CS 60002: Distributed Systems

T10: Distributed Mutual Exclusion

Department of Computer Science and Engineering

INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR

Sandip Chakraborty
sandipc@cse.iitkgp.ac.in

Mutual Exclusion (Mutex)

- A program object to prevent simultaneous access to the shared resources, used in concurrent programming with a **critical section** (a part of code where processes access shared resources)
  - Well understood in case of shared memory systems
  - **Reader-Writer:** Two processes can read simultaneously, but read-write or write-write cannot happen simultaneously

Mutual Exclusion (Mutex)

- A program object to prevent simultaneous access to the shared resources, used in concurrent programming with a **critical section** (a part of code where processes access shared resources)
  - Well understood in case of shared memory systems
  - **Reader-Writer:** Two processes can read simultaneously, but read-write or write-write cannot happen simultaneously

- **Requirements:**
  - At most one process should execute the critical section (**Safety**)
  - If more than one processes request to execute the critical section, at least one of them enters (**liveness**)
  - A process executes the critical section for a finite time (**no starvation**)
  - The requests are granted in some order (**fairness**)

Distributed Mutual Exclusion

- **Permission-based Algorithms**
  - A process takes permission from all other processes (or a subset of that) before entering the critical section
  - **Permission from all**: Costly, good for small systems
  - **Permission from subset:** scalable, widely used; however, the main challenge is how to choose the subset for each process?

Distributed Mutual Exclusion

- **Permission-based Algorithms**
  - A process takes permission from all other processes (or a subset of that) before entering the critical section
  - **Permission from all**: Costly, good for small systems
  - **Permission from subset:** scalable, widely used; however, the main challenge is how to choose the subset for each process?

- **Token-based Algorithms**
  - Single token in the system
  - A process enters the critical section if it has a token
  - There are different algorithms based on how the token is circulated across different requesting processes

Adaptive vs Non-adaptive Algorithms

- Less requests for entering the critical section → Low load
- Performance of a distributed mutual exclusion algorithm under low load should be better than that under high load
- An algorithm is called adaptive if the performance is dependent on the load
- But, how do we measure the performance?

Metrics for Performance Measurements

- **Message complexity** -- Number of messages per critical section entry
- **Synchronization delay** – Time required to enter the critical section after a process leaves the critical section
- **Response time** – The time interval a request waits for its critical section execution to be over after its request messages have been sent out
- **Throughput** – The rate at which critical sections are executed, computed as 1/(synchronization delay + critical section execution time)

Permission-based Algorithms

Lamport's Algorithm

- Permission from all, contention-based algorithm
  - Processes contend with each other to enter the critical section
- Uses logical clock to order the messages
  - Indeed, the algorithm was given as an example of the use of Logical clocks
- System model: Asynchronous, Completely-connected topology, Reliable and FIFO channel

Lamport's Algorithm

- Every process maintains a queue of pending requests for entering critical section in that order
  - The requests in the queue is ordered based on the logical clock timestamp
  - Total order is enforced by including process IDs in the timestamp

Lamport's Algorithm

- Every process maintains a queue of pending requests for entering critical section in that order
  - The requests in the queue is ordered based on the logical clock timestamp
  - Total order is enforced by including process IDs in the timestamp

- **Process i requests to enter the critical section**
  - Send timestamped REQUEST (ts_i, i) to all other processes
  - Put (ts_i, i) in its own queue

Lamport's Algorithm

- Every process maintains a queue of pending requests for entering critical section in that order
  - The requests in the queue is ordered based on the logical clock timestamp
  - Total order is enforced by including process IDs in the timestamp

- **Process i requests to enter the critical section**
  - Send timestamped REQUEST (ts_i, i) to ALL other processes
  - Put (ts_i, i) in its own queue

- **On receiving a request (ts_i, i)**
  - Send timestamped REPLY to the requesting node i, requests are granted in the order of timestamp
  - Put the REQUEST (ts_i, i) in the queue

Lamport's Algorithm

- **Process i enters the critical section if**
  - (ts_i, i) is at the top of its own queue
  - Process i has received a message (REQUEST or REPLY) with a timestamp larger than ts_i from ALL other processes

Lamport's Algorithm

- **Process i enters the critical section if**
  - (ts_i, i) is at the top of its own queue
  - Process i has received a message (REQUEST or REPLY) with a timestamp larger than ts_i from ALL other processes

- **Process i releases the critical section**
  - i removes the request from its own queue and sends a timestamped RELEASE message to ALL other processes

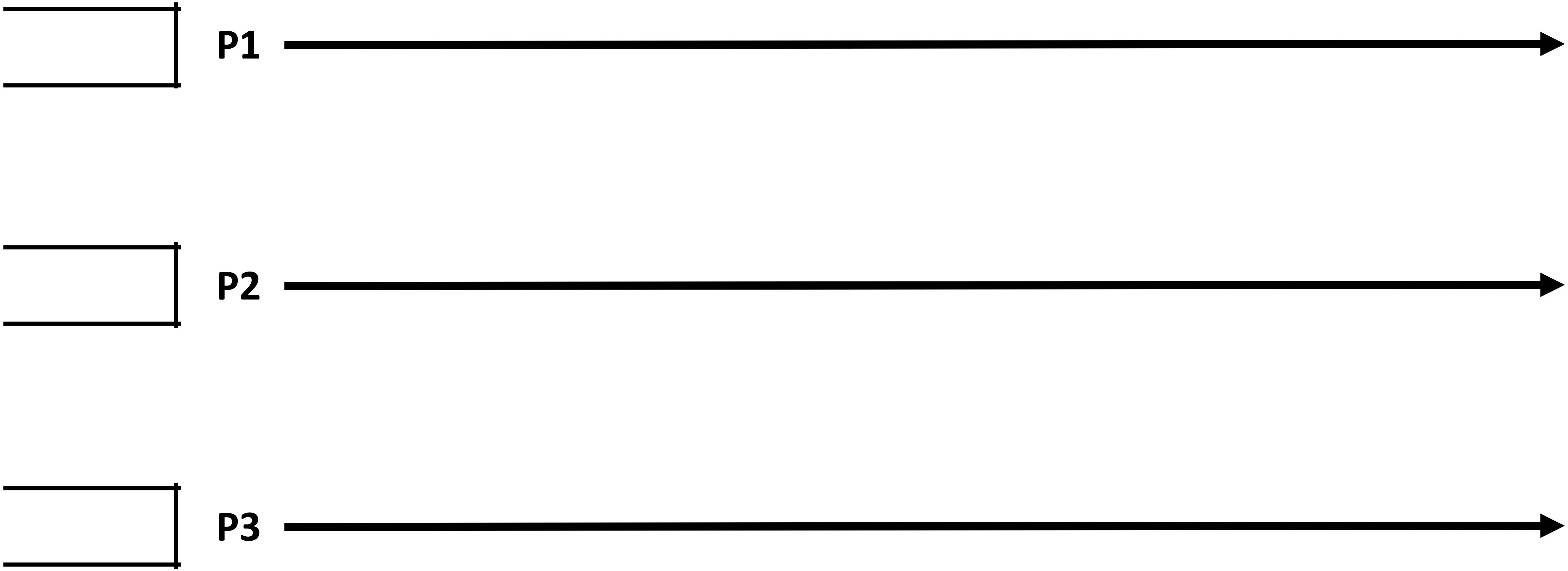
Lamport's Algorithm

- **Process i enters the critical section if**
  - (ts_i, i) is at the top of its own queue
  - Process i has received a message (REQUEST or REPLY) with a timestamp larger than ts_i from ALL other processes

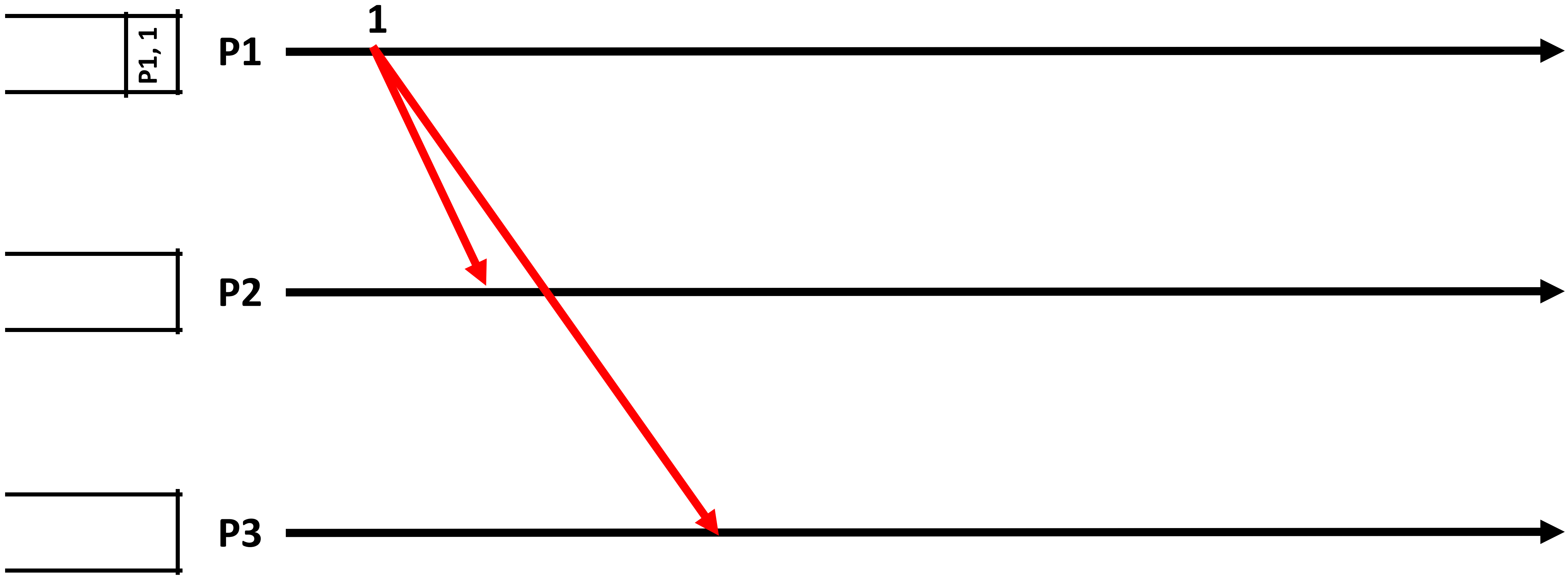
- **Process i releases the critical section**
  - i removes the request from its own queue and sends a timestamped RELEASE message to ALL other nodes

- **On receiving a RELEASE message from i**
  - i's request is removed from the local request queue

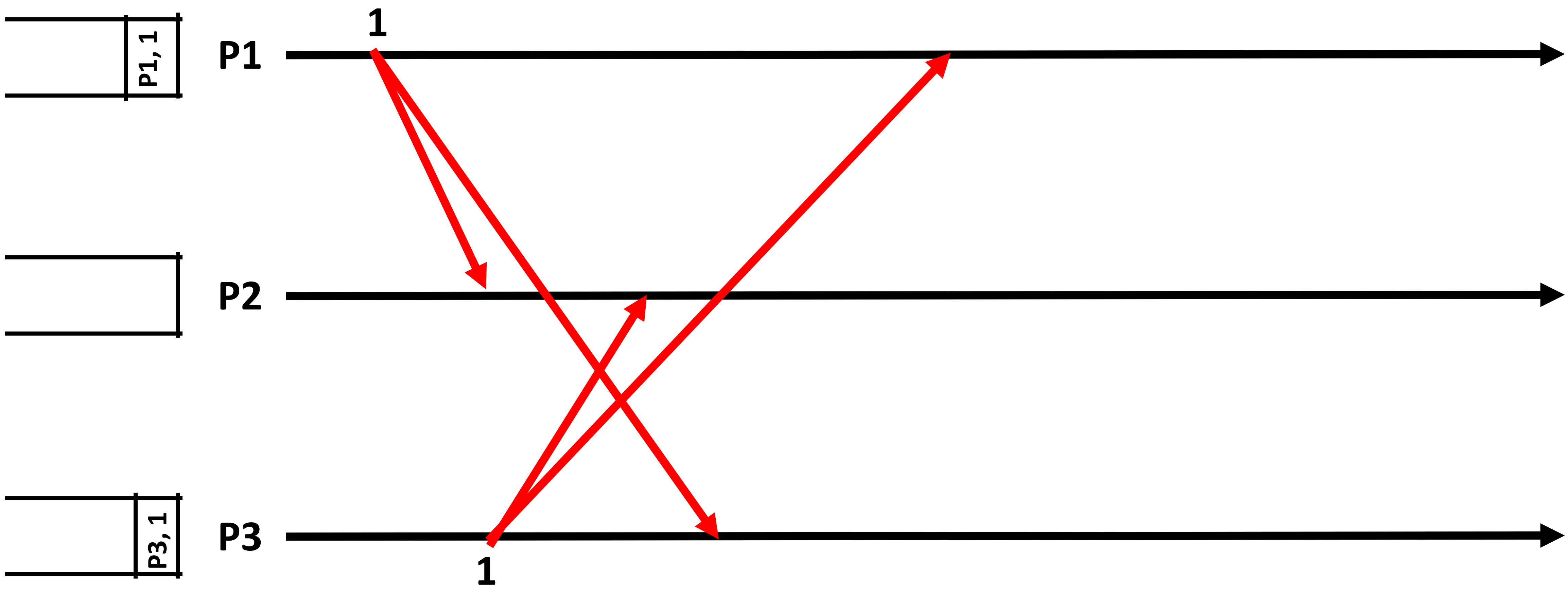
Lamport's Algorithm -- Illustration



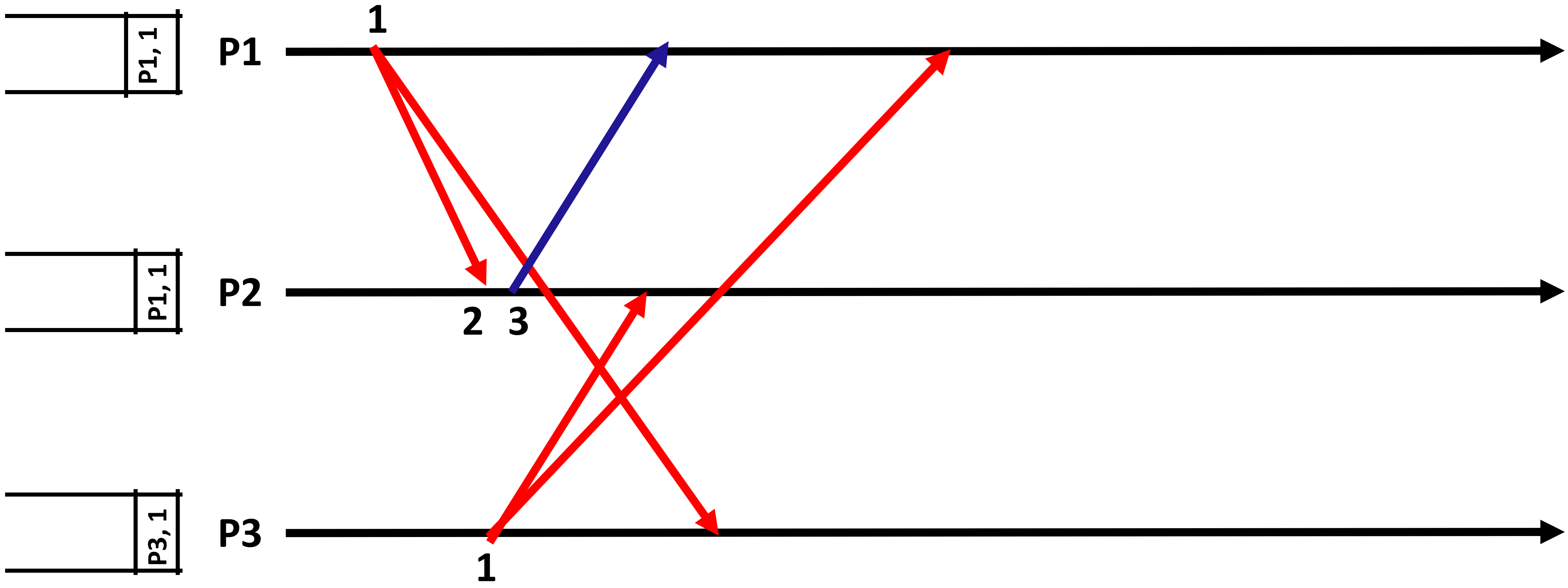
Lamport's Algorithm -- Illustration



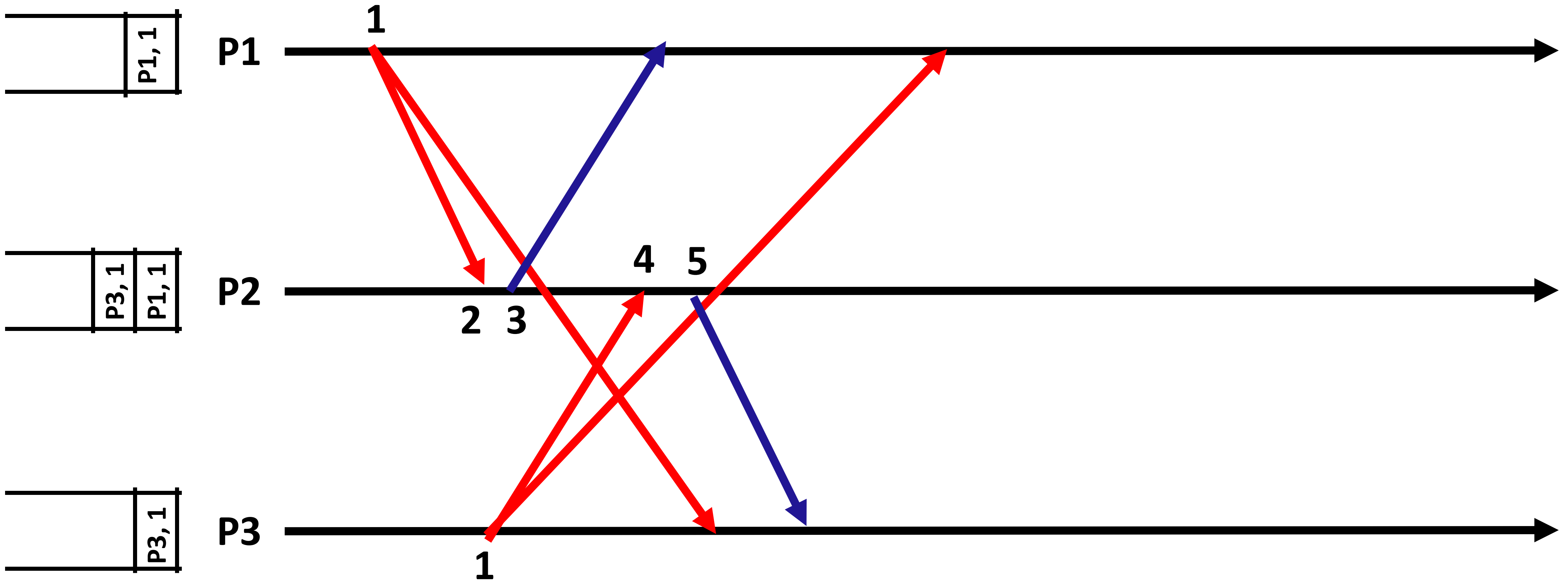
Lamport's Algorithm -- Illustration



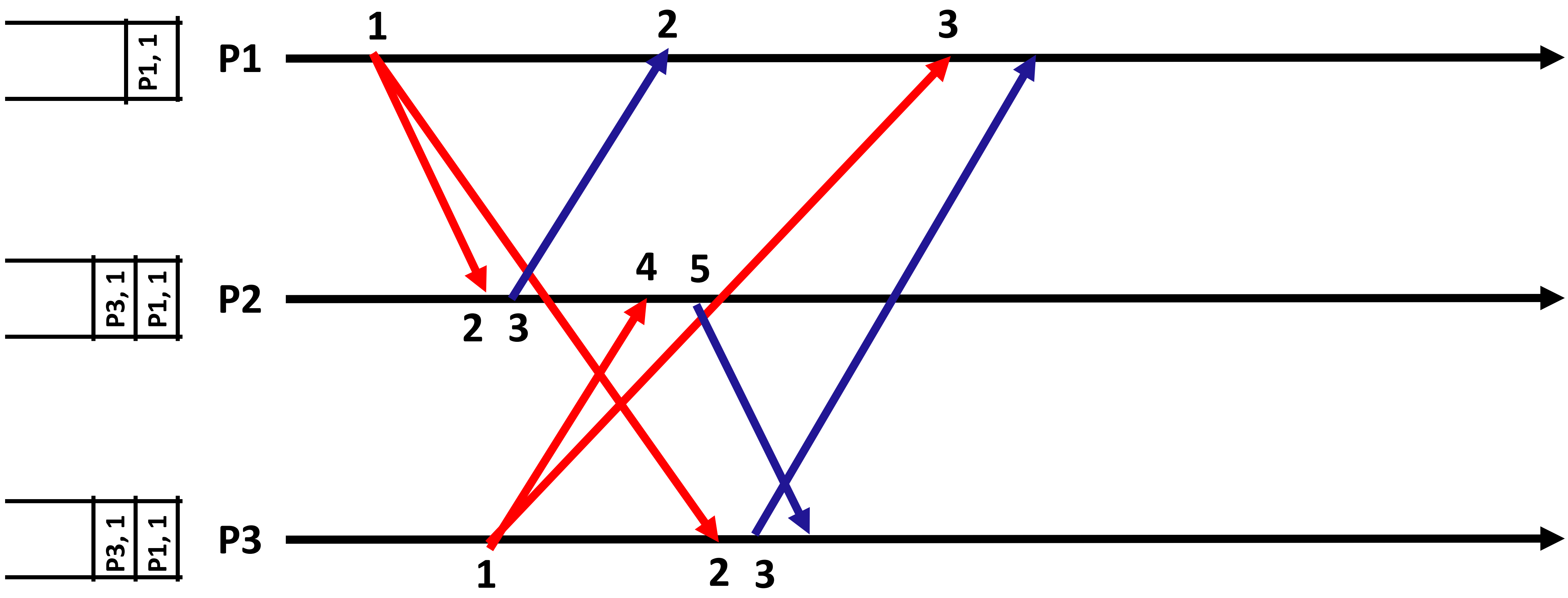
Lamport's Algorithm -- Illustration



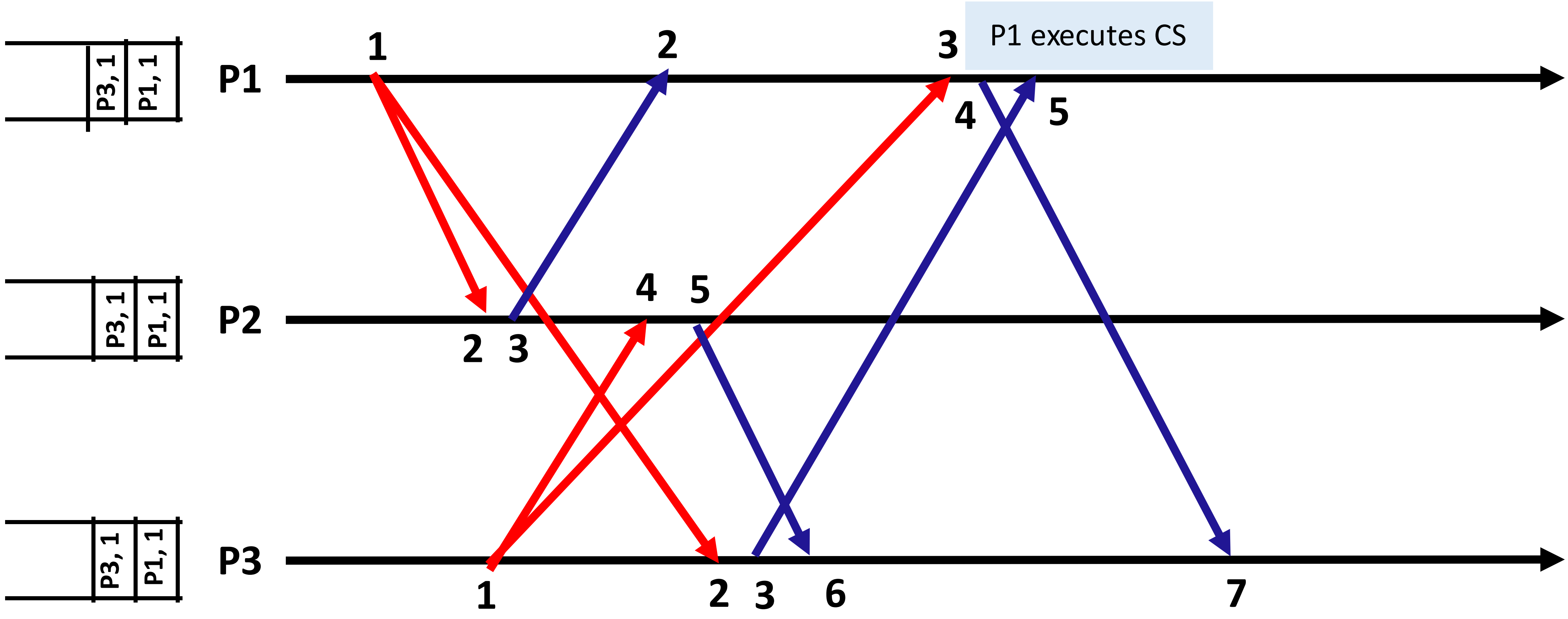
Lamport's Algorithm -- Illustration



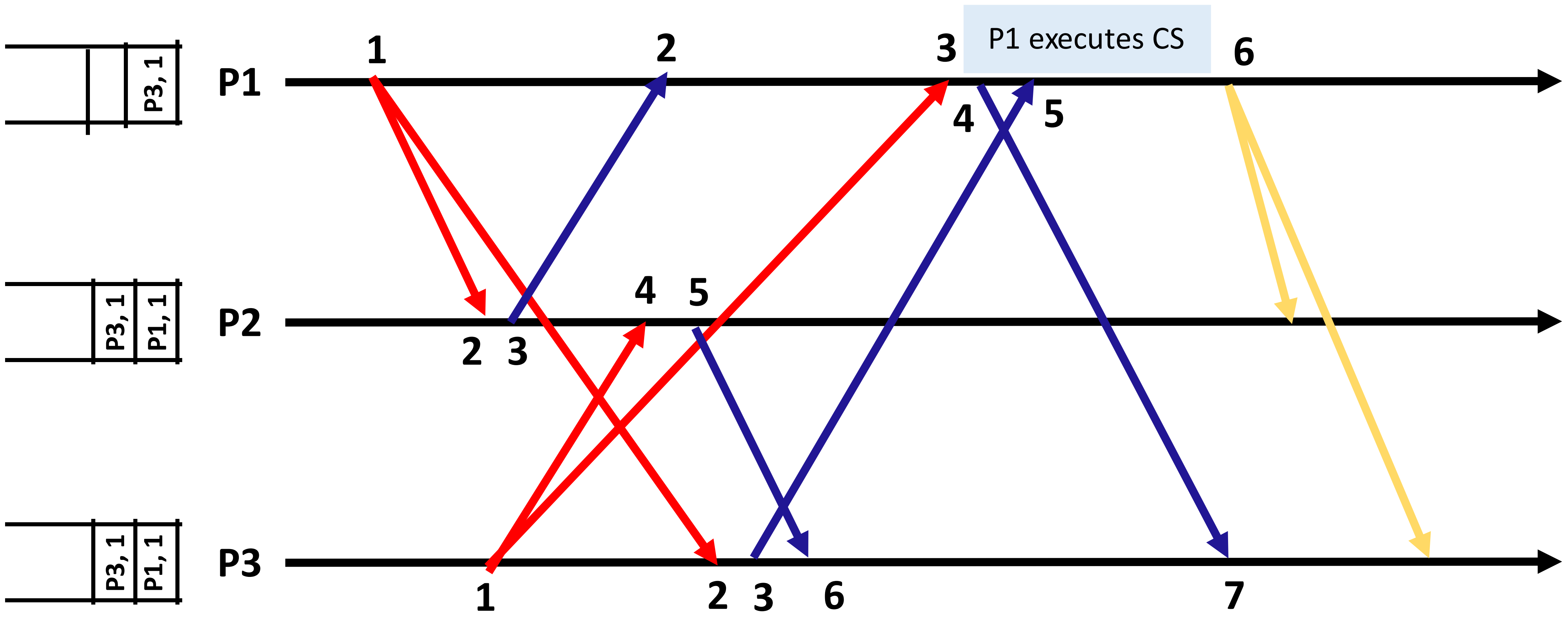
Lamport's Algorithm -- Illustration



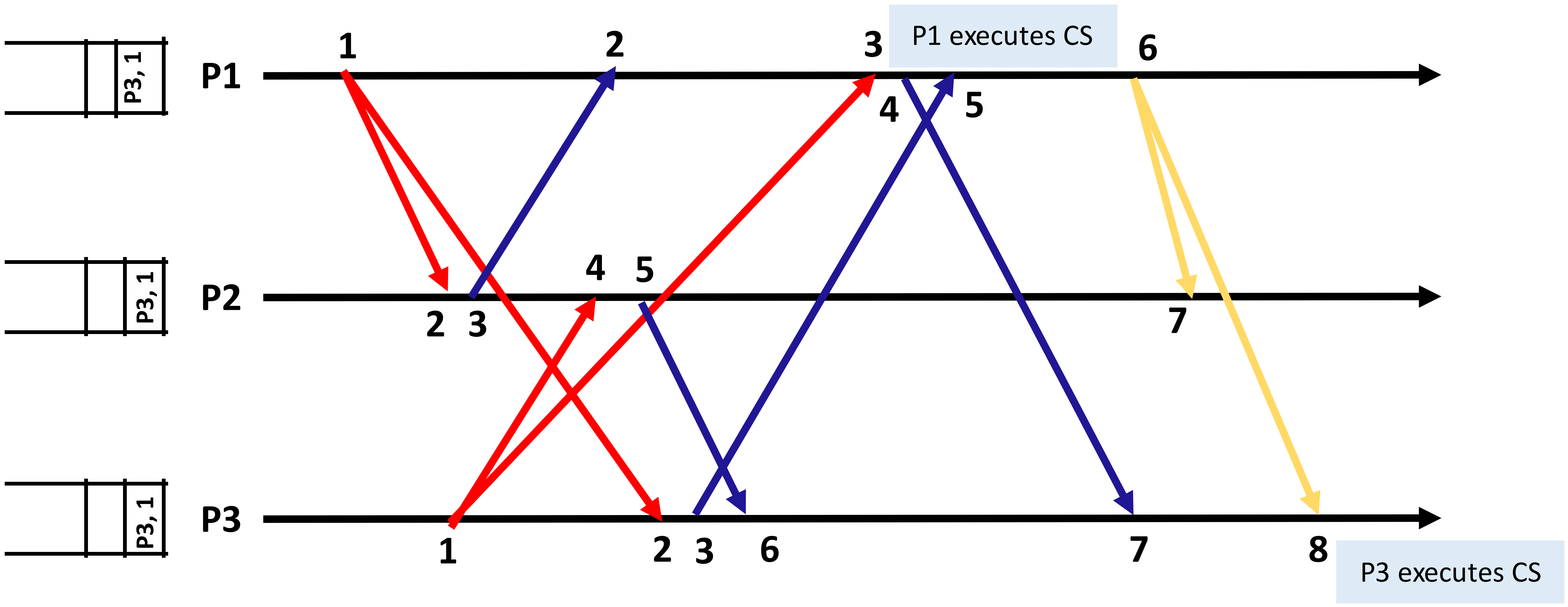
Lamport's Algorithm -- Illustration



Lamport's Algorithm -- Illustration



Lamport's Algorithm -- Illustration



Some Observations

- Timestamp of a REPLY message is greater than the timestamp of the corresponding REQUEST message

- The REPLY message from process i to process j ensures that j knows of all requests of i prior to sending the REPLY
  - This ensures that j knows any possible requests from i with a lower timestamp
  - Requires FIFO channel

- Requests are granted in the order of increasing timestamp

Performance Metrics

- 3(n-1) messages are required per critical section invocation
  - (n-1) REQUEST messages
  - (n-1) REPLY messages
  - (n-1) RELEASE messages

- Synchronization delay equals to maximum message transmission time

Ricart-Agrawala Algorithm

- Improvement over Lamport's Algorithm
  - Developed by Glenn Ricart and Ashok Agrawala
  - Ricart, Glenn; Agrawala, Ashok K. (1981). "An optimal algorithm for mutual exclusion in computer networks". Communications of the ACM. 24 (1): 9–17

Ricart-Agrawala Algorithm

- Improvement over Lamport's Algorithm
  - Developed by Glenn Ricart and Ashok Agrawala
  - Ricart, Glenn; Agrawala, Ashok K. (1981). "An optimal algorithm for mutual exclusion in computer networks". Communications of the ACM. 24 (1): 9–17

- **Core Idea:**
  - Process i has a request timestamp of ts_i and process j has a request timestamp of ts_j
  - $ts_j < ts_i \rightarrow$ i cannot enter the critical section before j
  - Process j needs not send a REPLY immediately to process i in this case
  - Process j sends the REPLY after it completes its critical section
  - No separate RELEASE message is needed

Ricart-Agrawala Algorithm

- **To Request the Critical Section:**
  - Send timestamped REQUEST message (ts_i, i)

- **On receiving Request (ts_i, i) at j:**
  - Send REPLY to i if
    - j is neither requesting nor executing critical section or
    - if j is requesting and i’s request timestamp is smaller than j’s request timestamp.
  - Otherwise, defer the request.

Ricart-Agrawala Algorithm

- **To enter critical section:**
  - i enters critical section on receiving REPLY from all nodes
- **To release critical section:**
  - Send REPLY to all deferred requests

Analysis of the Algorithm

- Requests are granted in order of increasing timestamps
- Does not require FIFO, but requires knowledge of all other processes in the network
- 2(n-1) messages per critical section invocation
- Synchronization delay equals to the maximum message transmission time
- Failure of a process may cause starvation – needs separate fault detection algorithm

Maekawa's Algorithm

- Permission is obtained from a subset of other processes
  - The subset is called the **Request Set** (or **Quorum Set**)
  - Separate request set R_i for each process i

Maekawa's Algorithm

- Permission is obtained from a subset of other processes
  - The subset is called the **Request Set** (or **Quorum Set**)
  - Separate request set R_i for each process i

- **Requirements for the Quorum Set:**
  - For all i, j: $R_i \cap R_j \neq \emptyset$
  - For all i: $i \in R_i$
  - For all i: $|R_i| = K$ for some K
  - Any process i is contained in exactly D quorum sets, for some D
- $K = D$
- $K \geq \mathrm{sqrt}(N-1)$

Maekawa's Algorithm – A Simple Version

- **To request critical section:**
  - i sends REQUEST message to all processes in its quorum set R_i

- **On receiving a REQUEST message**
  - Send a REPLY message if no REPLY message has been sent since the last RELEASE message is received.
  - Update status to indicate that a REPLY has been sent.
  - Otherwise, queue up the REQUEST

Maekawa's Algorithm – A Simple Version

- **To enter critical section:**
  - i enters critical section after receiving REPLY from all processes in R_i

- **To release critical section:**
  - Send RELEASE message to all processes in R_i
  - On receiving a RELEASE message, send REPLY to next process in the queue and delete the process from the queue.
  - If queue is empty, update status to indicate no REPLY message has been sent since last RELEASE is received.

Maekawa's Algorithm – Analysis

- The REPLY message works like a vote
  - To enter the critical section, every process needs to obtain the vote from ALL the processes in its quorum set
- **Safety**: Any two quorum set has at least one process in common
  - The common processes ensure the total ordering of requesting processes
- **Message Complexity:** 3 x sqrt (N)
- Synchronization delay = 2 message transmission delays
- A major issue with the algorithm: **deadlock is possible**
  - $i \in R_j$, $j \in R_i$, i waits for the reply from j, j waits for the reply from i
  - Try to solve this problem with additional messages and logical clock timestamp (indeed, Maekawa's algorithm already does that)
- How do we build the quorum set?

Properties of a Quorum

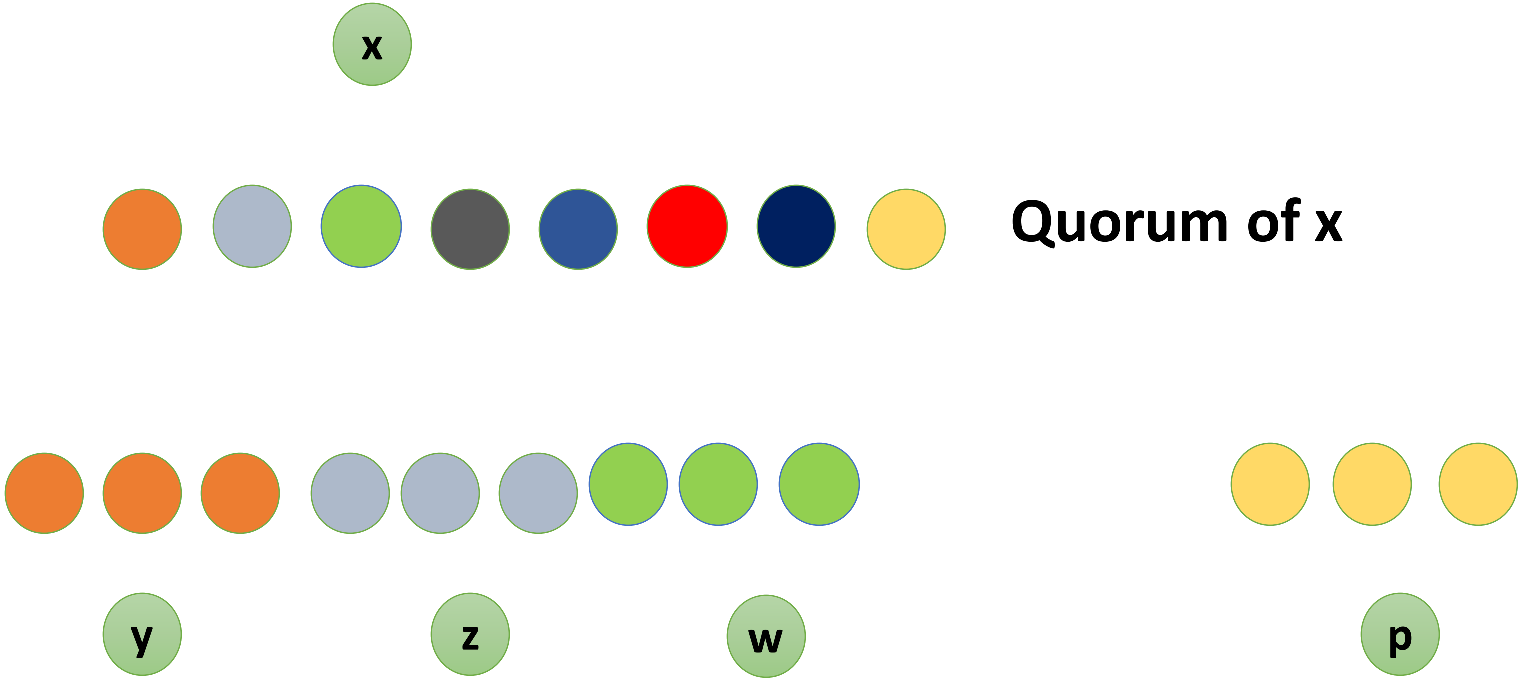
- The set of Quorums is called a **Coterie**. The following properties must hold for quorums in a coterie.

- **Intersection:** For every quorum Q1 and Q2 in C, $Q1 \cap Q2 \neq \emptyset$.
  - For example, {1, 2, 3}, {3, 4, 5} and {7, 8, 9} cannot be quorums in a coterie

- **Minimality:** There should not be any two quorums Q1 and Q2 in C, such that $Q1 \subseteq Q2$.
  - For example, {1, 2, 3} and {1, 2} cannot be quorums of a coterie
  - **Ensure efficiency rather than correctness**

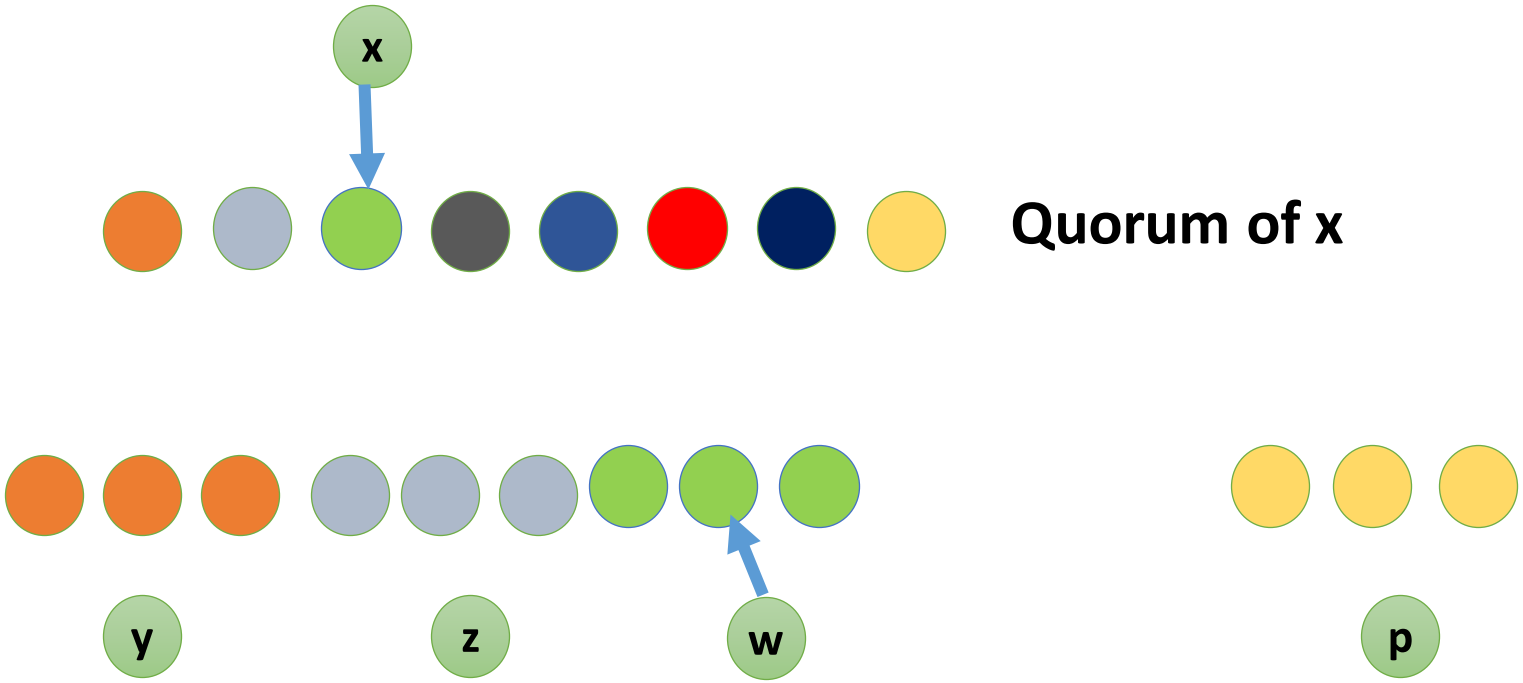
Properties of a Quorum

- Let x is process in quorum Q. If x wants to invoke mutual exclusion, the **Intersection property** ensures that, there is at least one process in the quorum Q that is common to the quorum of every other processes
  - These common processes send permission to only one processes at a time, thus ensuring mutual exclusion



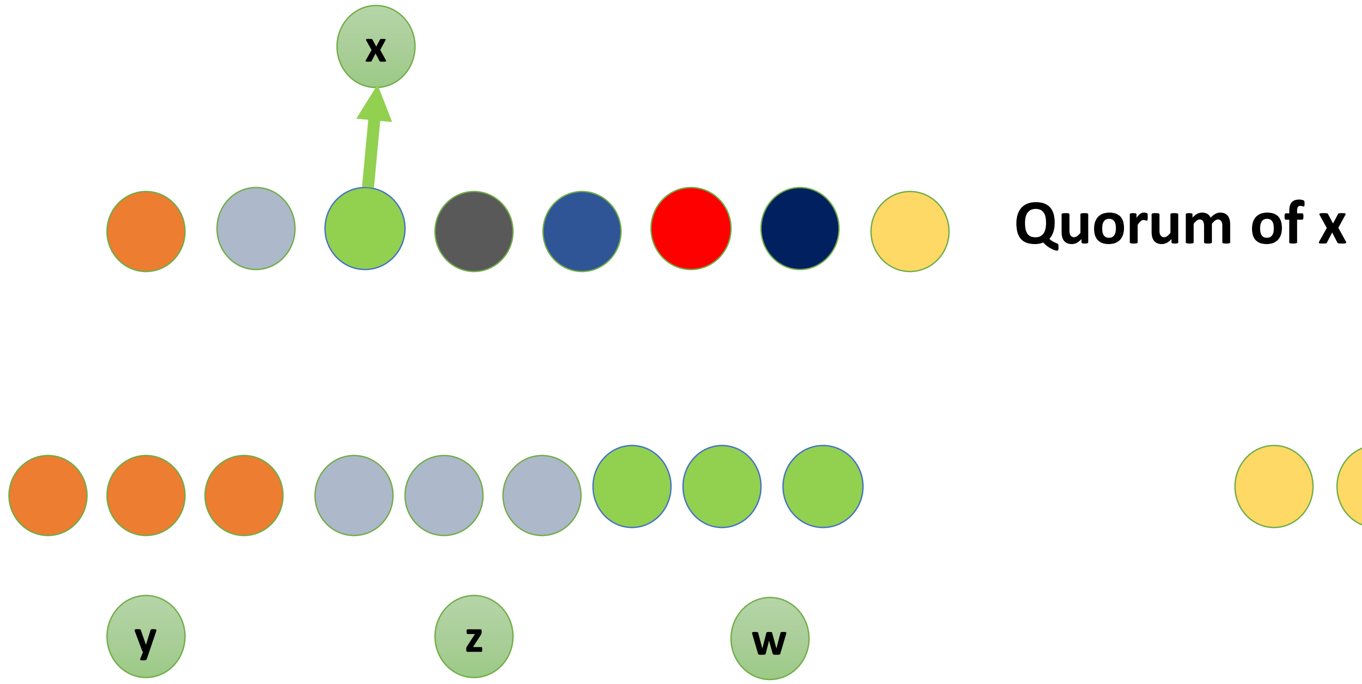
Properties of a Quorum

- Let x is process in quorum Q. If x wants to invoke mutual exclusion, the **Intersection property** ensures that, there is at least one process in the quorum Q that is common to the quorum of every other processes
  - These common processes send permission to only one processes at a time, thus ensuring mutual exclusion



Properties of a Quorum

- Let x is process in quorum Q. If x wants to invoke mutual exclusion, the **Intersection property** ensures that, there is at least one process in the quorum Q that is common to the quorum of every other processes
  - These common processes send permission to only one processes at a time, thus ensuring mutual exclusion



Maekawa's Algorithm – Building the Quorum Set

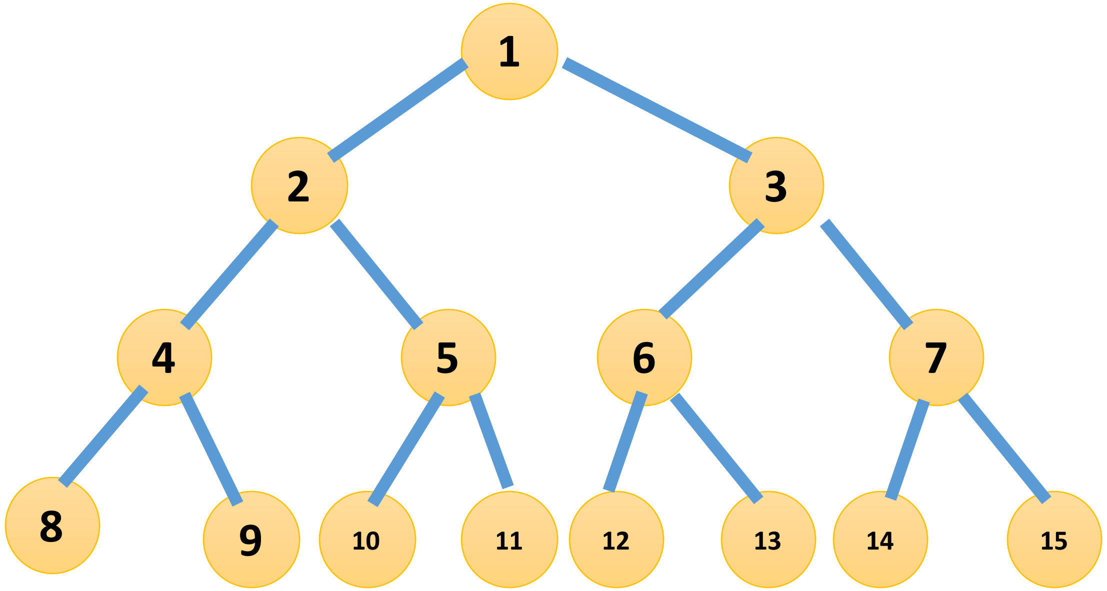
- One possible solution is to use a 2-D grid
  - Need 2*sqrt(N) votes and 6*sqrt(N) messages

1	2	3	4	5	6
7	8	9	10	11	12
13	14	15	16	17	18
19	20	21	22	23	24
25	26	27	28	29	30
31	32	33	34	35	36

Agarwal-El Abbadi Algorithm

- Uses '**tree-structured**' quorums – All the processes are logically organized in a complete binary tree

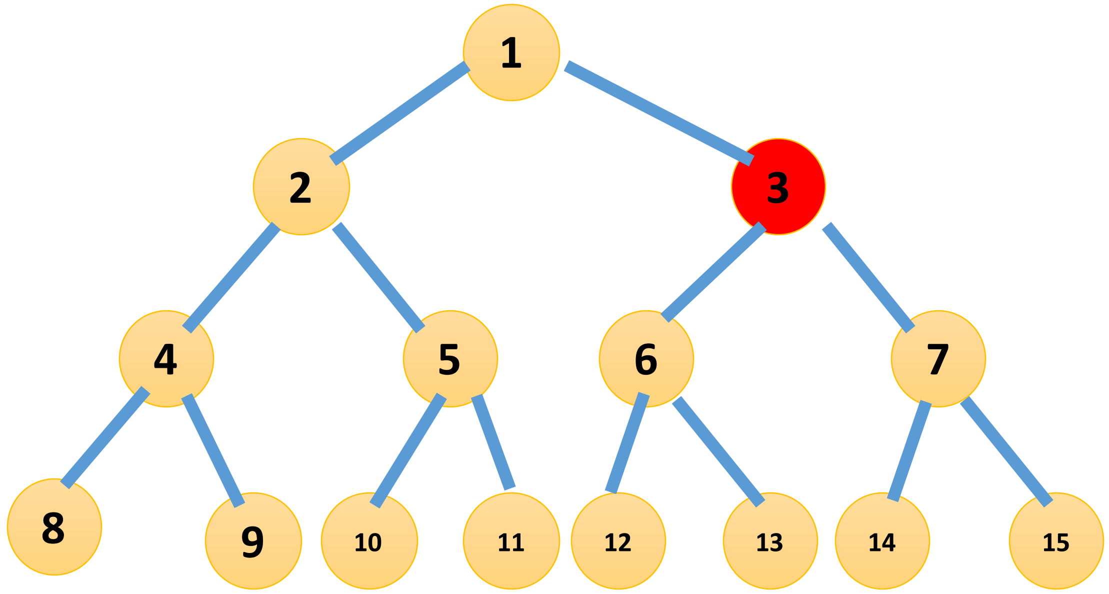
- Each quorum represents a path from the root to the leaf



- Eight quorums
  - {1, 2, 4, 8}
  - {1, 2, 4, 9}
  - {1, 2, 5, 10}
  - {1, 2, 5, 11}
  - {1, 3, 6, 12}
  - {1, 3, 6, 13}
  - {1, 3, 7, 14}
  - {1, 3, 7, 15}

Agarwal-El Abbadi Algorithm

- Number of processes in a quorum: log n
- Handle process failures till log n
  - If any process fails, the algorithm substitutes for that process two possible paths starting from the process's two children and ending in leaf nodes.
  - Cannot form quorums when failure is more than log n



- Eight quorums when **process 3 fails**
  - {1, 2, 4, 8}
  - {1, 2, 4, 9}
  - {1, 2, 5, 10}
  - {1, 2, 5, 11}
  - {1, 6, 12, 7, 14}
  - {1, 6, 13, 7, 15}
  - {1, 6, 13, 7, 14}
  - {1, 6, 12, 7, 15}

Token-based Algorithms

Broad Idea

- A single token circulates in the system; a process enters the critical section when it has the token
- Obvious mutual exclusion – no more than one process can possess the token at any point in time
- Algorithms differ in how to find and get the token
  - Token circulates, processes use it when it passes through them
  - Token stays at the process that used it last, other processes request for it when needed – need to differentiate between old and current requests

Suzuki Kasami Algorithm

- Broadcast a request for the token
- Process with the token sends it to the requestor if it does not need it

- **Issues:**
  - Current vs Outdates requests
  - Determining processes with pending requests
  - Deciding which process to give the token to

Suzuki Kasami Algorithm

- The token (called the PRIVILEGE message in the algorithm):
  - Queue (FIFO) Q of requesting processes
  - LN[1...n]: Sequence number of requests that j executed most recently

- The request message:
  - REQUEST(i, k): Request message from process i for its k^{th} critical section execution

- Other data structures:
  - RN_i[1...n] for each node i, where RN_i[j] is the largest sequence number received so far by i in a REQUEST message from j

Suzuki Kasami Algorithm

- To request critical section:
  - If i does not have the token, increment $RN_i[i]$ and send REQUEST(i, $RN_i[i]$) to all processes
  - If i has the token already, enter critical section if the token is idle (no pending requests in token Q), else follow rule to release critical section

- On receiving REQUEST(i, sn) at j:
  - set $RN_j[i] = max(RN_j[i], sn)$
  - If j has the token and the token is idle, send it to i if $RN_j[i] = LN[i] + 1$. If token is not idle, follow rule to release critical section.

Suzuki Kasami Algorithm

- To enter critical section:
  - Enter Critical Section if the token is received after sending REQUEST

- To release critical section:
  - set $LN[i] = RN_i[i]$
  - For every process j which is not in Q (in token), add process j to Q if $RN_i[j] = LN[j] + 1$
  - If Q is non-empty after the above, delete the first process from Q and send the token to that process

Suzuki Kasami Algorithm - Analysis

- **Message complexity**: zero if node holds the token already and token is idle, *n* otherwise
- **Synchronization delay**: zero (node has the token) or max. message delay (token is elsewhere)
- No starvation

Happy Learning