
  but which share an aim that is fundamental in distributed systems

  for a set of distributed processes to coordinate their actions or to agree on one or more values.

Failure Assumptions

- Each pair of processes is connected by reliable channels.
  - A reliable channel *eventually* delivers a message to the recipient's input buffer.
- No process failure implies a threat to the other processes' ability to communicate.
  - None of the processes depends upon another to forward messages.
- Unless we state otherwise, processes only fail by crashing.

Distributed Mutual Exclusion



Problem: Coordinate Access to Shared Resources

- Distributed processes often need to coordinate their activities.
- If a collection of processes share a resource or collection of resources, then often mutual exclusion is required to prevent interference and ensure consistency when accessing the resources.
- Critical section problem in the domain of operating systems.
- BUT in a distributed system, neither shared variables nor facilities supplied by a single local kernel can be used to solve the problem.
- We require a distributed mutual exclusion: one that is based solely on message passing!!!

Model (Without Failures)

- We consider a system of N processes p_i, $i = 1,...,N$ that do not share variables.
- The processes access common resources, but they do so in a critical section.
- The system is asynchronous.
- Processes do not fail.
- Message delivery is reliable: any message sent is eventually delivered intact, exactly once.
- Client processes are well-behaved and spend a finite time accessing resources within their CSs.

Critical Section (CS)

- The application-level protocol for executing a CS is as follows:
  - enter(): enter a critical section - block if necessary.
  - resourceAccess(): access shared resources in critical section.
  - exit(): leave critical section - other processes may now enter.

Requirements for ME

- A mutual exclusion algorithm should satisfy the following properties:
  - [ME1] Safety: at most one process can execute in the CS at a time.
  - [ME2] Liveness: requests to enter and exit the CS eventually succeed.
  - [ME3] Ordering: if one request to enter the CS *happened-before* another, then entry to the CS is granted in that order.
- The first property is absolutely necessary (correctness).
- The other two properties are considered important in ME algorithms.

On ME Requirements: Liveness

- [ME2] Liveness: requests to enter and exit the CS eventually succeed.
- Condition ME2 implies freedom from both deadlock and starvation.
  - A deadlock would involve two or more processes becoming stuck indefinitely while attempting to enter or exit the critical section, by virtue of their mutual interdependence.
  - Even without a deadlock, a poor algorithm might lead to starvation: the indefinite postponement of entry for a process that has requested it.
- The absence of starvation is a fairness condition.

On ME Requirements: Ordering

- [ME3] Ordering: if one request to enter the CS happened-before another, then entry to the CS is granted in that order.

- If a solution grants entry to the CS in happened-before order, and if all the requests are related by happened-before, then it is not possible for a process to enter the CS more than once while another waits to enter.

On ME Requirements: Ordering

- [ME3] Ordering: if one request to enter the CS happened-before another, then entry to the CS is granted in that order.

- If a solution grants entry to the CS in happened-before order, and if all the requests are related by happened-before, then it is not possible for a process to enter the CS more than once while another waits to enter.

- Example: a multi-threaded process may continue with other processing while a thread waits to be granted entry to a CS.
  - During this time, it might send a message to another process, which consequently also tries to enter the CS.
  - ME3 specifies that the first process be granted access before the second.

Performance Criteria

- Algorithms for ME can be evaluated by several metrics, such as:
  - The bandwidth consumed, which is proportional to the number of messages sent in each *entry* and *exit* operation.
  - The client delay incurred by a process at each *entry* and *exit* operation.
  - The algorithm's effect upon the throughput of the system: the rate at which the collection of processes as a whole can access the CS, given that some communication is necessary between successive processes.
    - Measured using the ***synchronization delay*** *(SD) between one process exiting the CS and the next process entering it*.
    - The throughput is greater when the synchronization delay is shorter.

$$\text{throughput} = \frac{1}{(SD + E)}$$

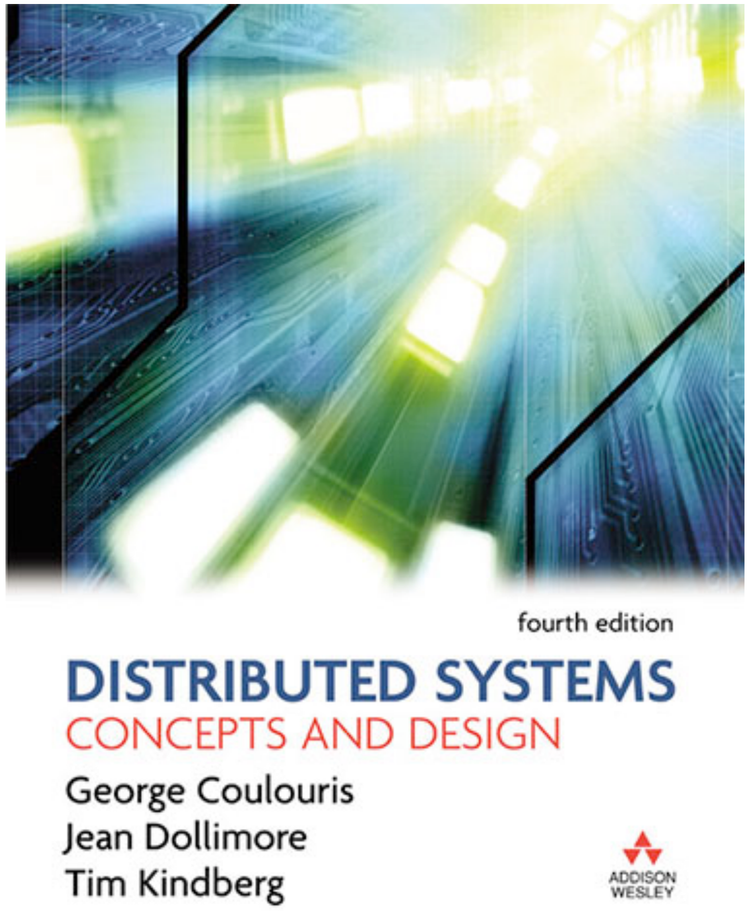
where E = average CS execution time

Design of Distributed ME Algorithms

- Complex because these algorithms have to deal with
  - unpredictable message delays
  - incomplete knowledge of the system state
- Three basic approaches:
  - Token based approaches
  - Non-token based approaches
  - Quorum based approaches

Distributed Mutual Exclusion

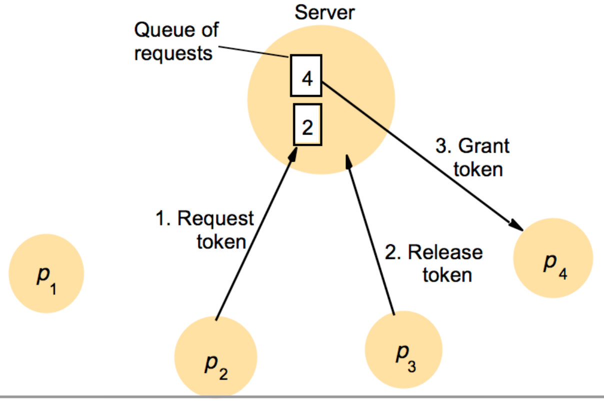
Token based approaches



[Distributed ME] Token Based Algorithms

- A unique token (PRIVILEGE msg) is shared among the processes.
- A process is allowed to enter its CS if it possesses the token.
- The process continues to hold the token until the execution of the CS is over.
- Mutual exclusion is ensured because the token is unique.
- The algorithms based on this approach essentially differ in the way a process carries out the search for the token.

The Central Server Algorithm



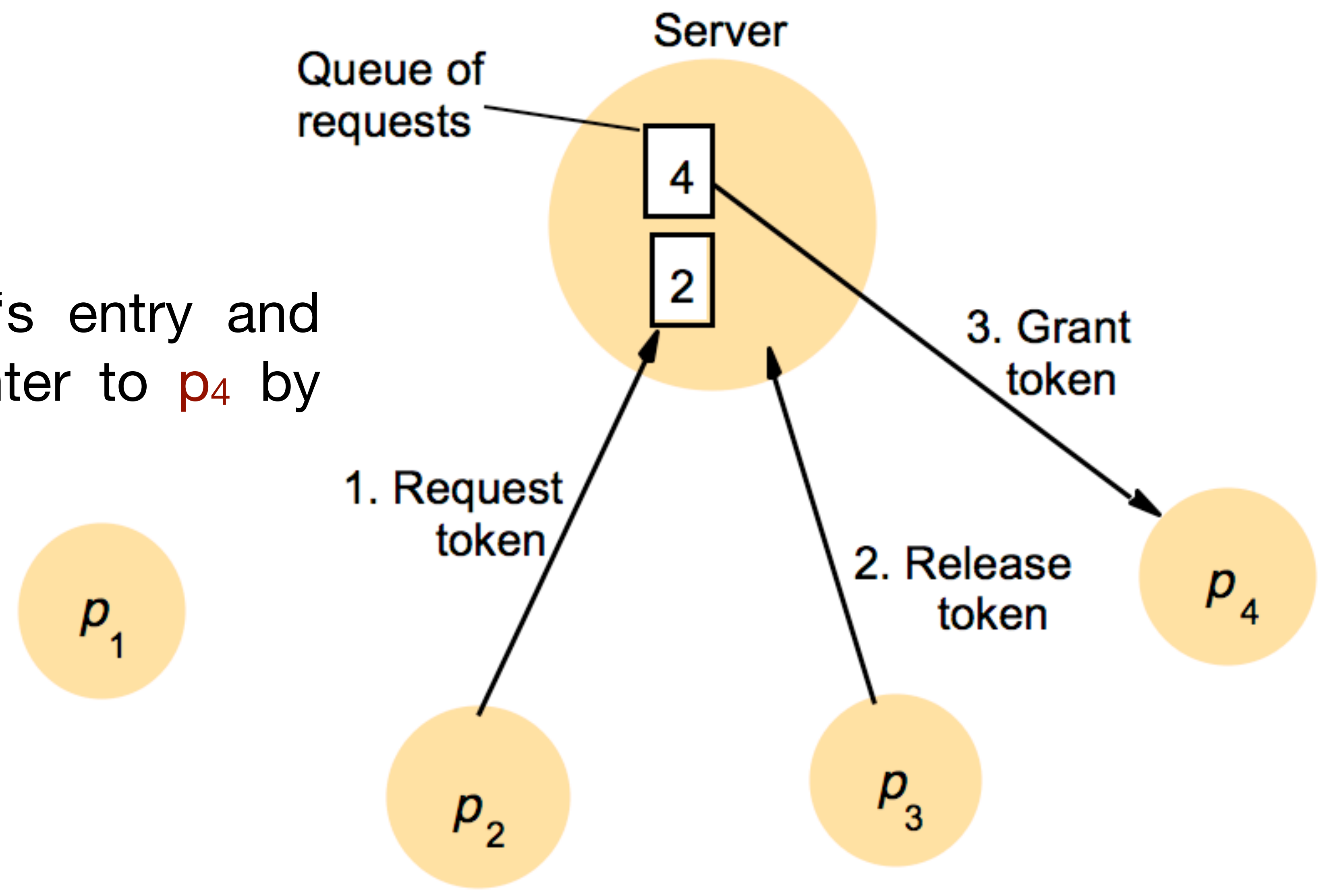
- The simplest way to achieve mutual exclusion is to employ a server that grants permission to enter the CS.

- To enter a CS, a process sends a requests to the server and awaits a reply from it.
- The reply constitutes a token signifying permission to enter the CS.
- If no other process has the token at the time of the request then the server replies immediately, granting the token.
- If the token is currently held by another process, then the server does not reply but queues the request.
- On exiting the CS, a message is sent to the server, giving it back the token.
- If the queue of waiting process is not empty, then the server chooses the oldest entry in the queue, removes it and replies to the corresponding process.
- The chosen process then holds the token.

Algorithm

[The Central Server Algorithm] Example

- Process p_1 does not currently require entry to the CS.
- Process p_2's request has been appended to the queue, which already contained p_4's request.
- Process p_3 exits the CS.
- The server removes p_4's entry and grants permission to enter to p_4 by replying to it.



Performance of the Central Server Algorithm

- Entering the CS:
    - It takes 2 messages: a *request* followed by a *grant*.
    - It delays the requesting process (client) by the time for this round-trip.
- Exiting the CS:
    - It takes 1 *release* message.
    - Assuming asynchronous message passing, this does not delay the exiting process.
- The server may become a performance bottleneck for the system as a whole.
    - Synchronization delay: time taken for a round-trip (a *release* msg to the server, followed be a *grant* msg to the next process to enter the CS).

Homework

- Provide a formal specification in CSP of the central server algorithm.

- Given the assumption that no failures occur, informally discuss:

    - why the safety and liveness conditions [ME1 and ME2] are met by the central server algorithm

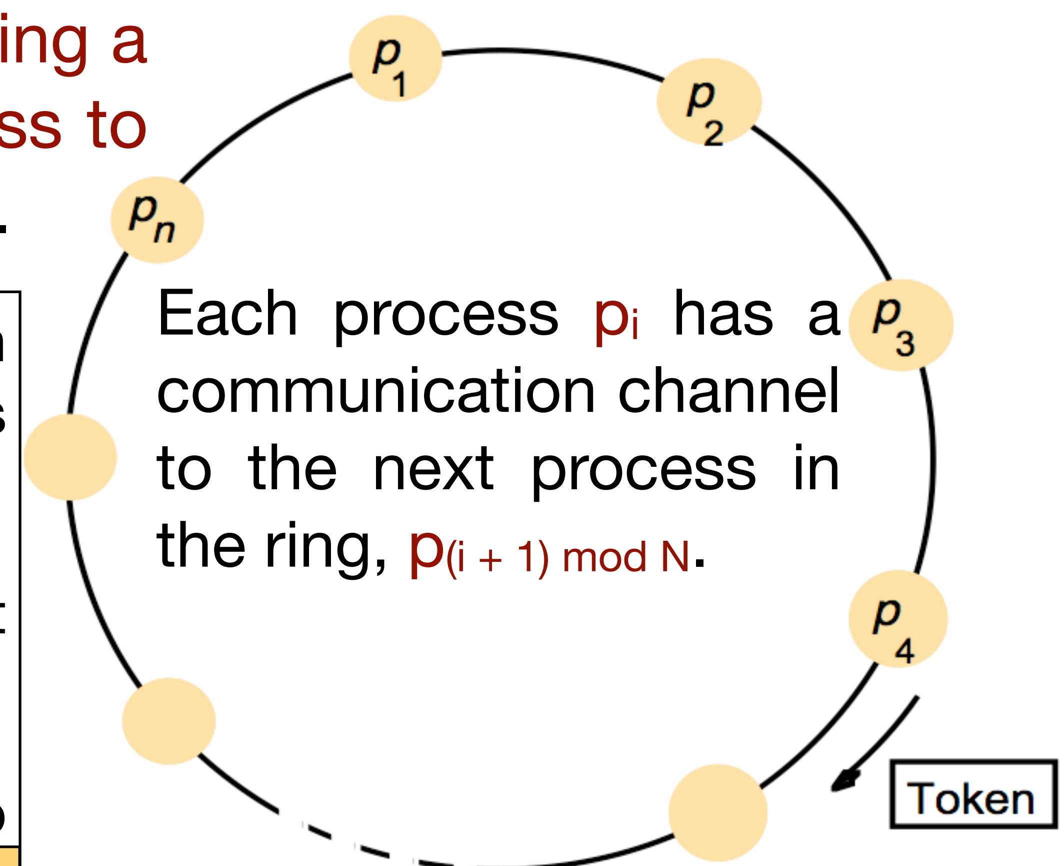
    - the algorithm does **not** satisfy property ME3

A Ring-Based Algorithm

- Logical ring: one of the simplest ways to arrange a ME between N processes without requiring an additional process.
- The ring topology may be unrelated to the physical interconnections between the underlying computers.
- Basic idea: exclusion is conferred by obtaining a token in the form of a message from process to process in a *single direction around the ring*.

Algorithm

- If a process does not require to enter the CS when it receives the token, then it immediately forwards the token to its neighbour.
- A process that requires the token waits until it receives it, but retains it.
- To exit the CS, the process sends the token on to its neighbour.



Each process p_i has a communication channel to the next process in the ring, $p_{(i+1) \bmod N}$.

Homework

- Given the assumption that no failures occur, informally discuss why the safety and liveness conditions [ME1 and 2] are met by the ring-based algorithm.

- Informally discuss why the ring-based algorithm does not necessarily satisfy the ordering property [ME3].

Performance of the Ring-Based Algorithm

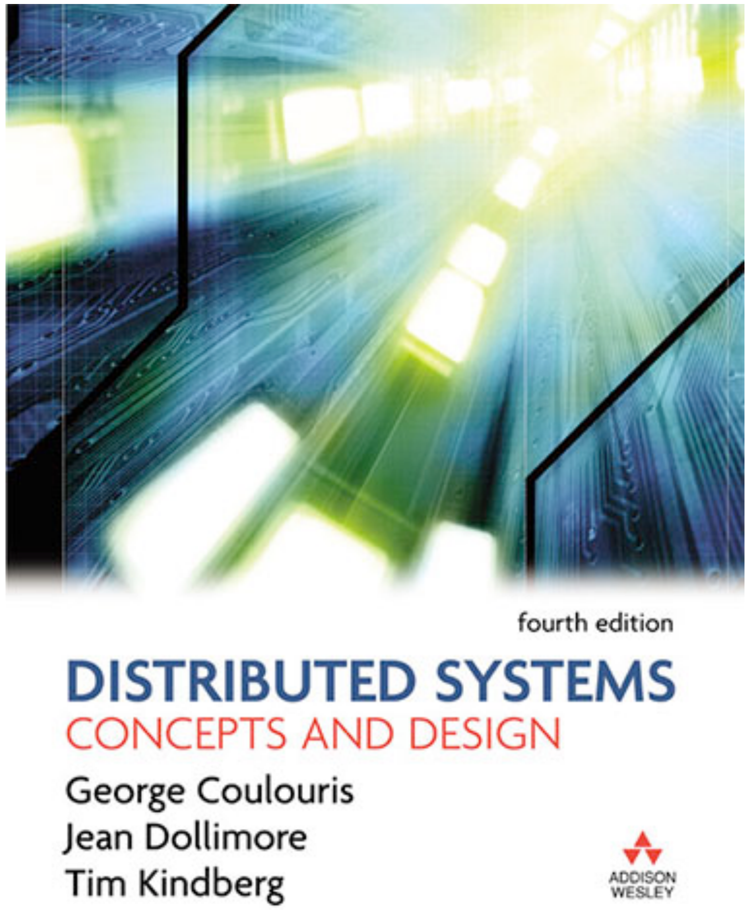
- The algorithm continuously consumes network bandwidth, expect when a process is inside the critical section.
  - The processes send messages around the ring even when no process requires entry to the CS.
- The delay experienced by a process requesting entry to the CS is between 0 messages (when it has just received the token) and N messages (when it has just passed on the token).
- To exit the CS requires only one message.
- The synchronization delay between one process's exit from the CS and the next process's entry is anywhere from 1 to N message transmissions.

Homework

- Provide a formal specification in CSP of the ring-based algorithm.

Distributed Mutual Exclusion

Non-token based approaches



[Distributed ME] Non-token Based Algorithms

- Two or more successive rounds of messages are exchanged among the processes to determine which process will enter the CS next.

- A process enters the CS when an assertion, defined on its local variables, becomes true.

- Mutual exclusion is enforced because the assertion becomes true *only at one site at any given time*.

Lamport's Algorithm

- Requires communication channels to deliver messages in FIFO order.
- Satisfies conditions ME1, ME2 and ME3.
- Based on Lamport logical clocks: timestamped requests for entering the CS.
- Timestamp: (clock value, id of the process)
- Every process p_i keeps a queue, $request\_queue_i$, which contains mutual exclusion requests ordered by their timestamps.
- The algorithm executes CS requests in the increasing order of timestamps.
- **Timestamps are totally ordered!!** Example: (1, 1) < (1, 2)

Extension of Happened-Before Relation (→)

- → defines a partial ordering of events in the system.

  CR1: If ∃ process p_i such that $e \rightarrow_i e'$, then $L_i(e) < L_i(e')$.

  CR2: If a is the sending of a message by p_i and b is the receipt of the same message by p_j, then $L_i(a) < L_j(b)$.

  CR3: If e, e', e'' are three events such that $L(e) < L(e')$ and $L(e') < L(e'')$ then $L(e) < L(e'')$.

- A total ordering ⇒ requires the further rule:

  CR4: a (in p_i) $\Rightarrow$ b (in p_j) if and only if

  either $L_i(a) < L_j(b)$
  or $L_i(a) = L_j(b) \wedge p_i < p_j$

  for some suitable ordering $<$ of the processes.

Lamport's Algorithm [1978]

Requesting the CS

Process p_i updates its local clock and timestamps the request (ts_i)
Process p_i broadcasts a REQUEST(ts_i, i) to all the other processes
Process p_i places the request on $request\_queue_i$

On Receiving REQUEST(ts_i, i) from a process p_i

Process p_j places p_i's request on $request\_queue_j$

Process p_j returns a timestamped REPLY msg to p_i

Executing the CS

Process p_i enters the CS when the following two conditions hold:

- L1: p_i has received a msg with timestamp larger than (ts_i, i) from all other processes
- L2: p_i's request is at the top of $request\_queue_i$

Releasing the CS

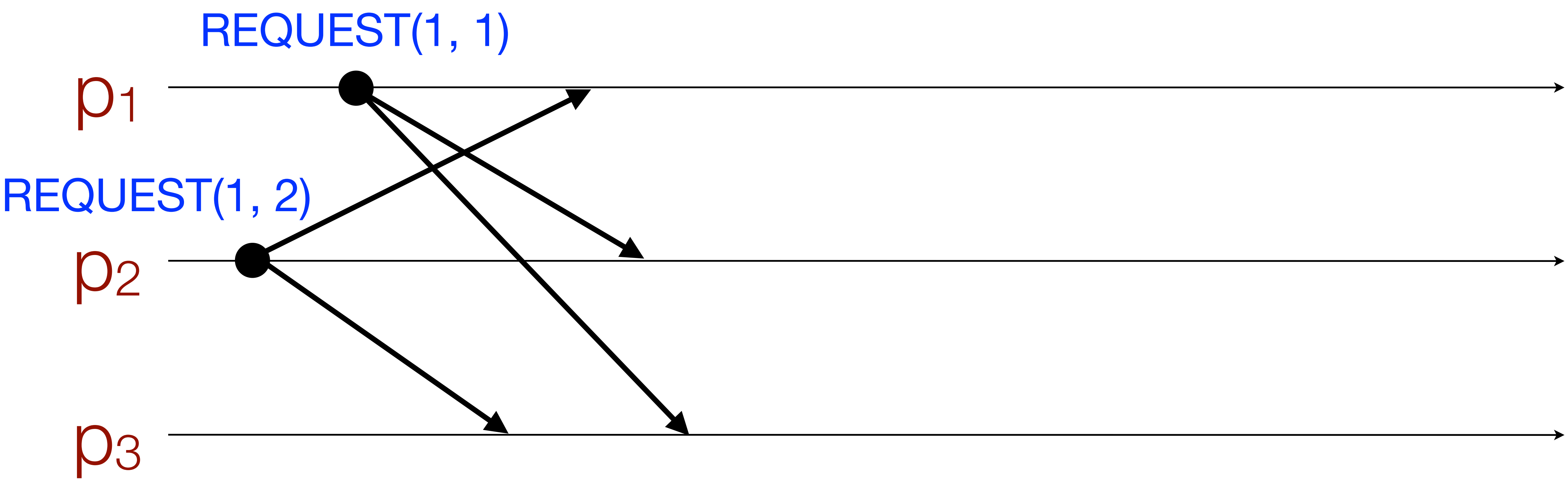
Process p_i removes its request from the top of $request\_queue_i$
Process p_i broadcasts a timestamped RELEASE msg to all other processes

On Receiving RELEASE from a process p_i

Process p_j removes p_i's request from its request queue $request\_queue_j$

The Algorithm in Action: Entering a CS

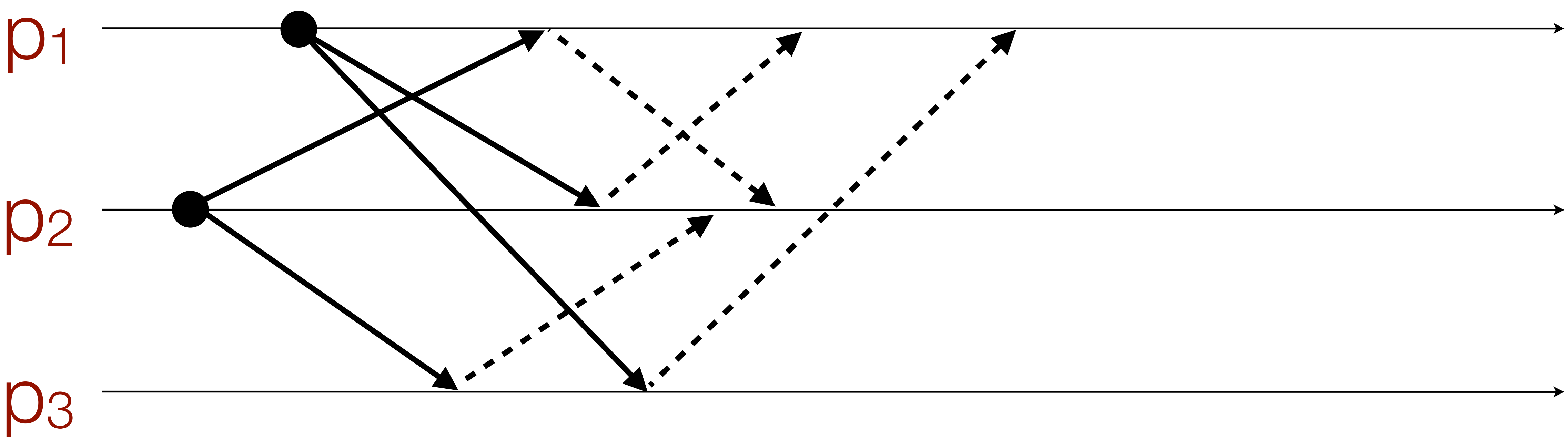
- p_1 and p_2 send out REQUEST messages for the CS to the other processes



request_queue$_1$	(1, 1)
request_queue$_2$	(1, 2)

The Algorithm in Action: Entering a CS

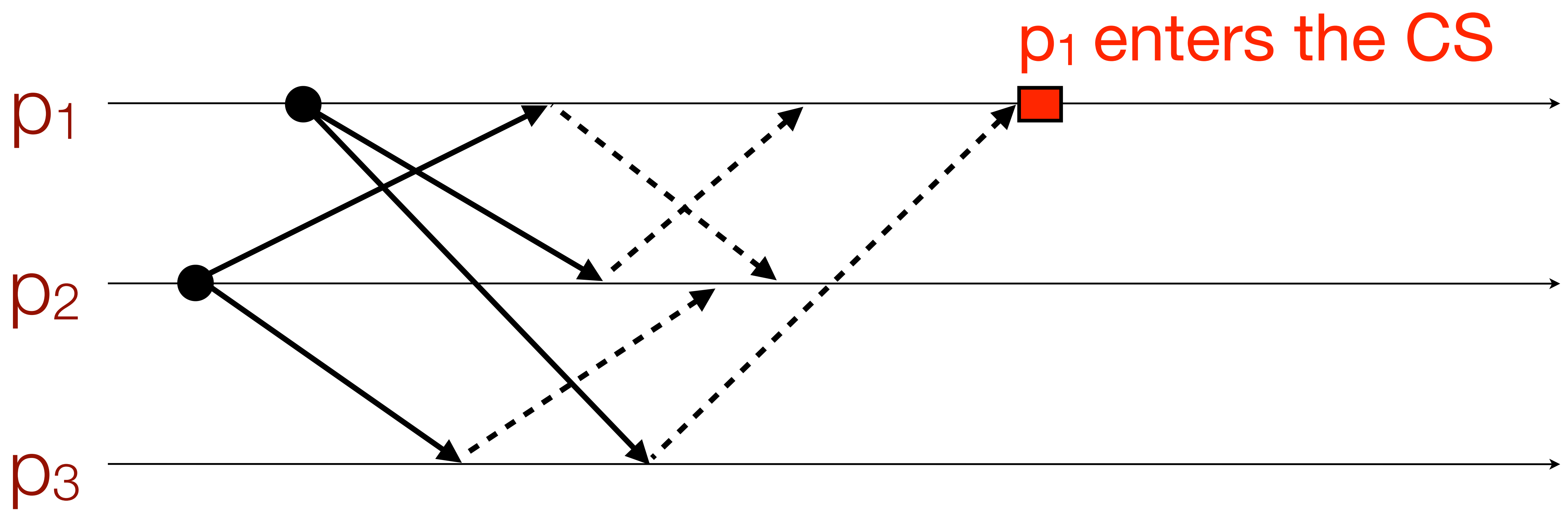
- Both p_1 and p_2 have received timestamped REPLY msgs from all processes



request_queue$_1$	(1, 2)	(1, 1)
request_queue$_2$	(1, 2)	(1, 1)

The Algorithm in Action: Entering a CS

- Both p_1 and p_2 have received timestamped REPLY msgs from all processes

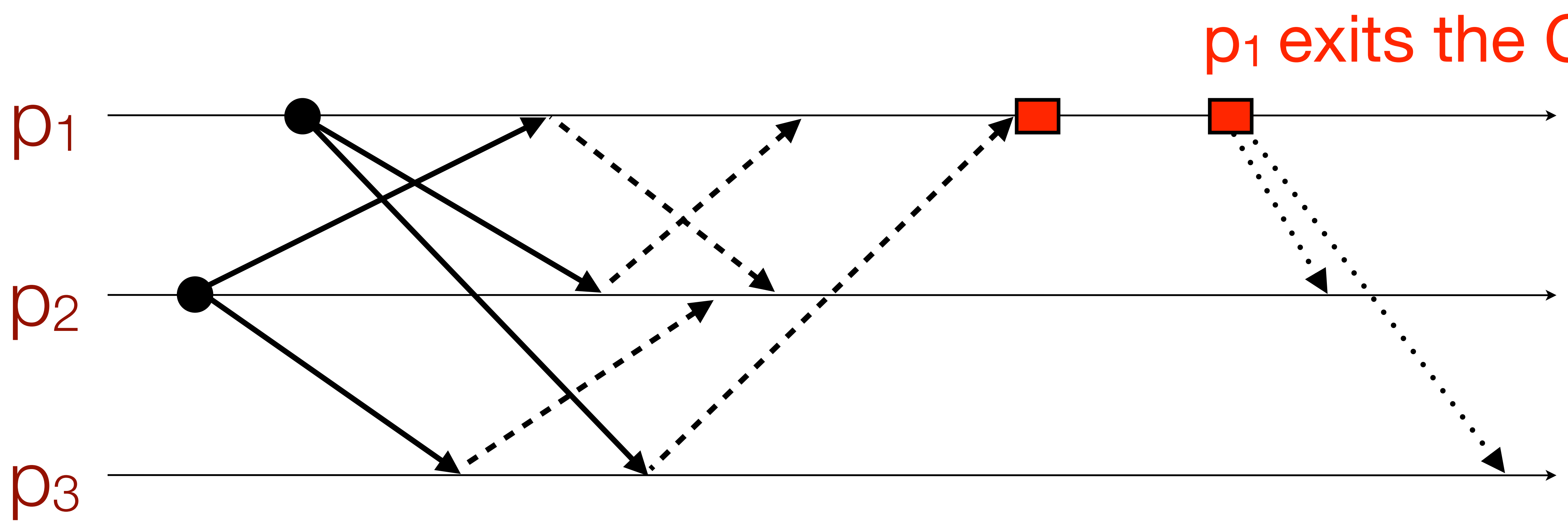


$request\_queue_1$	(1, 2)	(1, 1)
$request\_queue_2$	(1, 2)	(1, 1)

- L1: p_1 has received a msg with timestamp larger than (1, 1) from all other processes
- L2: p_1's request is at the top of $request\_queue_1$

The Algorithm in Action: Exiting a CS

- p_1 exits and sends RELEASE msgs to all other processes

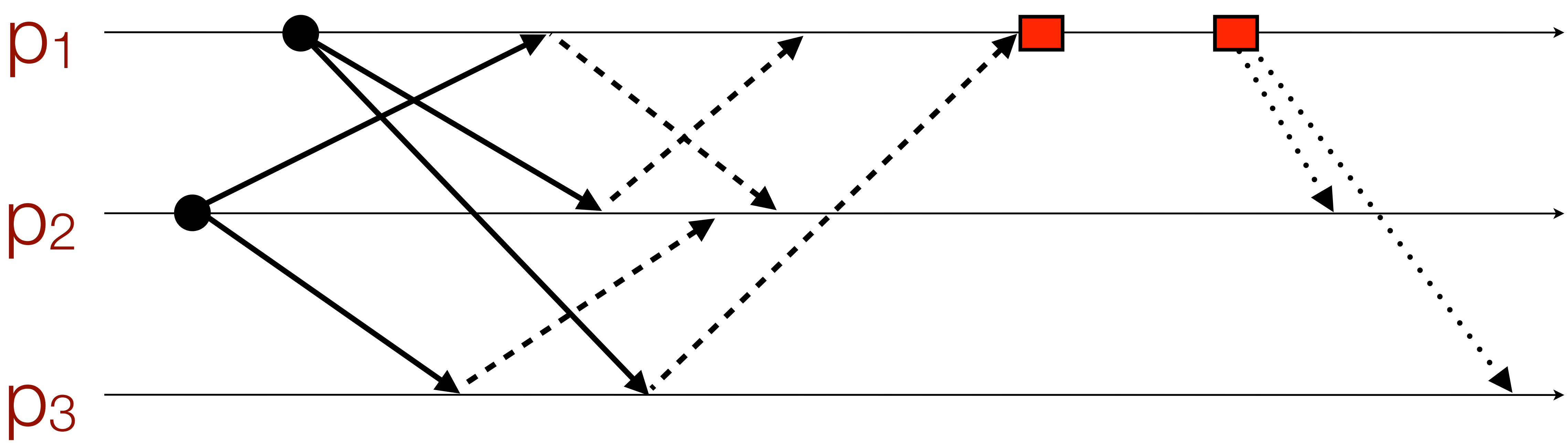


request_queue$_1$ | (1, 2) |

request_queue$_2$ | (1, 2) | (1, 1) |

The Algorithm in Action: Exiting a CS

- p_1 exits and sends RELEASE msgs to all other processes



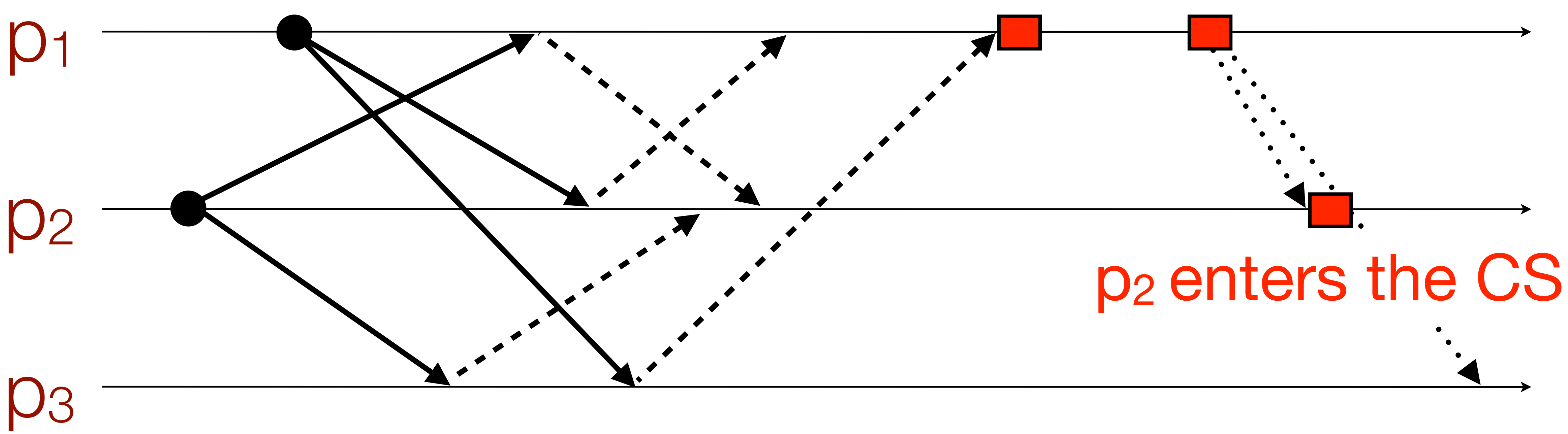
request_queue$_1$ (1, 2)

request_queue$_2$ (1, 2)

On Receiving RELEASE from process p_1
- Process p_2 removes p_1's request from its request queue request_queue$_2$

The Algorithm in Action: p_2 enters the CS...

- p_1 exits and sends RELEASE msgs to all other processes



request_queue$_1$ (1, 2)

request_queue$_2$ (1, 2)

- L1: p_2 has received a msg with timestamp larger than (1, 2) from all other processes
- L2: p_2's request is at the top of request_queue$_2$

Theorem

Lamport's algorithm achieves mutual exclusion (property ME1).

Proof [by contradiction]:

- suppose two processes p_i and p_j are executing the CS concurrently

➡ L1 and L2 must hold at both sites concurrently

➡ at some instant in time, say t, both p_i and p_j have their own requests at the top of their request_queue and condition L1 holds at them

- Without loss of generality, assume that $(ts_i, i) < (ts_j, j)$

- From L1 and FIFO property, at instant t the request of p_i must be in $request\_queue_j$ when p_j was executing its CS

➡ p_j's own request is at the top of $request\_queue_j$ when a smaller timestamp request, (ts_i, i) from p_i, is present in the queue - a contradiction!!

Performance of Lamport's Algorithm

- For each CS execution, the algorithm requires
  - (N - 1) REQUEST messages
  - (N - 1) REPLY messages
  - (N - 1) RELEASE messages
- Thus, the algorithm requires 3(N - 1) messages per CS invocation.
- The client delay in requesting entry is a round-trip time.
- The synchronization delay is 1 msg transmission (average message delay).

Ricart and Agrawala's Algorithm [1981]

- Basic idea: processes that require entry to a CS multicast a request message, and can enter it only when all the other processes have replied to this message.

- BUT the algorithm does NOT require communication channels to be FIFO.

- Each process p_i keeps a Lamport clock, updated according to LC1 and LC2.

- Messages requesting entry are of the form $<T, p_i>$, where T is the sender's timestamp and p_i is the sender's identifier.

- Every process records its state of being outside the CS (RELEASED), wanting entry (WANTED) or being in the CS (HELD) in a variable *state*.

Ricart and Agrawala’s Algorithm [1981]

On initialization
state := RELEASED;

To enter the Critical Section
state := WANTED;
Multicast REQUEST to all processes;
T := request’s timestamp;
Wait until (number of replies received = (N – 1));
state := HELD;

On receipt of a request $\langle T_i, p_i \rangle$ 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;

Ricart and Agrawala's Algorithm [1981]

On initialization
state := RELEASED;

To enter the Critical Section
state := WANTED;
Multicast REQUEST to all processes;
T := request's timestamp;
Wait until (number of replies received = (N – 1));
state := HELD;

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

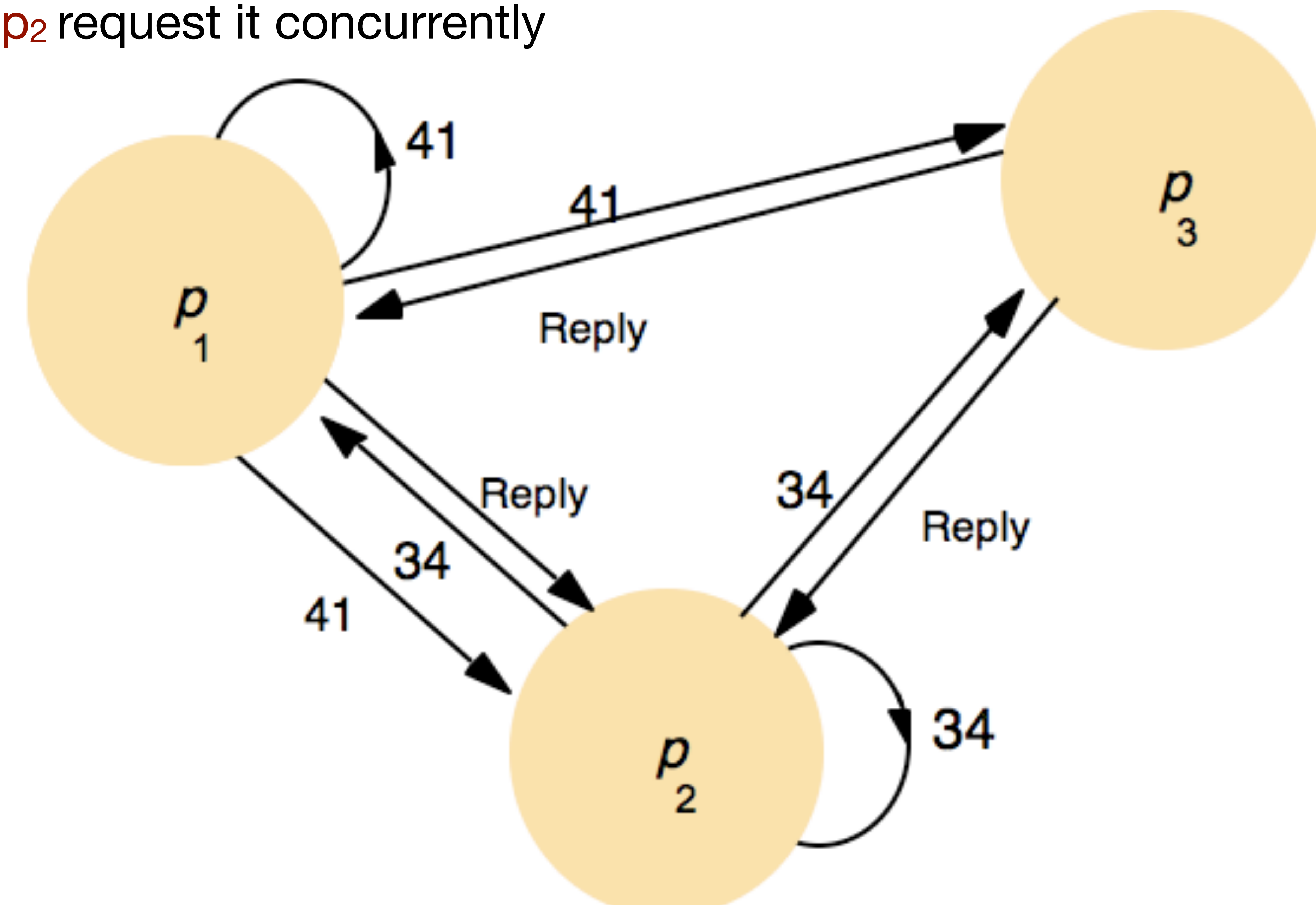
To exit the Critical Section
state := RELEASED;
reply to any queued requests;

If two or more processes request entry at the same time, then whichever process's request bears the lowest timestamp will be the first to collect N-1 replies, granting it entry next.

In case of equal timestamps, the requests are ordered according to the process identifiers.

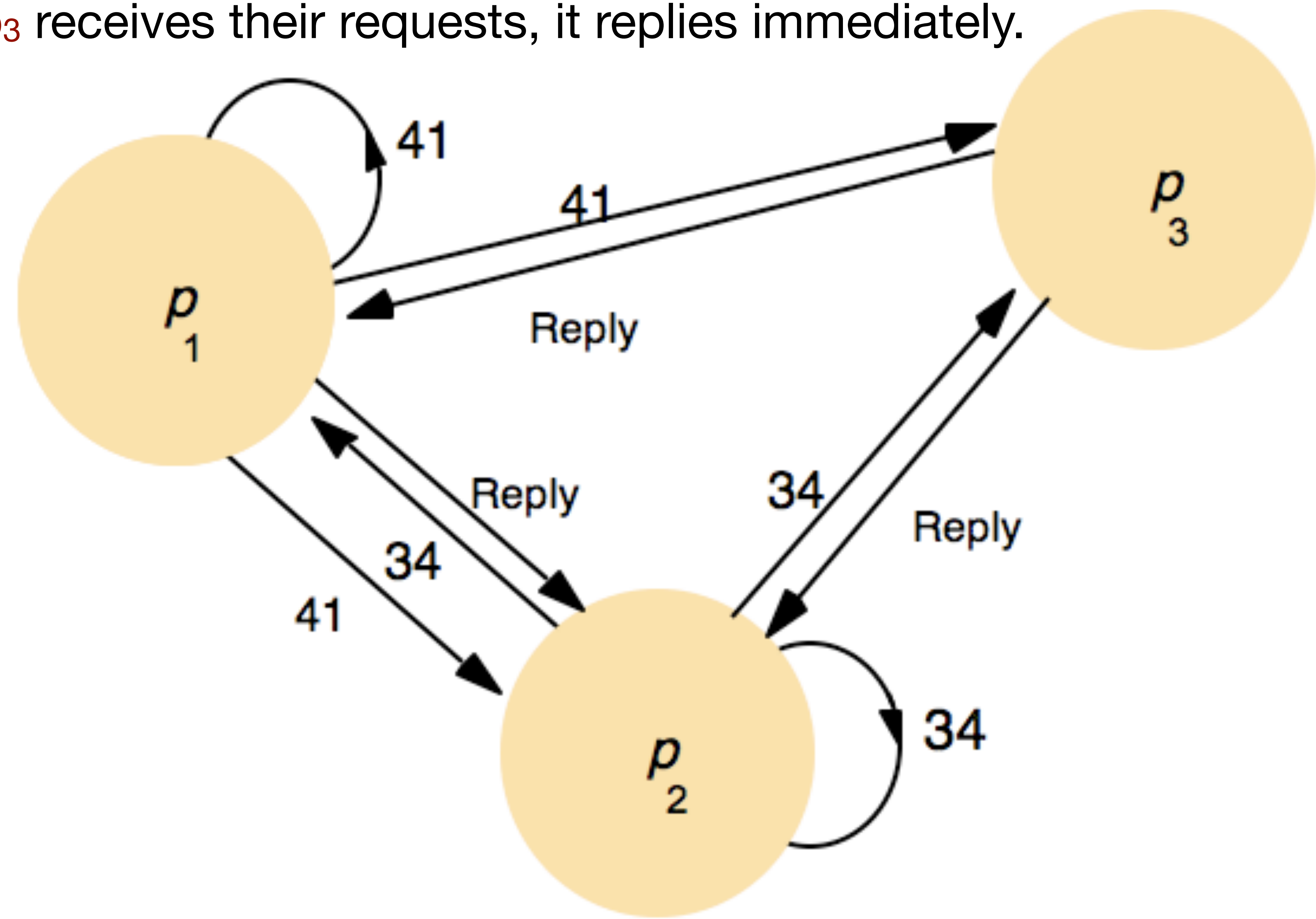
[Ricart and Agrawala's Algorithm] Example

- p_3 not interested in entering the CS
- p_1 and p_2 request it concurrently



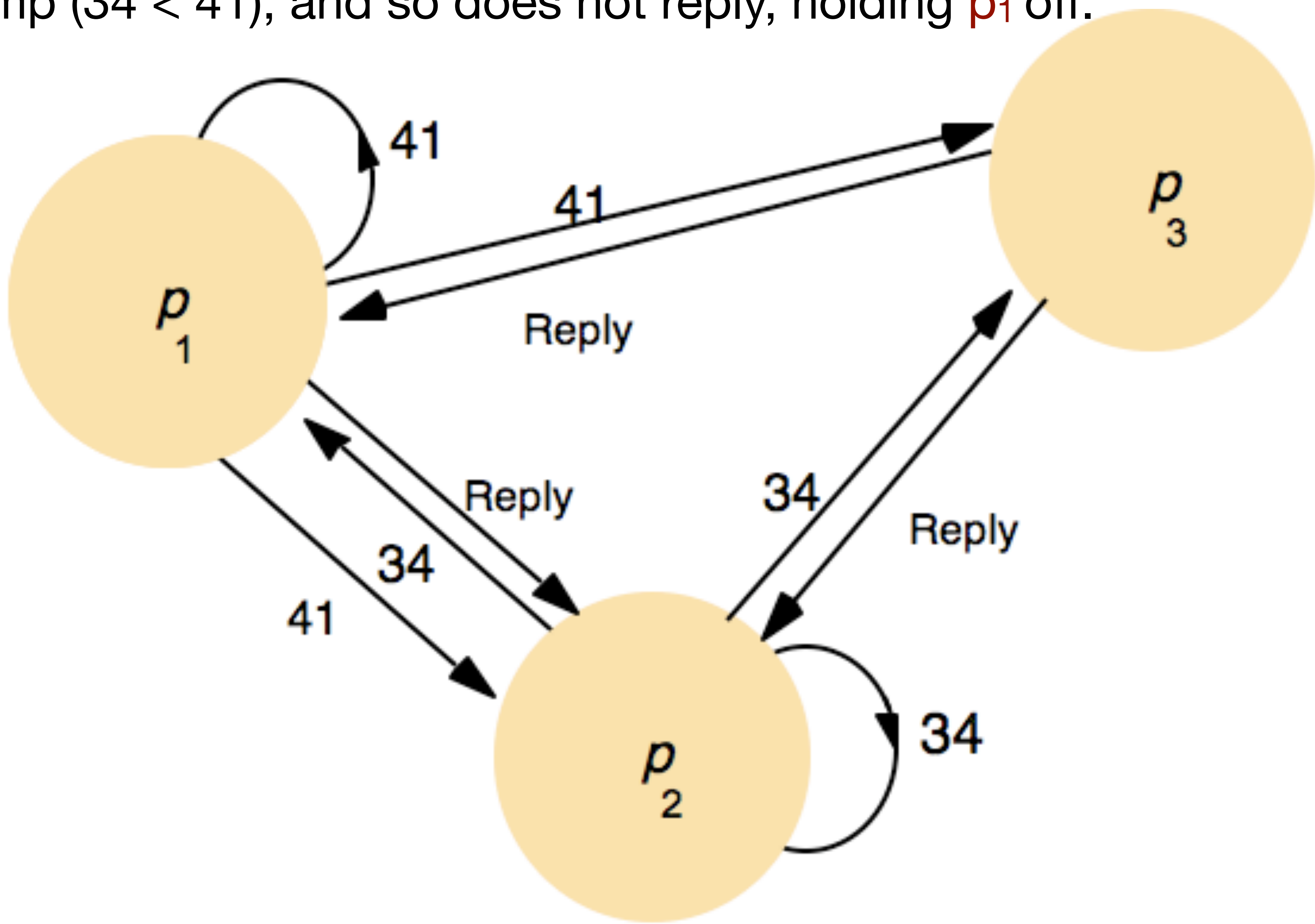
[Ricart and Agrawala's Algorithm] Example

- The timestamp of p_1's request is 41, that of p_2 is 34.
- When p_3 receives their requests, it replies immediately.



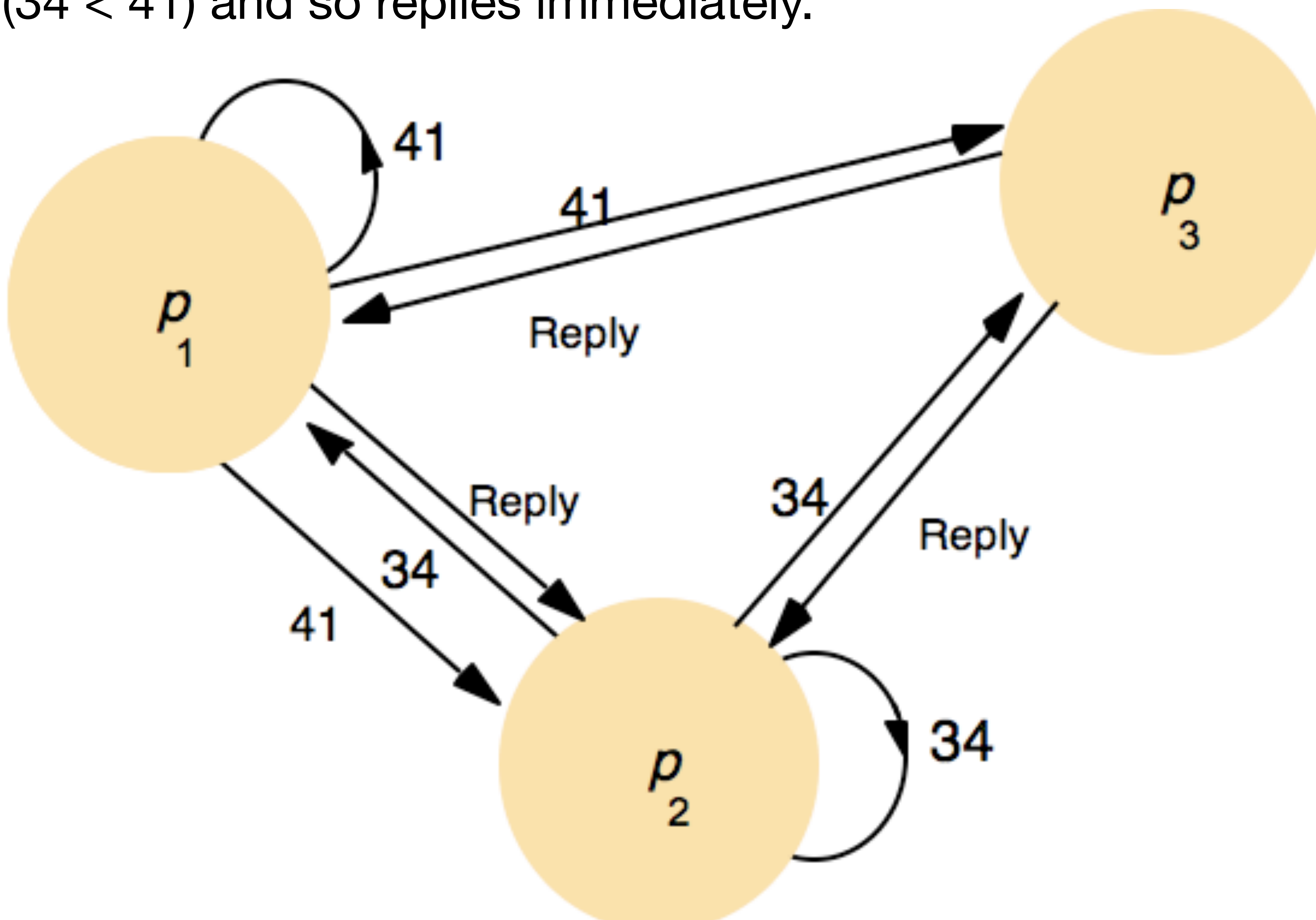
[Ricart and Agrawala's Algorithm] Example

- When p_2 receives p_1's request, it finds its own request has the lower timestamp (34 < 41), and so does not reply, holding p_1 off.



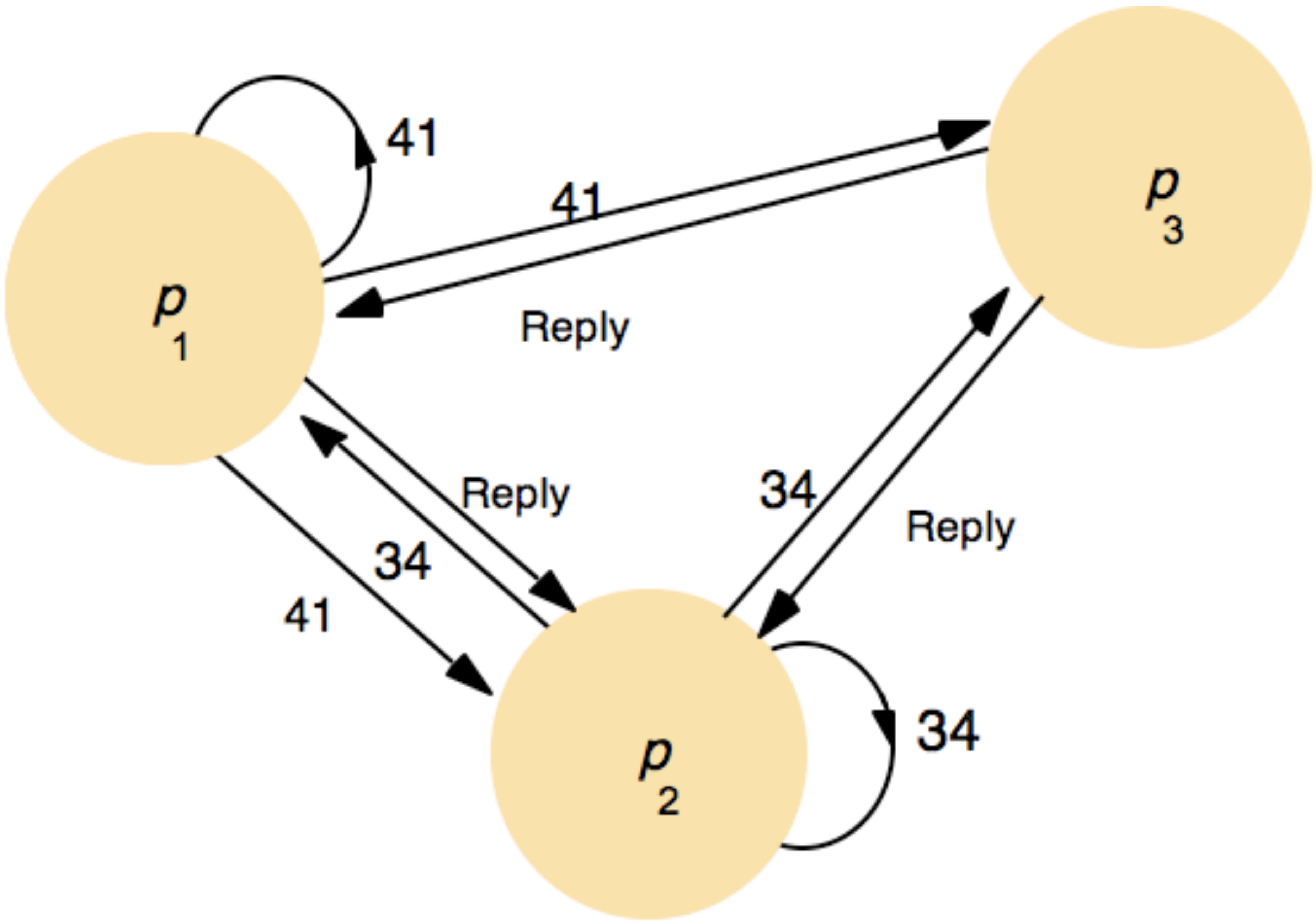
[Ricart and Agrawala's Algorithm] Example

- However, p_1 finds that p_2's request has a lower timestamp than that of its own request (34 < 41) and so replies immediately.



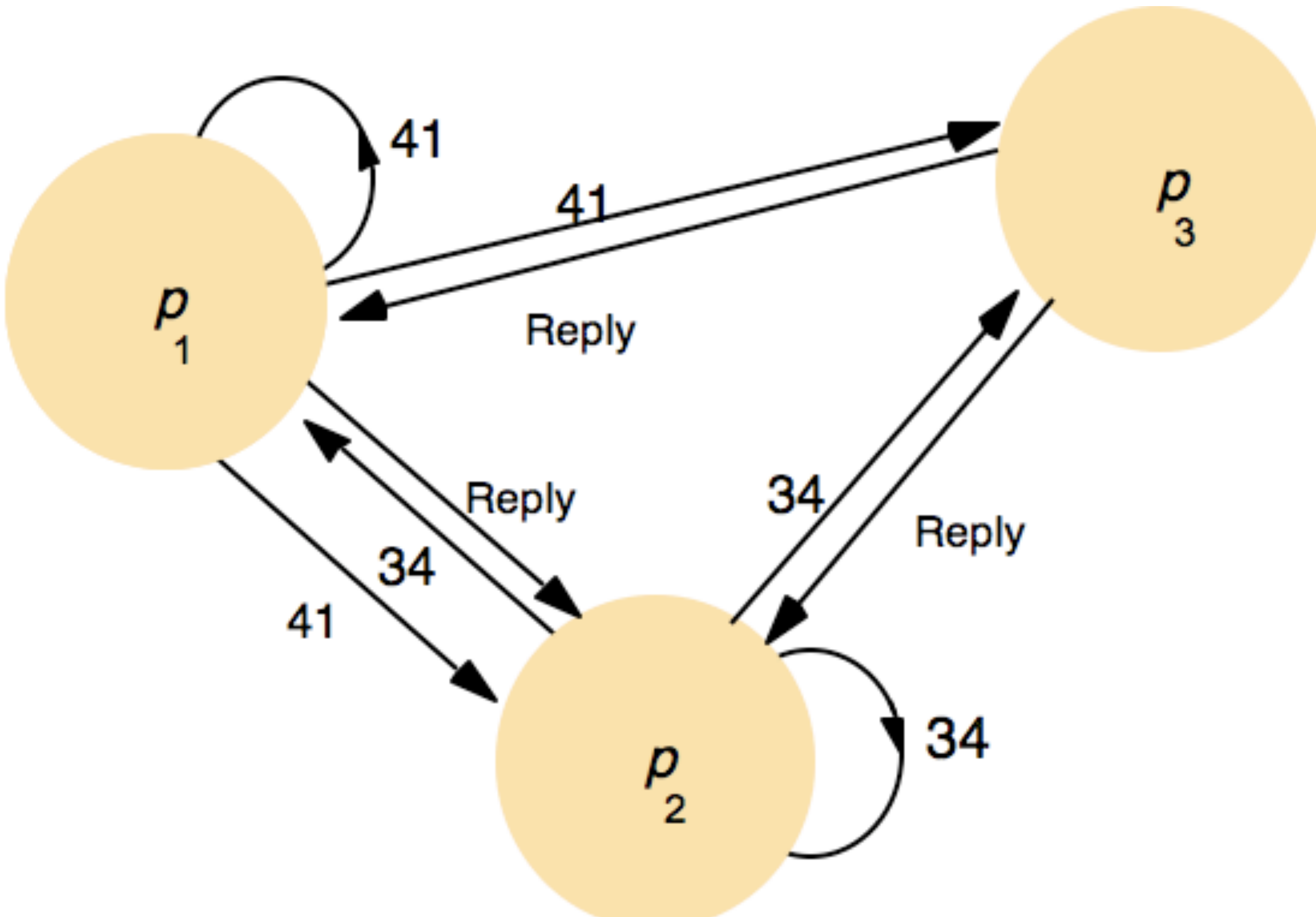
[Ricart and Agrawala's Algorithm] Example

- On receiving the 2nd reply, p_2 can enter the CS.



[Ricart and Agrawala's Algorithm] Example

- When p_2 exits the CS, it will reply to p_1's request and so grant it entry.



Homework

- Prove that Ricart and Agrawala's algorithm achieves the safety property ME1.

  Idea: if it were possible for two processes p_i and p_j ($i \neq j$) to enter the CS at the same time, then both of those processes would have to have replied to the other.

  But since the pairs $\langle T_i, p_i \rangle$ are totally ordered, this is impossible.

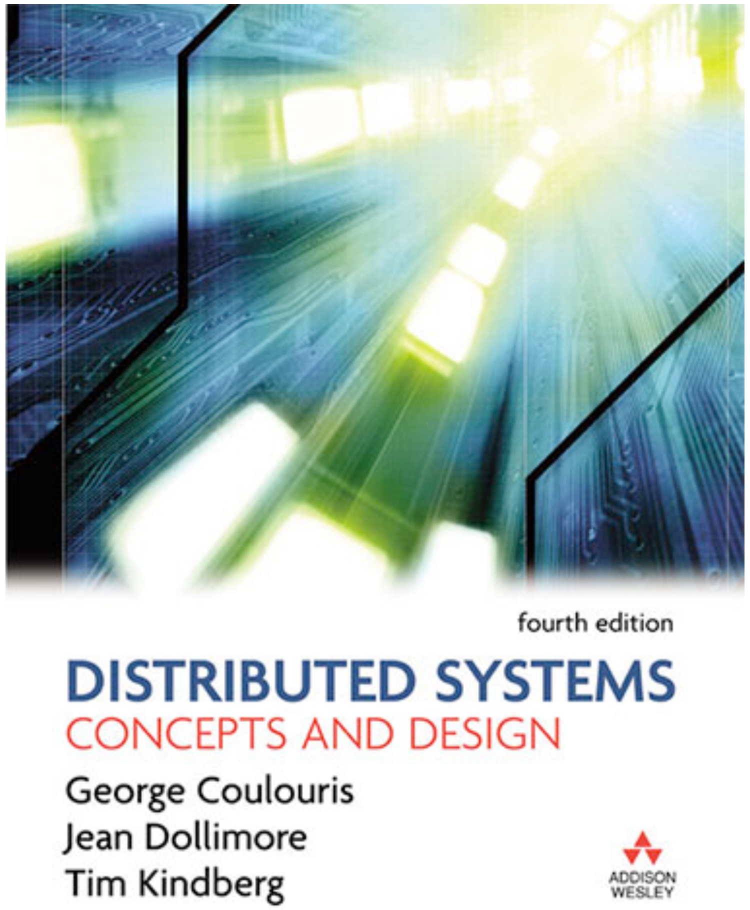
- Verify, in a similar way, that the algorithm also meets requirements ME2 and ME3.

Performance of the Ricart-Agrawala's Algorithm

- Gaining entry takes 2(N-1) messages:
  - N-1 to multicast the request
  - followed by N-1 replies
- The client delay in requesting entry is a round-trip time.
- The synchronization delay is 1 message transmission time.
- Ricart and Agrawala refined the algorithm so that it requires N messages to obtain entry in the worst (and common) case.
  [Raynal, M. (1988). *Distributed Algorithms and Protocols*. Wiley]

Distributed Mutual Exclusion

Quorum-Based Mutual Exclusion Algorithms



[Distributed ME] Quorum-Based Algorithms

- Each process requests permission to execute the CS from a subset of processes (QUORUM).

- The quorums are formed in such a way that when two processes concurrently request access to the CS
    - at least one process receives both the requests
    - this process is responsible to make sure that only one request executes the CS at any time.

Quorum-Based Mutual Exclusion Algorithms

- Idea:
  - processes vote for one another to enter the CS
  - a process can vote only one process per session
  - a "candidate" process must collect sufficient votes to enter the CS
    - a process does **NOT** need permission from ALL other processes, but only from a SUBSET of the processes (QUORUM)
- Intersection property: for every quorum $V_i, V_j \subseteq \{p_1, p_2, ..., p_N\}$, $V_i \cap V_j \neq \emptyset$.
  - Example: {2, 5, 7} and {5, 7 9} are suitable quorums, {1, 2, 3} and {2, 5, 7} are not suitable quorums
- Algorithms basically differ in how the quorum is constructed.

Quorum-Based Mutual Exclusion Algorithms

- A simple protocol works as follows:
  - let p_i be a process in quorum V_i
  - if p_i wants to invoke mutual exclusion, it requests permission from all processes in its quorum V_i
  - every process does the same to invoke mutual exclusion
  - due to the Intersection property, quorum Vi contains at least on process that is common to the quorum of every other site
  - these common processes send permission (i.e., vote) to only one process at any time
  - Thus, mutual exclusion is guaranteed.

Maekawa's Algorithm: Quorums

- The quorums are constructed to satisfy the following conditions:

M1 $\forall i\ \forall j : i \neq j,\ 1 \leq i, j \leq N$, then $V_i \cap V_j \neq \emptyset$

M2 $\forall i : 1 \leq i \leq N$, then $p_i \in V_i$

M3 $\forall i : 1 \leq i \leq N$, then $|V_i| = K$

M4 any process p_j is contained in K number of V_is, $1 \leq i, j \leq N$

- Optimal solution: $N = K(K - 1) + 1$, which gives $K = \sqrt{N}$

Maekawa's Algorithm: Quorums

- The quorums are constructed to satisfy the following conditions:

M1 $\forall i\ \forall j : i \neq j,\ 1 \leq i, j \leq N$, then $V_i \cap V_j \neq \emptyset$

M2 $\forall i : 1 \leq i \leq N$, then $p_i \in V_i$

necessary for correctness

M3 $\forall i : 1 \leq i \leq N$, then $|V_i| = K$

M4 any process p_j is contained in K number of V_is, $1 \leq i, j \leq N$

- Optimal solution: $N = K(K - 1) + 1$, which gives $K = \sqrt{N}$

Maekawa's Algorithm: Quorums

- The quorums are constructed to satisfy the following conditions:

M1 $\forall i\ \forall j : i \neq j,\ 1 \leq i, j \leq N$, then $V_i \cap V_j \neq \emptyset$

M2 $\forall i : 1 \leq i \leq N$, then $p_i \in V_i$

necessary for correctness

M3 $\forall i : 1 \leq i \leq N$, then $|V_i| = K$

M4 any process p_j is contained in K number of V_is, $1 \leq i, j \leq N$

desiderable features

- Optimal solution: $N = K(K - 1) + 1$, which gives $K = \sqrt{N}$

Maekawa's Algorithm [1985]

On initialization
state := RELEASED;
voted := FALSE;

For p_i to enter the critical section
state := WANTED;
Multicast REQUEST to all processes in V_i;
Wait until (number of replies received = K);
state := HELD;

On receipt of a REQUEST from p_i at p_j
if (state = HELD or voted = TRUE)
then
queue request from p_i without replying;
else
send REPLY to p_i;
voted := TRUE;
end if

For p_i to exit the critical section
state := RELEASED;
Multicast RELEASE to all processes in V_i;

On receipt of a RELEASE from p_i at p_j
if (queue of requests is non-empty)
then
remove head of queue – from p_k, say;
send REPLY to p_k;
voted := TRUE;
else
voted := FALSE;
end if

Correctness

- Theorem. *Maekawa's algorithm achieves mutual exclusion.*

- Proof: homework

Performance of Maekawa's Algorithm

- The size of each quorum is $\sqrt{N}$.

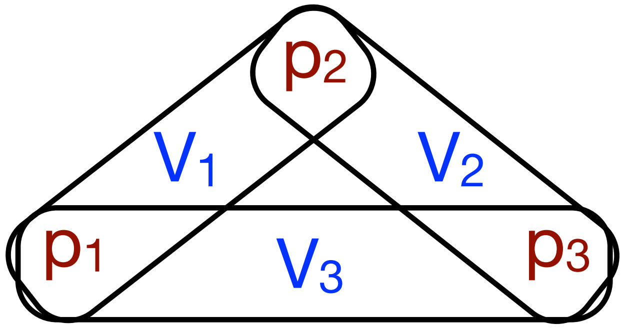
➡ The bandwidth utilization is $3\sqrt{N}$ messages per CS execution.

  - $2\sqrt{N}$ messages per entry to the CS ($\sqrt{N}$ REQUEST and $\sqrt{N}$ REPLY)
  - $\sqrt{N}$ messages per exit

- The client delay in requesting entry is a round-trip time.
- The synchronization delay is a round-trip time.

A Problematic Scenario

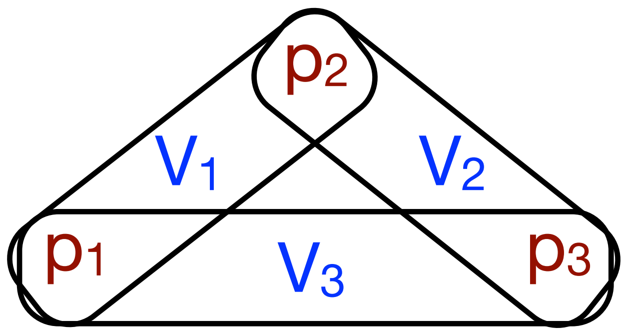
- Consider processes p_1, p_2 and p_3 with $V_1 = \{p_1, p_2\}$, $V_2 = \{p_2, p_3\}$, $V_3 = \{p_2, p_3\}$.



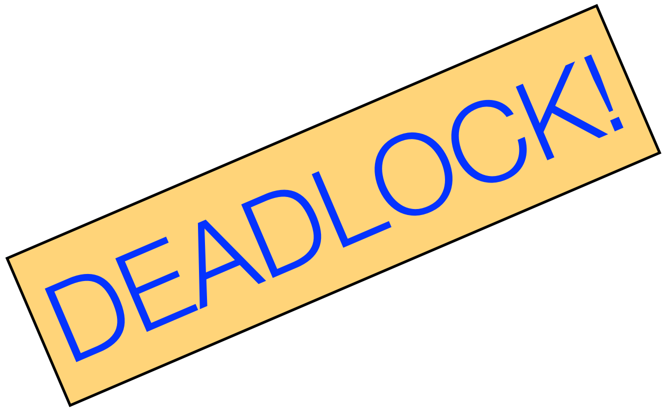
- If the processes *simultaneously* request entry to the CS, then the following scenario is possible:
  - p_1 is a candidate in V_1, waiting for p_2's REPLY
  - p_2 is a candidate in V_2, waiting for p_3's REPLY
  - p_3 is a candidate in V_3, waiting for p_1's REPLY

A Problematic Scenario

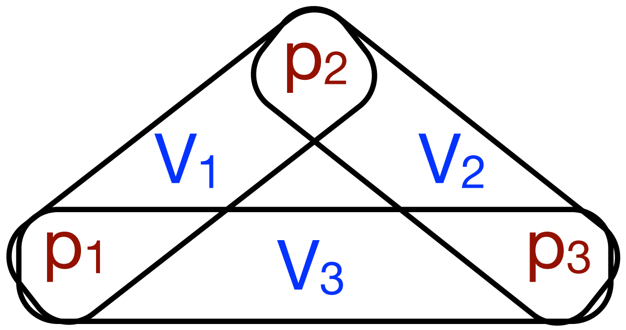
- Consider processes p_1, p_2 and p_3 with $V_1 = \{p_1, p_2\}$, $V_2 = \{p_2, p_3\}$, $V_3 = \{p_2, p_3\}$.

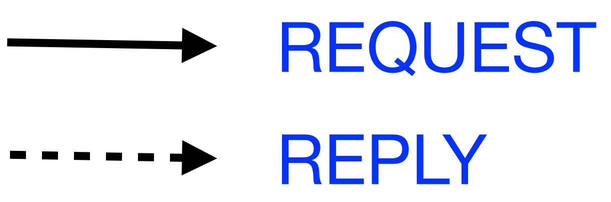


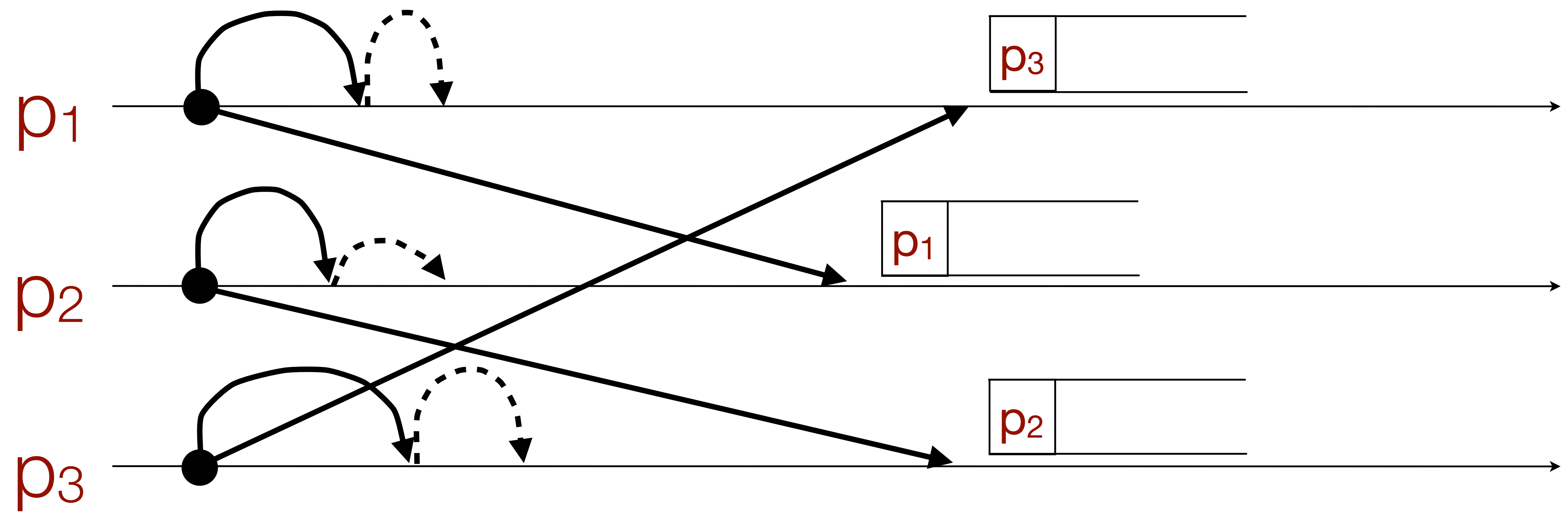
- If the processes *simultaneously* request entry to the CS, then the following scenario is possible:
  - p_1 is a candidate in V_1, waiting for p_2's REPLY
  - p_2 is a candidate in V_2, waiting for p_3's REPLY
  - p_3 is a candidate in V_3, waiting for p_1's REPLY



Deadlock Scenario







- Each process has received one out of two replies, and none can proceed!

Solving the Deadlock Problem

- Intuition: Maekawa's algorithm can deadlock because a process is exclusively locked by other processes and requests are not prioritized by their timestamps.
- The algorithm can be adapted so that it becomes deadlock-free.
- IDEA: in the adapted protocol, processes queue outstanding requests in happened-before order, so that requirements ME3 is also satisfied.
- See paper:

  B. Sanders.
  The Information Structure of Distributed Mutual Exclusion Algorithms.
  ACM Transactions on Computer Systems, Vol. 5, No. 3, pp. 284-99.

Fault Tolerance

- What happens when messages are lost?
- What happens when a process crashes?
- None of the algorithms would tolerate the loss of messages, *if the channels were unreliable*.
- Ring-based algorithm: cannot tolerate a crash failure of any single process.
- Central server algorithm: can tolerate the crash failure of a client process that neither holds nor has requested the token.
- Ricart-Agrawala algorithm: can be adapted to tolerate the crash failure of such a process, by taking it to grant all requests implicitly.
- Maekawa's algorithm: can tolerate some process crash failures: if a crashed process is not in a voting set that is required, then its failure will not affect the other processes.