What is Mutual exclusion?

- Mutual exclusion : makes sure that concurrent process access shared resources or data in a serialized way.

  If a process , say P_i , is executing in its critical section, then no other processes can be executing in that critical sections

Example: updating a DB

What is Mutual exclusion?

- Only one process is allowed to execute the critical section (CS) at any given time.
- The actions performed by a user on a shared resource must be *atomic*.
- In a distributed system, shared variables (semaphores) cannot be used to implement mutual exclusion.
- Message passing is the sole means for implementing distributed mutual exclusion.

Requirements:

- safety : at most one process in critical section
- Freedom from Deadlocks: if more than one requesting process, someone enters
- Freedom from starvation: a requesting process enters within a finite time
- Fairness: Requests are granted in order they are made. Fairness implies freedom from starvation but not vice-versa.

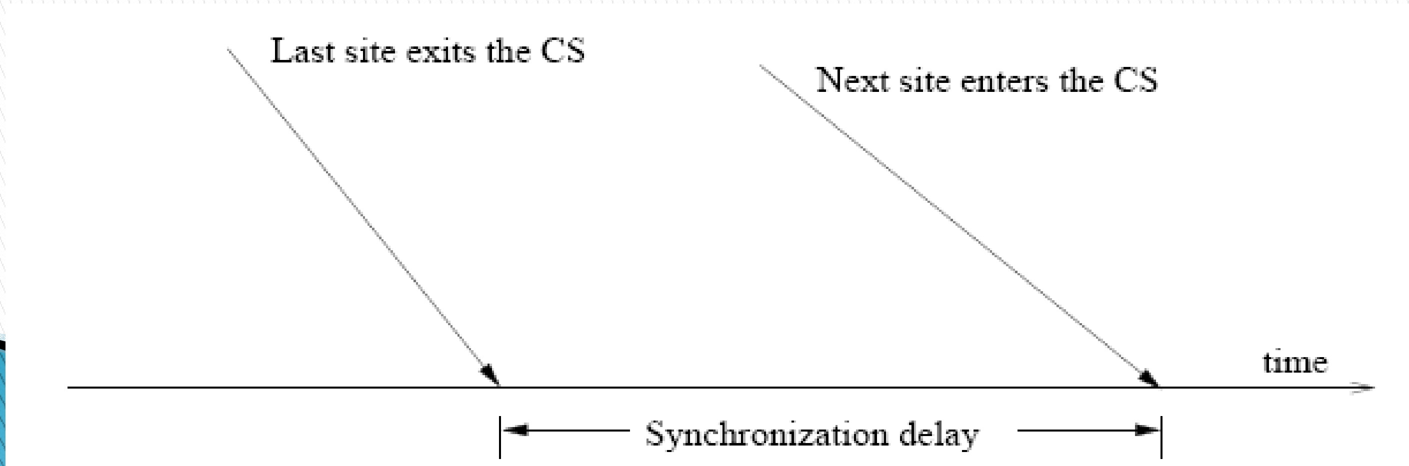
Classification of Distributed Mutual Exclusion Algorithms

- Non-token based/Permission based
  - Permission from all processes: e.g. Lamport, Ricart-Agarwala, Raicourol-Carvalho etc.
  - Permission from a subset: ex. Maekawa
- Token based
  - ex. Suzuki-Kasami

Performance Metrics

The performance is generally measured by the following four metrics:

- Message complexity: The number of messages required per CS execution by a process.
- Synchronization delay: After a site leaves the CS, it is the time required and before the next process enters the CS (see Figure 1)



Performance Metrics

- Response time: The time interval a request waits for its CS execution to be over after its request messages have been sent out (see Figure 2).

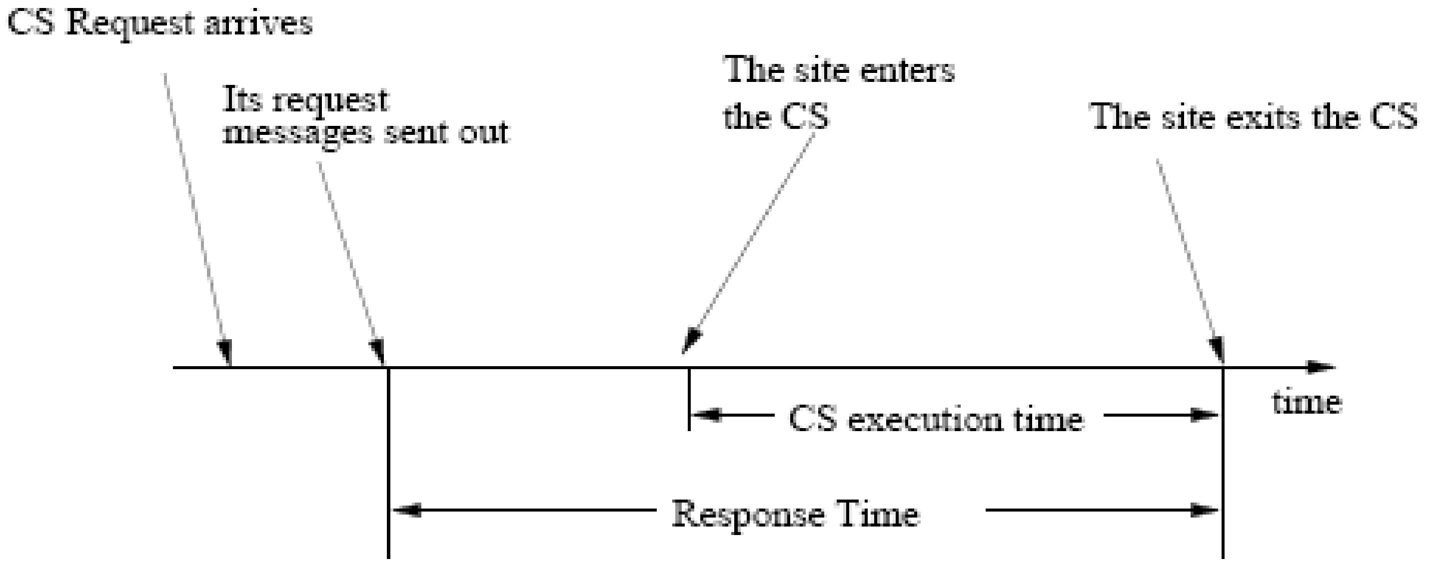


Figure 2: Response Time.

Performance Metrics

System throughput: The rate at which the system executes requests for the CS.

system throughput=1/(SD+E)

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

Low and High Load Performance:

- The load is determined by the arrival rate of CS execution requests.
- Under *low load* conditions, there is seldom more than one request for the critical section present in the system simultaneously.
- Under *high load* conditions, there is always a pending request for critical section at a site.

Non-Token-based Algorithms

- A site communicates with a set of other sites to arbitrate who should execute the CS next.
- Use timestamps to order requests for the CS and to resolve conflicts between simultaneous requests for the CS.
- Logical clocks are maintained and updated according to the Lamport's scheme.

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.

The Algorithm

<u>To request critical section:</u>

- send timestamped REQUEST (ts_i, i) to all other sites
- put (ts_i, i) in its own queue

<u>On receiving a request (ts_i, i):</u>

- send timestamped REPLY to the requesting site i
- put request (ts_i, i) in the queue

- <u>Executing the critical section</u>: Site S_i enters the CS when the following two conditions hold:
  - 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*$_i$.

The Algorithm

- Releasing the critical section:
  - removes its request from the top of its request queue and broadcasts a timestamped RELEASE message to all other sites.
  - On receiveing a RELEASE message from site Si , 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.

correctness

Theorem: Lamport's algorithm achieves mutual exclusion.

Proof:

- Proof is by contradiction. Suppose two sites S_i and S_j are executing the CS concurrently. For this to happen conditions L1 and L2 must hold at both the sites *concurrently*.

- This implies that at some instant in time, say t, both S_i and S_j have their own requests at the top of their *request_queues* and condition L1 holds at them. Without loss of generality, assume that S_i 's request has smaller timestamp than the request of S_j .

- From condition L1 and FIFO property of the communication channels, it is clear that at instant *t* the request of Si must be present in *request_queue*$_j$ when S_j was executing its CS. This implies that 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!

correctness

Theorem: Lamport's algorithm is fair.
Proof:

- The proof is by contradiction. Suppose a site Si 's request has a smaller timestamp than the request of another site S_j and S_j is able to execute the CS before S_i .
- For S_j to execute the CS, it has to satisfy the conditions L1 and L2. This implies that at some instant in time say t, S_j has its own request at the top of its queue and it has also received a message with timestamp larger than the timestamp of its request from all other sites.
- But request_queue at a site is ordered by timestamp, and according to our assumption S_i has lower timestamp. So Si 's request must be placed ahead of the S_j 's request in the $request\_queue_j$. This is a contradiction!

Performance

- For each CS execution, Lamport's algorithm requires $(N - 1)$ REQUEST messages, $(N - 1)$ REPLY messages, and $(N - 1)$ RELEASE messages.
- Thus, Lamport's algorithm requires $3(N - 1)$ messages per CS invocation.
- Synchronization delay in the algorithm is T.

An optimization

- In Lamport's algorithm, REPLY messages can be omitted in certain situations. For example, if 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.