Distributed Systems and Algorithms

Martin Quinson <martin.quinson@loria.fr>
Abdelkader Lahmadi <abdelkader.lahmadi@loria.fr>
Olivier Festor <olivier.festor@inria.fr>

LORIA – INRIA Nancy Grand Est

2018-2019
(compiled on: September 26, 2019)

Chapter 1

Some Distributed Algorithms

Premier chapitre

Some Distributed Algorithms

Some Distributed Algorithms

Goals of this section

Present some basic algorithms

- Mutual exclusion
- Election
- Consensus
- Group protocols

Present general approaches

- Ordering events (with abstract clocks)
- Applicative topologies (ring, tree, graph without circuit)

Some Distributed Algorithms

Goals of this section

Present some basic algorithms

- Mutual exclusion
- Election
- Consensus
- Group protocols
- Sequential equivalents
  - Sorting, Shortest path
  - Classical data structures (stack, list, hashing, trees)

Present general approaches

- Ordering events (with abstract clocks)
- Applicative topologies (ring, tree, graph without circuit)
- Sequential equivalents
  - Recursion, Divide&Conquer, Greedy algorithms

Mutual Exclusion

Problem Statement

- Force an order on the execution of critical sections
- Fairness (no infinite starvation of any process); Liveness (no deadlock)

Approaches

- Centralized coordinator: ask lock to coordinator, get lock, release lock
- Use a global order: using abstract clocks
  Ask everyones, and concurrent requests are handled "in order"
- Using quorums: Ask only members of specific groups
- Force a topology: virtual ring, virtual tree
  Gives an order on nodes, not only on requests

Algorithms

- A whole load of such algorithms in literature
- #messages $\in [O(log(n)); O(n)]$ (ask everyone, or distributed waiting queue)

What's coming now: Details of some algorithms

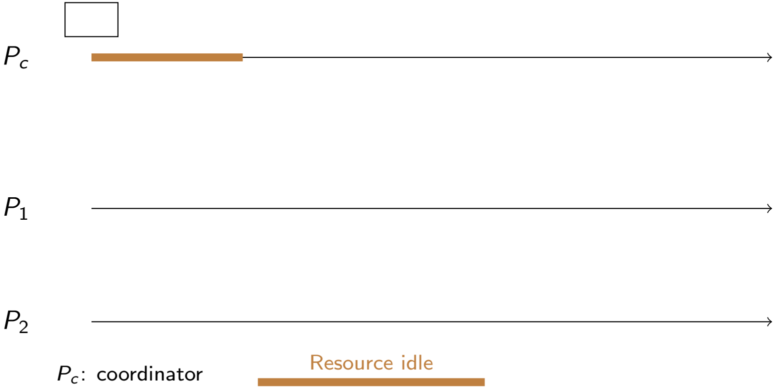
- For culture and to get a grip on distributed algorithms development approach

Centralized: Coordinator Based Algorithm

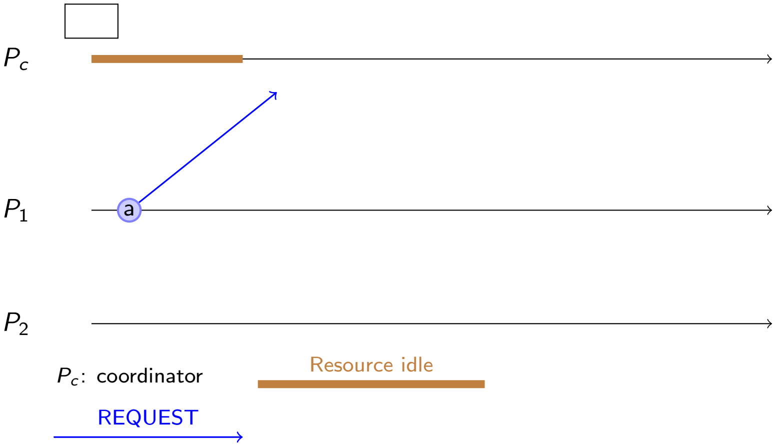
Main Idea

- One of the processes acts as coordinator (cf. Leader Election Algorithm)
  Coordinator decides the order in which critical section requests are fulfilled
- Processes send requests to coordinator and wait permission
  Requests are fulfilled in FIFO order at the coordinator
- Coordinator grants permission to requests one at a time
  All other requests are queued in a FIFO queue.

Coordinator Based Algorithm for Mutual Exclusion



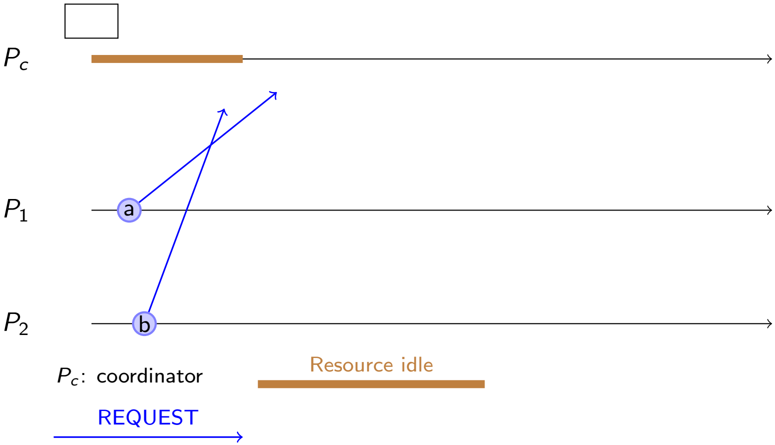
Coordinator Based Algorithm for Mutual Exclusion



Event explanation

a. P_1 requests the CS to coordinator

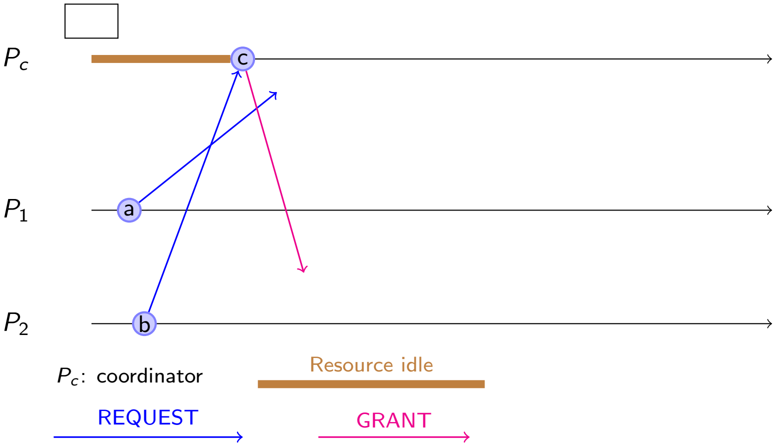
Coordinator Based Algorithm for Mutual Exclusion



Event explanation

b. P_2 requests the CS to coordinator

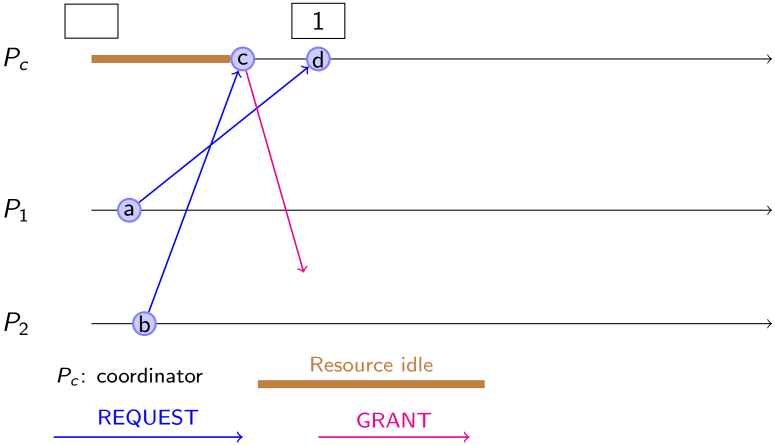
Coordinator Based Algorithm for Mutual Exclusion



Event explanation

c. coordinator receives the request from P_2
   - Idle token, so send reply back

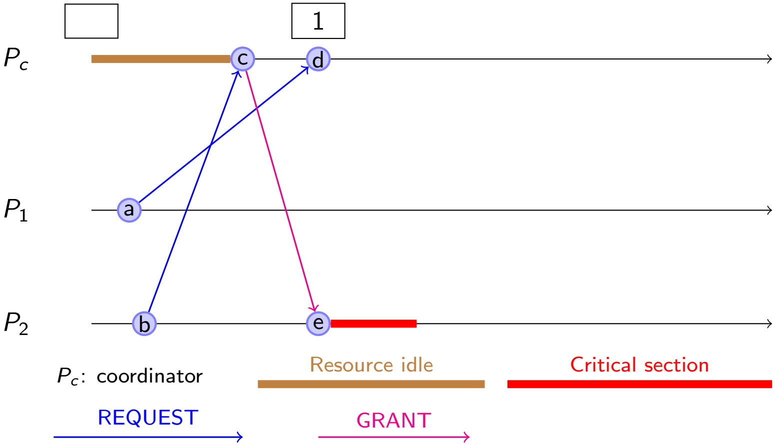
Coordinator Based Algorithm for Mutual Exclusion



Event explanation

d. coordinator receives the request from P_1
   - Token not there, so enqueue the request

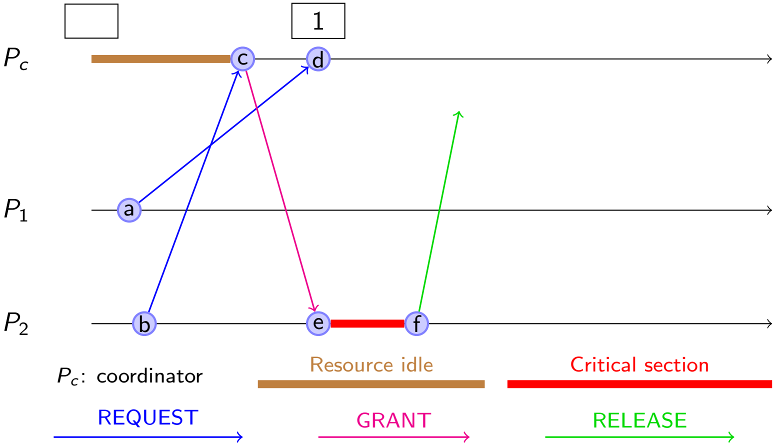
Coordinator Based Algorithm for Mutual Exclusion



Event explanation

e. P_2 receives the grant
  - Enters the CS

Coordinator Based Algorithm for Mutual Exclusion

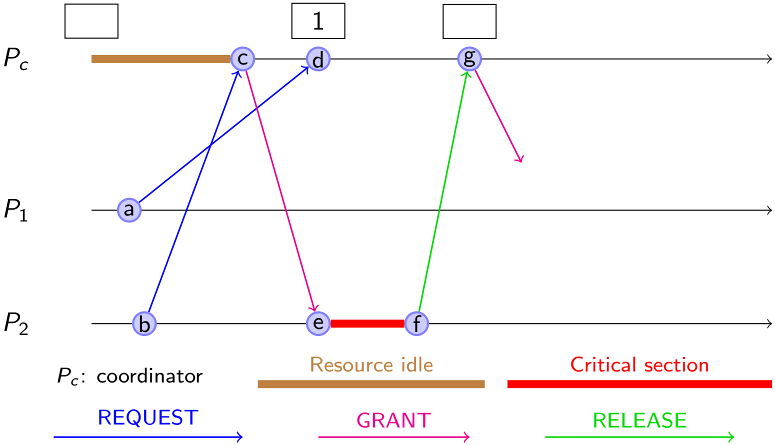


Event explanation

f. P_2 exits the CS

- Send release to coordinator

Coordinator Based Algorithm for Mutual Exclusion

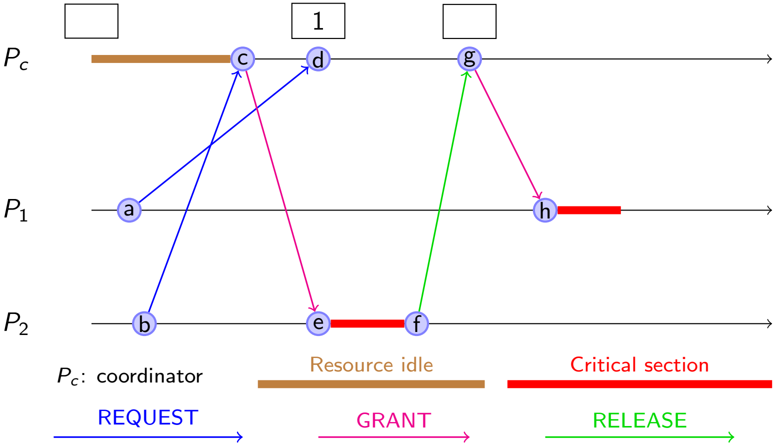


Event explanation

g. coordinator receives the release

- Someone (P_1) is waiting in the queue
- Unqueue P_1
- Send grant to P_1

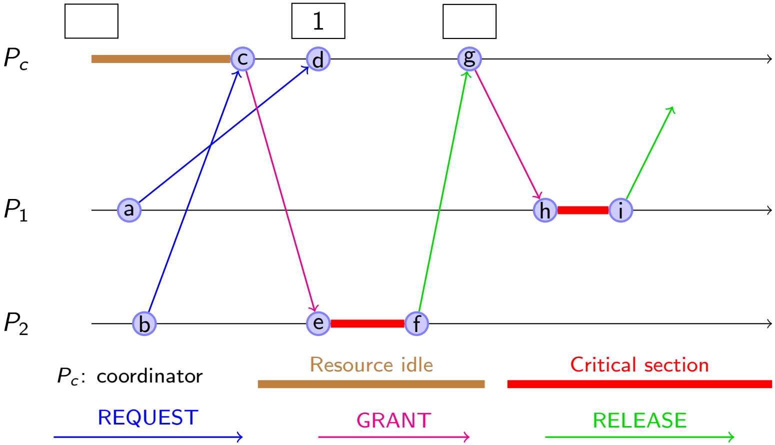
Coordinator Based Algorithm for Mutual Exclusion



Event explanation

h. P_1 receives the grant
- Enters the CS

Coordinator Based Algorithm for Mutual Exclusion

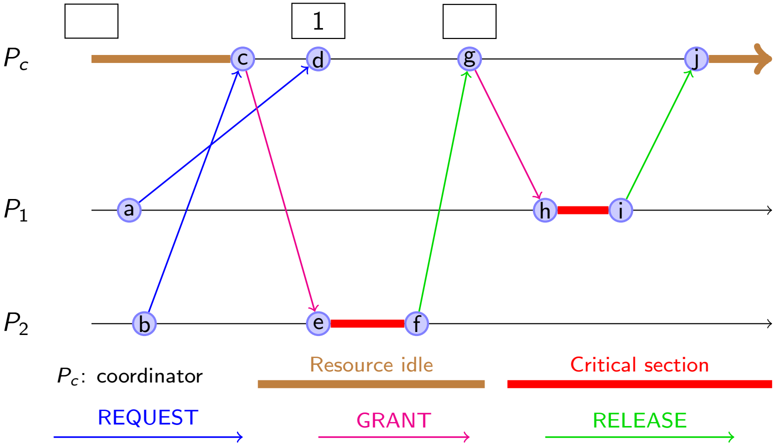


Event explanation

f. P_1 exits the CS

- Send release to coordinator

Coordinator Based Algorithm for Mutual Exclusion



Event explanation

g. coordinator receives the release

- Nobody in queue, nothing to do
- Let the token idling

Centralized Mutual Exclusion: Complexity Analysis

Parameters

N Number of processes in the system

T Message transmission time

E Critical section execution time

Message complexity: 3

- 1 REQUEST message + 1 GRANT message + 1 RELEASE message
- Message-size complexity: O(1)

Time complexity

- Response time (under light load): 2T + E
- Synchronization delay (under heavy load): 2T

Lamport's Algorithm for Mutual Exclusion

Assumptions

- Channels are FIFO
- Processes run a Lamport's Logical Clock

Main Idea

- Requests are timestamped using logical clocks, and fulfilled in timestamp order
- Processes maintain a priority queue of all requests they know about
- Lots of broadcasts to get the timestamps propagate to peers

Lamport's Mutual Exclusion: Steps for process P_i

On generating a critical section request

- Insert the request into the priority queue
- Broadcast the request to all processes

On receiving a critical section request from another process:

- Insert the request into the priority queue.
- Send a REPLY message to the requesting process.

Conditions to enter critical section:

- L1: P_i has received a REPLY message from all processes.
  Any request received in future will have larger timestamp than own request
- L2: P_i's own request is at the top of its queue.
  I have the smallest timestamp among all already received requests

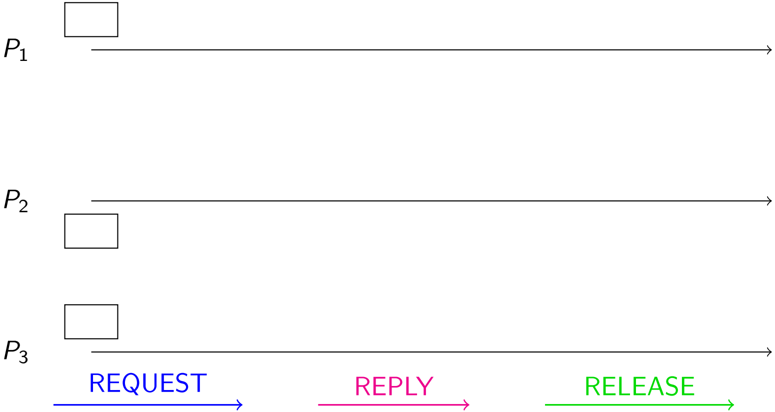
On leaving the critical section

- Remove the request from the queue
- Broadcast a RELEASE message to all processes

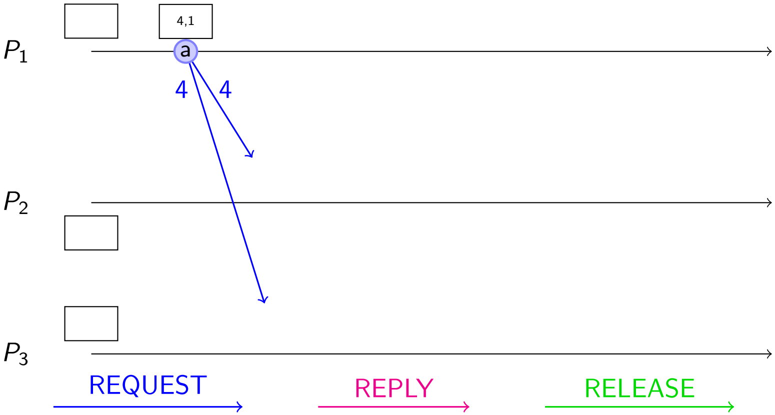
On receiving a RELEASE message from another process

- Remove the request of that process from the queue

Lamport's Mutual Exclusion: Illustration



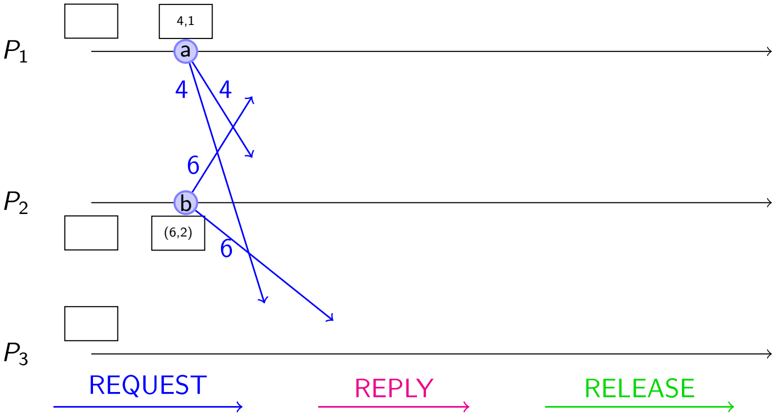
Lamport's Mutual Exclusion: Illustration



a. P_1 requests the CS (timestamp=4)

- Broadcast the request
- Enqueue the request locally

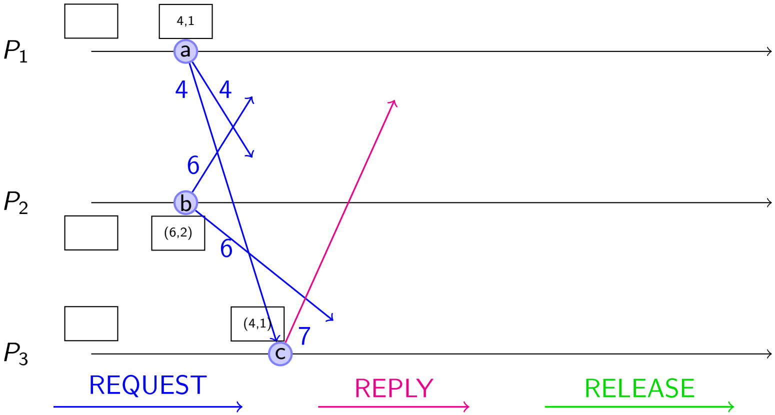
Lamport's Mutual Exclusion: Illustration



b. P_2 requests the CS (timestamp=6)

- Broadcast the request
- Enqueue the request locally

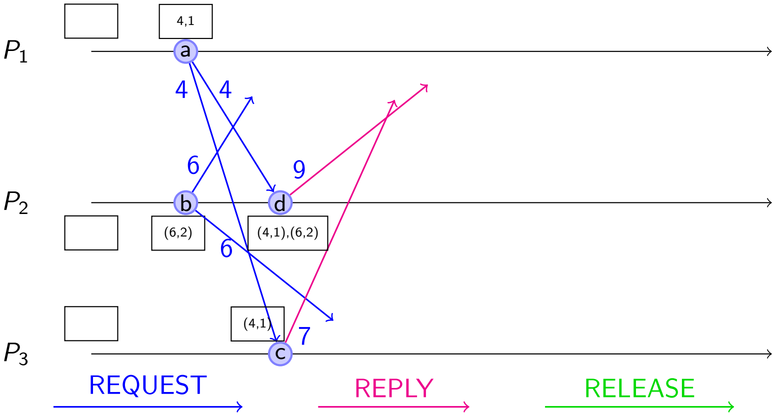
Lamport's Mutual Exclusion: Illustration



c. P_3 receives the request from P_1

- Answer REPLY with timestamp 7
- Enqueue the request locally

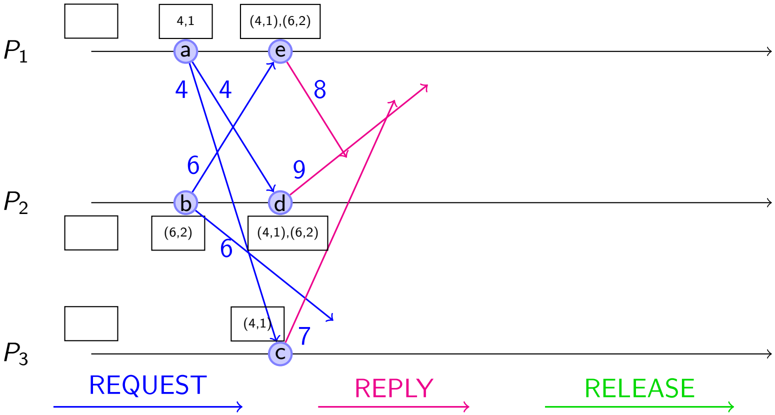
Lamport's Mutual Exclusion: Illustration



d. P_2 receives the request from P_1

- Answer REPLY with timestamp (max(6,4)+1)+1=8
- Enqueue the request locally (sorting on Lamport's clock)

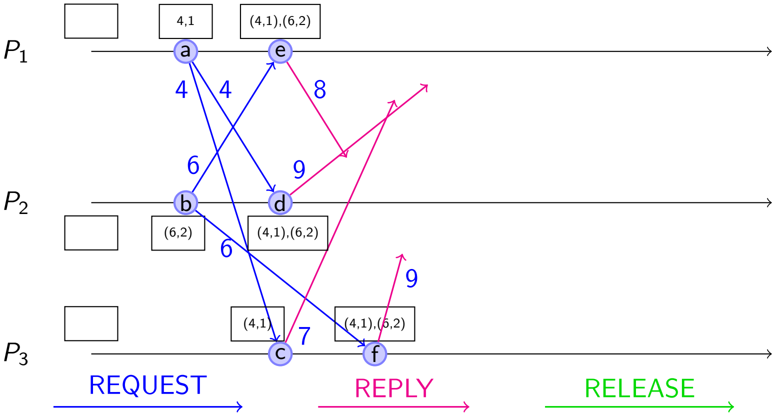
Lamport's Mutual Exclusion: Illustration



e. P_1 receives the request from P_2

- Answer REPLY with timestamp (max(4,6)+1)+1=8
- Enqueue the request locally (sorting on Lamport's clock)

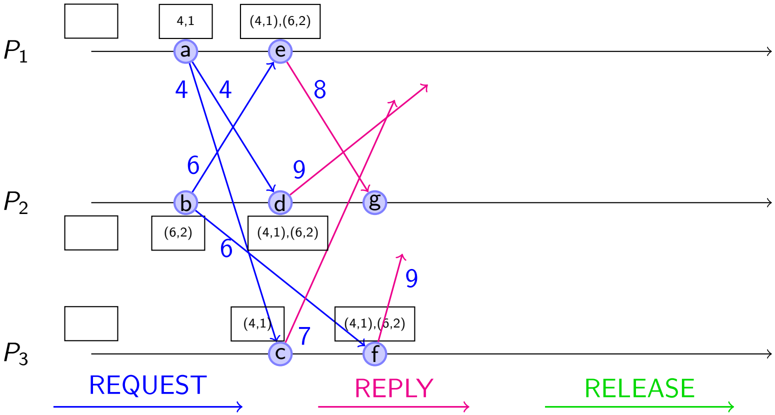
Lamport's Mutual Exclusion: Illustration



f. P_3 receives the request from P_2

- Answer REPLY with timestamp (max(7,6)+1)+1=9
- Enqueue the request locally

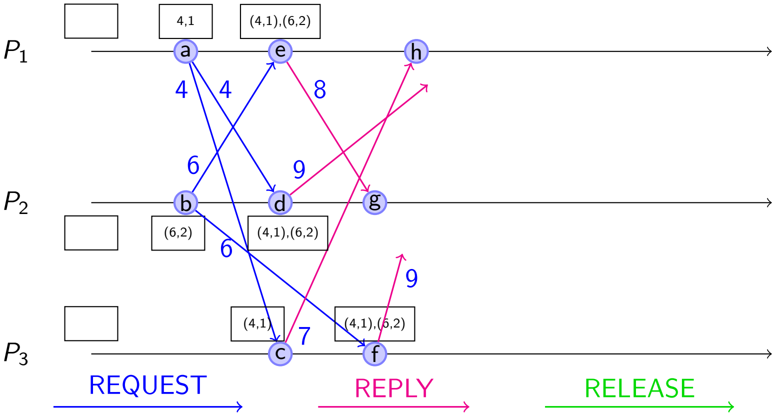
Lamport's Mutual Exclusion: Illustration



g. P_2 receives the reply from P_1

- (nothing to do, one request still missing)

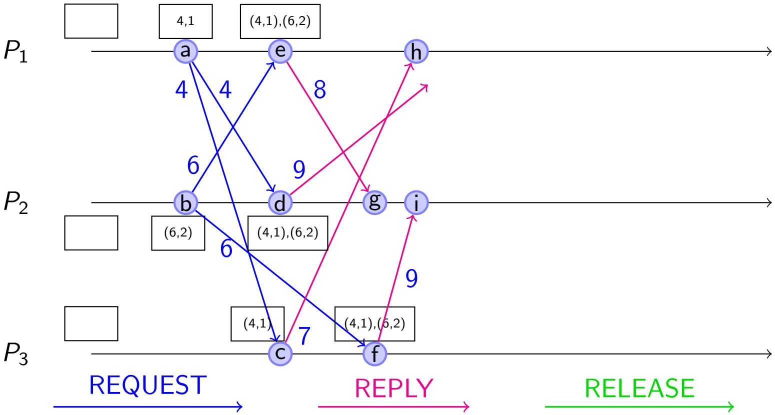
Lamport's Mutual Exclusion: Illustration



h. P_1 receives the reply from P_3

- (nothing to do, one request still missing)

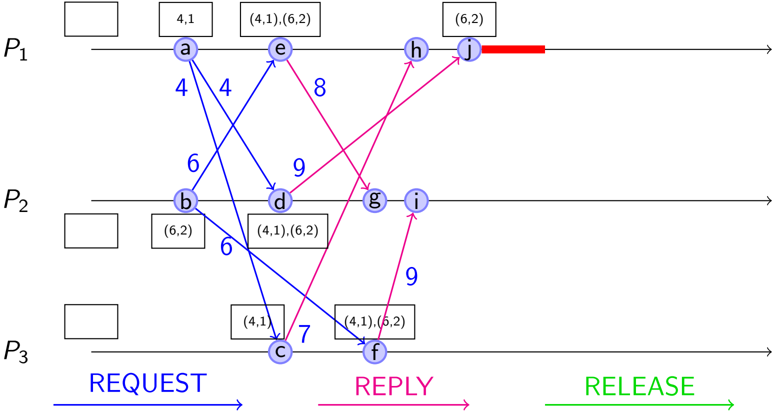
Lamport's Mutual Exclusion: Illustration



i. P_2 receives the reply from P_3

- Every request received, but not first in queue
- Thus nothing to do

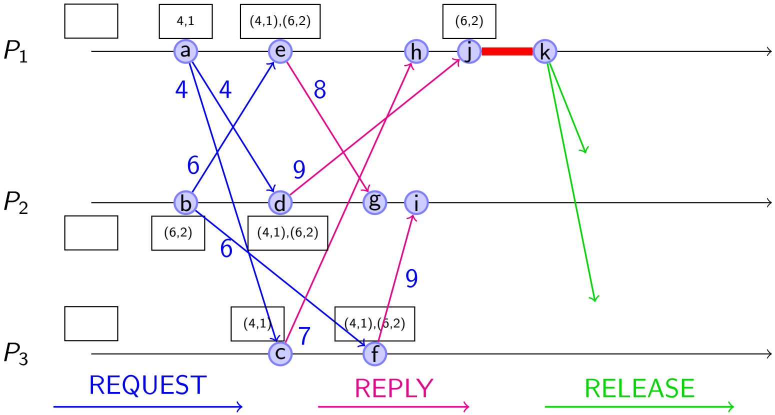
Lamport's Mutual Exclusion: Illustration



j. P_1 receives the reply from P_2

- Every request received, and first in queue
- Thus dequeuing self request and entering CS

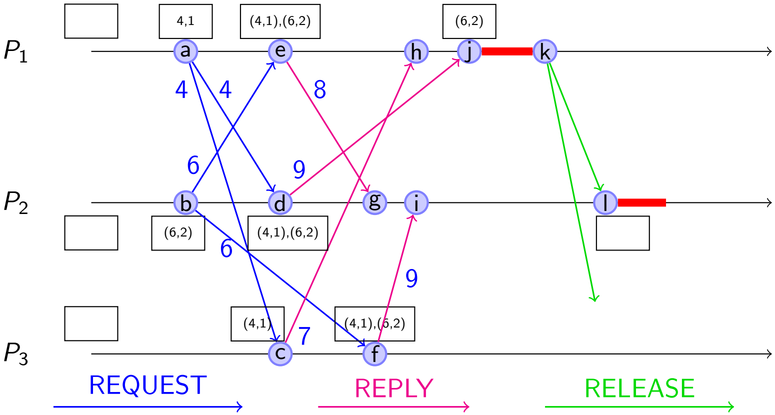
Lamport's Mutual Exclusion: Illustration



k. P_1 exits CS

- Broadcast RELEASE

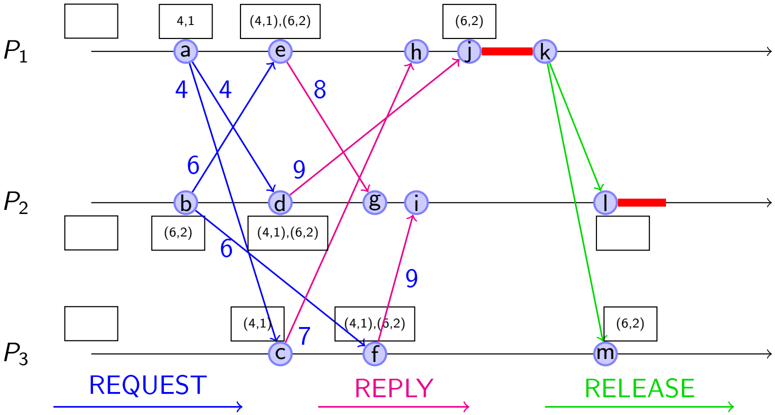
Lamport's Mutual Exclusion: Illustration



l. P_2 receives RELEASE from P_1

- Remove (4,1) from queue
- Every replies received and first of queue
- Thus entering CS (after removing myself from queue)

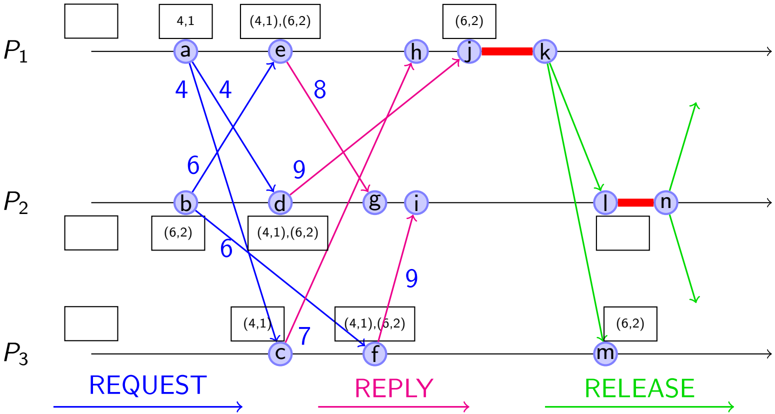
Lamport's Mutual Exclusion: Illustration



m. P_3 receives RELEASE from P_1

- Update the queue

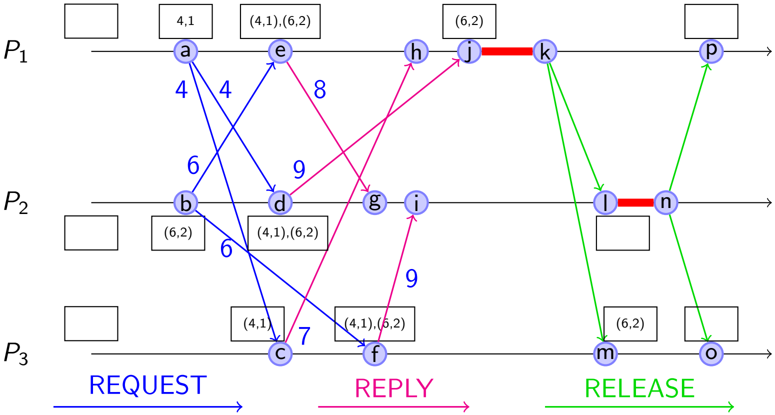
Lamport's Mutual Exclusion: Illustration



n. P_2 exits its CS

- Broadcast RELEASE

Lamport's Mutual Exclusion: Illustration



o&p. P_1 and P_2 receive RELEASE from P_2

- Update queues

Lamport's Mutual Exclusion: Optimization

Recap Conditions to enter critical section:

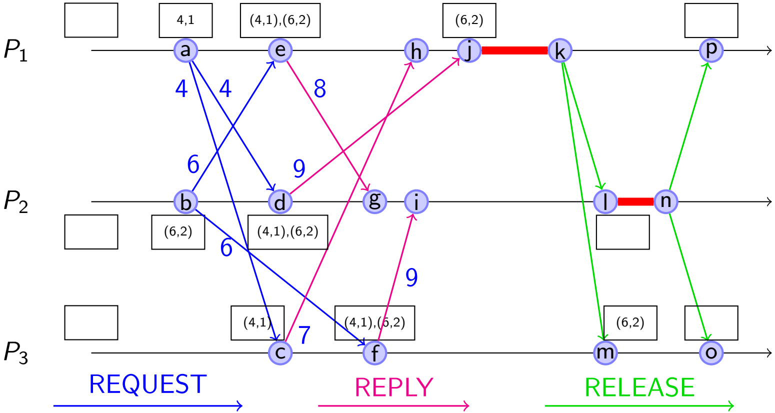
- L1: P_i has received a REPLY message from all processes.
  Any request received in future will have larger timestamp than own request
- L2: P_i's own request is at the top of its queue.
  I have the smallest timestamp among all already received requests

L1 is too restrictive wrt the wanted property

- Wait for any messages with higher timestamp from all processes is enough
  Any request received in future will *still* have larger timestamp than own request

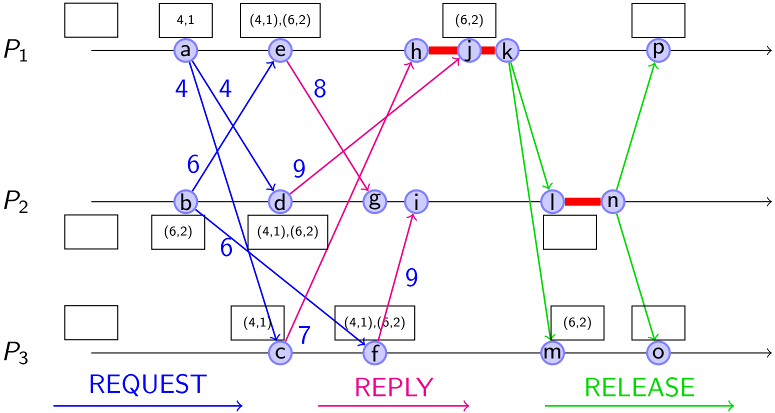
Lamport's Mutex Optimization: Illustration

Without the optimization



Lamport's Mutex Optimization: Illustration

With the optimization



Lamport's Mutex Algorithm: Complexity Analysis

Parameters

N Number of processes in the system

T Message transmission time

E Critical section execution time

Message complexity: 3(N - 1)

- $N-1$ REQUEST messages + $N-1$ REPLY messages + $N-1$ RELEASE messages
- Message-size complexity: O(1)

Time complexity

- Response time (under light load): 2T + E
- Synchronization delay (under heavy load): T

Ricart and Agrawala's Algorithm

Inefficiencies in Lamport's Algorithm

- Scenario 1
  - Situation: P_i and P_j concurrently request CS and $C(P_i) < C(P_j)$
  - Lamport: P_i first send REPLY and later RELEASE.
    P_j only acts on RELEASE
  - Improvement: P_i's REPLY can be ommited
- Scenario 2
  - Situation: P_i requests CS and P_j don't for some time
  - Lamport: P_i send RELEASE to P_j on exiting CS
  - Improvement: That message can be ommited
    (if P_j requests CS, it will contact P_i anyway)

Main ideas of Ricart and Agrawala's Algorithm

- Combine REPLY and RELEASE messages
- On leaving CS, only REPLY/RELEASE to processes with unfulfilled CS requests
- Eliminate priority queue

Ricart and Agrawala Mutex: Steps for process P_i

On generating a critical section request

- Broadcast the request to all processes

On receiving a critical section request from another process:

- Send a REPLY if any of these condition is true
  - P_i has no unfulfilled request of its own
  - P_i unfulfilled request has larger timestamp than that of the received request
- Else, defer sending the REPLY message

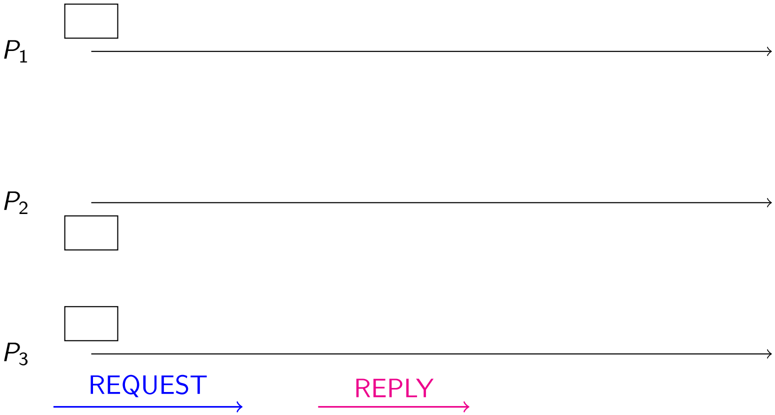
Conditions to enter critical section:

- P_i has received a REPLY message from all processes

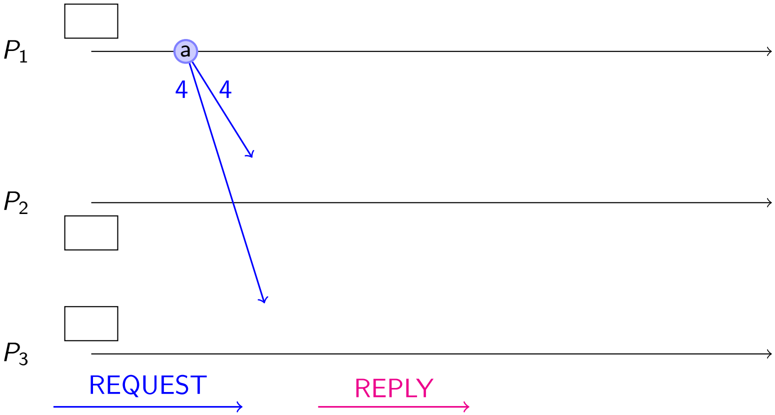
On leaving the critical section

- Send all defered REPLY messages

Ricart and Agrawala Mutex: Illustration



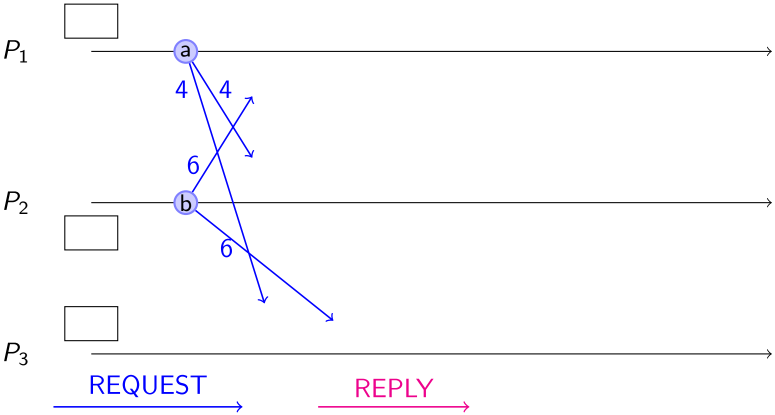
Ricart and Agrawala Mutex: Illustration



a. P_1 requests the CS (timestamp=4)

- Broadcast the request

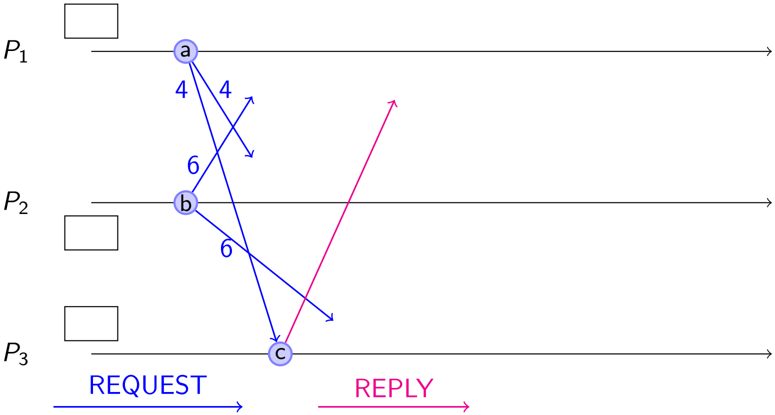
Ricart and Agrawala Mutex: Illustration



b. P_2 requests the CS (timestamp=6)

- Broadcast the request

Ricart and Agrawala Mutex: Illustration

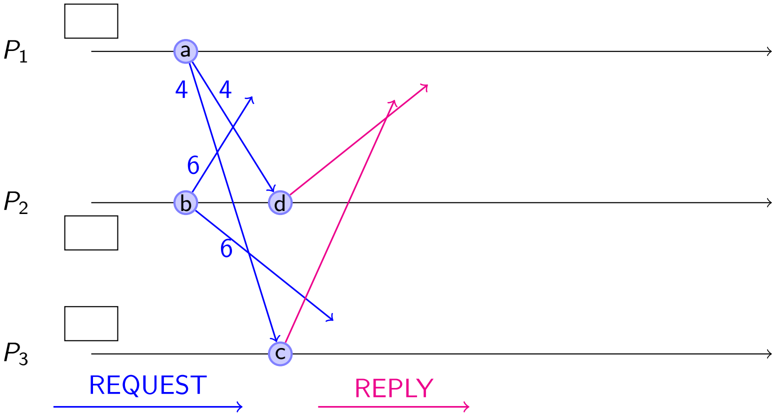


c. P_3 receives the request from P_1

- No unfulfilled request itself

$\rightsquigarrow$ Returns a REPLY

Ricart and Agrawala Mutex: Illustration

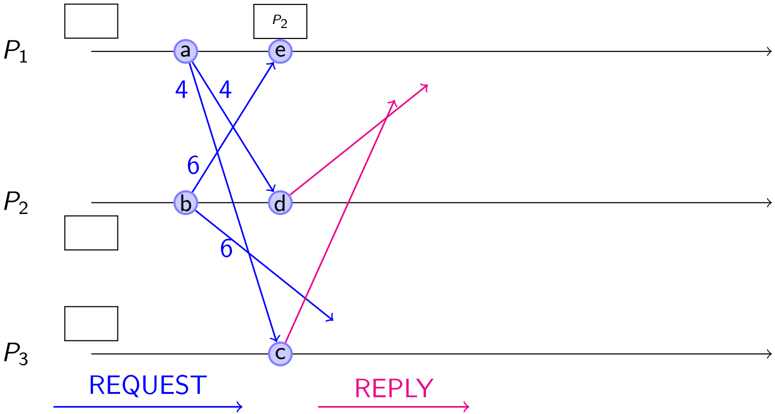


d. P_2 receives the request from P_1

- Own unfulfilled request has larger timestamp

$\rightsquigarrow$ Returns a REPLY

Ricart and Agrawala Mutex: Illustration

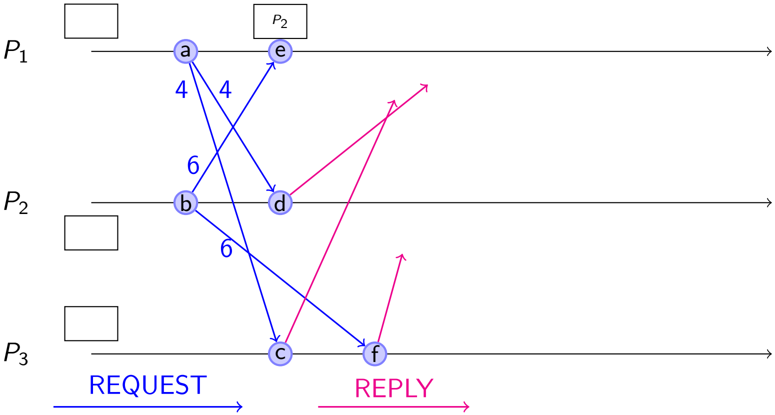


e. P_1 receives the request from P_2

- Own unfulfilled request has smaller timestamp

$\rightsquigarrow$ Defer the sending of REPLY

Ricart and Agrawala Mutex: Illustration

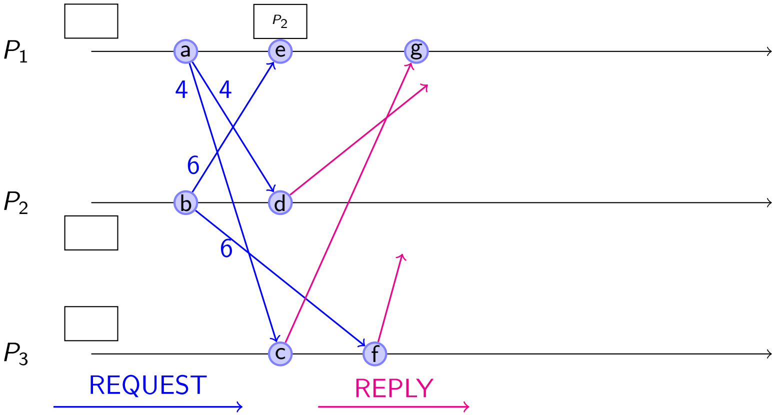


f. P_3 receives the request from P_2

- No unfulfilled request itself

$\leadsto$ Returns a REPLY

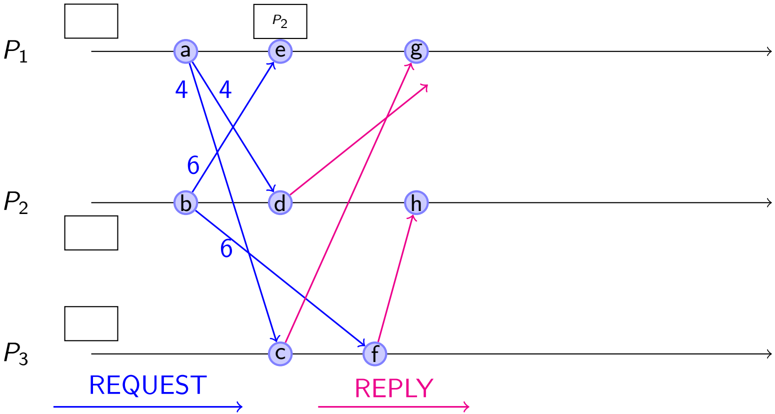
Ricart and Agrawala Mutex: Illustration



g. P_1 receives the reply from P_3

- Nothing to do, one request still missing

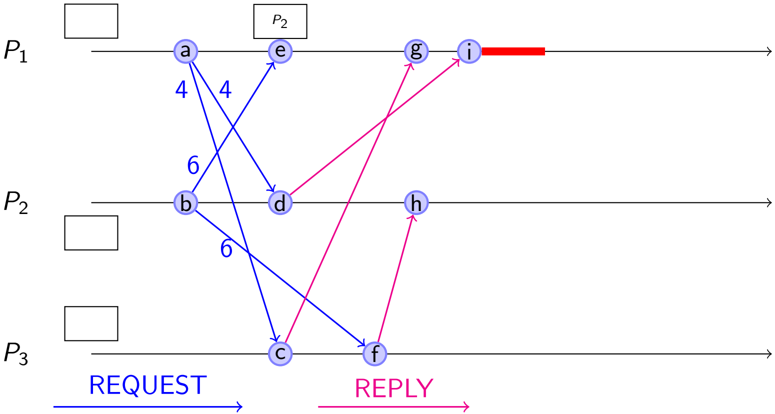
Ricart and Agrawala Mutex: Illustration



h. P_2 receives the reply from P_3

- Nothing to do, one request still missing (since it's delayed)

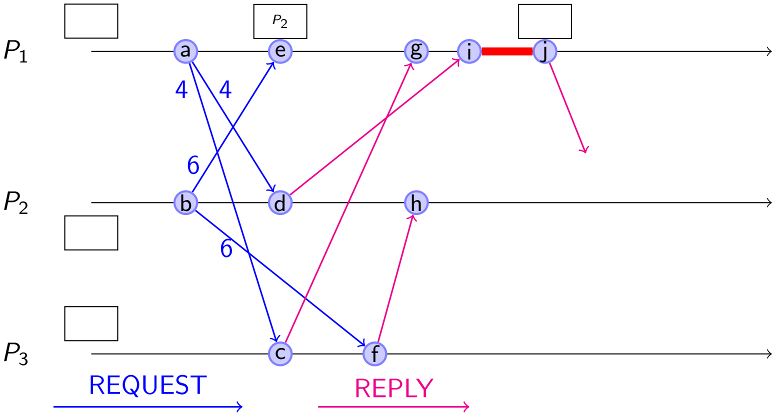
Ricart and Agrawala Mutex: Illustration



i. P_1 receives the reply from P_2

- Every request received
- Thus entering CS

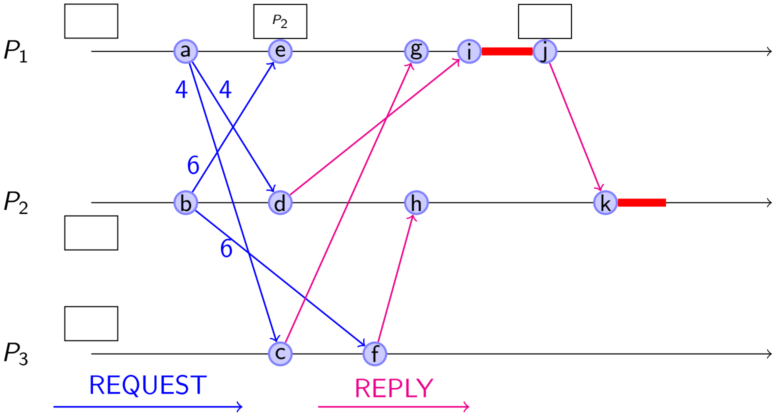
Ricart and Agrawala Mutex: Illustration



j. P_1 exits CS

- Send delayed REPLY to P_2

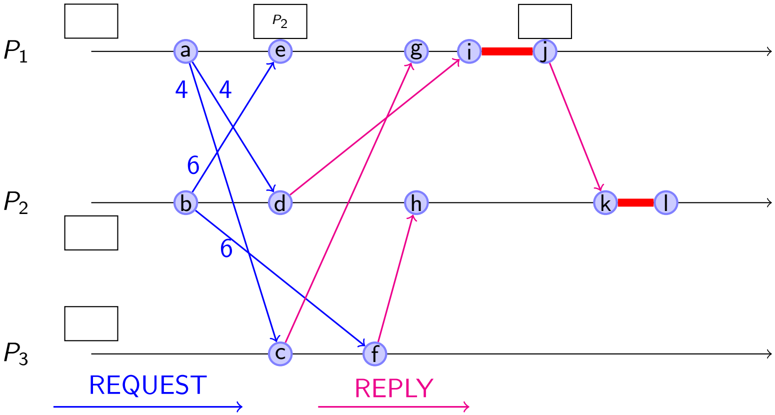
Ricart and Agrawala Mutex: Illustration



k. P_2 receives RELEASE from P_1

- Every replies received
- Thus entering CS

Ricart and Agrawala Mutex: Illustration



l. P_3 receives RELEASE from P_1

- No delayed REPLY, nothing to do

Ricart and Agrawala: Complexity Analysis

Parameters

N Number of processes in the system

T Message transmission time

E Critical section execution time

Message complexity: 2(N - 1)

- $N - 1$ REQUEST messages + $N - 1$ REPLY messages
- Message-size complexity: O(1)

Time complexity

- Response time (under light load): 2T + E
- Synchronization delay (under heavy load): T

Roucairol and Carvalho's Algorithm

Inefficiency in Ricart and Agrawala's Algorithm

- Every process handles every critical section request.

Goal of this new algorithm for conflict resolution

- Change algorithm so that only active processes (requesting CS) interact
- Process not requesting the CS will eventually stop receiving messages

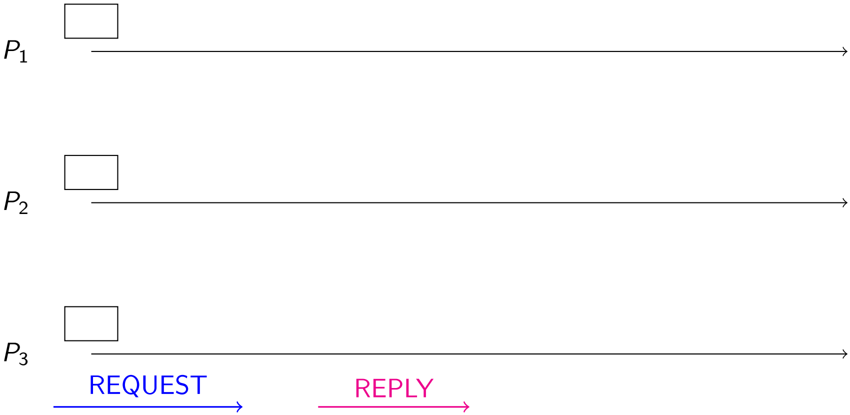
Main idea

- REPLY from P_j to P_i means: P_j grants permission to P_i to enter CS
- P_i keeps that permission until it send REPLY to someone else

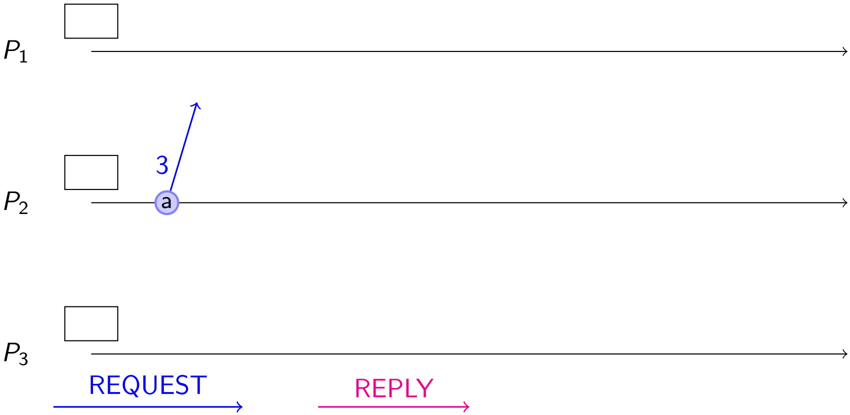
Modification to Ricart and Agrawala's Algorithm

- To enter CS, P_i asks for permission from P_j if either:
  - (P_i sent REPLY to P_j) AND (P_i didn't got REPLY from P_j since then)
  - (It's P_i's first request) AND ($i > j$)

Roucairol and Carvalho's Mutex: Illustration events



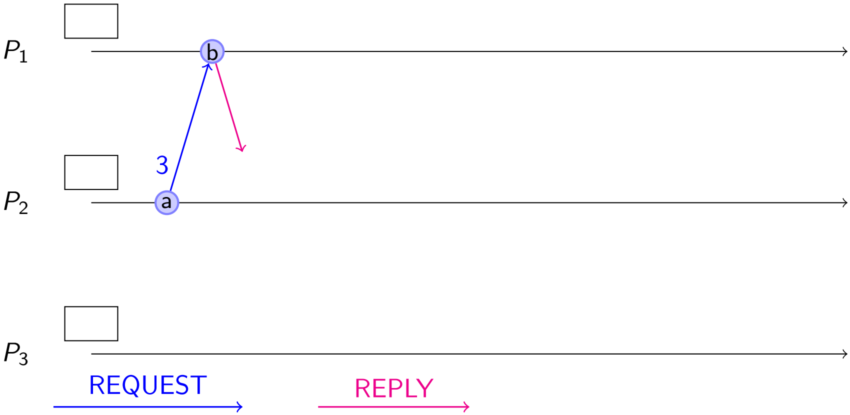
Roucairol and Carvalho's Mutex: Illustration events



a. P_2 requests the CS (timestamp=3)

$\rightsquigarrow$ Send the request to P_1 only ($1 < 2$)

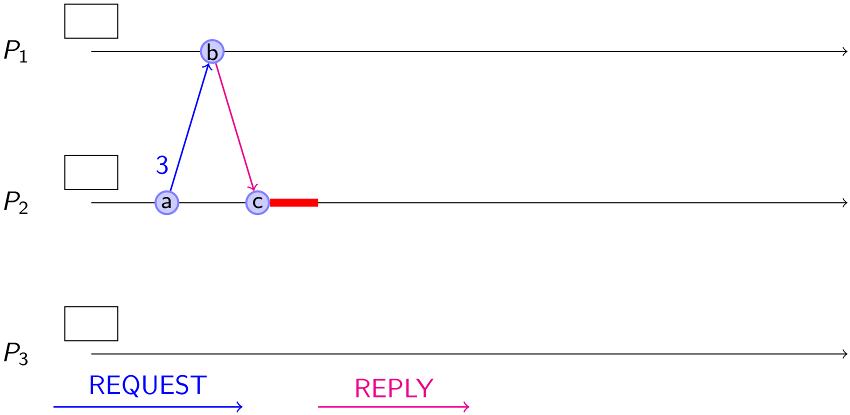
Roucairol and Carvalho's Mutex: Illustration events



b. P_1 receives P_2's REQUEST

$\rightsquigarrow$ returns REPLY

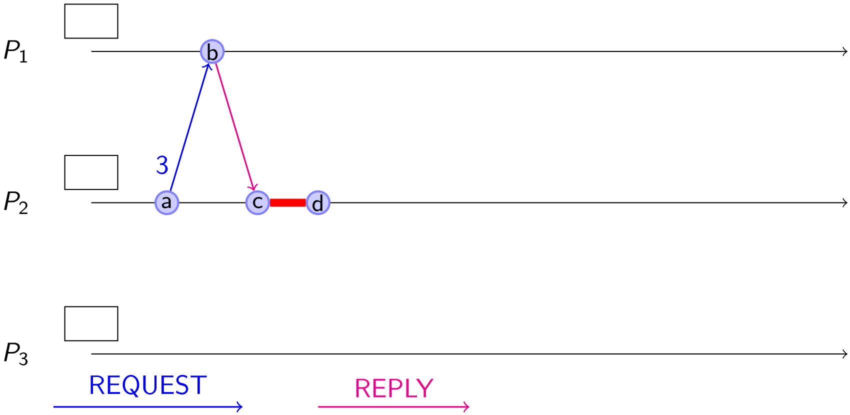
Roucairol and Carvalho's Mutex: Illustration events



c. P_2 receives REPLY from P_1.

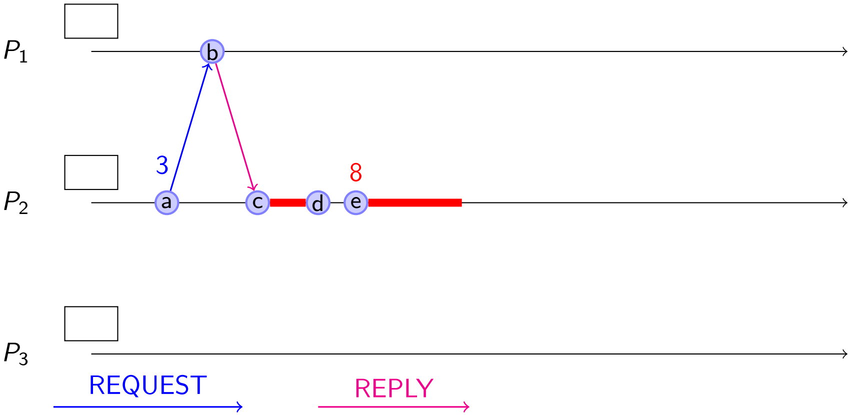
$\rightsquigarrow$ enters CS

Roucairol and Carvalho's Mutex: Illustration events



d. P_2 exists CS

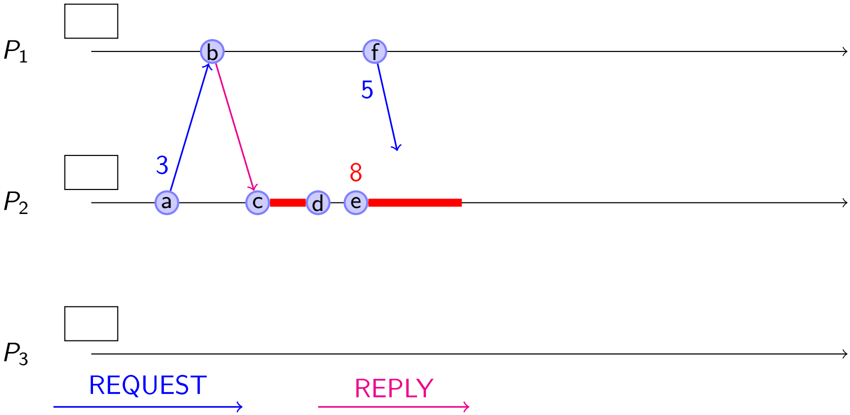
Roucairol and Carvalho's Mutex: Illustration events



e. P_2 requests CS again (stamp=8)

$\rightsquigarrow$ re-enter CS without any new message

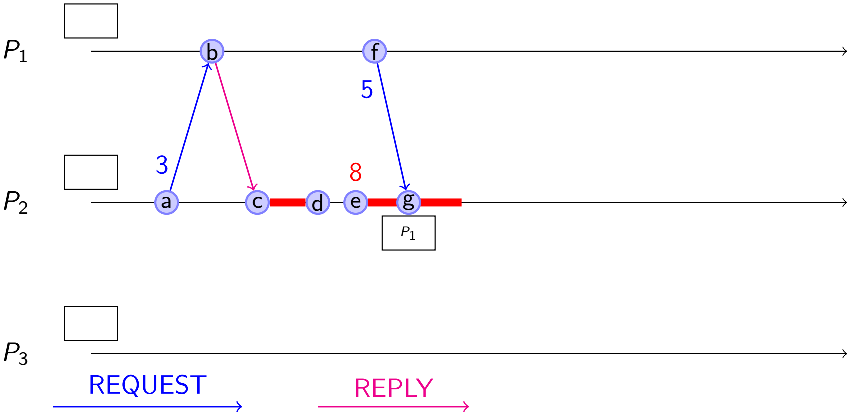
Roucairol and Carvalho's Mutex: Illustration events



f. P_1 requests CS (stamp=5)

$\rightsquigarrow$ send REQUEST to P_2 only (active known peer)

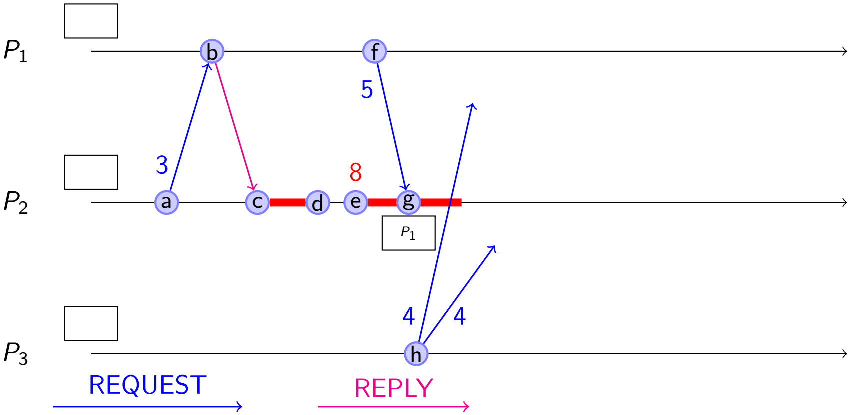
Roucairol and Carvalho's Mutex: Illustration events



g. P_2 receives REQUEST from P_1

$\rightsquigarrow$ defers REPLY because in CS

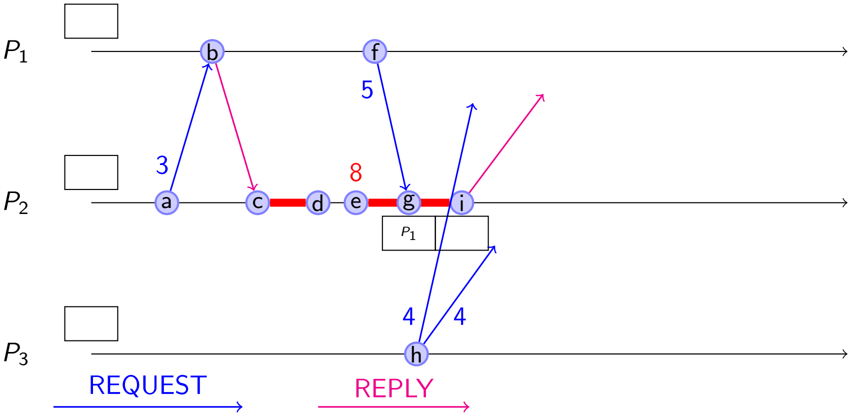
Roucairol and Carvalho's Mutex: Illustration events



h. P_3 requests the CS

$\rightsquigarrow$ broadcasts REQUEST to every processes

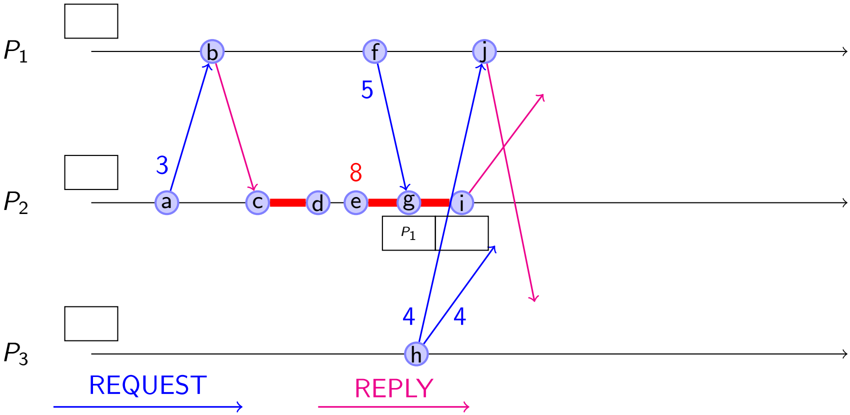
Roucairol and Carvalho's Mutex: Illustration events



i. P_2 exists CS

$\rightsquigarrow$ send defered REPLY to P_1

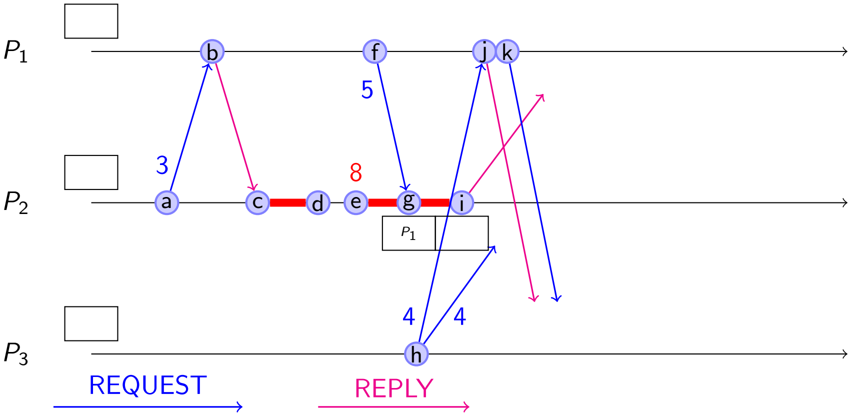
Roucairol and Carvalho's Mutex: Illustration events



j. P_1 receives REQUEST from P_3

returns REPLY since stamp lower than own

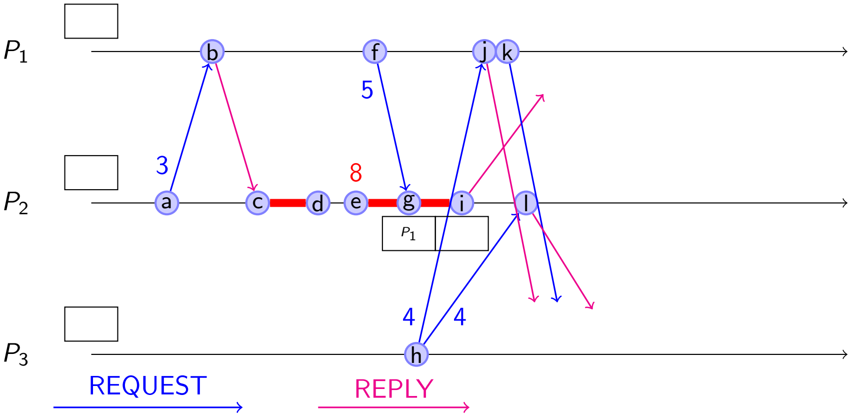
Roucairol and Carvalho's Mutex: Illustration events



k. P_1 thought P_3 not active, until j.

$\rightsquigarrow$ send previous REQUEST now

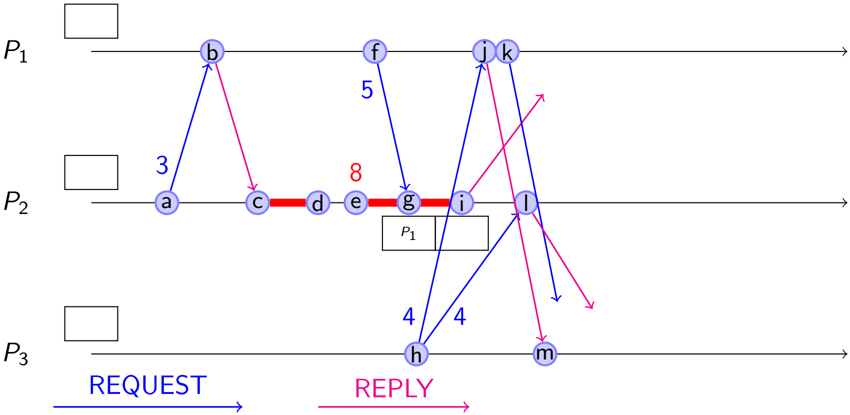
Roucairol and Carvalho's Mutex: Illustration events



l. P_2 receives request from P_3

$\rightsquigarrow$ returns REPLY

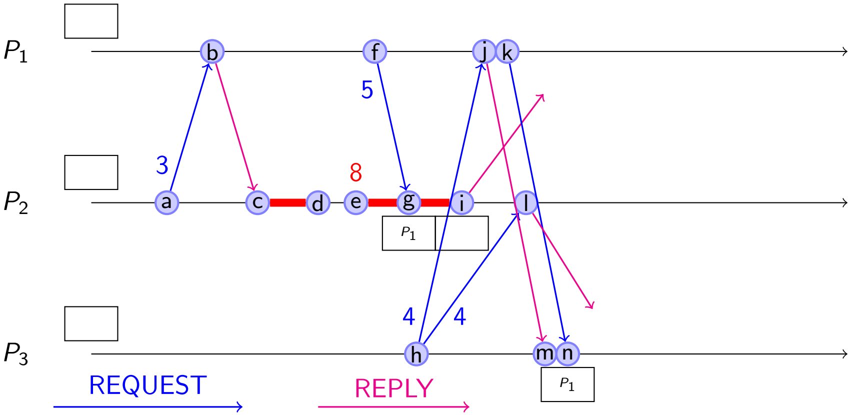
Roucairol and Carvalho's Mutex: Illustration events



m. P_3 receives REPLY from P_1

(one missing)

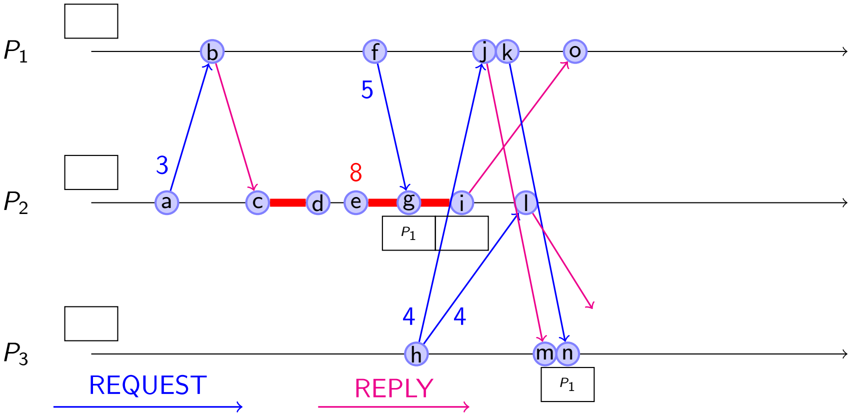
Roucairol and Carvalho's Mutex: Illustration events



n. P_3 receives REQUEST from P_1

$\rightsquigarrow$ queues it because own timestamp lower

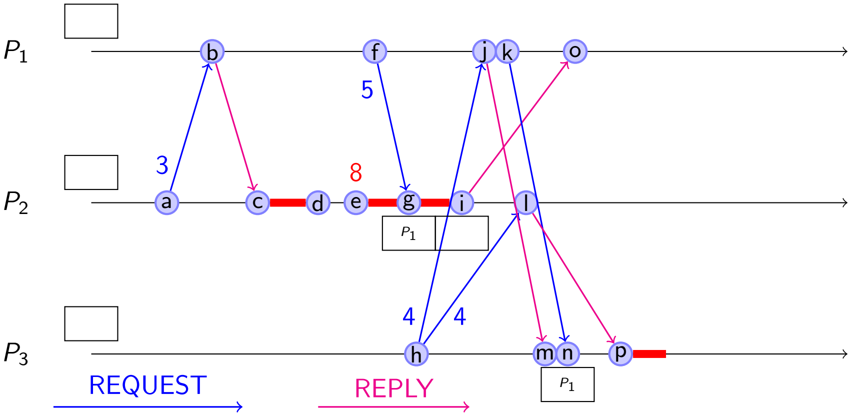
Roucairol and Carvalho's Mutex: Illustration events



o. P_1 receives REPLY from P_2

(one missing)

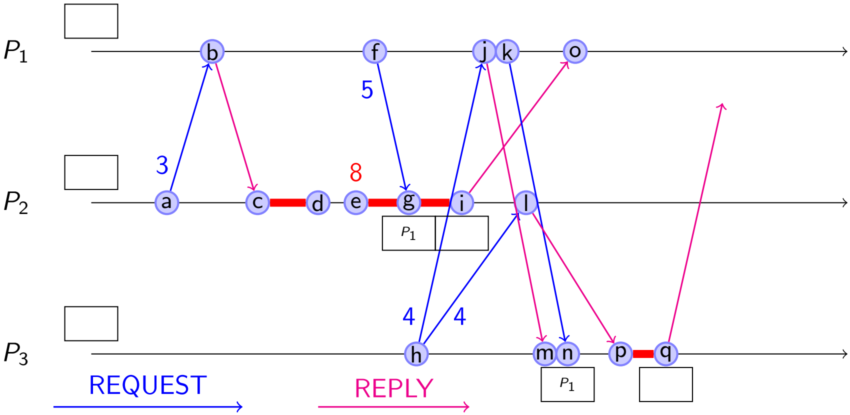
Roucairol and Carvalho's Mutex: Illustration events



p. P_3 receives REPLY from P_2

everyone answered $\rightsquigarrow$ enters CS

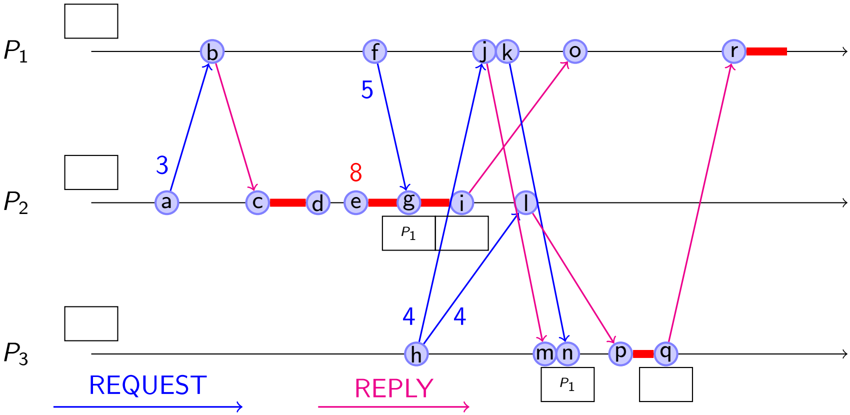
Roucairol and Carvalho's Mutex: Illustration events



q. P_3 exits CS

$\rightsquigarrow$ send delayed REPLY to P_1

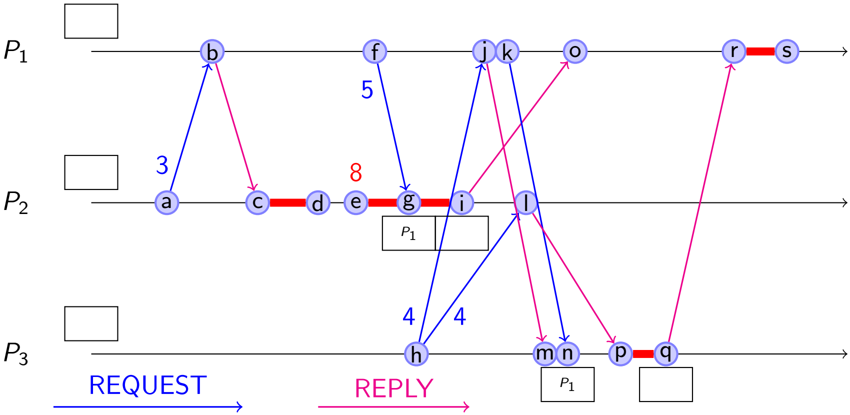
Roucairol and Carvalho's Mutex: Illustration events



r. P_1 receives REPLY from P_3

everyone answered $\rightsquigarrow$ enters CS

Roucairol and Carvalho's Mutex: Illustration events



s. P_1 exits CS

(nothing to do)

Roucairol and Carvalho's Mutex: Complexity Analysis

Parameters

N Number of processes in the system

T Message transmission time

E Critical section execution time

Message complexity:

- Best case: 0
- Worst case: 2(N-1): $N-1$ REQUEST messages + $N-1$ REPLY messages
- Message-size complexity: O(1)

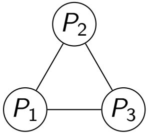
Time complexity

- Response time (under light load):
  - Best case: E
  - Worst case: 2T+E
- Synchronization delay (under heavy load): T

Token-Ring Algorithm

Main idea

- Processes are (logically) organized along a ring
- Permission to enter the CS is represented by a *token*
- When unused, token sent to the next process in ring



Illustration

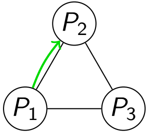
Events

- Initially, P_1 has the token, and P_2 and P_3 want the CS. P_1 sends the token

Token-Ring Algorithm

Main idea

- Processes are (logically) organized along a ring
- Permission to enter the CS is represented by a *token*
- When unused, token sent to the next process in ring



Illustration

Events

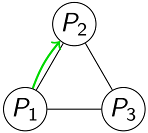
- Initially, P_1 has the token, and P_2 and P_3 want the CS. P_1 sends the token

d. P_2 gets the token $\leadsto$ enters CS.

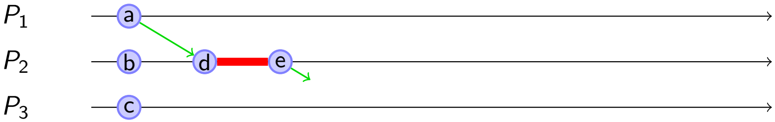
Token-Ring Algorithm

Main idea

- Processes are (logically) organized along a ring
- Permission to enter the CS is represented by a *token*
- When unused, token sent to the next process in ring



Illustration



Events

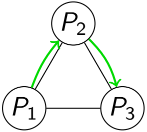
- Initially, P_1 has the token, and P_2 and P_3 want the CS. P_1 sends the token

d. P_2 gets the token $\rightsquigarrow$ enters CS. e. P_2 exits CS and send token to P_3

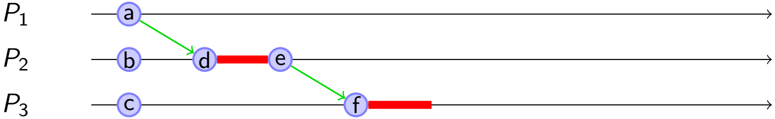
Token-Ring Algorithm

Main idea

- Processes are (logically) organized along a ring
- Permission to enter the CS is represented by a *token*
- When unused, token sent to the next process in ring



Illustration



Events

- Initially, P_1 has the token, and P_2 and P_3 want the CS. P_1 sends the token

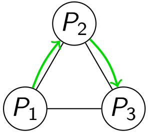
d. P_2 gets the token $\rightsquigarrow$ enters CS. e. P_2 exits CS and send token to P_3

f. P_3 gets the token $\rightsquigarrow$ enters CS.

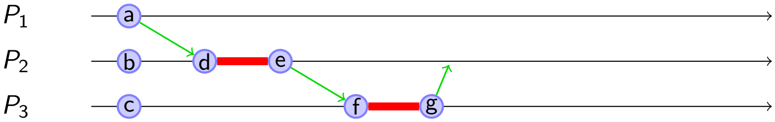
Token-Ring Algorithm

Main idea

- Processes are (logically) organized along a ring
- Permission to enter the CS is represented by a *token*
- When unused, token sent to the next process in ring



Illustration



Events

- Initially, P_1 has the token, and P_2 and P_3 want the CS. P_1 sends the token

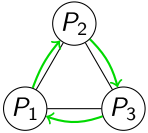
d. P_2 gets the token $\rightsquigarrow$ enters CS. e. P_2 exits CS and send token to P_3

f. P_3 gets the token $\rightsquigarrow$ enters CS. g. P_3 exits CS and send token to P_1

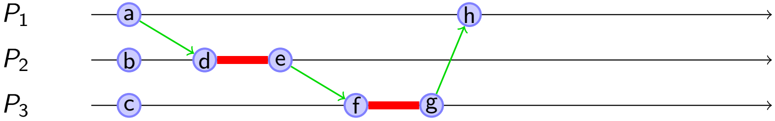
Token-Ring Algorithm

Main idea

- Processes are (logically) organized along a ring
- Permission to enter the CS is represented by a *token*
- When unused, token sent to the next process in ring



Illustration



Events

- Initially, P_1 has the token, and P_2 and P_3 want the CS. P_1 sends the token

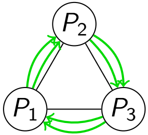
d. P_2 gets the token $\rightsquigarrow$ enters CS. e. P_2 exits CS and send token to P_3

f. P_3 gets the token $\rightsquigarrow$ enters CS. g. P_3 exits CS and send token to P_1

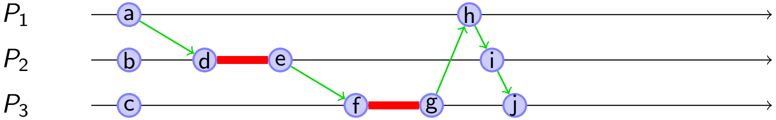
Token-Ring Algorithm

Main idea

- Processes are (logically) organized along a ring
- Permission to enter the CS is represented by a *token*
- When unused, token sent to the next process in ring



Illustration



Events

- Initially, P_1 has the token, and P_2 and P_3 want the CS. P_1 sends the token

d. P_2 gets the token $\rightsquigarrow$ enters CS. e. P_2 exits CS and send token to P_3

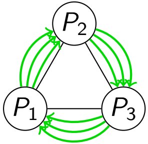
f. P_3 gets the token $\rightsquigarrow$ enters CS. g. P_3 exits CS and send token to P_1

- Seems interesting, but incredibly inefficient when nobody request the CS

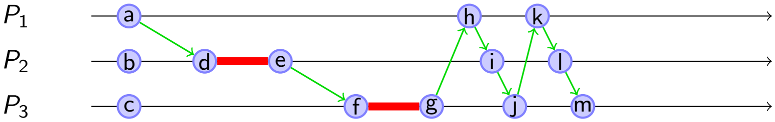
Token-Ring Algorithm

Main idea

- Processes are (logically) organized along a ring
- Permission to enter the CS is represented by a *token*
- When unused, token sent to the next process in ring



Illustration



Events

- Initially, P_1 has the token, and P_2 and P_3 want the CS. P_1 sends the token

d. P_2 gets the token $\rightsquigarrow$ enters CS. e. P_2 exits CS and send token to P_3

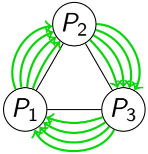
f. P_3 gets the token $\rightsquigarrow$ enters CS. g. P_3 exits CS and send token to P_1

- Seems interesting, but incredibly inefficient when nobody request the CS

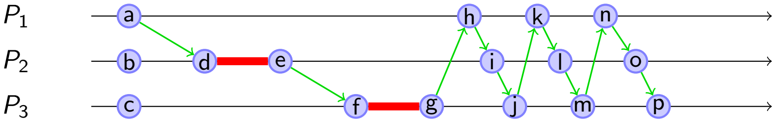
Token-Ring Algorithm

Main idea

- Processes are (logically) organized along a ring
- Permission to enter the CS is represented by a *token*
- When unused, token sent to the next process in ring



Illustration



Events

- Initially, P_1 has the token, and P_2 and P_3 want the CS. P_1 sends the token

d. P_2 gets the token $\leadsto$ enters CS. e. P_2 exits CS and send token to P_3

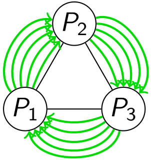
f. P_3 gets the token $\leadsto$ enters CS. g. P_3 exits CS and send token to P_1

- Seems interesting, but incredibly inefficient when nobody request the CS

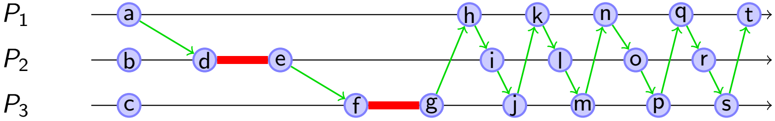
Token-Ring Algorithm

Main idea

- Processes are (logically) organized along a ring
- Permission to enter the CS is represented by a *token*
- When unused, token sent to the next process in ring



Illustration



Events

- Initially, P_1 has the token, and P_2 and P_3 want the CS. P_1 sends the token

d. P_2 gets the token $\leadsto$ enters CS. e. P_2 exits CS and send token to P_3

f. P_3 gets the token $\leadsto$ enters CS. g. P_3 exits CS and send token to P_1

- Seems interesting, but incredibly inefficient when nobody request the CS

Suzuki and Kasami's Algorithm

Main ideas

- Token-based (but not as inefficiently)
- The token is not passed automatically, but on request only

Data structures

- Each process has a vector: v[i]=amount of CS request received from P_i
  This is a local variable
- The token contains 2 informations:
  - A vector: v[i]= amount of CS run for P_i
  - A FIFO: processes with unfulfilled requests

  This is a "global" variable, spead when possible
- These are not vector clocks

Suzuki and Kasami's Algorithm Steps for P_i

On requesting the CS

- If have token, enter CS
- If not, update request vector, then broadcast REQUEST to every processes

On receiving a REQUEST from P_j

- Update request vector
- if (request is new) AND (have token) AND (token idle), then send token to P_j

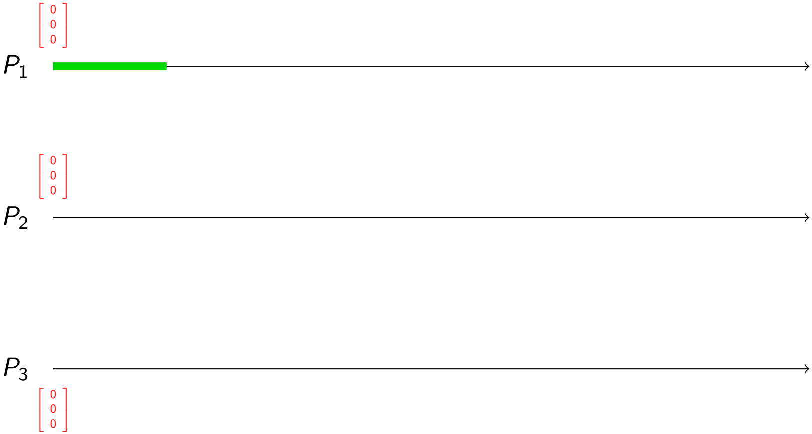
On receiving the token

- Enter the CS

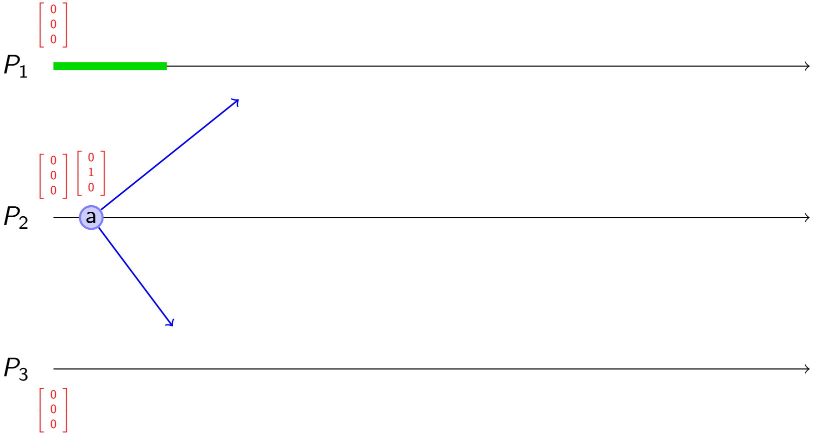
On leaving the CS

- Update the token vector
- Add any unfulfilled requests from request vector to the token queue
- If token queue non-empty, then remove first and send the token that process

Suzuki and Kasami's Algorithm: Illustration events



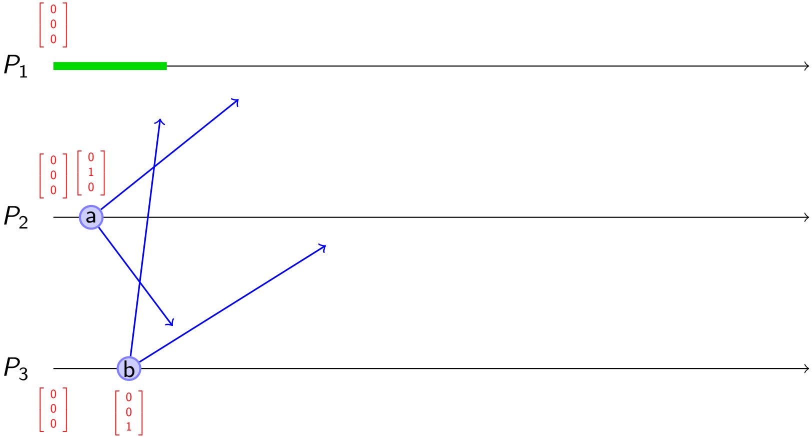
Suzuki and Kasami's Algorithm: Illustration events



a. P_2 requests the CS

$\rightsquigarrow$ broadcasts the REQUEST

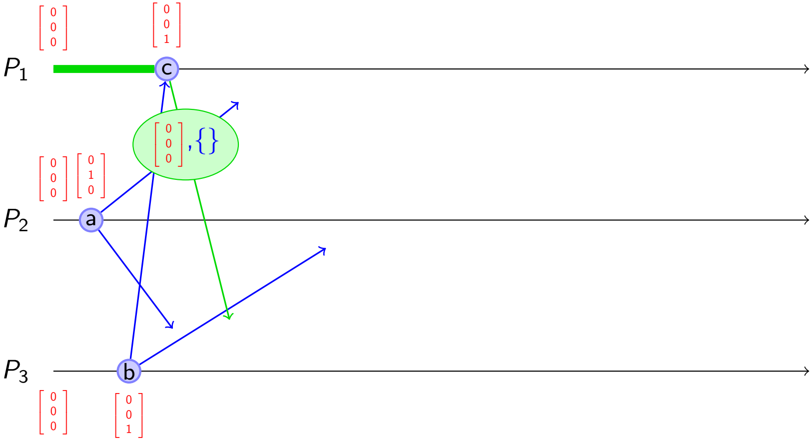
Suzuki and Kasami's Algorithm: Illustration events



b. P_3 requests the CS

$\rightsquigarrow$ broadcasts the REQUEST

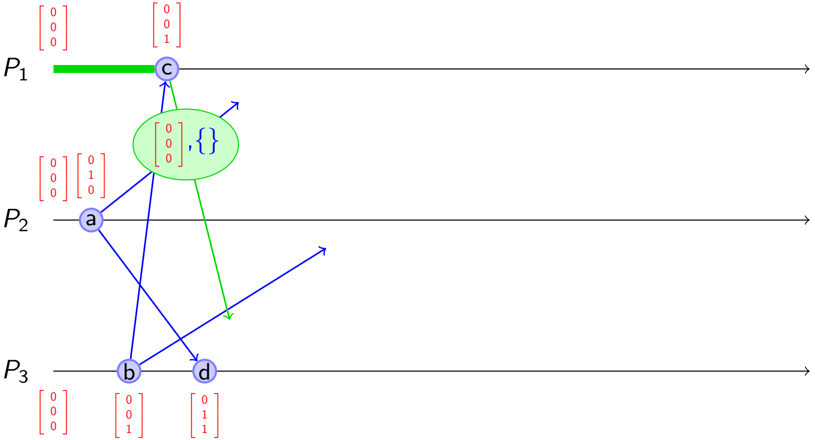
Suzuki and Kasami's Algorithm: Illustration events



c. P_1 receives REQUEST from P_3.

$\leadsto$ Update request vector and send TOKEN

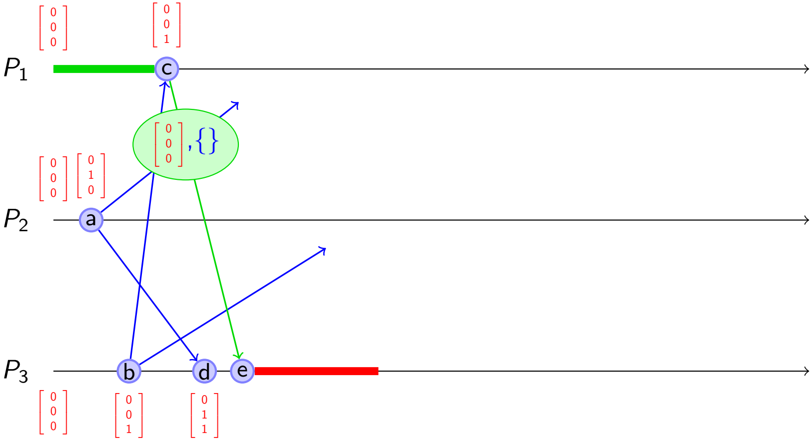
Suzuki and Kasami's Algorithm: Illustration events



d. P_3 receives REQUEST from P_2.

$\leadsto$ update request vector

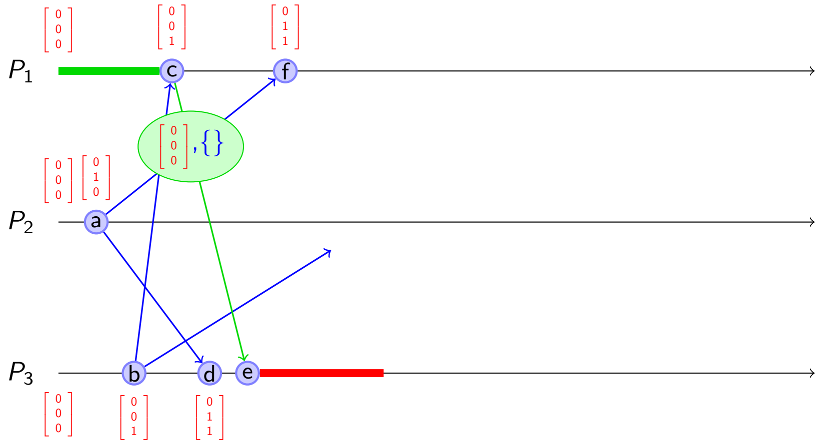
Suzuki and Kasami's Algorithm: Illustration events



e. P_3 receives TOKEN

$\rightsquigarrow$ enters CS

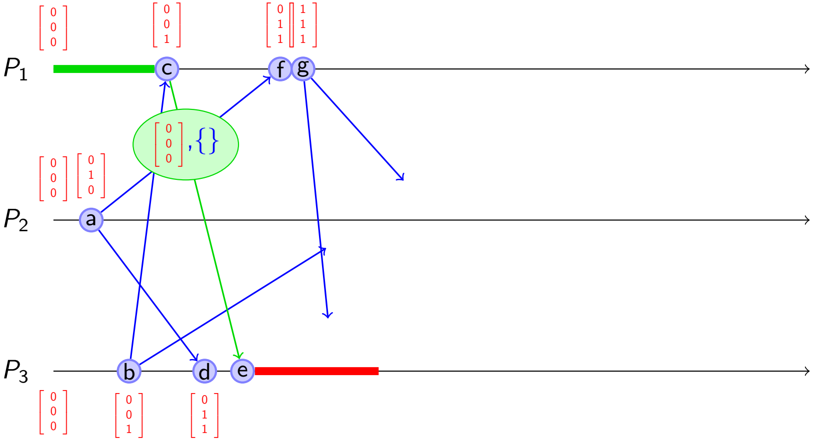
Suzuki and Kasami's Algorithm: Illustration events



f. P_1 receives REQUEST from P_2

$\rightsquigarrow$ update request vector

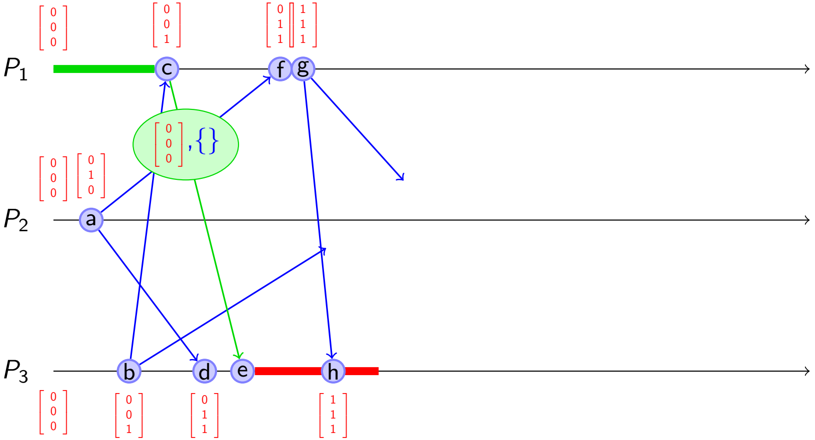
Suzuki and Kasami's Algorithm: Illustration events



g. P_1 requests the CS

$\rightsquigarrow$ increment own entry, broadcast REQUEST to all

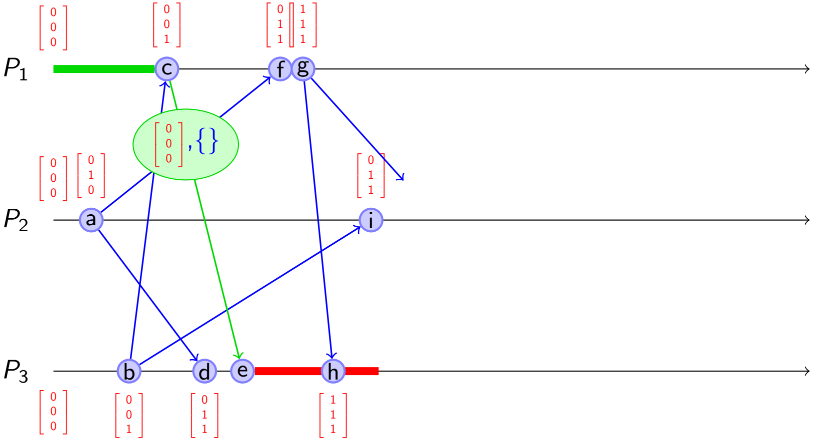
Suzuki and Kasami's Algorithm: Illustration events



h. P_3 receives REQUEST from P_1

$\rightsquigarrow$ update request vector

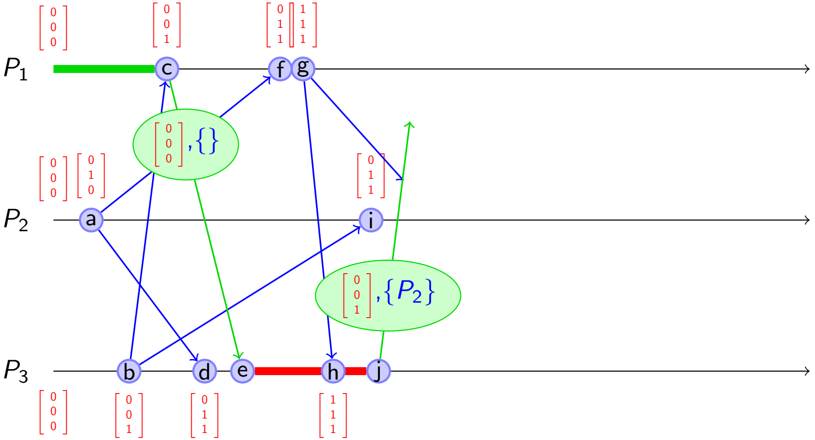
Suzuki and Kasami's Algorithm: Illustration events



i. P_2 receives REQUEST from P_3

$\rightsquigarrow$ update request vector

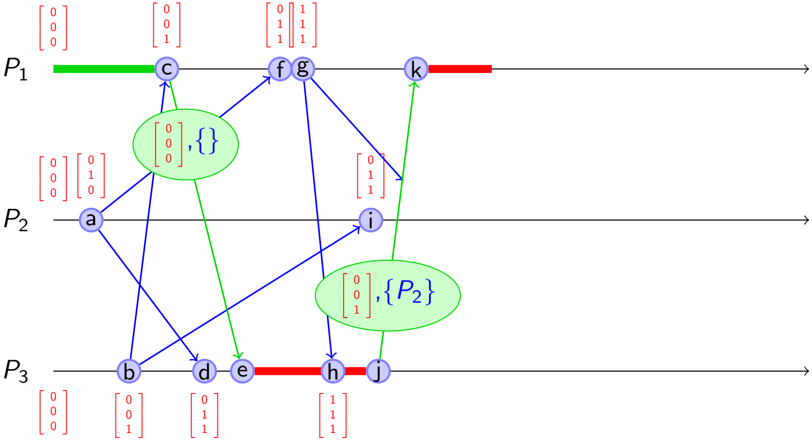
Suzuki and Kasami's Algorithm: Illustration events



j. P_3 exits C.

- Update token vector to $\begin{bmatrix}0\\0\\1\end{bmatrix}$ since it just did a CS
- Compares request and token vectors. $\{P_1, P_2\}$: $\#req. > \#runs \rightsquigarrow$ Enqueue
- Send TOKEN to first of queue, P_1

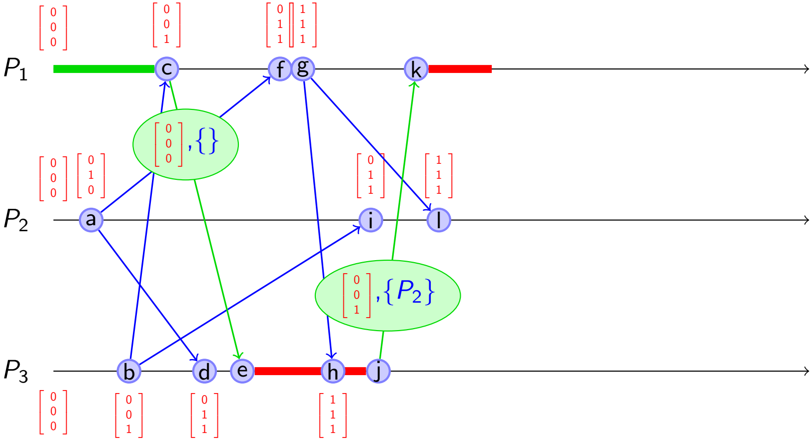
Suzuki and Kasami's Algorithm: Illustration events



k. P_1 receives TOKEN

$\rightsquigarrow$ enters CS

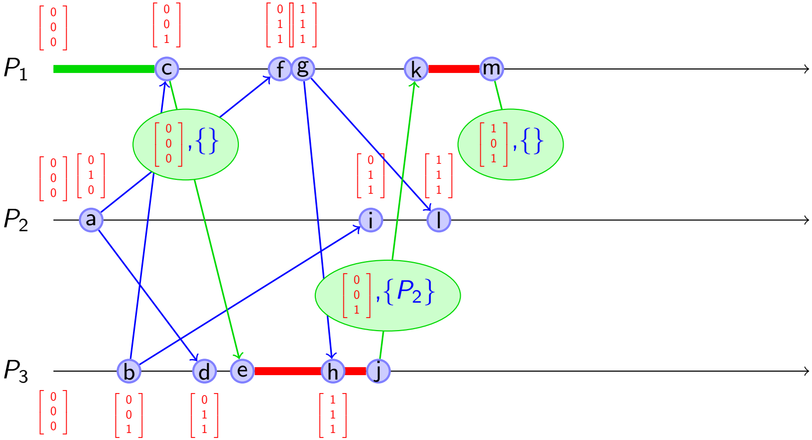
Suzuki and Kasami's Algorithm: Illustration events



l. P_2 receives REQUEST from P_1

$\rightsquigarrow$ updates request vector

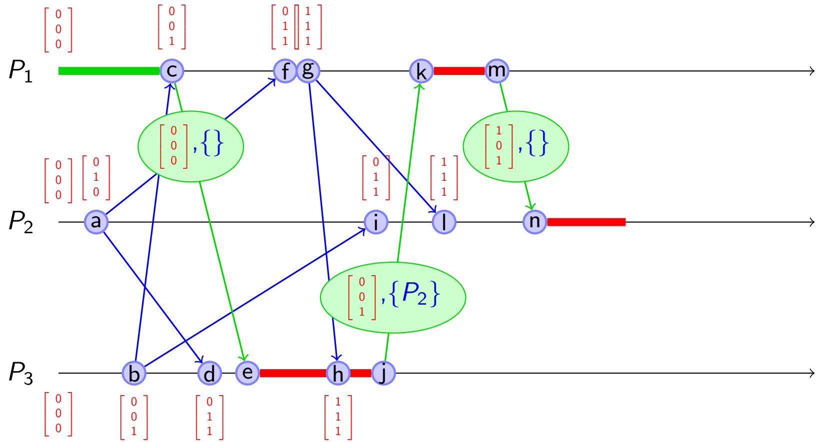
Suzuki and Kasami's Algorithm: Illustration events



m. P_1 exits CS

Update token and send it to P_2

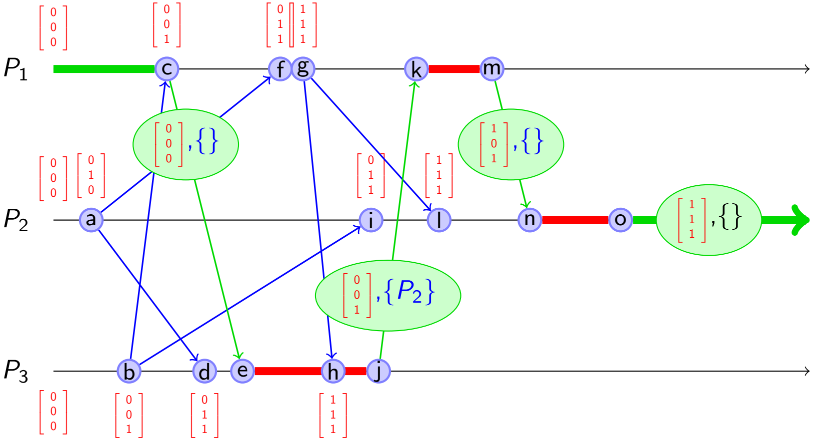
Suzuki and Kasami's Algorithm: Illustration events



n. P_2 receives TOKEN

$\rightsquigarrow$ enters CS

Suzuki and Kasami's Algorithm: Illustration events



o. P_2 exits CS

Update token and keep it

Suzuki and Kasami's Algorithm: Complexity Analysis

Parameters

N Number of processes in the system
T Message transmission time
E Critical section execution time

Message complexity:

- Best case: 0
- Worst case: N= $(N-1)$ REQUEST + 1 TOKEN

Message Size Complexity:

- Between 1 (REQUEST) and N (TOKEN)
- Average: O(1) (averaging over $(N-1)$ REQUEST and 1 TOKEN)

Time complexity

- Response time (under light load): Best case: E; Worst case: 2T+E
- Synchronization delay (under heavy load): T

(pedagical) Interest of this algorithm

Builds a sort of distributed data structure

- Explicit list in token, which travels
- (built lazily by comparing local request vector to token vector)
- Request vectors are updated when receiving a REQUEST

This concept is still somehow fuzzy

- List updated only when needed: when exiting the CS (lazy update)
- List updated by comparing local request vector to [global] token vector
- Request vectors are updated when receiving a REQUEST

Other algorithm use distributed data structures more explicitely

- Raymond and Naimi-Trehel build a waiting queue, and a tree pointing to the waiting queue entry point

Premier chapitre

Some Distributed Algorithms

- Some Distributed Algorithms
  - Mutual Exclusion
  - Leader Election
  - Consensus
- Conclusion on distributed algorithmic

Leader Election

Problem Statement

- The processes pick one and only one of them (and agree on which one)
- Use case: error recovery
  - Only one site recreates the (lost) token
  - Elect a new coordinator on need
- Election started by any process (maybe concurrent elections)
- Which one we pick is not important
- Difficulty: processes may fail during the election

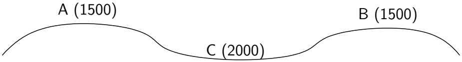
Some approaches

- Bully Algorithm
  - Main idea
    - The one starting the election broadcasts its process number
    - Processes answer (take over) elections with a number smaller than their own
    - A process receiving no answer consider that he got elected
  - Remarks
    - Not very efficient algorithm ($O(n^2)$ messages at worst)
    - Robust to process failures, but not to asynchronism
- Ring $\Rightarrow$ Algorithm in $O(n \log(n))$ on average [Chang, Roberts]

Consensus: First impossibility result

Byzantin generals problem

- A and B want to attack C
- They must absolutely do it at the same time to succeed
- C can intercept messengers



A $\rightarrow$ B: Attack tomorrow
B $\rightarrow$ A: Got(Attack tomorrow)
A $\rightarrow$ B: Got(Got(Attack tomorrow))

A cannot be absolutely sure that B got his last message $\Rightarrow$ he does not attack

messages lost without detection $\rightsquigarrow$ consensus impossible (in finite amount of steps)

- Proof (reductio ad absurdum): Suppose $\exists$ such a protocol, consider $p = \{\ldots; A \rightarrow B : m_{n-1}; B \rightarrow A : m_n\}$ minimal in amout of messages.
  - B don't receive messages anymore $\Rightarrow$ casted its decision before m_n
  - Since p works even if messages get lost, A casts its decision without m_n

  $\Rightarrow$ m_n useless, and can be omitted from p. Contradiction with "p is minimal"
- Only solution: detect message loss

Consensus: An algorithm amongst others

Lamport et al. (1982)

- Goal:
  - Generals want to inform each other of the present forces
- Assumptions:
  - Messages not corrupted (communication are *fail-stop*)
  - Receiver knows who sent the message
  - Communication time bounded (implementation: timestamp + timeouts + *fail-fast*)
- Result:
  - With m malicious generals, need $2m + 1$ generals in total
  - Cannot identify malicious generals, only find correct values out
- Principe:
  1. Everyone broadcasts its own force to everyone
  2. Everyone broadcasts the vector of received values to everyone
  3. Everyone uses the vectors getting the majority of the casts

Conclusion

What we saw

- Notion of distributed system (DS)
- Notion of time and state in a DS
- Main issues of faults in DS
- Expected properties of a DS:
  Safety, liveness (no deadlock, finishing), Scalability, Fault tolerance
- Classical problems in DS, and ideas of some algorithms
- Some classical approaches to solve these issues
  Order/abstract clocks, applicative topologies, Symmetry breaking (token, leader)

What we didn't saw (because of lack of time)

- Notion of security in DS
- Every details of every algorithms
- A whole load of other problems, also quite classical:
  Wave algorithms; Distributed commits (2PC/3PC); Checkpointing; Ending detection

What you should remember

The models

- No shared time, no shared memory
- Asynchronism, Failures

The tools

- Abstract clocks, applicative topologies, token-based

The presented algorithms

- Mutex: Centralized, Lamport, Ricart/Agrawala, Roucairol/Carvalho, Suzuki/Kasami
  (you should be able to run them on a provided initial situation)
- The other ones (only the spirit)

I hope you got the spirit of classical DS

- Even if I would need more time to get into real details