

Lecture: 07

Distributed Mutual Exclusion Algorithms & Non-Token Based Approaches



Rajiv Misra

Dept. of Computer Science & Engineering
Indian Institute of Technology Patna
rajivm@iitp.ac.in

Preface

Recap of Previous Lecture:

- In the previous lecture, we have discussed about the **Global states** (i.e. **consistent, strongly consistent & inconsistent**), **Cuts in the space time diagram**, **Models of communication** and
- **Snapshot algorithm** *i.e.* **Chandy-Lamport algorithm** to record the global snapshot.

Preface

Content of this Lecture:

- In this lecture, we will discuss about **Mutual exclusion algorithms** for **distributed computing systems** such as:
 - (i) Non-token-based approach,
 - (ii) Quorum based approach and
 - (iii) Token-based approach

Introduction

- **Mutual exclusion:** Concurrent access of processes to a shared resource or data is executed in mutually exclusive manner.
- Only one process is allowed to execute the **critical section (CS)** at any given time.
- In a distributed system, shared variables (**semaphores**) or a local kernel cannot be used to implement mutual exclusion.
- Message passing is the sole means for implementing distributed mutual exclusion.

Approaches of Distributed Mutual Exclusion

- Distributed mutual exclusion algorithms must deal with **unpredictable message delays** and **incomplete knowledge** of the **system state**.
- Three basic approaches for distributed mutual exclusion are:
 - i. **Non-Token based approach**
 - ii. **Quorum based approach**
 - iii. **Token based approach**

(i) 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.

e.g. Lamport Algorithm, RicartAgarwala Algorithm etc.

(ii) Quorum based approach:

- Each site requests permission to execute the CS from a subset of sites (called a **quorum**).
- Any two quorums contain a **common site**.
- This common site is responsible to make sure that only one request executes the CS at any time.
- e.g. Maekawa's Algorithm, Agarwal-El Abbadi Quorum-Based Algorithm etc.

(iii) Token-based approach:

- A unique token (also known as the **PRIVILEGE message**) 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**.
- e.g. Suzuki-Kasami's Broadcast Algorithm, Raymond's Tree-Based Algorithm etc.

Preliminaries: System Model

- The system consists of N sites, $S_1, S_2, \dots, S_N$.
- We assume that a single process is running on each site. The process at site S_i is denoted by p_i .
- 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.

Requirements of Mutual Exclusion Algorithms

- 1. Safety Property:** At any instant, **only one process** can execute the critical section.
- 2. Liveness Property:** This property states the absence of **deadlock and starvation**. Two or more sites should not endlessly wait for messages which will never arrive.
- 3. Fairness:** Each process gets a fair chance to execute the CS. Fairness property generally means the CS execution requests are **executed in the order of their arrival** (time is determined by a **logical clock**) in the system.

Performance Metrics

The performance is generally measured by the following four metrics:

(i) Message complexity: The **number of messages** required per CS execution by a site.

(ii) Synchronization delay: After a site leaves the CS, it is the time required and before the next site enters the CS (see Figure 7.1).

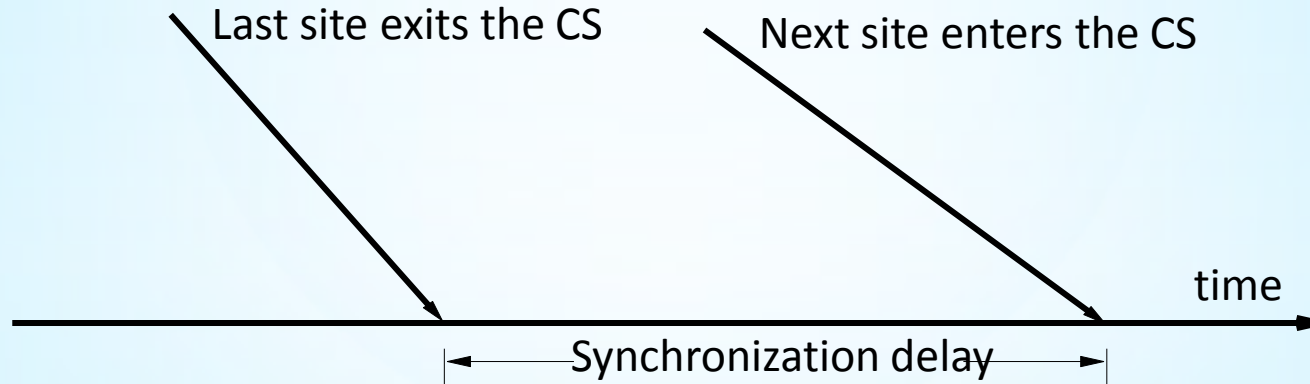
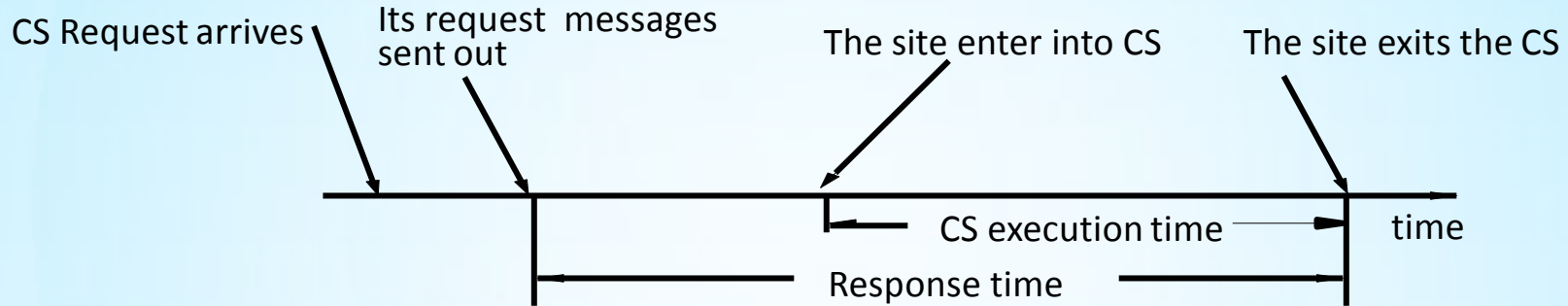


Figure 7.1: Synchronization Delay

Performance Metrics

(iii) Response time: The time interval a request waits for its CS execution to be over after its request messages have been sent out (see below Figure 7.2)



(iv) System throughput: The rate at which the system executes requests for the CS.
$$\text{system throughput} = 1 / (SD + E)$$

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

Performance Metrics

Low and High Load Performance:

- We often study the performance of mutual exclusion algorithms under two special loading conditions, viz., “*low load*” and “*high load*”.
- 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 *heavy load* conditions, there is always a pending request for critical section at a site.

(i) Non-token based Approaches

(i) 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 the **FIFO order**. Three types of messages are used- **Request, Reply and Release**. These messages with timestamps also **updates** logical clock.

The 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 S_j receives the **REQUEST**(ts_i, i) message from site S_i , S_j 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** 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:

- 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.

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 S_i 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

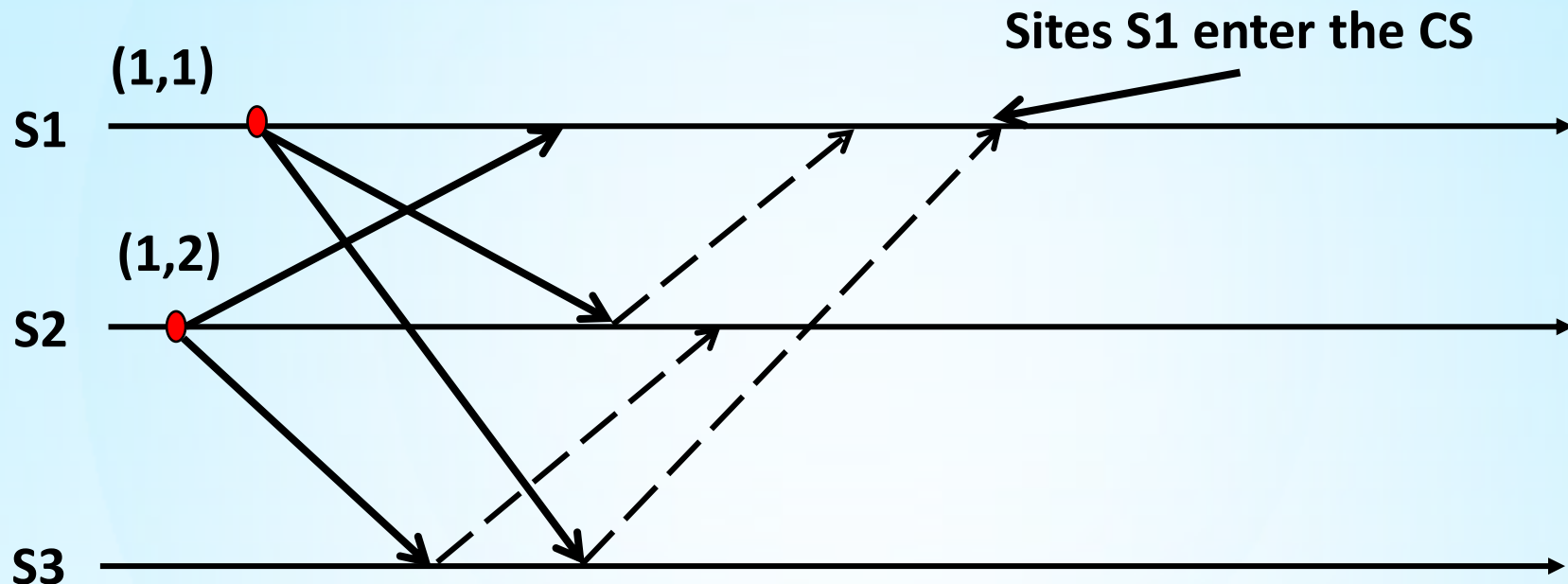
Theorem: Lamport's algorithm is fair.

Proof:

- The proof is by contradiction. Suppose a site S_i '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 S_i 's request must be placed ahead of the S_j 's request in the *request_queue_j*. This is a contradiction!

Lamport's Algorithm Example:

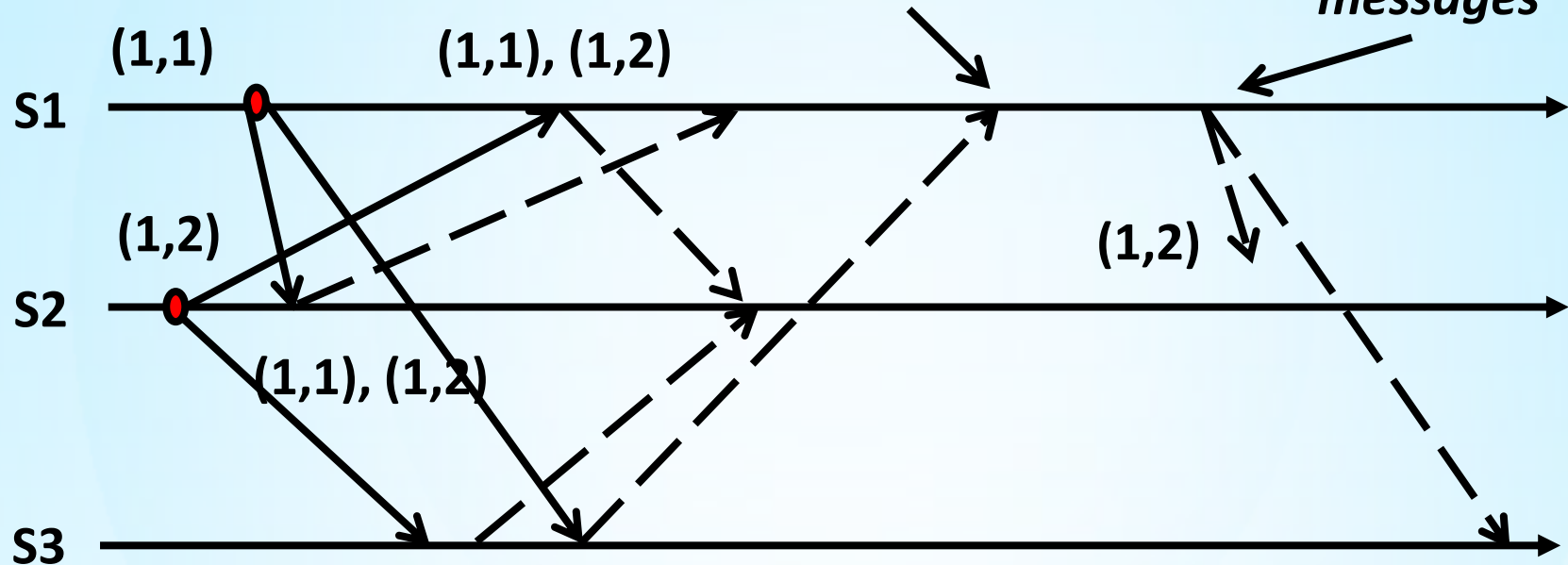
Sites S1 and S2 are Making Requests for the CS



Lamport's Algorithm Example:

Site *S1* enters the CS

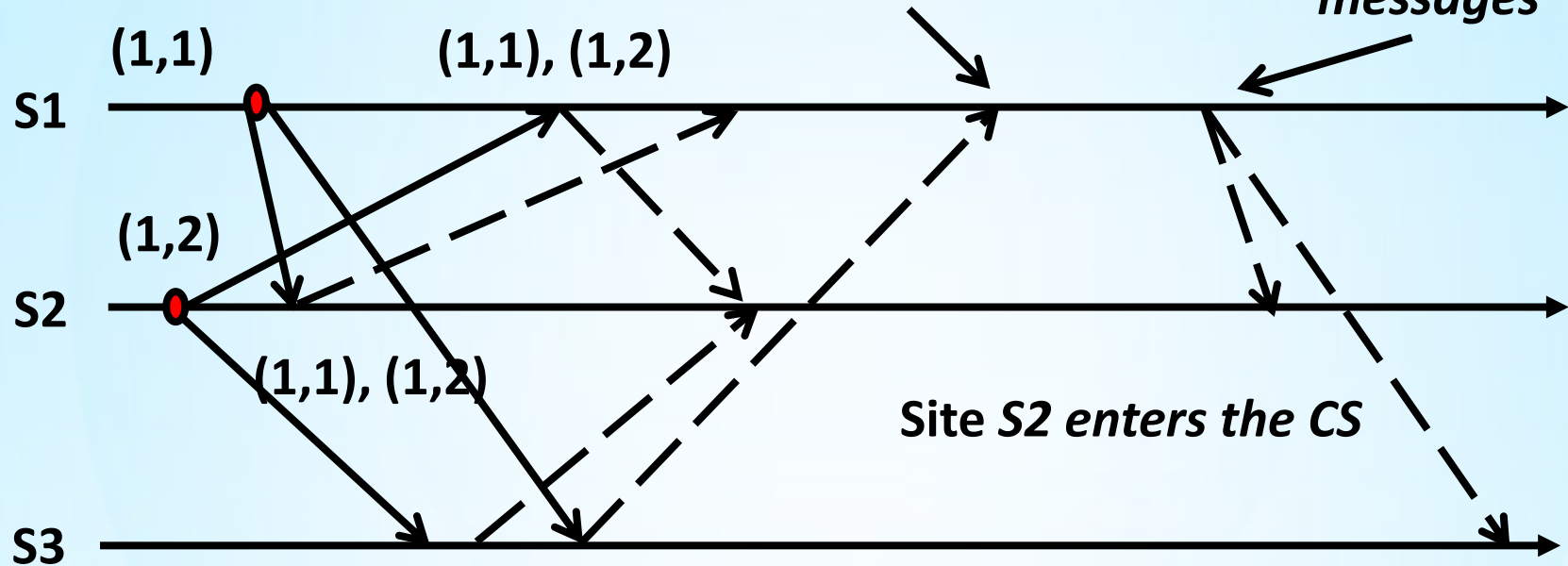
Site *S1* exits the CS
and sends *RELEASE*
messages



Lamport's Algorithm Example:

Site S1 enters the CS

*Site S1 exits the CS
and sends RELEASE
messages*



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_j receives site S_i '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.**

(ii) Ricart-Agrawala Algorithm

- The Ricart-Agrawala algorithm assumes 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 **Lamport-style 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 the **Request-Deferred array**, RD_i , the size of which is the same as the number of processes in the system.
- Initially, $\forall i \ \forall j: RD_i[j]=0$. Whenever p_i defer the request sent by p_j , it sets $RD_i[j]=1$ and after it has sent a REPLY message to p_j , it sets $RD_i[j]=0$.

Description of the 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 reply is deferred and S_j sets $RD_j[i]=1$

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.

Contd...

Releasing the critical section:

(d) When site S_i exits the CS, it sends all the deferred **REPLY** messages: $\forall j$ if $RD_i[j]=1$, then send a **REPLY** message to S_j and set $RD_i[j]=0$.

Notes:

- When a site receives a message, it updates its clock using the timestamp in the message.
- When a site takes up a request for the CS for processing, it updates its local clock and assigns a timestamp to the request.

Correctness

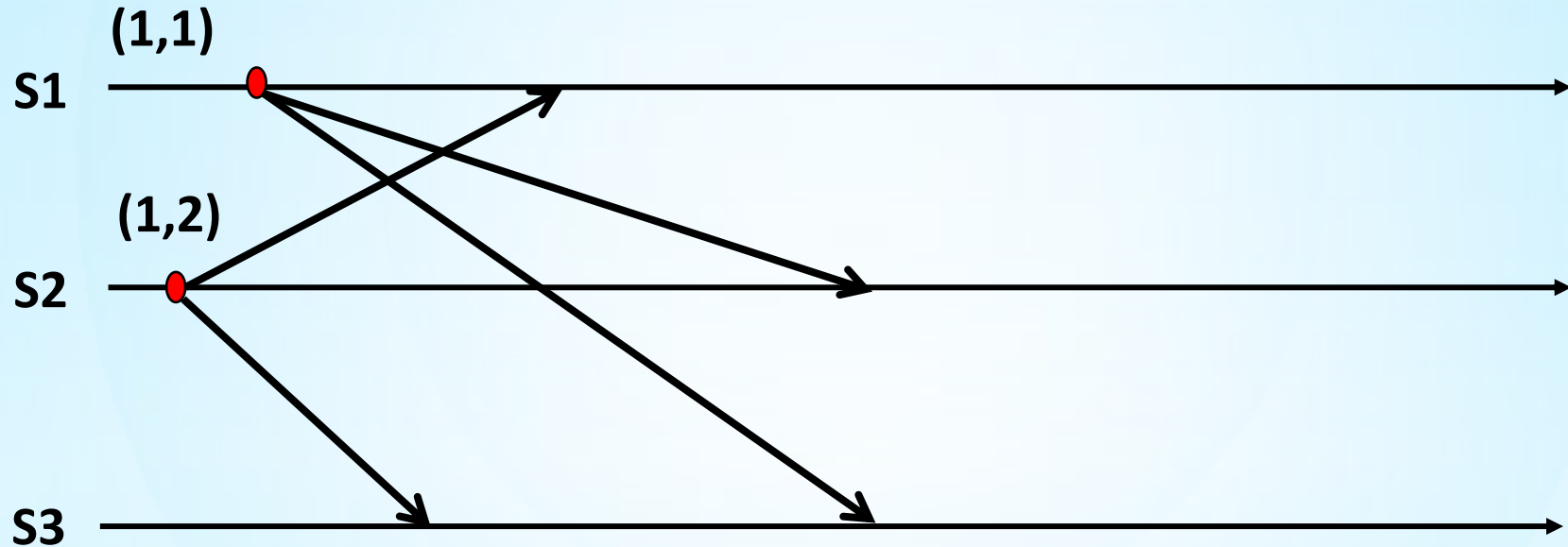
Theorem: Ricart-Agrawala algorithm achieves mutual exclusion.

Proof:

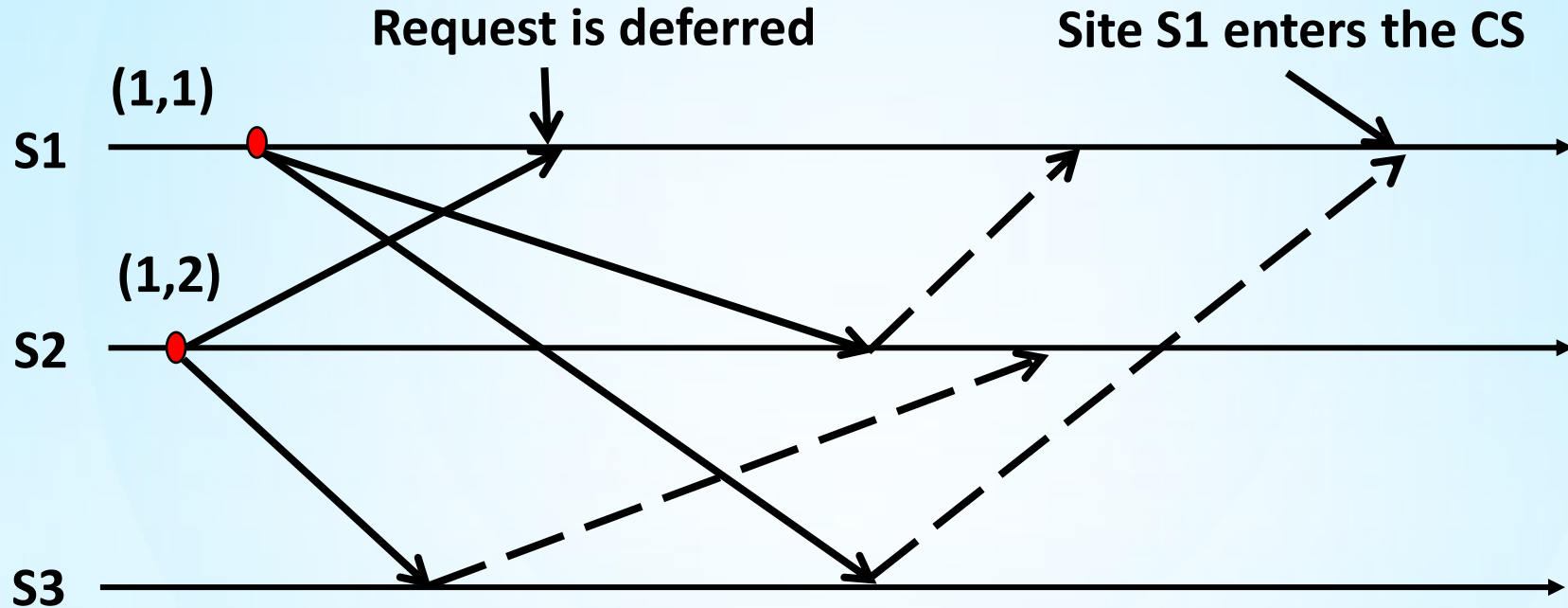
- Proof is by contradiction. Suppose two sites S_i and S_j are executing the CS concurrently and S_i 's request has higher priority than the request of S_j . Clearly, S_i received S_j 's request after it has made its own request.
- Thus, S_j can concurrently execute the CS with S_i only if S_i returns a **REPLY** to S_j (in response to S_j 's request) before S_i exits the CS.
- However, this is impossible because S_j 's request has lower priority.
- Therefore, Ricart-Agrawala algorithm achieves mutual exclusion.

Ricart-Agrawala algorithm Example:

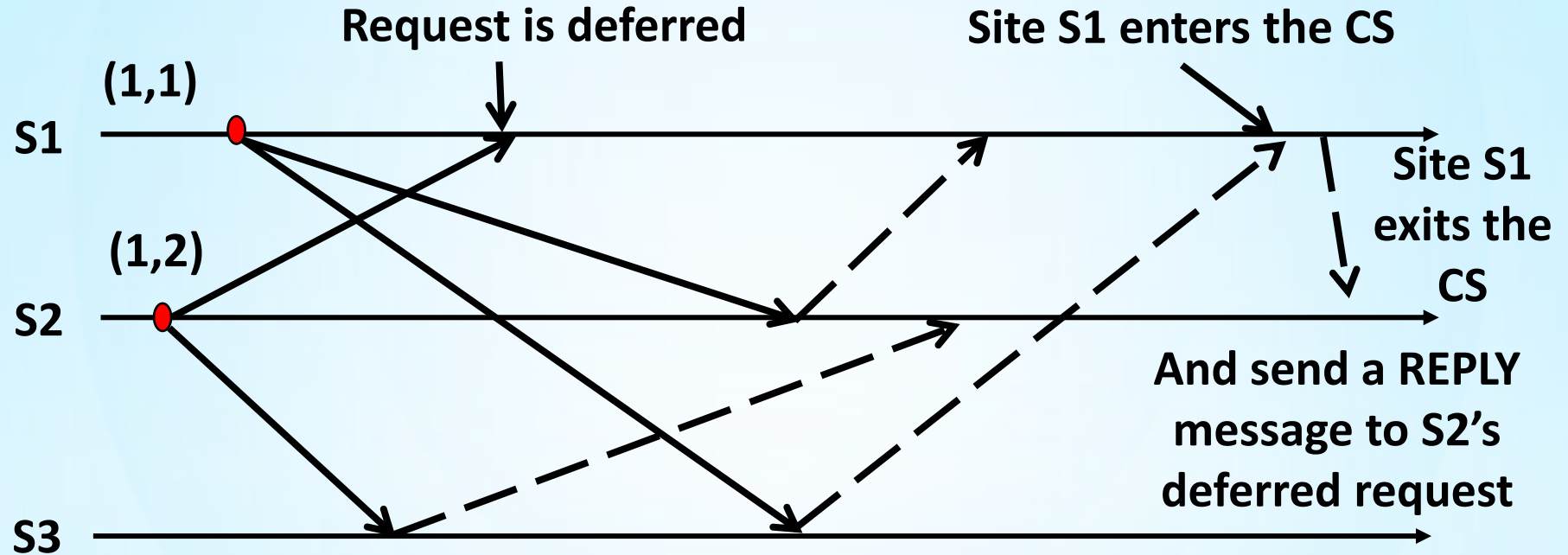
Sites S1 and S2 are Making Requests for the CS



Ricart-Agrawala algorithm Example:



Ricart-Agrawala algorithm Example:



Performance

- For each CS execution, Ricart-Agrawala algorithm requires $(N - 1)$ **REQUEST** messages and $(N - 1)$ **REPLY** messages.
- Thus, it requires $2(N - 1)$ **messages** per CS execution.
- Synchronization delay in the algorithm is T .

Conclusion

- **Mutual exclusion** is a fundamental problem in distributed computing systems, where concurrent access to a shared resource or data is serialized. Mutual exclusion in a distributed system requires that only one process be allowed to execute the critical section at any given time.
- In this lecture, we have discussed about the '**Concepts of Distributed Mutual Exclusion**' and its **Non-token based approaches** i.e. **Lamports algorithm** and **Ricart-Agrawala algorithm**
- In upcoming lectures, we will discuss about **Quorum based approaches** and **Token-based approaches**.