What is Mutual exclusion?

- A condition in which there is a set of processes, **only one of which is able to access a given resource** or perform a given function at any time

Requirements of Mutual Exclusion Algorithms

1. **Safety property**: At most one process may execute in the critical region (CR) at a time.

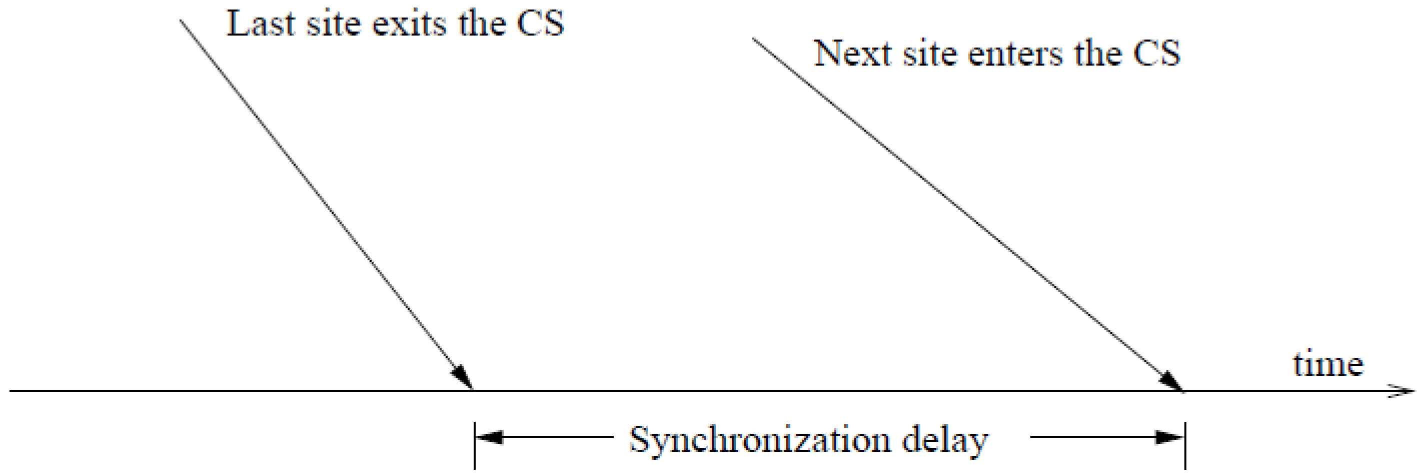
2. **Liveness property**: A process requesting entry to the CR is eventually granted it. There should not be deadlock and starvation

3. **Fairness:** Each process should get a fair chance to execute the CR.

PERFORMANCE METRICS

1. **Message complexity:** Number of messages that are required per CR execution by a process.

1. **Synchronization delay:** Time interval between critical region (CR) exit and new entry by any process.



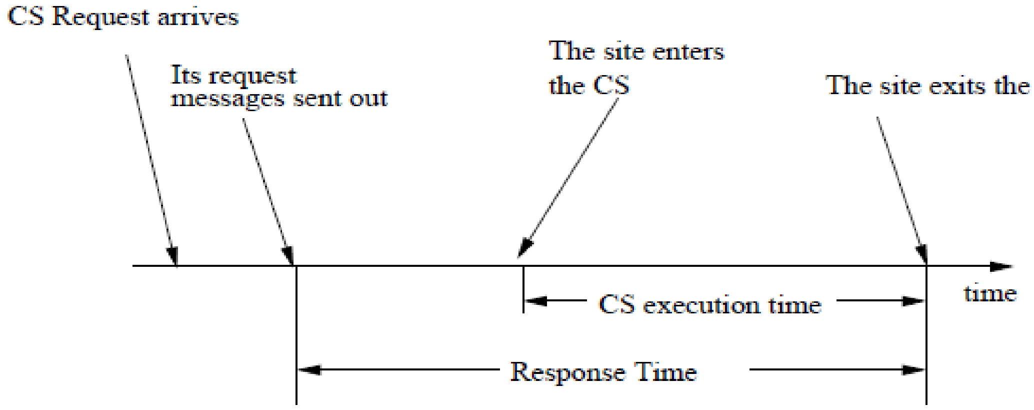
Performance Metrics

3. **System throughput:** Rate at which requests for the CR get executed.

system throughput=1/(SD+E)

where SD is the synchronization delay and E is the average critical section execution time.

4. **Response time:** Time interval from a request send to its CR execution completed.



Classification of Mutual Exclusion Algorithms

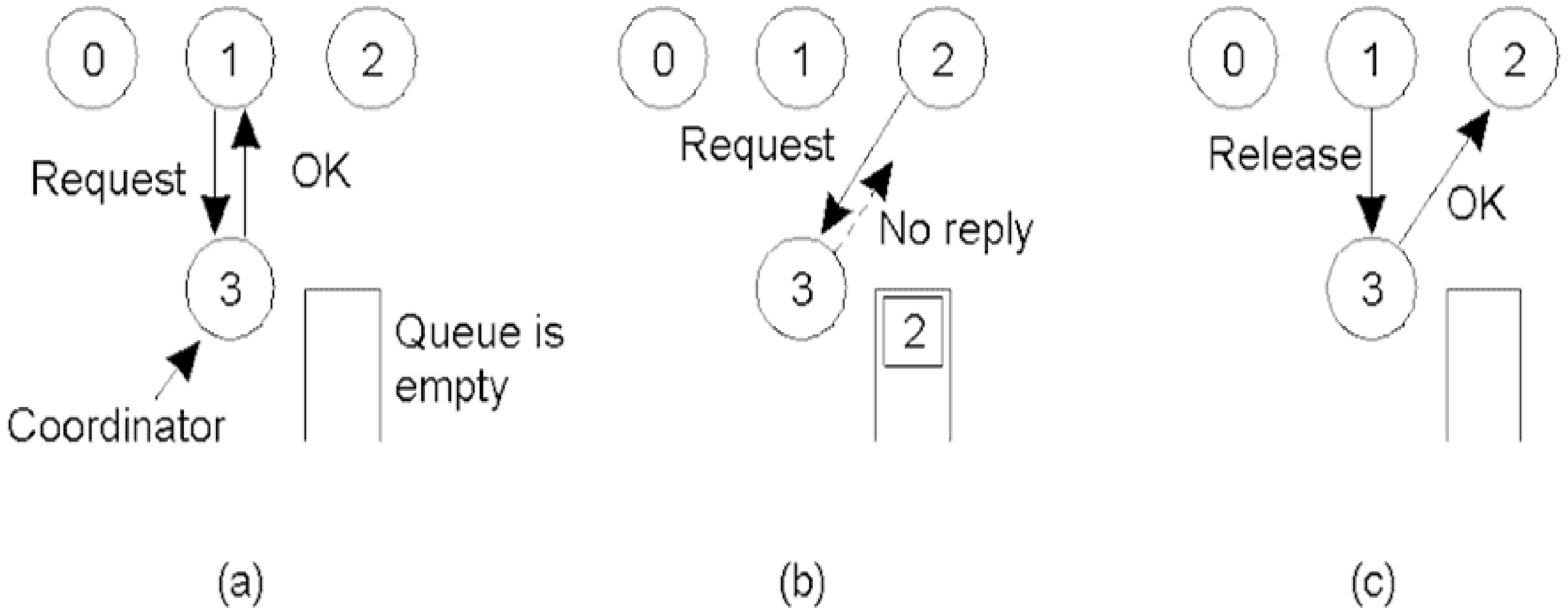
- **Centralized mutual exclusion algorithms**

- **Distributed mutual exclusion algorithms**
  - Non-token-based algorithm:
    - Lamport’s Distributed Mutual Algorithm
    - Ricart–Agrawala Algorithm
    - Maekawa’s Algorithm
  - Token-based algorithm:
    - Suzuki–Kasami’s Broadcast Algorithm
    - Singhal’s Heuristic Algorithm
    - Raymond’s Tree-Based Algorithm

System Model:Preliminaries

- The system consists of N sites, S1, S2, ..., SN.
- We assume that a single process is running on each site. The process at site Si is denoted by pi
- A site can be in one of the following three states: requesting the CS,executing the CS, or neither requesting nor executing the CS (i.e., idle).
- In the ‘requesting the CS’ state, the site is blocked and can not make further requests for the CS. In the ‘idle’ state, the site is executing outside the CS.
- In token-based algorithms, a site can also be in a state where a site holding the token is executing outside the CS (called the idle token state).
- 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.

Centralized Algorithm



(a) Process 1 asks the coordinator for permission to access a shared resource. Permission is granted.
(b) Process 2 then asks permission to access the same resource. The coordinator does not reply.
(c) When process 1 releases the resource, it tells the coordinator, which then replies to 2.

Centralized algorithms contd..

- Advantages
  - Fair algorithm, grants in the order of requests
  - The scheme is easy to implement
  - Scheme can be used for general resource allocation

  Critical Question: When there is no reply, does this mean that the coordinator is “dead” or just busy?

- Shortcomings
  - Single point of failure. No fault tolerance
  - Confusion between No-reply and permission denied
  - Performance bottleneck of single coordinator in a large system

Performance Parameters

1. The algorithm is simple and fair as it handles the request in the sequential order.

2. It guarantees no starvation.

3. It uses three messages as REQUEST, REPLY and RELEASE.

4. It has a single point of failure.

5. Coordinator selection could increase synchronization delay especially at times of frequent failures.

Distributed algorithm

Distributed Mutual Exclusion algorithm

Three basic approaches for distributed mutual exclusion:

1 Token based approach

2 Non-token based approach

3 Quorum based approach

Distributed Mutual Exclusion algorithm

Non-token based approach:

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

Token-based approach:

- A unique token is shared among the sites.
- A site is allowed to enter its CS if it possesses the token.
- Mutual exclusion is ensured because the token is unique.

Non-token-based algorithm:

- Lamport's Distributed Mutual Algorithm
- Ricart–Agrawala Algorithm
- Maekawa's Algorithm

1.Lamport's Distributed Algorithm

- Lamport developed a distributed mutual exclusion algorithm
- The algorithm is fair in the sense that a request for CS are executed in the order of their timestamps and time is determined by logical clocks.
- When a site processes a request for the CS, it updates its local clock and assigns the request a timestamp.
- The algorithm executes CS requests in the increasing order of timestamps.
- Every site Si keeps a queue, request_queue i, which contains   mutual exclusion requests ordered by their timestamps
- This algorithm requires communication channels to deliver messages in FIFO order.

Lamport's…..

Requesting the critical section

- •When a site Si wants to enter the CS, it broadcasts a REQUEST(tsi, i) message to all other sites and places the request on request_queuei. ((tsi, i) denotes the timestamp of the request.)
- When a site Sj receives the REQUEST(tsi, i) message from site Si, it places site Si's request on request_queue j and returns a timestamped REPLY message to Si.

Lamport's.....

Executing the critical section

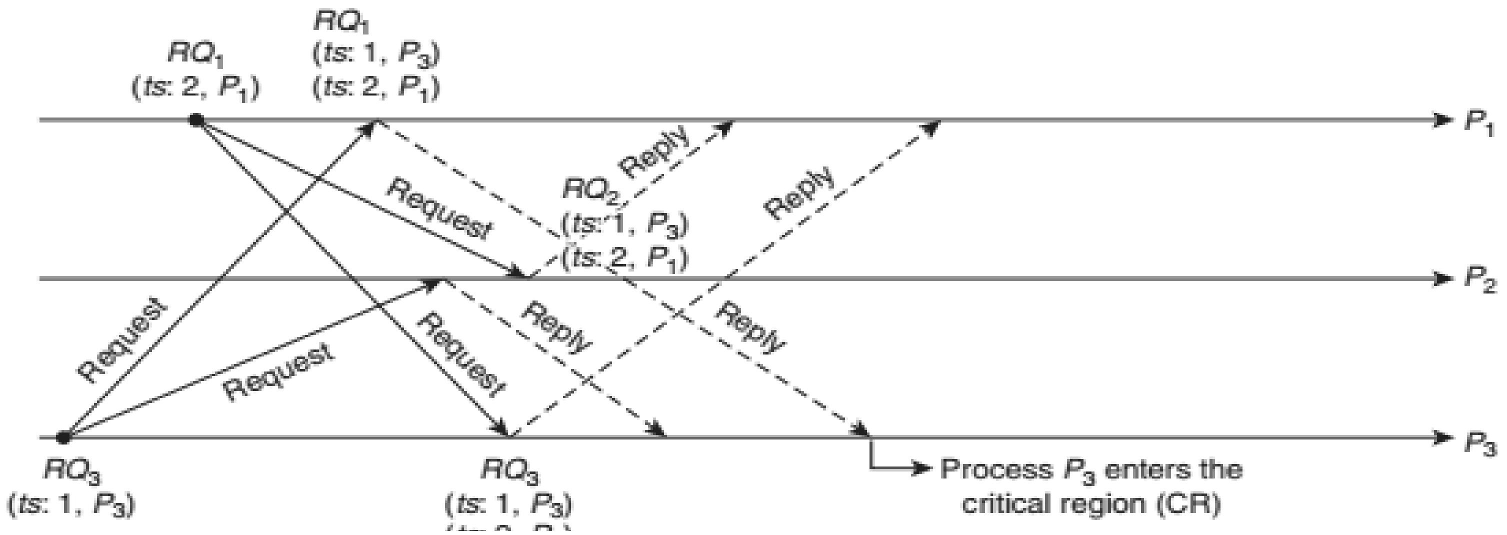
- Site Si enters the CS when the following two conditions hold:
  - **L1: Si has received a message with timestamp larger than (tsi, i) from all** other sites.
  - **L2: Si's request is at the top of request_queuei.**

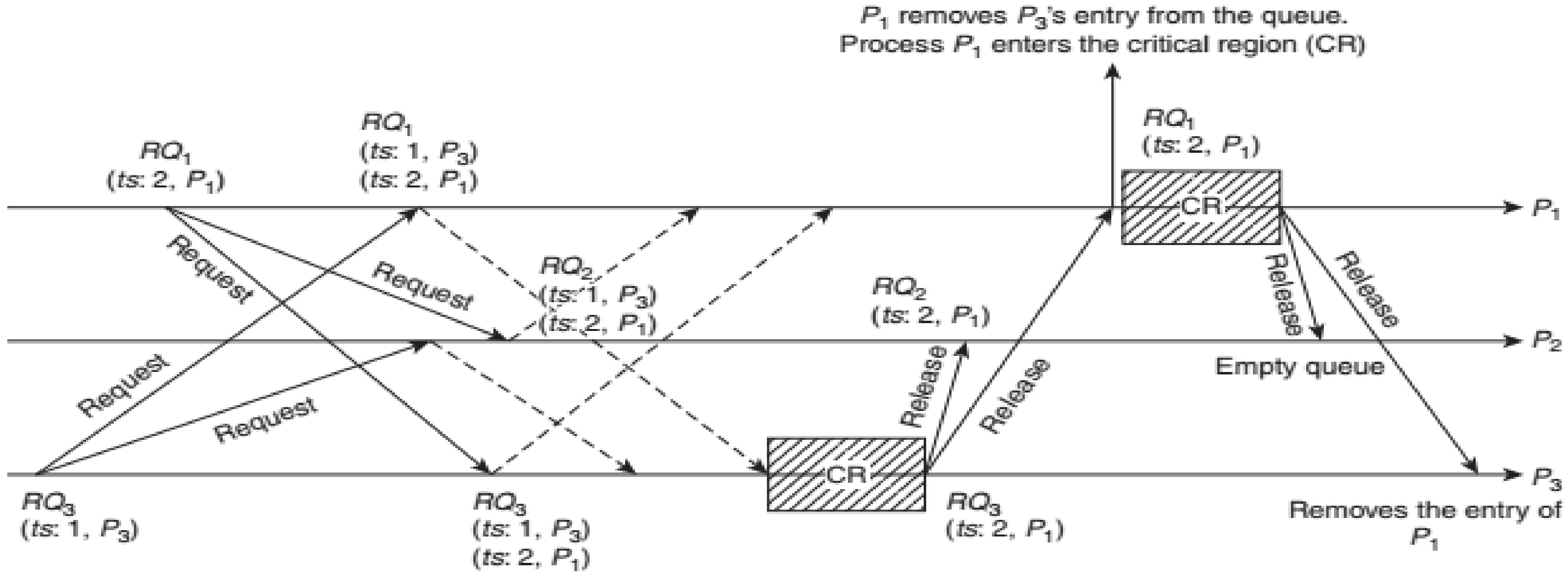
Lamport's…..

Releasing the critical section

- Site Si, 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 Sj receives a RELEASE message from site Si, it removes Si's request from its request queue.

1.Lamport's Distributed:Example



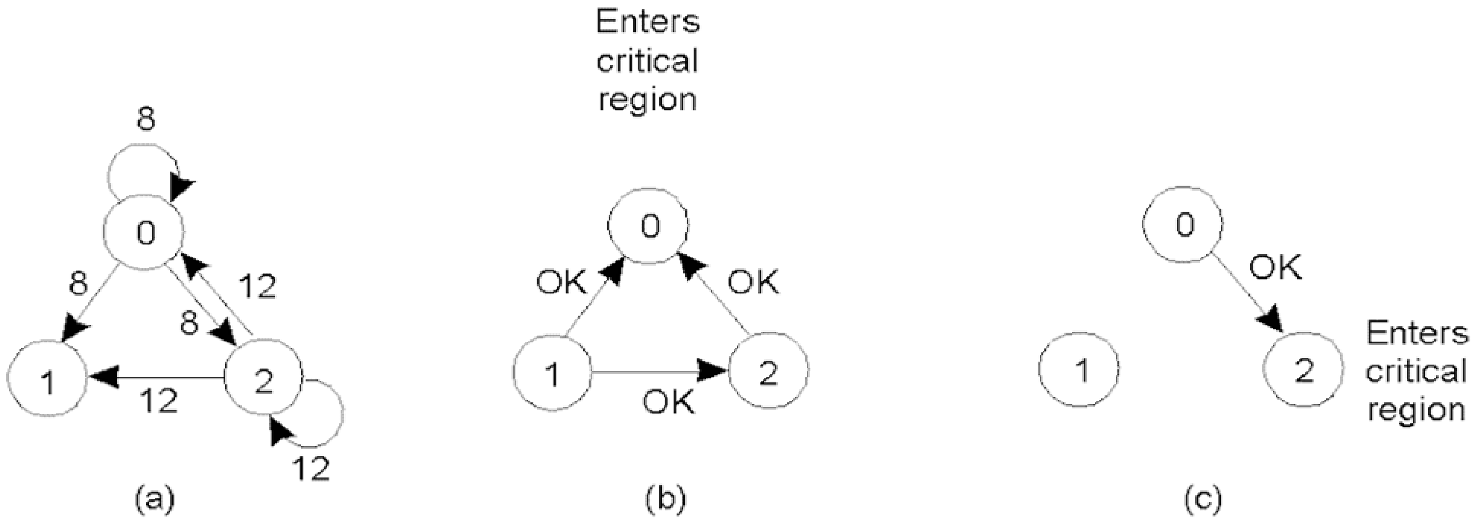


(b)

Performance Parameters

- Lamport's algorithm has message overhead of total **3(N − 1)** messages: N – 1 REQUEST messages to All process (N minus itself), N −1 REPLY messages, and N −1 RELEASE messages per CR invocation.
- The synchronization delay is T. Throughput is 1/(T + E).
- **The algorithm has been proven to be fair and correct. It can also be optimized by reducing the number of RELEASE messages sent.**

2.Ricart–Agrawala Algorithm



A> Two processes want to enter the same critical region .

B> Process 0 has the lowest timestamp, so it wins.

C> When process 0 is done, it sends an OK also, so 2 can now enter the critical region[2]

2.Ricart–Agrawala Algorithm

- **Requesting the critical section**

  (a) When a site Si wants to enter the CS, it broadcasts a timestamped

  REQUEST message to all other sites.

  (b) When site Sj receives a REQUEST message from site Si, it sends a

  REPLY message to site Si if site Sj is neither requesting nor executing the CS, or if the site Sj is requesting and Si's request's timestamp is smaller than site Sj's own request's timestamp. Otherwise, the reply is deferred and Sj sets RDj i = 1.

2.Ricart–Agrawala Algorithm

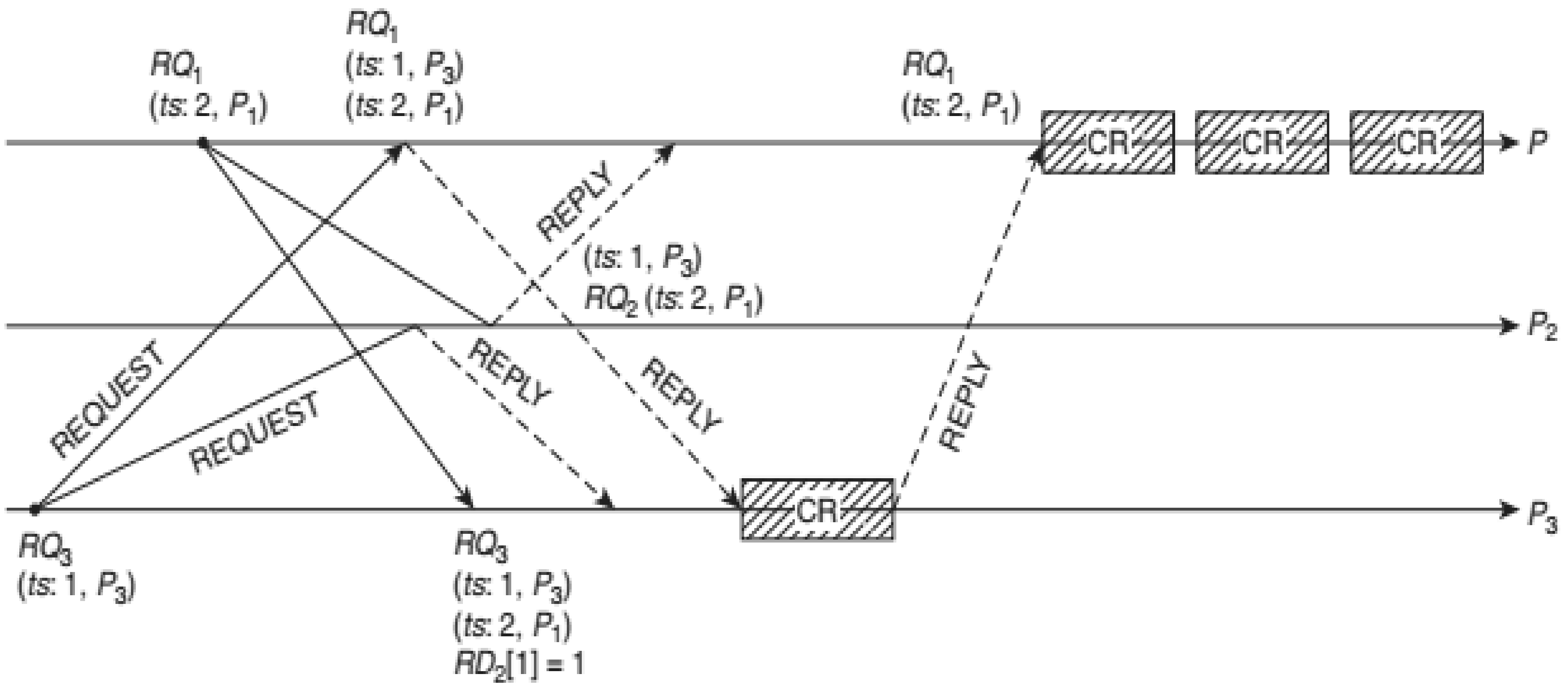
- **Executing the critical section**

  (c) Site Si enters the CS after it has received a REPLY message from every site it sent a REQUEST message to.

- **Releasing the critical section**

  (d) When site Si exits the CS, it sends all the deferred REPLY messages: ∀j if RDij = 1, then sends a REPLY message to Sj and sets RDij = 0.

RICART–AGRAWALA ALGORITHM



Performance Parameters

1.The algorithm does not use explicit RELEASE message. The dequeuing is done on the receipt of REPLY itself. Thus, total message overhead would be **2(N − 1) messages, that is, for entering a CR, (N − 1) requests and exiting (N − 1) replies.(Improvement over lamport's algo)**

2. The failure of any process almost halts the algorithm (recovery measures are needed) as it requires all replies.

3.Single point of failure is replaced by multiple point of failure.

Maekawa's Algorithm

- Maekawa's algorithm is a quorum or voting-based mutual exclusion algorithm.
- It suggests that a process *Pi does not require to request all processes, but only to a subset of processes (the quorum) called Ri.*
- Each process *Pi in the quorum set Ri gives permission to at most one process at a time.*
- **Data structures used by each process *Pi:***

  1. The request-deferred queue, *RDi*

  /* of processes REQUESTing and not REPLIED to*/

  2. A variable called 'Voted' = F(ALSE) is set initially; /* T(RUE) when a reply is sent indicating that it has already granted permission to a process in its quorum */

Maekawa's Algorithm

Properties of a Quorum Set

1. $(R_i \cap R_j != \text{Null})$, (for all i and j in $i != j, 1 \leq i, j \leq N$)

   /*Intersection of any two Quorum sets should not be Null*/

2. $(P_i \in R_i)$, (for all i, $1 \leq i \leq N$)

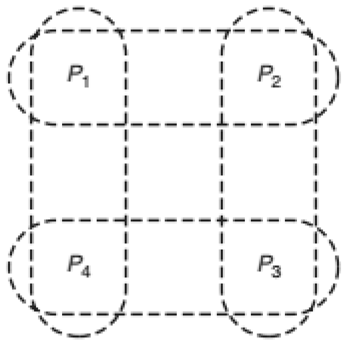
   /*Each process belongs to its own quorum set R_i*/

3. $(|R_i| = K)$, (for all i in $1 \leq i \leq N$)

   /*Size of each quorum set is K*/

4 Any process P_j is contained in some M number of R_i s (for $1 \leq i, j \leq N$.)

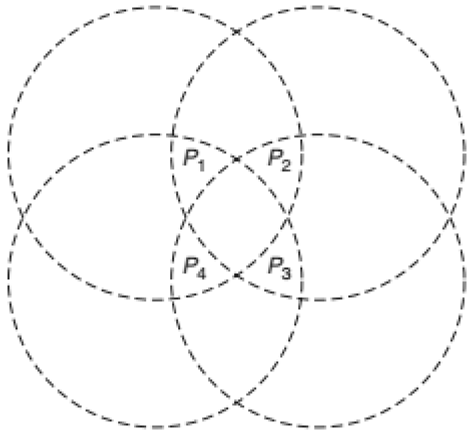
DIFFERENT VALID QUORUMS EXAMPLES



N = 4, R = 4, K = 2

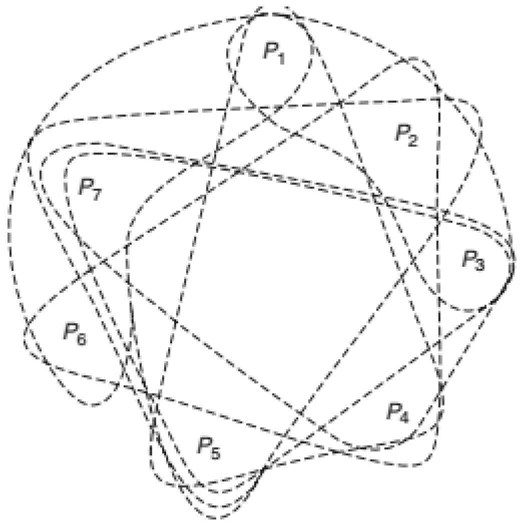
$R_1 = \{P_1, P_2\}$
$R_2 = \{P_2, P_3\}$
$R_3 = \{P_3, P_4\}$
$R_4 = \{P_4, P_1\}$

(a)



N = 4, R = 4, K = 3

$R_1 = \{P_1, P_2, P_4\}$
$R_2 = \{P_2, P_1, P_3\}$
$R_3 = \{P_3, P_2, P_4\}$
$R_4 = \{P_4, P_1, P_3\}$



N = 7, R = 7, K = 3

$R_1 = \{P_1, P_2, P_3\}$
$R_2 = \{P_2, P_4, P_6\}$
$R_3 = \{P_3, P_5, P_7\}$
$R_4 = \{P_1, P_4, P_5\}$
$R_5 = \{P_5, P_2, P_7\}$
$R_6 = \{P_6, P_1, P_7\}$
$R_7 = \{P_7, P_3, P_4\}$

(c)

N = No. of processes; R = No. of Quorums; K = size of Quorum

Maekawa's algorithm.

Requesting the critical section:
(a) A site Si requests access to the CS by sending REQUEST(i) messages to all sites in its request set Ri.
(b) When a site Sj receives the REQUEST(i) message, it sends a REPLY(j) message to Si 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(i) for later consideration.

Executing the critical section:
(c) Site Si executes the CS only after it has received a REPLY message from every site in Ri.

Releasing the critical section:
(d) After the execution of the CS is over, site Si sends a RELEASE(i) message to every site in Ri.
(e) When a site Sj receives a RELEASE(i) 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.

Algorithm

```
1. The critical region request message from $P_i$
   • $P_i$ sends a REQUEST(timestamp$_i$, $i$) to all $P_j$ in its quorum set $R_i$ ( for $j! = i$))
2. $P_j$ on receiving the REQUEST message from $P_i$
   • If variable Voted$_j$ = True or if the process $P_j$ itself is currently executing the CR
      o then REQUEST from $P_i$ is deferred and pushed in the queue $RD_j$.
      o else REPLY is send to $P_i$                /* termed as permission granted*/
        and variable Voted$_j$ is set to TRUE
3. The execution of the CR
   • If all REPLY received from processes in $R_i$, enter CR.
4. The release of the CR: $P_i$
   After execution of CR,
   Process $P_i$ sends RELEASE to all $P_j$ in $R_i$.
5. The receipt of RELEASE message:           /* $P_i$ sends to all $P_j$ in $R_i$*/
   • If $RD_j$ is nonempty
      ∘ dequeue top of the queue $RD_j$,
          /* looks out for requests which came while $P_i$ was in CR*/
      ∘ $P_j$ sends a REPLY message to only this dequeued process.
      ∘ Set Voted$_j$ to be TRUE
   • If queue $RD_j$ was empty
      ∘ Voted$_j$ = false
```

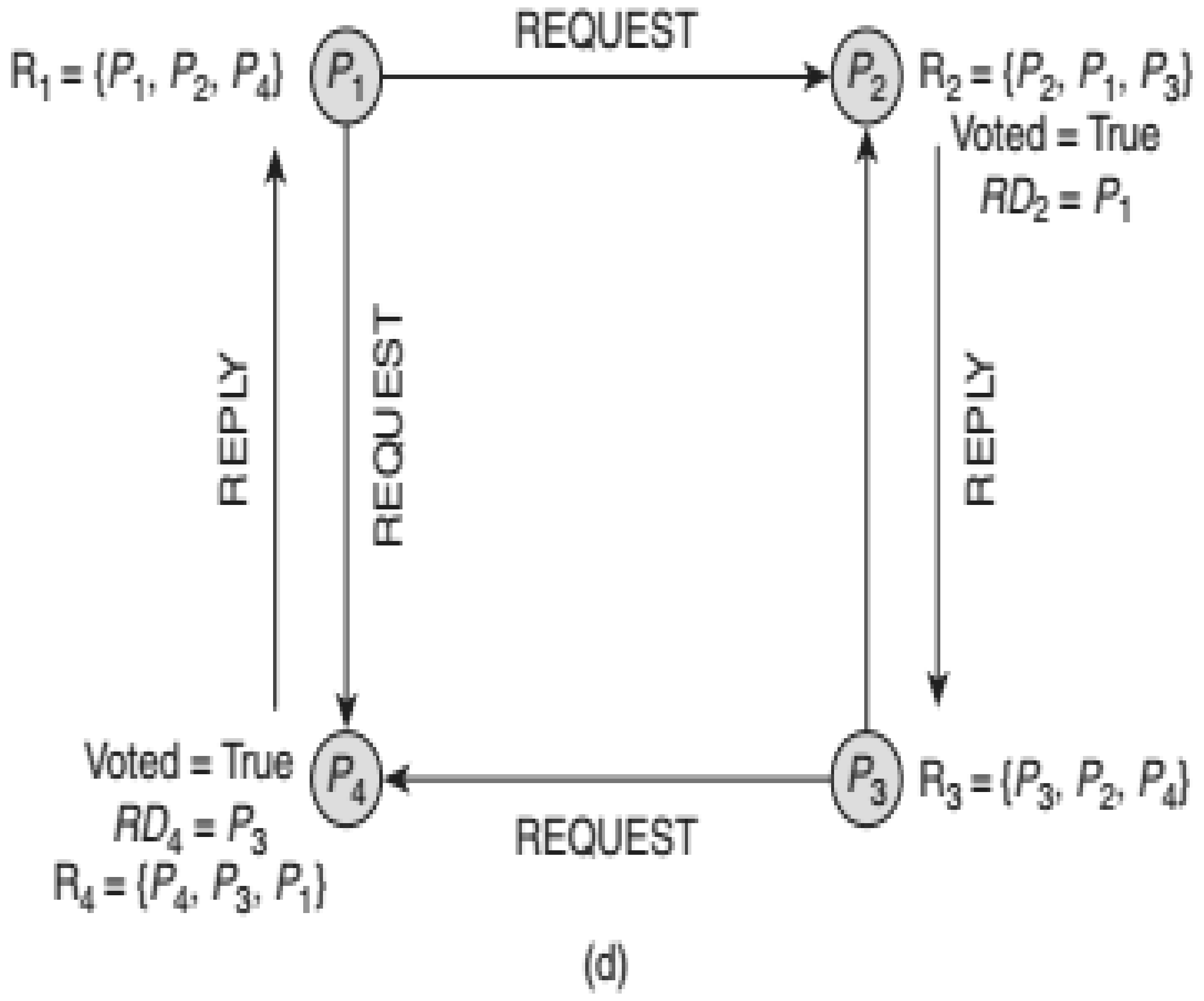
Problem of deadlocks

- Though Maekawa's algorithm has been proven to be correct and safe, but the property of liveness is not satisfied by it because it can lead to deadlock

- Maekawa's algorithm can deadlock because a site is exclusively locked by other sites and requests are not prioritized by their timestamps . Thus,

  a site may send a REPLY message to a site and later force a higher priority request from another site to wait.

Problem of deadlocks



Performance Parameters

Size of a request set is √ N.(QUORUM SIZE)

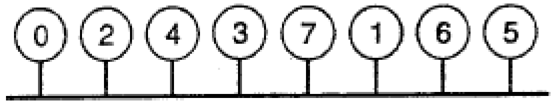
1. An execution of the CR requires √N REQUEST, √ N REPLY and √ N RELEASE messages, thus requiring **total 3 √N** messages per CR execution.

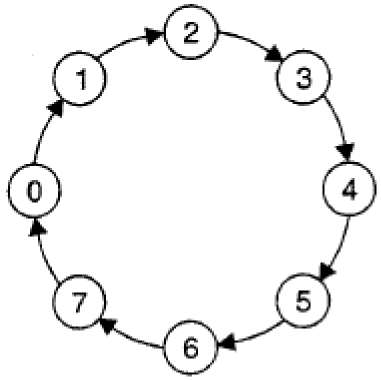
2. Synchronization delay is **2T.**

3. M = K = √N works best.

Token-Based Algorithms

- In software, a logical ring is constructed in which each process is assigned a position in the ring.
- The ring positions may be allocated in numerical order of network addresses or some other means. It does not matter what the ordering is. All that matters is that each process knows who is next in line after itself.





Token-Based Algorithms

- When the ring is initialized, process 0 is given a token. The token circulates around the ring. It is passed from process *k to process k +1* in point-to-point messages.
- When a process acquires the token from its neighbor, it checks to see if it needs to access the shared resource. If so, the process goes ahead, does all the work it needs to, and releases the resources.
- After it has finished, it passes the token along the ring
- If a process is handed the token by its neighbor and is not interested in the resource, it just passes the token along.
- As a consequence, when no processes need the resource, the token just circulates at high speed around the ring.

Token-Based Algorithms

- Correctness:

  Only one process has the token at any instant, so only one process can actually get to the resource.

- Problems:
  - If the token is ever lost, it must be regenerated. In fact, detecting that it is lost is difficult, since the amount of time between successive appearances of the token on the network is unbounded.
  - The algorithm also runs into trouble if a process crashes.

Suzuki–Kasami's Broadcast Algorithm

- Broadcast a request for the token
- Process with the token sends it to the requestor if it does not need it •
- Issues:
- – Current versus outdated requests

  – Determining sites with pending requests

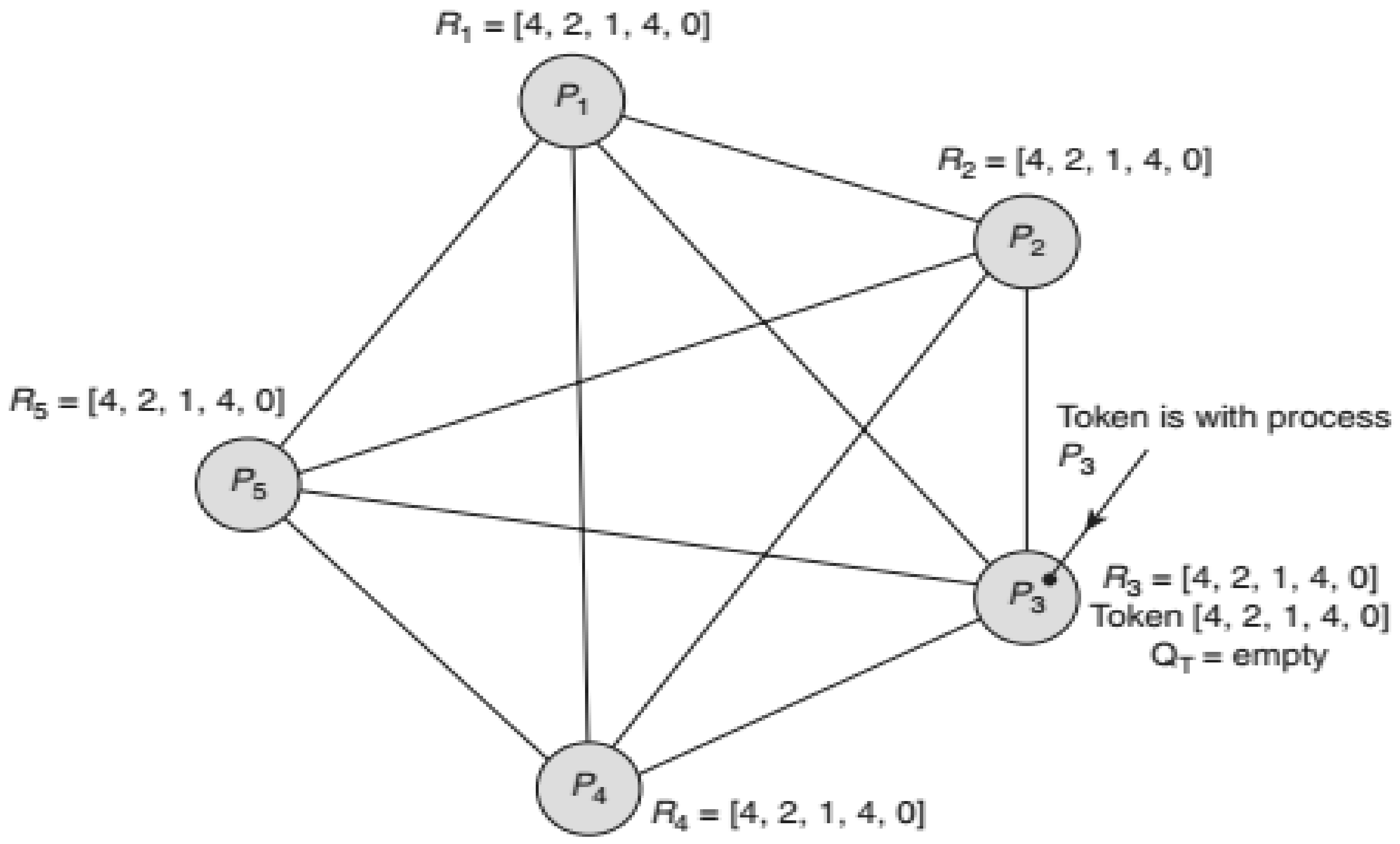
  – Deciding which site to give the token to

- The request message: REQUEST(i, n) : request message from node i for its for its kth critical section execution critical section execution

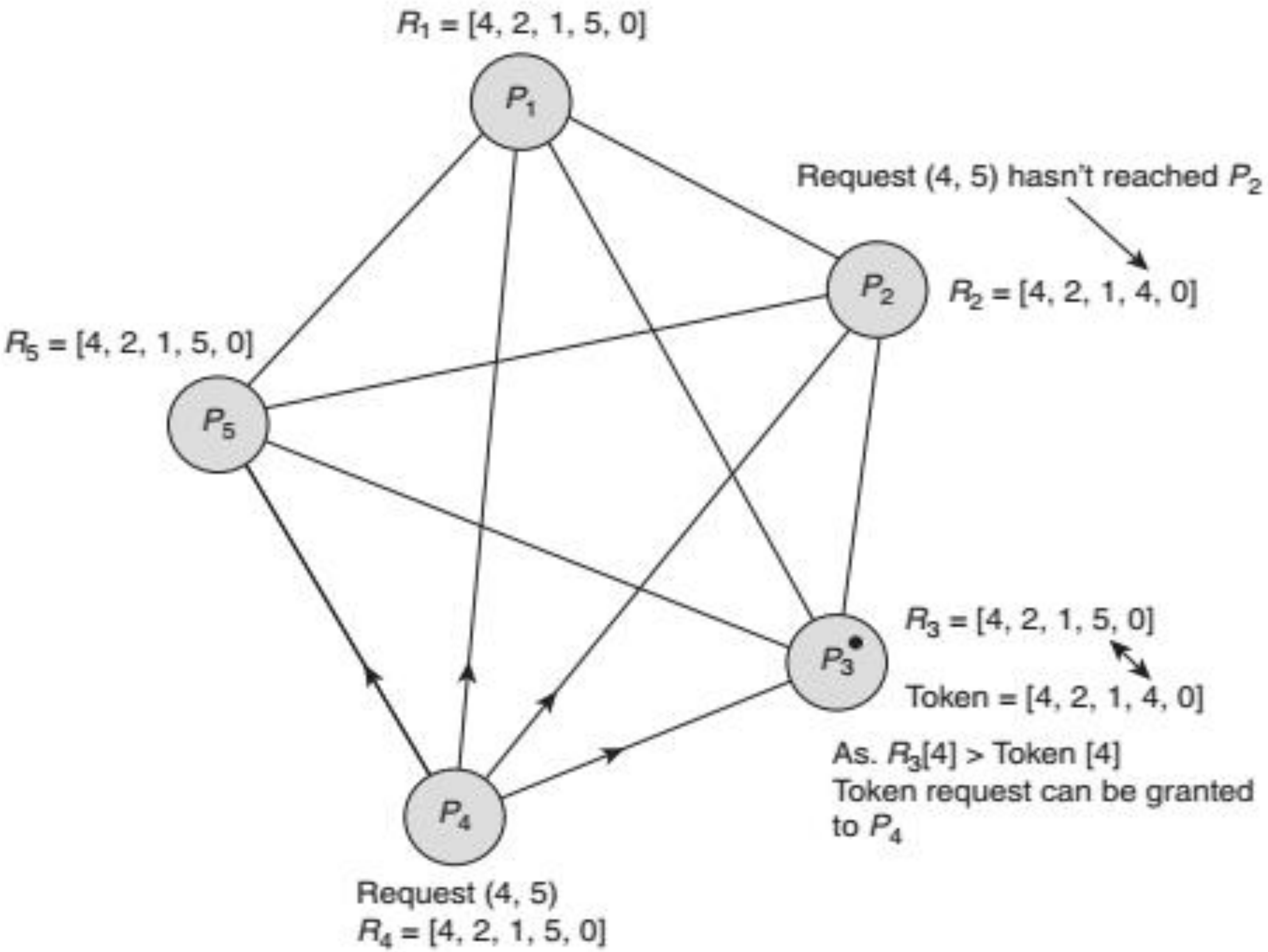
- **<u>Process data structure:</u>**
  - **Request Array Ri[j]; j=1—n**
  - **On receiving REQUEST(i, n) :**
    - **– Set Ri[j] = max(Ri[j], n)**

- **<u>Token data structure:</u>**
  - **Token Array Token[i]**
  - **Queue Qt**

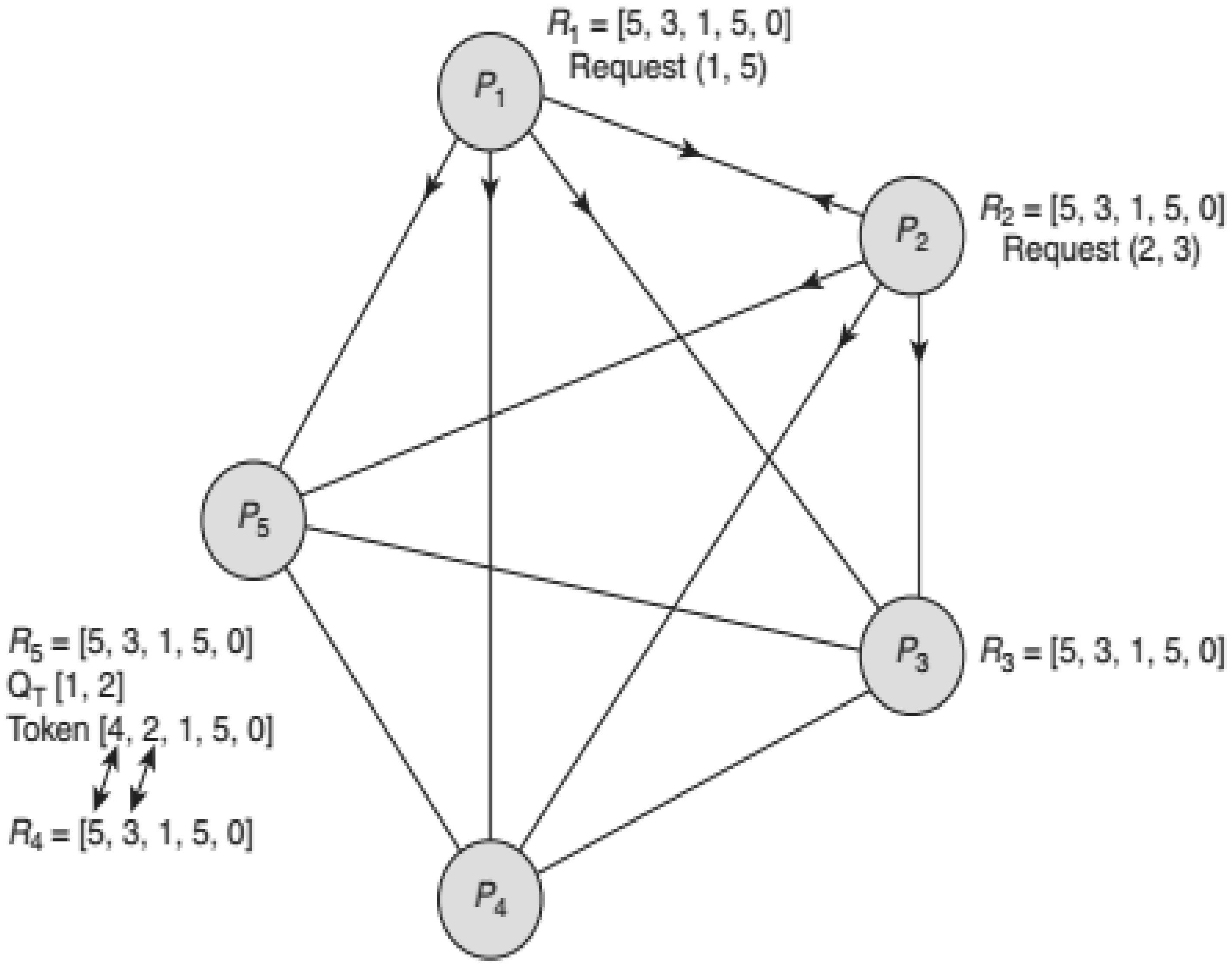
Suzuki–Kasami's Broadcast Algorithm



(a) Initial state: P_3 has the token. No other process is regnesting. Process P_1, P_2, and P_4 had earlier executed CR.



(b) Process P_4 wants to enter CR by broadcasting request (4,5)


R_1 = [5, 3, 1, 5, 0]
Request (1, 5)
P_1
P_2
R_2 = [5, 3, 1, 5, 0]
Request (2, 3)
P_5
P_3
R_3 = [5, 3, 1, 5, 0]
R_5 = [5, 3, 1, 5, 0]
Q_T [1, 2]
Token [4, 2, 1, 5, 0]
R_4 = [5, 3, 1, 5, 0]
P_4

Algorithm

1. The critical region request message (P_i does not possess the token)

   $n = R_i[i] = R_i[i] + 1$;

   REQUEST(i, n) message to all other processes, wait for the token to arrive.

2. P_j on receiving the request message from P_i: REQUEST(i, n)
   - P_j sets $R_j[i] = \max(R_j[i], n)$. /* to take of the outdated requests*/

     If P_j was not executing CR itself and is holding the token

     and if $R_j[i] = \text{Token}[i] + 1$;

     then it sends the token to P_i.

     /* token moves to P_i along with its data structure, Token[ ] and Q_T */

3. On the execution of the CR: P_i
   - Process P_i executes the CR after it has received the token.

4. On the release of the CR: P_i
   - Token[i] = $RN_i[i]$ / * update the token */
   - if ($RN_i[j] = \text{Token}[j] + 1$), for all j, puts the ID of P in the Q_T, if not already there.
   - If queue is nonempty

     Dequeue the top of the Q_T entry, and pass the token to the process at the top of the queue, Q_T

Performance Parameters

1. It is simple and efficient.

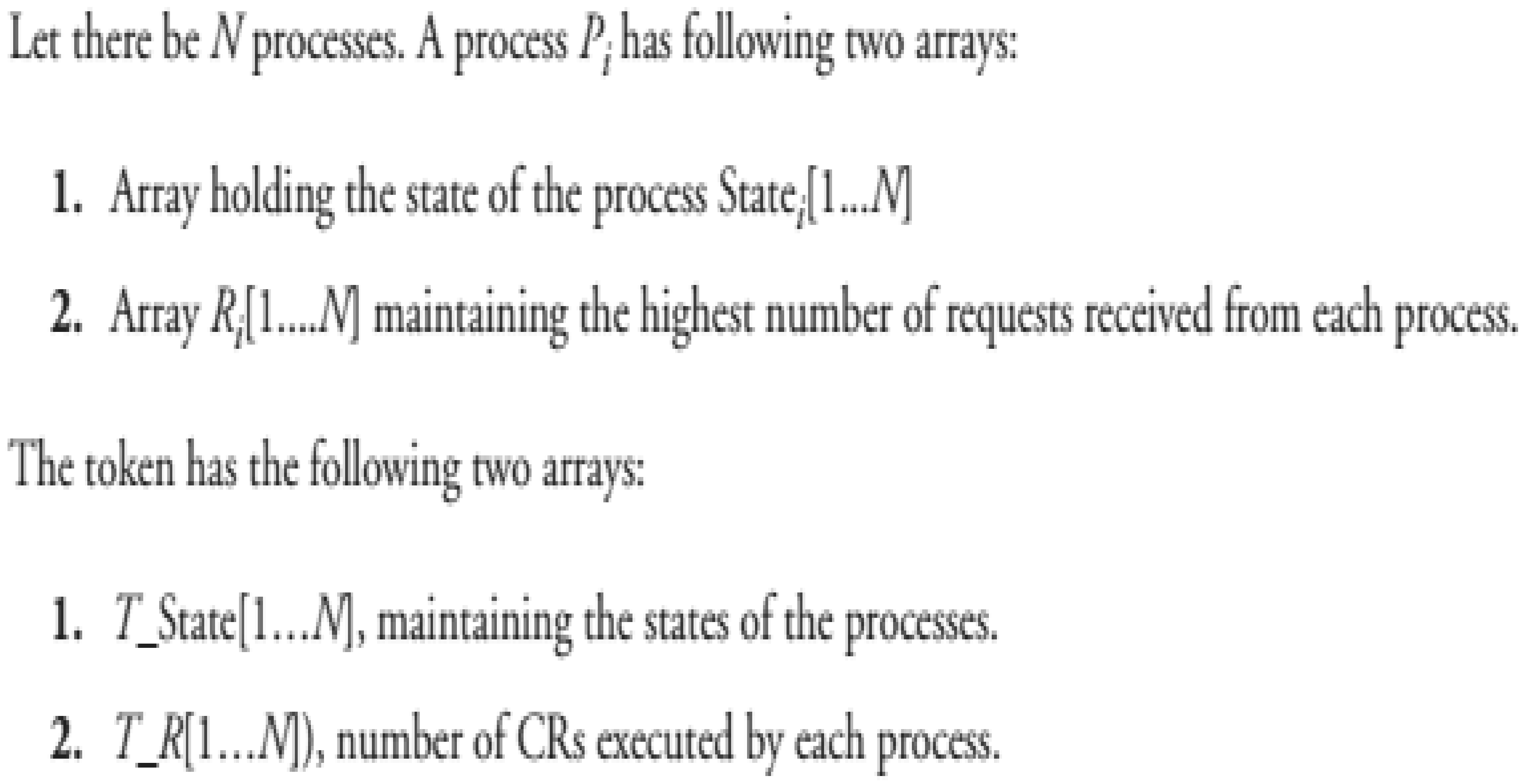
2. The algorithm requires at most N messages to obtain the token to enter CR.

3. The synchronization delay in this algorithm is 0 or T(message delay). Zero synchronization delay, if the process already holds the token or message delay

Singhal's Heuristic Algorithm

- 1. E = Executing and having the token
- 2. H = Holding the token and not executing
- 3. R = Requesting the token
- 4. N = Neutral, none of the above

Data Structures



Let there be N processes. A process P_i has following two arrays:

1. Array holding the state of the process State$_i$[1...N]
2. Array R_i[1....N] maintaining the highest number of requests received from each process.

The token has the following two arrays:

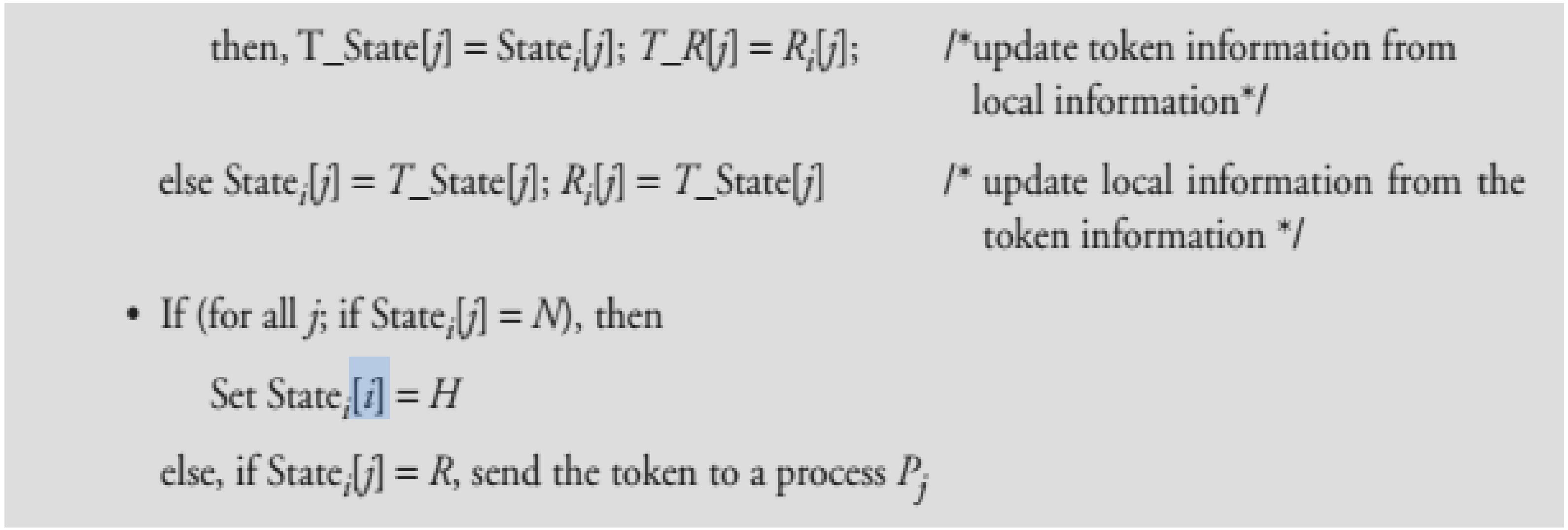
1. T_State[1…N], maintaining the states of the processes.
2. T_R[1…N]), number of CRs executed by each process.

Arrays Initialization

```
For each process, $P_i$ for $i = N...1$ do
    $State_i[j] = N$; for $j = N...k$   do;          /* All processes >= $k$ are set to neutral state = $N$
    $State_i[j] = R$; for $j = k - 1....1$  do        /* All processes < $k$ are assigned the state $R$
Requesting process
    $R_i[j] = 0$;   for $j = 1... N$ do              /* initially there is no request received by any process
    /*Let $P_1$ hold the token initially*/
    $State_1[1] = H$                                  /* $P_1$ has the state hold and it has the token */
For the token
    $T\_State[j] = N$; for $j = 1... N$              /* The token is not being used by any, the array is in state $N$
    $T\_R[j] = 0$; for $j = 1... N$                  /* There is no process that has executed the CR
```

Algorithm

```
1. The critical region request message ($P_i$ does not possess the token)
   • Set $State_i[i] = R$;                         /*requesting*/
     $n = R_i[i] = R_i[i] + 1$;
     REQUEST($i$, $n$) message to all other processes $P_j$ for which  $State_i[j] = R$
     /* This reduces the number of request messages sent as compared to Suzuki-Kasami
        algorithm. */
2. $P_j$ on receiving the request message from $P_i$: REQUEST($i$, $n$)
   • $P_j$ sets $R_j[i] = \max(R_j[i], n)$.            /* to take care of the outdated requests*/
   • If $State_j[j] = N$, then set $State_j[i] = R$.    /* Update the State of $P_i$ in $P_j$ to the requesting
                                                      state*/
   • If $State_j[j] = R$, if $State_i[j] != R$, Set $State_j[i] = R$, send REQUEST($j$, $R_j[j]$) to $P_i$
     /* If $P_j$ is requesting, let $P_i$ know of this request*/
   • If $State_j[j] = E$, then Set $State_j[i] = R$
   • If $State_j[j] = H$, then Set $State_j[i] = R$, $T\_State[i] = R$, $T\_R[i] = n$,  $State_j[j] = N$
     and send the token to process $P_i$.
3. On the execution of the CR: $P_i$
   • Once the token is received, Set $State_i[i] = E$;
   • Process $P_i$ executes the CR after it has received the token.
4. On the Release of the CR: $P_i$
   • Set $State_i[i] = N$ and  $T\_State[i] = N$,
   • For all $j = 1 \ldots N$ do
     if $State_i[j] > T\_State[j]$    /* according to hierarchical order */
```



```
        then, T_State[$j$] = State$_i$[$j$]; $T\_R$[$j$] = $R_i$[$j$];     /*update token information from
                                                                   local information*/
    else State$_i$[$j$] = $T\_$State[$j$]; $R_i$[$j$] = $T\_$State[$j$]     /* update local information from the
                                                                   token information */
• If (for all $j$; if State$_i$[$j$] = $N$), then
        Set State$_i$[$i$] = $H$
    else, if State$_i$[$j$] = $R$, send the token to a process $P_j$
```

Performance Parameters

Performance Parameters

1. The number of REQUEST messages can vary from N/2 (Average value of the identifier) to N (max value).

Raymond's Tree-Based Algorithm

- This algorithm uses a spanning tree to reduce the number of messages exchanged per critical section execution.
- The network is viewed as a graph, a spanning tree of a network is a tree that contains all the N nodes.
- The algorithm assumes that the underlying network guarantees message delivery.
- All nodes of the network are 'completely reliable.

Raymond's Tree-Based Algorithm

- A node needs to hold information about and communicate only to its immediate-neighboring nodes. Similar to the concept of tokens used in token-based algorithms, this algorithm uses a concept of privilege.
- Only one node can be in possession of the privilege (called the privileged node) at any time, except when the privilege is in transit from one node to another in the form of a PRIVILEGE message.
- When there are no nodes requesting for the privilege, it remains in possession of the node that last used it.

The HOLDER Variables

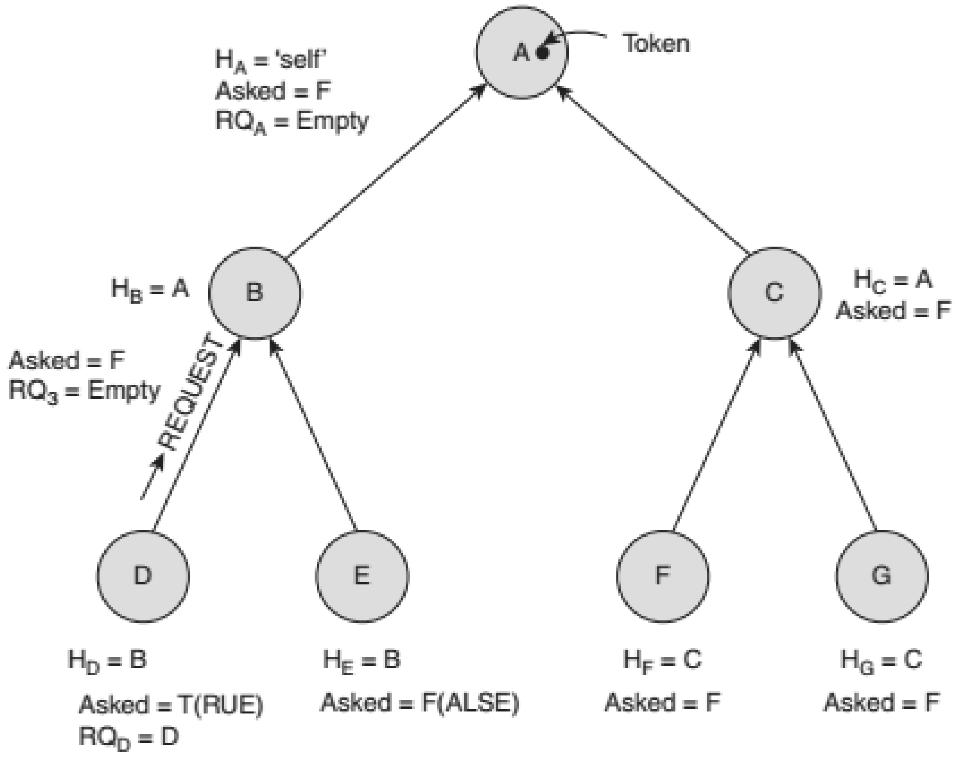
- Each node maintains a HOLDER variable that provides information about the placement of the privilege in relation to the node itself.
- A node stores in its HOLDER variable the identity of a node that it thinks has the privilege or leads to the node having the privilege.
- For two nodes X and Y, if HOLDERX = Y, we could redraw the undirected edge between the nodes X and Y as a directed edge from X to Y.
- For instance, if node G holds the privilege, Figure 4 can be redrawn with logically directed edges.

Data Structures

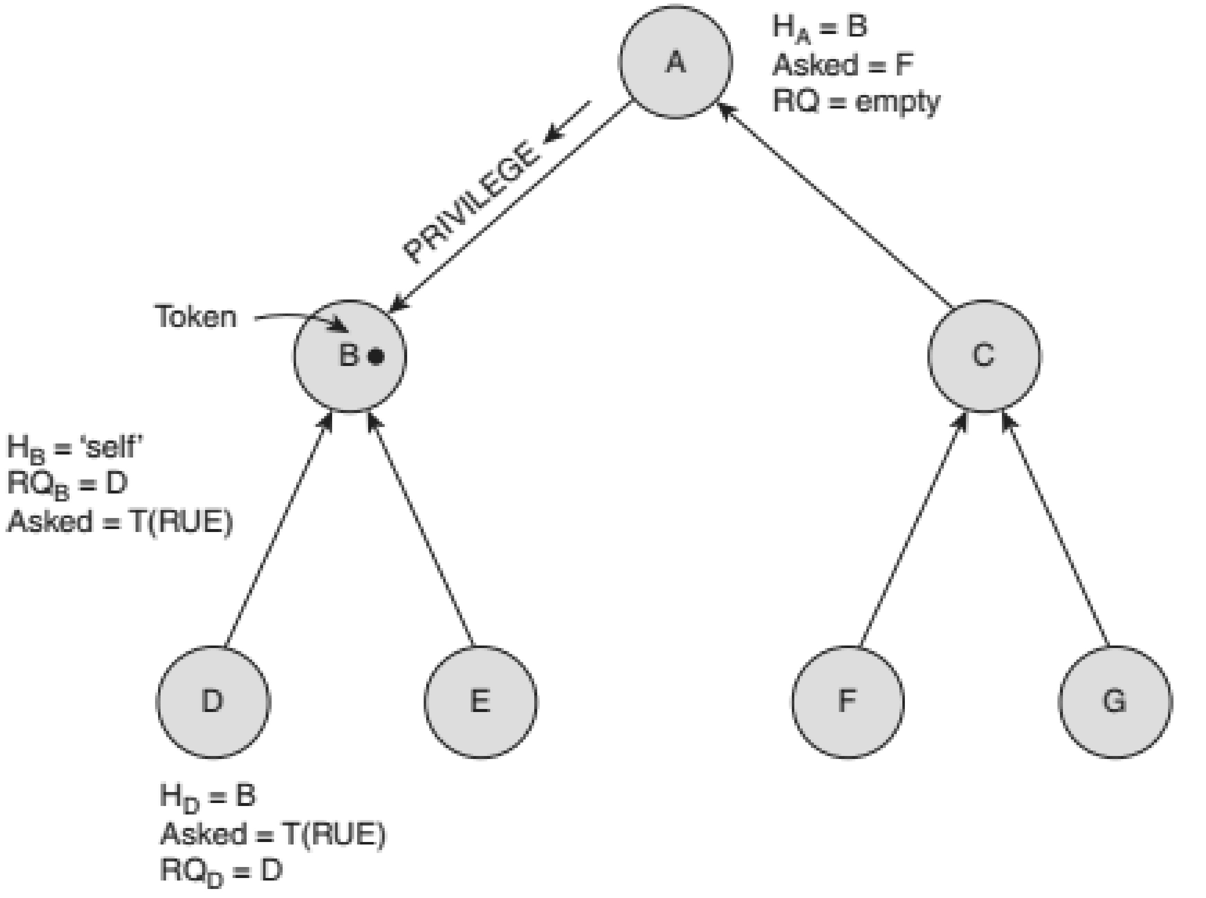
Each node to maintains the following variables:

Variable Name	Possible Values	Comments
HOLDER	"self" or the identity of one of the immediate neighbours.	Indicates the location of the privileged node in relation to the current node.
USING	True or false.	Indicates if the current node is executing the critical section.
REQUEST_Q	A FIFO queue that could contain "self" or the identities of immediate neighbors as elements.	The REQUEST_Q of a node consists of the identities of those immediate neighbors that have requested for privilege but have not yet been sent the privilege .
ASKED	True or false.	Indicates if node has sent a request for the privilege.

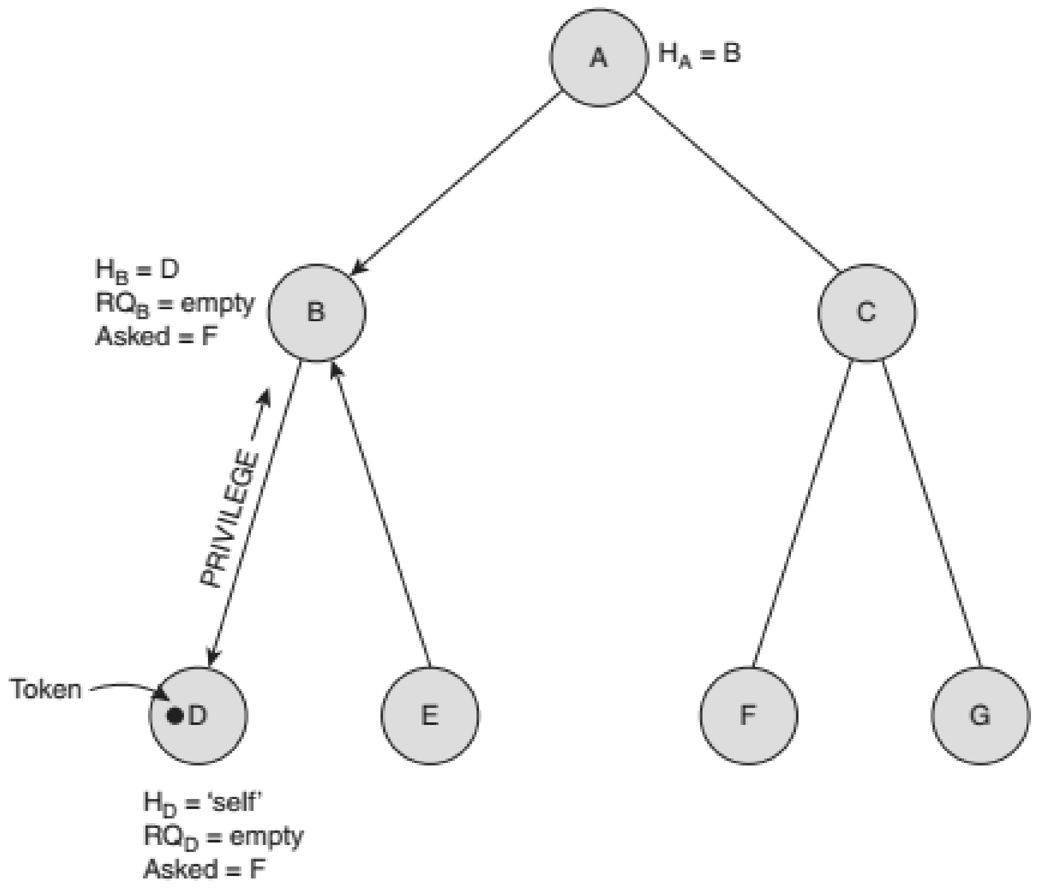
Raymond's Tree-Based Algorithm



(a) Process D sends the REQUEST to process B which forwards it to process A.



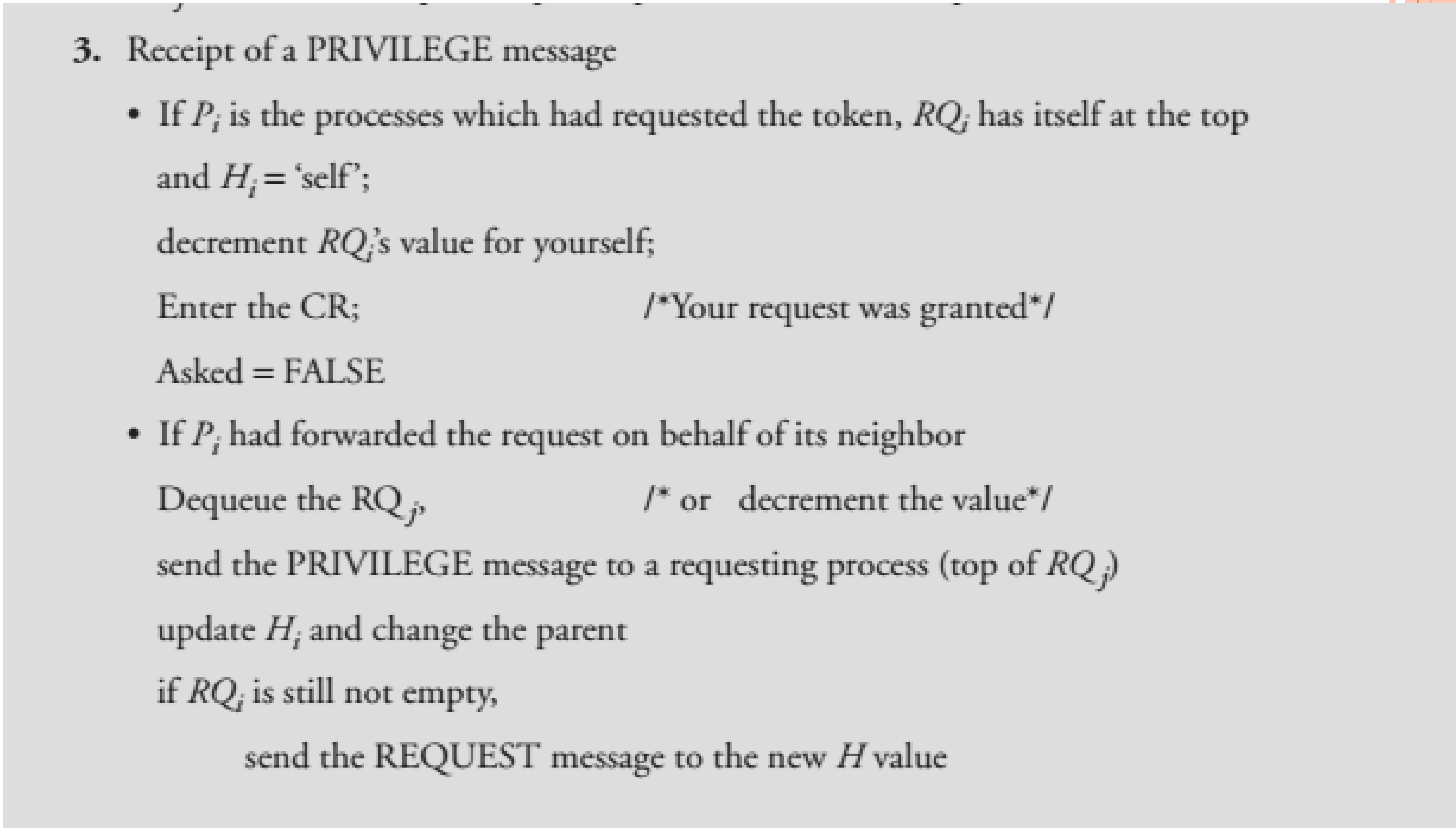
(b) Process B receives the token after granting the PRIVILEGE from process A.



(c) Process D receives the token after granting the PRIVILEGE from process B.

Algorithm

```
1. The critical region request message
   • If $P_i$ does not hold the token;   /* if token held, no need to send REQUEST*/
     and its $RQ_i$ is not empty;         /*process requires the token for itself or for its neighbors*/
     and Asked = FALSE                   /*it has not already sent a REQUEST message, either
         Then increment its $RQ_i$
         check $H_i$ and send the REQUEST to the process held in variable $H_i$;
         Asked = TRUE;
     else  Asked = FALSE;
2. $P_j$ receiving the request message from $P_i$
   • If $P_j$ has the token, i.e., $H_j$ = 'self'
     and if is not executing the CR and its RQ $_j$ is not empty
     and the element at the head of its RQ is not 'self'.
          send the PRIVILEGE message to a requesting process, update its $H_j = i$
     else, if the process is an unprivileged process
     $P_j$ forwards the request to parent process.         /*step 1 is executed then */
```



3. Receipt of a PRIVILEGE message

```
• If $P_i$ is the processes which had requested the token, $RQ_i$ has itself at the top
  and $H_i$ = 'self';
  decrement $RQ_i$'s value for yourself;
  Enter the CR;                       /*Your request was granted*/
  Asked = FALSE
• If $P_i$ had forwarded the request on behalf of its neighbor
  Dequeue the $RQ_j$,                 /* or   decrement the value*/
  send the PRIVILEGE message to a requesting process (top of $RQ_j$)
  update $H_i$ and change the parent
  if $RQ_i$ is still not empty,
      send the REQUEST message to the new $H$ value
```

```
            Asked = TRUE;
      Else Asked = FALSE;
4.    The execution of the CR
      • The process $P_i$ gets to execute the CR if the process $P_i$ has the token available
            only if its own ID is at the head of its RQ;
5.    $P_i$ exiting CR
      • if $RQ_i$ is nonempty then
            Dequeue $RQ_i$; Let us call it $P_d$        /* get the process who had send the request*/
            send the token to this process $P_d$;
            change the $H_i = d$; Asked = FALSE;
      • if RQ is still not empty, send REQUEST to the parent process, Asked = TRUE;
```

Performance Parameters

The algorithm exchanges only O(log N) messages under light load and four messages under heavy load
to execute the CR, where N is the number of nodes in the network.

References

- Andrew S. Tanenbaum and Maarten Van Steen, Distributed Systems: Principles and Paradigms, 2nd edition, Pearson Education.
- **Ajay Kshemkalyani and Mukesh Singhal** Distributed Computing: Principles, Algorithms, and Systems,Cambridge University Press