$b \leq \theta \leq a$.

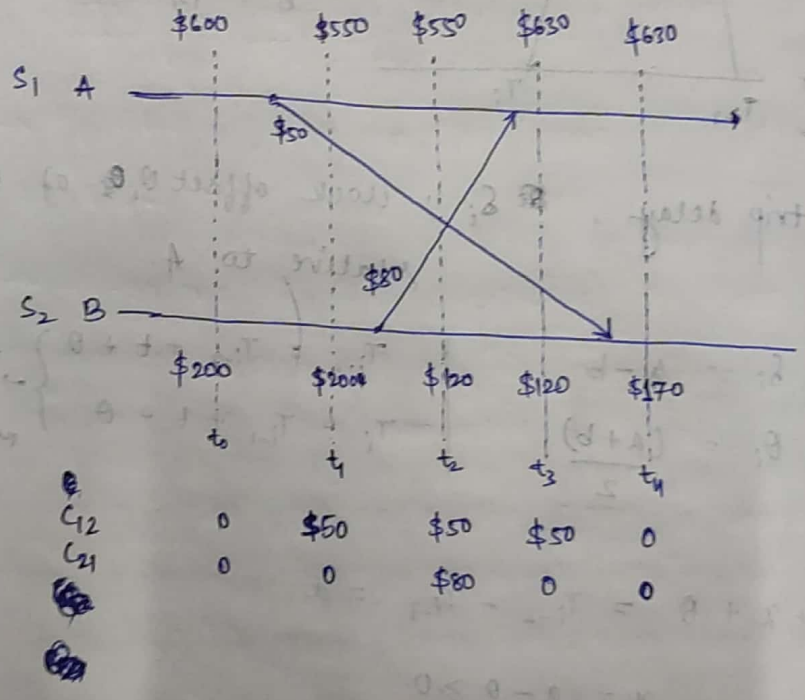
$$b = \frac{a+b}{2} - \frac{a-b}{2} \leq \theta \leq \frac{a+b}{2} + \frac{a-b}{2} = a$$

$$\theta_i - \frac{d_i}{2} \leq \theta \leq \theta_i + \frac{d_i}{2}$$

↳ Offset can be $-\frac{d_i}{2}$ or $\frac{d_i}{2}$.

Global State and Snapshot Recording Algorithm.

→ Recording global state is necessary for analyzing, testing or verifying properties associated with distributed execution.



	t_0	t_1	t_2	t_3	t_4
C_{12}	0	$50	$50	$50	0
C_{21}	0	0	$80	0	0

* Determination of global states is non-trivial.

Scanned by CamScanner

→ System Model

→ Transit

↳ $transit(LS_i, LS_j) = \{m_{ij} \mid send(m_{ij}) \in LS_i \wedge rec(m_{ij}) \notin LS_j\}$

→ A consistent global state,

$GS = \{\cup_i LS_i, \cup_{ij} SC_{ij}\}$ is achieved if the following 2 conditions are satisfied:

i) C_1 : $send(m_{ij}) \in LS_i \Rightarrow m_{ij} \in SC_{ij} \oplus rec(m_{ij}) \in LS_j$ (⊕ = XOR)

ii) C_2 : $send(m_{ij}) \notin LS_i \Rightarrow m_{ij} \notin SC_{ij} \wedge rec(m_{ij}) \notin LS_j$

↳ C1 : Indicates the law of conservation of messages.

↳ C2 : States that, ~~unless there is a cause~~ for every effect, its cause must be present.

→ Issues in Recording a Global State

↳ I1 : How to distinguish between messages to be recorded in the snapshot (i.e., either in the channel state or a process state) from those not to be recorded.

Any message that is sent by a process before recording its snapshot must be recorded in the global state (from c1).

Any message that is sent by a process after recording its snapshot must not be recorded in the global snapshot (from c2).

Scanned by CamScanner

↳ I2: How to determine the instant when a process takes its snapshot.
A process P_j must record its snapshot before processing a message that was sent by a process P_i after recording its snapshot.

→ 2 types of messages: computation messages and control messages.

→ Execution of snapshot algorithm is transparent to the underlying application, except for occasional delaying of some of the actions of the application

Snapshot Algorithm for FIFO Channels.

1. Chandy-Lamport Algorithm

→ Control message in this case is referred to as a marker.

→ After a site has recorded its state, it sends a marker along all of its outgoing channels, before sending out any more messages.
Since the channels are FIFO, a marker separates the messages in the channel into those to be included in the snapshot from those not to be recorded in the snapshot.

→ Marker Sending Rule for process P_i
i> Process P_i records its state, and
ii> For each outgoing channel c, on which a marker has not been sent P_i sends a marker

Scanned by CamScanner

along c before P_i sends further messages along c.

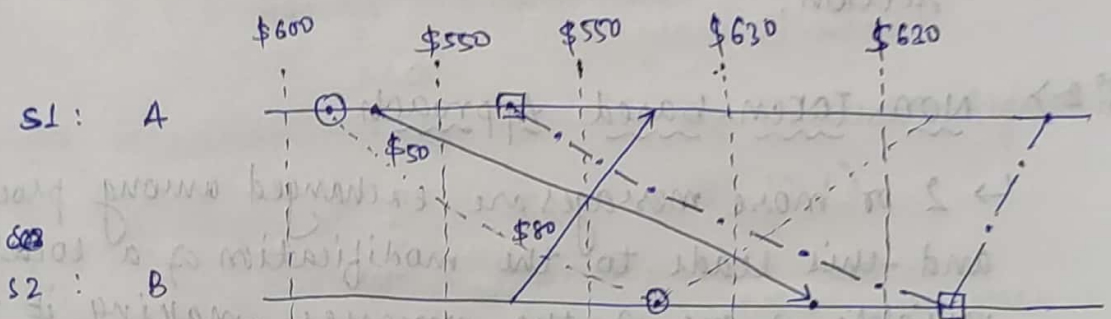
→ Marker Receiving Rule for process P_j

i) On receiving marker along channel c:

↳ If P_j has not recorded its state, then record the state of c as the empty set.

• Execute the marker send up rule.

↳ else, record the state of c as the set of messages received along c after P_j's state was recorded and before P_j receive the marker along c.



	t_0	t_1	t_2	t_3	t_4
c_{12}	0	$50	$50	$50	0
c_{21}	0	0	$80	0	0

□ --- : A = $550 ; B = $170 ; c_{12} = $0 ; c_{21} = $80

⊙ --- : A = $600 ; B = $120 ; c_{12} = $0 ; c_{21} = $80

These are consistent global states.

Scanned by CamScanner

Distributed Mutual Exclusion Algorithms

* Sleeping Barber Problem; Bakery Algorithm; Dining philosopher's problem; Dekker's Algorithm; semaphore

→ Message passing is the sole means of implementing critical section systems.

→ Three approaches:

i.) Token-based Approach.

↳ Token is a privileged message

↳ The token is allowed to enter the critical section.

ii.) Non Token-based Approach

↳ 2 or more messages are exchanged among process and this leads to the modification of a local variable of one of the processes, making it to enter the critical section.

iii.) Quorum-based Approach

↳ A process will make a subset of processes (referred to as quorum), and not all the process to enter the critical section.

↳ Effort required will be less than the above 2 approaches.

** In approaches (i) & (ii), the process wanting to enter the critical section makes request to all the processes.

Scanned by CamScanner

→ System Model.

↳ A process can be in 3 states:

i) Requesting the critical section

ii) Executing the critical section

iii) Neither requesting nor executing the critical section (idle).

** Time stamp is the measure for priority of requesting processes for critical section.
lower the timestamp, higher the priority.

Q: N = Total processes requesting for CS (Critical section)
T = Average Delay Time
E = Average Execution Time

→ Requirements of Mutual Exclusion Algorithms
(3 properties need to be satisfied)

↳ Safety Property: At any instant, only one process can execute the critical section.

↳ Liveness Property: Absence of deadlocks and starvation; there should be progress; No process should wait indefinitely to enter the critical section.

↳ Fairness Property: Each process gets a fair chance to execute the critical section.

→ Performance Metrics

↳ Message complexity
- No. of messages that are required per critical section execution by a site (assumption is that there is one process per site).

Scanned by CamScanner

↳ Synchronization Delay

↳ Response Time → (It means the instant from which request is sent)

- Time from request made by the process for the critical section till the execution is complete.

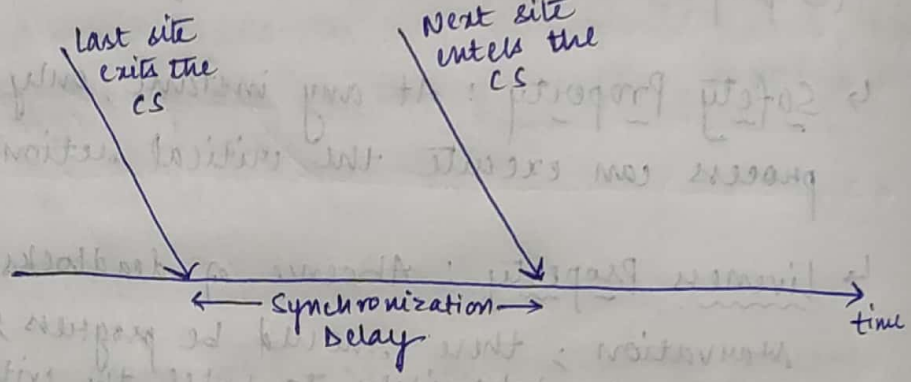
↳ System Throughput

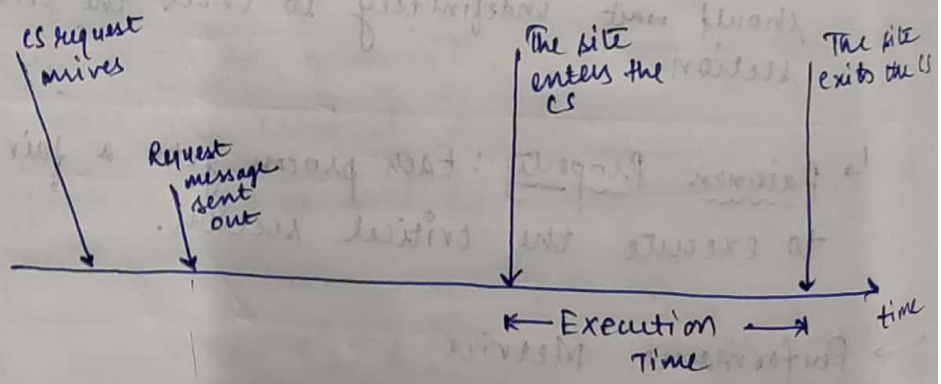
- Rate at which system is able to execute the critical section.

SD : Synchronization Delay

E : Average execution time

$$\text{System Throughput} = \frac{1}{SD + E}$$





Response Time

Scanned by CamScanner

↳ Low load and high load performance
- Low load & high load means how frequently the messages arrive.

↳ Best and worst case performance.
- Low load (less frequent messages), so best performance
- In high load, processes may have to wait to get the messages.

1. Lamport's Algorithm
→ Requests for CS are executed in order of their timestamp.
Time is defined by Lamport's logical clock.

i) → Requesting the critical section.
↳ When a site wants to enter the CS, it broadcasts a request message, i.e., (ts_i, i) message to all other sites and places the requests on the request-queue$_i$. (Every process maintains a request queue).

↳ When a site S_j receives the ~~request~~ REQUEST (ts_i, i) message from site S_i, it places site S_i's request on its request-queue$_j$, and returns a timestamp REPLY message to S_i.

ii) → Executing the critical section.
↳ L1: S_i has received a message with timestamp larger than (ts_i, i) from all other sites.
↳ L2: S_i's request is at the top of request-queue$_i$.

Scanned by CamScanner

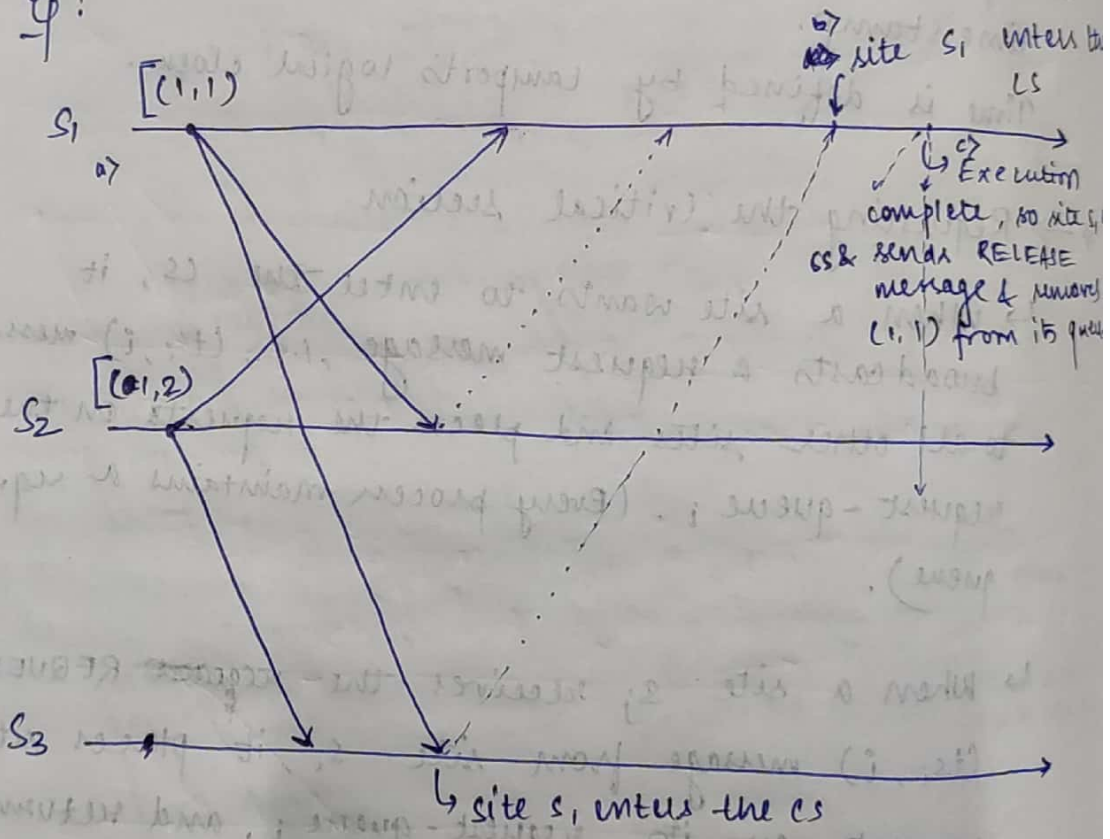
iii) → Releasing the Critical section

↳ Site S_i upon exiting the CS removes its request from the top of its queue and broadcasts a time stamp RELEASE message to all other sites.

↳ When a site S_j receives RELEASE message from site S_i, it removes S_i's request from its queue.

→ 3(N-1) messages need to be sent for one CS execution of a process.

Eg:



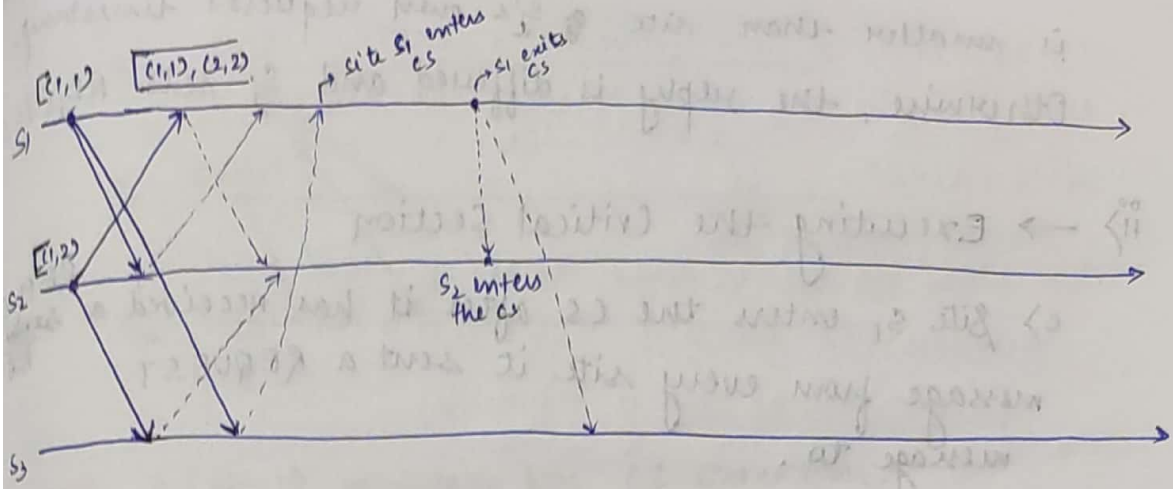
a: S_1 and S_2 are making requests for the CS.

b: Site S_1 enters the CS

c: Site S_1 exits the CS and sends RELEASE message.

(a, b, c are 3 separate diagrams, each containing the preceding one)

Scanned by CamScanner



State S_2 enters the CS

→ Performance: For each CS execution, $3(N-1)$ messages for

→ Performance can be optimized from $3(N-1)$ to $2(N-1)$ by removing some of the reply messages.

→ Communication channel need to be FIFO.

2. Ricart - Agrawala Algorithm

→ REQUEST and REPLY messages.

→ Initially, $\forall i, \forall j : RD_i[j] = 0$.

RD_i: Request - Differed array of site S_i.

→ Does not require communication channel to be FIFO.

i) → Requesting the critical section

a) when a site S_i wants to enter the CS, it broadcasts a timestamp REQUEST message to all other sites.

b) When a site S_j receives a REQUEST message from 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 timestamp

Scanned by CamScanner

is smaller than site S_i's own request's timestamp. Otherwise, the reply is differed and S_j sets $RD_j[i]$.

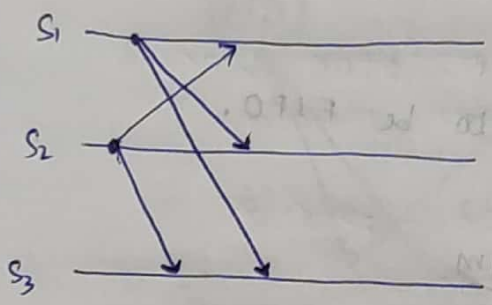
ii) → Executing the Critical Section

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

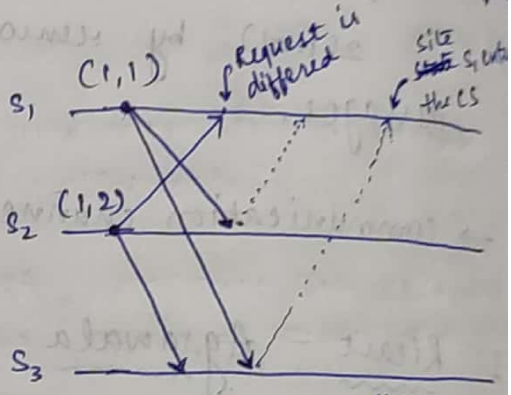
d) When site S_i exits the CS, it sends all the differed REPLY messages, i.e.,

$$\forall j : RD_i[j] = 1.$$

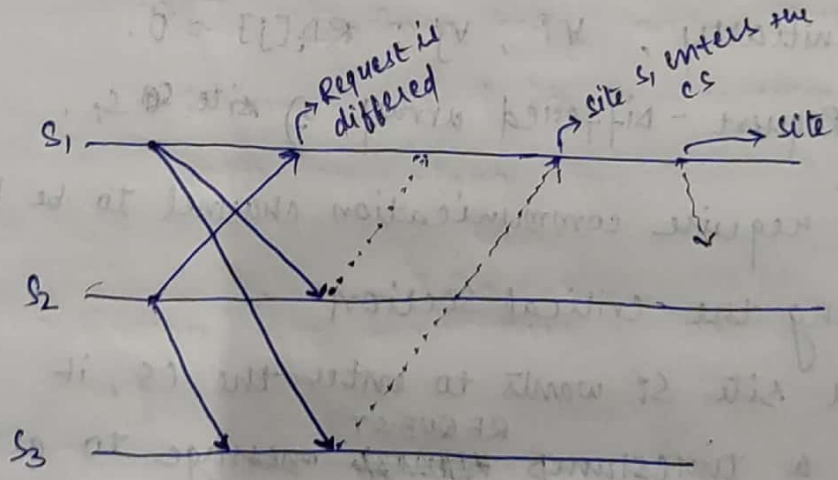
and, after sending the REPLY message, sets $RD_i[j]$



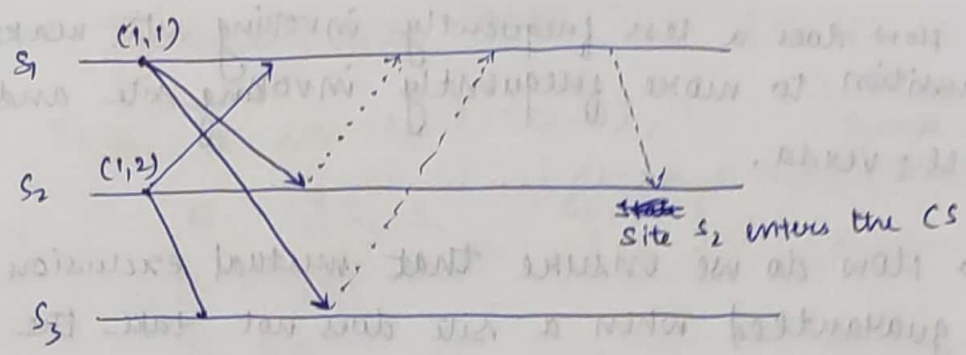
Sites S_1 and S_2 make a request for the CS.



Site S_1 enters the CS



Scanned by CamScanner



→ 2(N-1) messages per CS execution.

3. Singhal's Dynamic Information Structure Algorithm

→ Most mutual exclusion algorithms have a static approach.

→ Lack of efficiency: fail to exploit the changing condition in the system.

An algorithm can exploit changing condition to optimize the performance.

→ If a few sites are invoking mutual exclusion very frequently and other sites invoke mutual ~~lock~~ less frequently, then a frequently invoking site need not ask for the permission of less frequently invoking site everytime it requests an access for the CS.

→ Some of the design challenges:

↳ How does a site efficiently knows what sites are currently active, invoking mutual exclusion?

↳ When a less frequently invoking site needs to invoke mutual exclusion, how does it do it?

Scanned by CamScanner

↳ How does a less frequently invoking site makes a transition to more frequently invoking site and vice-versa.

↳ How do we ensure that mutual exclusion is guaranteed when a site does not take the permission of every other site?

↳ How do we ensure that a dynamic mutual exclusion algorithm does not waste resources and time in collecting the system state, offsetting any gain?

→ System Model

↳ 2 Data structures are maintained by each process (they are basically sets):

i) R_i : Request set, contains the sites from which S_i must acquire permission before executing CS.

ii) I_i : Inform set, contains the sites to which S_i must send its permission to execute it after executing its CS.

↳ Every site maintains a logical clock C_i.

↳ Three boolean variables:

- Requesting
- Executing
- My-Priority : will be true in case if requesting site S_i has a high priority

Scanned by CamScanner

i) → Initialization State $\{S_n, S_{n-1} \ldots S_i, \ldots S_2, S_1\}$

↳ For a site S_i $(i = 1, \ldots N)$, ↑ Initial set is like this

$R_i = \{S_1, S_2 \ldots S_{i-1}, S_i\}$

$I_i = \{S_i\}$

$C_i = 0$

Requesting = Executing = false

The cardinality of R_i decreases in a step-wise manner from left to right.

This is referred to a staircase pattern based on the topological sense.

Step 1: (Request CS)

↳ Requesting = true

↳ $C_i = C_i + 1$

↳ send REQUEST (C_i, i) message to all sites in R_i

↳ wait until ~~all sites in~~ $R_i = \phi$

(i.e., wait until all sites in R_i have sent a reply to S_i)

Step 2: (Executing CS)

↳ Executing = true

↳ Execute CS

↳ Executing = false

Step 3: (Release CS)

↳ for every site S_k in I_i (except S_i), do the following:

(on next page →)

Scanned by CamScanner

```
begin
    I_i = I_i - {S_k}
    send a REPLY(C_i, i) message to S_k
    R_i = R_i + {S_k}
end.
```

→ REQUEST Message handler :

/* site S_i is handling message REQUEST (C_j, j) */

$C_i = \max(C_i, C)$

↳ Case : Requesting = true.

```
begin
    if My-priority, then I_i = I_i + {S_j}
    /* My-Priority is true if the pending request
    of S_i has priority over the incoming request */

    else
        begin
            send REPLY(C_i, i) message to S_j
            if not S_j in R_i then,
                begin
                    R_i = R_i + {S_j}
                    send REQUEST(C_i, i) message to S_j
                end
        end
end
```

Scanned by CamScanner

↳ Case : Executing = true.

$I_i = I_i + \{S_j\}$

↳ Case : Executing = false ∧ Requesting = false.

begin

$R_i := R_i + \{S_j\}$

Send REPLY(C_i, i) message to S_j

end.

→ REPLY Message Handler :

/* Site S_i is handling a message REPLY(c, j) */

begin

$C_i = \max(C_i, c)$;

$R_i := R_i - \{S_j\}$

end.

Eg: 4 sites: S_4 S_3 S_2 S_1

So, $R_1 = \{S_1\}$; $R_2 = \{S_2\ S_1\}$; $R_3 = \{S_3\ S_2\ S_1\}$

$R_4 = \{S_4\ S_3\ S_2\ S_1\}$

Now, S_3 requests.

⇒ $R_1 = \{S_3\ S_1\}$; $R_2 = \{S_3\ S_2\ S_1\}$

R_3 will be empty, and S_3 will become the rightmost thing, i.e., S_4 S_2 S_1 S_3.

Now, if S_4 requests after this, we will have the positions like : S_2 S_1 S_3 S_4.

* Stateless pattern is maintained.

Scanned by CamScanner

4. Quorum - Based Mutual Exclusion Algorithm

$\forall i, \forall j \quad 1 \le i, j \le N :: R_i \cap R_j \neq \phi$

→ A site can send only one REPLY message at a time

→ A coterie C is defined as a set of sets where each set g is a member of C ($g \in C$), and g is referred to as the quorum.

→ Intersection Property.

↳ For every quorum, $g, h \in C$, $g \cap h \neq \phi$

→ Minimality Property

↳ There should be no quorums, $g, h \in C$, such that $g \supseteq h$ (g is superset of h).

i) Maekawa's Algorithm

→ 4 conditions.

↳ M1: $(\forall i, \forall j : i \neq j, 1 \le i, j \le N :: R_i \cap R_j \neq \phi)$

↳ M2: $(\forall i ; 1 \le i \le N :: S_i \in R_i)$

↳ M3: $(\forall i : 1 \le i \le N :: |R_i| = k)$

Now, If there are N sites,

$k = \sqrt{N}$

↳ M4: Any site S_j is contained in k number of R_i's $(1 \le i, j \le N)$

Scanned by CamScanner

* M1 & M2 : Correctness

M3 : Request sets of all sites must be equal.

M4 : Enforces that exactly the same number of sites should request permission from any site.

→ Requesting the critical section

(a) A site S_i requests access to the CS by sending REQUEST(i) message to all sites in its request set R_i.

(b) When a site S_j receives the REQUEST(i) message, it sends a REPLY(j) message to S_i, provided it has not sent the REPLY message to a site since its receipt of the last RELEASE message, otherwise, it queues up the REQUEST(i) message for later consideration.

→ Executing the Critical Section.

(c) Site S_i executes the CS only after it has received a REPLY message from every site in R_i.

→ Releasing the Critical Section

(d) After the execution of CS is over, site S_i sends a RELEASE(i) message to every site in R_i.

(e) When a site S_j receives a RELEASE(i) message from site S_i, it sends a REPLY message to the next site in the queue and deletes that entry corresponding to S_i from the queue. If the queue is

Scanned by CamScanner

empty, then the state updates its state to reflect that it has not sent any REPLY message since the receipt of the last RELEASE message.

→ Performance

↳ $\sqrt{N}$ REQUEST, $\sqrt{N}$ REPLY and $\sqrt{N}$ RELEASE messages.

↳ In total, $3\sqrt{N}$ messages.

*. There is no notion of time, so requests can go in any order.

So, there is a possibility of deadlock.

→ Problem of Deadlock:

S_i S_j S_k enter the system simultaneously.

Suppose, $R_i \cap R_j = \{S_{ij}\}$

$R_j \cap R_k = \{S_{jk}\}$

$R_k \cap R_i = \{S_{ki}\}$

Suppose, S_{ij} has been locked by S_i, S_{jk} is locked by S_j, and S_{ki} has been locked by S_k.

Now, due to deadlock, we have 3 other types of commands/messages to avoid the deadlocks:

FAILED INQUIRE YIELD

Scanned by CamScanner

(ii) Agarwal El Abbadi Quorum-Based Algorithm

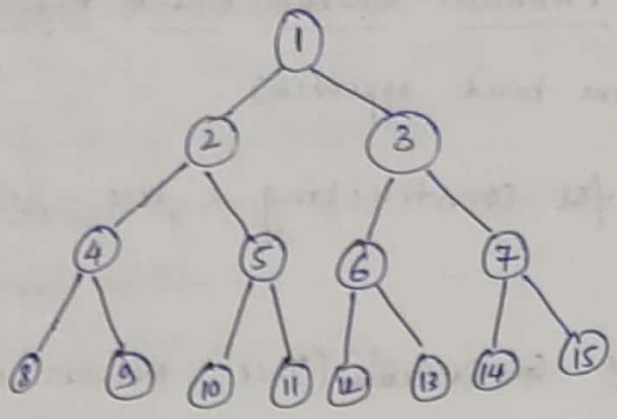
(Tree - quorum based approach)

→ Algorithm for constructing a tree-structured quorum:

```
↳ function GetQuorum (Tree : Network Hierarchy):
                                        Quorumset
↳ var : left, right, quorumset
begin:
    if Empty (Tree), then return ({});
    else if GrantPermission (Tree↑.Node), then
        return ((Tree↑.Node) ∪ (Tree↑.LeftChild))
                    or
        return ((Tree↑.Node) ∪ (Tree↑.RightChild))
    else left ← GetQuorum (Tree↑.Left);
         right ← GetQuorum (Tree↑.right);
    if (left = ϕ ∨ right = ϕ)
        exit (-1)
    else
        return (left ∪ right)
    end
  end
 end
end GetQuorum
```

Scanned by CamScanner

Eg:



{1,2,4,8} {1,2,4,9} {1,2,5,10} {1,2,5,11}

{1,3,6,12} {1,3,6,13} {1,3,7,14} {1,3,7,15}

↳ Now, when node 3 fails, the possible paths are: 6-12, 6-13 & 7-14, 7-15

{1,2,4,8} {1,2,4,9} {1,2,5,10} {1,2,5,11}

{1,6,12,7,14} {1,6,12,7,15} {1,6,13,7,14} {1,6,13,7,15}

↳ If sites 1 and 2 are down, then the quorums must include either {4,8} or {4,9} and either {5,10} or {5,11}.

{3,6,12} {3,6,13} {3,7,14} {3,7,15}

{4,5,3,6,8,10,12} ~~{4,5,3,6,8,10,12}~~

{4,5,3,6,8,10,13}

{4,5,3,6,8,11,12}

{4,5,3,6,8,11,13}

{4,5,3,6,9,10,12}

{4,5,3,6,9,10,13}

{4,5,3,6,9,11,12}

{4,5,3,6,9,11,13}

{4,5,3,7,8,10,14} {4,5,3,7,8,10,15}

↓ combinations of these sets with the above 4 combinations.

(In all, total 16 combinations)

{4,5,3,7,9,10,14}

{4,5,3,7,9,10,15}

{4,5,3,7,9,11,14}

{4,5,3,7,9,11,15}

Scanned by CamScanner

↳ With the number of node failures greater than or equal to $\log n$, the algorithm may not be able to form tree-structured quorums.

If a failed site is a leaf node, the operation has to be aborted and a tree-structured quorum cannot be formed.

→ Suppose a site S wants to enter the critical section, the following events should occur:

1. Site S sends a "Request" message to all other sites in the structured quorum it belongs.
2. Each site in the quorum stores incoming requests in a request queue ordered by their timestamps.
3. A site sends a "Reply" message indicating its consent to enter CS only to the request at the head of its request queue, having the lowest timestamp.
4. If the site S gets a "Reply" message from all sites in the structured quorum, it belongs to, it enters the CS.
5. After exiting the CS, S sends a "Relinquish" message to all sites in the structured quorum. On the receipt of a "Relinquish" message, each site removes S's request from the head of its request queue.
6. If a new request arrives with a timestamp smaller than the request at the head of the queue, an "Inquire" message will be sent to the process whose request is at the head of the queue and the site waits for "Yield" or "Relinquish" message.

Scanned by CamScanner

7. When a site receives an "Inquire" message, it acts as follows:

- If S has acquired all of its necessary to access the CS, then it simply ignores the message and proceeds normally and sends a "Relinquish" message after it exits the CS.
- If S has not yet collected enough replies from its quorum, then it sends a "Yield" message to the enquiring site.

8. When a site gets a "Yield" message, it puts the pending request (on behalf of which the 'Inquire' message was sent) at the head of the queue and sends a "Reply" message to the requester.

Deadlock Detection in Distributed Systems

→ If the allocation sequence of process resources is not controlled, deadlocks can occur.

→ A deadlock can be defined as a condition where a set of processes request resources that are held by other processes in the set.

→ 3 Approaches to handle deadlocks:

↳ Deadlock Prevention
↳ Deadlock Avoidance
↳ Deadlock Detection

Scanned by CamScanner

→ Deadlock detection requires the examination of the status of the process-~~resource~~ receiver interaction.

→ To resolve a deadlock, we have to abort a deadlocked process.
(Generally the lowest priority process is aborted)

→ Assumption: The systems have only reusable resources.

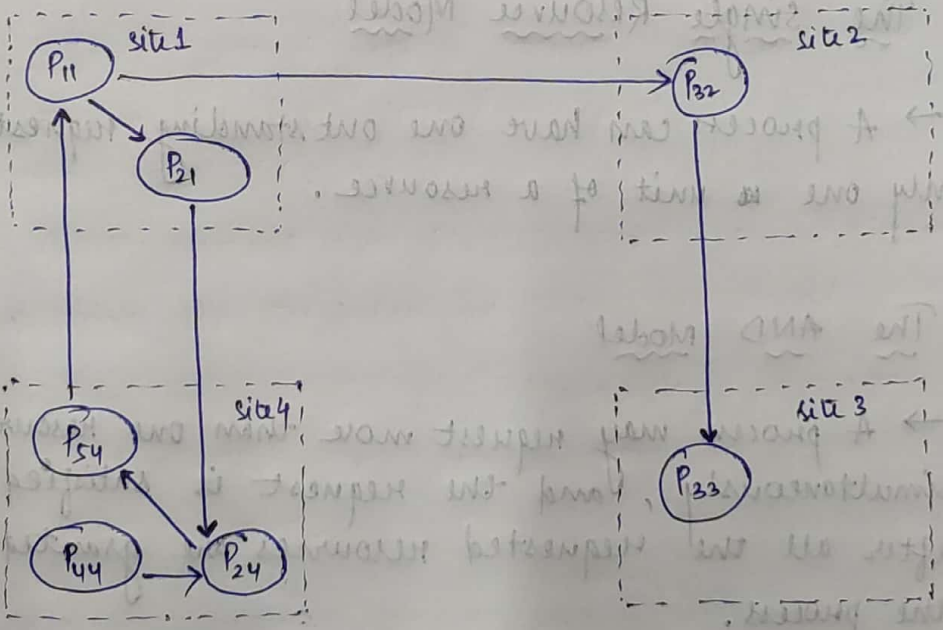
: Processes are allowed to make only exclusive access (not shared) to resources.

: There is only one copy of each resource.

→ Wait-for-Graph (WFG)

↳ Reflects the state of a system.

↳ Shows which process is waiting for which resource.



A WFG for 4 processes.

There is a cycle between sites 1 and 4, so it is a deadlock condition.

Scanned by CamScanner

→ A system is deadlocked iff there exists a directed cycle or knot in the WFG.

Deadlock Handling Strategies

→ Prevention : inefficient

Avoidance : impractical

Detection : only thing that is practically efficient.

→ Detection:

↳ Progress (no undetected deadlocks)

↳ Safety (no false deadlocks)

Models of Deadlocks

1. The single Resource Model

→ A process can have one outstanding request for only one unit of a resource.

2. The AND Model

→ A process may request more than one resource simultaneously, and the request is satisfied only after all the requested resources are granted to the process.

3. The OR Model

Scanned by CamScanner