Mutual Exclusion

Distributed Systems

Ali Kamandi, PH.D.

School of Engineering Science

College of Engineering

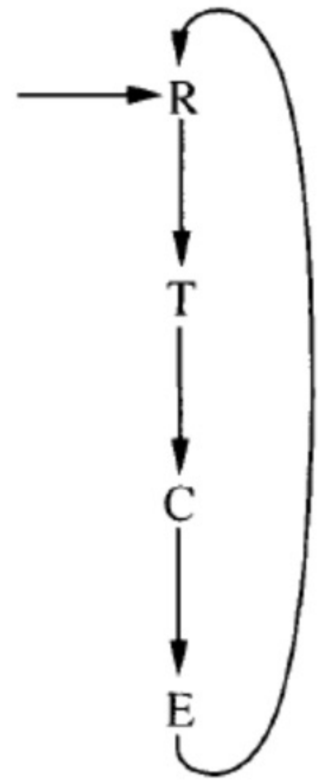
University of Tehran

kamandi@ut.ac.ir

2024

Mutual Exclusion

- **Trying** : trying to get into critical section.
- **Termination**: in critical section.
- **Exiting** : cleaning up so that other processes can enter their critical sections.
- **Remainder** : everything else—essentially just going about its non-critical business.



Goals

Mutual exclusion At most one process is in the critical state at a time.

No deadlock (progress) If there is at least one process in a trying state, then eventually some process enters a critical state; similarly for exiting and remainder states.

No lockout (lockout-freedom): If there is a particular process in a trying or exiting state, that process eventually leaves that state. This means that I don't starve because somebody else keeps jumping past me and seizing the critical resource before I can.

Stronger versions of lockout-freedom include explicit time bounds (how many rounds can go by before I get in) or **bounded bypass** (nobody gets in more than k times before I do).

approach

- Token-based approach.
- Non-token-based approach.
- Quorum-based approach.

Mutual exclusion using strong primitives

A **test-and-set** operation does the following sequence of actions atomically:

```
oldValue ← read(bit)
write(bit, 1)
return oldValue
```

```
while true do
    // trying
    while TAS(lock) = 1 do nothing
    // critical
    (do critical section stuff)
    // exiting
    reset(lock)
    // remainder
    (do remainder stuff)
```

A lockout-free algorithm using an atomic queue

```
while true do
    // trying
    enq(q, myId)
    while peek(q) ≠ myId do nothing
    // critical
    (do critical section stuff)
    // exiting
    deq(q)
    // remainder
    (do remainder stuff)
```

Mutual exclusion using only atomic registers

```
shared data:
waiting, initially arbitrary
present[i] for i ∈ {0, 1}, initially 0
Code for process i:
while true do
    // trying
    present[i] ← 1
    waiting ← i
    while true do
        if present[¬i] = 0 then
            break
        if waiting ≠ i then
            break
    // critical
    (do critical section stuff)
    // exiting
    present[i] = 0
    // remainder
    (do remainder stuff)
```

Peterson's mutual exclusion algorithm for two processes

Requirements of mutual exclusion algorithms

- **Safety property** The safety property states that at any instant, only one process can execute the critical section. This is an essential property of a mutual exclusion algorithm.
- **Liveness property** This property states the absence of deadlock and starvation Two or more sites should not endlessly wait for messages that will never arrive. In addition, a site must not wait indefinitely to execute the CS while other sites are repeatedly executing the CS. That is, every requesting site should get an opportunity to execute the CS in finite time.
- **Fairness** Fairness in the context of mutual exclusion means that each process gets a fair chance to execute the CS. In mutual exclusion algorithms, the fairness property generally means that the CS execution requests are executed in order of their arrival in the system (the time is determined by a logical clock).

The first property is absolutely necessary and the other two properties are considered important in mutual exclusion algorithms.

Performance Metrics

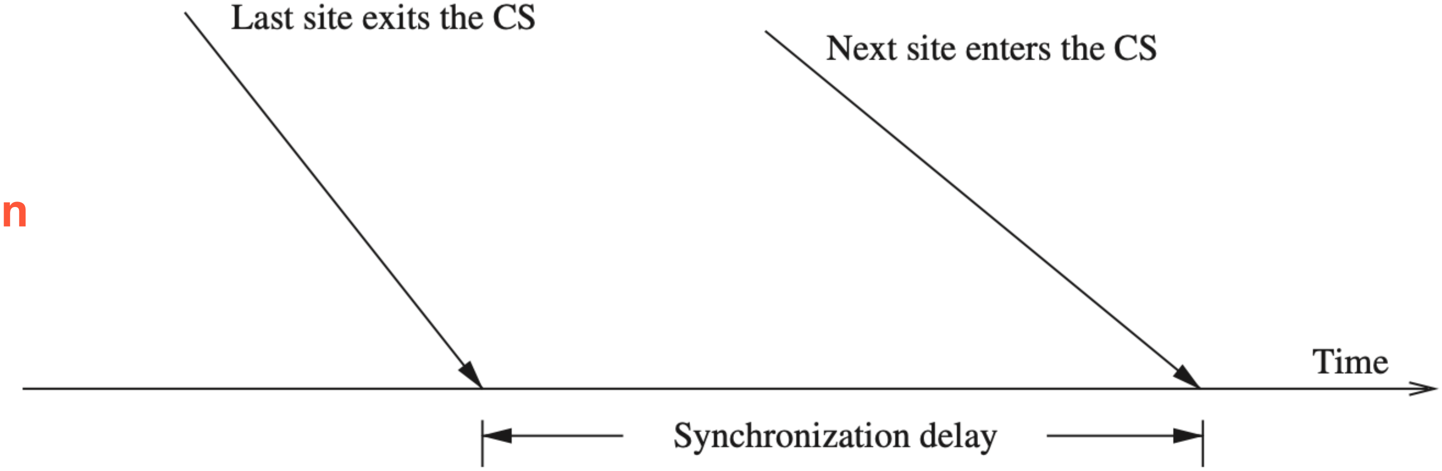
- **Message complexity** This is the number of messages that are required per CS execution by a site.
- **Synchronization delay** After a site leaves the CS, it is the time required and before the next site enters the CS). Note that normally one or more sequential message exchanges may be required after a site exits the CS and before the next site can enter the CS.
- **Response time** This is the time interval a request waits for its CS execution to be over after its request messages have been sent out. Thus, response time does not include the time a request waits at a site before its request messages have been sent out. The first property is absolutely necessary and the other two properties are considered important in mutual exclusion algorithms.

  System throughput This is the rate at which the system executes requests for the CS. If SD is the synchronization delay and E is the average critical section execution time, then the throughput is given by the following equation:

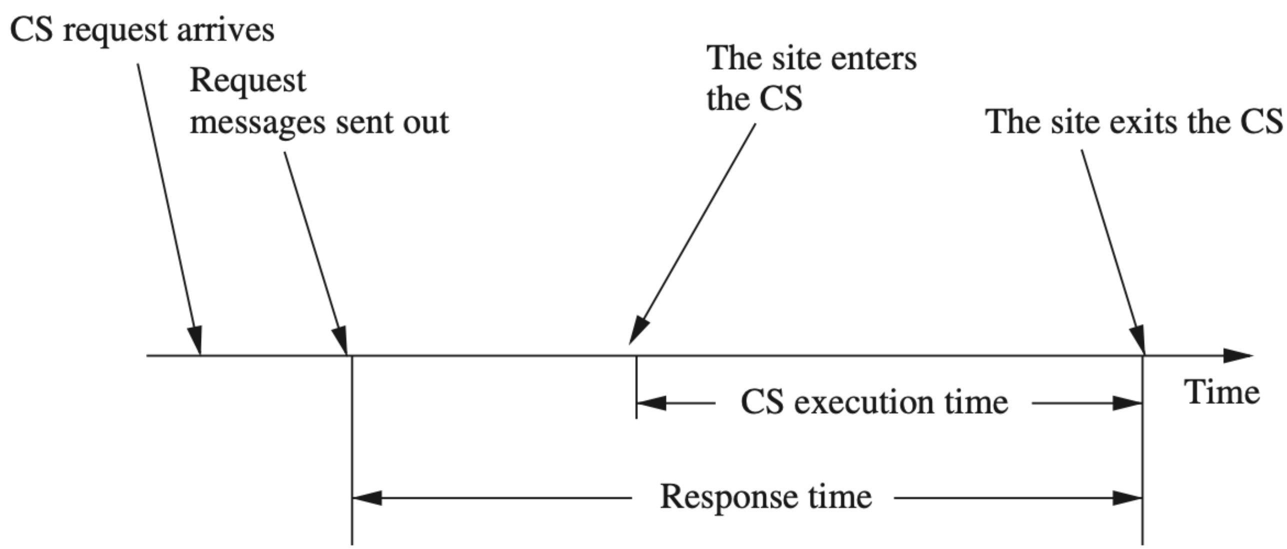
$$\boldsymbol{System\ Throughput = \frac{1}{(SD + E)}}$$

Performance Metrics

Synchronization delay



Response time



Lamport's 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 a site S_j receives the REQUEST(ts_i, i) message from site S_i, it places site S_i's request on *request_queue*$_j$ and 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.

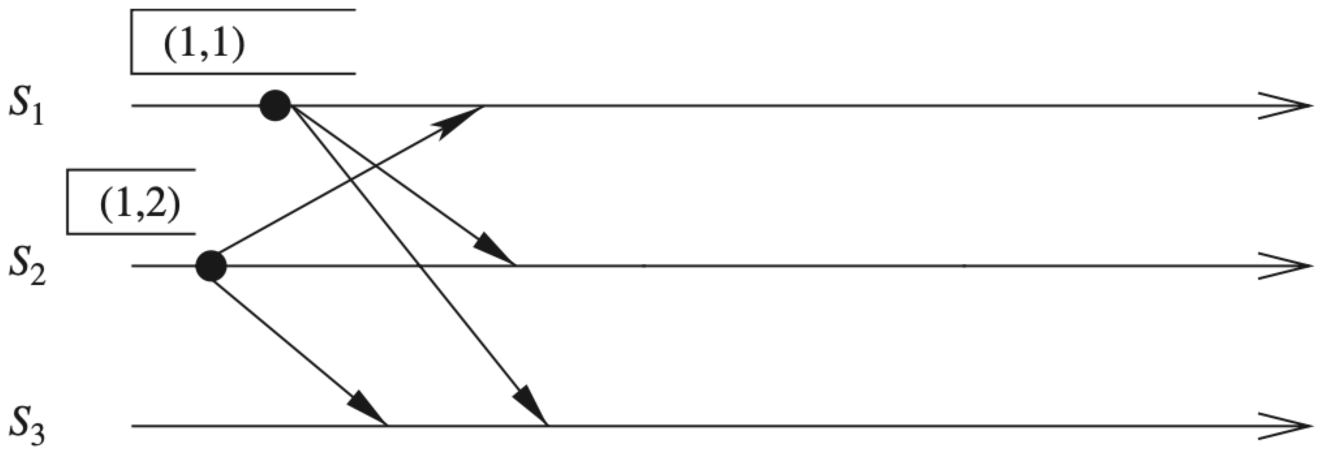
L2: S_i's request is at the top of *request_queue*$_i$.

Releasing the critical section

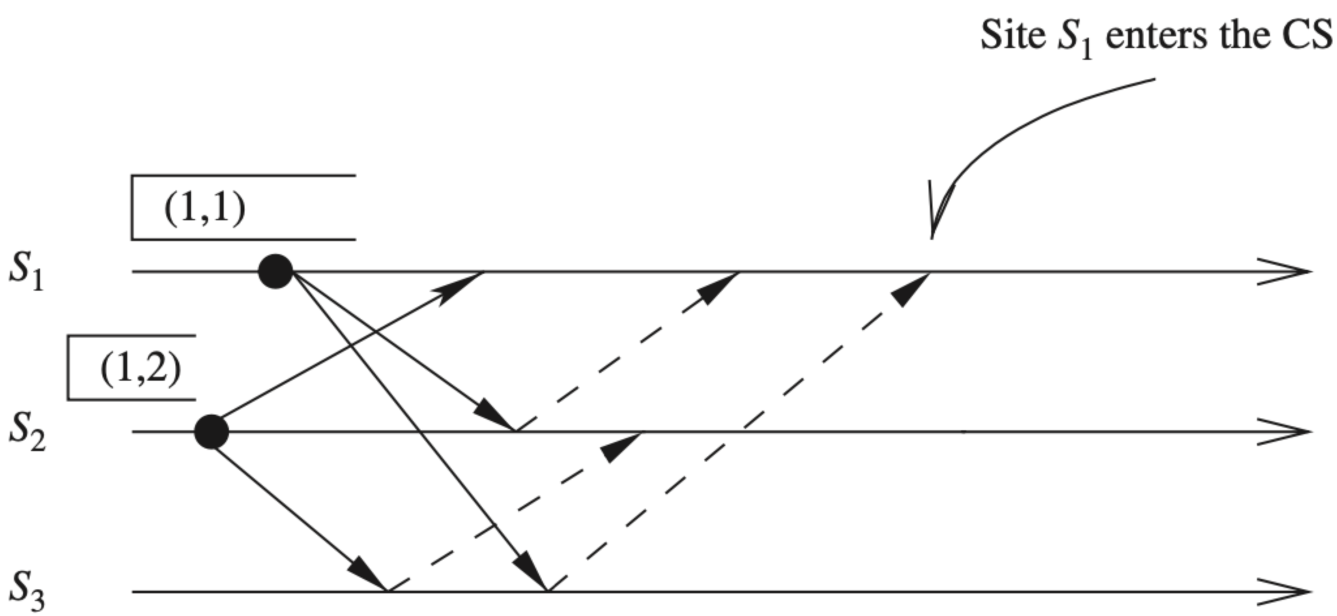
- Site S_i, upon exiting the CS, removes its request from the top of its request queue and broadcasts a timestamped RELEASE message to all other sites.
- When a site S_j receives a RELEASE message from site S_i, it removes S_i's request from its request queue.

Lamport's Algorithm

Site S_1 enters the **CS**.

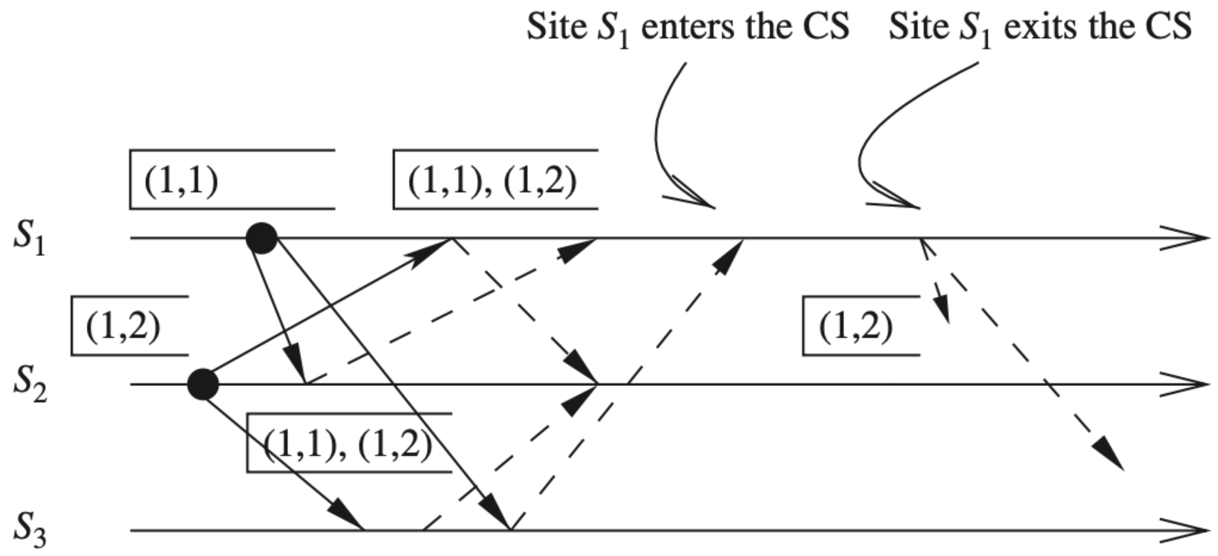


Site S_1 exits the **CS** and sends **RELEASE** messages.

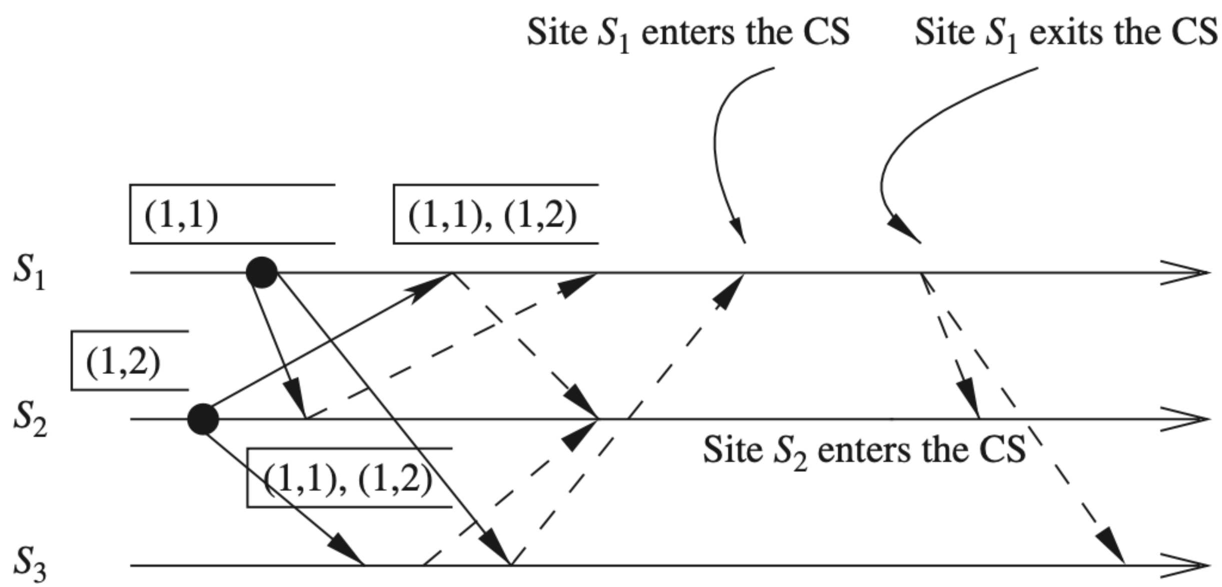


Lamport's Algorithm

Site S_1 exits the **CS** and sends **RELEASE** messages.



Site S_2 enters the **CS**.



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 **3N−1 messages per CS invocation**. The synchronization delay in the algorithm is T.

بهینه سازی: **عدم ارسال پاسخ**

کاهش تعداد پیام بین 2(N-1) و 3(N-1)

Ricart–Agrawala 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 $RD_j[i] := 1$.

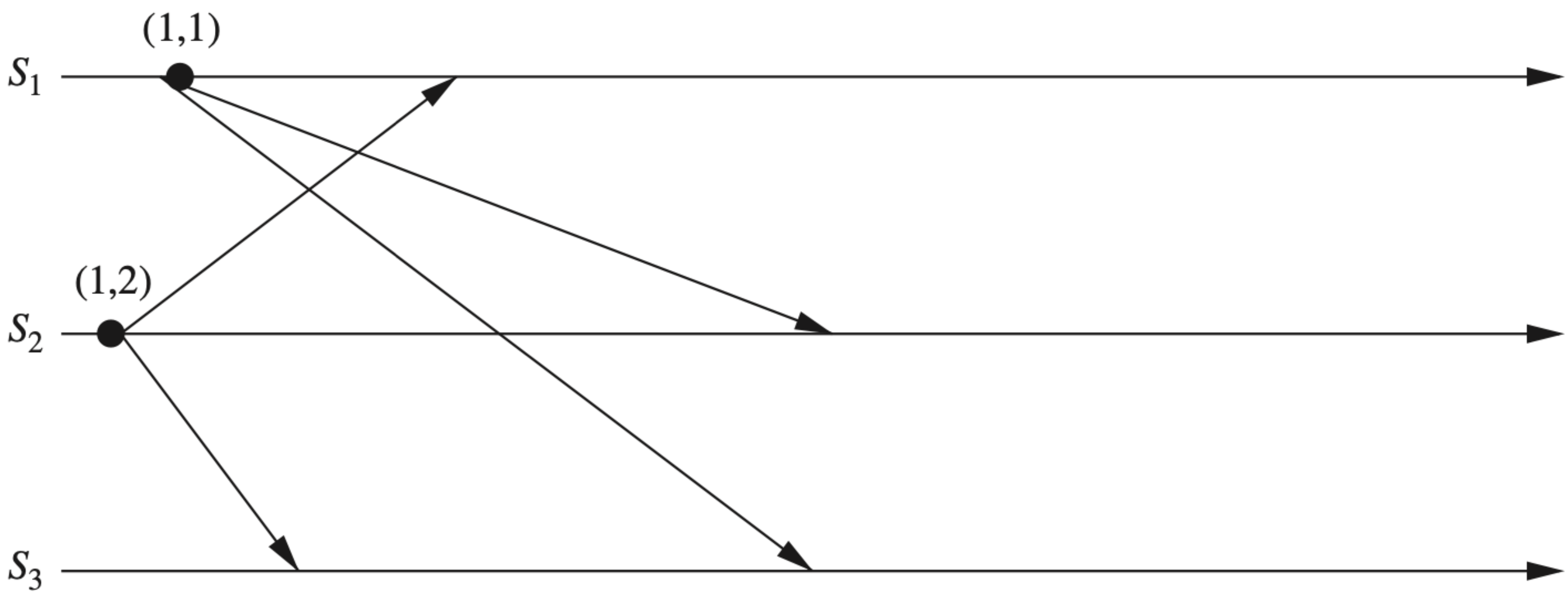
Executing the critical section

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

Releasing the critical section

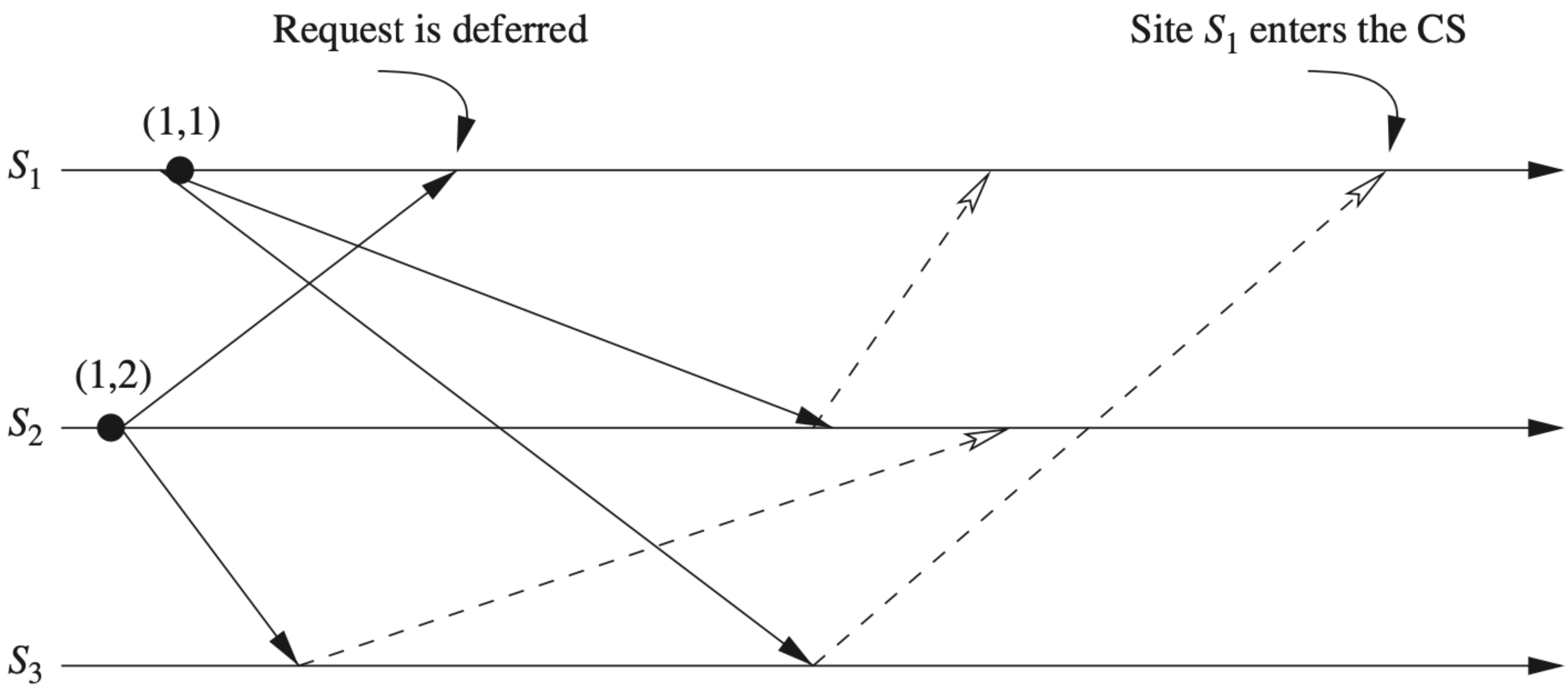
(d) When site S_i exits the CS, it sends all the deferred REPLY messages: $\forall j$ if $RD_i[j] = 1$, then sends a REPLY message to S_j and sets $RD_i[j] := 0$.

Ricart–Agrawala algorithm: example



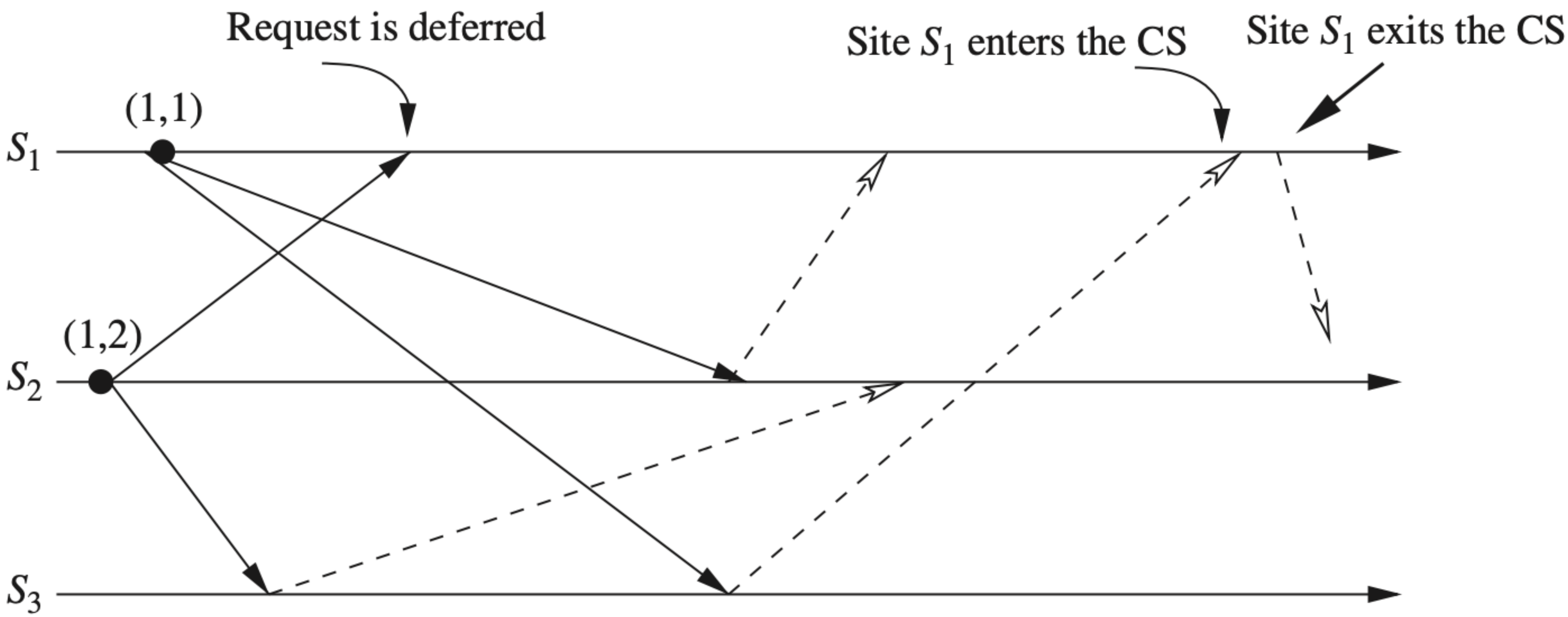
Sites S_1 and S_2 each make a request for the **CS**.

Ricart–Agrawala algorithm: example



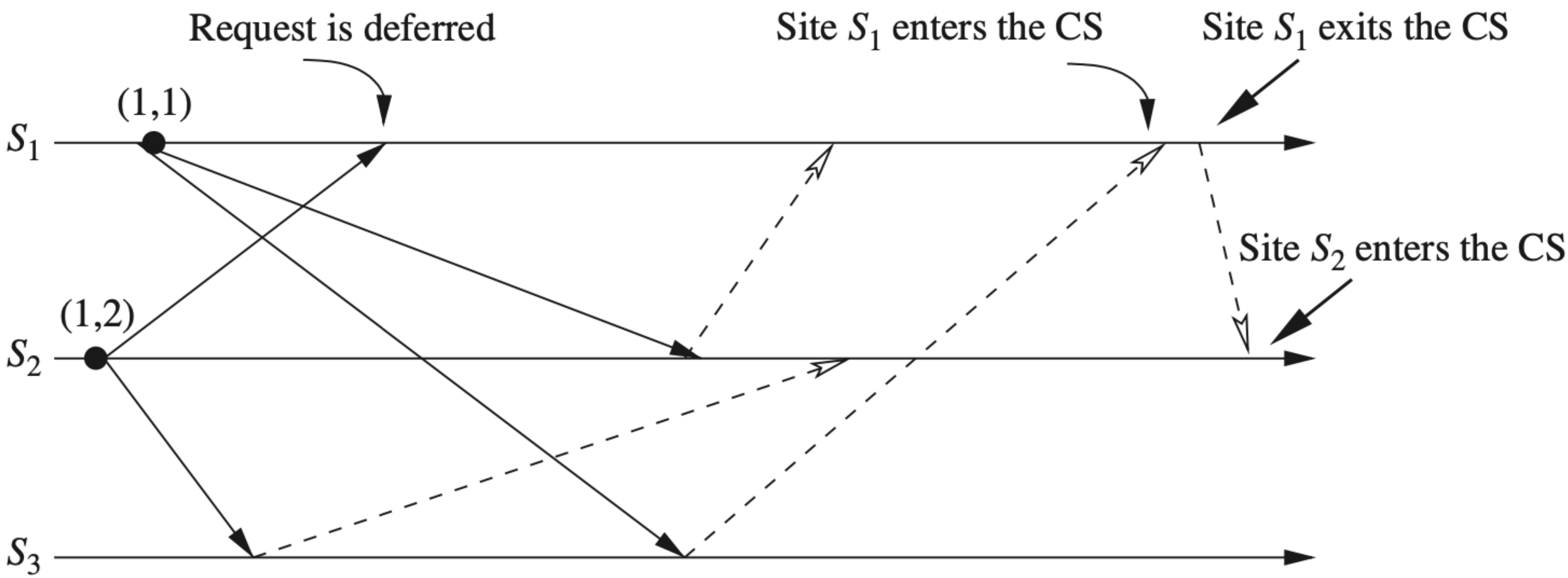
Site S_1 enters the **CS**.

Ricart–Agrawala algorithm: example



Site S_1 exits the **CS** and sends a REPLY message to S_2's deferred request.

Ricart–Agrawala algorithm: example



Site S_2 enters the **CS**.

Performance

For each CS execution, the Ricart–Agrawala algorithm requires **N−1 REQUEST messages** and **N−1 REPLY messages**. Thus, it requires **2N−1 messages per CS execution**. The synchronization delay in the algorithm is T.

Quorum-based mutual exclusion algorithms

Meakawa's algorithm

M1 $(\forall i \; \forall j : i \neq j, 1 \leq i, j \leq N :: R_i \cap R_j \neq \phi)$.

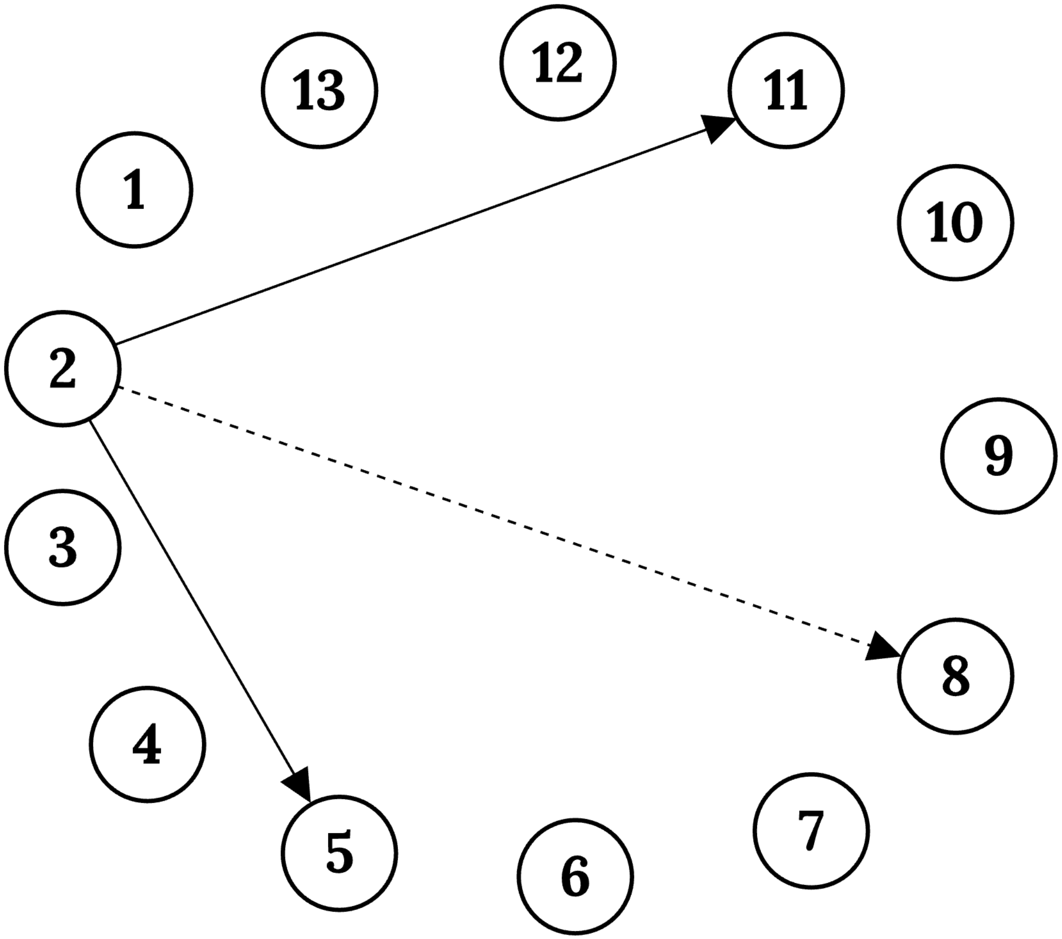
M2 $(\forall i : 1 \leq i \leq N :: S_i \in R_i)$.

M3 $(\forall i : 1 \leq i \leq N :: |R_i| = K)$.

M4 Any site S_j is contained in K number of R_is, $1 \leq i, j \leq N$.

Maekawa used the theory of projective planes and showed that $N=K(K-1)+1$. This relation gives $|R_i| = \sqrt{N}$.

Meakawa's algorithm



$S_1 = \{1,2,3,4\}$
$S_2 = \{2,5,8,11\}$
$S_3 = \{3,5,9,13\}$
$S_4 = \{4,5,10,12\}$
$S_5 = \{5,1,6,7\}$
$S_6 = \{6,2,9,12\}$
$S_7 = \{7,3,8,12\}$
$S_8 = \{8,1,9,10\}$
$S_9 = \{9,4,7,11\}$
$S_{10} = \{10,2,7,13\}$
$S_{11} = \{11,3,6,10\}$
$S_{12} = \{12,1,11,13\}$
$S_{13} = \{13,4,6,8\}$

Meakawa's algorithm

Requesting the critical section:

(a) A site S_i requests access to the CS by sending REQUEST(i) messages 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 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 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 the CS is over, site S_i sends a RELEASE(i) message to every site in R_i.

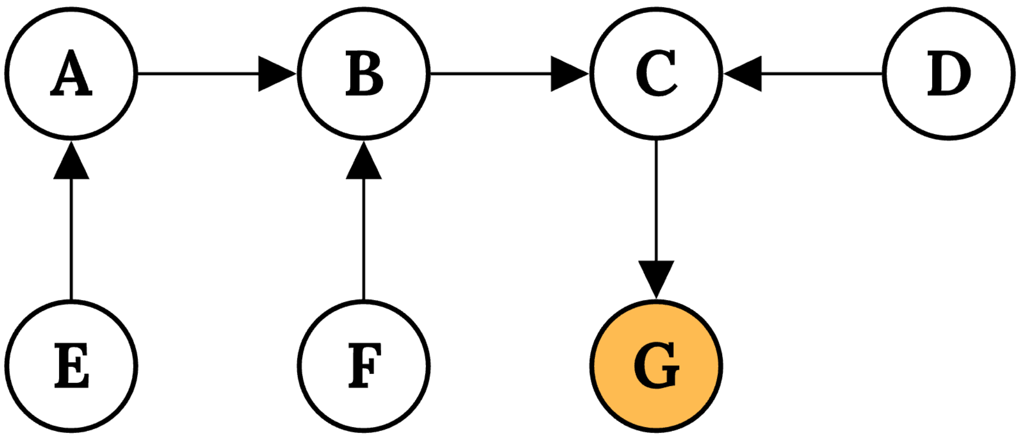
(e) When a site S_j receives a RELEASE(i) message from site S_i, it sends a REPLY message to the next site waiting in the queue and deletes that entry from the queue. If the queue is empty, then the site updates its state to reflect that it has not sent out any REPLY message since the receipt of the last RELEASE message.

Performance

Note that the size of a request set is $\sqrt{N}$. Therefore, an execution of the CS requires $\sqrt{\boldsymbol{N}}$ **REQUEST**, $\sqrt{\boldsymbol{N}}$ **REPLY** and $\sqrt{\boldsymbol{N}}$ **RELEASE messages**, resulting in **$3\sqrt{\boldsymbol{N}}$ messages per CS execution**. Synchronization delay in this algorithm is 2T. This is because after a site S_i exits the CS, it first releases all the sites in R_i and then one of those sites sends a **REPLY** message to the next site that executes the CS. Thus, two sequential message transfers are required between two successive CS executions. Maekawa's algorithm is deadlock-prone. Measures to handle deadlocks require additional messages.

Token-based algorithms

Raymond's tree-based algorithm



$\mathrm{HOLDER_B} = \mathrm{C}$

$\mathrm{HOLDER_C} = \mathrm{G}$

$\mathrm{HOLDER_D} = \mathrm{C}$

$\mathrm{HOLDER_E} = \mathrm{A}$

$\mathrm{HOLDER_F} = \mathrm{B}$

$\mathrm{HOLDER_G} = \mathrm{self}$

References

- Aspnes, Chapter 18
- Singhal, Chapter 9