Global System State Collection

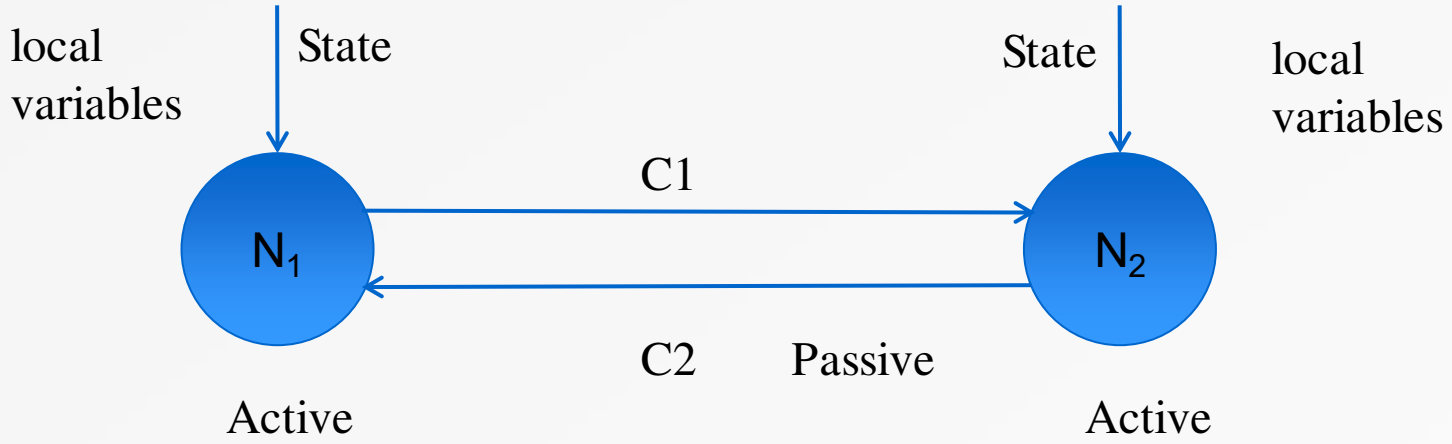
State in Distributed System

- For a centralized system, the log is kept on the secondary storage.
- In case of a Distributed System, the state comprises of
  - Local state of each process and messages that are sent by a node together with the messages that are currently in transit, that is, that have been sent but not delivered.

- The processes in a distributed system in general would have only a limited view of the system, which consists of its local state and the messages it has sent or received.

- Difficult to take the state at each node at the same time because of absence of a global clock and imperfect clock synchronization.

State in Distributed System

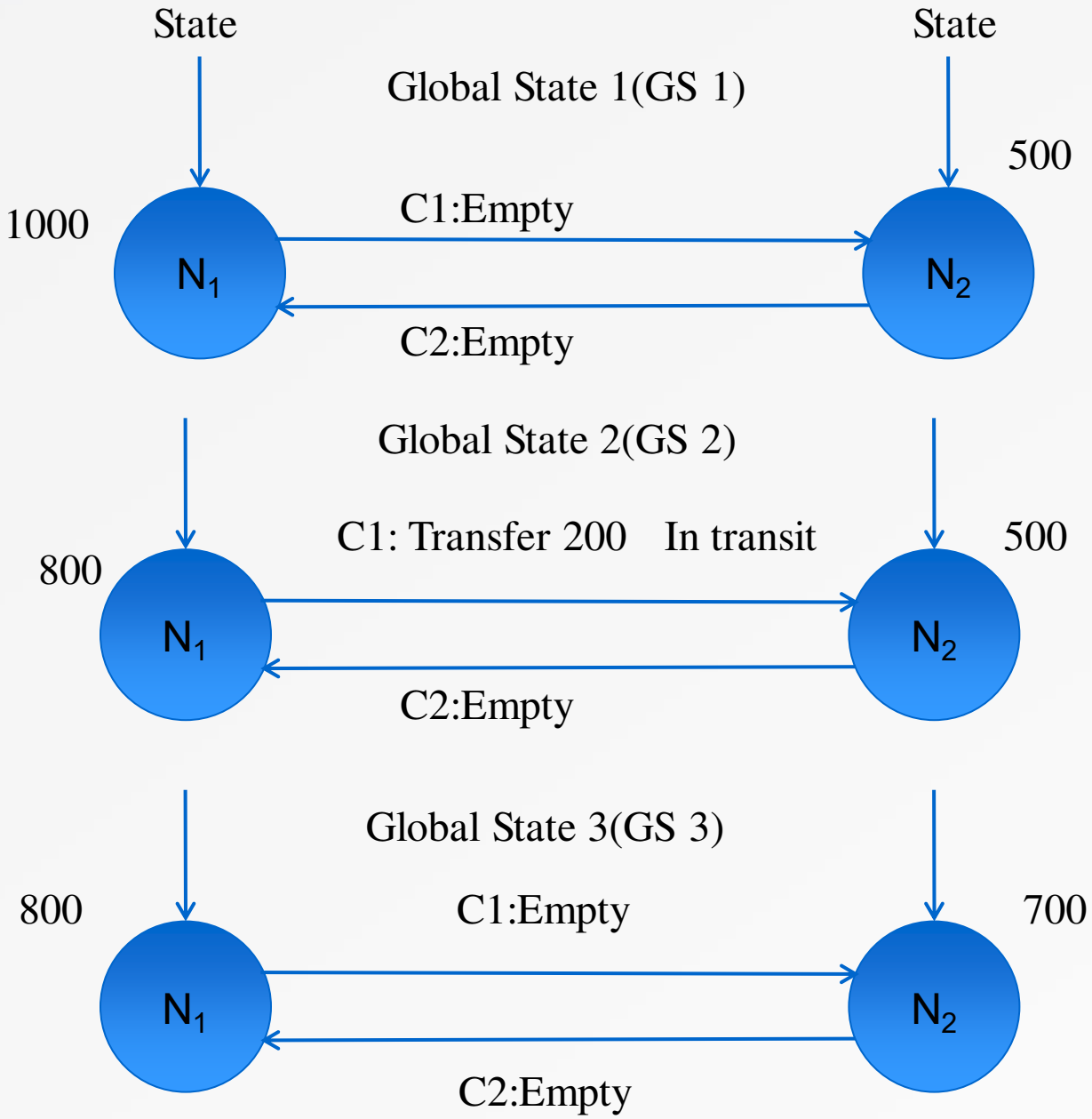


- States of N_1 and N_2 and the state of the channel itself.
- Some messages might be in transit.
- N_1 sends a message that is on the channel and has not been received by N_2.
- When to collect the state?
- Difficult to find the perfect global state as it requires the communication and computation

Why do we need to record/collect state??

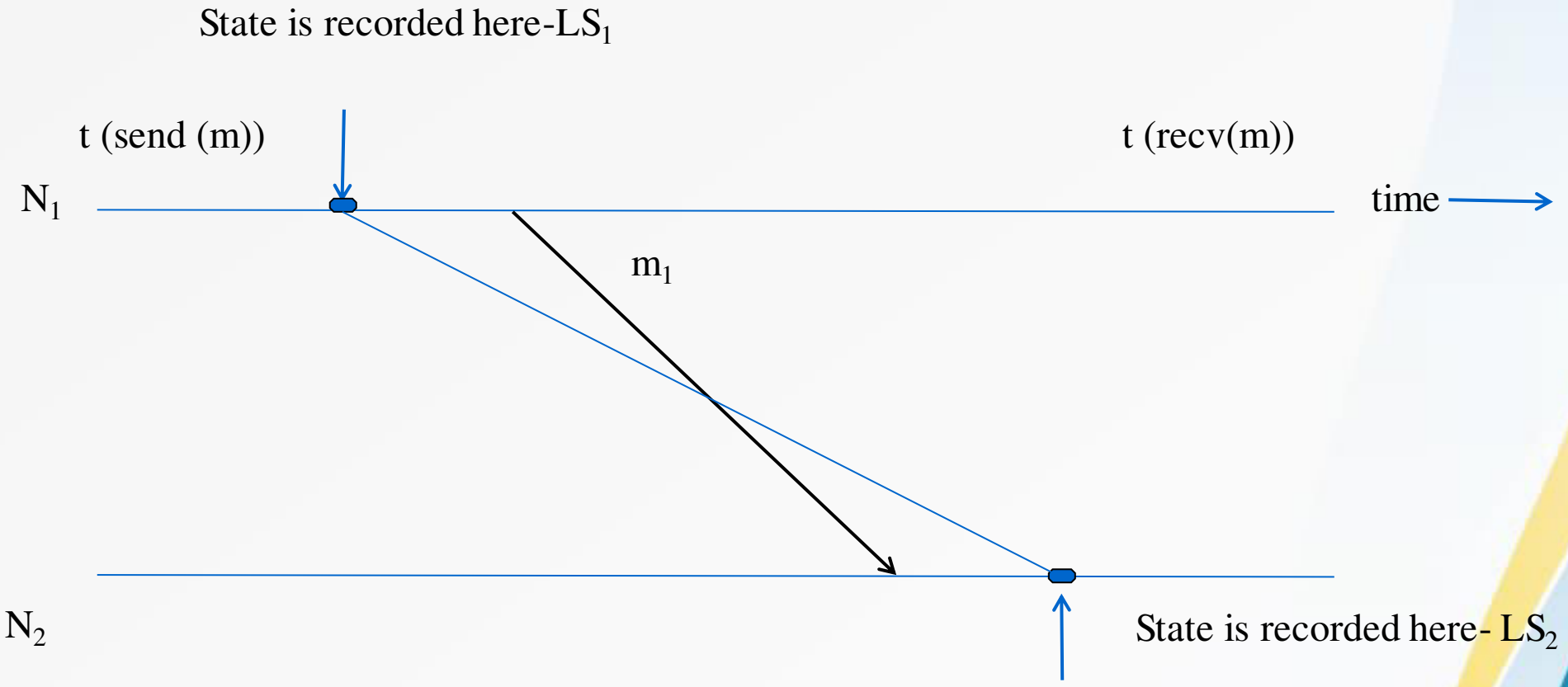
- Distributed Deadlock Detection.
- Distributed Debugging

State in Distributed System Cont..



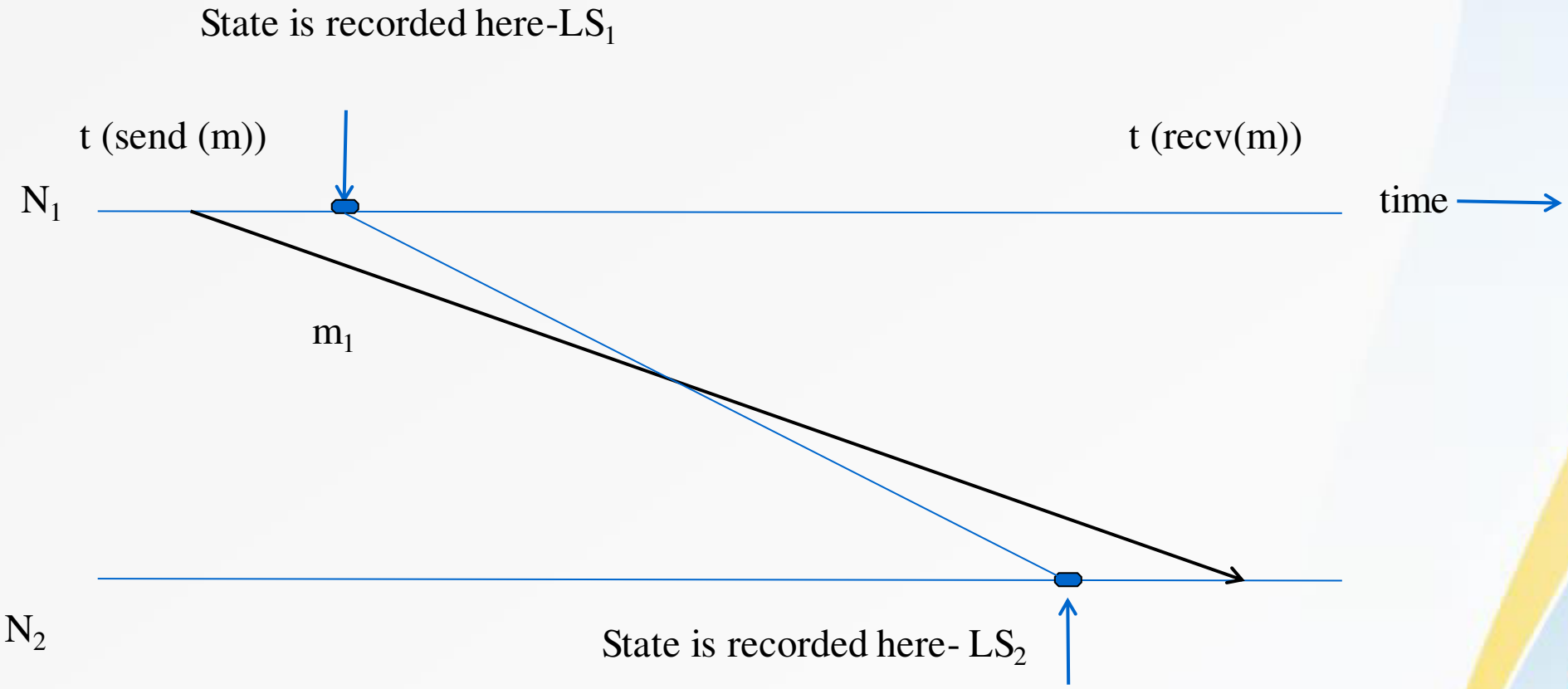
- Case 1
  - State collected at N_1 when the initial amount is Rs. 1000 (GS1)
  - N_1 sends Rs. 200 to N_2
  - Channel state collected (GS2), message (Sending of Rs. 200) is on the channel, i.e. in transit)
  - State collected at N_2 when the amount is Rs. 500 (GS1)
  - Total amount in the system is 1700.
- Case 2
  - Channel (C1: Empty) state collected at GS1.
  - State of N_1 (800) and N_2 (500) collected at GS2.
  - Total amount in the system is 1300.

Distributed Cut



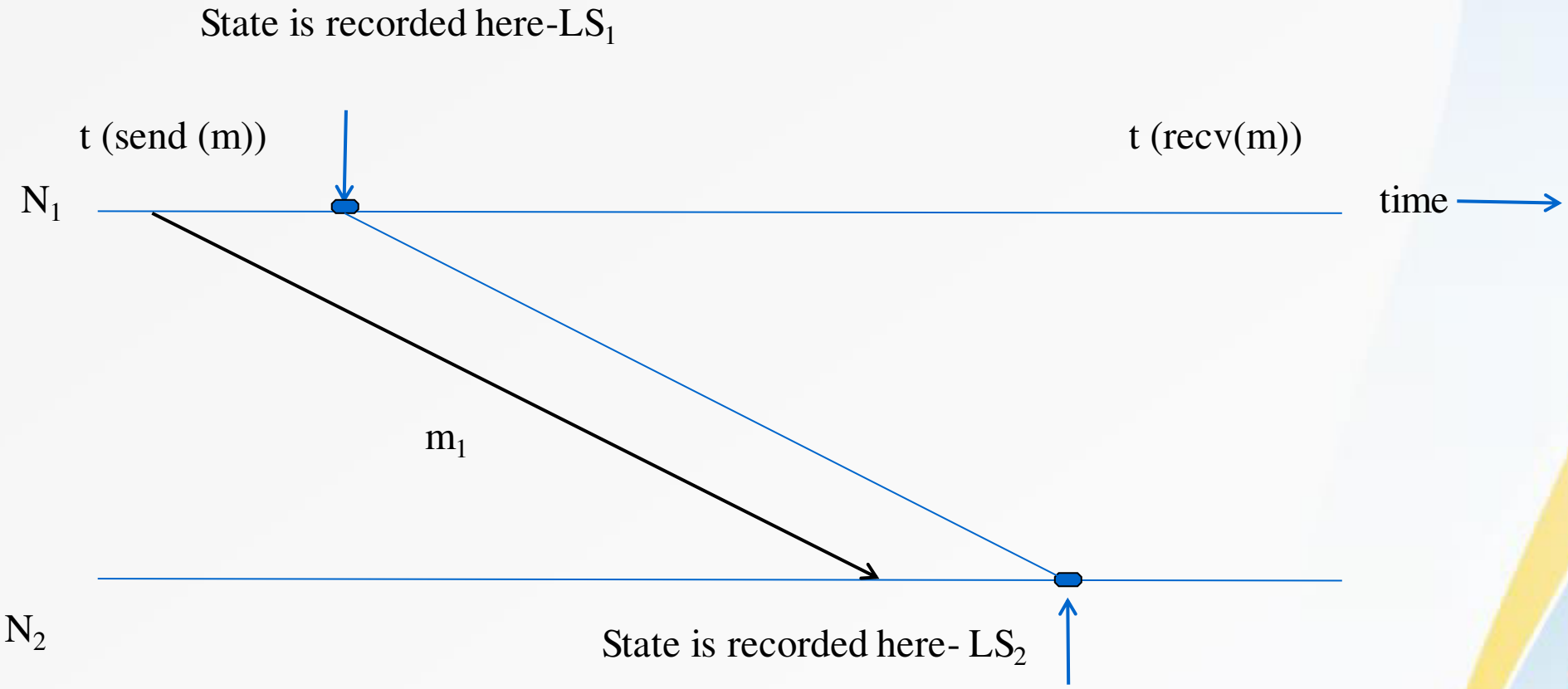
- N_2 receives the message and N_1 says it did not send.
- If the distributed cut (G) has received the message, then that message is not consistent and the state is also not consistent.

Distributed Cut Cont..



- m_1 is recorded in LS_1, m_1 is in transit.
- m_1 is not recorded in LS_2
- The state is called as consistent with m_1 in transit.

Distributed Cut Cont..



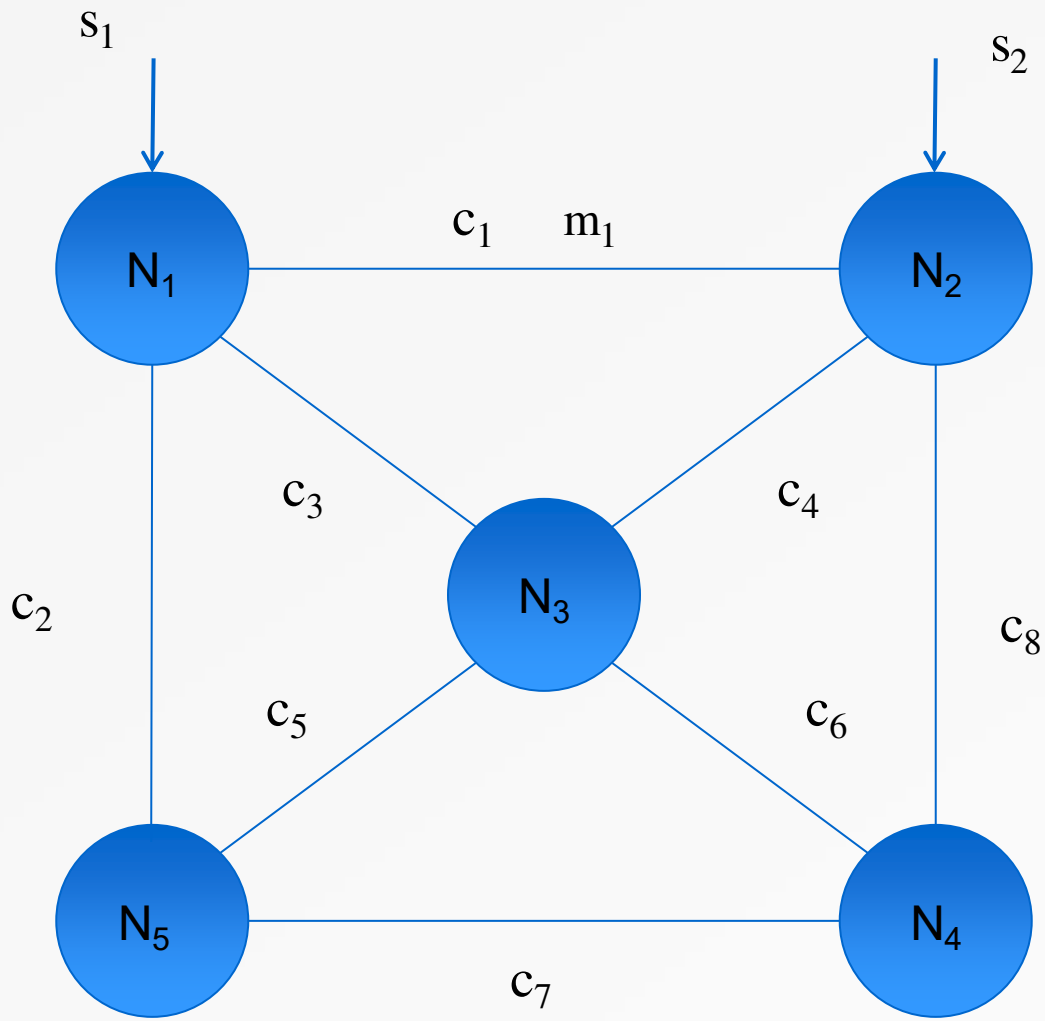
- The perfect situation.
- The state is called as consistent and transitless.

Chandy-Lamport Global State Collection Algorithm

- Assumptions
  - FIFO channels are required (TCP).
  - Connected topology.
  - Asynchronous system (No bound on message latency)
  - No failures
    - No node failures and no message failures
  - Record the states locally
  - Any process can begin the global state recording algorithm

Chandy-Lamport Global State Collection Algorithm

- M - Marker or Control Message
- M does not intefere with a data message.
- C - Channel



- Provides the consistent global state but does not guarantee actual state.

Protocol

1. When a node N decides to initialize global state collection, it performs the marker sending rule mentioned below.

   Marker sending rule for N

a). N records its state

b). For each outgoing channel (C) incident on N, N sends marker msg before it sends any further computation msg.

Protocol Cont..

2. When a node N' receives marker msg on channel C,
   it performs the following actions.

a). If N' has not recorded its state, then
   i). N' records the state of C as empty.
   ii). N' follows the marker sending rule.

b). Else
   i). (Marker indicates that the channel state must be recorded)
      N' records state of C as all those computation msgs
      that it had received after it recorded its state and before
      it received the marker msg.

3. N' is finished when it has received a marker along each of its incoming channels.

More details cont..

- When we say the algorithm is over?

- If a process N_j receives the marker requesting for recording the state for the first time, it considers the process that sent the marker as its predecessor.

- When N_j completes its part of the state recording, it sends its predecessor a DONE message.

- When the initiator receives a DONE message from all its successors, it knows that the state has been completely recorded.

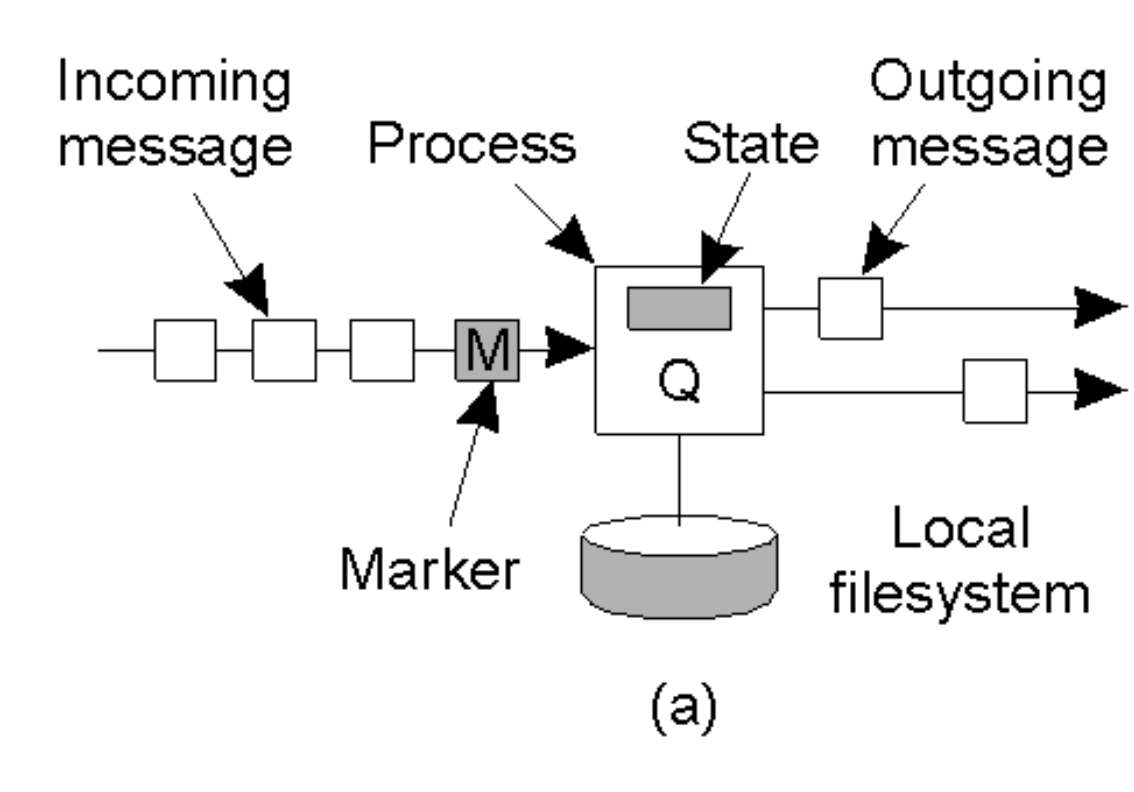
More details

- The network is considered as a bidirectional graph.
- Marker msg goes from N_1 to N_2 (outgoing) and N_2 to N_1 (incoming)
- Contents of Marker message?
  - <Initiator id, Sequence Number>

State collection activity

- The algorithm can be initiated by any node at any time and may have multiple initiations going concurrently.
- The consistent state that a node gets may be different for every node.

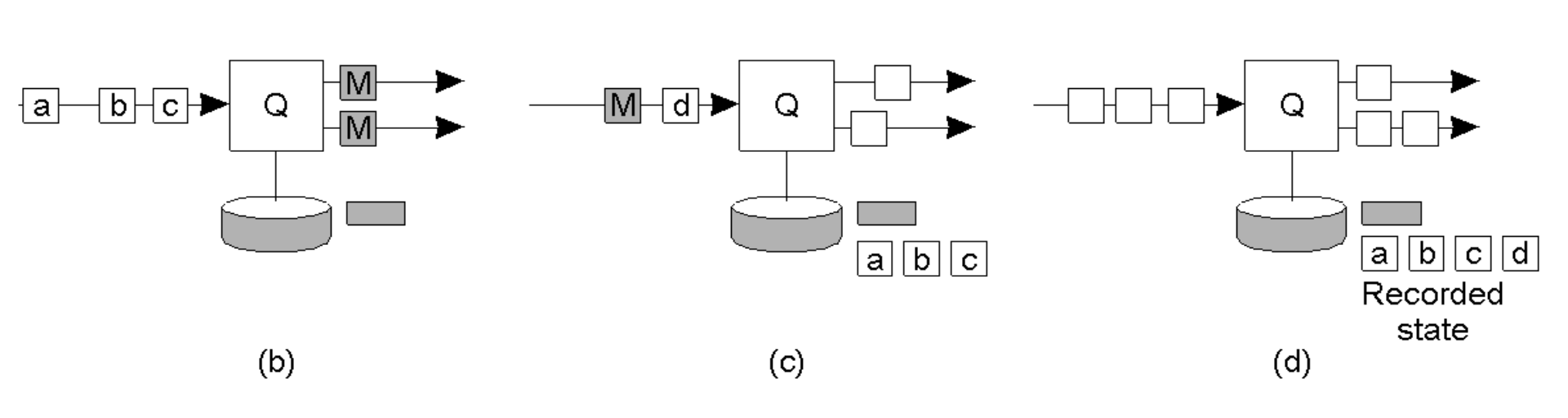
Algorithm



a) Organization of a process and channels for a distributed snapshot

Reference: Western Science- Computer Science (https://www.csd.uwo.ca/)

Algorithm



b) Process Q receives a marker for the first time and records its local state

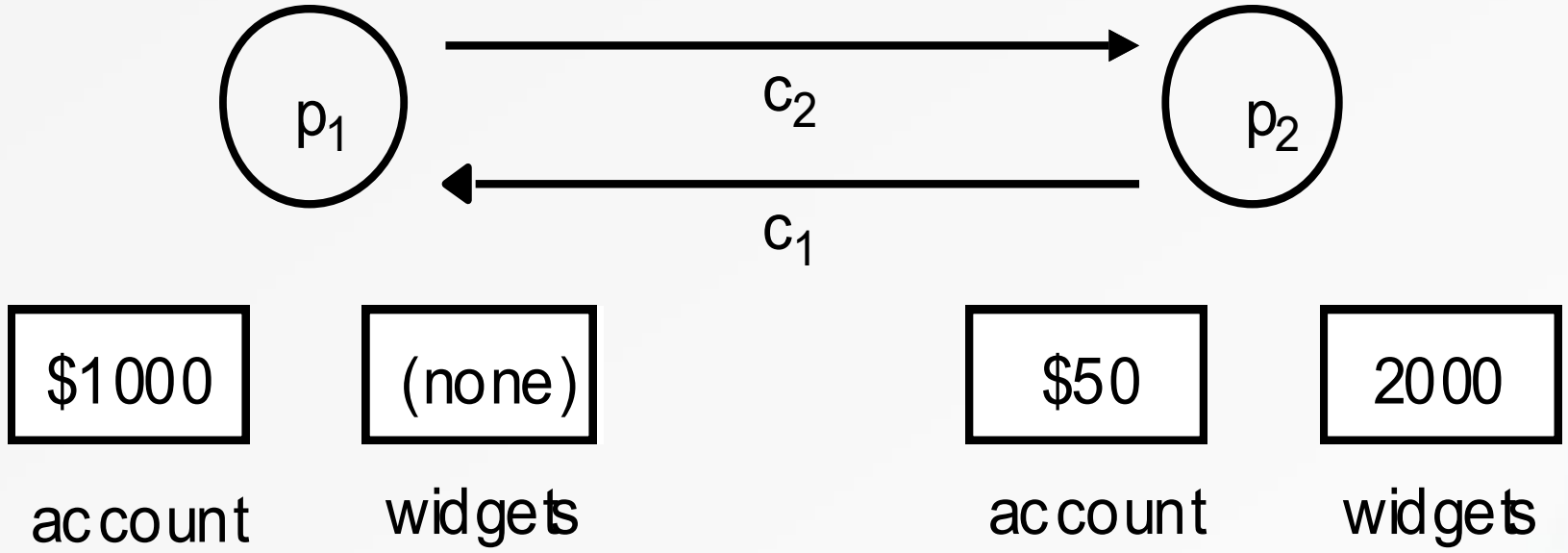
c) Q records all incoming messages

d) *Q* receives a marker for its incoming channel and finishes recording the state of the incoming channel

Example

- Assume two process p_1 and p_2 connected by two unidirectional channels, c_1 and c_2.
- The two processes trade in widgets.
- Process p_1 sends orders for widgets over c_2 to p_2, enclosing payment at the rate of $10 per widget.
- Process p_2 sends widgets along channel c_1 to p_1.
- The processes have the initial states shown in the figure on the next slide. It is assumed that p_2 has already received an order for 5 widgets, which it will shortly dispatch to p_1.

Example

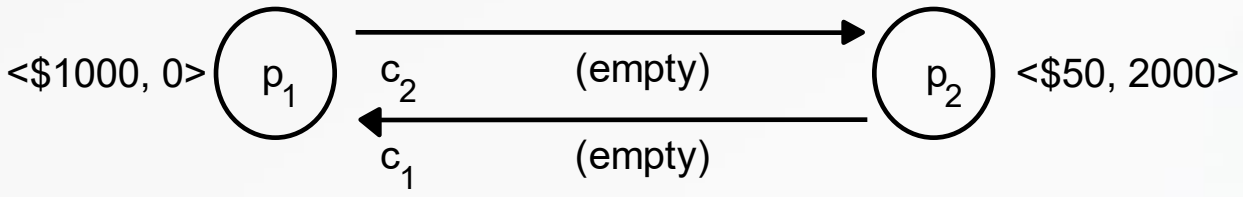


This is the initial state when the algorithm was initiated;
Before initiation, the process p_2 had received an order for 5 widgets;
Thus account has $50

Example

- Upon request for global snapshot, process p_1 records its state in the actual global state, S_0, when p_1's state (initial) is <$1000,0>.

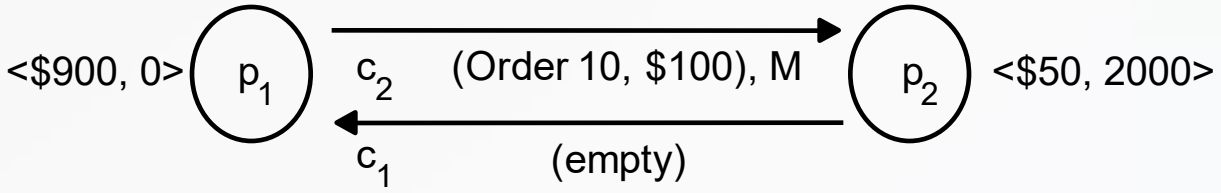
1. Actual Global State S_0



Example

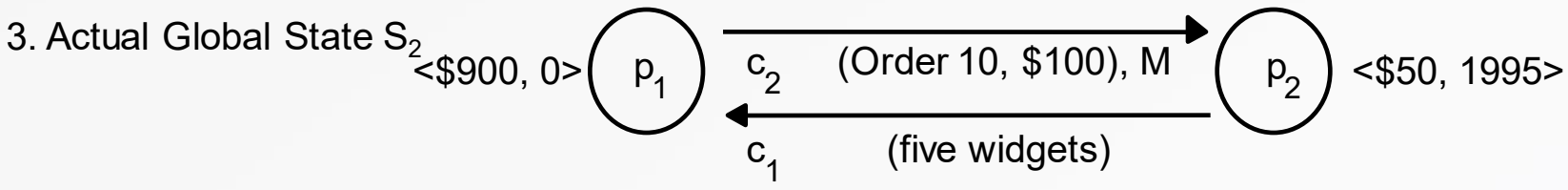
- Process p_1 then emits a marker message over its outgoing channel c_2 before it sends the next application-level message: (Order 10, $100) over channel c_2.
- The system enters actual global state S_1.

2. Actual Global state S_1



Example

- Before p_2 receives the marker, it emits an application message (5 widgets) over c_1 in response to p_1's previous order, yielding a new actual global state S_2.

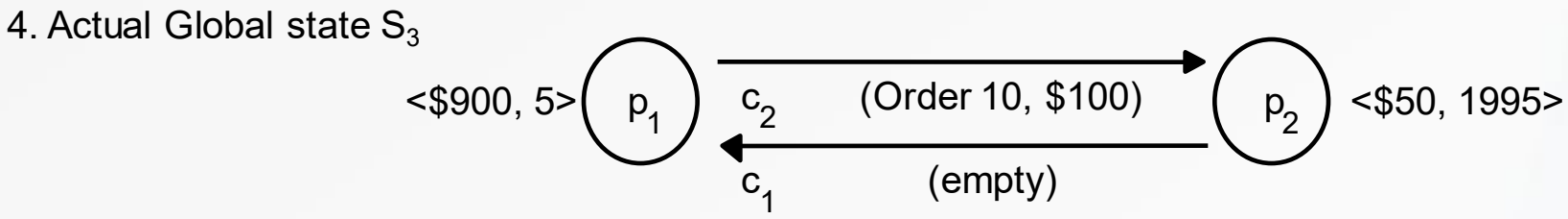


Example

- Now process p_1 receives p_2's message (5 widgets) and p_2 receives the marker.
- p_2 records the state of channel c_2 as the empty sequence and follows the marker sending rule.
- p_2 records its state as <$50,1995>
- p_2 sends a marker message over c_1.
- When process p_1 receives p_2's marker message, it records the state of c_1 which is the single message (5 widgets) that it received after it first recorded its state.

Example

- Final recorded state



Example

- The sequence of actual global states:
  - S_0: p_1: <$1000,0>; p_2:<$50,2000>;c_1:<>; c_2:<>
  - S_1: p_1: <$900,0>; p_2:<$50,2000>; c_1:<>; c_2:<(order 10, $100)>;
  - S_2: p_1: <$900,0>; p_2:<$50,1995>; c_1:<five widgets>; c_2:<(order 10, $100)>;
  - S_3: p_1: <$900,5>; p_2:<$50,1995>; c_1:<empty>;c_2:<(order 10, $100)>;

Termination Detection

Introduction

- Distributed Computation:
- A set of cooperating processes that communicate with each other by exchanging messages.

- In case of a distributed computation, it is necessary to know when the computation has terminated.

- Many situations where it is desirable to know about the termination detection (TD), e.g., leader election, a deadlock detection, token generating algorithm, etc.

- TD is an example of consistent global state of a distributed system.

Introduction Cont..

- Problem: To determine that the distributed computation is indeed terminated.
- Difficult (in fact, non-trivial) problem as every process has a local knowledge and they do not have the knowledge of a global state.
- The distributed computation is said to be terminated if every process terminates that computation locally and there is no message in transit between any two processes.
- Local termination: a state where the process has finished its computation and the process will not restart it's operation unless it receives a message.
- A process (or processes) must have some way to decide when the underlying computation has terminated.

Introduction Cont..

- Computation messages :
  - Messages used in the underlying computation.
- Control messages:
  - Messages used for the purpose of detecting the termination.

- A termination detection (TD) algorithm requirements
  - The execution of TD cannot defer the underlying computation indefinitely.

  - No additional communication channels between processes.

System Model

- A process needs to be either active or passive.
- Active state:
  - In this state, process is doing local computation.
- Passive state:
  - The process has completed the execution of its local computation and it will become active again only when it receives a message from another process.
- An active process can become passive at any time.
- A passive process becomes active only when it receives a message from another process.
- Only active processes can send messages.
- The process can receive a message when it is in active or passive state.

Termination Detection

- A distributed computation is said to be terminated iff:
- All processes have become passive.
- No message in transit in any channel.

TD by Weight Throwing (Huang)

- System Model
- One of the cooperating processes monitors the computation and is called as controlling agent.
- There will be a communication channel between each of the cooperating processes and the controlling agent.
- A communication channel between every pair of processes.
- All processes are in the idle state at the start of algorithm.

Termination detection by Weight Throwing

- The weight at each process is 0 and the weight at the controlling agent is 1.
- The computation will start when the controlling agent sends a “computation” message to one of the processes.
- A non-zero weight W ($0 < W \leq 1$) is assigned to each process in the active state and to each message in transit.

Basic Idea

- Whenever a process has a message to send, the process's weight is split between itself and the process receiving the message (the message carries the weight for the receiving process).
- A process receiving a message adds the weight received in the message to it's weight.
- The weights with the processes added with the weights on the messages in transit is always 1.
- After becoming passive, a process sends its weight to the controlling agent in a control message and the controlling agent adds the weight to its weight.
- The controlling agent concludes termination if its weight becomes 1.

Notations

- The weight on the controlling agent and a process is in general is represented by W.
- B(DW) - a computation message B sent as a part of the computation, where DW is the weight assigned to it.
- C(DW) - a control message C sent from the processes to the controlling agent where DW is the weight assigned to it.

Algorithm

- Rule 1:
  - The controlling agent or an active process may send a computation message to one of the processes, say P, by splitting its weight W into W1 and W2 such that W1+W2=W, W1>0 and W2>0.
  - It then assigns its weight W:=W1 and sends a computation message B(DW:=W2) to P.

- Rule 2:
  - On the receipt of the message B(DW), process P adds DW to its weight W:

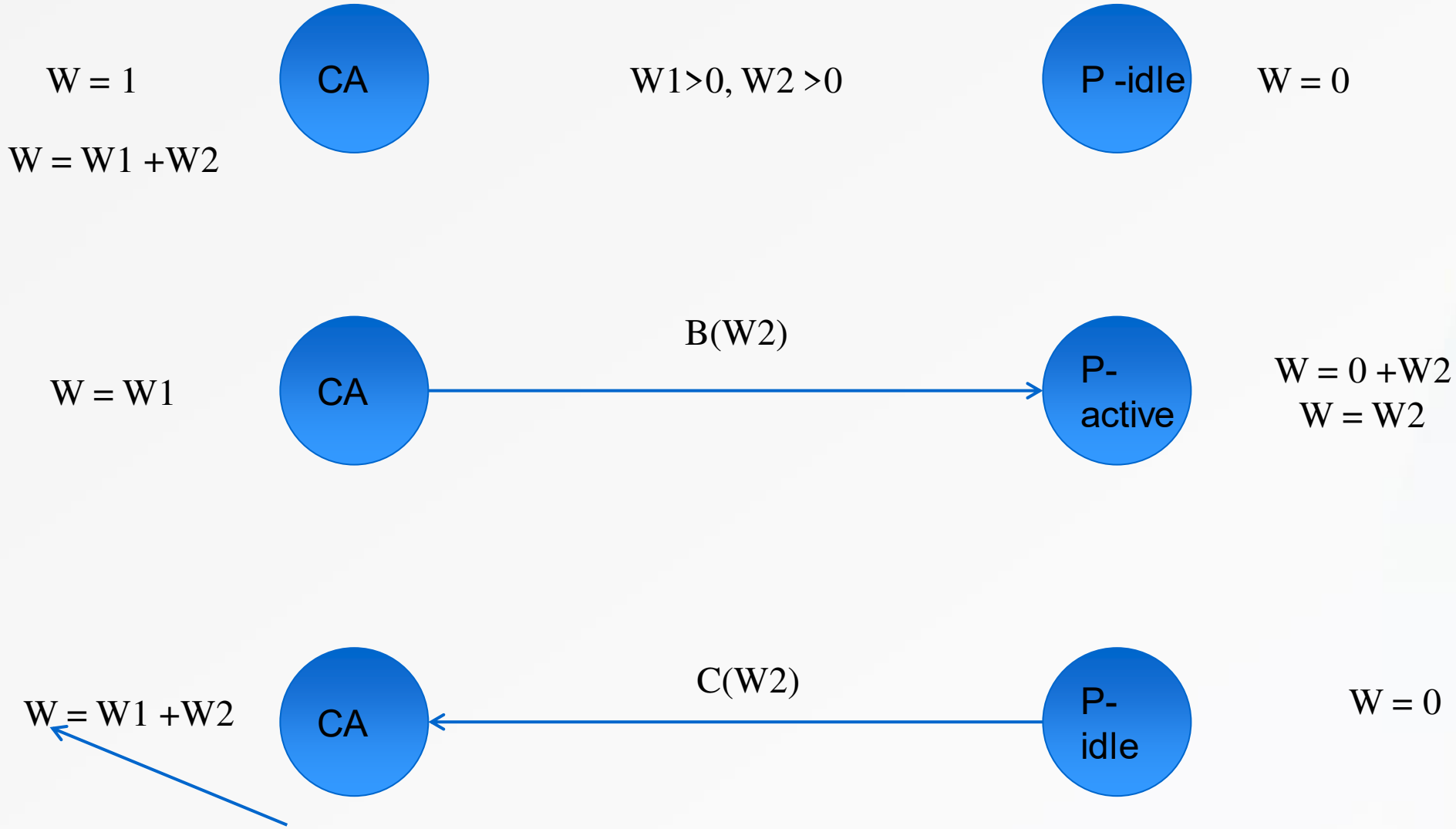
    W:=W+DW.
  - If the receiving process is in the idle state, it becomes active.

Algorithm Cont..

- Rule 3:
- A process switches from the active state to the idle state at any time by sending a control message C(DW:=W) to the controlling agent and making its weight W:=0.
- The process becomes passive.

- Rule 4:
- On the receipt of a message C(DW), the controlling agent adds DW to its weight (W:=W+DW).
- If W=1, then it concludes that the computation has terminated.

Example



If W becomes 1 at the end, TD is said to be done

Correctness of the Algorithm

Notations

- A: set of weights on all active processes
- B: set of weights on all basic messages in transit
- C: set of weights on all control messages in transit
- W_c: weight on the controlling agent.
- Two invariants I_1 and I_2 are defined for the algorithm:
- I_1: $W_c + \sum_{W \in (A \cup B \cup C)} W = 1$
- I_2: $\forall\ W \in (A \cup B \cup C),\ W > 0$

Correctness of Algorithm

- Invariant I1 states that sum of weights at the controlling process, at all active processes, on all basic messages in transit, and on all control messages in transit is always equal to 1.

- Invariant I2 states that weight at each active process, on each basic message in transit, and on each control message in transit is non-zero.

Correctness of Algorithm

$W_c = 1$

$\Rightarrow \sum_{W \in (A \cup B \cup C)} W = 0$ (by I_1)

$\Rightarrow (A \cup B \cup C) = \phi$ (by I_2)

$\Rightarrow (A \cup B) = \phi$.

$(A \cup B) = \phi$ implies the computation has terminated. Therefore, the algorithm never detects a false termination.

Further,

$(A \cup B) = \phi$

$\Rightarrow W_c + \sum_{W \in C} W = 1$ (by I_1)

Leader Election

Introduction

- Given a set of nodes, one of them is to be set as a leader.
- So many situations, where we need to elect a leader in the distributed system.
- A centralize control simplifies process synchronization
- It is a single point failure, and can limit the service.
- If existing leader fails, a new leader must be chosen
- Many algorithms for leader election, one of them can be selected based on specific requirements.
- Examples
  - Berkley's algorithm for physical synchronization
  - Controlling Agent in Termination Detection
  - Centralized Mutual Exclusion Algorithm

Introduction Cont..

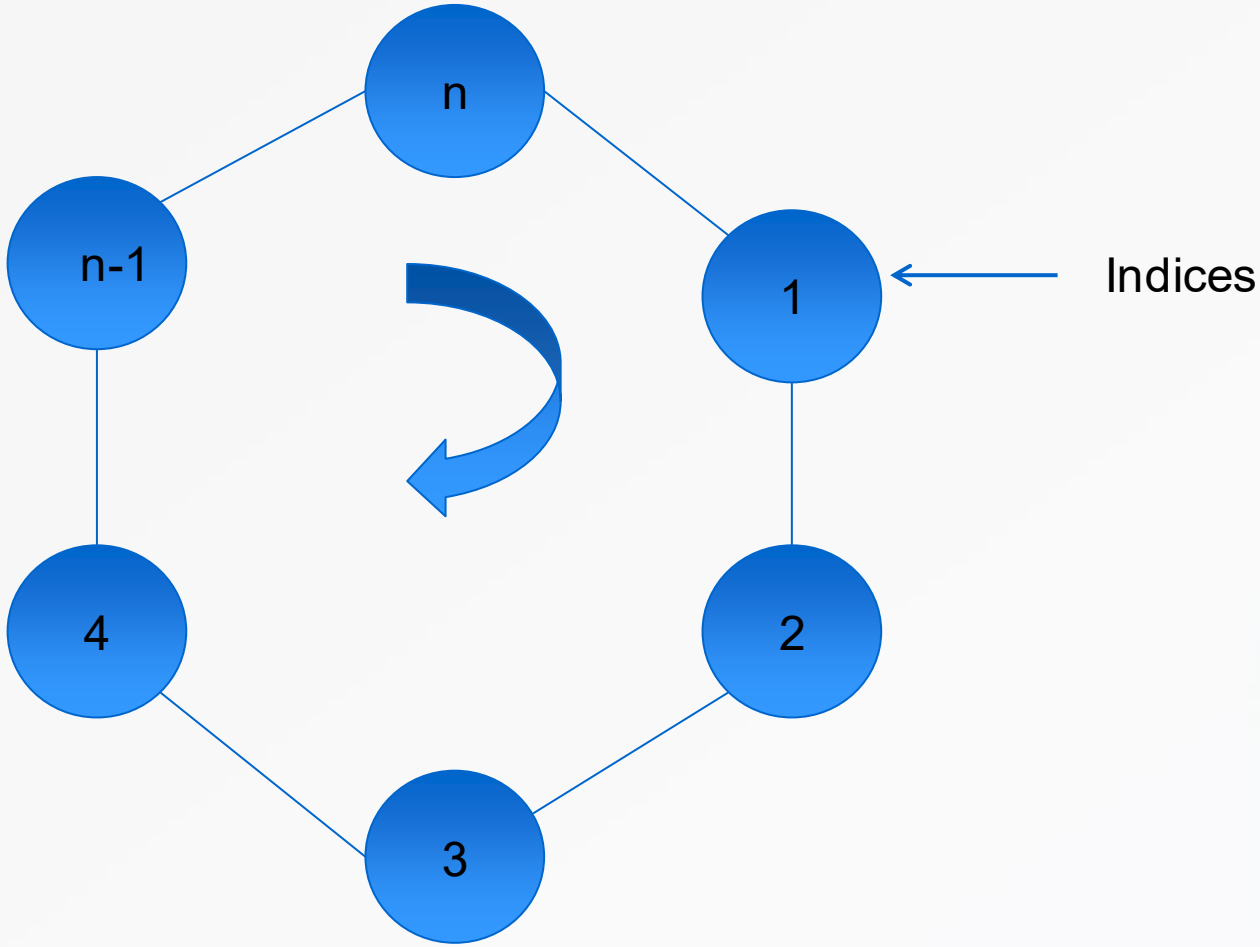
- When a leader is to be elected?

- The system initiation
- When the existing leader fails

- May have a timeout period wherein if a timeout period ends, then some node will initiate the election.

Lelann-Chang-Robert's Algorithm

- Logical Ring of Processes
- It uses unidirectional communication
- Processor nodes with numbers as n = 0, 1, 2, ..., n-1



Assumptions

- Every node knows its neighbour node.
- Every node has a unique node.
- Nodes are placed in a clockwise unidirectional ring (logical).

Algorithm

1. One or more processes P_i's can start an election, by sending an election message containing its own identifier (id) to P_i+1.

2. When a process P_i by itself or upon the receipt of a message goes in an election, it marks itself as a participant.

3. If P_i receiving an election message has a greater id and is not already a participant, then it sends an election message with its own id to P_i +1.

4. If its own id is smaller, it forwards the message with the “id” it has received.

5. If it receives a message with its own “id” then it declares itself as the leader.

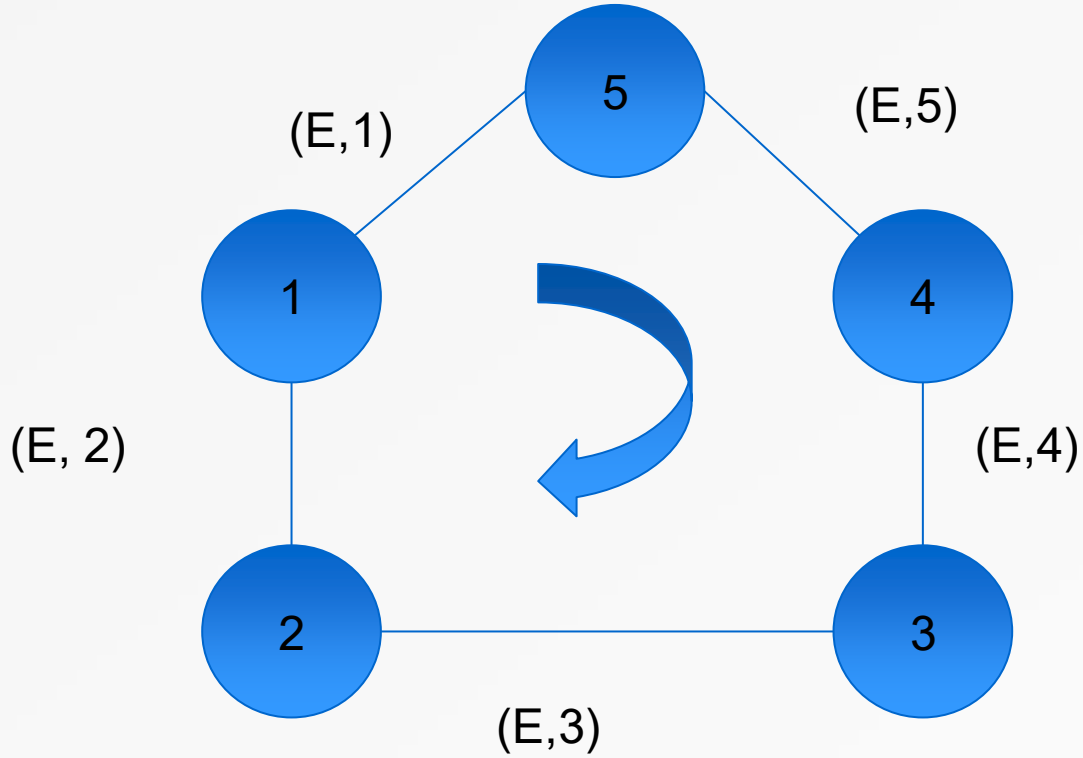
6. The leader node then sends the (Leader, id) to all the processes in the ring.

Algorithm details

- The message having the largest identity completes the round trip and returns to its originator and becomes the leader.

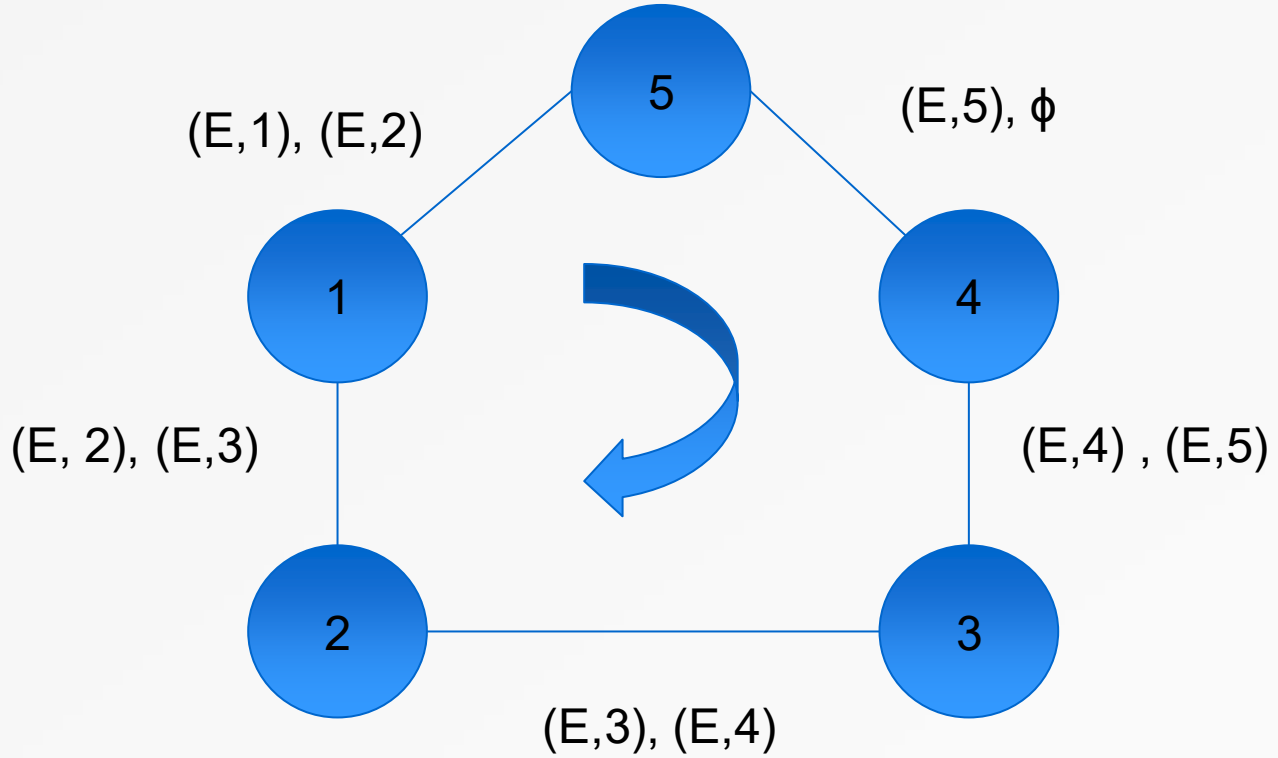
- The leader must announce itself to all other processes through the leader messages, so that termination of the algorithm is guaranteed and everybody knows who the leader is.

Example



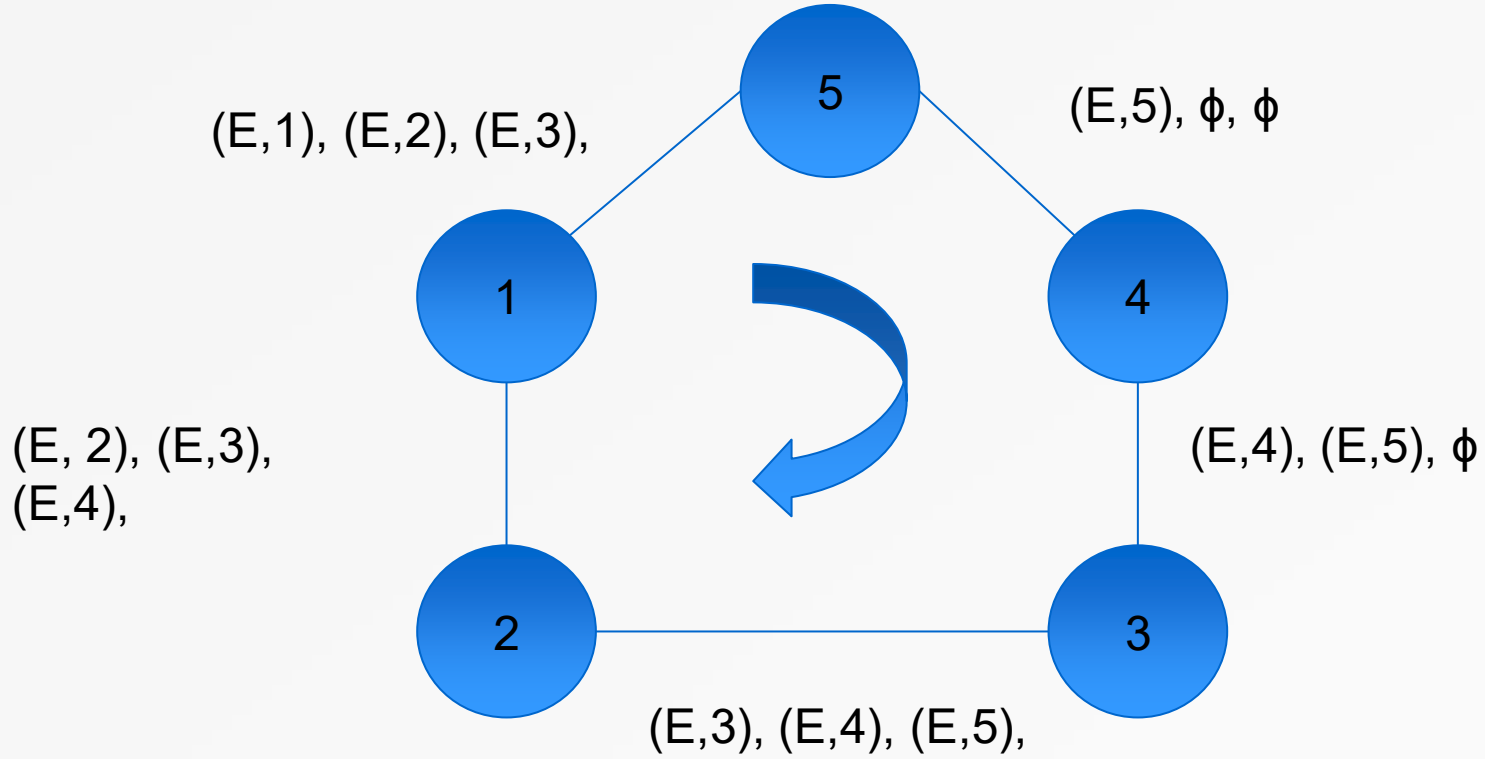
Node	Msg
1	1 time
2	1 time
3	1 time
4	1 time
5	1 time
Total	5 times

Example



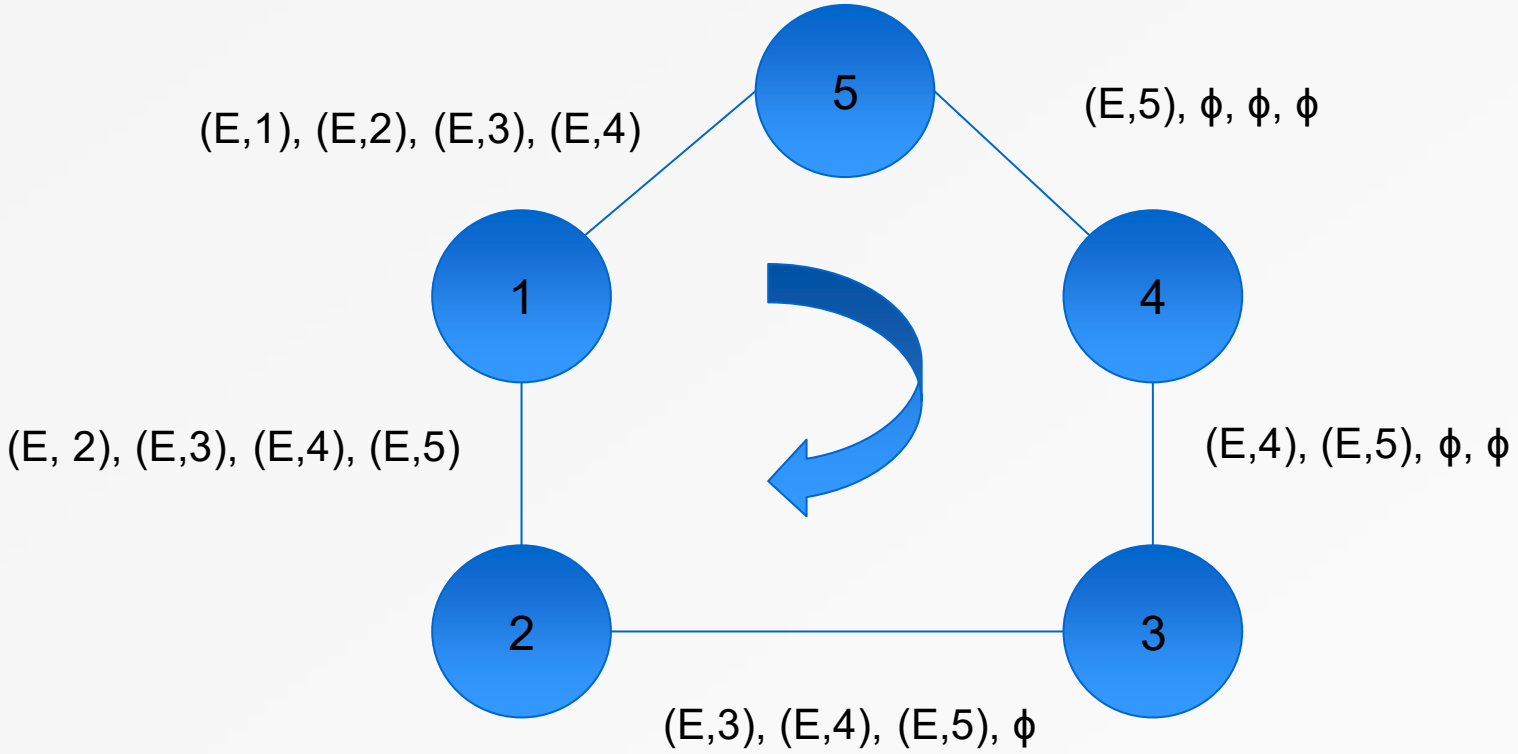
Node	Msg
1	2 times
2	2 times
3	2 times
4	2 times
5	1 time
Total	5 times

Example



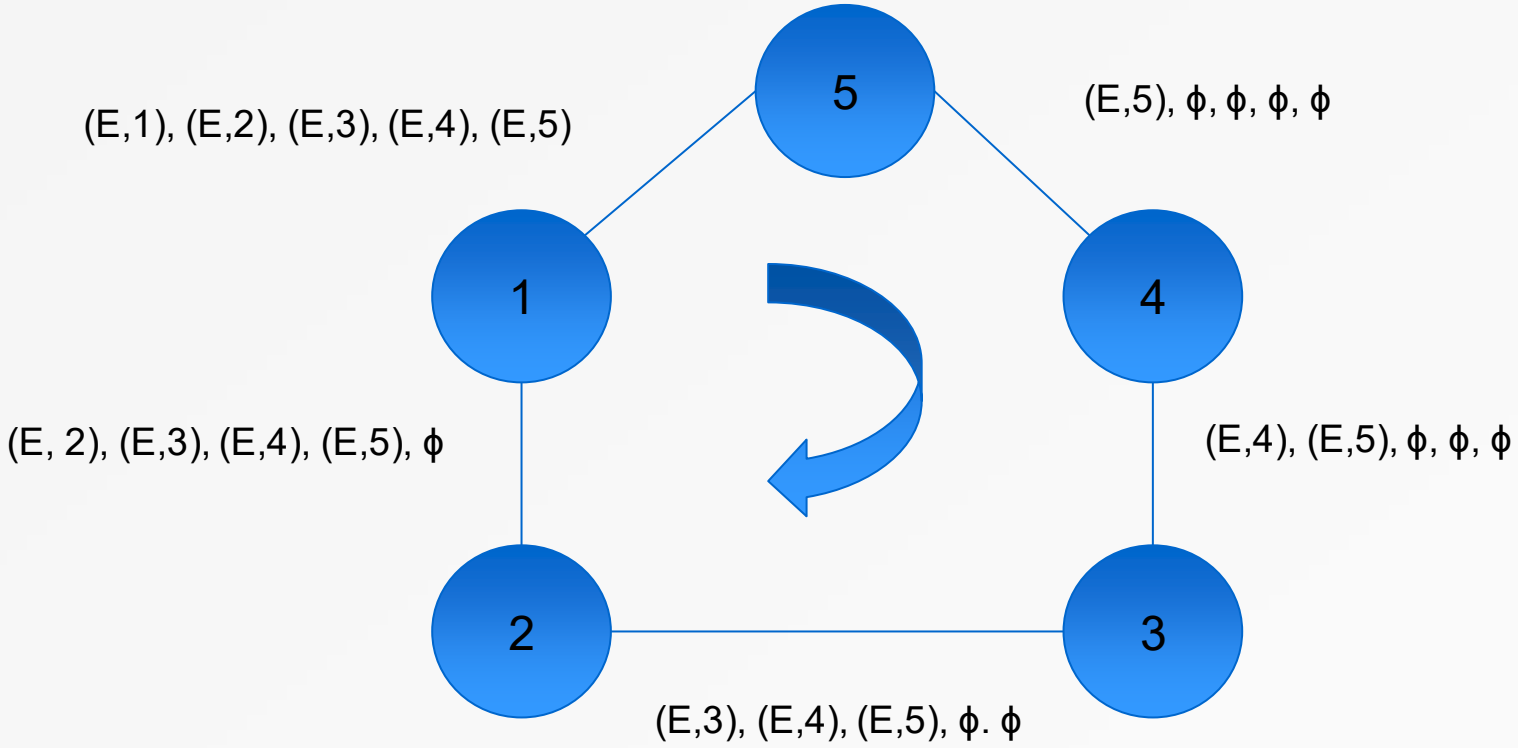
Node	Msg
1	3 times
2	3 times
3	3 times
4	2 times
5	1 time
Total	12 times

Example



Node	Msg
1	4 times
2	4 times
3	3 times
4	2 times
5	1 time
Total	14 times

Example



Node	Msg
1	5 times
2	4 times
3	3 times
4	2 times
5	1 time
Total	15 times

Message Complexity

- The largest id always travels all around the ring (n msgs).
- 2nd largest id travels until reaching the largest.
- 3rd largest id travels until reaching largest or second largest, and so on
- 2nd largest causes n-1 messages
- 3rd largest causes n-2 messages
- Number of msgs = $(n + (n - 1) + \ldots + 1) + n = n(n+1)/2 + n$ (including the n leader messages at the end).
- Hence, the worst case complexity is $O(n^2)$.

Leader Election-II

Le Lelann-Chang-Robert's (LCR) Algorithm

- Logical ring of Processes
- It uses unidirectional communication
- Processor nodes with numbers as n = 0, 1, 2, ...,n-1
- Asynchronous systems
- The worst case complexity is $O(n^2)$

Franklin's Algorithm

- Complexity reduction is done by an algorithm proposed by Franklin.
- The algorithm is ring-based and the communication is assumed to be bidirectional.
- Nodes who initiate the election are called as active nodes and remaining nodes are called as passive nodes

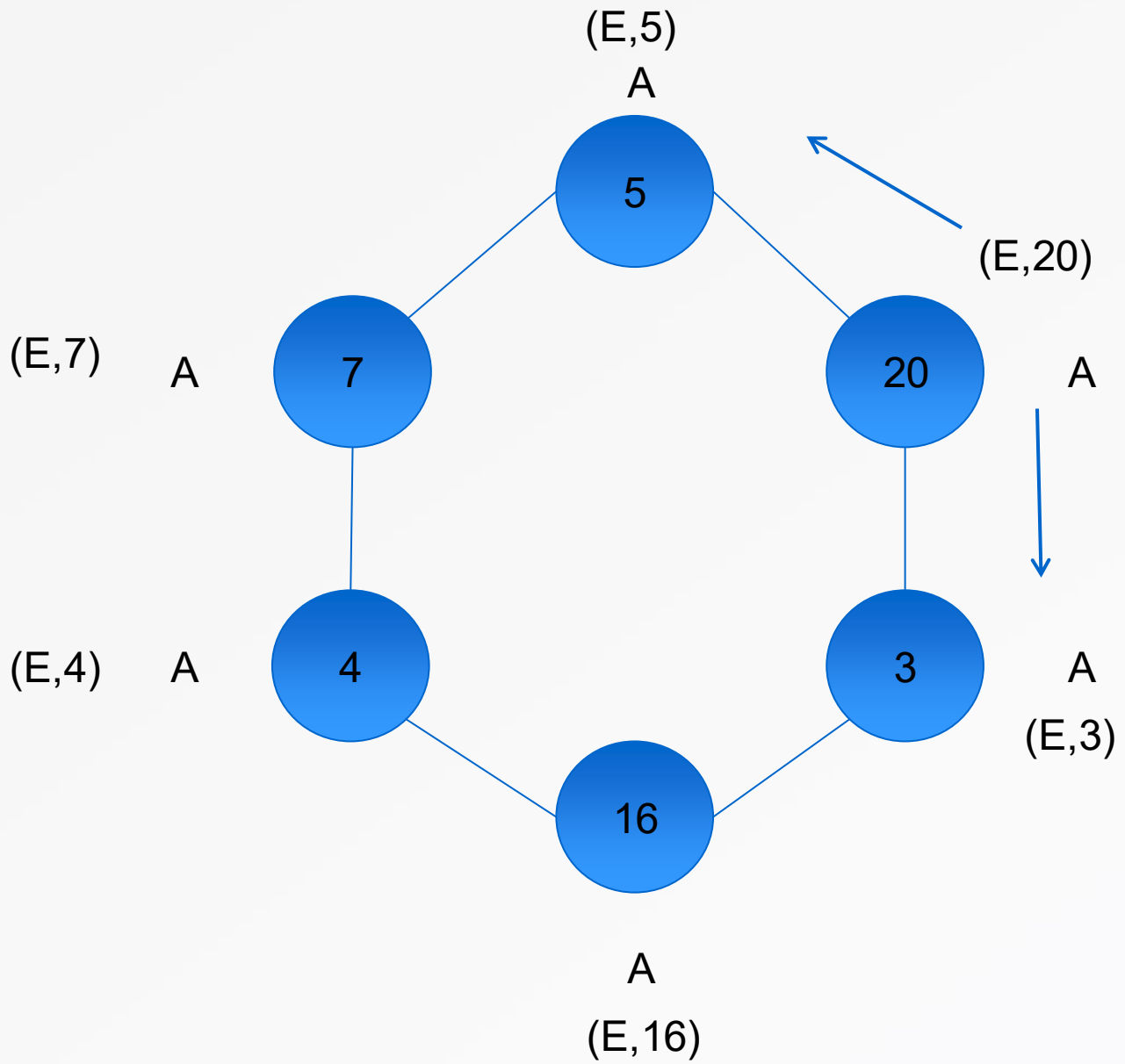
Rules of the algorithm

- Every node will initiate the election in both directions.
- When an active node N_i receives the an Election message from nodes N_j and N_k, then it checks id of both and compares it's own id with N_j and N_k.
- If it's id is less than N_j and N_k, then it becomes passive and it does not initiate the election.
- If it's id is greater than N_j and N_k, then it sends id again and re-initiate the election.
- When a node with highest id gets a message from both the directions, it concludes that it is a leader and sends a leader message to everyone.

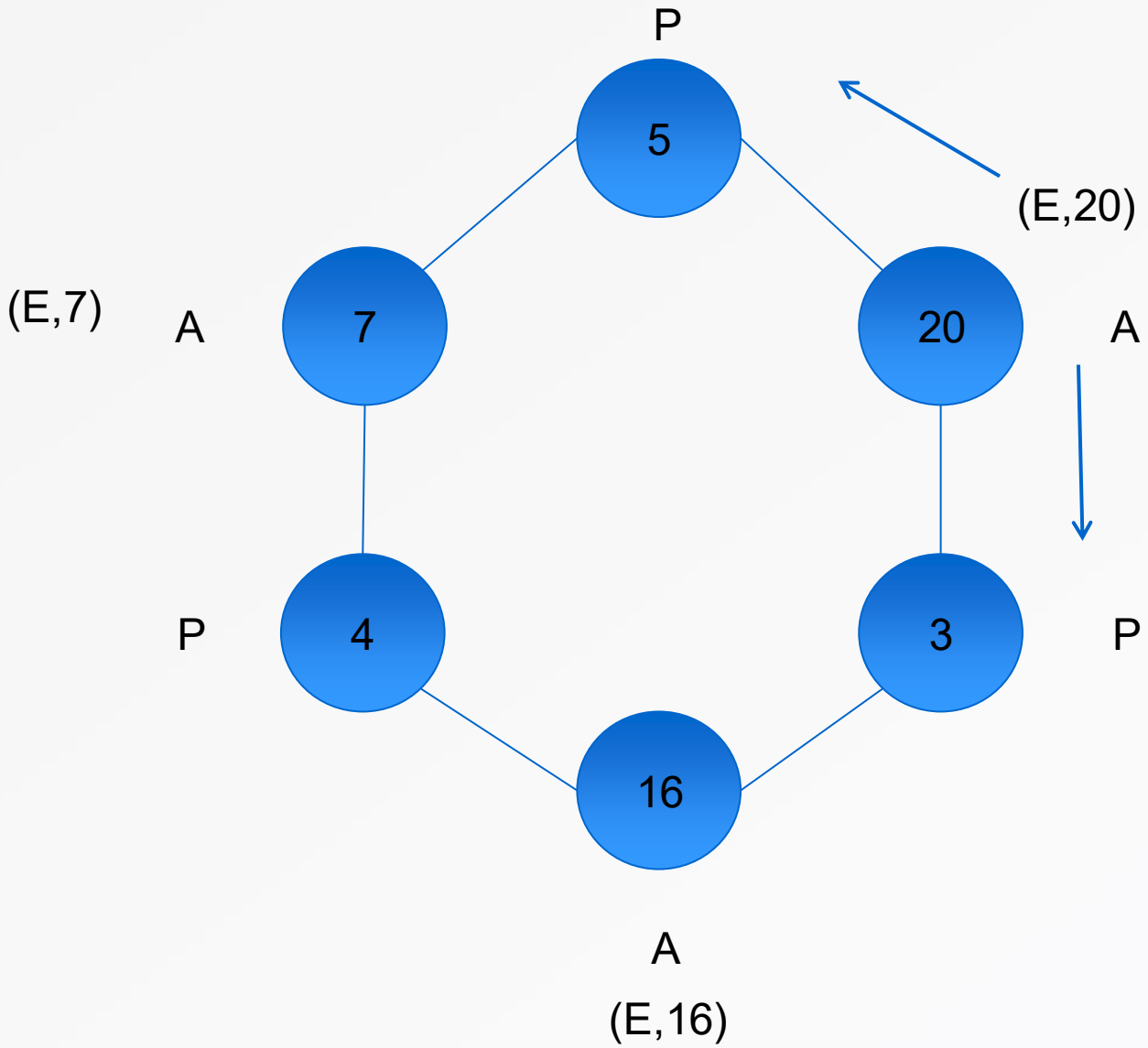
Rules of the algorithm

- Initially, initiators are active, and non-initiators are passive. Each active process sends its id to its neighbors on either side.
- Let active process p receive q and r
  - if max{q, r} < p, then p sends its id again.
  - if max{q, r} > p, then p becomes passive.
  - if max{q, r} = p, then p becomes the leader.
- Passive processes pass on incoming messages.

Example



Example



Complexity

- Worst-case message complexity: O(N logN).
- In each round, at least half of the active processes become passive. So there are at most $\log_2 N$ rounds.
- Each round takes 2N messages.

Detecting Failures in Leader Election

- Timeouts are used to detect failures
  - $T = 2T_{trans} + T_{process}$
    - Where T_{tran} is maximum transmission delay and $T_{process}$ represents the maximum delay for processing a message.

- If a process fails to respond to a message request within T seconds then an election is initiated.

Bully Algorithm

- When a process, P, notices that the coordinator is no longer responding to requests, it initiates an election.
  - P sends an ELECTION message to all processes with higher numbers.
  - If no one responds, P wins the election and becomes a coordinator.
  - If one of the higher-ups answers, it takes over. P’s job is done.

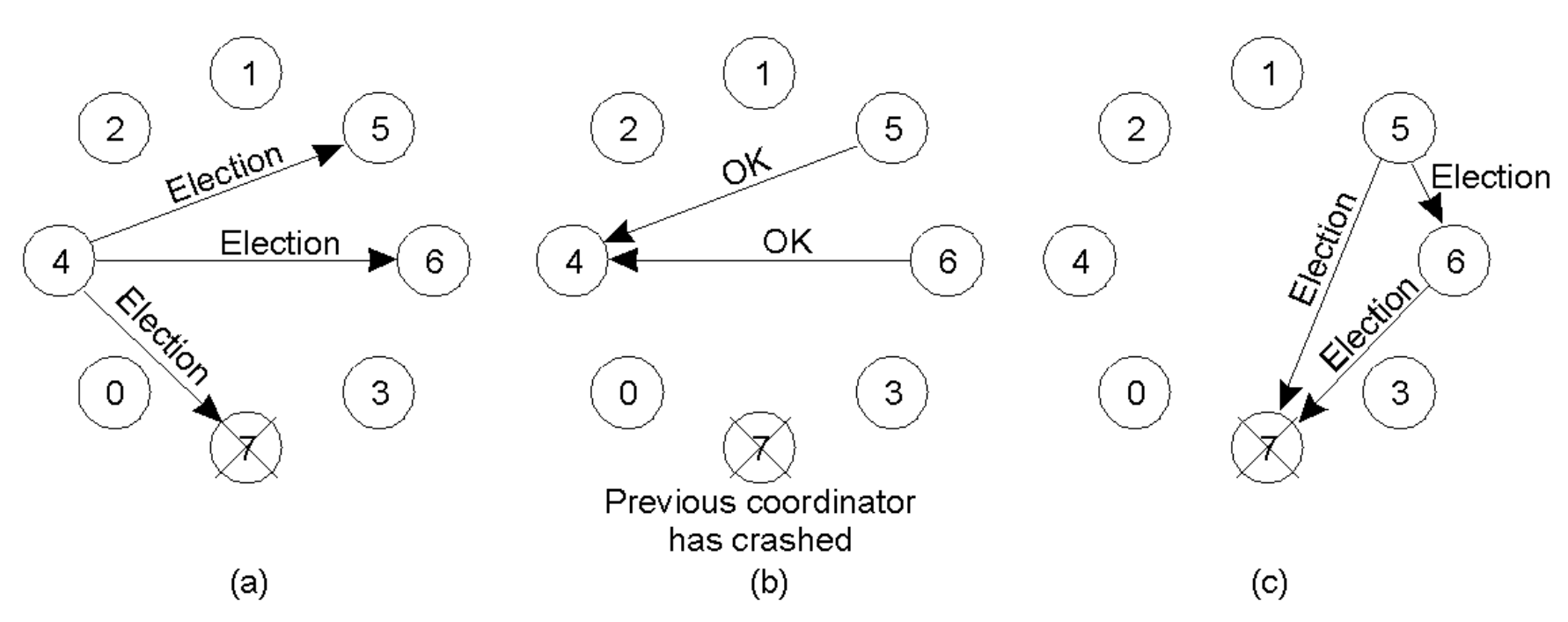
Bully Algorithm

- When a process gets an ELECTION message from one of its lower-numbered colleagues:
  - Receiver sends an OK message back to the sender to indicate that it is alive and will take over.
  - Receiver holds an election, unless it is already holding one.
  - Eventually, all processes give up but one, and that one is the new coordinator.
  - The new coordinator announces its victory by sending all processes a message telling them that starting immediately it is the new coordinator.

Bully Algorithm

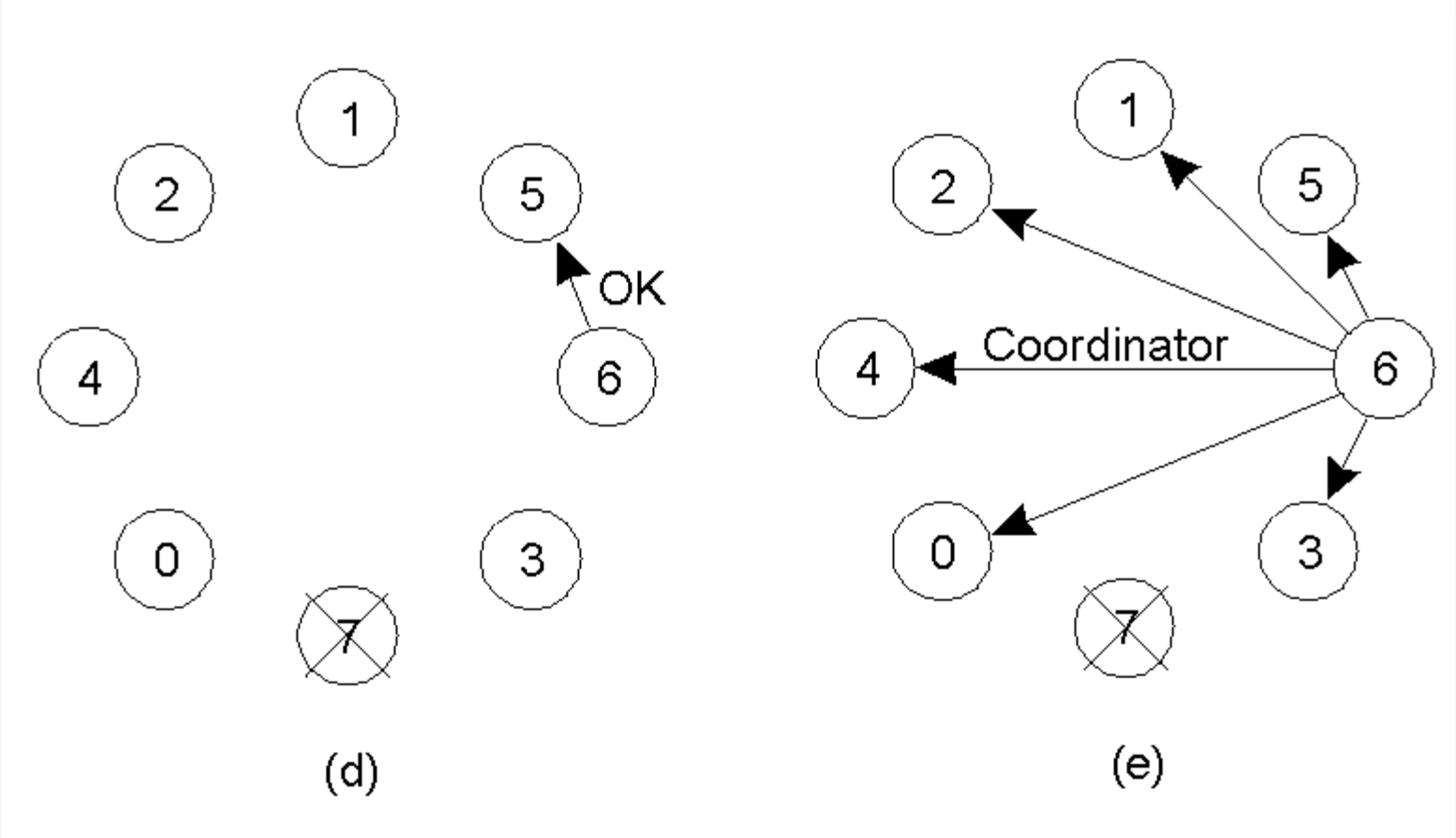
- If a process that was previously down comes back:
  - It holds an election.
  - If it happens to be the highest process currently running, it will win the election and take over the coordinator’s job.
- “Biggest guy” always wins and hence the name “bully” algorithm.

The Bully Algorithm (Example)



- The bully election algorithm
- Process 4 holds an election
- Process 5 and 6 respond, telling 4 to stop
- Now 5 and 6 each hold an election

The Bully Algorithm (Example)



Bully Algorithm Analysis

- Best case
  - The node with second highest identifier detects failure
  - Total messages = N-2
    - One message for each of the other processes indicating the process with the second highest identifier is the new coordinator.
- Worst case
  - The node with lowest identifier detects failure. This causes N-1 processes to initiate the election algorithm each sending messages to processes with higher identifiers.
  - Total messages = $O(N^2)$

Bully Algorithm Discussion

- How many processes are used to detect a coordinator failure?
  - As many as you want. You could have all other processes check out the coordinator.

- It is impossible for two processes to be elected at the same time.

Distributed Mutual Exclusion

Introduction

- What is Mutual Exclusion?
- Concurrent access to a shared resource or data by processes in mutually exclusive manner.
- Only one process is allowed to execute the critical section (CS) at any given time.
- Shared variables or a local OS cannot be used to implement mutual exclusion in a distributed system.
- Only way to implement mutual exclusion in DS is by message passing.
- The mutual exclusion algorithms will have consider the underlying message delays that might be unpredictable and partial knowledge of the system state.

System Model

- The system is assumed to have N sites - S_1, S_2, ..., S_N.
- A single process is assumed to be running on each site. Let P_i be the process at site S_i.
- A site can be in one of the following three states:
  - Requesting the CS
  - Executing the CS
  - Neither requesting nor executing the CS (i.e., idle).
- The site is blocked in the 'requesting the CS' state, and it can not make further requests for the CS.
- In the 'idle' state, the site is executing outside the CS.
- At any instant, a site may have several pending requests for CS. A site queues up these requests and serves them one at a time.

Requirements

- **Safety Property:**
- Only one process can execute the critical section at any instant.

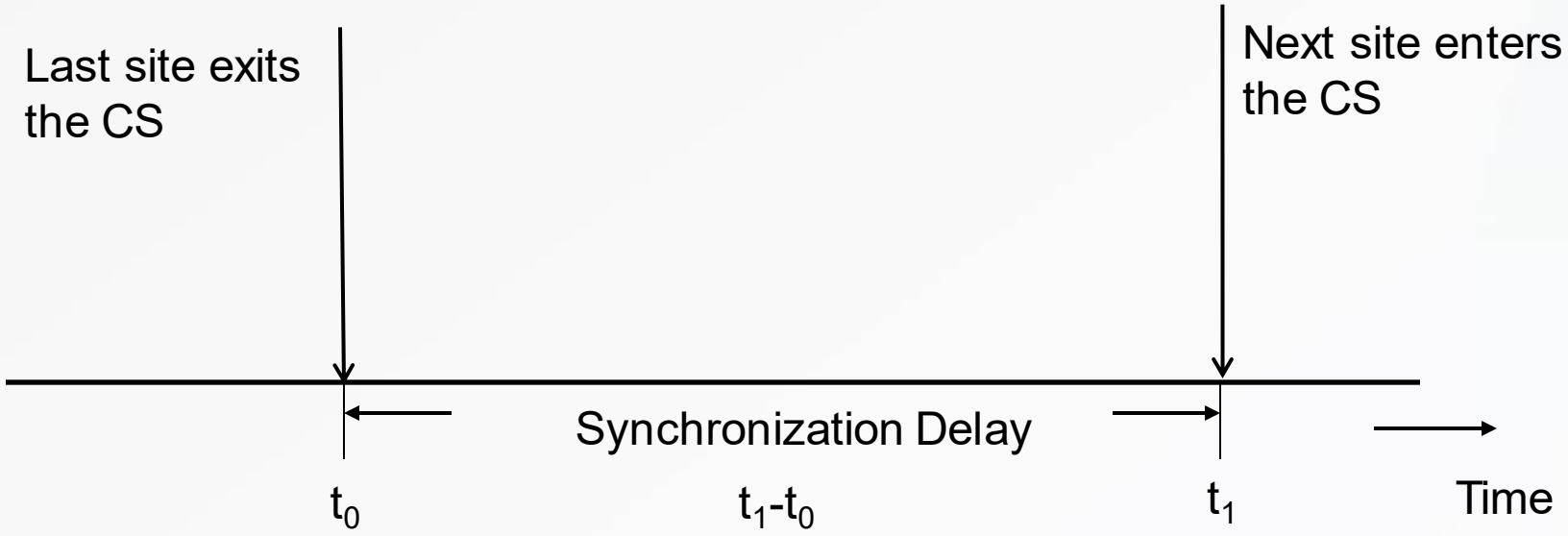
- **Liveness Property:**
- No deadlock and starvation.
- Two or more sites should not endlessly wait for messages which will never arrive.

- **Fairness Property:**
- Each process gets a fair chance to execute the CS.
- In general, it means that the CS execution requests are executed in the order of their arrival in the system.

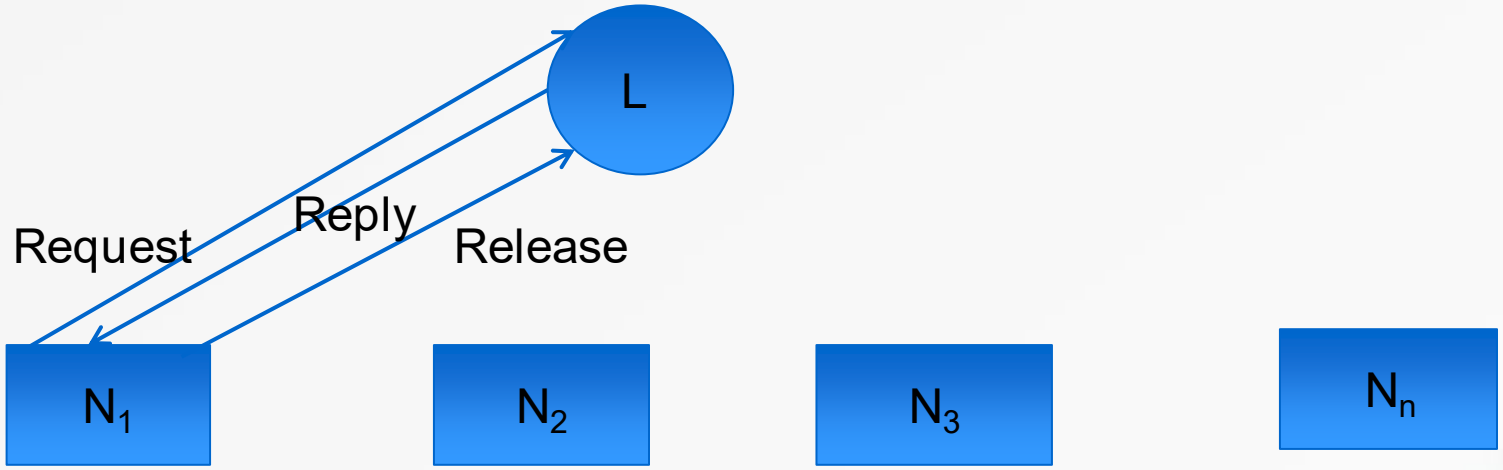
Performance Metrics

- **Message complexity:**
- The number of messages required per CS entry/execution by a site.

- **Synchronization delay:** After a site leaves the CS, it is the time (in terms of messages) required before the next site enters the CS.



Centralized Algorithm



- Message Complexity- 3 messagess
- Synchronization delay: 2T, where T is the message latency.
- Problem: Scalability

Types of Algorithms

- Permission-based Algorithms
  - A node N_i has a request set R_i. N_i asks permission from all nodes in R_i if it wants to enter in its CS.

- Token-based Algorithms
  - There is a single token available. A node having token goes in CS.

Lamport's Algorithm

- Requests for CS are executed in the increasing order of timestamps and time is determined by logical clocks.
- Every site S_i keeps a queue, request $queue_i$, which contains mutual exclusion requests ordered by their timestamps.
- This algorithm requires communication channels to deliver messages in the FIFO order.

Lamport's Algorithm

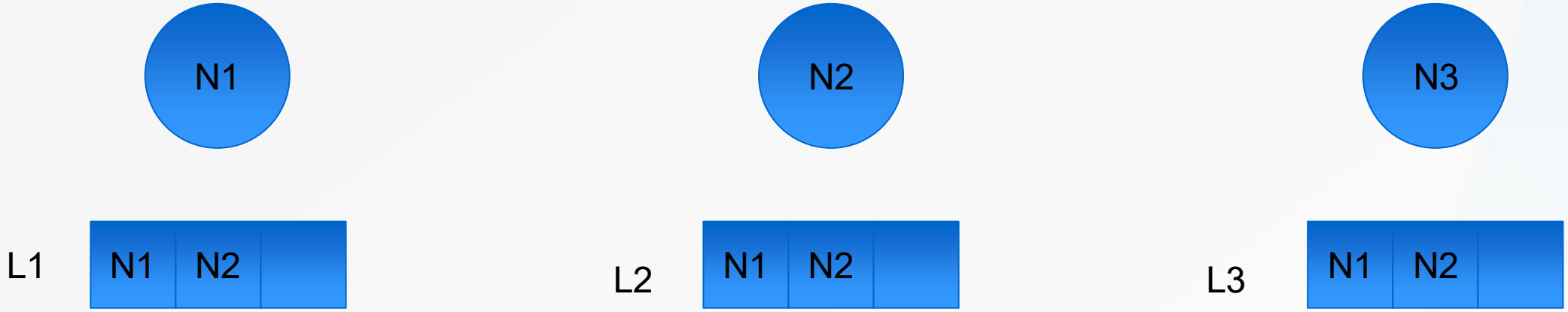
- Requesting the critical section:
  - When a site S_i wants to enter the CS, it broadcasts a Request(ts_i, i) message to all other sites and places the request on request $queue_i$. ((ts_i, i) denotes the timestamp of the request)
  - When a site S_j receives the Request (ts_i, i) message from site S_i, it places site S_i 's request on request $queue_j$ and it returns a timestamped Reply message to S_i.
- Executing the critical section:
  - Site S_i enters the CS when the following two conditions hold:
    - L1: S_i has received a message (REPLY) with timestamp larger than (ts_i, i) from all other sites.
    - L2: S_i's request is at the top of request $queue_i$.

Lamport's Algorithm

- Releasing the critical section:
  - Site S_i , upon exiting the CS, removes its request from the top of its request queue and broadcasts a timestamped Release message to all other sites.
  - When a site S_j receives a Release message from site S_i, it removes S_i's request from its request queue.
  - When a site removes a request from its request queue, its own request may come at the top of the queue, enabling it to enter the CS.

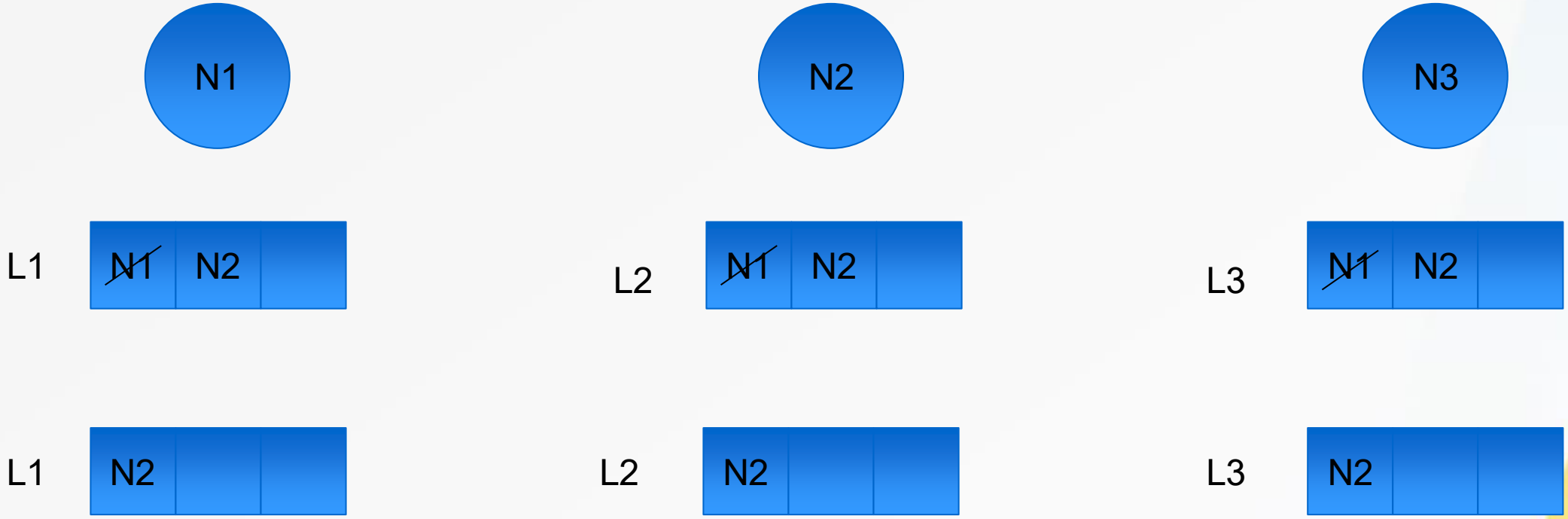
Example

- Entering CS



Example

- Exiting CS



Correctness of the Algorithm

- The proof is by contradiction.
- Suppose Sites S_i and S_j are executing the CS concurrrently.
- Conditions L1 and L2 must hold at both the sites concurrrently.
- At some instant t, both S_i and S_j have their own requests at the top of their request queues and condition L1 holds for them.
- Assume that S_i's request has a smaller timestamp than that of S_j.

- Using a condition L1 and FIFO channels, at t, S_i must be present in request $queue_j$, when S_j was executing its CS.
- S_j's own request is at the top of its own request queue, when a smaller timestamp request (S_i's request) is present in the request $queue_j$- a contradiction.

Properties of ME

- N_1 enters CS as its timestamp smaller.
- Release message exits CS and deletes its (N_1's) entry from all queues.
- Now, N_2 finds itself at the head of the queue and eventually goes into CS.

a. Mutual Exclusion is guarnteed as FIFO queue, only one of them will go in CS and not two at the same time.

b. Progress is guarnteed as nodes are deleted from local queues (L_i, L_j, ...) after release messages.

c. Bounded wait or starvation is avoided as the last node comes to the front.

Performance

- Message Complexity
  - For each CS execution, Lamport's algorithm requires $(N - 1)$ Request messages, $(N - 1)$ Reply messages, and $(N - 1)$ Release messages.
- Synchronization Delay
  - SD in the algorithm is T (1 msg), i.e. release.

Reduction in Message Complexity

- Reply messages can be omitted in certain situations.
- For example, if a site S_j receives a Request message from site S_i after it has sent its own Request message with timestamp higher than the timestamp of site S_i's request, then site S_j need not send a Reply message to site S_i.

- This is because when site S_i receives site S_j 's request with timestamp higher than its own, it can conclude that site S_j does not have any smaller timestamp request which is still pending.

- With this optimization, Lamport's algorithm requires between $3(N - 1)$ and $2(N - 1)$ messages per CS execution.

Ricart-Agrawala's Algorithm

- The Ricart-Agrawala algorithm assumes that the communication channels are FIFO. The algorithm uses two types of messages: Request and Reply.
- A process sends a Request message to all other processes to request their permission to enter the critical section. A process sends a Reply message to a process to give its permission to that process.
- Processes use logical clocks to assign a timestamp to critical section requests and timestamps are used to decide the priority of requests.
- Each process P_i maintains a Deferred queue, which is initially empty.

Ricart-Agrawala's Algorithm

- Requesting the critical section:

  (a) When a site S_i wants to enter the CS, it broadcasts a timestamped Request message to all other sites.

  (b) When site S_j receives a Request message from site S_i, it sends a Reply message to site S_i if site S_j is neither requesting nor executing the CS, or if the site S_j is requesting and S_i's request's timestamp is smaller than site S_j's own request's timestamp. Otherwise, the request is inserted in the Deferred queue.

- Executing the critical section:

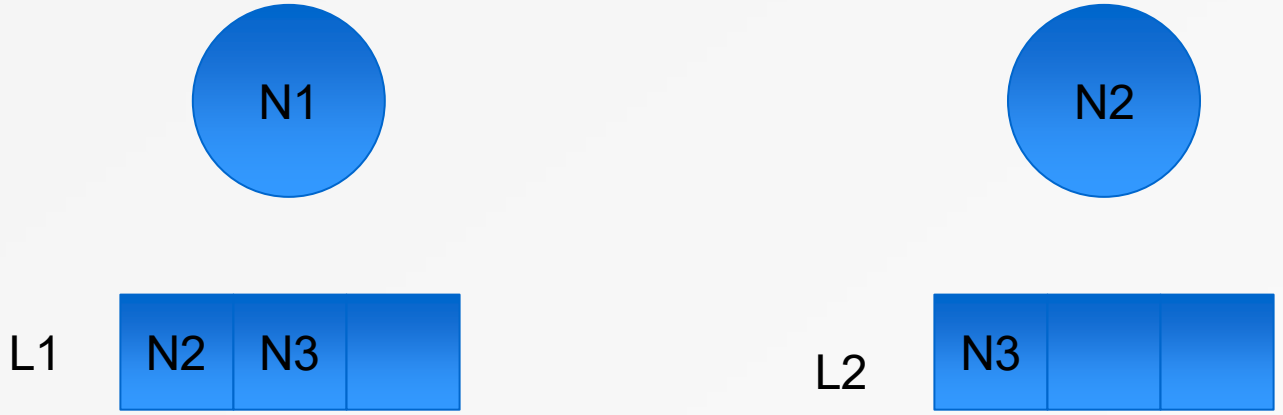
  - Site S_i enters the CS after it has received a Reply message from every site it sent a Request message to.

Ricart-Agrawala's Algorithm

- Releasing the critical section:
  - When site Si exits the CS, it sends Reply messages to all the nodes in the Deferred queue. It then deletes the entries in the Deferred queue.

Example

- Entering CS -When N1 is in CS, N2 has already made a request and N3 now wants to go in CS



Performance

- For each CS execution, Ricart-Agrawala algorithm requires (N − 1) Request messages and (N − 1) Reply messages.
- Synchronization delay in the algorithm is T.

Maekawa's Algorithm

- A node requires permission to be obtained from only a subset of other sites, called the Request Set (or Quorum)
- Each site S_i has a separate Request Set, R_i.

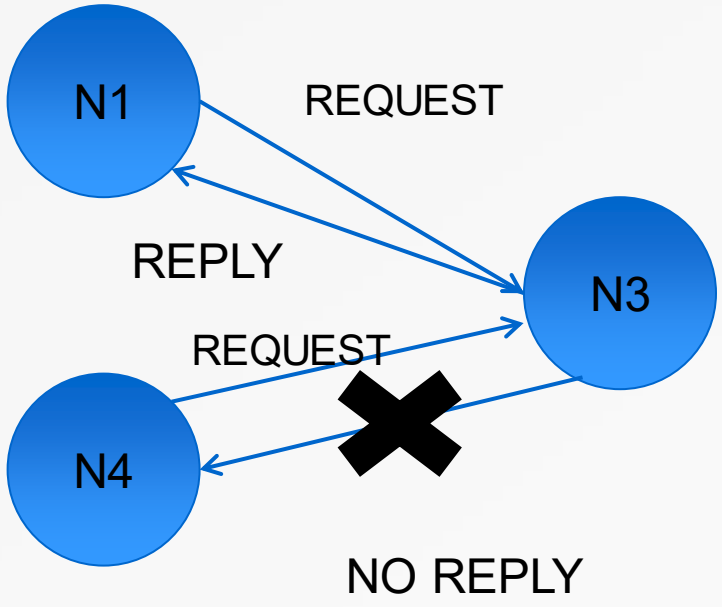
- Requirements:
  - for all i, j $R_i \cap R_j \neq \Phi$
  - for all i: $S_i \in R_i$
  - for all i: $|R_i| = K$, for some K
  - for all i: Si is contained in exactly is K Request Sets

- Condition: $N = K(K - 1) + 1$ $K = O(\sqrt{N})$
- How the request set is chosen?

Maekawa's Algorithm

- Conditions M1 and M2 are necessary for correctness; whereas conditions M3 and M4 provide other desirable features to the algorithm.
- Condition M3 states that the size of the requests sets of all sites must be equal implying that all sites should have to do equal amount of work to invoke mutual exclusion.
- Condition M4 enforces that exactly the same number of sites should request permission from any site implying that all sites have "equal responsibility" in granting permission to other sites.

Maekawa's Algorithm



Request Sets

R1 = { N1, N2, N3}

R2 = {N2, N4, N5}

R3 = {N3, N4, N6}

R4 = {N4, N1, N5}

R5 = {N5, N3, N6}

R6 = { N6, N1, N2}

N = 6

K = 3

Maekawa's Algorithm

- Requesting the critical section
  - (a) A site S_i requests access to the CS by sending Request messages to all sites in its request set R_i .
  - (b) When a site S_j receives the Request message from S_i, it sends a Reply message to S_i provided it hasn't sent a Reply message to a site since its receipt of the last Release message. Otherwise, it queues up the request of S_i for later consideration.
- Executing the critical section
  - Site S_i executes the CS only after it has received a Reply message from every site in R_i .

Maekawa's Algorithm

- Releasing the critical section

  (a) After the execution of the CS is over, site Si sends a Release message to every site in Ri .

  (b) When a site Sj receives a Release message from site Si, it sends a Reply message to the next site waiting in the queue and deletes that entry from the queue. If the queue is empty, then the site updates its state to reflect that it has not sent out any Reply message since the receipt of the last Release message.

Correctness

- Proof is by contradiction.
- Suppose two sites Si and Sj are concurrently executing the CS.
- This means site Si received a REPLY message from all sites in Ri and concurrently site Sj was able to receive a REPLY message from all sites in Rj.
- If $Ri \cap Rj = \{Sk\}$, then site Sk must have sent REPLY messages to both Si and Sj concurrently, which is a contradiction.

Performance

- Since the size of a request set is $\sqrt{N}$, an execution of the CS requires $\sqrt{N}$ Request, $\sqrt{N}$ Reply, and $\sqrt{N}$ Release messages
- Synchronization delay in this algorithm is 2T.

Token-based Algorithms

- Unique token circulates among the sites present in the system.
- A site is permitted to enter CS if it has token.
- Token-based algorithms use sequence numbers instead of time stamps (used to differenciate between old and new sequence nos.)
- Request for a token contains a sequence number.
- Sequence number of sites are advanced independently.
- Non-FIFO channels are assumed.

Suzuki-Kasami's Algorithm

- If a site does not have a token and wants to enter a CS, it sends a broadcast REQUEST message to all other sites to get the token
- A site having the token sends it to the requesting site upon the receipt of its REQUEST message.
- A site if executing the CS upon the receipt of a REQUEST message, then it sends the token to the requesting site after exiting the CS.
- Token holder can make multiple CS accesses.

Suzuki-Kasami's Algorithm Cont..

- How to distinguish an outdated REQUEST message from a current REQUEST message?
- When a site S_i receives a Request from a node S_j with a seq. no. 'n' then the request is outdated if the sequence no. (for node S_j's requests for CS) stored at S_i is greater than what is received in the current request message (n).
- How to determine which site has an outstanding request for the CS?
  - After a site has finished the execution of the CS, it must determine which sites have an outstanding request for the CS so that the token can be dispatched to one of them.

Suzuki-Kasami's Algorithm

- A Request message of site S_j has the form Request(j, n) where n (n=1, 2, ...) is a sequence number which indicates that site S_j is requesting its n^{th} CS execution.
- A site S_i keeps an array of integers RN_i [1..N] (RN-requesting node) (array is local to every node) where RN_i [j] denotes the largest sequence number received in a Request message so far from site S_j.
- When site S_i receives a Request(j, n) message, it sets RN_i [j]:= max(RN_i [j], n).
- When a site S_i receives a Request(j, n) message, the request is outdated if RN_i [j] > n.

Suzuki-Kasami's Algorithm

- The token consists of a queue of requesting sites, Q, and an array of integers **LN[1..N]** (LN: Last Node), where LN[j] is the sequence number of the request which site S_j executed most recently (both Q and array flow among the nodes).

- After executing its CS, a site Si updates **$LN[i] := RN_i [i]$** to indicate that its request corresponding to sequence number **$RN_i [i]$** has been executed.

- At site S_i, if **$RN_i [j] = LN[j]+1$**, then site S_j is currently requesting token.

Suzuki-Kasami's Algorithm

- Requesting the critical section

  (a) If requesting site S_i does not have the token, then it increments its sequence number, $RN_i[i]$, and sends a Request(i, sn) message to all other sites. ('sn' is the updated value of $RN_i[i]$.)

  (b) When a site S_j receives this message,
  It sets $RN_j[i] = \max(RN_j[i], sn)$.
  If S_j has the idle token, then it sends the token to S_i if $RN_j[i]=LN[i]+1$.

- Executing the critical section
  - Site S_i executes the CS after it has received the token.

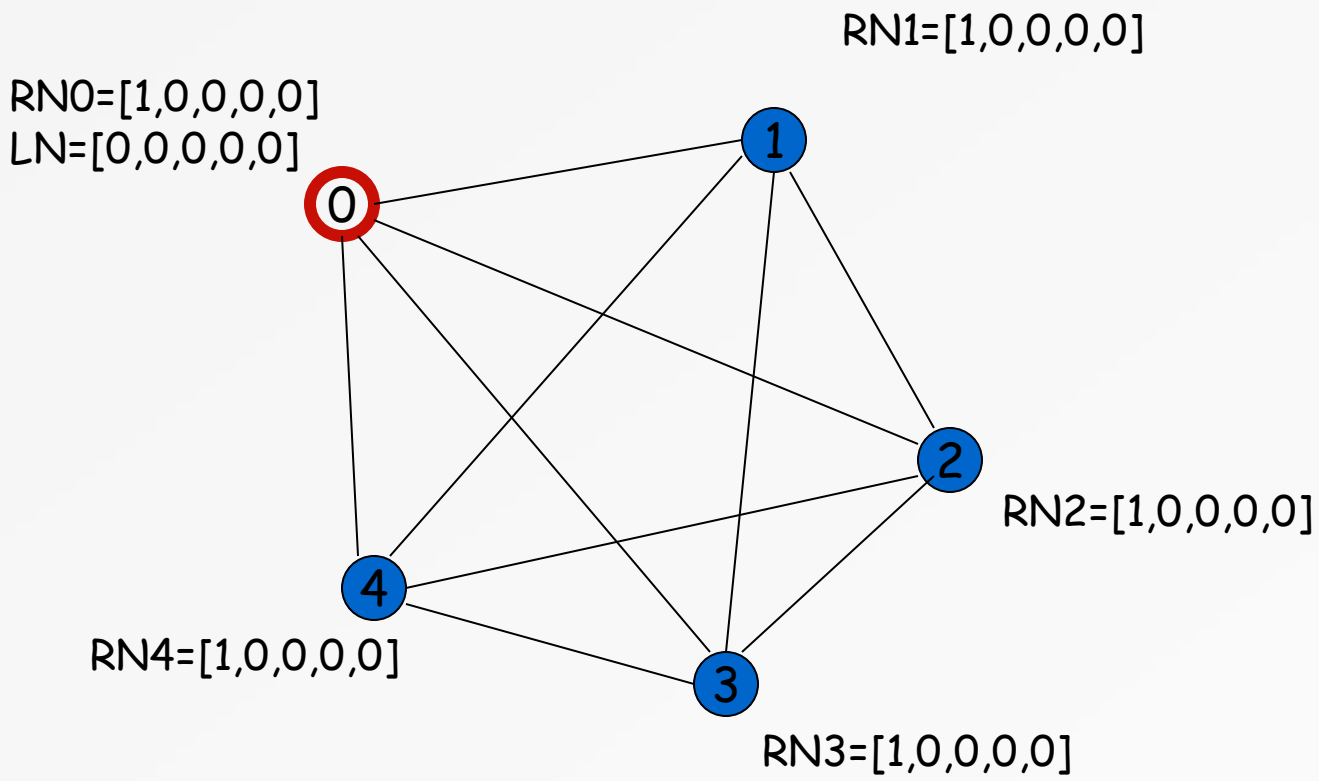
Suzuki-Kasami's Algorithm

- Releasing the critical section

  Having finished the execution of the CS, site S_i takes the following actions:

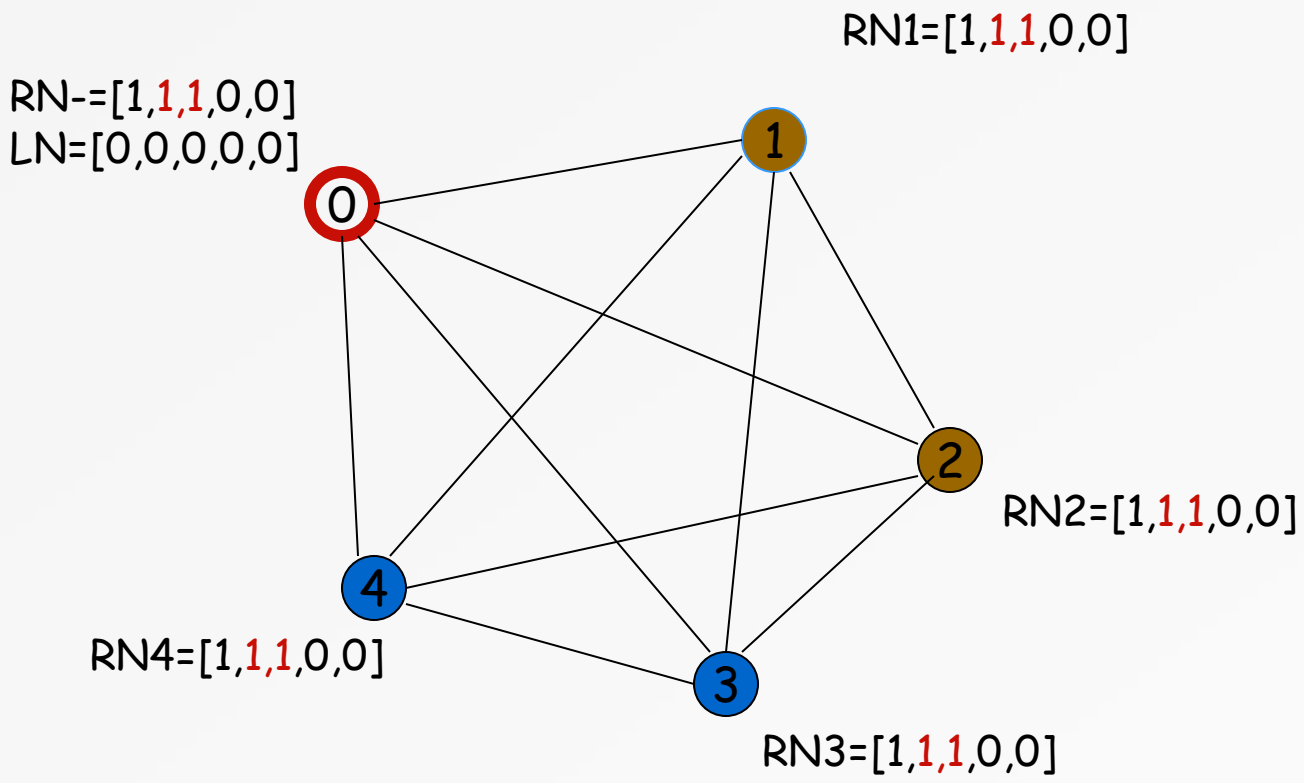
  - It sets LN[i] = RN_i [i].
  - For every site S_j whose id is not in the token queue, it appends S_j to the token queue (Q) if RN_i [j]= LN[j] + 1.
  - If the token queue is nonempty after the above update, S_i deletes the top site id from the token queue and sends the token to the site indicated by the id.

Example



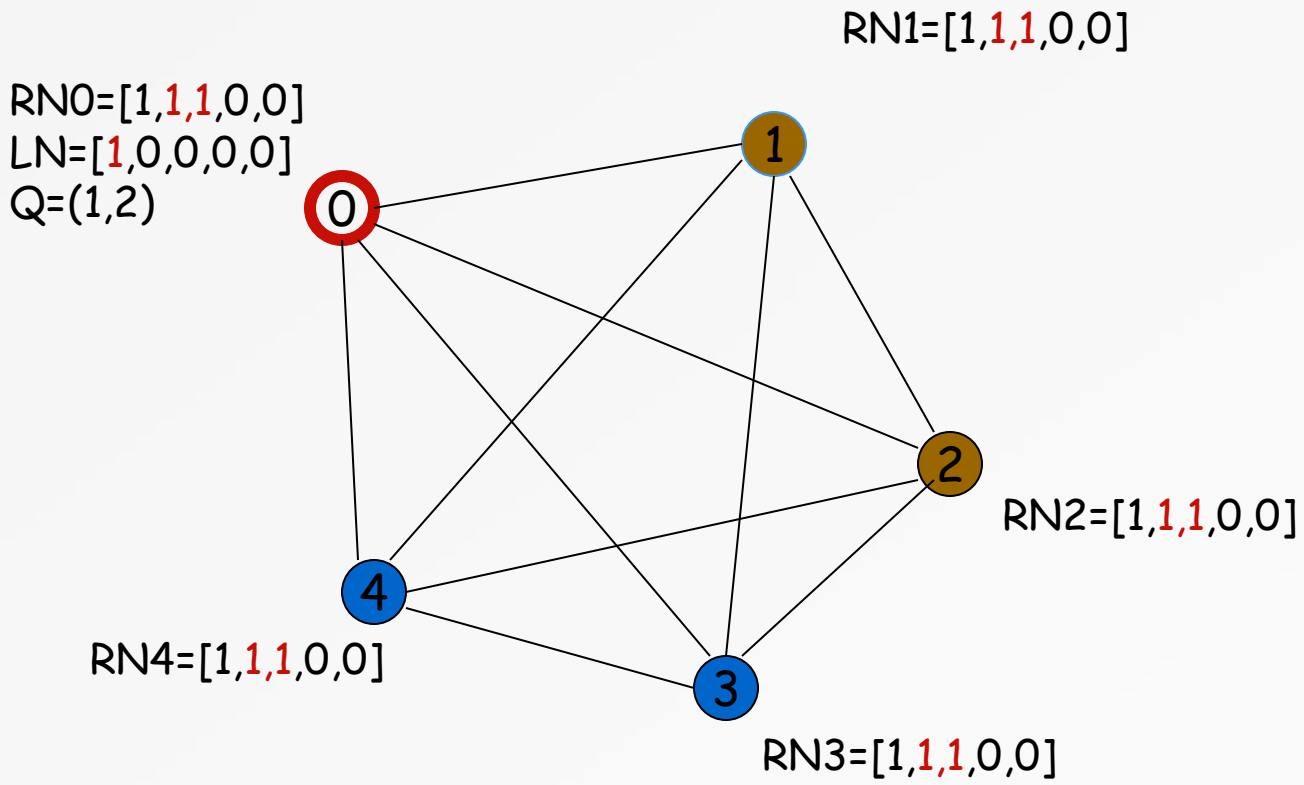
initial state: process 0 has sent a request to all, and grabbed the token

Example



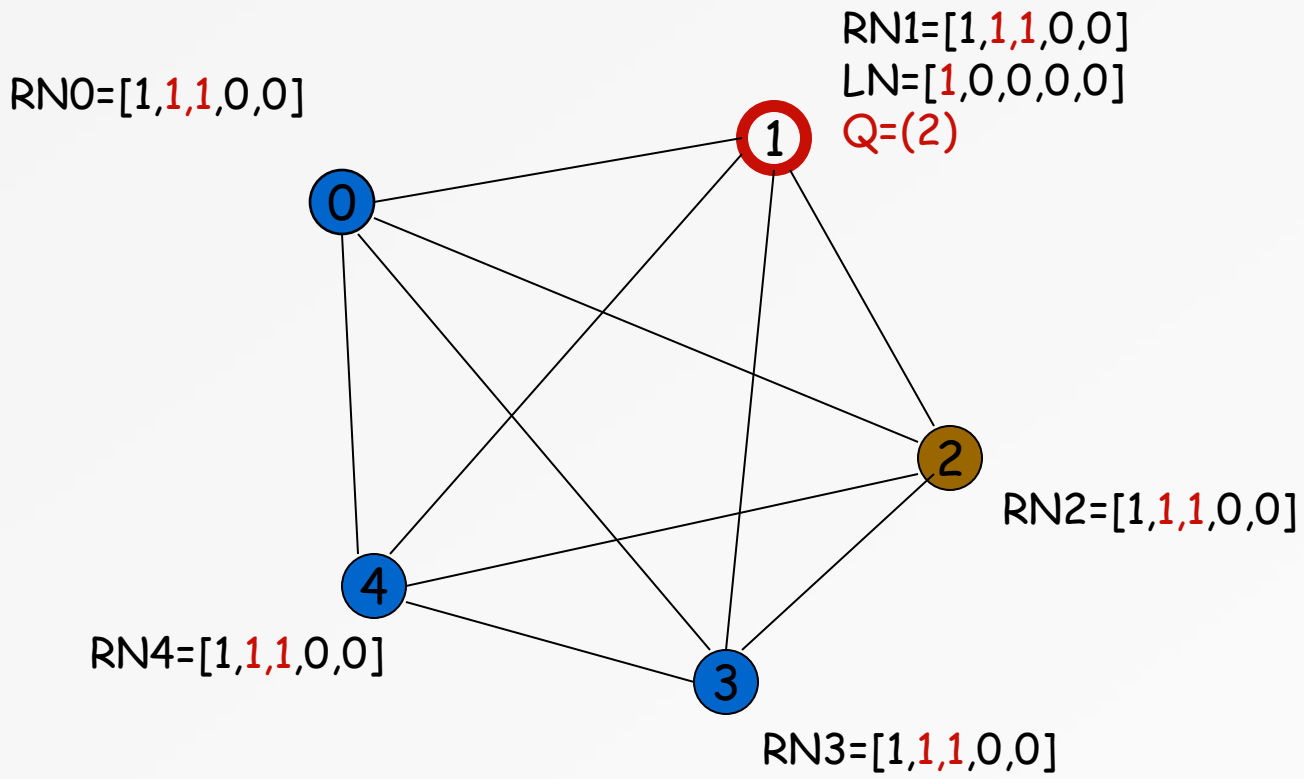
1 & 2 send requests to enter CS

Example



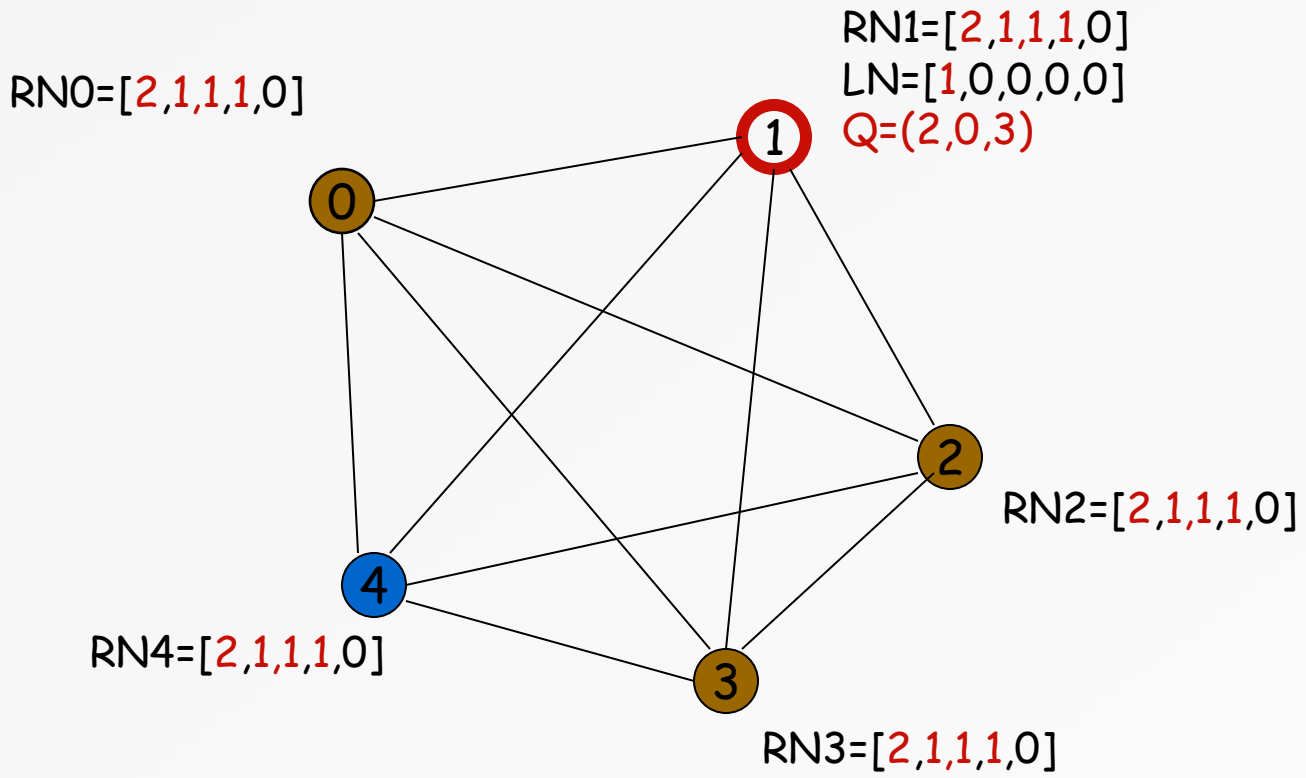
0 prepares to exit CS

Example



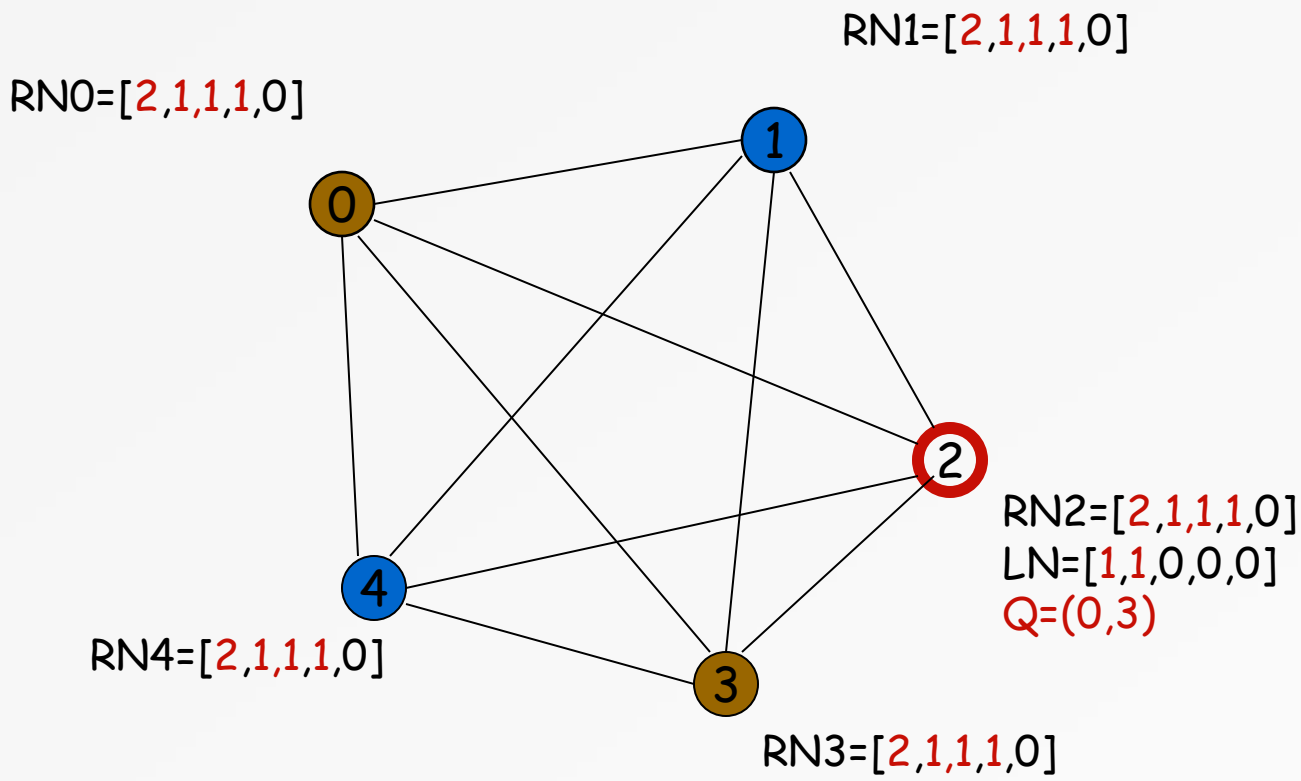
0 passes token (Q and last) to 1

Example



0 and 3 send requests

Example



1 sends token to 2

Suzuki-Kasami's Algorithm

- Correctness
  - Mutual Exclusion – Trivial (Token Based)
  - Progress and Bounded Wait –
    - Token has FIFO Queue in which every node can be there only once.
    - Eventually a node enters its CS
- Performance Measures
  - Message Complexity – either 0 or N
  - Synchronization Delay either 0 or T

Raymond's Algorithm

- Forms a directed logical tree with the token holder as the root.
- Each node has a variable '*parent*' that points to its parent. Root's *parent* variable points to itself.
- Each node N_i has a FIFO request queue Q_i.

Raymond's Algorithm

- To Request the CS a node N_i does the following
  - Send REQUEST msg to its parent if N_i does not hold the token and Q_i is empty. Then place the entry of N_i in Q_i.

- When a non-root node N_j gets a REQUEST msg from a node N_i, it does the following
  - Place the request of N_i in Q_j
  - Send REQUEST to parent if not sent already

Raymond's Algorithm

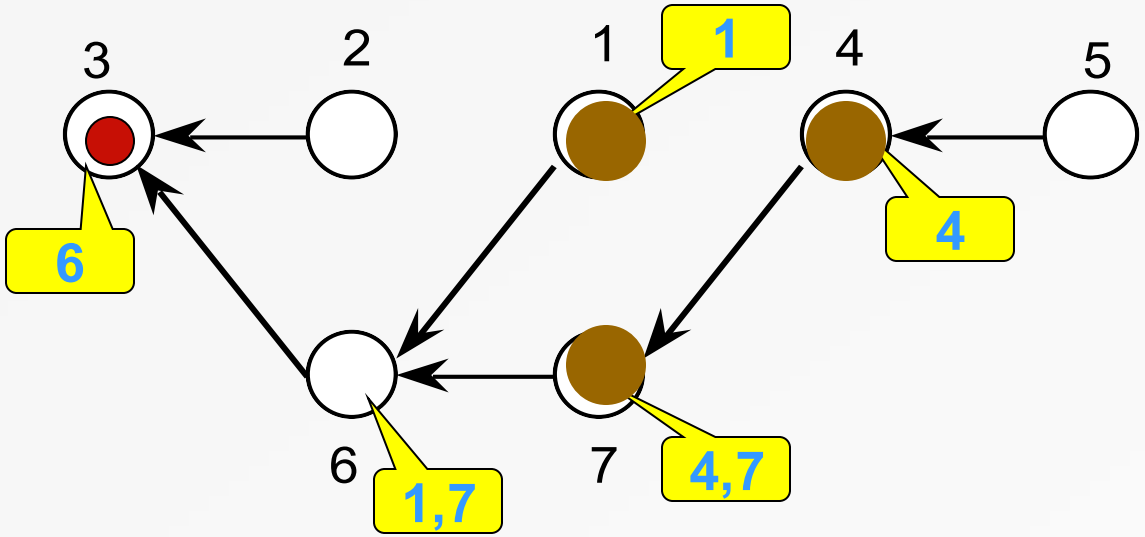
- When the root node receives a REQUEST message, it does the following
  - Send token to the requesting node
  - Make the parent pointer point to the requesting node

- When a node N_i receives a token, it does the following
  - Delete the first entry, say N_j, from Q_i
  - Send token to N_j and sets parent to N_j
  - If Q_i is non-empty, send REQUEST msg to N_j.

Raymond's Algorithm

- Entering the CS
  - A node N_i can enter its CS if it gets the token and its entry is in the front of Q_i
  - After entering, it deletes its entry from Q_i.
- When a node N_i exits the CS, it does the following
  - It deletes the front entry from Q_i, say N_j, and sends token to N_j.
  - It sets its parent to Nj.
  - If Q_i is still not empty, it sends REQUEST msg to N_j.

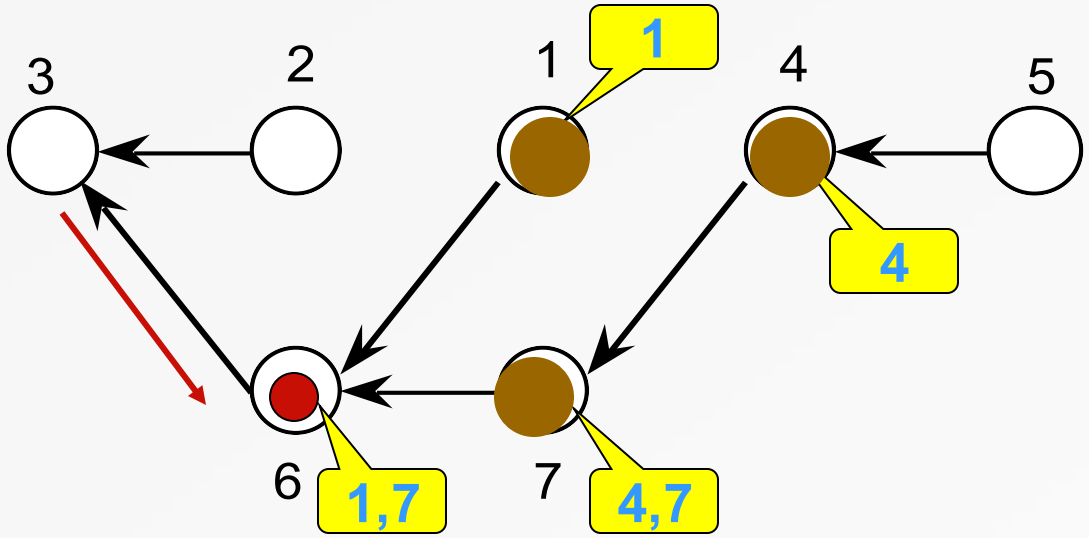
Raymond's tree-based algorithm



3 is the root

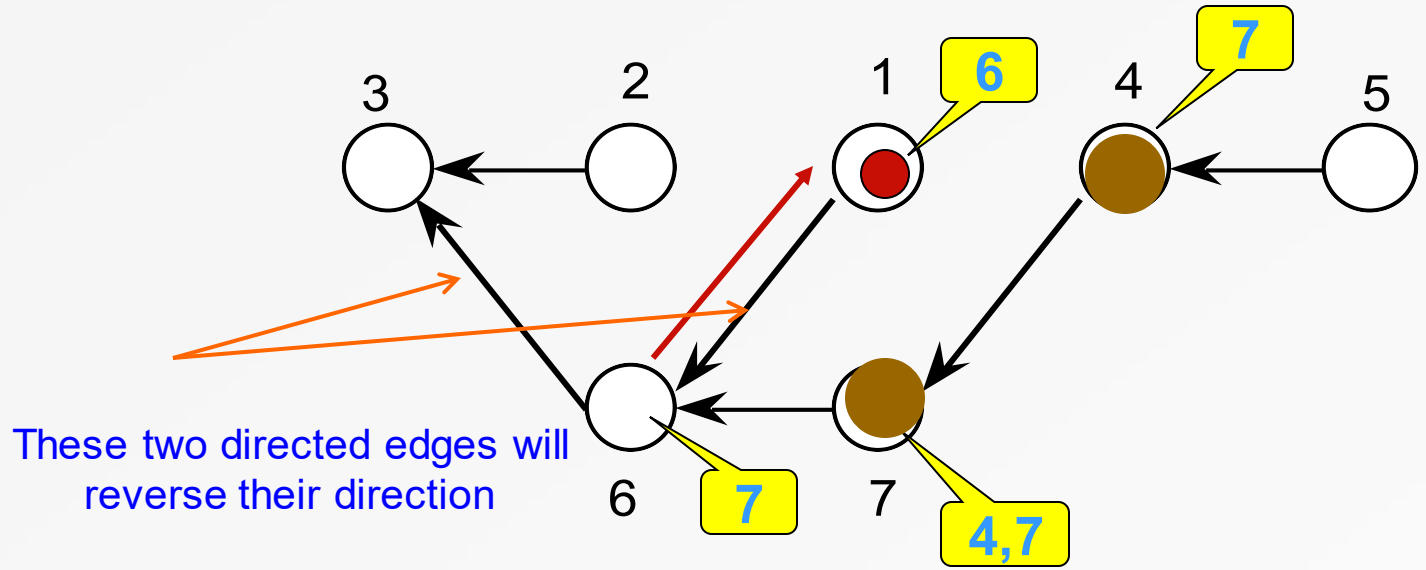
1,4,7 want to enter their CS

Raymond's Algorithm



3 sends the token to 6

Raymond's Algorithm



6 forwards the token to 1

Raymond's Algorithm

- Correctness
  - Mutual Exclusion – Trivial (token based)
  - Deadlock Freedom – Due to tree structure (no cycles)
  - No Starvation – Requests at a node are entered in a FIFO queue. A node can be present in a queue atmost once. So, every node eventually gets to enter its CS.