# Distributed Computing (2020)

## Synchronization

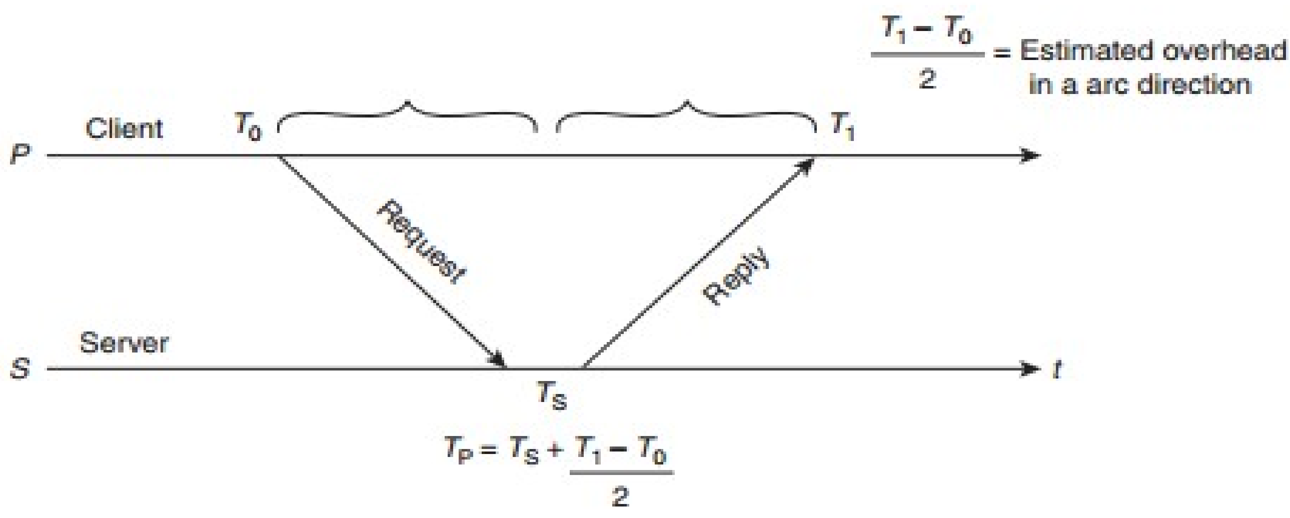
# Christian' Algorithm

**Algorithm**

Let $S$ be the timeserver and $T_s$ be its time.

- Process $P$ requests the time from $S$.
- After receiving the request from $P$, $S$ prepares a response and appends the time $T_s$ from its own clock and sends it back to $P$
- $P$ then sets its time to be $T_p = T_s + RTT/2$

Figure 7.1(a) illustrates the idea behind the algorithm more clearly.



# Berkeley's Algorithm

**Algorithm**

Elect* the master amongst $N$ nodes. Let $T_m$ be the time estimate of the master's clock.

Let $t[i]$ contain the time at each i slave at master

**If master**

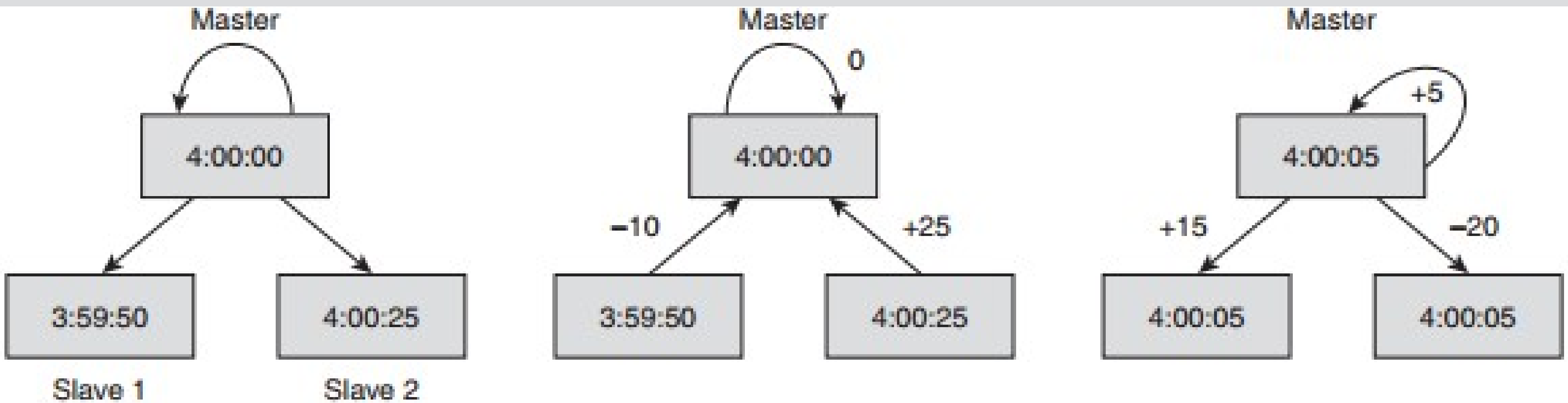
send its $T_m$ along with query for $t[i]$ to slaves; /* for $i = 1.....N-1$*/

Adjust = Sum($t[i]$)/$N$ /* take average including masters

send offset[$i$] = Adjust-$t[i]$ to each slave;

**If slave**

sends query response as $t[i] = T_m - T[i]$; /* for $i = 1..N-1$; calculates the difference between master timestamp $T_m$, and its own timestamp $T$ */




(a) Query-poll (b) Response (c) Adjust

# Lamport's Logical(scalar) Clock

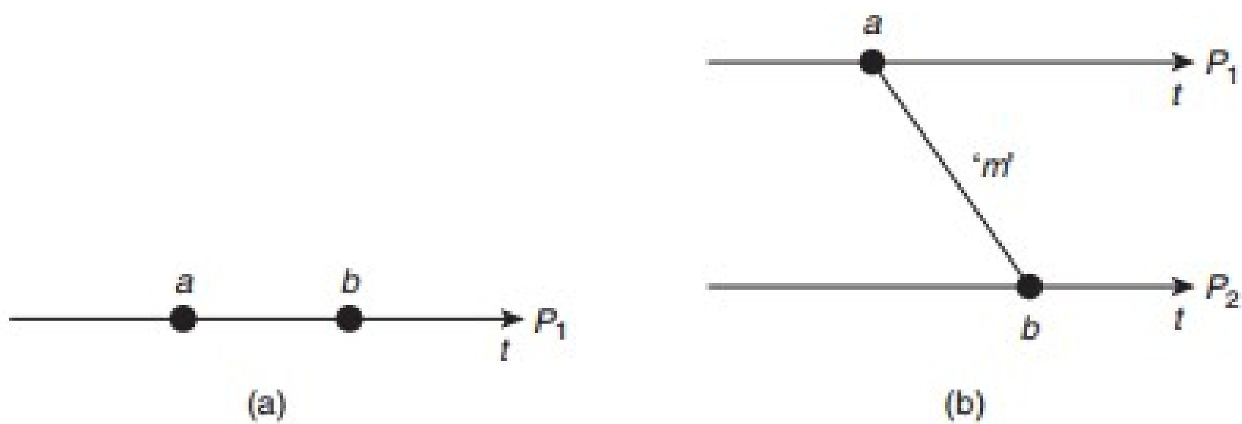
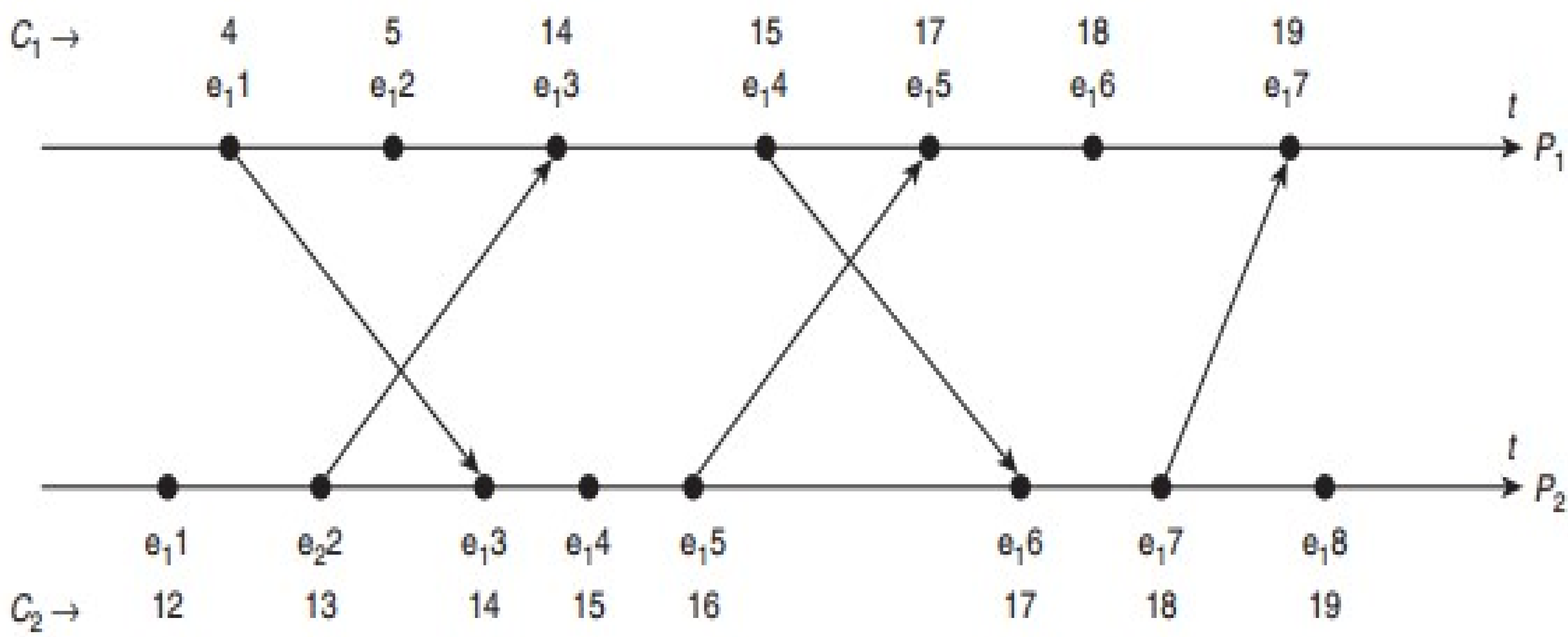



**Figure 7.4** (a) Events $a$ and $b$ are events of process $P_1$. (b) Events $a$ and $b$ are send and receive events of process $P_1$ and $P_2$, respectively, of the same message $m$.

If $a$ and $b$ are two events, $a \rightarrow b$ would mean $a$ happened before $b$:

1. If $a$ and $b$ are events internal in the same process.
2. If $a$ is the event corresponding to the sending of message $m$ in one process, and $b$ is the event corresponding to receiving of the same message $m$ in the other process.

# Lamport's Logical Clock



# Vector Timestamp Ordering

*Each process $P_i$ maintains a vector of integer clock with elements of the vector being N, N is the number of processes.*

**Algorithm**

For every local event,

- $V_i[i] = V_i[i] + 1;$ /* increment only *i*th element in the vector

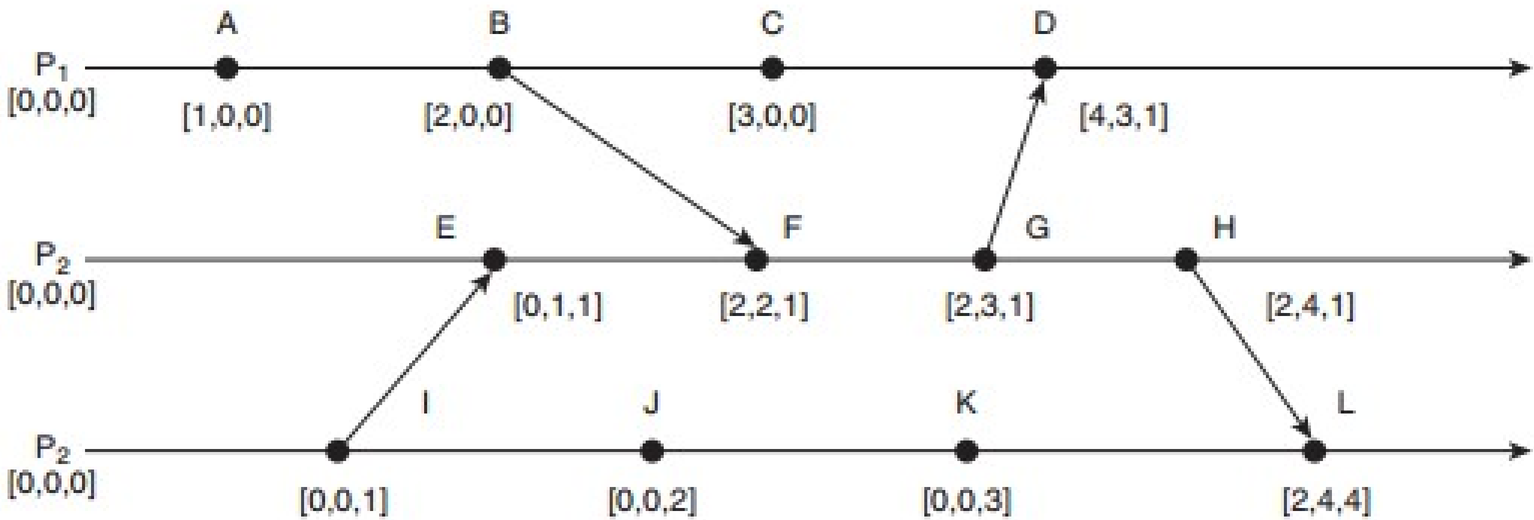
For every send event at $P_i$,

- $V_i[i] = V_i[i] + 1;$ /* increment only *i*th element in the vector
- Send the Message + $V_i$ to the receiver;

For every received message at $P_i$ from $P_j$

- $V_i[i] = V_i[i] + 1;$
- $V_i[j] = \max(V_i[j], V_j[j]);$ /* for $j != i$

# Vector Timestamp Ordering




Causal events:

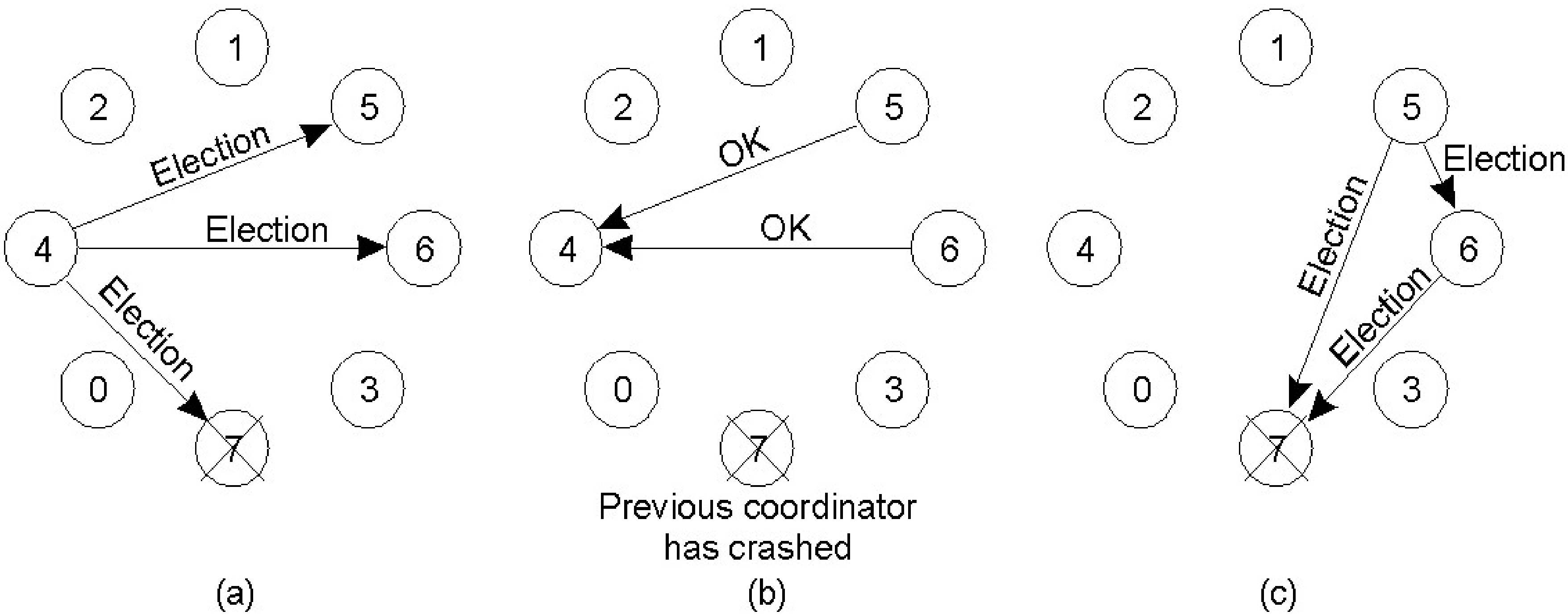
H → G : [0,0,1] < [2,3,1] /* any one of the element is smaller*/
F → L : [2,2,1] < [2,4,4]
A → G : [1,0,0] < [2,3,1] /* all elements are smaller*/

Concurrent events:

C and F : [3,0,0] || [2,2,1) /* 3 > 2 ; 0 < 1*/
F and J : [2,2,1] || [0,0,2] /* 2 > 0 ; 1 < 2*/
I and C : [0,0,1] || [3,0,0] /* 0 < 3 ; 1 > 0*/

# Coordinator Selection
# Election Algorithms

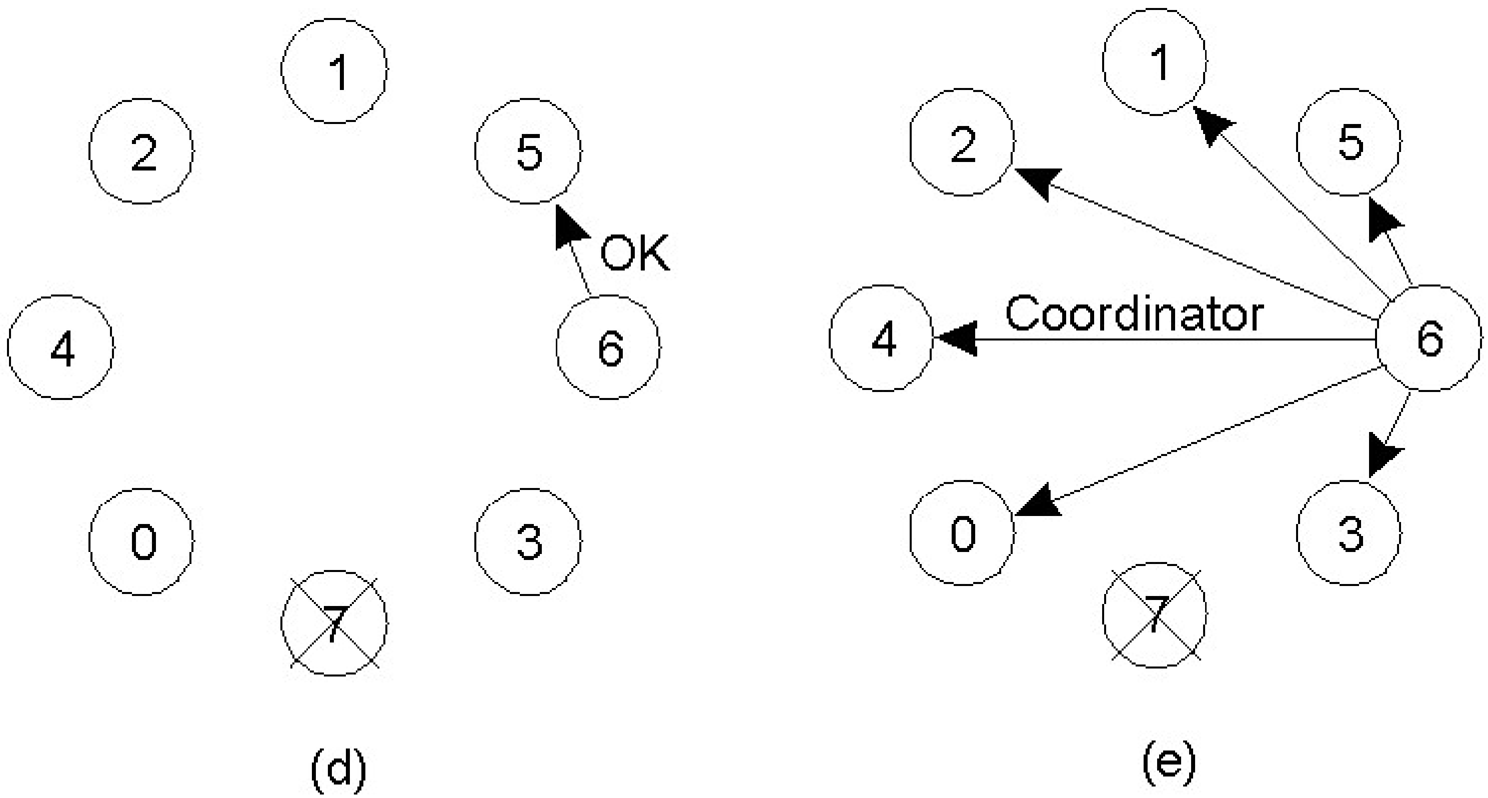
# The Bully Algorithm (1)




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

# Global State (3)




d) Process 6 tells 5 to stop

e) Process 6 wins and tells everyone

## Algorithm

1. Start of ELECTION
   - Process $P_i$ initiates an election if it just recovered from failure or it notices that the coordinator has failed.
   - $P_i$ sends ELECTION($i$) messages to all processes $P_j$ with higher IDs ($j > i$)
     - o Awaits OK messages.
     - o If timeout /*declares itself as the coordinator process.*/
       - o Sends COORDINATOR($i$) message to all lower ID nodes $P_j$ ($j < i$))

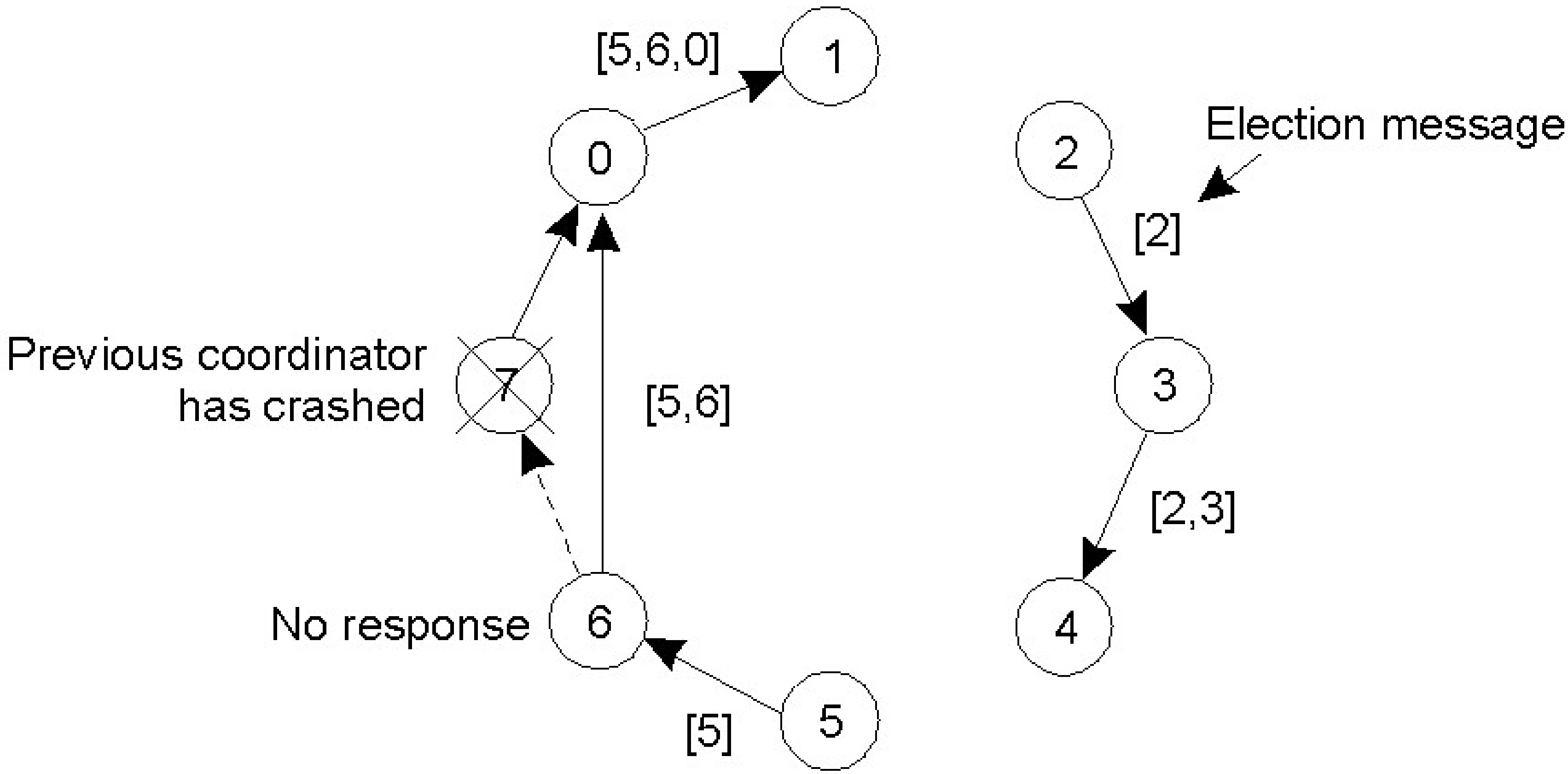
       /* As no higher ID process responded, it is the highest ID process by default */
2. $P_j$ receives ELECTION($i$) message
   - If $j > i$ , send OK to $P_i$;
   - Send ELECTION($j$) messages to higher ID process.

3. $P_i$ receives OK from $P_j$
   - Stop ELECTION($i$) process, /*higher ID process has been found*/
     - o Awaits, COORDINATOR($j$) message from any $P_j$ ($j > i$)
     - o Else sends COORDINATOR($i$) message to all lower ID nodes $P_j$ ($j < i$))

     /* because no higher ID process responded, so it is the highest ID process*/

     /* Some versions of algorithm suggest to restart the ELECTION process instead */
4. $P_i$ receives COORDINATOR($j$) from $P_j$
   - It stops its election process and $P_j$ is termed as elected coordinator node

# A Ring Algorithm




Election algorithm using a ring.

**Algorithm**

1. Start of ELECTION
   - $P_i$ initiates an election if it just recovered from failure or it notices that the coordinator has failed.
   - The ELECTION[$i$] message along with the identifier of the node is sent to the next downstream process alive and forwarded along the ring by each process.
2. Receiving an ELECTION message
   - If $P_i$ receives the its own ELECTION message, it
     - Removes the message and selects the highest ID process from the message.
     - Sends a COORDINATOR message with ID of the highest process
   - Else, if $P_j$ receives the process, i.e., ($j! = i$), it
     - Appends its own ID into the message.
     - Forwards the ELECTION [] message to the next downstream node.
3. Receiving COORDINATOR message
   - If $P_i$ receives the COORDINATOR message, it
     - Removes this message from the ring. /* the message has traversed the ring*/
     - Marks the election process to be over.
   - For other processes $P_j$ ($j! = i$)
     - The ID of the coordinator is noted.
     - Forward the message to downstream process.

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

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

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

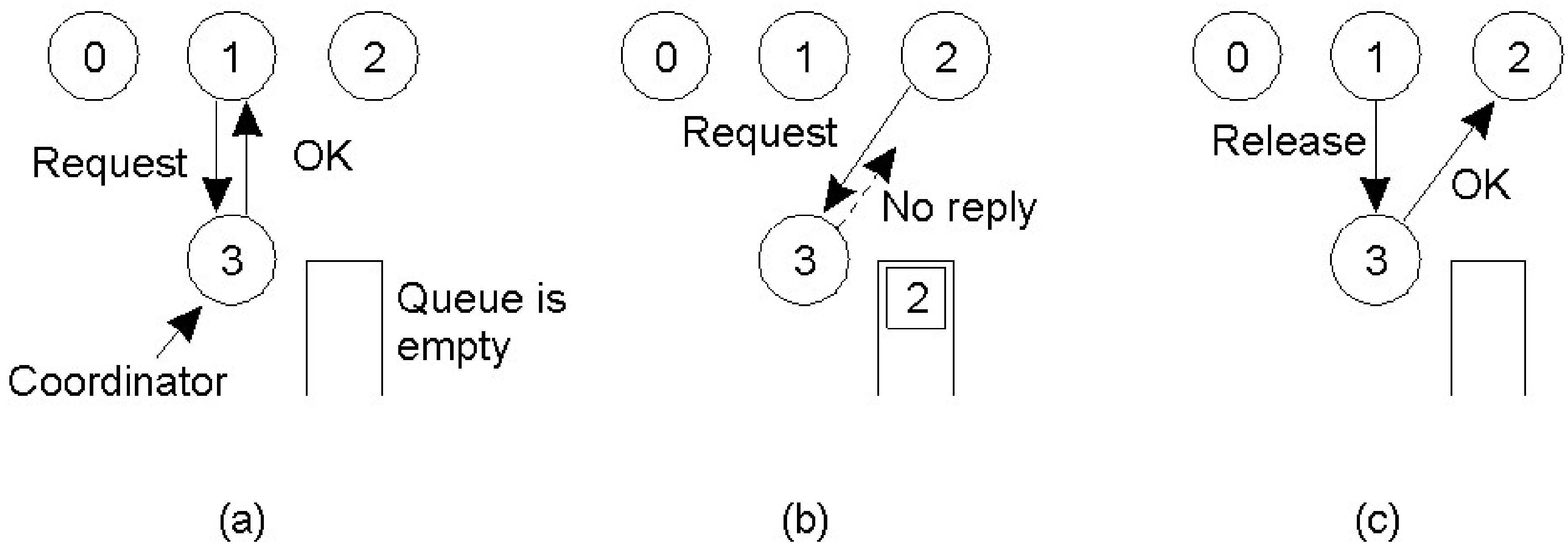
4. Response time: Time interval from a request send to its CR execution completed.

# 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

# Mutual Exclusion: A Centralized Algorithm




a) Process 1 asks the coordinator for permission to enter a critical region. Permission is granted
b) Process 2 then asks permission to enter the same critical region. The coordinator does not reply.
c) When process 1 exits the critical region, it tells the coordinator, when then replies to 2

# Performance Parameters

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

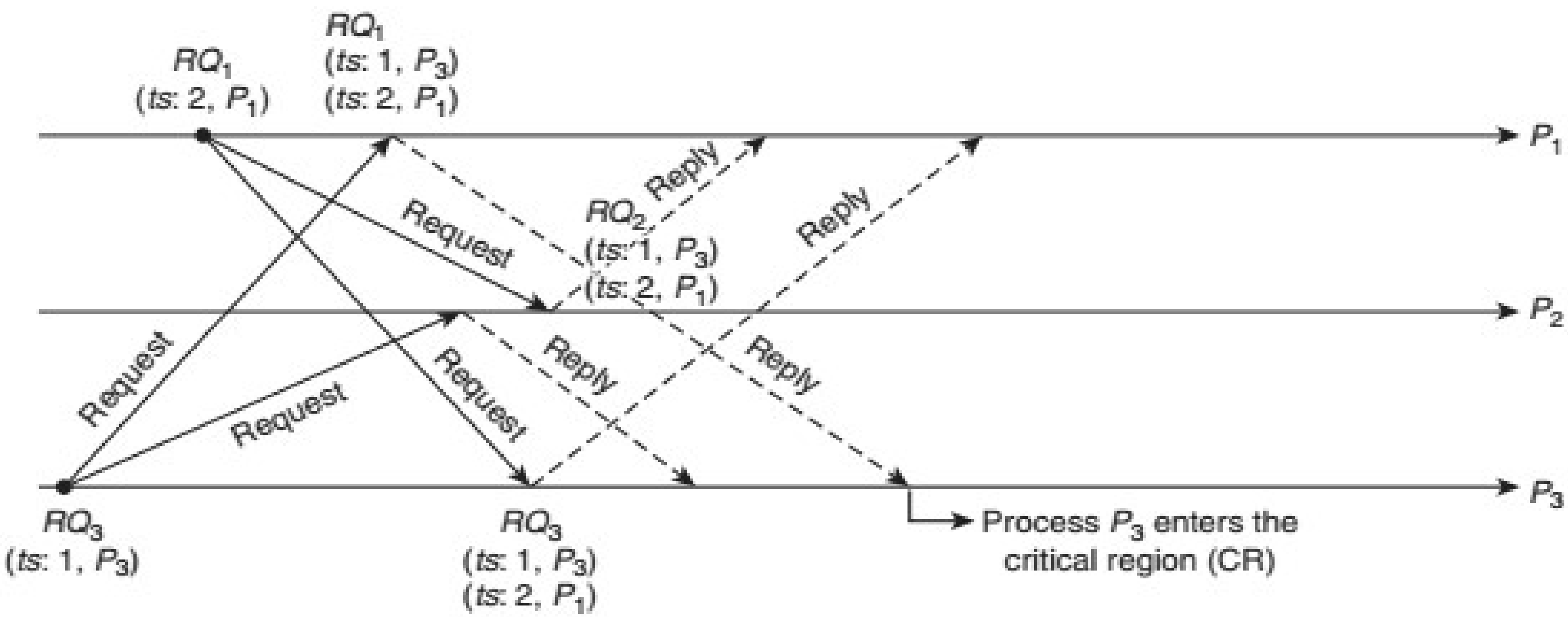
2. It guarantees no starvation.

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

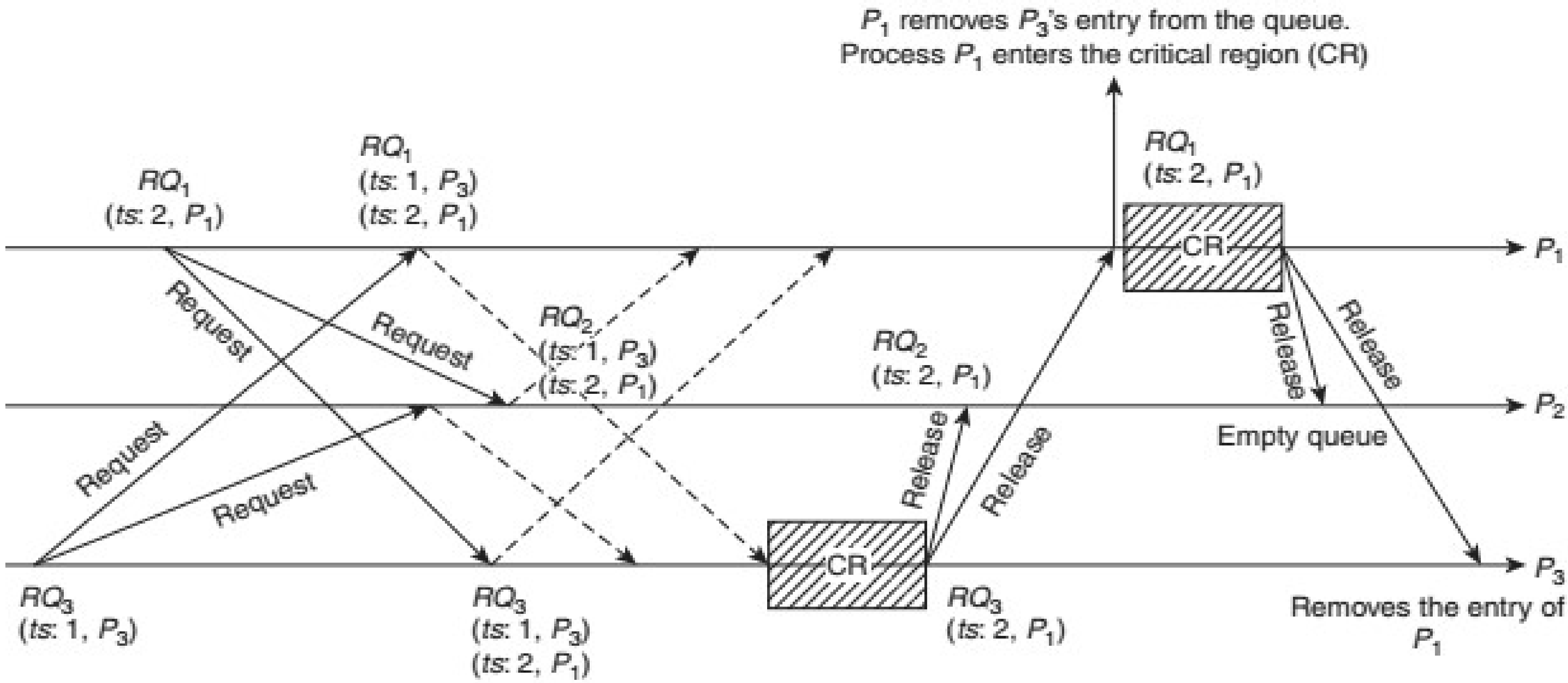
4. It has a single point of failure.

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

# Lamport's Distributed Mutual Algorithm




(a)



P1 removes P3's entry from the queue.
Process P1 enters the critical region (CR)
RQ1
(ts: 2, P1)
RQ1
(ts: 1, P3)
(ts: 2, P1)
RQ1
(ts: 2, P1)
CR
P1
Request
Request
RQ2
(ts: 1, P3)
(ts: 2, P1)
RQ2
(ts: 2, P1)
Release
Release
P2
Empty queue
Request
Request
Release
Release
CR
P3
RQ3
(ts: 1, P3)
RQ3
(ts: 1, P3)
(ts: 2, P1)
RQ3
(ts: 2, P1)
Removes the entry of
P1


(b)

## 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

     (a) dequeue top of the queue $RD_j$,

     /* looks out for requests which came while $P_i$ was in CR*/

     (b) $P_j$ sends a REPLY message to only this dequeued process.

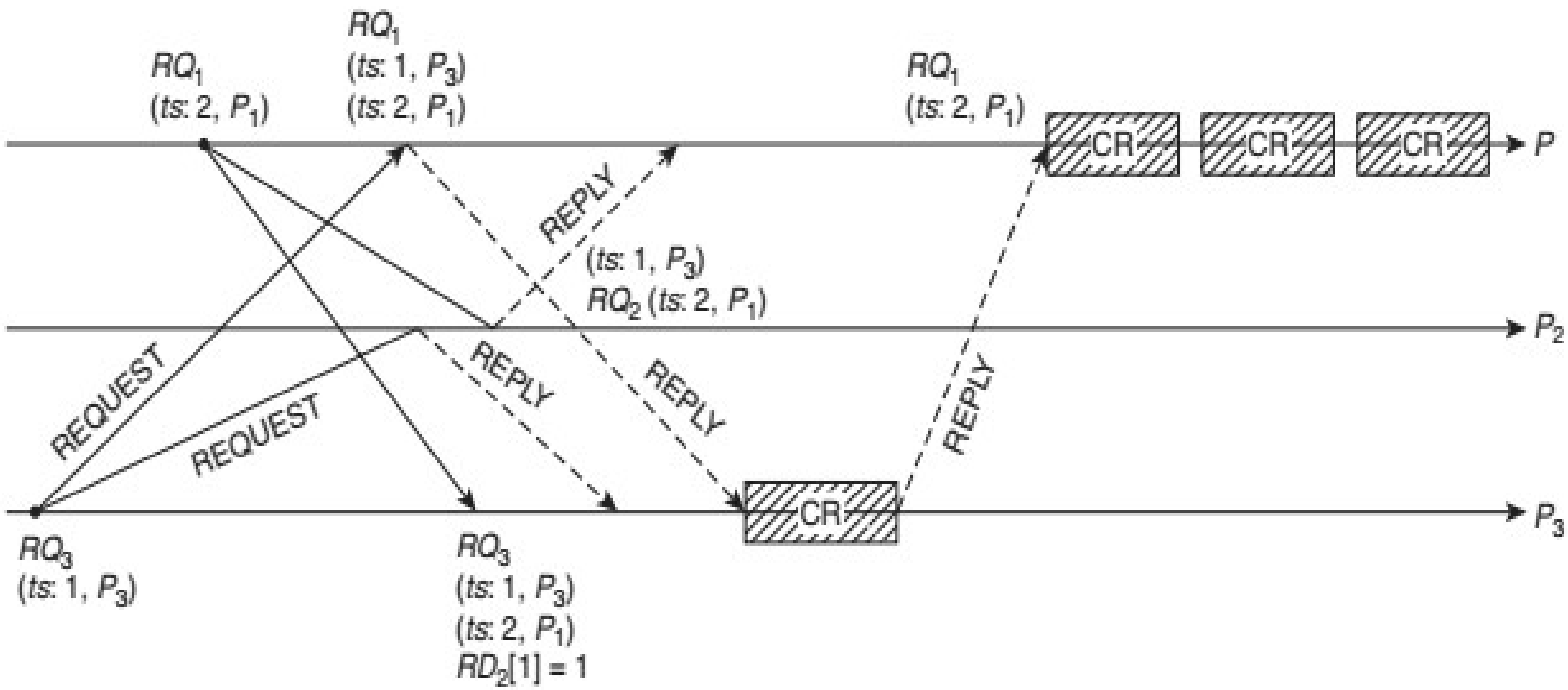
     (c) Set voted$_j$ to be TRUE
   - If queue $RD_j$ was empty

     (a) voted$_j$ = false

# 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.

# Ricart–Agrawala Algorithm



# Performance Parameters

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.

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

# 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*/

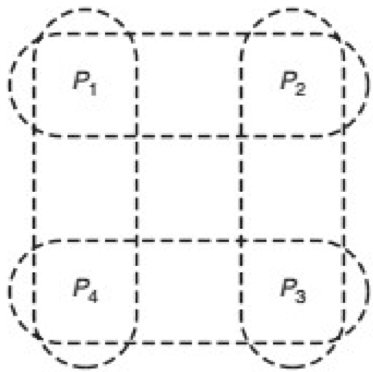
**2.** $(P_i \in R_i)$ , (for all $i$, $1 \leq i \leq N$)

/*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 $1 \leq i, j \leq N$. )




N = 4, R = 4, K = 2

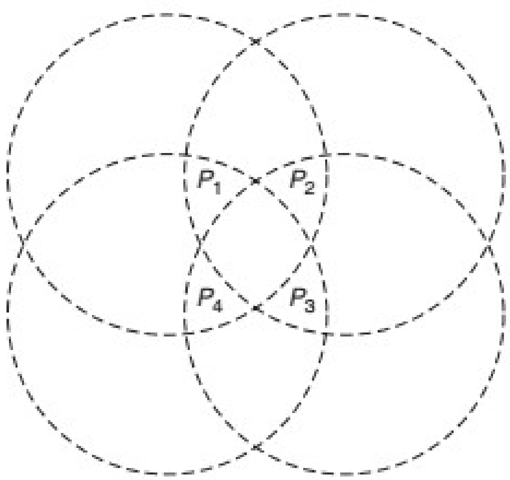
$R_1 = \{P_1, P_2\}$

$R_2 = \{P_2, P_3\}$

$R_3 = \{P_3, P_4\}$

$R_4 = \{P_4, P_1\}$

(a)



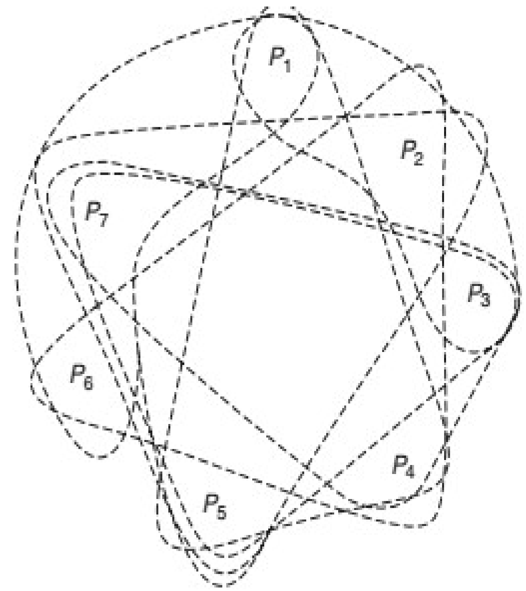

N = 4, R = 4, K = 3

$R_1 = \{P_1, P_2, P_4\}$

$R_2 = \{P_2, P_1, P_3\}$

$R_3 = \{P_3, P_2, P_4\}$

$R_4 = \{P_4, P_1, P_3\}$




N = 7, R = 7, K = 3

$R_1 = \{P_1, P_2, P_3\}$

$R_2 = \{P_2, P_4, P_6\}$

$R_3 = \{P_3, P_5, P_7\}$

$R_4 = \{P_1, P_4, P_5\}$

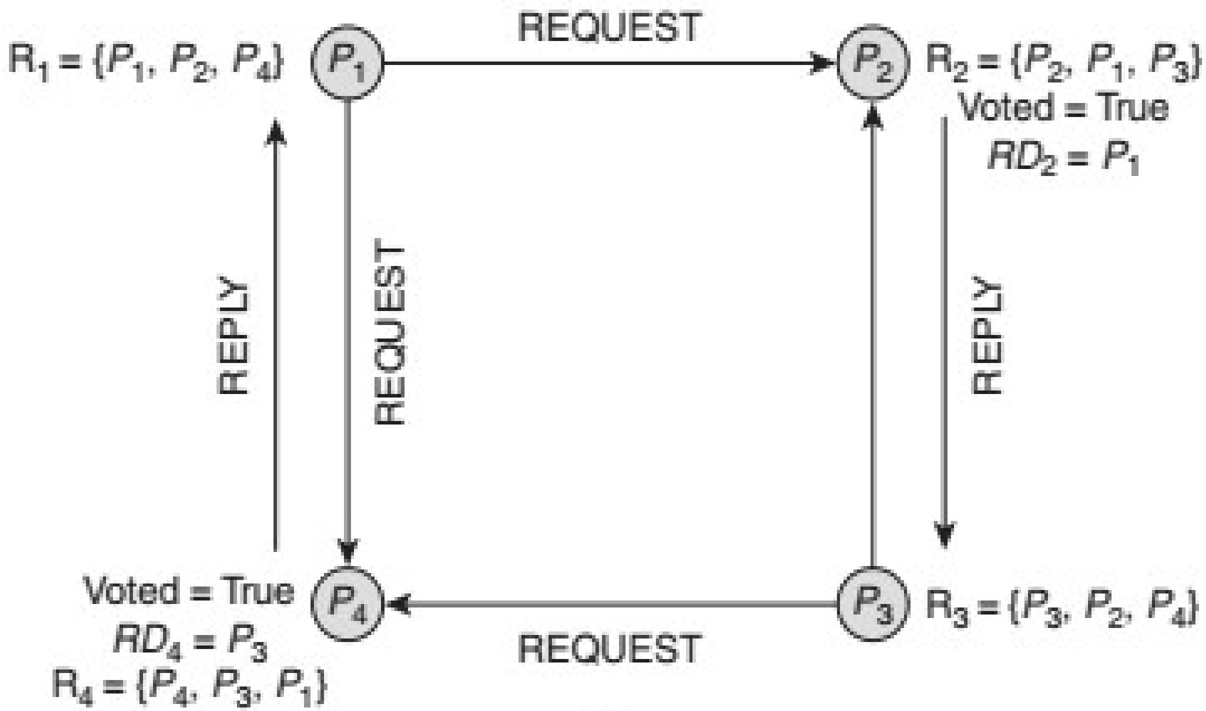
$R_5 = \{P_5, P_2, P_7\}$

$R_6 = \{P_6, P_1, P_7\}$

$R_7 = \{P_7, P_3, P_4\}$

(c)

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


REQUEST
$R_1 = \{P_1, P_2, P_4\}$
$P_1$
$P_2$
$R_2 = \{P_2, P_1, P_3\}$
Voted = True
$RD_2 = P_1$
REPLY
REQUEST
REPLY
Voted = True
$RD_4 = P_3$
$R_4 = \{P_4, P_3, P_1\}$
$P_4$
$P_3$
$R_3 = \{P_3, P_2, P_4\}$
REQUEST

(d)

**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
     - then REQUEST from $P_i$ is deferred and pushed in the queue $RD_j$.
     - 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

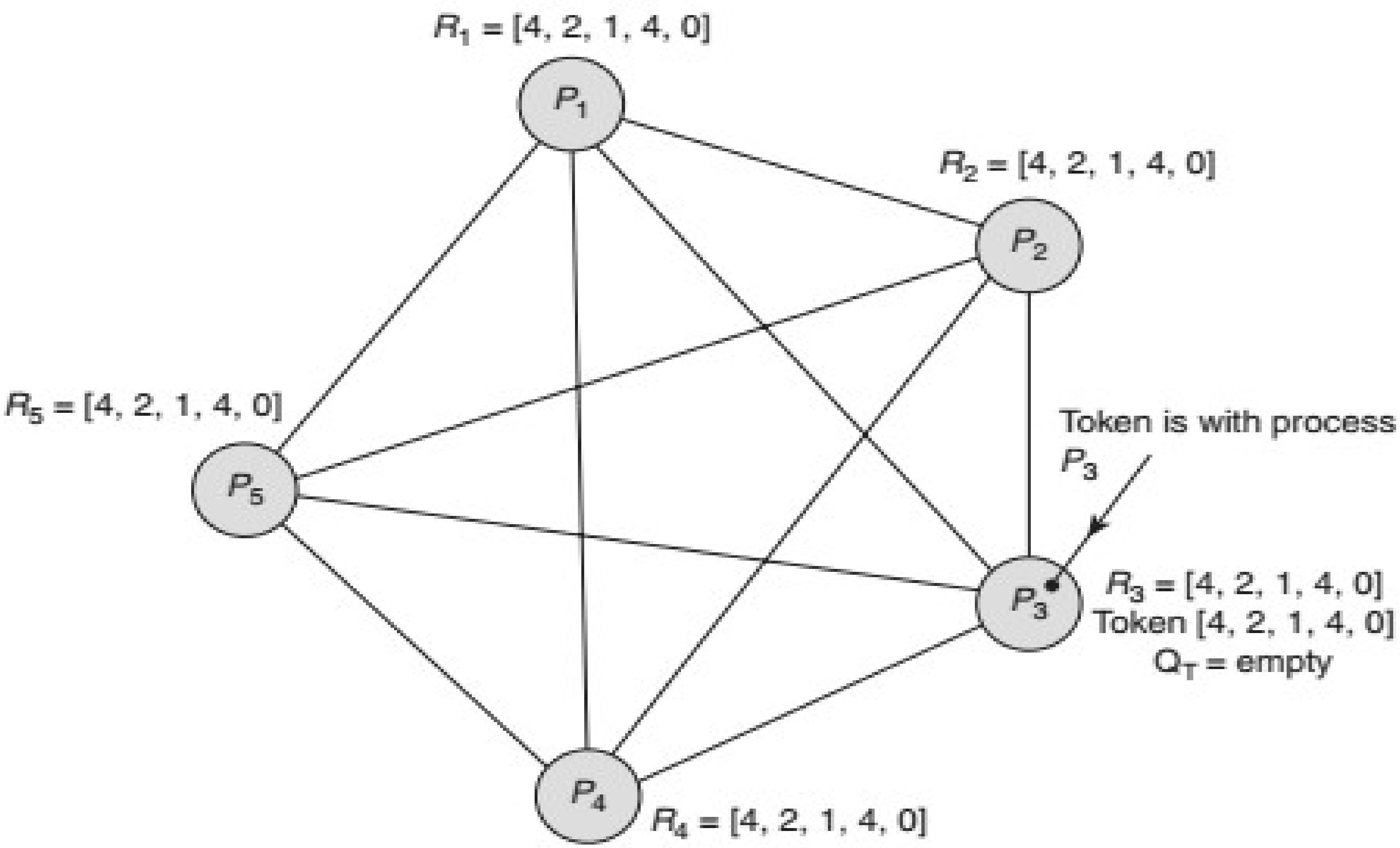
# Performance Parameters

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

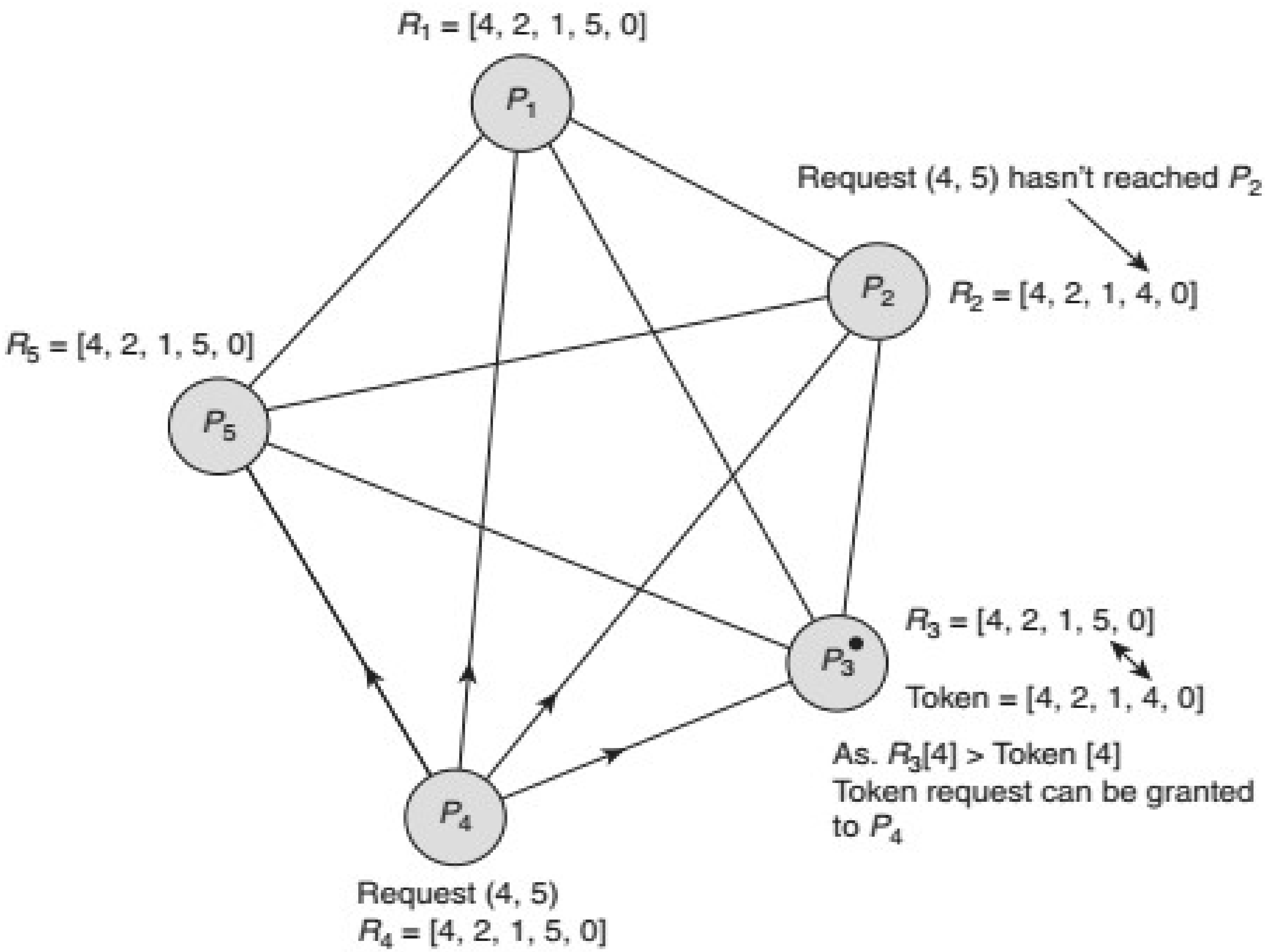
2. Synchronization delay is 2T.

3. M = K = N works best.

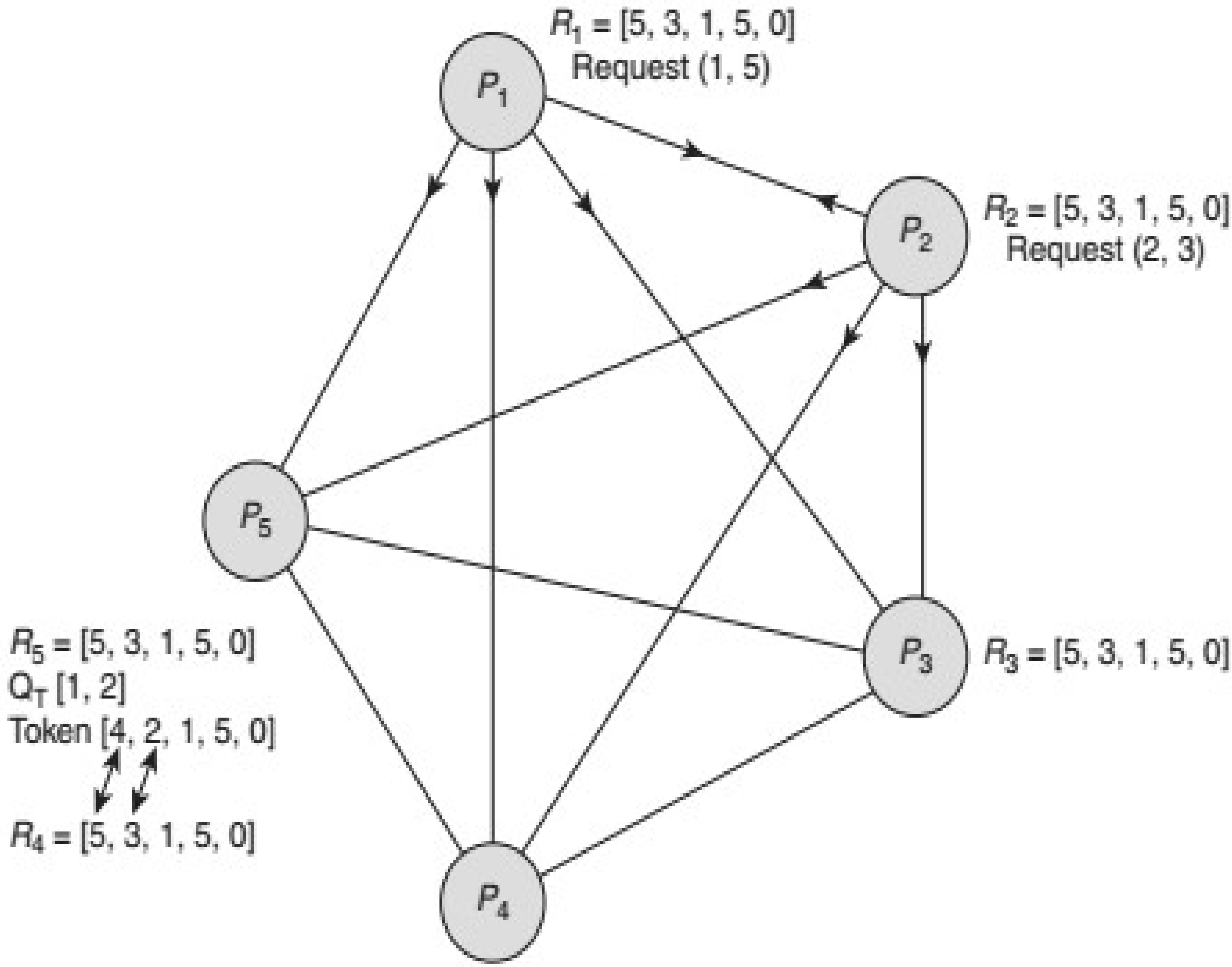
# Token-Based Algorithms
# Suzuki–Kasami's Broadcast Algorithm




(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)
$R_2$ = [5, 3, 1, 5, 0]
Request (2, 3)
$P_1$
$P_2$
$P_3$
$P_4$
$P_5$
$R_3$ = [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]

**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: $P_i$

   - $\text{Token}[i] = RN_i[i]$ / * 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. Zero synchronization delay, if the process already holds the token.

# 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

$\text{State}_i[j] = N$; for $j = N...k$ do; /* All processes >= $k$ are set to neutral state = $N$

$\text{State}_i[j] = R$; 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*/

$\text{State}_1[1] = H$ /* $P_1$ has the state hold and it has the token */

For the token

$T\_\text{State}[j] = N$; 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$_i$[$i$] = $H$

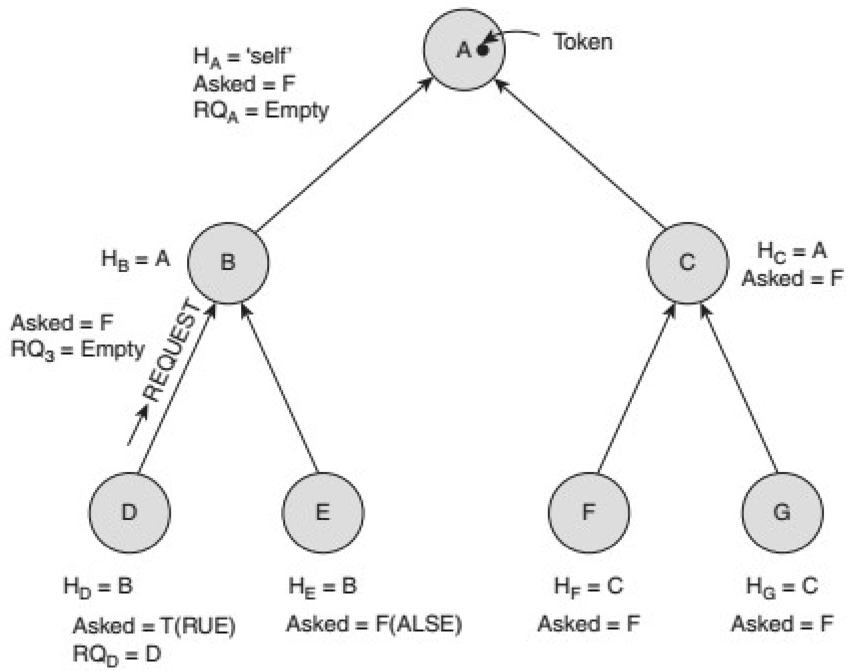
  else, if State$_i$[$j$] = $R$, send the token to a process $P_j$

# Performance Parameters

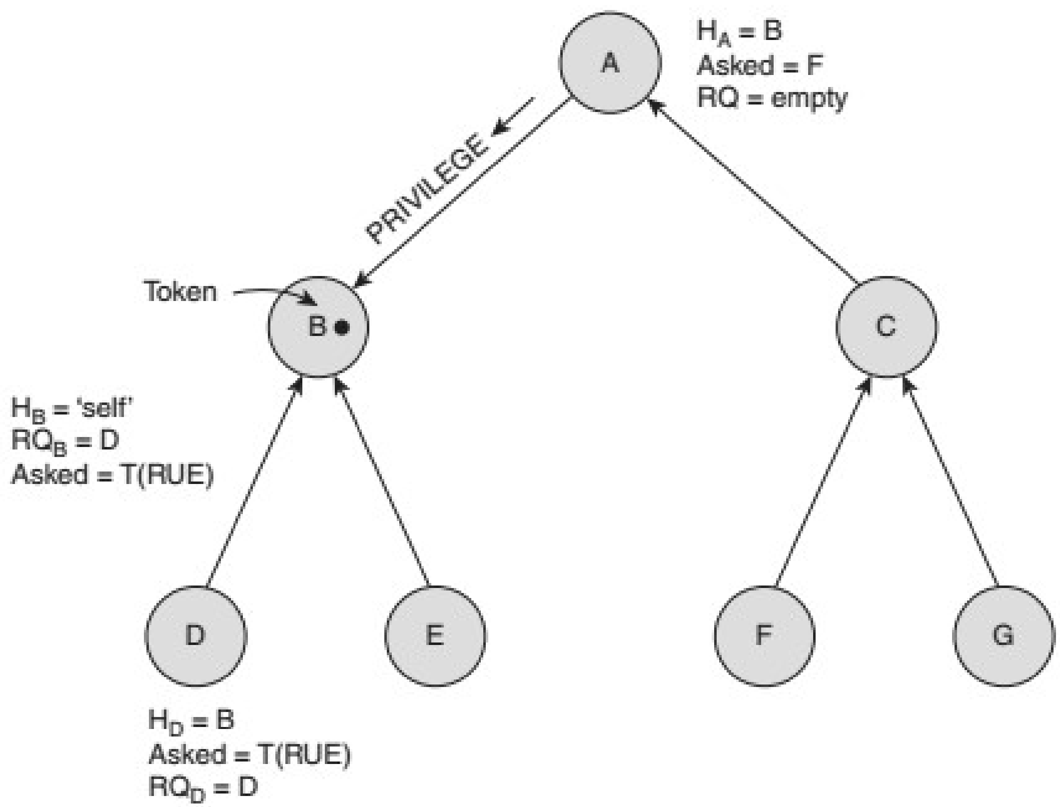
Performance Parameters

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

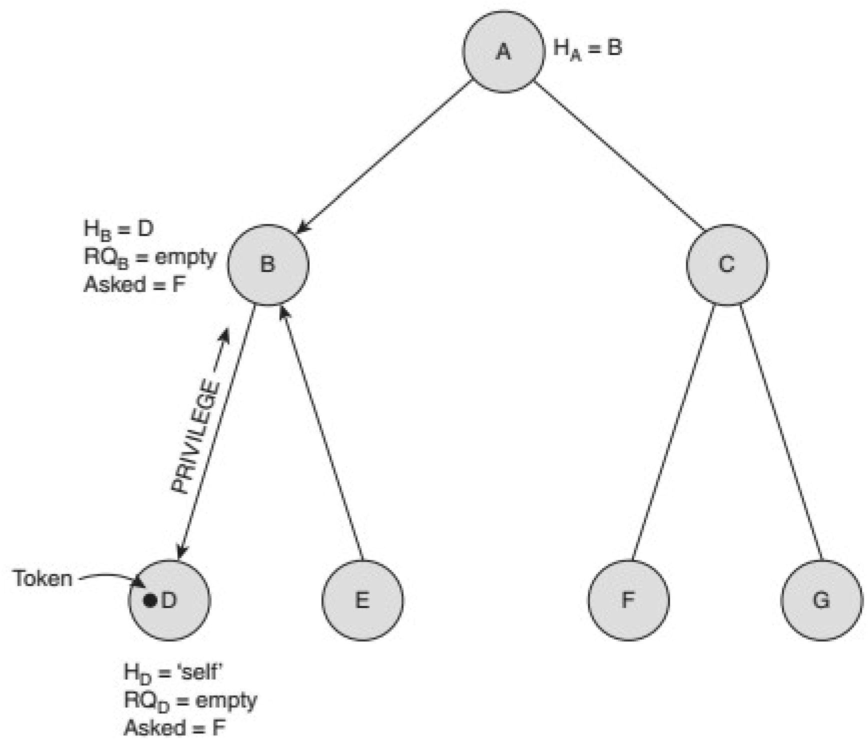
# Raymond’s Tree-Based Algorithm




(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.